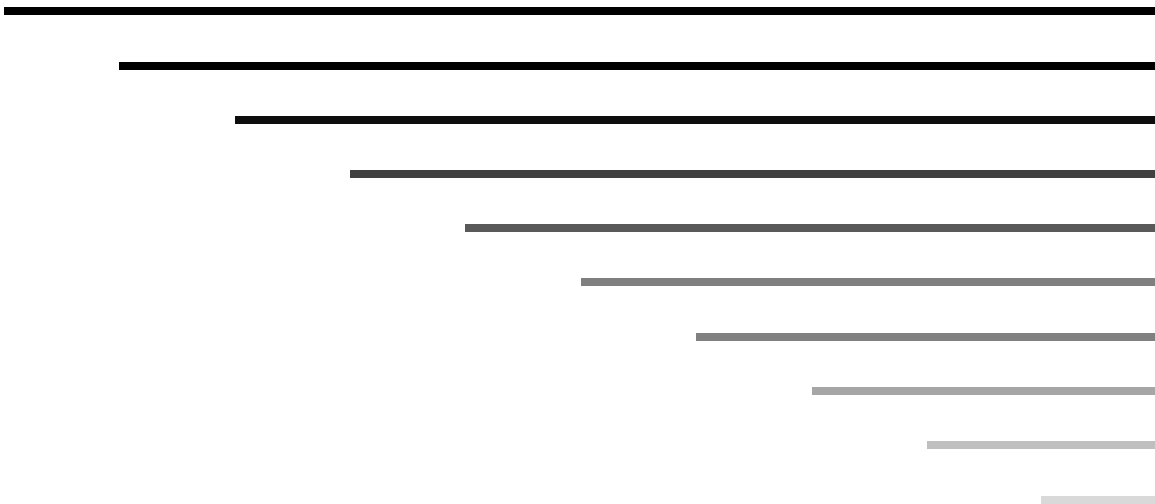


*Embedding it better...*



μTasker Document

**FTP Client**



## Table of Contents

1. Introduction.....	3
2. FTP Log-In.....	4
3. FTP Operation Modes .....	4
4. µTasker Project FTP Client Interface.....	5
4.1. Simple TCP Socket Data Mode .....	5
4.2. Buffered TCP Socket Data Mode.....	6
4.3. User Supplied TCP Socket Data Mode .....	6
5. Connecting to an FTP Server .....	8
6. Moving between Directories and Listing Content on the FTP Server .....	10
7. Creating a New Directory .....	12
8. Retrieving Data from the FTP Server.....	13
9. Sending or Appending Data to Files at the FTP Server .....	14
10. Renaming Files and Directories .....	15
11. Deleting Files and Directories .....	16
12. Conclusion.....	17

## 1. Introduction

The µTasker FTP Client is an interface that allows applications to command a TCP connection to an FTP server. Once connected to the server, involving a login procedure that the FTP client interface takes over, the application can control data transfers with the server via a second TCP data connection that is also managed by the FTP client.

Examples of the use of the TCP client interface:

- The application checks the content of a directory on an FTP server to see whether there are new or updated files available.
- The application retrieves files from an FTP server to store locally (including firmware update content) or to pass on to another interface (such as a serial interface).
- The application copies local data to a FTP, where it is stored as a file for later retrieval by the same application or another user.
- The application establishes a data connection and uses the FTP server as remote data logging storage space (in the form of a file which is either created or appended to).

The FTP client therefore needs to offer a convenient interface to the application and perform details of the FTP operation in a transparent manner so that the application can concentrate on making use of the FTP capabilities to simply achieve the advantages offered by it.

FTP operation involves the use of two TCP ports and two TCP connections when data is transferred. The first TCP connection is usually on the well-known port 21 and is known as the FTP command connection. This connection is used to log on to the FTP server and to exchange parameters as well as to command actions that subsequently require a second data connection for the data transfer. The data connection is established when data is to be transferred between the client and the server and is open only as long as the transfer takes place. A new data connection is established for each individual data transfer.

## 2. FTP Log-In

FTP servers are protected by a user login sequence consisting of a user name and a matching password. When the access is intended for anyone the FTP server will accept an anonymous login, whereby it is usual for the anonymous user to login with an email address. The use of an email address is however not usually a requirement and any password is normally accepted as long as the anonymous user name has been correctly entered. Anonymous users may have restricted access to the data on the FTP server (for example they may be able to read data but not write, modify or delete it).

The µTasker FTP client supports plain-text and anonymous login. It doesn't support security such as TLS or SSL.

## 3. FTP Operation Modes

There are two main operation modes concerning data connection. These are active and passive connection modes. The server may not support both modes and will inform the FTP client correspondingly in case it can't work in any requested mode.

- **Active** data connection mode involves the FTP server establishing the data connection when data transfer is to take place.
- **Passive** data connection mode involves the FTP client establishing the data connection when data transfer is to take place.

The data transfer content can also be of two types: **binary** or **ASCII**. Binary mode of operation allows 8 bit bytes to be sent and received. ASCII mode allows only 7 bit characters to be sent and received.

## 4. µTasker Project FTP Client Interface

Since FTP client use can vary between applications the µTasker project uses a practical user interface to work with a remote FTP server. This interface allows logging on to the server, viewing directories and files, moving around the directory structure, as well as renaming and deleting files and directories.

Files can be retrieved and their content is displayed at the terminal interface and the user can also create files and write, or append to them, using the terminal interface input.

Usually this interface can be used in general projects and the user will then only need to decide what should be done with retrieved data (for example save it to a file or pass it on to a different interface) and where data to be saved comes from (example from another input of from a local file).

The data socket used during data transfer can be of three different types. Two of these are integrated in the TCP client itself and the other one is supplied by the user. The three types are explained in more detail below and it is up to the user of the FTP client to decide which type of operation is most suitable for the project in which it is used. The mode used is defines on a project basis.

### 4.1. Simple TCP Socket Data Mode

The data socket is managed by the FTP client. It is a simple TCP socket offering the lowest memory utilisation but supports only single frame transfer (no TCP windowing), which doesn't allow highest throughput. In projects where the transfer rate is not of priority but rather simple operation with lowest memory footprint is preferred this type of socket can be used.

The socket type only affects data transmission (PUT or APPEND) whereby the user receives initially a call-back event `FTP_CLIENT_EVENT_DATA_CONNECTED` when the data connection has been established. This is followed by either `FTP_CLIENT_EVENT_PUT_CAN_START` or `FTP_CLIENT_EVENT_APPEND_CAN_START` to inform the user that one frame of data can be sent using the `fnSendTCP()` command, whereby the data socket's socket number was passed in the call-back message box along with the IP address of the FTP server.

When this frame has been successfully transmitted and acknowledged by the FTP server the user receives the call-back event `FTP_CLIENT_EVENT_DATA_SENT` and can send a further TCP frame using `fnSendTCP()` if more data is ready to be transmitted. This can continue until the user doesn't want to send any more data, in which case the user returns `APP_REQUEST_CLOSE` if the user hasn't already commanded a close of the data socket.

In this mode of operation it is up to the user to resend any data that cannot be delivered to the FTP server. In case of data loss the user receives the call-back event `FTP_CLIENT_EVENT_DATA_LOST` and must repeat the previous data (or that data plus additional data that is waiting).

When the data connection is closed the call-back event `FTP_CLIENT_EVENT_DATA_DISCONNECTED` is received.

The mode of operation is used when no extra options are defined.

## 4.2. Buffered TCP Socket Data Mode

The data socket is managed by the FTP client. It uses a buffered TCP socket and manages TCP windowing for fastest data throughput as well as repetitions in case of lost data. This mode is simplest to use but requires a TCP buffer size of

`FTP_CLIENT_TCP_BUFFER_LENGTH` for its operation, whereby the user must ensure that this buffer is not overrun – by monitoring it with `fnSendBufTCP()` and the flag `TCP_BUF_CHECK` before sending data with `fnSendBufTCP()` rather than with `fnSendTCP()`.

The user call-back events to be handled during file transmission are

`FTP_CLIENT_EVENT_DATA_CONNECTED` and `FTP_CLIENT_EVENT_DATA_SENT`, whereby this event is only received when all outstanding data has been acknowledged, thus allowing the user to return `APP_REQUEST_CLOSE` to close a connection when all data has been sent. Optimal data transmission and repetitions in case of data loss are handled by the FTP client's listener function.

This mode of operation is enabled by the define `FTP_CLIENT_BUFFERED_SOCKET_MODE`.

## 4.3. User Supplied TCP Socket Data Mode

The FTP client doesn't manage a data socket. In this case the FTP client informs the user that a data connection is required via call-back events

`FTP_CLIENT_EVENT_ACTIVE_LISTEN_DATA` or

`FTP_CLIENT_EVENT_PASSIVE_CONNECT_DATA` and it is the responsibility of the user to supply this socket. It either puts the socket to the listening mode on the port supplied in the call-back message box or else it actively establishes a TCP connection to the port on the FTP server with IP and port number supplied in the call-back message box.

An example of handling this is shown below, illustrating how the user's code handles the data connection and also retrieves details of the exact action taking place.

```
if (ptrClientMessageBox->iCallbackEvent & (FTP_CLIENT_EVENT_ACTIVE_LISTEN_DATA |
    FTP_CLIENT_EVENT_PASSIVE_CONNECT_DATA)) { // data connection request
    FTP_Client_Control_socket = ptrClientMessageBox->uControlSocket;
    // the socket used by the FTP client for control purposes
    if (ptrClientMessageBox->iCallbackEvent &
        FTP_CLIENT_EVENT_ACTIVE_LISTEN_DATA) { // we must listen on the port
        number passed so that the FTP server can
        establish its data connection with us
        iFTP_mode = FTP_DATA_LISTENER; // the data connection state
        fnListenFTPDataConnection(ptrClientMessageBox->usDataPort);
        // set the socket to listen on the port number
    }
    else { // FTP_CLIENT_EVENT_PASSIVE_CONNECT_DATA - IP address and port number of
        data connection known - we can establish a connection with it
        iFTP_mode = FTP_DATA_CLIENT; // mark that a passive FTP data connection is
        being established
        fnEstablishFTPDataConnection(ptrClientMessageBox->ucIP_data_address,
            ptrClientMessageBox->usDataPort);
    }
    if (ptrClientMessageBox->iCallbackEvent & FTP_CLIENT_EVENT_FLAG_ASCII_MODE) {
        iFTP_mode |= FTP_DATA_ASCII;
    }
    if (ptrClientMessageBox->iCallbackEvent & FTP_CLIENT_EVENT_FLAG_PUT_DIRECTION) {
```

```
        iFTP_mode |= FTP_DATA_PUT_MODE; // we are allowed to send but not receive
    }
    else {
        iFTP_mode |= FTP_DATA_GET_MODE; // we are allowed to receive but not send
    }
    if (ptrClientMessageBox->iCallbackEvent & FTP_CLIENT_EVENT_FLAG_LISTING) {
        iFTP_mode |= FTP_DATA_LISTING; // not file content
    }
    else if (temp_pars->temp_parameters.ucFTPmode & _FTP_GET_ESCAPING) {
        // set present escape sequencing flags when getting data (not listing)
        iFTP_mode |= FTP_DATA_ESCAPE_SEQUENCING_GET;
    }
    if (temp_pars->temp_parameters.ucFTPmode & _FTP_PUT_ESCAPING) {
        iFTP_mode |= FTP_DATA_ESCAPE_SEQUENCING_PUT;
    }
    return 0;
}
```

The details of the TCP socket actually used are application specific, as is how the connections are established.

This type of socket is useful in applications which can share an existing socket for the data transfer which may be used for other purposes when the FTP client is not in use. The user socket has to handle the data but has full control over this operation. When data transmission to the FTP server has completed the user closes this data socket to terminate the PUT/APPEND transaction.

This mode of operation is enabled by the define `FTP_CLIENT_EXTERN_DATA_SOCKET`.

## 5. Connecting to an FTP Server

The connection to a remote FTP server is controlled by the FTP client module. The user starts the connection process – including logging on to the server – by calling the function

```
extern USOCKET fnFTP_client_connect(
    unsigned char ucIP_address[IPV4_LENGTH],
    unsigned short ucPortNumber,
    unsigned short usFTPTimeout,
    int (*user_callback)(TCP_CLIENT_MESSAGE_BOX *));
```

The IP address of the server, the TCP port number to be used for the session, the connection timeout plus a user call-back routine are passed to the.

The function returns the socket number of the FTP client's control socket if the process can start or an error if not connection attempt can be made.

During the login process the user call-back function is used when the FTP client needs to know the user name and user password that is required by the server. The call-back function must therefore return pointers to when requested for this information as show by the example below.

```
extern int fnFTP_client_user_callback_handler(
    TCP_CLIENT_MESSAGE_BOX *ptrClientMessageBox)
{
    switch (ptrClientMessageBox->iCallbackEvent) { // FTP client events
    case FTP_CLIENT_EVENT_LOGGED_IN: // FTP connection now established
        fnDebugMsg("FTP connection established\r\n");
        break;
    case FTP_CLIENT_EVENT_REQUEST_FTP_USER_NAME: // return pointer to user name
        ptrClientMessageBox->ptrData =
            (unsigned char *)temp_pars->temp_parameters.cFTPUserName;
        break;
    case FTP_CLIENT_EVENT_REQUEST_FTP_USER_PASSWORD: // return pointer to password
        ptrClientMessageBox->ptrData =
            (unsigned char *)temp_pars->temp_parameters.cFTPUserPass;
        break;
    case FTP_CLIENT_USER_NAME_ERROR:
        fnDebugMsg("FTP User failed\r\n");
        break;
    case FTP_CLIENT_USER_PASS_ERROR:
        fnDebugMsg("FTP Pass failed\r\n");
        break;
    case FTP_CLIENT_EVENT_LOGGED_FAILED: // login was
        fnDebugMsg("FTP-bad_login\r\n");
        break;
    case FTP_CLIENT_EVENT_CONNECTION_CLOSED: // connection closed or aborted
        fnDebugMsg("FTP connection terminated\r\n");
        break;
    }
}
```

*Note that returning a zero pointer, or empty string, in response to the user name request will cause the FTP client to attempt anonymous login.*

Using the command line interface (serial, Telnet or USB) a connection can be made to the remote FTP server by entering

```
ftp_con
```

When the connection is successful the message

```
FTP connection established
```

is seen. The connection will timeout after a period of non-use, or can be terminated by entering

```
ftp_dis
```

In each case the termination is seen by

```
FTP connection terminated
```

In case of login errors corresponding error messages are displays, such as

```
FTP-bad_login
```

The command used to disconnect the FTP session is

```
extern int fnFTP_client_disconnect(void);
```

This function returns a positive value equivalent to the length of data send by the socket to command the termination or `SOCKET_STATE_INVALID` when there is no session to terminate.

## 6. Moving between Directories and Listing Content on the FTP Server

The function used for all directory type command is

```
extern int fnFTP_client_dir(CHAR *ptrPath, int iAction);
```

To perform a directory listing the command

```
fnFTP_client_dir(0, FTP_DIR_LIST);
```

is executed. This causes a listing to be started from the present location on the FTP server. After a log-on, this location will be the user's root directory on the FTP server.

To start a directory listing at a different location a path string can be passed, such as

```
fnFTP_client_dir("dir1/dir2", FTP_DIR_LIST);
```

The maximum length for path names is configured by the define `MAX_FTP_CLIENT_PATH_LENGTH`, for example

```
#define MAX_FTP_CLIENT_PATH_LENGTH 64
```

The FTP client module will then negotiate the listing with the FTP server and set the transfer mode to ASCII. The call-back is used to request the application whether the data connection should be performed in active or passive mode and then the data connection will also be opened accordingly.

Once the data connection has been established the FTP server sends the directory content listing which is received on a TCP frame base via the call-back function. The µTasker project sends this to the command line interface so that the user can view it.

The call-back function case entries responsible for this are shown below:

```
...
    case FTP_CLIENT_EVENT_ACTIVE_PASSIVE_LIST: // the FTP client is asking whether
                                                we want to transfer data in active or
                                                passive mode
        return (!(temp_pars->temp_parameters.usServers & PASSIVE_MODE_FTP_CLIENT));
        // return the FTP mode setting

    case FTP_CLIENT_EVENT_LISTING_DATA: // receiving listing data
        fnWrite(DebugHandle, ptrClientMessageBox->ptrData,
                ptrClientMessageBox->usDataLength); // write to debug output
        break;

    case FTP_CLIENT_EVENT_LISTING_DATA_COMPLETE: // listing terminated and the data
                                                connection was terminated by the FTP server
        fnDebugMsg("FTP directory listing complete\r\n");
        break;

    case FTP_CLIENT_EVENT_DATA_CONNECTION_FAILED: // data connection failed
        fnDebugMsg("Data connection failed\r\n");
        break;
...

```

In this example the received data is simply sent to the debug output. Usually some flow control will be used in addition in case the debug output cannot handle amount of data without buffer overrun. See the µTasker project code for the complete solution.

A directory listing performed using the command line menu is shown below:

```
ftp_dir

#drwxr-x---  2 0      33          4096 Jun 19  2007 atd
drwxr-x---   2 0      757         4096 Dec 09 21:05 backup
drwxr-x---   2 757    33          4096 Nov 20  2007 files
drwxr-x---  19 757    33          4096 Dec 08 16:37 html
drwxr-x---   3 0      757         4096 Dec 13 01:07 log
drwxrwx---   2 757    33          4096 Dec 13 21:11 phptmp
drwxrwx---   2 0      757         4096 Jun 19  2007 restore
FTP directory listing complete

#ftp_dir html/dir_test

#-rw-r--r--   1 757    757         17575 Dec 13 09:25 1.txt
-rw-r--r--   1 757    757           60 Dec 08 15:26 2.txt
drwxr-xr-x    2 757    757         4096 Dec 13 09:53 test
-rw-r--r--   1 757    757           10 Dec 10 00:31 test10.txt
-rw-r--r--   1 757    757         6090 Dec 12 21:25 test3.txt
-rw-r--r--   1 757    757        33991 Dec 12 21:36 test5.txt
-rw-r--r--   1 757    757        34055 Dec 12 21:54 test6.txt
-rw-r--r--   1 757    757           26 Dec 08 19:13 test7.bin
-rw-r--r--   1 757    757           25 Dec 08 19:12 test8.txt
-rw-r--r--   1 757    757           15 Dec 08 19:12 test9.bin
-rw-r--r--   1 757    757           786 Dec 11 22:21 testp2.txt
-rw-r--r--   1 757    757          1179 Dec 11 22:42 testp3.txt
-rw-r--r--   1 757    757          2629 Dec 11 22:51 testp4.txt
-rw-r--r--   1 757    757        20835 Dec 11 22:40 testput.txt
FTP directory listing complete
```

When work will be performed in a directory other than the user's root directory on the remote FTP server it is often convenient to move to that location rather than working with relative path strings. The following shows moving to the new location via the command line interface:

```
ftp_path html/test_dir
#New path set
```

This is commanded at the FTP client interface using

```
fnFTP_client_dir("dir1/dir2", FTP_DIR_SET_PATH);
```

Unlike the directory listing command the FTP client doesn't need to open a data connection but instead can negotiate the operation entirely using the existing FTP command connection.

If the new path could not be set or some other error occurs the FTP client notifies the user via error events. The command line interface displays the error number and also the error text that the FTP server sent so that the reason for the operation failure is clear. For example:

```
ftp_path bad_dir
#FTP ERROR:[0204] 550 Failed to change directory.
```

The call-back interface code responsible for the error output and the event handling cases used by the path command are shown below.

```
...
    case FTP_CLIENT_EVENT_LOCATION_SET: // requested path successfully set
        fnDebugMsg("New path set\r\n");
        break;
...
```

```
if (ptrClientMessageBox->iCallbackEvent & FTP_CLIENT_ERROR_FLAG) {
    // error event code
    fnDebugMsg("FTP ERROR: [");
    fnDebugHex(ptrClientMessageBox->iCallbackEvent, 2);
    fnDebugMsg("] ");
    fnWrite(DebugHandle, ptrClientMessageBox->ptrData,
            ptrClientMessageBox->usDataLength);
    // display exact error sent by FTP server as text
    return 0;
}
```

Note that the general error handling is recognised by the error flag in the state value and is positioned before the other event handling.

## 7. Creating a New Directory

Sometimes it may be desirable to create a new directory at the FTP server where new files will be stored.

The FTP client interface command to create a new empty directory at the FTP server is

```
fnFTP_client_dir("DIRECTORY_NAME", (FTP_DIR_MAKE_DIR));
```

The directory name string can also be a path with new directory name.

If the directory creation is successful the call-back event `FTP_CLIENT_EVENT_DIR_CREATED` is received.

The command line input is

```
ftp_mkdir test_dir
```

*New directories can only be created when the user has the necessary rights at the FTP server to do this.*

## 8. Retrieving Data from the FTP Server

The FTP GET command is used by the FTP control connection to start the file content retrieval process. Transfers of content can be performed in ASCII or binary mode. ASCII mode is only suitable for files that contain 7-bit ASCII content.

The FTP client interface command to perform a GET of a file from the FTP server is

```
fnFTP_client_transfer("FILE_NAME",  
                      (FTP_DO_GET | FTP_TRANSFER_BINARY));
```

or

```
fnFTP_client_transfer("FILE_NAME",  
                      (FTP_DO_GET | FTP_TRANSFER_ASCII));
```

If the binary/ascii mode option is not set binary is the default used.

The file name string can also be a path with file name.

The command line input is

```
ftp_get test_dir/file1.bin
```

or

```
ftp_get_a test_dir/file2.txt
```

The first starts the retrieval of a file in binary mode and the second in ascii mode. The path is optional in case the file is not located at the present directory location on the remote FTP server.

Once the GET command has been executed the FTP data connection is established (either in active or passive mode). Following the successful connection the FTP server starts sending the file's content and this is received via the data connection's TCP socket.

Each received TCP frame content is passed to the user via the call-back and event `FTP_CLIENT_EVENT_GET_DATA` along with a pointer to the received data and its length.

As in the case of the directory listing, the µTasker reference project sends all received file content to the debug output so that file content listings can be made. To aid in controlling the reception of large file content the µTasker reference project allows the scrolling to be paused by hitting the 'p' key – this will halt reception from the FTP server by pausing the debug output. Hitting the 'p' key a second time will allow the reception to continue. If it is required to stop the reception of a large file before it has completed the CTRL + C sequence can be used to abort the transfer, thus closing the FTP data connection, but retaining the FTP control connection.

The data connection is terminated by the FTP server once it has sent the complete content and this event represents that the file has been completely received.

One successful termination the call-back event `FTP_CLIENT_EVENT_GET_DATA_COMPLETE` is received.

## 9. Sending or Appending Data to Files at the FTP Server

The FTP put command is the inverse of the GET command, whereby data is sent to the FTP server and stored to the file that is created. The file transfer can take place in ascii or binary mode.

The APPEND command is very similar to the PUT command but, instead of creating a new file, or first deleting an existing file, it opens an existing file and appends new data to it.

The FTP client interface command to perform a PUT to a file on the FTP server is

```
fnFTP_client_transfer("FILE_NAME",  
                      (FTP_DO_PUT | FTP_TRANSFER_BINARY));
```

or

```
fnFTP_client_transfer("FILE_NAME",  
                      (FTP_DO_PUT | FTP_TRANSFER_ASCII));
```

If the binary/ascii mode option is not set binary is the default used.

The file name string can also be a path with file name.

The command line input is

```
ftp_put test_dir/file1.bin
```

or

```
ftp_put_a test_dir/file2.txt
```

The first starts by creating the file at the FTP server for subsequent data transfer to it in binary mode and the second in ascii mode. The path is optional in case the file is not located at the present directory location on the remote FTP server.

The FTP client interface command to perform an APPEND to a file on the FTP server is

```
fnFTP_client_transfer("FILE_NAME",  
                      (FTP_DO_APPEND | FTP_TRANSFER_BINARY));
```

or

```
fnFTP_client_transfer("FILE_NAME",  
                      (FTP_DO_APPEND | FTP_TRANSFER_ASCII));
```

If the binary/ascii mode option is not set binary is the default used.

The file name string can also be a path with file name.

The command line input is

```
ftp_app test_dir/file1.bin
```

or

```
ftp_app_a test_dir/file2.txt
```

The first starts by opening the file at the FTP server for subsequent data append to it in binary mode and the second in ascii mode. The path is optional in case the file is not located at the present directory location on the remote FTP server.

Once the PUT or APPEND command has been executed the FTP data connection is established (either in active or passive mode). Following the successful connection file data content can be sent to the FTP server.

The µTasker reference project sends all debug input data to the file at the FTP server. This means that all key entry will be written to the file. Since the FTP server doesn't know when the file has completed it is necessary for the FTP client to close the data connection at the end of the session. The end of the session can be commanded by CTRL + C .

*Files can only be created and written to when the user has the necessary rights at the FTP server to do this.*

Data is sent to the FTP server using the data socket type as specified in the FTP client project configuration (see the chapter on the FTP client configuration for details of the possible types). The µTasker project supports both modes based on an integrated data socket in the FTP client to aid in deciding which best suits the user (with and without `FTP_CLIENT_BUFFERED_SOCKET_MODE` enabled). The mode based on a shared data socket supplied by the user is not included in the µTasker project but a reference interface is shown in the chapter discussing this mode.

## 10. Renaming Files and Directories

The FTP client interface command to rename a file or a directory at the FTP server is

```
fnFTP_client_dir("NAME1 NAME2", (FTP_DIR_RENAME));
```

The name string, containing both the original name and the new name – separated by a space, can also be paths to the file or directory.

If the rename is successful the call-back event `FTP_CLIENT_EVENT_RENAMED` is received.

The command line input is

```
ftp_ren file1.txt file2.txt
```

*Files and directories can only be renamed at the FTP server if the user has permission to do this.*

## 11. Deleting Files and Directories

The FTP client interface command to delete a file at the FTP server is

```
fnFTP_client_dir("FILE_NAME", (FTP_DIR_DELETE));
```

The file name string can also be a path to the file.

The command line input is

```
ftp_del file1.txt
```

The FTP client interface command to delete an empty directory at the FTP server is

```
fnFTP_client_dir("DIRECTORY_NAME", (FTP_DIR_REMOVE_DIR));
```

The directory name string can also be a path to the directory.

If the directory removal was successful the call-back event

`FTP_CLIENT_EVENT_DIR_DELETED` is received.

The command line input is

```
ftp_remove test_dir
```

FTP servers will generally not allow a directory to be deleted if it still contains files.

*Files and directories can only be deleted from the FTP server if the user has permission to do this.*

## 12. Conclusion

The μTasker FTP client allows simple user interaction with a remote FTP server as demonstrated by the μTasker reference project. The interface gives applications powerful methods to create directories and files at a remote FTP server to save data collected locally into and also to retrieve files, for example to obtain new configuration data or new firmware.

The data connection interface is flexible, allowing a simple integrated socket method for the simplest solution with least memory overhead, an integrated buffered TCP socket for fastest throughput and simplest user interface or a user-supplied data socket, which may be shared with other applications uses to optimise TCP socket resources.

### Modifications:

V0.00 14.12.2011: Initial draft

V0.01 20.12.2011: Command set completed

V0.02 23.12.2011: First complete version

V1.00 24.12.2011: First release