LibNcFTP: FTP Client Library

What is this?

The *LibNcFTP FTP Client Library* (or simply, "*LibNcFTP*", or "the library") is used to build custom applications that use the File Transfer Protocol (FTP) to exchange files between machines on a TCP/IP network. *LibNcFTP* provides an Application Programming Interface (API) in the "C" language. *LibNcFTP* is most often used as a quick way to add FTP functionality to an existing application, but can be used to build custom middleware, or even dedicated FTP client browser programs.

The rest of the document provides information on how to install and use the library. This document assumes you have a good working knowledge of how the FTP protocol works; we suggest you read "An Overview of the File Transfer Protocol," which explains how FTP works without getting into too many technical details.

For more information about *LibNcFTP* itself, visit the *NcFTP Software* page at http://www.ncftp.com/libncftp/ .

System Requirements

LibNcFTP can be built and used on a variety of UNIX platforms, as well as Microsoft Windows and Mac OS X. The library is provided only in source code form, which means you need a C compiler installed on the machine that *LibNcFTP* itself is to be installed on. Most vendors' proprietary C compilers can be used, as well as the GNU "gcc" compiler. For Microsoft

Windows, Visual Studio 6 or later is required. For Mac OS X, the Developer Tools package must be installed.

The library itself will only need about 10 megabytes of disk space to hold all the source code, object code, and sample program files.

Programs using the library can support any of the platforms that the library can be compiled on, which means your programs can run on the same wide variety of UNIX platforms, Microsoft Windows, or Mac OS X. With a little work, you can write code that can be compiled by all of these platforms.

Programs using the library will use approximately 100 kB of memory for library variables, internal buffers, etc. Likewise, your programs' executable will increase in size by approximately 100 kB, but this can vary wildly depending on what type of optimization and debugging compiler options used, as well as which library functions your program uses.

Downloading and Extracting the *LibNcFTP* Package File

The library source code is provided in two package formats, a gzipped TAR file (.tar.gz) and ZIP (.zip) file. Before proceeding, you should check the official download page at http://www.ncftp.com/download/ to see if updated packages are available.

UNIX users can extract the downloaded package file by typing the following from a shell prompt, where *X.Y.Z* denotes the library version:

```
$ gzip -d -c libncftp-X.Y.Z-src.tar.gz | tar xf -
$ cd libncftp-X.Y.Z
```

The package contains 4 subdirectories: doc, Strn, sio, and libncftp. The doc directory contains documentation and READMEs. The Strn and sio directories contain the source code for the Strn string utility library and sio socket utility library respectively.

Windows users should use their favorite ZIP file utility to extract the libncftp-*X.Y.Z*-src.zip file, and then skip the following section.

Building and Installing *LibNcFTP* for UNIX

From a shell prompt, change to the libncftp subdirectory. There is a script you must run which will checks your system for certain features, so that the library can be compiled on a variety of UNIX systems. If you've built GNU or other open source software before, you're probably familiar with the "./configure ; make ; make install" sequence.

First, decide on which C compiler to use, if you have more than one installed. It's common to have both a vendor's compiler as well as gcc available on the same machine. You can set the "CC" environment variable to the path to your preferred compiler, for example using the Bourne shell:

```
$ CC=/usr/local/gcc/bin/gcc
$ export CC
```

You can also set a CFLAGS environment variable, but you do not need to do this since the configure script will choose the library's recommended set of compiler flags automatically, depending on the C compiler in use and the host operating system.

You are now ready to run the configuration script:

```
$ sh ./configure --prefix=/usr/local
```

This will produce several pages of output, and if all goes well, will create all the necessary Makefiles for you. If you like, you can inspect the Makefiles and config.h files, but you probably won't need to do that. The "--prefix=/usr/local" option you passed to the configure script tells the script to have the library files (.a files, .so files) set to install in /usr/local/lib and the header files (.h files) set to install in /usr/local/include. If you don't want to use /usr/local, you can use a different directory using the --prefix option.

You are now ready to build the libraries and sample programs. Type:

```
$ make
```

This should build everything, including the code in the Strn and sio subdirectories, as well as the sample programs in the libncftp subdirectory. Your build should say "Done" at the end, like this:

```
...
Compiling simpleget: [OK]
-rwx------ 1 gleason gleason   77492 Dec 7 19:41 simpleget
Done.
$
```

If the build did not succeed, send us e-mail and we should be able to help you build it, since we support almost all of the major UNIX variants.

Finally, copy the finished libraries and headers to a permanent location. The easiest way to do that is to do this while logged in as root:

```
$ make install
```

That will copy the libraries into /usr/local/lib, and the header files into /usr/local/include. You can also "make install" while not logged in as root, but you'll need to make sure that /usr/local/lib and /usr/local/include have been created and that the user you are logged in as has write permission to those directories.

You are now ready to build your own programs using *LibNcFTP*. You will need to ensure that your compiler can find the library files and header files. You may need to add "-I/usr/local/include" when compiling and "-L/usr/local/lib" when linking, if your compiler does not search those directories by default. When linking, your programs will need to link with all three of the installed libraries; an easy way to do that is to add the flags "-lncftp -lsio -lStrn" when linking.

Building and Installing *LibNcFTP* for Windows

You will need Visual C++ 6.0 or greater to build the library and sample programs. This version includes two supplementary libraries which you must build and link with your applications: a string utility library (Strn) and a Winsock utility library (sio).

Keep the source hierarchy intact, so that the samples and libraries build without problems. Each project directory contains a visual studio project (.dsp and .dsw files), but you don't need to manually build each project. Instead, open the "LibNcFTP_ALL.dsw" workspace file in the libncftp directory, and then "Rebuild All" to build all the libraries and all the samples.

To build your own applications using *LibNcFTP*, you'll need to make sure you configure your project to find the header files and library files.

Your application may not need to use the sio or Strn libraries directly, but you still need to link with them. For example, the "simpleget" sample

uses "..\..\Debug,..\..\..\Strn\Debug,..\..\..\sio\Debug" in the project option for additional library paths for the linker, and "kernel32.lib user32.lib gdi32.lib advapi32.lib shell32.lib ws2_32.lib Strn.lib sio.lib libncftp.lib" for the list of libraries to link with. Note that *LibNcFTP* uses advapi32.lib and shell32.lib, in addition to the usual "kernel32.lib user32.lib gdi32.lib". Of course, it also needs to link with Winsock (ws2_32.lib).

Similarly, you'll need to make sure one of your additional include directories points to the *LibNcFTP* directory containing ncftp.h. The "simpleget" sample uses "..\.." since it is in a subdirectory of the library itself. If you actually use functions from Strn or sio (as some of the samples do), you'll need to have your project look in their directories for their headers as well.

About Winsock2: This version of the library was designed for use with Winsock version 2. Note that older versions of Windows 95 do not include Winsock version 2, but can be upgraded by getting the updater from Microsoft.

However, the library should also work with Winsock 1.1. That is left as an exercise to the coder to change the Winsock initialization to use 1.1 and to link with the 1.1 library (wsock32.lib).

Tutorial

First, you'll need to include the library header, ncftp.h. This header file includes ncftp_errno.h, which is the library's equivalent to <errno.h>. It also includes many system headers, including <stdio.h>, <time.h>, <sys/types.h>, etc.. Therefore, for our simple example, we only need to:
#include <ncftp.h>

The library avoids using global variables. However, you give each library routine a pointer to a structure with the library's state data. This state data must be maintained between calls, and you must pass a pointer to the same structure for each library function. For simplicity, this example puts the two library structures you pass around in two global variables:

```
FTPLibraryInfo li;
FTPConnectionInfo ci;
```

Somewhere early in your program, before you try any FTP, you must initialize the library. Do that like this:

```
FTPInitLibrary(&li);
```

Then, when you want to open a host, you initialize a session structure. Do that by calling FTPInitConnectionInfo() to initialize to the default values, and then you set the fields that define the session:

```
FTPInitConnectionInfo(&li, &ci, kDefaultFTPBufSize);
```

For this example, we will try a non-anonymous login to a server named ftp.cs.unl.edu with a username of mgleason and a password of au29X-b7.

```
ci.debugLog = myDebugFile;
strcpy(ci.user, "mgleason");
strcpy(ci.pass, "au29X-b7");
strcpy(ci.host, "ftp.cs.unl.edu");
```

I recommend that you use the optional debugLog parameter while you are testing. That is a FILE * pointer, and you are responsible for opening and closing it. You may want to use just stdout.

Then open the host:

```
if ((result = FTPOpenHost(&ci)) < 0) {
        fprintf(stderr, "Cannot open host: %s.\n",
FTPStrError(result));
        exit(1);
}
```

Now try some downloads. Note that our example no longer checks the return code (result) from the library functions, but you should check for errors by checking if the return code is negative.

```
FTPChdir(&ci, "/pub/foobar/downloads/April2001");
 /* remote directory */
globPattern = "log.????2000";
dstDir = "/usr/log/junk";                    /* local
directory */
result = FTPGetFiles3(&ci, globPattern, dstDir, kRecursiveNo,
kGlobYes,
                kTypeBinary, kResumeNo, kAppendNo, kDeleteNo,
                kTarNo, kNoFTPConfirmResumeDownloadProc, 0
                );
```

Make a few directories, setting the recursive parameter to kRecursiveYes to simulate "mkdir -p":

```
newDir = "/pub/foobar/uploads/May2001";      /* remote
directory */
result = FTPMkdir(&ci, newDir, kRecursiveYes);
```

Upload some files into the directory we just created:

```
result = FTPPutFiles3(&ci, "/usr/logs/*.05??2001", newDir,
```

```
                                        kRecursiveNo, kGlobYes,
kTypeBinary, kAppendNo,
                                        NULL, NULL, kResumeNo,
kDeleteNo,
                                        kNoFTPConfirmResumeUploadProc,
0
                                        );
```

Finally, close the connection. If your FTPOpenHost succeeded, you should make a good effort to make sure you close it:

```
FTPCloseHost(&ci);
```

To see a simple example in entirety, see the simpleget.c source file included with the sample code. You should be able to compile that program and run it, to see how things work. This sample is located in the samples/simpleget subdirectory.

Library Data Structures

FTPLibraryInfo

Each program that uses the library should have one of these structures. This needs to be referenced by all parts of your code that call a FTP library routine, so declare this a global variable or local variable in scope of all subroutines that use FTP.

```
typedef struct FTPLibraryInfo {
                char magic[16];
                ...
                char defaultAnonPassword[80];
} FTPLibraryInfo, *FTPLIPtr;
```

The magic field is used internally by the library to make sure the programmer (that would be you :-) isn't using an invalid structure with the library. The library routines check this field against a well-known value to ensure validity.

The defaultAnonPassword field can be changed after the library has been initialized. You may need to set this manually if the library does not calculate a valid password for use with anonymous logins.

There are other fields in the structure, but they are used internally and should not be modified or relied upon.

FTPConnectionInfo

Each program that uses the library may have one or more of these structures in use at a time. Like the FTPLibraryInfo structure, you should declare these as global variables or local variables that maintain scope throughout use of your FTP operations, because you need to pass a pointer to this structure to each library function.

This structure is a mixture of mostly internal-use variables and configurable options. The structure is documented below, but some fields which are used internally have been omitted. The structure is large, but in practice you'll only use a handful of the fields and let the library use appropriate default values for the others.

```
typedef struct FTPConnectionInfo {
        char magic[16];

        char host[64];
        char user[64];
        char pass[64];
        char acct[64];
        unsigned int port;

        int errNo;
        char lastFTPCmdResultStr[128];
        FTPLineList lastFTPCmdResultLL;
        int lastFTPCmdResultNum;

        FILE *debugLog;
        FTPLogProc debugLogProc;

        unsigned int xferTimeout;
        unsigned int connTimeout;
        unsigned int ctrlTimeout;
        unsigned int abortTimeout;

        int maxDials;
        int redialDelay;

        int dataPortMode;

        int firewallType;
        char firewallHost[64];
        char firewallUser[64];
        char firewallPass[64];
        unsigned int firewallPort;
```

```
size_t ctrlSocketRBufSize;
size_t ctrlSocketSBufSize;
size_t dataSocketRBufSize;
size_t dataSocketSBufSize;

const char *asciiFilenameExtensions;

unsigned short ephemLo;
unsigned short ephemHi;

FTPConnectMessageProc onConnectMsgProc;
FTPRedialStatusProc redialStatusProc;
FTPPrintResponseProc printResponseProc;
FTPLoginMessageProc onLoginMsgProc;

FTPGetPassphraseProc passphraseProc;

FTPProgressMeterProc progress;
longest_int bytesTransferred;
int useProgressMeter;
struct timeval t0;
double sec;
double secLeft;
double kBytesPerSec;
double percentCompleted;
longest_int expectedSize;
time_t mdtm;
time_t nextProgressUpdate;
const char *rname;
const char *lname;
int stalled;
int dataTimedOut;
int cancelXfer;

char actualHost[64];
char ip[32];

char *startingWorkingDirectory;
...

int iUser;
void *pUser;
longest_int llUser;
```

```
        ...
} FTPConnectionInfo;
```

The magic field is used internally by the library to make sure the programmer isn't using an invalid structure with the library. The library routines check this field against a well-known value to ensure validity. Do not modify this field!

User authentication fields

Before connecting, you will always set the host field, which is the hostname or IP address of the remote FTP server to connect to. If the server is running on a non-standard port number (not 21), you may set the port field.

If you are logging in non-anonymously, you will need to use username and password authentication, which requires that you need to set the user and pass fields (and very rarely, the acct field). Otherwise, you can leave the user field unset to indicate you wish to login anonymously.

Error detection fields

The most interesting field is the errNo field. If you an error occurs within a library function, a negative result code is returned and the errNo field is set to the error number, which you can find listed in <ncftp_errno.h>.

If you get an error, you may also want to inspect the lastFTPCmdResultStr field. This will contain the first line of the most recent reply from the FTP server (i.e. "550 Permission Denied"). Similarly, the lastFTPCmdResultNum contains the numeric FTP protocol result code that came along with it (i.e. 550), and the lastFTPCmdResultLL is the FTPLineList containing the complete reply.

Logging fields

For debugging purposes you may want to use the debugLog field which you can set to stdio FILE * pointers. The debugLog writes the whole conversation that took place on the control connection, and would be what you would get had you done an FTP manually.

Instead of having the library write to files directly, you may wish to use your own logging function. You can use the debugLogProc field to point to a custom logging function. The function should be of this type:

```
typedef void (*FTPLogProc)(const FTPCIPtr, char *);
```

The first argument is a pointer to the FTPConnectionInfo structure, and the second is the string produced for logging.

Redialing fields

The library can "redial" automatically, so if the connection or login attempt failed, it can retry. To do this, you set the maxDials field to 2 or more. The related field redialDelay can be set to the number of seconds to wait between each attempt.

Timeout fields

There are several timeout fields you can set if you want certain FTP operations to abort after a period of time. These fields are xferTimeout, connTimeout, ctrlTimeout, and abortTimeout, and each value is in seconds. If you set a timeout field, library functions may return an error code when the timeout occurs.

The xferTimeout refers to the amount of time to wait on an data transfer read or write; The connTimeout refers to the amount of time to wait before establishing a successful control (FTP conversation) connection; The ctrlTimeout refers to the amount of time to wait for a response from the server on an established control connection; And the abortTimeout refers to a special case of the above when an abort transfer has been requested. (Usually that is used when the user hits ^C and is intending to quit the program as soon as possible, so the time is much less than a regular control connection response.)

You would use the connTimeout field when you are first trying to connect to a remote FTP server and want to place a reasonable time limit for the connection and login procedure.

Once you have established a session, you could use the ctrlTimeout to prevent a particular command from blocking forever, since the server may become unresponsive (or time you out!) at any time after the session has been established.

Once a file transfer connection has been established, the xferTimeout is used to prevent uploads or downloads from stalling forever. Because of network congestion, it is not uncommon for FTP data transfers to timeout despite having previously transferred hundreds of kilobytes of data. Note that the connTimeout is used when first trying to *establish* the data connection, but once the data transfer has started the xferTimeout is used. The reason for this is that it is usually best to have a relatively short timeout to see if you can connect to the remote

host, and then to use a longer timeout for the actual data blocks so that the sending and receiving hosts have extra leeway to survive transient network congestion.

The abortTimeout is used in one special case. When a client wants to abort a transfer, the client is supposed to send an abort request (ABOR). This timeout is used to place a timeout on the response to the abort request; typically this is very short (3 seconds) as a courtesy to the server before assuming the server can't abort the transfer (which happens frequently). The alternative is to just close the data connection, but the protocol specification implies that is not good behavior. The abortTimeout is available so that the ctrlTimeout would not need to be used for this, since the ctrlTimeout is typically much longer. You don't want to wait around for an abort request to finish when you will most likely be closing the session anyway.

Data connection type fields ("PASV" or "PORT")

The dataPortMode may be set to one of kSendPortMode, kPassiveMode, or kFallBackToSendPortMode. If you want to try passive FTP, you can use kPassiveMode. You can also use kFallBackToSendPortMode which means the library will try passive FTP first, and if the server does not support it, it will then try regular port FTP.

FTP proxy fields

The library provides a means to work with FTP proxies. These proxies are traditionally older firewall implementations which do not implement automatic and transparent FTP with network address translation. Hopefully it will not be necessary to use this feature, but if your network administrator says you have to "login to the firewall" in order to use FTP, you will need to use these fields and get the necessary details on how your firewall is accessed.

Note that the library does **not** support HTTP proxies which handle FTP! You would need to connect to the proxy server using the HTTP protocol and issue a "GET" request with a FTP URL, and *LibNcFTP* does not support HTTP. Examples of proxy servers which are **not** supported: Microsoft Proxy Server, Netscape Proxy Server, and Squid.

The firewallType field can be set to a value from 0 to 7. Each of the permutations that the library supports are listed below.

- **Type 1**: Connect to firewall host, but send "USER *user@real.host.name*". The "*real.host.name*" will be filled in by the library.
- **Type 2**: Connect to firewall, login with "USER *fwuser*" and "PASS *fwpass*", and then "USER *user@real.host.name*".
- **Type 3**: Connect to and login to firewall, and then use "SITE *real.host.name*", followed by the regular USER and PASS.
- **Type 4**: Connect to and login to firewall, and then use "OPEN *real.host.name*", followed by the regular USER and PASS.
- **Type 5**: Connect to firewall host, but send "USER *user@fwuser@real.host.name*" and "PASS *pass@fwpass*" to login.
- **Type 6**: Connect to firewall host, but send "USER *fwuser@real.host.name*" and "PASS *fwpass*" followed by a regular "USER *user*" and "PASS *pass*" to complete the login.
- **Type 7**: Connect to firewall host, but send "USER *user@real.host.name fwuser*" and "PASS *pass*" followed by "ACCT *fwpass*" to complete the login.
- **Type 0**: Do NOT use a firewall (the default).

The firewallHost field must be set to the hostname or IP address string of the firewall machine. You may need to use an IP address string (i.e. "192.168.200.250" if Domain Name Service is not available.

The firewallPort field may be set to a TCP port number, if the firewall requires you to connect to the firewall on a port other than the default port (21).

The firewallUser and firewallPass fields is available if your firewall requires you to login with a username and password (*fwuser* and *fwpass* as noted above).

Socket buffer size fields

The socket buffers used for the control connection and data connections can be tuned if you want to use something other than the default values (which vary by platform and individual kernel tuning). Each of the options below can be set to the number of bytes you wish to use for the buffer. The library will then attempt to set the socket option SO_SNDBUF or SO_RCVBUF as needed.

- ctrlSocketRBufSize - control connection, receive buffer
- ctrlSocketSBufSize - control connection, send buffer
- dataSocketRBufSize - data connection, receive buffer
- dataSocketSBufSize - data connection, send buffer

If the data connection socket buffers are set by you, the library will also attempt to negotiate *TCP Large Window* support with the server by using SITE commands (SITE RETRBUFSIZE, RBUFSIZ, RBUFSZ, BUFSIZE, STORBUFSIZE, SBUFSIZ, SBUFSZ). This could also significantly increase performance depending on the type of links (i.e. satellite) between client and server. The downside to this is that not many FTP servers support the needed SITE commands to enable this feature.

Note that it is usually not a good idea to set the buffers for the control connection. The default sizes are already more than large enough, so setting the sizes larger will not increase performance.

Automatic type selection fields

Functions such as FTPGetFiles and FTPPutFiles transfer groups of files at a time. The asciiFilenameExtensions field gives you an opportunity to automatically have files in these groups which end in certain extensions be transferred in ASCII rather than binary. This field requires a special format, which is best illustrated by an example:

```
cip->asciiFilenameExtensions = "|.txt|.asc|.html|.htm|";
```

The extensions are delimited by pipe characters ('|') and the string must both begin and end in a pipe character.

Ephemeral port selection fields

If you set both the ephemLo and ephemHi fields, this range of ports will be used when selecting port numbers to use for active data connections (i.e. what we use with PORT). For example, you could set ephemLo to 50000 and ephemHi to 60000 to have PORT only use ports between 50000 and 60000.

Message callback fields

The library provides a few hooks so that if your program wants to update its user interface with FTP status information it can do so. The following callbacks are defined:

```
typedef void (*FTPConnectMessageProc)(const FTPCIPtr, ResponsePtr);
typedef void (*FTPLoginMessageProc)(const FTPCIPtr, ResponsePtr);
typedef void (*FTPRedialStatusProc)(const FTPCIPtr, int mode, int
value);
typedef void (*FTPPrintResponseProc)(const FTPCIPtr, ResponsePtr);
```

Some of the callbacks pass you a pointer to a Response structure, which looks like this:

```
typedef struct Response {
        FTPLineList msg;
        int codeType;
        int code;
        int printMode;
        int eofOkay;
        int hadEof;
} Response, *ResponsePtr;
```

The only thing you should need to access in the structure is the msg field, which contains a FTPLineList with the text from the FTP server.

The onConnectMsgProc and onLoginMsgProc callbacks provide a way for you to display the connect and login messages, respectively. When you FTP to a remote server, the first thing the server does is send the connect message (also known as a welcome message). The server then expects you to login with a username and password. Once you have logged in, it sends a login message in reply. Neither the connect nor the login message are guaranteed to contain anything except a terse FTP reply message, but popular servers often have useful information in one or both.

The redialStatusProc is called to indicate what the library is doing. The mode parameter will be either kRedialStatusDialing or kRedialStatusSleeping. If it is kRedialStatusDialing, then the value parameter will contain how many connection attempts to the server (number of "dials") and after your function is called the library will immediately attempt to connect. If it is kRedialStatusSleeping, value will be how many seconds the library is delaying until the next attempt.

The printResponseProc is called whenever the library would be printing a Response structure to the debug log. It could be used to modify the printMode field of the Response structure to determine if the Response should be printed, but generally you should be filtering that information from a debugLogProc instead.

Password request callback fields

When attempting a user (not anonymous) login, you may leave the pass field blank (set to an empty C-string, "") and set passphraseProc to a:

```
typedef void (*FTPGetPassphraseProc)(const FTPCIPtr, FTPLineListPtr
pwPrompt, char *pass, size_t dsize);
```

When the library needs the password, it will call your function to allow you to present a prompt to an end-user. As parameters your function will receive the message from the remote FTP server (useful for one-time password systems which provide information to the user) in the pwPrompt parameter, a pointer to a C string in pass, and the size of the string in dsize (i.e., do not write more than dsize - 1 bytes to pass).

Progress meter fields

The library computes statistics for each data transfer. If you are interested in those, you can inspect the bytesTransferred, sec, and kBytesPerSec fields for the results. Those correspond to the size of the file, how long it took in seconds to transfer, and how fast it was in kilobytes per second.

You may also want to implement a progress meter, which updates while the transfer is in progress. To do that you set the progress field to a:

```
typedef void (*FTPProgressMeterProc)(const FTPCIPtr, int);
```

The second argument tells you which state the transfer is in. You will get a kPrInitMsg message before the first block of data transferred, and an kPrEndMsg at the end. In addition, you get a kPrUpdateMsg for each block transferred. The sample source code included with the package shows how to use progress meters.

During the transfer (and when your progress meter function is called) the following additional fields are valid.

- t0 - the time when the transfer started;
- expectedSize - How many bytes the destination file should contain after the transfer completes;
- secLeft - An estimate of how many seconds are remaining based on the current throughput (if expectedSize is known)
- percentCompleted - Percentage of the transfer that has been completed (0 to 100%) (if expectedSize is known)
- mdtm - The modification timestamp of the source file;
- nextProgressUpdate - the next timestamp we will update the progress meter;
- rname - name of the remote file;
- lname - name of the local file;
- stalled - if non-zero, the transfer is currently waiting for the remote host for data;
- dataTimedOut - if non-zero, the transfer was aborted because the xferTimeout expired;

There is one more important field to mention, the cancelXfer field. This can be set during a transfer (from within your progress meter function) to non-zero to indicate that the current transfer should be aborted. Note that aborting a transfer will often cause the remote server to not only stop sending over the data connection, but it may also disconnect your FTP session altogether.

Besides providing a way for you to abort a transfer in progress, progress meter functions also provide a way for you to have your code do something else while the transfer proceeds. This can be something related to the transfer, such as drawing a bar graph on screen, or something totally unrelated.

DNS information fields

When the library connects to the FTP server you designate with the host field, the library does a DNS lookup and stores the actual DNS hostname for the host in actualHost, and an IP address string in ip.

Starting directory fields

One of the first things the library does after successfully logging in to a remote server is determine the current working directory. The startingWorkingDirectory is set as a result. It may be NULL if the login failed. This is often the root directory if you are performing an anonymous login. For user logins, it is often the home directory of the user.

Piggyback fields

Near the end of the structure the library provides a few fields for private use by the programmer. A common use for these fields is to set one to contain a pointer to another structure which you want to associate with the library structure without having to use a global variable.

The iUser is an integer, the pUser is a void pointer, and the llUser is a longest_int field.

Working with FTPLineLists

Some of the library routines work with a FTPLineList structure. This is simply a linked-list of dynamically allocated C-strings.

```
typedef struct FTPLine *FTPLinePtr;
typedef struct FTPLine {
```

```
                FTPLinePtr prev, next;
                char *line;
} FTPLine;

typedef struct FTPLineList {
                FTPLinePtr first, last;
                int nLines;
} FTPLineList, *FTPLineListPtr;
```

Here is an example use that shows how to print the contents of a remote wildcard match:

```
FTPLineList fileList;
FTPLinePtr lp;
int i, err;

err = FTPRemoteGlob(cip, &fileList, "*.c", &fileList, kGlobYes);
if (err == kNoErr) {
        for (lp = fileList.first, i=0; lp != NULL; lp = lp->next) {
                ++i;
                printf("item #%d: %s\n", i, lp->line);
        }
}

DisposeLineListContents(&list);
```

Working with longest_ints

Some of the library routines work with longest_int variables. The longest_int is actually a #define to the largest integral type your compiler supports -- you can see this by browsing the <ncftp.h> header file. When you built *LibNcFTP* from the source code, the configure script ran a test and modified your copy of the ncftp.h header file to reflect the correct definition of longest_int.

Hopefully your longest_int will be at least a 64-bit value, such as "long long", otherwise it will fallback to being a 32-bit value, "long". If longest_int is only 32-bit, then *Large File Support* will be unavailable on your system which will effect your capability to work with files larger than 2 gigabytes.

Use longest_int and longest_uint variables just as you would with int or unsigned int variables. Be cognizant when casting to and from

longest_int variables, since this could result in truncation similar to the effect of miscasting shorts and longs. In effect, you can do:

```
int x = 1234567;
longest_int y;

y = (longest_int) x;
```

But you cannot do:

```
int x;
longest_int y = 990000000003LL;

x = (int) y;
```

You also must take care when passing longest_ints to the C library functions in the printf() and scanf() family. You will need to know your operating system's method of printing and scanning 64-bit values. *Usually* for UNIX systems you use "%lld", and on Microsoft Windows you use "%I64d". If the manual pages for printf() and scanf() aren't helpful, you can look at the libncftp/config.h file in your extracted and configured *LibNcFTP* source code -- look for PRINTF_LONG_LONG and SCANF_LONG_LONG.

FtwInfo

The "file tree walk" (Ftw) family of functions work with a FtwInfo structure, which looks like this:

```
typedef struct FtwInfo {
        unsigned int init;
        FtwProc proc;
        char *curPath;
        size_t curPathLen;
        size_t curPathAllocSize;
        size_t startPathLen;
        char *curFile;
        size_t curFileLen;
        int curType;
        struct Stat curStat;
        int noAutoMallocAndFree;
        int dirSeparator;
        char rootDir[4];
```

```
        int autoGrow;
        size_t depth;
        size_t maxDepth;
        size_t numDirs;
        size_t numFiles;
        size_t numLinks;
        int reserved;
        void *cip;
        void *userdata;
} FtwInfo;
```

Do **not** use the init, proc, curPathAllocSize, noAutoMallocAndFree, dirSeparator, rootDir, or reserved fields.

The proc field is of type FtwProc, which is:

```
typedef int (*FtwProc)(const FtwInfoPtr ftwip);
```

Your FtwProc is called during Ftw functions, and it should return (-1) if they should abort directory traversal, or 0 if they should continue to iterate through the directory tree.

You may use the following fields only from within your FtwProc callback function:

The curPath field is a C-string which contains the complete or relative pathname of the file or directory. Important: treat this field as read-only!

curPathLen is the length of the string (i.e. strlen(curPath)).

startPathLen is the length of the starting directory. An easy way to compute pathnames relative to the starting directory is to simply use curPath + startPathLen + 1.

curFile is the filename-only portion of the pathname (also known as the basename of the path). Note that curFile is really an offset into curPath (so do not modify curFile).

curFileLen is the length of curFile.

curType will be set to 'd' if the pathname is a directory, '-' for a regular file, or 'l' for a symbolic link. This corresponds to the first character in the lines from UNIX "ls -l".

curStat contains useful information such as the modification timestamp, file size, etc.

depth is the current subdirectory recursion level, i.e., how deep into the directory tree you are. You may want to monitor the depth from your FtwProc and have your function return an error if the depth gets too deep.

cip is a pointer to your FTPConnectionInfo structure, but only if your FtwProc was called by FTPFtw.

userdata is a variable given to you for private use, so you can set it to whatever you like. This can be useful if you need to access other parts of your program from within your FtwProc without having to declare a global variable.

The following fields are kept as statistics and may be monitored from within your FtwProc callback function, and also after you call Ftw or FTPFtw. You will need to cast it to a FTPCIPtr before using it.

maxDepth is the deepest level reached into the directory tree.

numDirs is the number of items that were directories.

numFiles is the number of items that were regular files.

numLinks is the number of items that were symbolic links.

Function reference

FTPAbortDataTransfer	FTPGetOneFile3	FTPRmdir
FTPChdir	FTPInitConnectionInfo	FTPShutdownHost
FTPChdirAndGetCWD	FTPInitLibrary	FTPStrError
FTPChdir3	FTPIsDir	FTPStrError2
FTPChdirList	FTPIsRegularFile	FTPSymlink
FTPChmod	FTPList	FTPUmask
FTPCloseHost	FTPListToMemory	FTPUtime
FTPCmd	FTPListToMemory2	CopyLineList

FTP functions

FTPAbortDataTransfer

```
void FTPAbortDataTransfer(const FTPCIPtr cip);
```

Prior versions of the library advised you to call this function directly. The new recommendation is to use a progress meter function, and if you would like to abort a transfer in progress, set the cancelXfer field to 1 (in the FTPCIPtr passed to your progress meter callback function). This gives you the capability of aborting a transfer without having to use a signal handler, and gives the library a better chance of aborting the transfer at a point where the server won't abruptly disconnect the entire session.

Nevertheless, you can call this function yourself if you must. Note that not doing this from a progress meter callback (in which case you should use the method described above) essentially means you'll be doing this from a signal handler, which is generally undesirable.

FTPChdir

```
int FTPChdir(const FTPCIPtr cip, const char *const cdCwd);
```

Changes the current remote working directory to the path specified by cdCwd. The path is system-dependent, so it need not be a UNIX-style path if the system you are connected to is not a UNIX system.

If the change succeeded, 0 is returned, otherwise a number less than zero is returned upon failure.

When you first connect, the default working directory is assumed to be the root directory for anonymous FTP access. For a user login, the default is often the home directory of the user.

FTPChdirAndGetCWD

int FTPChdirAndGetCWD(const FTPCIPtr cip, const char *const cdCwd, char *const newCwd, const size_t newCwdSize);

This is a combination of FTPChdir and FTPGetCWD. The reason this can be useful is that some servers return the new path after a change of directory, thus saving a PWD.

If the change succeeded, 0 is returned, otherwise a number less than zero is returned upon failure.

FTPChdir3

int FTPChdir3(FTPCIPtr cip, const char *const cdCwd, char *const newCwd, const size_t newCwdSize, int flags);

Changes the current remote working directory to the path specified by cdCwd. The path is system-dependent, so it need not be a UNIX-style path if the system you are connected to is not a UNIX system.

This is a more advanced version of FTPChdir, which is controlled by the flags parameter. The flags must be either 0 (kChdirOnly), or a bitwise-OR of one or more of the following:

- kChdirAndMkdir - Change to the directory, creating it if necessary.
- kChdirAndGetCWD - Change to the directory, and note the new full working directory in the newCwd parameter.
- kChdirOneSubdirAtATime - Change to the directory, one sub-directory at a time. For example, if the directory was

"pub/FreeBSD/ISO-IMAGES", it would essentially be doing "cd pub" followed by "cd FreeBSD" followed by "cd ISO-IMAGES".

- kChdirFullPath - Change to the directory, as one complete pathname, i.e. "cd pub/FreeBSD/ISO-IMAGES".

The kChdirOneSubdirAtATime option is useful for strict RFC 1738 URL compliance. However, if strict compliance is not required, it is recommended that you use both kChdirOneSubdirAtATime and kChdirFullPath, i.e. (kChdirFullPath|kChdirOneSubdirAtATime). This would have the library attempt to change to the complete path, and if that fails, then fallback and try one subdirectory at a time.

If kChdirAndGetCWD is not used, then you may pass NULL for newCwd and 0 for newCwdSize.

FTPChdirList

```
int FTPChdirList(FTPCIPtr cip, FTPLineListPtr const cdlist, char
*const newCwd, const size_t newCwdSize, int flags);
```

This is identical to FTPChdir3, only instead of a pathname (which can be thought of as a list of subdirectory nodes) you pass a FTPLineList with each line being one subdirectory node.

Odds are you won't use this function, unless you use FTPDecodeURL which provides you with a FTPLineList as the directory path component.

FTPChmod

```
int FTPChmod(const FTPCIPtr cip, const char *const pattern, const
char *const mode, const int doGlob);
```

This is equivalent of the UNIX /bin/chmod program, only for remote files. This is not in the FTP standard, but many UNIX hosts implement this as a site-specific command.

If the mode change succeeded, 0 is returned, otherwise a number less than zero is returned upon failure.

FTPCloseHost

```
int FTPCloseHost(const FTPCIPtr cip);
```

Closes the connection to the current host, and disposes the library's data structures associated with it. This function may block because the remote server is notified that we want to close the connection via the FTP protocol command "QUIT" and a reply to that command is read back before cleanup is complete.

Upon a normal close, 0 is returned, otherwise if something bizarre happened a number less than zero is returned.

FTPCmd

```
int FTPCmd(const FTPCIPtr cip, const char *const cmdspec, ...);
```

This allows you to issue an FTP command directly on the control connection. You do not get back the textual response string, but you are returned the first digit of the numeric response, which will be from 1 to 5, or a negative error code.

The function behaves like printf, so you can pass a variable number of parameters.

Example:

```
idleAmt = 300;
err = FTPCmd(cip, "SITE IDLE %d", idleAmt);
if (err == 2) {
        /* success */
}
```

FTPDecodeURL

```
int FTPDecodeURL(const FTPCIPtr cip, char *const url,
FTPLineListPtr cdlist, char *const fn, const size_t fnsize, int
*const xtype, int *const wantnlst);
```

The purpose of this function is to parse a RFC 1738 FTP URL, such as ftp://ftp.ncftp.com/pub/ncftp/README. The URL is of the form ftp://<user>:<password>@<host>:<port>/<url-path><;type>.

Upon return, it will return a "chdir list" (FTPLineList), with each line representing one directory level. This is required by RFC 1738, so that instead of doing a chdir /pub/ncftp, you need to do a chdir pub followed by a chdir ncftp. See FTPChdirList for an easy way to change to chdir list.

It returns an integer indicating if the URL was successfully parsed. Zero means success, while the error code kMalformedURL means that it appeared to be a URL, but it had syntax errors. The error code kNotURL means that it didn't even resemble a URL at all.

The fn parameter points to a string to hold the filename for the requested file. For the example above, it would be set to "README". For a directory URL, this will be set to an empty string. The fnsize parameter should be the maximum size of the fn buffer.

Besides giving you back a chdir list, it may also set the user, pass, and port field of the connection structure, if the URL contains those fields, and optionally the transfer type or whether the URL should be a directory listing.

If the xtype parameter is not NULL, it will write whether the transfer type was specified, as either kTypeAscii or kTypeBinary. If the wantnlst parameter is not NULL, it will write whether the user requested a listing instead of a download of that directory.

The url is modified by the function, so make a copy of it if you need to preserve the contents.

Example:

```
rc = FTPDecodeURL(&ci, url, &cdlist, urlfile,
sizeof(urlfile),
                &xtype, NULL);
if (rc == kMalformedURL) {
                fprintf(stderr, "Malformed URL: %s\n", url);
} else if (rc == kNotURL) {
                fprintf(stderr, "Not a URL: %s\n", url);
} else {
                /* URL okay */
                printf("open %s %u\n", ci.host, (unsigned
int) ci.port);
                printf("user %s\n", ci.user);
                if (ci.pass[0] != '\0')
                                printf("pass %s\n",
ci.pass);
                for (lp = cdlist.first; lp != NULL; lp =
lp->next)
                                printf("cd %s\n",
lp->line);
                printf("type %c\n", xtype);
```

```
                    if (urlfile[0] != '\0')
                                printf("get %s\n",
        urlfile);
        }
```

FTPDelete

int FTPDelete(const FTPCIPtr cip, const char *const pattern, const int recurse, const int doGlob);

Removes remote files on the remote system, like /bin/rmdir does locally. This can also delete entire directories, if recursion is specified.

The doGlob parameter must be set to either kGlobYes or kGlobNo. When set, the pattern is considered a shell-wildcard-style regular expression.

The recurse parameter must be set to either kRecursiveYes or kRecursiveNo. When set, the library attempts to remove all files and subdirectories also (i.e. like /bin/rm -rf on UNIX).

Example 1: Delete all files in the current directory whose names end in ".zip."

```
err = FTPDelete(cip, "*.zip", kRecursiveNo, kGlobYes);
```

Example 2: Delete one file whose name is "*README*", but not files named "README" nor "*README-NOW*".

```
err = FTPDelete(cip, "*README*", kRecursiveNo, kGlobNo);
```

If all deletions succeeded, 0 is returned, otherwise a number less than zero is returned if one or more deletions failed. All files matched are attempted to be deleted, so if one deletion fails, that does not cause the remaining list to be aborted.

FTPFileExists

int FTPFileExists(const FTPCIPtr cip, const char *const file);

This tries to determine if a file or directory specified in the file parameter exists on the remote server.

Unfortunately this can be a very expensive operation on older servers because there was no standard functionality available in the protocol specification. On these servers, the library tries a variety of methods until it succeeds or exhausts the list. The good

news is that once the library has found a reliable way that works on the remote server, it remembers this for subsequent iterations so it does not need to repeat the learning process.

If the item exists, 0 (kNoErr) is returned. If the server has a reliable way to determine file existence and the file did not exist, kErrNoSuchFileOrDirectory is returned. Otherwise, the result is inconclusive and a negative error code is returned.

FTPFileModificationTime

int FTPFileModificationTime(const FTPCIPtr cip, const char *const file, time_t *const mdtm);

This tries to determine the last modification timestamp of the remote file specified by file.

This may or may not work, because this relies upon the implementation of the "MDTM" low-level FTP command. There are still a lot of traditional servers out there that do not support it nor the "SIZE" command.

There may also be a question of whether the time returned is in local time or GMT. Unfortunately this also varies among servers, most likely because there are no formal specifications of MDTM in RFC-959.

If the query succeeded, 0 is returned and the mdtm parameter is set to the timestamp of last modification, otherwise a number less than zero is returned upon failure.

FTPFileSize

int FTPFileSize(const FTPCIPtr cip, const char *const file, longest_int *const size, const int type);

This tries to determine how many bytes would be transferred if you downloaded file. The size in bytes is returned in the size parameter. The type parameter exists because this number varies depending on the transfer type. Set it to kTypeBinary, kTypeAscii, or kTypeEbcdic.

If the query succeeded, 0 is returned and the size parameter is set, otherwise a number less than zero is returned upon failure.

FTPFileSizeAndModificationTime

int FTPFileSizeAndModificationTime(const FTPCIPtr cip, const char *const file, longest_int *const size, const int type, time_t *const mdtm);

This function is a hybrid of FTPFileSize and FTPFileModificationTime. Some newer FTP servers have the capability to return both of these values at the same time, so it is more efficient than doing them individuallly. This tries to determine how many bytes would be transferred if you downloaded file. The size in bytes is returned in the size parameter. The type parameter exists because this number varies depending on the transfer type. Set it to kTypeBinary, kTypeAscii, or kTypeEbcdic.

The mdtm parameter is set to the time of last modification for the file or directory specified.

If the query succeeded, 0 is returned and the size and mdtm parameters are set, otherwise a number less than zero is returned if both could not be determined.

FTPFileType

int FTPFileType(const FTPCIPtr cip, const char *const file, int *const ftype);

This function can be used to determine if a particular pathname is a directory or regular file. It returns 'd' in the ftype parameter if the item was a directory, or '-' if it was a regular file. The ftype parameter return result should only be used if the function returns 0 (kNoErr). If the item exists but the type could not be determined, the error code kErrFileExistsButCannotDetermineType is returned. Other types of errors are returned as negative error codes.

This function calls FTPFileExists, which may be an expensive call.

Example:

```
FTPConnectionInfo ci;
int ftype, result;

if ((result = FTPFileType(&ci, "/pub/linux", &ftype)) == kNoErr)
{
        if (ftype == 'd')
```

```
                printf("It was a directory.\n");
        else
                printf("It was a file.\n");
} else {
                printf("An error occurred (%d).\n", result);
}
```

FTPFtw

```
int FTPFtw(const FTPCIPtr cip, const FtwInfoPtr ftwip, const char
*const dir, FtwProc proc);
```

This can be used like the C library function ftw(), except it works on the remote server rather than the local machine. "Ftw" stands for "file tree walk" and like the ftw() function, this function provides you an opportunity to recurse through an entire directory and process each file within it.

This function is actually the remote version of the Ftw function. The setup and necessary details for FTPFtw are the same as Ftw, so refer to Ftw for more information.

The dir parameter specifies the remote directory tree to walk.

The proc parameter is a callback to a custom function which you provide. This function will be called by FTPFtw for file or directory in the tree. A proc is of the FtwProc type, which is:

```
typedef int (*FtwProc)(const FtwInfoPtr ftwip);
```

When your FtwProc is called by FTPFtw on a remote directory tree, there are some slight differences than when it is called by Ftw on a local directory tree. The primary difference is that the curStat field (which is of type struct stat) of the FtwInfo structure is not entirely populated. The reason is that FTPFtw has to simulate stat() on the remote filesystem, whereas Ftw can actually call stat() and get all the information.

Specifically, you should only use the st_mode, st_size and st_mtime fields. In addition, st_size and st_mtime may not be available if the remote FTP server doesn't support SIZE or MDTM, but you can check the field values against kSizeUnknown and kModTimeUnknown respectively. You may find that st_size is invalid for a directory, or st_mtime is invalid for a directory (*wu-ftpd* is one example of a server that doesn't let you do MDTM on a directory).

The st_mode field is primarily populated so you can do S_ISDIR(st.st_mode) and S_ISREG(st.st_mode) like you would be accustomed to. The actual permission bits will be obtained if possible, but you shouldn't rely upon the presence or validity of these bits.

FTPFtw tries a variety of methods to figure out how to recurse a remote directory using the FTP protocol, but since the protocol doesn't formally specify a mechanism to do that, FTPFtw may not be able to traverse the directory structure if a remote FTP server doesn't support enough functionality to enable FTPFtw to do this.

Compounding this problem, the FTP protocol does not provide a mechanism to detect symbolic links. FTPFtw can sometimes detect links depending on the server software, but some servers could fool FTPFtw into following a symbolic link to another directory which may result in infinite recursion! To prevent this, your FtwProc should check the depth field and return an error if the remote directory tree is getting too deep.

This function may call FTPFileType, which could be an expensive call. It may call it *numerous* times! In any event, traversing a remote directory tree is a very expensive operation. Be very careful and considerate when you use FTPFtw.

Example: Traverse "/pub" on the current remote host, and attempt to remove any "core" files found. This is an abridged example, but the library includes a more detailed example, ncftpftw.c, in the samples directory.

```
static int
MyRemoteFtwProc(const FtwInfoPtr ftwip)
{
    FTPCIPtr cip = (FTPCIPtr) ftwip->cip;
    longest_int fSize;
    time_t fTime;

    if (ftwip->depth > 20) {
            /* Prevent possible infinite recursion */
            return (-1);    /* cancel traversal */
    }

    if (ftwip->curType == 'd') {
            printf("Directory: %s\n", ftwip->curPath);
    } else if (ftwip->curType == 'l') {
```

```
                printf("Symlink: %s\n", ftwip->curPath);
        } else {
                fSize = ftwip->curStat.st_size;
                fTime = ftwip->curStat.st_mtime;
                printf("File: %s (size=%lld, mtime=%u)\n",
                        ftwip->curPath, fSize, fTime);

                if (strcmp(ftwip->curFile, "core") == 0)
                        (void) FTPDelete(cip, ftwip->curPath, 0,
0);
        }

        return (0);     /* continue traversal */
}  /* MyFtwProc */

/* ... */
{
        FtwInfo ftwi;
        int rc;

        FtwInit(&ftwi);
        if ((rc = FTPFtw(cip, &ftwi, "/pub",
MyRemoteFtwProc)) != 0) {
                /* Traversal failed */
                /* ... */
        }

        (void) printf("Stats: rc=%d #dirs=%u #files=%u #links=%u
maxdepth=%u\n",
                rc, ftwi.numDirs, ftwi.numFiles, ftwi.numLinks,
ftwi.maxDepth);

        FtwDispose(&ftwi);
}
```

FTPGetCWD

int FTPGetCWD(const FTPCIPtr cip, char *const newCwd, const size_t newCwdSize);

This writes up to newCwdSize bytes of the pathname of the current remote working directory in newCwd.

If the request succeeded, 0 is returned, otherwise a number less than zero is returned upon failure.

FTPGetFiles, FTPGetFiles2

int FTPGetFiles(const FTPCIPtr cip, const char *const pattern, const char *const dstdir, const int recurse, const int doGlob);

int FTPGetFiles2(const FTPCIPtr cip, const char *const pattern, const char *const dstdir, const int recurse, const int doGlob, const int xtype, const int resumeflag, const int appendflag);

These functions have been superceded by FTPGetFiles3 and are only provided for backward-compatibility with code based off of older versions of the library.

FTPGetFiles3

int FTPGetFiles3(const FTPCIPtr cip, const char *pattern, const char *const dstdir, const int recurse, int doGlob, const int xtype, const int resumeflag, int appendflag, const int deleteflag, const int tarflag, const FTPConfirmResumeDownloadProc resumeProc, int reserved);

Downloads (reads) files from the remote system.

The pattern parameter is the remote pathname to download. When coupled with globbing, the pattern can denote a regular expression so that multiple files can be downloaded with a single wildcard expression.

The dstdir parameter is local directory where the files are to be written. If you want them placed in the current local directory, just use "." as the dstdir. The files retrieved are named to be the same as they were on the remote system.

The recurse parameter must be set to either kRecursiveYes or kRecursiveNo. When set the library attempts to download the entire directory structure if pattern is a directory. Recursion is not very portable; For it to work properly, the remote server must produce UNIX-like listings for the LIST primitive, and it must also support the "-R" flag (LIST -R). The good news is that many non-UNIX FTP servers do try to emulate that behavior.

The doGlob parameter must be set to either kGlobYes or kGlobNo. When set, the pattern is considered a shell wildcard-style regular expression, and FTPRemoteGlob is used if needed to build a list of files to retrieve.

The xtype parameter must be set to either kTypeAscii or kTypeBinary. Unless the file is known to be stored in the host's native text format, where ASCII text translation to your host's text format would be useful, you should use binary transfer type.

The resumeflag parameter must be set to either kResumeYes or kResumeNo. When set, the library will attempt to resume the download. This is done if the local file already exists and is smaller than the remote file. In addition, the library tries to use modification times when possible to determine if this should be done.

The appendflag parameter must be set to either kAppendYes or kAppendNo. When set, the entire remote file is downloaded, and the local file is appended to, if present. Generally that is not very useful.

The deleteflag parameter must be set to either kDeleteYes or kDeleteNo. When set, after the file is downloaded successfully the remote file is deleted. This requires the applicable permissions on the remote file.

The tarflag parameter must be set to either kTarYes or kTarNo. When set and the recurse parameter is also set, the library attempts to see if the server supports "on-the-fly TAR" and uses that to transfer the entire directory. The advantages to this are two-fold; first, it will be faster since a separate data connection is not required for each file in the directory, and second, since it is a TAR file it also preserves the exact file permissions and timestamps. The downside to using this mode is that you always get the entire directory structure. There is no resumption of broken downloads.

The resumeProc parameter can be set to a callback function to give you fine-grained control on what to do when the local file already exists. Your function can determine whether to resume the download, overwrite and download the entire file, append to the local file, or skip the file transfer. This is useful for interactive programs where you want the user to choose which action to take. This parameter must be set to kNoFTPConfirmResumeDownloadProc to indicate you do not want a callback function, which is most of the time. Otherwise your callback function should be a valid FTPConfirmResumeDownloadProc, which is declared as follows:

```
typedef int (*FTPConfirmResumeDownloadProc)(
```

```
                    const char *volatile *localpath,
                    volatile longest_int localsize,
                    volatile time_t localmtime,
                    const char *volatile remotepath,
                    volatile longest_int remotesize,
                    volatile time_t remotetime,
                    volatile longest_int *volatile startPoint
    );
```

Your callback function should examine the parameters passed into it, and return one of the following result codes:

- kConfirmResumeProcSaidSkip
- kConfirmResumeProcSaidResume
- kConfirmResumeProcSaidOverwrite
- kConfirmResumeProcSaidAppend
- kConfirmResumeProcSaidBestGuess

Your callback function should also set the startPoint to the offset into the file where to resume the download at. You also have the option of changing the localpath parameter, which could be useful if your function decides it can save to a new name.

The reserved parameter must be set to zero. This is reserved for future use.

If all transfers succeeded, FTPGetFiles3 returns 0 (kNoErr), otherwise a number less than zero is returned if one or more transfers failed. All files matched are attempted to be transferred, so if one fails, that does not cause the remaining list to be aborted.

Example 1: Retrieve all files in the current directory whose names end in ".zip" and write them to the "/tmp" local directory.

```
    err = FTPGetFiles3(cip, "*.zip", "/tmp", kRecursiveNo,
kGlobYes, kTypeBinary,
                                        kResumeNo, kAppendNo,
kDeleteNo, kTarNo, kNoFTPConfirmResumeDownloadProc, 0);
```

Example 2: Fetch one file whose name is "*README*", but not files named "README" nor "*README-NOW*", and write it to the current local directory.

```
    err = FTPGetFiles3(cip, "*README*", ".", kRecursiveNo,
kGlobNo, kTypeBinary,
```

kResumeNo, kAppendNo, kDeleteNo, kTarNo, kNoFTPConfirmResumeDownloadProc, 0);

Example 3: Fetch the entire contents of the directory "/pub/zzz" into a local directory named "/tmp/aaa/bbb/zzz:"

```
err = FTPGetFiles3(cip, "/pub/zzz", "/tmp/aaa/bbb",
kRecursiveYes, kGlobNo, kTypeBinary,
kResumeNo, kAppendNo, kDeleteNo, kTarNo, kNoFTPConfirmResumeDownloadProc, 0);
```

FTPGetOneFile, FTPGetOneFile2

```
int FTPGetOneFile(const FTPCIPtr cip, const char *const file, const char *const dstfile);

int FTPGetOneFile2(const FTPCIPtr cip, const char *const file, const char *const dstfile, const int xtype, const int fdtouse, const int resumeflag, const int appendflag);
```

These functions have been superceded by FTPGetOneFile3 and are only provided for backward-compatibility with code based off of older versions of the library.

FTPGetOneFile3

```
int FTPGetOneFile3(const FTPCIPtr cip, const char *const file, const char *const dstfile, const int xtype, const int fdtouse, const int resumeflag, const int appendflag, const int deleteflag, const FTPConfirmResumeDownloadProc resumeProc, int reserved);
```

This is provides a way to download a single remote file and write it under a different name (if required) locally.

The file parameter specifies the remote file name to download, and the dstfile parameter specifies the local file name to save it as.

The xtype parameter must be set to either kTypeAscii or kTypeBinary. Unless the file is known to be stored in the host's native text format, where ASCII text translation to your host's text format would be useful, you should use binary transfer type.

The fdtouse parameter must be either an opened file descriptor for writing, or less than zero if the function should open the file as needed. Most of the time you will use (-1) for this parameter and let the library open the local file for you.

The resumeflag parameter must be set to either kResumeYes or kResumeNo. When set, the library will attempt to resume the download. This is done if the local file already exists and is smaller than the remote file. In addition, the library tries to use modification times when possible to determine if this should be done.

The appendflag parameter must be set to either kAppendYes or kAppendNo. When set, the entire remote file is downloaded, and the local file is appended to, if present. Generally that is not very useful.

The deleteflag parameter must be set to either kDeleteYes or kDeleteNo. When set, after the file is downloaded successfully the remote file is deleted. This requires the applicable permissions on the remote file.

The resumeProc parameter can be set to a callback function to give you fine-grained control on what to do when the local file already exists. See the description of FTPGetFiles3 for more information on how to use this.

The reserved parameter must be set to zero. This is reserved for future use.

Example: Retrieve a file named "/pub/README.TXT" and write it as "/tmp/xx".

```
err = FTPGetOneFile2(cip, "/pub/README.TXT", "/tmp/xx",
kTypeBinary, -1, kResumeYes, kAppendNo);
```

If the fetch succeeded, 0 is returned, otherwise a number less than zero is returned upon failure.

FTPInitConnectionInfo

```
int FTPInitConnectionInfo(const FTPLIPtr lip, const FTPCIPtr cip,
size_t bufSize);
```

Before you can attempt to connect to a host using the FTP protocol, you must have first initialized the library using FTPInitLibrary .

Then, when you want to open a host, you use this function to initialize a session structure to the default values. After you have done that, you may change whatever non-default values you need.

Typically you use a global variable to hold the library's session information, and then pass a pointer to it for all FTP functions.

The bufsize parameter specifies the size of the data transfer I/O buffer to use, which is reserved using malloc(). You should use kDefaultFTPBufSize as the value for bufsize in most cases.

Here are some default values that may be of interest:

```
cip->port = kDefaultFTPPort;
cip->maxDials = 3;
cip->redialDelay = 20;
cip->xferTimeout = 600;
cip->connTimeout = 30;
cip->ctrlTimeout = 135;
cip->dataPortMode = kFallBackToSendPortMode;
```

If the initialization succeeded, 0 is returned, otherwise a number less than zero is returned upon failure.

Example:

```
FTPLibraryInfo li;
FTPConnectionInfo fi;

if (FTPInitLibrary(&li) != kNoErr) {
        /* ... init library failed ... */
}
if (FTPInitConnectionInfo(&li, &fi, kDefaultFTPBufSize) !=
kNoErr) {
        /* ... init session failed ... */
}
strcpy(fi.host, "hostname.here.com");
strcpy(fi.user, "username");
strcpy(fi.pass, "password");
/* ... */
```

FTPInitLibrary

```
int FTPInitLibrary(const FTPLIPtr lip);
```

Before you can attempt to connect to a host using the FTP protocol, you must have first initialized the library using this function. Typically, you use a global variable to hold the library's internal data structures.

Example:

```
FTPLibraryInfo li;

if (FTPInitLibrary(&li) != kNoErr) {
        /* ... init library failed ... */
}
```

FTPIsDir

```
int FTPIsDir(const FTPCIPtr cip, const char *const dir);
```

This function returns 1 if the pathname is a valid directory, zero if it is a plain file, or a negative error code if an error occurred. This function may also return 0 if the item is known to exist but the file type could not be determined (so, if the item exists it is assumed to be a regular file; however due to the way the test is performed, the item in question could not be changed to so it is most likely a regular file, and less likely an inaccessible directory).

This function calls FTPFileExists, which may be an expensive call.

Example:

```
FTPConnectionInfo ci;

if (FTPIsDir(&ci, "/pub/linux") > 0) {
        /* directory */
}
```

FTPIsRegularFile

```
int FTPIsRegularFile(const FTPCIPtr cip, const char *const file);
```

This function returns 1 if the pathname is a file, zero if it is a valid directory, or a negative error code if an error occurred. This function may also return 0 if the item is known to exist but the file type could not be determined (so, if the item exists it is assumed to be a regular file; however due to the way the test is performed, the item in question could not be changed to so it is most likely a regular file, and less likely an inaccessible directory).

This function calls FTPFileExists, which may be an expensive call.

Example:

```
FTPConnectionInfo ci;

if (FTPIsRegularFile(&ci, "/pub/linux/README") > 0) {
        /* file */
}
```

FTPList

int FTPList(const FTPCIPtr cip, const int outfd, const int longMode, const char *const lsflag);

This is a simple way to get a remote directory listing dumped to the screen. The outfd parameter specifies which file descriptor to write to, so you do not necessarily have to use stdout (file descriptor 1) here.

The longMode parameter determines which method of listing you want. The FTP Protocol currently has two methods, one which is a simple one file per line style (longMode == 0), and another host-specific output method (longMode == 1). Typically for UNIX systems, these methods equate to "/bin/ls -1" and "/bin/ls -la".

The lsflag parameter can be used to give a specific directory (or file) to list, and also as a way to specify alternate flags. For example, many systems accept UNIX's /bin/ls flags, like "-CF".

Example 1: Dump a simple listing of the current directory to the screen.

```
err = FTPList(cip, 1, 0, NULL);
```

Example 2: Dump a long listing of the directory "/pub" to the screen.

```
err = FTPList(cip, 1, 1, "/pub");
```

Example 3: Simulate "/bin/ls -CF" behavior on the current remote working directory.

```
err = FTPList(cip, 1, 0, "-CF");
```

Note: This really isn't too useful for doing programmatical analysis with. If you want to do that, FTPListToMemory is a much better choice.

If the listing succeeded, 0 is returned, otherwise a number less than zero is returned upon failure.

FTPListToMemory

int FTPListToMemory(const FTPCIPtr cip, const char *const pattern, const FTPLineListPtr lines, const char *const lsflags);

This allows you to get a directory listing of a remote directory, and have it loaded into a dynamic data structure.

The pattern parameter specifies a directory to list, or a wildcard expression of files to list. The output is loaded into the FTPLineList specified by the lines parameter. The lsflags parameter lets you specify additional /bin/ls style flags. If you use it, you must have a trailing space, like "-CF " and if you don't want any flags, you must use an empty string, like "".

Example 1: Get a listing of files in the /pub directory.

```
FTPLineList fileList;

err = FTPListToMemory(cip, "/pub", &fileList, "");
```

Example 2: Get a long listing of files in the current directory, sorted by older files first.

```
FTPLineList fileList;

err = FTPListToMemory(cip, ".", &fileList, "-lrt ");
```

Note: If all you want is a list of files, it may be easier to just use FTPRemoteGlob. That function calls FTPListToMemory for you.

If the listing succeeded, 0 is returned, otherwise a number less than zero is returned upon failure.

FTPListToMemory2

int FTPListToMemory2(const FTPCIPtr cip, const char *const pattern, const FTPLineListPtr llines, const char *const lsflags, const int blanklines, int *const tryMLSD);

This function is identical to FTPListToMemory, except it has a few more parameters (which you probably won't need, which is why you'll probably not use this function). In fact, FTPListToMemory is actually FTPListToMemory(cip, pattern, llines, lsflags, 1, NULL).

The blanklines parameter can be set to 0 if you want blank lines removed from the list output.

The tryMLSD parameter can be used if you want to try the new "MLSD" FTP command primitive which newer FTP servers support to enable

machine-readable (i.e. parseable!) directory listings. To use it, set an integer local variable to 1 and pass a pointer to it to this function. If the listing was successful and MLSD was used, your variable will remain 1. If MLSD failed or could not be used, your variable will be set to 0. If you pass NULL or a pointer to a variable containing 0 for the tryMLSD parameter, then MLSD is not attempted.

FTPLocalGlob

```
int FTPLocalGlob(FTPCIPtr cip, FTPLineListPtr fileList, const char
*pattern, int doGlob);
```

This gives you a way to do a shell-expansion of a wildcard pattern on the local host. You can use this to gather a list of files, and then do something with the list.

The pattern parameter specifies a wildcard expression. The pattern may also contain the tilde-notation popularized by /bin/csh. These are expanded by the library, and then /bin/sh is used in conjunction with /bin/ls to produce the list of files for you.

The doGlob parameter may seem redundant, but if you set it to kGlobNo you can have the function only do the tilde expansion.

Example: Get a list of all C source files in the current directory.

```
FTPLineList fileList;

err = FTPLocalGlob(cip, &fileList, "*.c", &fileList,
kGlobYes);
```

If the globbing succeeded, 0 is returned, otherwise a number less than zero is returned upon failure.

FTPLoginHost

```
int FTPLoginHost(const FTPCIPtr cip);
```

This uses the values for the user, pass, and acct fields from the FTPConnectionInfo structure to sign on to the remote system, and reads the connect message from the server.

You should not need to use this function, since FTPOpenHost does this for you automatically.

If the login was successful, 0 is returned, otherwise a number less than zero is returned upon failure.

FTPMkdir

int FTPMkdir(const FTPCIPtr cip, const char *const newDir, const int recurse);

This creates a new directory on the remote host. The recurse parameter specifies whether it should attempt to create all directories in the path and not just the last node (this emulates "/bin/mkdir -p"). The recurse parameter must be set to either kRecursiveYes or kRecursiveNo.

If the directory creation succeeded, 0 is returned, otherwise a number less than zero is returned upon failure.

FTPOpenHost

int FTPOpenHost(const FTPCIPtr cip);

This is routine is used to establish the connection to the remote host. Before you can use it, you must set some of the fields in your FTPConnectionInfo structure.

The host field must be set to the name of the remote host. You may also use an IP address in place of a hostname.

The user and pass fields must be set if you are not logging in anonymously. If you leave them unset, an anonymous login is attempted, with the password being a guess at a the email address for the user running your program. There is also an acct field which you may set if for some reason the remote server requires an account designation in addition to a user and password.

The port parameter may be set to a non-standard port if you wish to connect to an FTP server running on a port other than the default port number, 21.

The library has a built-in facility to "redial" a host if it could not login in the first time. You may set the maxDials field to a number greater than one to turn that on. If you do that, you may want to tune the time delay between dials by setting the redialDelay field.

Example 1: Establish an anonymous connection to ftp.cdrom.com.

```
	FTPLibraryInfo li;
	FTPConnectionInfo fi;

	if (FTPInitLibrary(&li) != kNoErr) {
		/* ... init library failed ... */
	}
	if (FTPInitConnectionInfo(&li, &fi, kDefaultFTPBufSize) !=
kNoErr) {
		/* ... init session failed ... */
	}

	strcpy(fi.host, "ftp.cdrom.com");
	if (FTPOpenHost(&fi) != kNoErr) {
		/* ... could not open a connection there ... */
	}
```

Example 2: Establish an non-anonymous connection to ftp.cs.unl.edu.

```
	FTPLibraryInfo li;
	FTPConnectionInfo fi;

	if (FTPInitLibrary(&li) != kNoErr) {
		/* ... init library failed ... */
	}
	if (FTPInitConnectionInfo(&li, &fi, kDefaultFTPBufSize) !=
kNoErr) {
		/* ... init session failed ... */
	}

	strcpy(fi.host, "ftp.cs.unl.edu");
	strcpy(fi.user, "gleason");
	strcpy(fi.pass, "mypassword");
	fi.maxDials = 2;
	fi.redialDelay = 120;                    /* Wait two
minutes between. */

	if (FTPOpenHost(&fi) != kNoErr) {
		/* ... could not open a connection there ... */
	}
```

If the connection was established and the login succeeded, 0 is returned, otherwise a number less than zero is returned upon failure.

FTPOpenHostNoLogin

int FTPOpenHostNoLogin(const FTPCIPtr cip);

This is identical to FTPOpenHost, except that FTPLoginHost is not performed.

If the connection was established, 0 is returned, otherwise a number less than zero is returned upon failure.

FTPPerror

void FTPPerror(const FTPCIPtr cip, const int err, const int eerr, const char *const s1, const char *const s2);

This is roughly the library's equivalent to the perror() C library function. It returns a textual error message from a library error number (which will be negative numbers). Most non-console applications won't have a stderr stream to print to, so instead of this function you would use FTPStrError2.

It is often useful to print an error message when an error occurs, like:

```
if (FTPChdir(cip, "/pub") < 0) {
        /* print an error */
}
```

However, many of the errors aren't very helpful, because the error code is meant be propagated up through a calling chain. So for this particular example, you'd often see get the error code kErrCWDFailed ("remote chdir failed") as the reason. For this reason, when a non-library related error occurs, it is often best to inspect the textual response *from the server*, so the error from the server is printed instead of the generic library error. The server message would be the one you would see if you were logging to the debugLog, such as "550 Permission Denied".

To handle this, you can call the FTPPerror function with an expected error code, so when that particular error occurs the server's error string is printed.

The s1 and s2 parameters are words to print along with the error. One or both may be NULL if you only need to print one or no extra strings.

```
        if (FTPChdir(cip, dirname) < 0)
                FTPPerror(cip, cip->errNo, kErrCWDFailed, "cd",
dirname);
```

FTPPutFiles, FTPPutFiles2

```
int FTPPutFiles(const FTPCIPtr cip, const char *const pattern,
const char *const dstdir, const int recurse, const int doGlob);

int FTPPutFiles2(const FTPCIPtr cip, const char *const pattern,
const char *const dstdir, const int recurse, const int doGlob, const
int xtype, const int appendflag, const char *const tmppfx, const
char *const tmpsfx);
```

These functions have been superceded by FTPPutFiles3 and are only provided for backward-compatibility with code based off of older versions of the library.

FTPPutFiles3

```
int FTPPutFiles3(const FTPCIPtr cip, const char *const pattern,
const char *const dstdir, const int recurse, const int doGlob, const
int xtype, const int appendflag, const char *const tmppfx, const
char *const tmpsfx, const int resumeflag, const int deleteflag,
const FTPConfirmResumeUploadProc resumeProc, int reserved);
```

Uploads (writes) files to the remote system.

The pattern parameter is the remote pathname to upload. When coupled with globbing, the pattern can denote a regular expression so that multiple files can be uploaded with a single wildcard expression.

The dstdir parameter is remote directory where the files are to be written. If you want them placed in the current remote directory, just use "." as the dstdir. The files sent are named to be the same as they were on the local system.

The recurse parameter must be set to either kRecursiveYes or kRecursiveNo. When set the library attempts to upload the entire directory structure if pattern is a directory. Recursion for this function should be portable. It does not require any special treatment from the remote server (unlike the FTPGetFiles3 function).

The doGlob parameter must be set to either kGlobYes or kGlobNo. When set, the pattern is considered a shell wildcard-style regular expression, and FTPLocalGlob is used if needed to build a list of files to retrieve.

The xtype parameter must be set to either kTypeAscii or kTypeBinary. Unless the file is known to be stored in the local host's native text format, where ASCII text translation to the remote host's text format would be useful, you should use binary transfer type.

The appendflag parameter must be set to either kAppendYes or kAppendNo. When set, the entire local file is uploaded, and the remote file is appended to, if present. Generally that is not very useful.

The tmppfx parameter must be set to the prefix of the temporary file to use, or NULL if no prefix should be used. The tmpsfx parameter must be set to the suffix of the temporary file to use, or NULL if no suffix should be used. These two parameters are used to create a temporary filename based on the real file name you want. If either of these is set, then the library uploads to a temporary file, and when the upload finishes, renames the temporary file to the real name. For example, if the file to upload is to be named "/tmp/aaa/bbb.txt" and the tmpsfx is ".TMP", then "/tmp/aaa/bbb.txt.TMP" is uploaded, and it it worked, renamed to "/tmp/aaa/bbb.txt".

The resumeflag parameter must be set to either kResumeYes or kResumeNo. When set, the library will attempt to resume the upload. This is done if the remote file already exists and is smaller than the local file. In addition, the library tries to use modification times when possible to determine if this should be done.

The deleteflag parameter must be set to either kDeleteYes or kDeleteNo. When set, after the file is uploaded successfully the local file is deleted.

The resumeProc parameter can be set to a callback function to give you fine-grained control on what to do when the remote file already exists. Your function can determine whether to resume the upload, overwrite and upload the entire file, append to the remote file, or skip the file transfer. This is useful for interactive programs where you want the user to choose which action to take. This parameter must be set to

kNoFTPConfirmResumeUploadProc to indicate you do not want a callback function, which is most of the time. Otherwise your callback function should be a valid FTPConfirmResumeUploadProc, which is declared as follows:

```
typedef int (*FTPConfirmResumeUploadProc)(
                const char *volatile localpath,
                volatile longest_int localsize,
                volatile time_t localmtime,
                const char *volatile *remotepath,
                volatile longest_int remotesize,
                volatile time_t remotetime,
                volatile longest_int *volatile startPoint
);
```

Your callback function should examine the parameters passed into it, and return one of the following result codes:

- kConfirmResumeProcSaidSkip
- kConfirmResumeProcSaidResume
- kConfirmResumeProcSaidOverwrite
- kConfirmResumeProcSaidAppend
- kConfirmResumeProcSaidBestGuess

Your callback function should also set the startPoint to the offset into the local file where to resume the upload at. You also have the option of changing the remotepath parameter, which could be useful if your function decides it can save to a new name.

The reserved parameter must be set to zero. This is reserved for future use.

If all transfers succeeded, FTPPutFiles3 returns 0 (kNoErr), otherwise a number less than zero is returned if one or more transfers failed. All files matched are attempted to be transferred, so if one fails, that does not cause the remaining list to be aborted.

Example 1: Send all files in the current directory whose names end in ".zip" and write them to the "/tmp" remote directory.

```
err = FTPPutFiles3(cip, "*.zip", "/tmp", kRecursiveNo,
kGlobYes, kTypeBinary,
```

kAppendNo, NULL, NULL, kResumeNo, kDeleteNo, kNoFTPConfirmResumeUploadProc, 0);

Example 2: Send one file whose name is "*README*", but not files named "README" nor "*README-NOW*", and write it to the current remote directory.

err = FTPPutFiles3(cip, "*README*", ".", kRecursiveNo, kGlobNo, kTypeBinary,
kAppendNo, NULL, NULL, kResumeNo, kDeleteNo, kNoFTPConfirmResumeUploadProc, 0);

Example 3: Send the entire contents of the local directory "/usr/local/bin", creating a "/pub/bin" directory tree on the remote server:

err = FTPPutFiles3(cip, "/usr/local/bin", "/pub", kRecursiveYes, kGlobNo, kTypeBinary,
kAppendNo, NULL, NULL, kResumeNo, kDeleteNo, kNoFTPConfirmResumeUploadProc, 0);

Example 4: Send all files in the current directory whose names end in ".zip" and write them to the "/tmp" remote directory, using "AA" as the prefix to temporary files:

err = FTPPutFiles3(cip, "*.zip", "/tmp", kRecursiveNo, kGlobYes, kTypeBinary,
kAppendNo, "AA", NULL, kResumeNo, kDeleteNo, kNoFTPConfirmResumeUploadProc, 0);

FTPPutOneFile, FTPPutOneFile2

```
int FTPPutOneFile(const FTPCIPtr cip, const char *const file, const char *const dstfile);

int FTPPutOneFile2(const FTPCIPtr cip, const char *const file, const char *const dstfile, const int xtype, const int fdtouse, const int appendflag, const char *const tmppfx, const char *const tmpsfx);
```

These functions have been superceded by FTPPutOneFile3 and are only provided for backward-compatibility with code based off of older versions of the library.

FTPPutOneFile3

```
int FTPPutOneFile3(const FTPCIPtr cip, const char *const file, const char *const dstfile, const int xtype, const int fdtouse, const int appendflag, const char *const tmppfx, const char *const tmpsfx, const int resumeflag, const int deleteflag, const FTPConfirmResumeUploadProc resumeProc, int reserved);
```

This is provides a way to upload a single local file and write it under a different name (if needed) on the remote server.

The file parameter specifies the remote file name to upload, and the dstfile parameter specifies the remote file name to save it as.

The xtype parameter must be set to either kTypeAscii or kTypeBinary. Unless the file is known to be stored in the local host's native text format, where ASCII text translation to the remote host's text format would be useful, you should use binary transfer type.

The fdtouse parameter must be either an opened file descriptor for reading, or less than zero if the function should open the file as needed. Most of the time you will use (-1) for this parameter and let the library open the local file for you.

The appendflag parameter must be set to either kAppendYes or kAppendNo. When set, the entire local file is uploaded, and the remote file is appended to, if present. Generally that is not very useful.

The tmppfx parameter must be set to the prefix of the temporary file to use, or NULL if no prefix should be used. The tmpsfx parameter must be set to the suffix of the temporary file to use, or NULL if no suffix should be used. These two parameters are used to create a temporary filename based on the real file name you want. If either of these is set, then the library uploads to a temporary file, and when the upload finishes, renames the temporary file to the real name. For example, if the file to upload is to be named "/tmp/aaa/bbb.txt" and the tmpsfx is ".TMP", then "/tmp/aaa/bbb.txt.TMP" is uploaded, and it it worked, renamed to "/tmp/aaa/bbb.txt".

The resumeflag parameter must be set to either kResumeYes or kResumeNo. When set, the library will attempt to resume the upload. This is done if the remote file already exists and is smaller than the local file. In addition, the library tries to use modification times when possible to determine if this should be done.

The deleteflag parameter must be set to either kDeleteYes or kDeleteNo. When set, after the file is uploaded successfully the local file is deleted.

The resumeProc parameter can be set to a callback function to give you fine-grained control on what to do when the local file already exists. See the description of FTPPutFiles3 for more information on how to use this.

The reserved parameter must be set to zero. This is reserved for future use.

Example 1: Send a file named "/tmp/xx" and write it as "/pub/README".

```
err = FTPPutOneFile2(cip, "/tmp/xx", "/pub/README",
kTypeAscii, -1, kAppendNo,
                NULL, NULL, kResumeNo, kDeleteNo,
kNoFTPConfirmResumeUploadProc, 0);
```

Example 2: Send a file named "/tmp/xx" and write it temporarily as "/pub/README~", and when finished, name it "/pub/README".

```
err = FTPPutOneFile2(cip, "/tmp/xx", "/pub/README",
kTypeAscii, -1, kAppendNo,
                NULL, "~", kResumeNo, kDeleteNo,
kNoFTPConfirmResumeUploadProc, 0);
```

If the send succeeded, 0 is returned, otherwise a number less than zero is returned upon failure.

FTPRemoteGlob

```
int FTPRemoteGlob(FTPCIPtr cip, FTPLineListPtr fileList, const
char *pattern, int doGlob);
```

This gives you a way to do a shell-expansion of a wildcard pattern on the remote host. You can use this to gather a list of files, and then do something with the list.

The pattern parameter specifies a wildcard expression. The pattern is interpreted in a host-specific manner, but most hosts obey /bin/csh or /bin/sh notation.

The doGlob parameter is not used, so just set it to kGlobYes.

Example: Get a list of all C source files in the current remote directory.

```
FTPLineList fileList;
```

```
err = FTPRemoteGlob(cip, &fileList, "*.c", &fileList,
kGlobYes);
```

If the globbing succeeded, 0 is returned, otherwise a number less than zero is returned upon failure.

FTPRename

```
int FTPRename(const FTPCIPtr cip, const char *const oldname, const
char *const newname);
```

Lets you rename a remote file or directory.

If the renaming succeeded, 0 is returned, otherwise a number less than zero is returned upon failure.

FTPRmdir

```
int FTPRmdir(const FTPCIPtr cip, const char *const pattern, const
int recurse, const int doGlob);
```

Removes remote directories on the remote system, like /bin/rmdir does locally. Trying to remove a non-empty directory may or may not work, depending on the remote server. It won't on most UNIX servers.

The doGlob parameter must be set to either kGlobYes or kGlobNo. When set, the pattern is considered a shell-wildcard-style regular expression.

The recurse parameter must be set to either kRecursiveYes or kRecursiveNo. When set, the library attempts to remove all files and subdirectories also (i.e. like /bin/rm -rf on UNIX).

Example 1: Delete all subdirectories in the current remote directory whose names contain "tmp".

```
err = FTPRmdir(cip, "*tmp*", kRecursiveNo, kGlobYes);
```

Example 2: Delete one remote directory whose name is "/tmp".

```
err = FTPDelete(cip, "/tmp", kRecursiveNo, kGlobNo);
```

If all deletions succeeded, 0 is returned, otherwise a number less than zero is returned if one or more deletions failed. All files matched are attempted to be deleted, so if one deletion fails, that does not cause the remaining list to be aborted.

FTPShutdownHost

```
void FTPShutdownHost(const FTPCIPtr cip);
```

Forcibly closes the connection to the current host, and disposes the library's data structures associated with it.

Unlike FTPCloseHost, This function will not block, but you should use FTPCloseHost whenever possible because it does a close that is more polite to the remote host.

FTPStrError

```
const char *FTPStrError(int errNo);
```

This is the library's equivalent to the strerror() C library function. It returns a textual error message from a library error number (which will be negative numbers).

This function is often useful to dump an error message when an error occurs, like:

```
if (FTPChdir(cip, "/pub") < 0) {
        fprintf(stderr, "Could not cd to /pub: %s.\n",
                FTPStrError(cip->errNo));
}
```

However, many of the library's error messages aren't very helpful, because the error code is meant be propagated up through a calling chain. So for this particular example, you'd often see "remote chdir failed" as the reason. For this reason, it is often best to inspect the response *from the server*, which will probably have something more useful, such as "Permission denied".

Here's one way you could do this using FTPStrError (but in practice, see below for FTPStrError2 which does this better):

```
if (FTPChdir(cip, "/pub") < 0) {
        if (cip->errNo == kErrCWDFailed) {
                fprintf(stderr, "Could not cd to /pub:
%s.\n", cip->lastFTPCmdResultStr);
        } else {
                fprintf(stderr, "Could not cd to /pub:
%s.\n", FTPStrError(cip->errNo));
        }
}
```

FTPStrError2

```
char *FTPStrError2(const FTPCIPtr cip, int err, char *const dst,
const size_t dstsize, int expectedErr);
```

This function is the library's rough equivalent to strncpy(dst, strerror(errno), dstsize), except FTPStrError is used to get the library error message and you pass a library errNo rather than errno.

This function is often useful to handle errors that occur with library functions, such as:

```
        if (FTPChdir(cip, "/pub") < 0) {
                /* Inspect cip->errNo and log the error ... */
        }
```

However, many of the library's error messages aren't very helpful, because the error code is meant be propagated up through a calling chain. So for this particular example, you'd often see the error code kErrCWDFailed ("remote chdir failed") which isn't very helpful. For this reason, it is often best to inspect the response *from the server*, if the error code equals the generic error code for the function, then use the server message which will probably have something more useful, such as "Permission denied".

Here's our example again, this time using FTPStrError2:

```
        char errStr[256];

    if (FTPChdir(cip, "/pub") < 0) {
                fprintf(stderr, "Could not cd to /pub: %s.\n",
                        FTPStrError2(cip, cip->errNo, errStr,
sizeof(errStr), kErrCWDFailed));
        }
```

FTPSymlink

```
int FTPSymlink(const FTPCIPtr cip, const char *const linkpathfrom,
const char *const linkpathto);
```

A few FTP server types support a special extension which allows creation of symbolic links. This function allows you to attempt to take advantage of that functionality by creating a symbolic link on the remote host.

If the link succeeded, 0 is returned, otherwise a negative error code is returned.

FTPUmask

int FTPUmask(const FTPCIPtr cip, const char *const umsk);

This attempts to emulate the umask command that UNIX shells and programs use. This is not in the FTP standard, but many UNIX hosts implement this as a site-specific command.

Example: Set the umask for future uploads to 022.

```
err = FTPUmask(cip, "022");
```

If the umask was set, 0 is returned, otherwise a number less than zero is returned upon failure.

FTPUtime

int FTPUtime(const FTPCIPtr cip, const char *const file, time_t actime, time_t modtime, time_t crtime);

This attempts to set a remote file's timestamps, similar to to the way you would use the utime() system call on a local file. This is not in the FTP standard, but some UNIX hosts implement this as a site-specific command.

Example: Set the times for the remote file "/pub/README":

```
if (stat(localfile, &st) == 0)
        err = FTPUtime(cip, "/pub/README", st.st_atime, st.st_mtime, st.st_ctime);
```

If the times were set, 0 is returned, otherwise a number less than zero is returned upon failure.

FTPLineList functions

CopyLineList

int CopyLineList(FTPLineListPtr new, FTPLineListPtr orig);

This makes a duplicate of a FTPLineList structure. The dynamically allocated sub-structures are duplicated dynamically, so that disposing the original FTPLineList does not invalidate the copies in the new FTPLineList.

DisposeLineListContents

```
void DisposeLineListContents(FTPLineListPtr list);
```

This frees all dynamic memory allocations associated with list.

Example:

```
FTPLineList list;
int e;

InitLineList(&list);
for (e=1; (strerror(e) != NULL) && (e <= 200); e++)
        if (AddLine(&list, strerror(e)) == NULL)
                break;

/* ... do something with the list you made ... */

DisposeLineListContents(&list);
```

InitLineList

```
void InitLineList(FTPLineListPtr list);
```

Prepares a FTPLineList structure for use. The best way to use these is to simply declare a FTPLineList local variable and then pass a pointer to it. (i.e., you don't need to declare a FTPLineListPtr and then malloc space for it.)

RemoveLine

```
FTPLinePtr RemoveLine(FTPLineListPtr list, FTPLinePtr killMe);
```

This unlinks the FTPLine structure and then disposes its contents, and of course re-links the list together.

Example: Remove the second line of a list.

```
FTPLineList list;
FTPLinePtr lp;

/* ... create the list ... */
lp = list.first;
lp = lp->next;
if (lp != NULL)
        RemoveLine(lp, &list);
```

AddLine

FTPLinePtr AddLine(FTPLineListPtr list, const char *buf);

This makes a dynamically-allocated copy of buf using malloc and attaches it to the last node of the list.

This returns a pointer to the allocated FTPLine, or NULL if it could not be allocated.

Ftw functions

FtwInit

void FtwInit(FtwInfo *const ftwip);

Before calling any of the Ftw functions, you must initialize your FtwInfo structure. Once it has been initialized, you may re-use the same structure as often as you like until you call FtwDispose which releases resources associated with the structure.

FtwDispose

void FtwDispose(FtwInfo *const ftwip);

FtwDispose which releases resources associated with the FtwInfo structure.

Ftw

int Ftw(FtwInfo *const ftwip, const char *const dir, FtwProc proc);

This can be used like the C library function ftw(), if your C library has one. "Ftw" stands for "file tree walk" and like the ftw() function, this function provides you an opportunity to recurse through an entire directory on the local machine and process each file within it. Our implementation has the following features:

- It can handle pathnames of any length, so extremely complex directory trees can be processed if there is enough memory and stack space.
- It keeps only one directory open at a time. You won't run out of file descriptors!

- It's available (and works the same) on all platforms.
- You are given a copy of the stat() for each file, so your callback doesn't need to do a stat() when ftw() just did one before calling your function.

The dir parameter specifies the remote directory tree to walk.

The proc parameter is a callback to a custom function which you provide. This function will be called by Ftw for file or directory in the tree. A proc is of the FtwProc type, which is:

```
typedef int (*FtwProc)(const FtwInfoPtr ftwip);
```

When your FtwProc is called by Ftw it should inspect the FtwInfo structure, and return (-1) if directory traversal should stop, or return 0 if the traversal should continue.

Example: Traverse "/home/joeuser" on the local machine, and attempt to remove any "core" files found. This is an abridged example, but the library includes a more detailed example, ncftpftw.c, in the samples directory (which also can use FTPFtw to do remote Ftw traversals on remote FTP servers).

```
static int
MyLocalFtwProc(const FtwInfoPtr ftwip)
{
    longest_int fSize;
    time_t fTime;

    if (ftwip->curType == 'd') {
            printf("Directory: %s\n", ftwip->curPath);
    } else if (ftwip->curType == 'l') {
            printf("Symlink: %s\n", ftwip->curPath);
    } else {
            fSize = ftwip->curStat.st_size;
            fTime = ftwip->curStat.st_mtime;
            printf("File: %s (size=%lld, mtime=%u)\n",
                    ftwip->curPath, fSize, fTime);

            if (strcmp(ftwip->curFile, "core") == 0)
                    (void) unlinkk(ftwip->curPath);
    }

    return (0);     /* continue traversal */
}   /* MyFtwProc */
```

```
/* ... */
{
   FtwInfo ftwi;
   int rc;

   FtwInit(&ftwi);
   if ((rc = Ftw(&ftwi, "/home/joeuser",
MyLocalFtwProc)) != 0) {
            /* Traversal failed */
            /* ... */
   }

   (void) printf("Stats: rc=%d #dirs=%u #files=%u #links=%u
maxdepth=%u\n",
            rc, ftwi.numDirs, ftwi.numFiles, ftwi.numLinks,
ftwi.maxDepth);

   FtwDispose(&ftwi);
}
```

FtwSetBuf

void FtwSetBuf(FtwInfo *const ftwip, char *const buf, const size_t bufsize, int autogrow);

This can be used if you wish to provide your own buffer rather than allowing Ftw to dynamically allocate and grow its own buffer.

The buf parameter should be a pointer to the buffer, or NULL if you want Ftw to allocate it.

The bufsize parameter should be the size of the buffer, or the starting size of the buffer Ftw should allocate.

The autogrow parameter should be set to kFtwAutoGrow if Ftw should dynamically extend its own buffer. Set it to kFtwNoAutoGrowAndFail if Ftw should stop traversing and return an error if the buffer is completely full. Set it to kFtwNoAutoGrowButContinue if Ftw should continue traversing, and truncate pathnames to the size of the buffer. You can test for truncation by checking if buf[bufsize - 2] is not a NUL byte.

Questions & Answers

1. How do I get the library to use passive FTP?

The dataPortMode field of your FTPConnectionInfo structure may be set to one of kSendPortMode, kPassiveMode, or kFallBackToSendPortMode. If you want to try passive FTP, you can use kPassiveMode. You can also use kFallBackToSendPortMode which means the library will try passive FTP first, and if the server does not support it, it will then try regular port FTP.

2. How can I do non-blocking FTP library calls?

Technically, you can't. The library isn't coded for non-blocking I/O and it would be ugly it was. But you can have an operation timeout after a number of seconds so you don't hang forever on a slow or unresponsive server.

To do that, the xferTimeout field of your FTPConnectionInfo structure can be set to a positive integer and the library function will return an error when the timer expires.

3. How do I debug an application using the library?

The easiest is to take advantage of the debugLog field in your FTPConnectionInfo structure (i.e. set it to stdout), then look at the log. You should be able to see the whole FTP conversation, and what errors the remote server reported, if any. You can then repeat that same conversation using an FTP client to investigate possible problems with the server you are communicating with.

4. Does the library support SFTP/SSH/SSL?

No, sorry!

5. Is the library thread safe?

Currently our test suite nor any of the sample programs use multi-threading on platforms that support it. Therefore, we can't say we've explicitly tested it in a multi-threading environment. However, some daring customers are using the library in multi-threaded applications with apparent success.

The general guidelines would be:

- Only one thread should own and use a FTPConnectionInfo structure (and thus a FTP session). If you can't adhere to that, you absolutely must not allow multiple threads to call library functions using the same FTPConnectionInfo structure at the same time.
- The library relies upon standard C library functions such as malloc() and snprintf(), and will try to use reentrant versions of C library functions (such as strtok_r) if they are available. When you compile the library, you must compile it with any necessary compiler options that enable multi-threading and reentrancy (for example, -D_REENTRANT). For example, thread-safe versions of C library functions such as malloc() and snprintf() must be used!
- Library functions must be allowed to finish. Signal handlers, longjmp()ing, or killing threads which are in the middle of library functions may leave the library in an unknown state.
- You are responsible for ensuring that only one thread is accessing a particular local file at any given time.

6. How can I contact the library maintainers?

For technical support, send us mail.

7. Is SOCKS supported?

Not officially, since we do not use SOCKS, but you can try it. When the library is built, the configure script must be told to look for SOCKS, as in ./configure --enable-socks5.