


# TFTP

Copyright © 1997-2026 Ericsson AB. All Rights Reserved.
TFTP 1.1.1.1
November 20, 2026

**Copyright © 1997-2026 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..

**November 20, 2026**

# 1 TFTP User's Guide

The TFTP application provides a TFTP client and server.

## 1.1 Introduction

### 1.1.1 Purpose

The Trivial File Transfer Protocol or TFTP is a very simple protocol used to transfer files.

It has been implemented on top of the User Datagram protocol (UDP) so it may be used to move files between machines on different networks implementing UDP. It is designed to be small and easy to implement. Therefore, it lacks most of the features of a regular FTP. The only thing it can do is read and write files (or mail) from/to a remote server. It cannot list directories, and currently has no provisions for user authentication.

The `tftp` application implements the following IETF standards:

- **RFC 1350**, The TFTP Protocol (revision 2)
- **RFC 2347**, TFTP Option Extension
- **RFC 2348**, TFTP Blocksize Option
- **RFC 2349**, TFTP Timeout Interval and Transfer Size Options

The only feature that not is implemented is the `netascii` transfer mode.

### 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 TFTP protocol.

### 1.1.3 Security Considerations

As stated in (**RFC 1350**) be aware that "Since TFTP includes no login or access control mechanisms, care must be taken in the rights granted to a TFTP server process so as not to violate the security of the server hosts file system. TFTP is often installed with controls such that only files that have public read access are available via TFTP and writing files via TFTP is disallowed."

This essentially means that any machine on the network that can reach the TFTP server is able to read and write, without authentication, any file on the machine that runs the TFTP server, that the user (or group) that runs the TFTP server (in this case the Erlang VM) is allowed to read or write. The machine configuration has to be prepared for that.

**Warning:**

The default behavior mentioned above is in general very risky, and as a remedy, this TFTP application's default callback `tftp_file` implements an initial state option `{root_dir,Dir}` that restricts the callback's file accesses to `Dir` and subdirectories. It is recommended to use that option when starting start this TFTP server.

# 1.2 Getting Started

## 1.2.1 General Information

The start/1 function starts a daemon process listening for UDP packets on a port. When it receives a request for read or write, it spawns a temporary server process handling the transfer.

On the client side, function read_file/3 and write_file/3 spawn a temporary client process establishing contact with a TFTP daemon and perform the file transfer.

`tftp` uses a callback module to handle the file transfer. Two such callback modules are provided, `tftp_binary` and `tftp_file`. See read_file/3 and write_file/3 for details. You can also implement your own callback modules, see CALLBACK FUNCTIONS. A callback module provided by the user is registered using option `callback`, see DATA TYPES.

## 1.2.2 Using the TFTP client and server

This is a simple example of starting the TFTP server and reading the content of a sample file using the TFTP client.

**Step 1.** Create a sample file to be used for the transfer:

```
$ echo "Erlang/OTP 21" > /tmp/file.txt
```

**Step 2.** Start the TFTP server:

```
1> Callback = {callback,{"",tftp_file,[{root_dir,"/tmp"}]}}.
2> {ok, Pid} = tftp:start([{port, 19999}, Callback]).
{ok,<0.65.0>}
```

**Step 3.** Start the TFTP client (in another shell):

```
1> tftp:read_file("file.txt", binary, [{port, 19999}]).
{ok,<<"Erlang/OTP 21\n">>}
```

# 2 Reference Manual

The TFTP application.

# tftp

Erlang module

Interface module for the `tftp` application.

## DATA TYPES

`ServiceConfig = Options`

`Options = [option()]`

Most of the options are common for both the client and the server side, but some of them differs a little. The available `option()`s are as follows:

`{debug, Level}`

`Level = none | error | warning | brief | normal | verbose | all`

Controls the level of debug printouts. Default is `none`.

`{host, Host}`

`Host = hostname()`, see inet(3).

The name or IP address of the host where the TFTP daemon resides. This option is only used by the client.

`{port, Port}`

`Port = int()`

The TFTP port where the daemon listens. Defaults is the standardized number 69. On the server side, it can sometimes make sense to set it to 0, meaning that the daemon just picks a free port (which one is returned by function `info/1`).

If a socket is connected already, option `{udp, [{fd, integer()}]}` can be used to pass the open file descriptor to `gen_udp`. This can be automated by using a command-line argument stating the prebound file descriptor number. For example, if the port is 69 and file descriptor 22 is opened by `setuid_socket_wrap`, the command-line argument "-tftpd_69 22" triggers the prebound file descriptor 22 to be used instead of opening port 69. The UDP option `{udp, [{fd, 22}]}` is automatically added. See `init:get_argument/` about command-line arguments and `gen_udp:open/2` about UDP options.

`{port_policy, Policy}`

`Policy = random | Port | {range, MinPort, MaxPort}`

`Port = MinPort = MaxPort = int()`

Policy for the selection of the temporary port that is used by the server/client during the file transfer. Default is `random`, which is the standardized policy. With this policy a randomized free port is used. A single port or a range of ports can be useful if the protocol passes through a firewall.

`{udp, Options}`

`Options = [Opt]`, see gen_udp:open/2.

`{use_tsize, Bool}`

`Bool = bool()`

Flag for automated use of option `tsize`. With this set to `true`, the `write_file/3` client determines the filesize and sends it to the server as the standardized `tsize` option. A `read_file/3` client acquires only a filesize from the server by sending a zero `tsize`.

`{max_tsize, MaxTsize}`

```
MaxTsize = int() | infinity
```

Threshold for the maximal filesize in bytes. The transfer is aborted if the limit is exceeded. Default is `infinity`.

`{max_conn, MaxConn}`

```
MaxConn = int() | infinity
```

Threshold for the maximal number of active connections. The daemon rejects the setup of new connections if the limit is exceeded. Default is `infinity`.

`{TftpKey, TftpVal}`

```
TftpKey = string()
TftpVal = string()
```

Name and value of a TFTP option.

`{reject, Feature}`

```
Feature = Mode | TftpKey
 Mode = read | write
 TftpKey = string()
```

Controls which features to reject. This is mostly useful for the server as it can restrict the use of certain TFTP options or read/write access.

`{callback, {RegExp, Module, InitialState}}`

```
RegExp = string()
Module = atom()
InitialState = term()
```

Registration of a callback module. When a file is to be transferred, its local filename is matched to the regular expressions of the registered callbacks. The first matching callback is used during the transfer. See read_file/3 and write_file/3.

The callback module must implement the `tftp` behavior, see CALLBACK FUNCTIONS.

At the end of the list of callbacks there are always the default callbacks `tftp_file` and `tftp_binary` with the `RegExp` `""` and `InitialState` `[]`.

The `InitialState` should be an option list, and the empty list should be accepted by any callback module. The `tftp_file` callback module accepts an `InitialState = [{root_dir, Dir}]` that restrict local file operations to files in `Dir` and subdirectories. All file names received in protocol requests, relative or absolute, are regarded as relative to this directory.

> **Warning:**
>
> The default callback module configuration allows access to any file on any local filesystem that is readable or writable by the user running the Erlang VM. This can be a security vulnerability. It is therefore recommended to explicitly configure the `tftp_file` callback module to use the `root_dir` option.

`{logger, Module}`

```
Module = module()
```

Callback module for customized logging of errors, warnings, and info messages. The callback module must implement the `tftp_logger` behavior. The default module is `tftp_logger`.

```
{max_retries, MaxRetries}
```

```
MaxRetries = int()
```

Threshold for the maximal number of retries. By default the server/client tries to resend a message up to five times when the time-out expires.

## Exports

```
change_config(daemons, Options) -> [{Pid, Result}]
```

Types:

**Options = [option()]**
**Pid = pid()**
**Result = ok | {error, Reason}**
**Reason = term()**

Changes configuration for all TFTP daemon processes.

```
change_config(servers, Options) -> [{Pid, Result}]
```

Types:

**Options = [option()]**
**Pid = pid()**
**Result = ok | {error, Reason}**
**Reason = term()**

Changes configuration for all TFTP server processes.

```
change_config(Pid, Options) -> Result
```

Types:

**Pid = pid()**
**Options = [option()]**
**Result = ok | {error, Reason}**
**Reason = term()**

Changes configuration for a TFTP daemon, server, or client process.

```
info(daemons) -> [{Pid, Options}]
```

Types:

**Pid = [pid()]**
**Options = [option()]**
**Reason = term()**

Returns information about all TFTP daemon processes.

```
info(servers) -> [{Pid, Options}]
```

Types:

**Pid = [pid()]**
**Options = [option()]**
**Reason = term()**

Returns information about all TFTP server processes.

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

Types:

**Options = [option()]**
**Reason = term()**

Returns information about a TFTP daemon, server, or client process.

```
read_file(RemoteFilename, LocalFilename, Options) -> {ok, LastCallbackState}
| {error, Reason}
```

Types:

**RemoteFilename = string()**
**LocalFilename = binary | string()**
**Options = [option()]**
**LastCallbackState = term()**
**Reason = term()**

Reads a (virtual) file `RemoteFilename` from a TFTP server.

If `LocalFilename` is the atom `binary`, `tftp_binary` is used as callback module. It concatenates all transferred blocks and returns them as one single binary in `LastCallbackState`.

If `LocalFilename` is a string and there are no registered callback modules, `tftp_file` is used as callback module. It writes each transferred block to the file named `LocalFilename` and returns the number of transferred bytes in `LastCallbackState`.

If `LocalFilename` is a string and there are registered callback modules, `LocalFilename` is tested against the regexps of these and the callback module corresponding to the first match is used, or an error tuple is returned if no matching regexp is found.

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

Types:

**Options = [option()]**
**Pid = pid()**
**Reason = term()**

Starts a daemon process listening for UDP packets on a port. When it receives a request for read or write, it spawns a temporary server process handling the actual transfer of the (virtual) file.

The request filename is matched against the regexps of the registered callback modules, and the first match selects the callback to handle the request.

If there are no registered callback modules, `tftp_file` is used, with the initial state `[]`.

**Warning:**

The default callback module configuration allows access to any file on any local filesystem that is readable or writable by the user running the Erlang VM. This can be a security vulnerability. See the **{callback,_}** option at the start of this module reference for a remedy.

```
write_file(RemoteFilename, LocalFilename, Options) -> {ok, LastCallbackState}
| {error, Reason}
```

Types:

**`RemoteFilename = string()`**
**`LocalFilename = binary() | string()`**
**`Options = [option()]`**
**`LastCallbackState = term()`**
**`Reason = term()`**

Writes a (virtual) file `RemoteFilename` to a TFTP server.

If `LocalFilename` is a binary, `tftp_binary` is used as callback module. The binary is transferred block by block and the number of transferred bytes is returned in `LastCallbackState`.

If `LocalFilename` is a string and there are no registered callback modules, `tftp_file` is used as callback module. It reads the file named `LocalFilename` block by block and returns the number of transferred bytes in `LastCallbackState`.

If `LocalFilename` is a string and there are registered callback modules, `LocalFilename` is tested against the regexps of these and the callback module corresponding to the first match is used, or an error tuple is returned if no matching regexp is found.

## CALLBACK FUNCTIONS

A `tftp` callback module is to be implemented as a `tftp` behavior and export the functions listed in the following.

On the server side, the callback interaction starts with a call to `open/5` with the registered initial callback state. `open/5` is expected to open the (virtual) file. Then either function `read/1` or `write/2` is invoked repeatedly, once per transferred block. At each function call, the state returned from the previous call is obtained. When the last block is encountered, function `read/1` or `write/2` is expected to close the (virtual) file and return its last state. Function `abort/3` is only used in error situations. Function `prepare/5` is not used on the server side.

On the client side, the callback interaction is the same, but it starts and ends a bit differently. It starts with a call to `prepare/5` with the same arguments as `open/5` takes. `prepare/5` is expected to validate the TFTP options suggested by the user and to return the subset of them that it accepts. Then the options are sent to the server, which performs the same TFTP option negotiation procedure. The options that are accepted by the server are forwarded to function `open/5` on the client side. On the client side, function `open/5` must accept all option as-is or reject the transfer. Then the callback interaction follows the same pattern as described for the server side. When the last block is encountered in `read/1` or `write/2`, the returned state is forwarded to the user and returned from `read_file`/3 or `write_file/3`.

If a callback (performing the file access in the TFTP server) takes too long time (more than the double TFTP time-out), the server aborts the connection and sends an error reply to the client. This implies that the server releases resources attached to the connection faster than before. The server simply assumes that the client has given up.

If the TFTP server receives yet another request from the same client (same host and port) while it already has an active connection to the client, it ignores the new request if the request is equal to the first one (same filename and options). This implies that the (new) client will be served by the already ongoing connection on the server side. By not setting up yet another connection, in parallel with the ongoing one, the server consumes less resources.

## Exports

```
Module:abort(Code, Text, State) -> ok
```

Types:

**`Code = undef | enoent | eacces | enospc`**

```
    | badop | eexist | baduser | badopt
    | int()
  Text = string()
  State = term()
```

Invoked when the file transfer is aborted.

The callback function is expected to clean up its used resources after the aborted file transfer, such as closing open file descriptors and so on. The function is not invoked if any of the other callback functions returns an error, as it is expected that they already have cleaned up the necessary resources. However, it is invoked if the functions fail (crash).

```
Module:open(Peer, Access, Filename, Mode, SuggestedOptions, State) -> {ok,
AcceptedOptions, NewState} | {error, {Code, Text}}
```

Types:

```
  Peer = {PeerType, PeerHost, PeerPort}
  PeerType = inet | inet6
  PeerHost = ip_address()
  PeerPort = integer()
  Access = read | write
  Filename = string()
  Mode = string()
  SuggestedOptions = AcceptedOptions = [{Key, Value}]
   Key = Value = string()
  State = InitialState | term()
   InitialState = [] | [{root_dir, string()}]
  NewState = term()
  Code = undef | enoent | eacces | enospc
    | badop | eexist | baduser | badopt
    | int()
  Text = string()
```

Opens a file for read or write access.

On the client side, where the `open/5` call has been preceded by a call to `prepare/5`, all options must be accepted or rejected.

On the server side, where there is no preceding `prepare/5` call, no new options can be added, but those present in `SuggestedOptions` can be omitted or replaced with new values in `AcceptedOptions`.

```
Module:prepare(Peer, Access, Filename, Mode, SuggestedOptions, InitialState)
-> {ok, AcceptedOptions, NewState} | {error, {Code, Text}}
```

Types:

```
  Peer = {PeerType, PeerHost, PeerPort}
  PeerType = inet | inet6
  PeerHost = ip_address()
  PeerPort = integer()
  Access = read | write
  Filename = string()
  Mode = string()
```

```
SuggestedOptions = AcceptedOptions = [{Key, Value}]
 Key = Value = string()
InitialState = [] | [{root_dir, string()}]
NewState = term()
Code = undef | enoent | eacces | enospc
  | badop | eexist | baduser | badopt
  | int()
Text = string()
```

Prepares to open a file on the client side.

No new options can be added, but those present in `SuggestedOptions` can be omitted or replaced with new values in `AcceptedOptions`.

This is followed by a call to `open/4` before any read/write access is performed. `AcceptedOptions` is sent to the server, which replies with the options that it accepts. These are then forwarded to `open/4` as `SuggestedOptions`.

```
Module:read(State) -> {more, Bin, NewState} | {last, Bin, FileSize} | {error,
{Code, Text}}
```

Types:

```
State = NewState = term()
Bin = binary()
FileSize = int()
Code = undef | enoent | eacces | enospc
  | badop | eexist | baduser | badopt
  | int()
Text = string()
```

Reads a chunk from the file.

The callback function is expected to close the file when the last file chunk is encountered. When an error is encountered, the callback function is expected to clean up after the aborted file transfer, such as closing open file descriptors, and so on. In both cases there will be no more calls to any of the callback functions.

```
Module:write(Bin, State) -> {more, NewState} | {last, FileSize} | {error,
{Code, Text}}
```

Types:

```
Bin = binary()
State = NewState = term()
FileSize = int()
Code = undef | enoent | eacces | enospc
  | badop | eexist | baduser | badopt
  | int()
Text = string()
```

Writes a chunk to the file.

The callback function is expected to close the file when the last file chunk is encountered. When an error is encountered, the callback function is expected to clean up after the aborted file transfer, such as closing open file descriptors, and so on. In both cases there will be no more calls to any of the callback functions.

# tftp_logger

Erlang module

A `tftp_logger` callback module is to be implemented as a `tftp_logger` behavior and export the following functions:

## Exports

`Module:error_msg(Format, Data) -> ok`

Types:

**`Format = string()`**
**`Data = [term()]`**
**`Reason = term()`**

Logs an error message. See `error_logger:error_msg/2` for details.

`Module:info_msg(Format, Data) -> ok`

Types:

**`Format = string()`**
**`Data = [term()]`**
**`Reason = term()`**

Logs an info message. See `error_logger:info_msg/2` for details.

`Module:warning_msg(Format, Data) -> ok`

Types:

**`Format = string()`**
**`Data = [term()]`**
**`Reason = term()`**

Logs a warning message. See `error_logger:warning_msg/2` for details.