

FTP

Copyright © 1997-2025 Ericsson AB. All Rights Reserved.
FTP 1.0.5
December 10, 2025

Copyright © 1997-2025 Ericsson AB. All Rights Reserved.
Licensed under the Apache License, Version 2.0 (the "License"); you may not use this file except in compliance with the License. You may obtain a copy of the License at http://www.apache.org/licenses/LICENSE-2.0 Unless required by applicable law or agreed to in writing, software distributed under the License is distributed on an "AS IS" BASIS, WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied. See the License for the specific language governing permissions and limitations under the License. Ericsson AB. All Rights Reserved..

December 10, 2025

1 FTP User's Guide

The FTP application provides an FTP client.

1.1 Introduction

1.1.1 Purpose

An FTP client.

1.1.2 Prerequisites

It is assumed that the reader is familiar with the Erlang programming language, concepts of OTP, and has a basic understanding of the FTP protocol.

1.2 FTP Client

1.2.1 Getting Started

FTP clients are considered to be rather temporary. Thus, they are only started and stopped during runtime and cannot be started at application startup. The FTP client API is designed to allow some functions to return intermediate results. This implies that only the process that started the FTP client can access it with preserved sane semantics. If the process that started the FTP session dies, the FTP client process terminates.

The client supports IPv6 as long as the underlying mechanisms also do so.

The following is a simple example of an FTP session, where the user `guest` with password `password` logs on to the remote host `erlang.org`:

```
1> ftp:start().
ok
2> {ok, Pid} = ftp:start_service([{host, "erlang.org"}]).
{ok,<0.22.0>}
3> ftp:user(Pid, "guest", "password").
ok
4> ftp:pwd(Pid).
{ok, "/home/guest"}
5> ftp:cd(Pid, "appl/examples").
ok
6> ftp:lpwd(Pid).
{ok, "/home/fred"}.
7> ftp:lcd(Pid, "/home/eproj/examples").
ok
8> ftp:recv(Pid, "appl.erl").
ok
9> ftp:stop_service(Pid).
ok
10> ftp:stop().
ok
```

The file `appl.erl` is transferred from the remote to the local host. When the session is opened, the current directory at the remote host is `/home/guest`, and `/home/fred` at the local host. Before transferring the file, the current

local directory is changed to `/home/eproj/examples`, and the remote directory is set to `/home/guest/appl/examples`.

2 Reference Manual

An FTP client.

ftp

Erlang module

This module implements a client for file transfer according to a subset of the File Transfer Protocol (FTP), see **RFC 959**.

The FTP client always tries to use passive FTP mode and only resort to active FTP mode if this fails. This default behavior can be changed by start option mode.

An FTP client can be started in two ways. One is using the service_start function, the other is to start it directly as a standalone process using function open.

For a simple example of an FTP session, see FTP User's Guide.

In addition to the ordinary functions for receiving and sending files (see `recv/2`, `recv/3`, `send/2`, and `send/3`) there are functions for receiving remote files as binaries (see `recv_bin/2`) and for sending binaries to be stored as remote files (see `send_bin/3`).

A set of functions is provided for sending and receiving contiguous parts of a file to be stored in a remote file. For send, see `send_chunk_start/2`, `send_chunk/2`, and `send_chunk_end/1`. For receive, see `recv_chunk_start/2` and `recv_chunk/`).

The return values of the following functions depend much on the implementation of the FTP server at the remote host. In particular, the results from `ls` and `nlist` varies. Often real errors are not reported as errors by `ls`, even if, for example, a file or directory does not exist. `nlist` is usually more strict, but some implementations have the peculiar behaviour of responding with an error if the request is a listing of the contents of a directory that exists but is empty.

FTP CLIENT SERVICE START/STOP

The FTP client can be started and stopped dynamically in runtime by calling the `ftp` application API `ftp:start_service(ServiceConfig)` and `ftp:stop_service(Pid)`.

The available configuration options are as follows:

{host, Host}

Host = `string() | ip_address()`

{port, Port}

Port = `integer() > 0`

Default is `21`.

{mode, Mode}

Mode = `active | passive`

Default is `passive`.

{verbose, Verbose}

Verbose = `boolean()`

Determines if the FTP communication is to be verbose or not.

Default is `false`.

{debug, Debug}

Debug = `trace | debug | disable`

Debugging using the dbg toolkit.

Default is `disable`.

{ipfamily, IpFamily}

IpFamily = `inet | inet6 | inet6fb4`

With `inet6fb4` the client behaves as before, that is, tries to use IPv6, and only if that does not work it uses IPv4).

Default is `inet` (IPv4).

{timeout, Timeout}

Timeout = `non_neg_integer()`

Connection time-out.

Default is `60000` (milliseconds).

{dtimeout, DTimeout}

DTimeout = `non_neg_integer() | infinity`

Data connect time-out. The time the client waits for the server to connect to the data socket.

Default is `infinity`.

{progress, Progress}

Progress = `ignore | {CBModule, CBFunction, InitProgress}`

`CBModule = atom()`, `CBFunction = atom()`

`InitProgress = term()`

Default is `ignore`.

Option `progress` is intended to be used by applications that want to create some type of progress report, such as a progress bar in a GUI. Default for the progress option is `ignore`, that is, the option is not used. When the progress option is specified, the following happens when `ftp:send/[3,4]` or `ftp:recv/[3,4]` are called:

- Before a file is transferred, the following call is made to indicate the start of the file transfer and how large the file is. The return value of the callback function is to be a new value for the `UserProgressTerm` that will be used as input the next time the callback function is called.

  ```
  CBModule:CBFunction(InitProgress, File, {file_size, FileSize})
  ```

- Every time a chunk of bytes is transferred the following call is made:

  ```
  CBModule:CBFunction(UserProgressTerm, File, {transfer_size, TransferSize})
  ```

- At the end of the file the following call is made to indicate the end of the transfer:

  ```
  CBModule:CBFunction(UserProgressTerm, File, {transfer_size, 0})
  ```

The callback function is to be defined as follows:

```
CBModule:CBFunction(UserProgressTerm, File, Size) -> UserProgressTerm

CBModule = CBFunction = atom()

UserProgressTerm = term()

File = string()

Size = {transfer_size, integer()} | {file_size, integer()} | {file_size, unknown}
```

For remote files, `ftp` cannot determine the file size in a platform independent way. In this case the size becomes `unknown` and it is left to the application to determine the size.

Note:

The callback is made by a middleman process, hence the file transfer is not affected by the code in the progress callback function. If the callback crashes, this is detected by the FTP connection process, which then prints an info-report and goes on as if the progress option was set to `ignore`.

The file transfer type is set to the default of the FTP server when the session is opened. This is usually ASCCI mode.

The current local working directory (compare `lpwd/1`) is set to the value reported by `file:get_cwd/1`, the wanted local directory.

The return value `Pid` is used as a reference to the newly created FTP client in all other functions, and they are to be called by the process that created the connection. The FTP client process monitors the process that created it and terminates if that process terminates.

DATA TYPES

The following type definitions are used by more than one function in the FTP client API:

`pid()` = identifier of an FTP connection

`string()` = list of ASCII characters

`shortage_reason()` = `etnospc | epnospc`

`restriction_reason()` = `epath | efnamena | elogin | enotbinary` - all restrictions are not always relevant to all functions

`common_reason()` = `econn | eclosed | term()` - some explanation of what went wrong

Exports

```
account(Pid, Account) -> ok | {error, Reason}
```

Types:

```
Pid = pid()
Account = string()
Reason = eacct | common_reason()
```

Sets the account for an operation, if needed.

```
append(Pid, LocalFile) ->
append(Pid, LocalFile, RemoteFile) -> ok | {error, Reason}
```

Types:

```
Pid = pid()
LocalFile = RemoteFile = string()
Reason = epath | elogin | etnospc | epnospc | efnamena | common_reason
```

Transfers the file `LocalFile` to the remote server. If `RemoteFile` is specified, the name of the remote file that the file is appended to is set to `RemoteFile`, otherwise to `LocalFile`. If the file does not exists, it is created.

```
append_bin(Pid, Bin, RemoteFile) -> ok | {error, Reason}
```

Types:

```
Pid = pid()
Bin = binary()
```

RemoteFile = string()
Reason = restriction_reason()| shortage_reason() | common_reason()

Transfers the binary `Bin` to the remote server and appends it to the file `RemoteFile`. If the file does not exist, it is created.

`append_chunk(Pid, Bin) -> ok | {error, Reason}`

Types:

Pid = pid()
Bin = binary()
Reason = echunk | restriction_reason() | common_reason()

Transfers the chunk `Bin` to the remote server, which appends it to the file specified in the call to `append_chunk_start/2`.

For some errors, for example, file system full, it is necessary to call `append_chunk_end` to get the proper reason.

`append_chunk_start(Pid, File) -> ok | {error, Reason}`

Types:

Pid = pid()
File = string()
Reason = restriction_reason() | common_reason()

Starts the transfer of chunks for appending to the file `File` at the remote server. If the file does not exist, it is created.

`append_chunk_end(Pid) -> ok | {error, Reason}`

Types:

Pid = pid()
Reason = echunk | restriction_reason() | shortage_reason()

Stops transfer of chunks for appending to the remote server. The file at the remote server, specified in the call to `append_chunk_start/2`, is closed by the server.

`cd(Pid, Dir) -> ok | {error, Reason}`

Types:

Pid = pid()
Dir = string()
Reason = restriction_reason() | common_reason()

Changes the working directory at the remote server to `Dir`.

`close(Pid) -> ok`

Types:

Pid = pid()

Ends an FTP session, created using function open.

`delete(Pid, File) -> ok | {error, Reason}`

Types:

Pid = pid()

File = string()
Reason = restriction_reason() | common_reason()

Deletes the file `File` at the remote server.

`formaterror(Tag) -> string()`

Types:

Tag = {error, atom()} | atom()

Given an error return value `{error, AtomReason}`, this function returns a readable string describing the error.

`lcd(Pid, Dir) -> ok | {error, Reason}`

Types:

Pid = pid()
Dir = string()
Reason = restriction_reason()

Changes the working directory to `Dir` for the local client.

`lpwd(Pid) -> {ok, Dir}`

Types:

Pid = pid()

Returns the current working directory at the local client.

`ls(Pid) ->`
`ls(Pid, Pathname) -> {ok, Listing} | {error, Reason}`

Types:

Pid = pid()
Pathname = string()
Listing = string()
Reason = restriction_reason() | common_reason()

Returns a list of files in long format.

`Pathname` can be a directory, a group of files, or a file. The `Pathname` string can contain wildcards.

`ls/1` implies the current remote directory of the user.

The format of `Listing` depends on the operating system. On UNIX, it is typically produced from the output of the `ls -l` shell command.

`mkdir(Pid, Dir) -> ok | {error, Reason}`

Types:

Pid = pid()
Dir = string()
Reason = restriction_reason() | common_reason()

Creates the directory `Dir` at the remote server.

```
nlist(Pid) ->
nlist(Pid, Pathname) -> {ok, Listing} | {error, Reason}
```

Types:

```
Pid = pid()
Pathname = string()
Listing = string()
Reason = restriction_reason() | common_reason()
```

Returns a list of files in short format.

`Pathname` can be a directory, a group of files, or a file. The `Pathname` string can contain wildcards.

`nlist/1` implies the current remote directory of the user.

The format of `Listing` is a stream of filenames where each filename is separated by <CRLF> or <NL>. Contrary to function `ls`, the purpose of `nlist` is to enable a program to process filename information automatically.

```
open(Host) -> {ok, Pid} | {error, Reason}
open(Host, Opts) -> {ok, Pid} | {error, Reason}
```

Types:

```
Host = string() | ip_address()
Opts = options()
options() = [option()]
option() = start_option() | open_option()
start_option() = {verbose, verbose()} | {debug, debug()}
verbose() = boolean() (default is false)
debug() = disable | debug | trace (default is disable)
open_option() = {ipfamily, ipfamily()} | {port, port()} | {mode, mode()}
| {tls, tls_options()} | {timeout, timeout()} | {dtimeout, dtimeout()}
| {progress, progress() | {sock_ctrl, sock_opts()} | {sock_data_act,
sock_opts()} | {sock_data_pass, sock_opts()} }
ipfamily() = inet | inet6 | inet6fb4 (default is inet)
port() = integer() > 0 (default is 21)
mode() = active | passive (default is passive)
tls_options() = [ssl:tls_option()]
sock_opts() = [gen_tcp:option() except for ipv6_v6only, active, packet,
mode, packet_size and header
timeout() = integer() > 0 (default is 60000 milliseconds)
dtimeout() = integer() > 0 | infinity (default is infinity)
pogress() = ignore | {module(), function(), initial_data()} (default is
ignore)
module() = atom()
function() = atom()
initial_data() = term()
Reason = ehost | term()
```

Starts a standalone FTP client process (without the `ftp` service framework) and opens a session with the FTP server at `Host`.

If option `{tls, tls_options()}` is present, the FTP session is transported over `tls` (`ftps`, see **RFC 4217**). The list `tls_options()` can be empty. The function `ssl:connect/3` is used for securing both the control connection and the data sessions.

The options `sock_ctrl`, `sock_data_act` and `sock_data_pass` passes options down to the underlying transport layer (tcp). The default value for `sock_ctrl` is `[]`. Both `sock_data_act` and `sock_data_pass` uses the value of `sock_ctrl` as default value.

A session opened in this way is closed using function close.

```
pwd(Pid) -> {ok, Dir} | {error, Reason}
```

Types:

Pid = pid()
Reason = restriction_reason() | common_reason()

Returns the current working directory at the remote server.

```
recv(Pid, RemoteFile) ->
recv(Pid, RemoteFile, LocalFile) -> ok | {error, Reason}
```

Types:

Pid = pid()
RemoteFile = LocalFile = string()
Reason = restriction_reason() | common_reason() | file_write_error_reason()
file_write_error_reason() = see file:write/2

Transfers the file `RemoteFile` from the remote server to the file system of the local client. If `LocalFile` is specified, the local file will be `LocalFile`, otherwise `RemoteFile`.

If the file write fails (for example, `enospc`), the command is aborted and `{error, file_write_error_reason()}` is returned. However, the file is **not** removed.

```
recv_bin(Pid, RemoteFile) -> {ok, Bin} | {error, Reason}
```

Types:

Pid = pid()
Bin = binary()
RemoteFile = string()
Reason = restriction_reason() | common_reason()

Transfers the file `RemoteFile` from the remote server and receives it as a binary.

```
recv_chunk_start(Pid, RemoteFile) -> ok | {error, Reason}
```

Types:

Pid = pid()
RemoteFile = string()
Reason = restriction_reason() | common_reason()

Starts transfer of the file `RemoteFile` from the remote server.

```
recv_chunk(Pid) -> ok | {ok, Bin} | {error, Reason}
```

Types:

Pid = pid()
Bin = binary()
Reason = restriction_reason() | common_reason()

Receives a chunk of the remote file (`RemoteFile` of `recv_chunk_start`). The return values have the following meaning:

- `ok` = the transfer is complete.
- `{ok, Bin}` = just another chunk of the file.
- `{error, Reason}` = transfer failed.

```
rename(Pid, Old, New) -> ok | {error, Reason}
```

Types:

Pid = pid()
CurrFile = NewFile = string()
Reason = restriction_reason() | common_reason()

Renames `Old` to `New` at the remote server.

```
rmdir(Pid, Dir) -> ok | {error, Reason}
```

Types:

Pid = pid()
Dir = string()
Reason = restriction_reason() | common_reason()

Removes directory `Dir` at the remote server.

```
send(Pid, LocalFile) ->
send(Pid, LocalFile, RemoteFile) -> ok | {error, Reason}
```

Types:

Pid = pid()
LocalFile = RemoteFile = string()
Reason = restriction_reason() | common_reason() | shortage_reason()

Transfers the file `LocalFile` to the remote server. If `RemoteFile` is specified, the name of the remote file is set to `RemoteFile`, otherwise to `LocalFile`.

```
send_bin(Pid, Bin, RemoteFile) -> ok | {error, Reason}
```

Types:

Pid = pid()
Bin = binary()
RemoteFile = string()
Reason = restriction_reason() | common_reason() | shortage_reason()

Transfers the binary `Bin` into the file `RemoteFile` at the remote server.

```
send_chunk(Pid, Bin) -> ok | {error, Reason}
```

Types:

Pid = pid()

Bin = binary()
Reason = echunk | restriction_reason() | common_reason()

Transfers the chunk `Bin` to the remote server, which writes it into the file specified in the call to `send_chunk_start/2`.

For some errors, for example, file system full, it is necessary to to call `send_chunk_end` to get the proper reason.

`send_chunk_start(Pid, File) -> ok | {error, Reason}`

Types:

Pid = pid()
File = string()
Reason = restriction_reason() | common_reason()

Starts transfer of chunks into the file `File` at the remote server.

`send_chunk_end(Pid) -> ok | {error, Reason}`

Types:

Pid = pid()
Reason = restriction_reason() | common_reason() | shortage_reason()

Stops transfer of chunks to the remote server. The file at the remote server, specified in the call to `send_chunk_start/2` is closed by the server.

`start_service(ServiceConfig) -> {ok, Pid} | {error, Reason}`

Types:

ServiceConfig = [{Option, Value}]
Option = property()
Value = term()

Dynamically starts an `FTP` session after the `ftp` application has been started.

> **Note:**
>
> As long as the `ftp` application is operational, the FTP sessions are supervised and can be soft code upgraded.

`stop_service(Reference) -> ok | {error, Reason}`

Types:

Reference = pid() | term() - service-specified reference
Reason = term()

Stops a started FTP session.

`type(Pid, Type) -> ok | {error, Reason}`

Types:

Pid = pid()
Type = ascii | binary
Reason = etype | restriction_reason() | common_reason()

Sets the file transfer type to `ascii` or `binary`. When an FTP session is opened, the default transfer type of the server is used, most often `ascii`, which is default according to **RFC 959**.

```
user(Pid, User, Password) -> ok | {error, Reason}
```

Types:

```
Pid = pid()
User = Password = string()
Reason = euser | common_reason()
```

Performs login of `User` with `Password`.

```
user(Pid, User, Password, Account) -> ok | {error, Reason}
```

Types:

```
Pid = pid()
User = Password = string()
Reason = euser | common_reason()
```

Performs login of `User` with `Password` to the account specified by `Account`.

```
quote(Pid, Command) -> [FTPLine]
```

Types:

```
Pid = pid()
Command = string()
FTPLine = string(
```

Note:

The telnet end of line characters, from the FTP protocol definition, CRLF, for example, "\\r\\n" has been removed.

Sends an arbitrary FTP command and returns verbatim a list of the lines sent back by the FTP server. This function is intended to give application accesses to FTP commands that are server-specific or that cannot be provided by this FTP client.

Note:

FTP commands requiring a data connection cannot be successfully issued with this function.

ERRORS

The possible error reasons and the corresponding diagnostic strings returned by `formaterror/1` are as follows:

`echunk`

Synchronization error during chunk sending according to one of the following:

- A call is made to `send_chunk/2` or `send_chunk_end/1` before a call to `send_chunk_start/2`.
- A call has been made to another transfer function during chunk sending, that is, before a call to `send_chunk_end/1`.

`eclosed`

The session is closed.

`econn`
: Connection to the remote server is prematurely closed.

`ehost`
: Host is not found, FTP server is not found, or connection is rejected by FTP server.

`elogin`
: User is not logged in.

`enotbinary`
: Term is not a binary.

`epath`
: No such file or directory, or directory already exists, or permission denied.

`etype`
: No such type.

`euser`
: Invalid username or password.

`etnospc`
: Insufficient storage space in system [452].

`epnospc`
: Exceeded storage allocation (for current directory or dataset) [552].

`efnamena`
: Filename not allowed [553].

SEE ALSO

file(3) filename(3) and J. Postel and J. Reynolds: File Transfer Protocol (**RFC 959**).