SSH

SSH 3.1
August 24, 2015

August 24, 2015

1 SSH User's Guide

The *SSH* application implements the SSH (Secure Shell) protocol and provides an SFTP (Secret File Transfer Protocol) client and server.

1.1 Introduction

1.1.1 Purpose

Secure Shell (SSH) is a protocol for secure remote login and other secure network services over an insecure network. SSH provides a single, full-duplex, byte-oriented connection between client and server. The protocol also provides privacy, integrity, server authentication and man-in-the-middle protection.

The Erlang SSH application is an implementation of the SSH protocol in Erlang which offers API functions to write customized SSH clients and servers as well as making the Erlang shell available via SSH. Also included in the SSH application are an SFTP (SSH File Transfer Protocol) client *ssh_sftp* and server *ssh_sftpd*.

1.1.2 Prerequisites

It is assumed that the reader is familiar with the concepts of *OTP* and has a basic understanding of **public keys**.

1.2 Secure Shell (SSH)

1.2.1 SSH Protocol Overview

Conceptually the SSH protocol can be partitioned into four layers:

Figure 2.1: SSH Protocol Architecture

Transport Protocol

The SSH Transport Protocol is a secure, low level transport. It provides strong encryption, cryptographic host authentication and integrity protection. Currently, only a minimum of MAC- (message authentication code, a short piece of information used to authenticate a message) and encryption algorithms are supported see *ssh(3)*

Authentication Protocol

The SSH authentication protocol is a general-purpose user authentication protocol run over the SSH transport protocol. Erlang SSH supports user authentication using public key technology (RSA and DSA, X509-certificates are currently not supported). It is also possible to use a so called keyboard interactive authentication. This method is suitable for interactive authentication methods that do not need any special software support on the client side. Instead, all authentication data should be entered via the keyboard. It is also possible to use a pure password based authentication scheme, note that in this case the the plain text password will be encrypted before sent over the network. There are several configuration options for authentication handling available in *ssh:connect/[3,4]* and *ssh:daemon/[2,3]* It is also possible to customize the public key handling by implementing the behaviours *ssh_client_key_api* and *ssh_server_key_api*

Connection Protocol

The SSH Connection Protocol provides application-support services over the transport pipe, such as channel multiplexing, flow control, remote program execution, signal propagation, connection forwarding, etc. Functions for handling the SSH Connection Protocol can be found in the module *ssh_connection*.

Channels

All terminal sessions, forwarded connections etc., are channels. Multiple channels are multiplexed into a single connection, and all channels are flow-controlled. Typically an SSH client will open a channel, send data/commands, receive data/"control information" and when it is done close the channel. The *ssh_channel* behaviour makes it easy to write your own SSH client/server processes that use flow control. It handles generic parts of SSH channel management and lets you focus on the application logic.

Channels comes in three flavors

- *Subsystem* - named services that can be run as part of an SSH server such as SFTP *ssh_sftpd*, that is built in to the SSH daemon (server) by default but may be disabled. The Erlang SSH daemon may be configured to run any Erlang implemented SSH subsystem.
- *Shell* - interactive shell. By default the Erlang daemon will run the Erlang shell. It is possible to customize the shell by providing your own read-eval-print loop. It is also possible, but much more work, to provide your own CLI (Command Line Interface) implementation.
- *Exec* - one-time remote execution of commands. See *ssh_connection:exec/4*

Channels are flow controlled. No data may be sent to a channel peer until a message is received to indicate that window space is available. The 'initial window size' specifies how many bytes of channel data that can be sent to the channel peer without adjusting the window.

For more detailed information about the SSH protocol, see the following RFCs:

- **RFC 4250** - Protocol Assigned Numbers.
- **RFC 4251** - Protocol Architecture.
- **RFC 4252** - Authentication Protocol.
- **RFC 4253** - Transport Layer Protocol.
- **RFC 4254** - Connection Protocol.
- **RFC 4255** - Key Fingerprints.
- **RFC 4344** - Transport Layer Encryption Modes.
- **RFC 4716** - Public Key File Format.

1.3 Getting started

1.3.1 General information

The examples in the following sections use the utility function *ssh:start/0* that starts all needed applications (crypto, public_key and ssh). All examples are run in an Erlang shell, or in a bash shell using openssh to illustrate how the erlang ssh application can be used. The examples are run as the user otptest on a local network where the user is authorized to login in over ssh to the host "tarlop". If nothing else is stated it is persumed that the otptest user has an entry in tarlop's authorized_keys file (may log in via ssh without entering a password). Also tarlop is a known host in the user otptest's known_hosts file so that host verification can be done without user interaction.

1.3.2 Using the Erlang SSH Terminal Client

The user otptest, that has bash as default shell, uses the ssh:shell/1 client to connect to the openssh daemon running on a host called tarlop. Note that currently this client is very simple and you should not be expected to be as fancy as the openssh client.

```
1>  ssh:start().
ok
2> {ok, S} = ssh:shell("tarlop").
>pwd
/home/otptest
>exit
logout
3>
```

1.3.3 Running an Erlang SSH Daemon

The option system_dir must be a directory containing a host key file and it defaults to /etc/ssh. For details see section Configuration Files in *ssh(6)*.

> **Note:**
>
> Normally the /etc/ssh directory is only readable by root.

The option user_dir defaults to the users ~/.ssh directory

In the following example we generate new keys and host keys as to be able to run the example without having root privilages

```
$bash> ssh-keygen -t rsa -f /tmp/ssh_daemon/ssh_host_rsa_key
[...]
$bash> ssh-keygen -t rsa -f /tmp/otptest_user/.ssh/id_rsa
[...]
```

Create the file /tmp/otptest_user/.ssh/authorized_keys and add the content of /tmp/otptest_user/.ssh/id_rsa.pub Now we can do

```
1> ssh:start().
ok
2> {ok, Sshd} = ssh:daemon(8989,  [{system_dir, "/tmp/ssh_daemon"},
{user_dir, "/tmp/otptest_user/.ssh"}]).
{ok,<0.54.0>}
3>
```

Use the openssh client from a shell to connect to the Erlang ssh daemon.

```
$bash> ssh tarlop -p 8989  -i /tmp/otptest_user/.ssh/id_rsa\
```

```
       -o UserKnownHostsFile=/tmp/otptest_user/.ssh/known_hosts
The authenticity of host 'tarlop' can't be established.
RSA key fingerprint is 14:81:80:50:b1:1f:57:dd:93:a8:2d:2f:dd:90:ae:a8.
Are you sure you want to continue connecting (yes/no)? yes
Warning: Permanently added 'tarlop' (RSA) to the list of known hosts.
Eshell V5.10  (abort with ^G)
1>
```

There are two ways of shutting down an SSH daemon

1: Stops the listener, but leaves existing connections started by the listener up and running.

```
3> ssh:stop_listener(Sshd).
ok
4>
```

2: Stops the listener and all connections started by the listener.

```
3> ssh:stop_daemon(Sshd)
ok
4>
```

1.3.4 One Time Execution

In the following example the Erlang shell is the client process that receives the channel replies.

> **Note:**
>
> If you run this example in your environment you may get fewer or more messages back as this depends on the OS and shell on the machine running the ssh daemon. See also *ssh_connection:exec/4*

```
1>  ssh:start().
ok
2> {ok, ConnectionRef} = ssh:connect("tarlop", 22, []).
{ok,<0.57.0>}
3>{ok, ChannelId} =  ssh_connection:session_channel(ConnectionRef, infinity).
{ok,0}
4> success = ssh_connection:exec(ConnectionRef, ChannelId, "pwd", infinity).
5>  flush().
Shell got {ssh_cm,<0.57.0>,{data,0,0,<<"/home/otptest\n">>}}
Shell got {ssh_cm,<0.57.0>,{eof,0}}
Shell got {ssh_cm,<0.57.0>,{exit_status,0,0}}
Shell got {ssh_cm,<0.57.0>,{closed,0}}
ok
6>
```

Note only the channel is closed the connection is still up and can handle other channels

```
     6> {ok, NewChannelId} =  ssh_connection:session_channel(ConnectionRef, infinity).
       {ok,1}
...
```

1.3.5 SFTP (SSH File Transport Protocol) server

```
     1> ssh:start().
     ok
     2> ssh:daemon(8989,  [{system_dir, "/tmp/ssh_daemon"},
        {user_dir, "/tmp/otptest_user/.ssh"},
        {subsystems, [ssh_sftpd:subsystem_spec([{cwd, "/tmp/sftp/example"}])]}]).
     {ok,<0.54.0>}
     3>
```

Run the openssh sftp client

```
     $bash> sftp -oPort=8989 -o IdentityFile=/tmp/otptest_user/.ssh/id_rsa\
            -o UserKnownHostsFile=/tmp/otptest_user/.ssh/known_hosts tarlop
     Connecting to tarlop...
     sftp> pwd
     Remote working directory: /tmp/sftp/example
     sftp>
```

1.3.6 SFTP (SSH File Transport Protocol) client

```
     1> ssh:start().
     ok
     2> {ok, ChannelPid, Connection} = ssh_sftp:start_channel("tarlop", []).
     {ok,<0.57.0>,<0.51.0>}
     3>  ssh_sftp:read_file(ChannelPid, "/home/otptest/test.txt").
     {ok,<<"This is a test file\n">>}
```

1.3.7 Creating a subsystem

A very small SSH subsystem that echos N bytes could be implemented like this. See also *ssh_channel(3)*

```
-module(ssh_echo_server).
-behaviour(ssh_subsystem).
-record(state, {
   n,
   id,
   cm
  }).
-export([init/1, handle_msg/2, handle_ssh_msg/2, terminate/2]).

init([N]) ->
    {ok, #state{n = N}}.
```

```
handle_msg({ssh_channel_up, ChannelId, ConnectionManager}, State) ->
    {ok, State#state{id = ChannelId,
       cm = ConnectionManager}}.

handle_ssh_msg({ssh_cm, CM, {data, ChannelId, 0, Data}}, #state{n = N} = State) ->
    M = N - size(Data),
    case M > 0 of
 true ->
    ssh_connection:send(CM, ChannelId, Data),
    {ok, State#state{n = M}};
 false ->
    <<SendData:N/binary, _/binary>> = Data,
           ssh_connection:send(CM, ChannelId, SendData),
           ssh_connection:send_eof(CM, ChannelId),
    {stop, ChannelId, State}
    end;
handle_ssh_msg({ssh_cm, _ConnectionManager,
  {data, _ChannelId, 1, Data}}, State) ->
    error_logger:format(standard_error, " ~p~n", [binary_to_list(Data)]),
    {ok, State};

handle_ssh_msg({ssh_cm, _ConnectionManager, {eof, _ChannelId}}, State) ->
    {ok, State};

handle_ssh_msg({ssh_cm, _, {signal, _, _}}, State) ->
    %% Ignore signals according to RFC 4254 section 6.9.
    {ok, State};

handle_ssh_msg({ssh_cm, _, {exit_signal, ChannelId, _, _Error, _}},
        State) ->
    {stop, ChannelId,  State};

handle_ssh_msg({ssh_cm, _, {exit_status, ChannelId, _Status}}, State) ->
    {stop, ChannelId, State}.

terminate(_Reason, _State) ->
    ok.
```

And run like this on the host tarlop with the keys generated in section 3.3

```
1> ssh:start().
ok
2>  ssh:daemon(8989, [{system_dir, "/tmp/ssh_daemon"},
{user_dir, "/tmp/otptest_user/.ssh"}
{subsystems, [{"echo_n", {ssh_echo_server, [10]}}]}]).
{ok,<0.54.0>}
3>
```

```
1> ssh:start().
ok
2>{ok, ConnectionRef} = ssh:connect("tarlop", 8989, [{user_dir, "/tmp/otptest_user/.ssh"}]).
 {ok,<0.57.0>}
3>{ok, ChannelId} = ssh_connection:session_channel(ConnectionRef, infinity).
4> success = ssh_connection:subsystem(ConnectionRef, ChannelId, "echo_n", infinity).
5> ok = ssh_connection:send(ConnectionRef, ChannelId, "0123456789", infinity).
6> flush().
{ssh_msg, <0.57.0>, {data, 0, 1, "0123456789"}}
{ssh_msg, <0.57.0>, {eof, 0}}
```

```
{ssh_msg, <0.57.0>, {closed, 0}}
7> {error, closed} = ssh_connection:send(ConnectionRef, ChannelId, "10", infinity).
```

2 Reference Manual

The SSH application is an erlang implementation of the secure shell protocol (SSH) as defined by RFC 4250 - 4254

SSH

Application

DEPENDENCIES

The ssh application uses the Erlang applications public_key and crypto to handle public keys and encryption, hence these applications needs to be loaded for the ssh application to work. In an embedded environment that means they need to be started with application:start/[1,2] before the ssh application is started.

CONFIGURATION

The ssh application does not currently have an application specific configuration file as described in application(3), however it will by default use the following configuration files from openssh: known_hosts, authorized_keys, authorized_keys2, id_dsa and id_rsa, ssh_host_dsa_key and ssh_host_rsa_key. By default Erlang SSH will look for id_dsa, id_rsa, known_hosts and authorized_keys in ~/.ssh, and the host key files in /etc/ssh . These locations may be changed by the options user_dir and system_dir. Public key handling may also be customized by providing a callback module implementing the behaviors *ssh_client_key_api* and *ssh_server_key_api*.

PUBLIC KEYS

id_dsa and id_rsa are the users private key files, note that the public key is part of the private key so the ssh application will not use the id_<*>.pub files. These are for the users convenience when he/she needs to convey their public key.

KNOW HOSTS

The known_hosts file contains a list of approved servers and their public keys. Once a server is listed, it can be verified without user interaction.

AUTHORIZED KEYS

The authorized key file keeps track of the user's authorized public keys. The most common use of this file is to let users log in without entering their password which is supported by the Erlang SSH daemon.

HOST KEYS

Currently rsa and dsa host keys are supported and are expected to be found in files named ssh_host_rsa_key and ssh_host_dsa_key.

SEE ALSO

application(3)

ssh

Erlang module

Interface module for the SSH application.

SSH

- SSH requires the crypto and public_key applications.
- Supported SSH version is 2.0
- Supported MAC algorithms: hmac-sha2-256 and hmac-sha1
- Supported encryption algorithms: aes128-ctr, aes128-cb and 3des-cbc
- Supports unicode filenames if the emulator and the underlaying OS supports it. See the DESCRIPTION section in *file* for information about this subject
- Supports unicode in shell and cli

DATA TYPES

Type definitions that are used more than once in this module and/or abstractions to indicate the intended use of the data type:

```
boolean() = true | false
```

```
string() = [byte()]
```

```
ssh_daemon_ref() - opaque to the user returned by ssh:daemon/[1,2,3]
```

```
ssh_connection_ref() - opaque to the user returned by ssh:connect/3
```

```
ip_address() - inet::ip_address()
```

```
subsystem_spec() = {subsystem_name(), {channel_callback(), channel_init_args()}}
```

```
subsystem_name() = string()
```

```
channel_callback() = atom() - Name of the erlang module implementing the subsystem using the ssh_channel behavior see ssh_channel(3)
```

```
channel_init_args() = list()
```

Exports

```
close(ConnectionRef) -> ok
```

Types:

ConnectionRef = ssh_connection_ref()

Closes an SSH connection.

```
connect(Host, Port, Options) ->
connect(Host, Port, Options, Timeout) -> {ok, ssh_connection_ref()} | {error, Reason}
```

Types:

Host = string()

Port = integer()

The default is 22, the assigned well known port number for SSH.

Options = [{Option, Value}]

Timeout = infinity | integer(milliseconds)

Negotiation timeout, for connection timeout use the option {connect_timeout, timeout()}.

Connects to an SSH server. No channel is started. This is done by calling *ssh_connection:session_channel/[2, 4]*.

Options are:

{inet, inet | inet6}

IP version to use.

{user_dir, string()}

Sets the user directory i.e. the directory containing ssh configuration files for the user such as known_hosts, id_rsa, id_dsa and authorized_key. Defaults to the directory normally referred to as ~/.ssh

{dsa_pass_phrase, string()}

If the user dsa key is protected by a passphrase it can be supplied with this option.

{rsa_pass_phrase, string()}

If the user rsa key is protected by a passphrase it can be supplied with this option.

{silently_accept_hosts, boolean()}

When true hosts are added to the file known_hosts without asking the user. Defaults to false.

{user_interaction, boolean()}

If false disables the client to connect to the server if any user interaction is needed such as accepting that the server will be added to the known_hosts file or supplying a password. Defaults to true. Even if user interaction is allowed it can be suppressed by other options such as silently_accept_hosts and password. Do note that it may not always be desirable to use those options from a security point of view.

{public_key_alg, 'ssh-rsa' | 'ssh-dss'}

Sets the preferred public key algorithm to use for user authentication. If the the preferred algorithm fails for some reason, the other algorithm is tried. The default is to try 'ssh-rsa' first.

{pref_public_key_algs, list()}

List of public key algorithms to try to use, 'ssh-rsa' and 'ssh-dss' available. Will override {public_key_alg, 'ssh-rsa' | 'ssh-dss'}

{connect_timeout, timeout()}

Sets a timeout on the transport layer connection. Defaults to infinity.

{user, string()}

Provides a user name. If this option is not given, ssh reads from the environment (LOGNAME or USER on unix, USERNAME on Windows).

{password, string()}

Provide a password for password authentication. If this option is not given, the user will be asked for a password if the password authentication method is attempted.

{key_cb, atom()}

Module implementing the behaviour *ssh_client_key_api*. Can be used to customize the handling of public keys.

{quiet_mode, atom() = boolean()}

If true, the client will not print out anything on authorization.

`{fd, file_descriptor()}`

Allow an existing file descriptor to be used (simply passed on to the transport protocol).

`{rekey_limit, integer()}`

Provide, in bytes, when rekeying should be initiated, defaults to one time each GB and one time per hour.

`{idle_time, integer()}`

Sets a timeout on connection when no channels are active, default is infinity

`connection_info(ConnectionRef, [Option]) ->[{Option, Value}]`

Types:

```
Option = client_version | server_version | user | peer | sockname
Value = [option_value()]
option_value() = {{Major::integer(), Minor::integer()},
VersionString::string()} | User::string() | Peer::{inet:hostname(),
{inet::ip_adress(), inet::port_number()}} | Sockname::{inet::ip_adress(),
inet::port_number()} ()
```

Retrieves information about a connection.

`daemon(Port) ->`

`daemon(Port, Options) ->`

`daemon(HostAddress, Port, Options) -> {ok, ssh_daemon_ref()} | {error, atom()}`

Types:

```
Port = integer()
HostAddress = ip_address() | any
Options = [{Option, Value}]
Option = atom()
Value = term()
```

Starts a server listening for SSH connections on the given port.

Options are:

`{inet, inet | inet6}`

IP version to use when the host address is specified as `any`.

`{subsystems, [subsystem_spec()]`

Provides specifications for handling of subsystems. The "sftp" subsystem spec can be retrieved by calling ssh_sftpd:subsystem_spec/1. If the subsystems option in not present the value of `[ssh_sftpd:subsystem_spec([])]` will be used. It is of course possible to set the option to the empty list if you do not want the daemon to run any subsystems at all.

`{shell, {Module, Function, Args} | fun(string() = User) - > pid() | fun(string() = User, ip_address() = PeerAddr) -> pid()}`

Defines the read-eval-print loop used when a shell is requested by the client. Default is to use the erlang shell: `{shell, start, []}`

`{ssh_cli, {channel_callback(), channel_init_args()} | no_cli}`

Provides your own CLI implementation, i.e. a channel callback module that implements a shell and command execution. Note that you may customize the shell read-eval-print loop using the option `shell` which is much less work than implementing your own CLI channel. If set to `no_cli` you will disable CLI channels and only subsystem channels will be allowed.

`{user_dir, String}`

Sets the user directory i.e. the directory containing ssh configuration files for the user such as `known_hosts`, `id_rsa, id_dsa` and `authorized_key`. Defaults to the directory normally referred to as `~/.ssh`

`{system_dir, string()}`

Sets the system directory, containing the host key files that identifies the host keys for ssh. The default is `/etc/ssh`, note that for security reasons this directory is normally only accessible by the root user.

`{auth_methods, string()}`

Comma separated string that determines which authentication methodes that the server should support and in what order they will be tried. Defaults to `"publickey,keyboard-interactive,password"`

`{user_passwords, [{string() = User, string() = Password}]}`

Provide passwords for password authentication.They will be used when someone tries to connect to the server and public key user authentication fails. The option provides a list of valid user names and the corresponding password.

`{password, string()}`

Provide a global password that will authenticate any user. From a security perspective this option makes the server very vulnerable.

`{pwdfun, fun(User::string(), password::string()) -> boolean()}`

Provide a function for password validation. This is called with user and password as strings, and should return `true` if the password is valid and `false` otherwise.

`{negotiation_timeout, integer()}`

Max time in milliseconds for the authentication negotiation. The default value is 2 minutes. If the client fails to login within this time, the connection is closed.

`{max_sessions, pos_integer()}`

The maximum number of simultaneous sessions that are accepted at any time for this daemon. This includes sessions that are being authorized. So if set to `N`, and `N` clients have connected but not started the login process, the `N+1` connection attempt will be aborted. If `N` connections are authenticated and still logged in, no more loggins will be accepted until one of the existing ones log out.

The counter is per listening port, so if two daemons are started, one with `{max_sessions,N}` and the other with `{max_sessions,M}` there will be in total `N+M` connections accepted for the whole ssh application.

Note that if `parallel_login` is `false`, only one client at a time may be in the authentication phase.

As default, the option is not set. This means that the number is not limited.

`{parallel_login, boolean()}`

If set to false (the default value), only one login is handled a time. If set to true, an unlimited number of login attempts will be allowed simultanously.

If the `max_sessions` option is set to `N` and `parallel_login` is set to `true`, the max number of simultaneous login attempts at any time is limited to `N-K` where `K` is the number of authenticated connections present at this daemon.

> Warning:
>
> Do not enable `parallel_logins` without protecting the server by other means, for example the `max_sessions` option or a firewall configuration. If set to `true`, there is no protection against DOS attacks.

`{key_cb, atom()}`

Module implementing the behaviour *ssh_server_key_api*. Can be used to customize the handling of public keys.

`{fd, file_descriptor()}`

Allow an existing file-descriptor to be used (simply passed on to the transport protocol).

`{failfun, fun(User::string(), PeerAddress::ip_address(), Reason::term()) -> _}`

Provide a fun to implement your own logging when a user fails to authenticate.

`{connectfun, fun(User::string(), PeerAddress::ip_address(), Method::string()) ->_}`

Provide a fun to implement your own logging when a user authenticates to the server.

`{disconnectfun, fun(Reason:term()) -> _}`

Provide a fun to implement your own logging when a user disconnects from the server.

```
shell(Host) ->
shell(Host, Option) ->
shell(Host, Port, Option) -> _
```

Types:

`Host = string()`
`Port = integer()`
`Options - see ssh:connect/3`

Starts an interactive shell via an SSH server on the given `Host`. The function waits for user input, and will not return until the remote shell is ended (i.e. exit from the shell).

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

Types:

`Type = permanent | transient | temporary`
`Reason = term()`

Utility function that starts crypto, public_key and the SSH application. Defult type is temporary. See also *application(3)*

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

Types:

`Reason = term()`

Stops the SSH application. See also *application(3)*

```
stop_daemon(DaemonRef) ->
stop_daemon(Address, Port) -> ok
```

Types:

DaemonRef = ssh_daemon_ref()
Address = ip_address()
Port = integer()

Stops the listener and all connections started by the listener.

```
stop_listener(DaemonRef) ->
stop_listener(Address, Port) -> ok
```

Types:

DaemonRef = ssh_daemon_ref()
Address = ip_address()
Port = integer()

Stops the listener, but leaves existing connections started by the listener up and running.

ssh_channel

Erlang module

SSH services (clients and servers) are implemented as channels that are multiplexed over an SSH connection and communicates via the **SSH Connection Protocol**. This module provides a callback API that takes care of generic channel aspects such as flow control and close messages and lets the callback functions take care of the service (application) specific parts. This behavior also ensures that the channel process honors the principal of an OTP-process so that it can be part of a supervisor tree. This is a requirement of channel processes implementing a subsystem that will be added to the SSH applications supervisor tree.

> ### Note:
>
> When implementing a SSH subsystem use the `-behaviour(ssh_daemon_channel).` instead of `-behaviour(ssh_channel).` as the only relevant callback functions for subsystems are init/1, handle_ssh_msg/2, handle_msg/2 and terminate/2, so the ssh_daemon_channel behaviour is limited version of the ssh_channel behaviour.

DATA TYPES

Type definitions that are used more than once in this module and/or abstractions to indicate the intended use of the data type:

```
boolean() = true | false

string() = list of ASCII characters

timeout() = infinity | integer() - in milliseconds.

ssh_connection_ref() - opaque to the user returned by ssh:connect/3 or sent to an SSH channel process

ssh_channel_id() = integer()

ssh_data_type_code() = 1 ("stderr") | 0 ("normal") are currently valid values see RFC 4254  section 5.2.
```

Exports

```
call(ChannelRef, Msg) ->
call(ChannelRef, Msg, Timeout) -> Reply | {error, Reason}
```

Types:

`ChannelRef = pid()`

As returned by start_link/4

`Msg = term()`

`Timeout = timeout()`

`Reply = term()`

`Reason = closed | timeout`

Makes a synchronous call to the channel process by sending a message and waiting until a reply arrives or a timeout occurs. The channel will call *Module:handle_call/3* to handle the message. If the channel process does not exist `{error, closed}` is returned.

```
cast(ChannelRef, Msg) -> ok
```

Types:

`ChannelRef = pid()`

As returned by start_link/4

`Msg = term()`

Sends an asynchronous message to the channel process and returns ok immediately, ignoring if the destination node or channel process does not exist. The channel will call *Module:handle_cast/2* to handle the message.

```
enter_loop(State) -> _
```

Types:

`State = term() - as returned by ssh_channel:init/1`

Makes an existing process an `ssh_channel` process. Does not return, instead the calling process will enter the `ssh_channel` process receive loop and become an `ssh_channel process.` The process must have been started using one of the start functions in proc_lib, see *proc_lib(3)*. The user is responsible for any initialization of the process and needs to call *ssh_channel:init/1*

```
init(Options) -> {ok, State} | {ok, State, Timeout} | {stop, Reason}
```

Types:

`Options = [{Option, Value}]`

`State = term()`

`Timeout = timeout()`

`Reason = term()`

The following options must be present:

`{channel_cb, atom()}`
: The module that implements the channel behaviour.

`{init_args(), list()}`
: The list of arguments to the callback module's init function.

`{cm, connection_ref()}`
: Reference to the ssh connection as returned by *ssh:connect/3*

`{channel_id, channel_id()}`
: Id of the SSH channel.

> **Note:**
>
> This function is normally not called by the user. The user only needs to call if for some reason the channel process needs to be started with help of `proc_lib` instead of calling `ssh_channel:start/4` or `ssh_channel:start_link/4`

```
reply(Client, Reply) -> _
```

Types:

`Client - opaque to the user, see explanation below`

Reply = term()

This function can be used by a channel to explicitly send a reply to a client that called `call/[2,3]` when the reply cannot be defined in the return value of *Module:handle_call/3*.

`Client` must be the `From` argument provided to the callback function `handle_call/3`. `Reply` is an arbitrary term, which will be given back to the client as the return value of *ssh_channel:call/[2,3].>*

```
start(SshConnection, ChannelId, ChannelCb, CbInitArgs) ->
start_link(SshConnection, ChannelId, ChannelCb, CbInitArgs) -> {ok,
ChannelRef} | {error, Reason}
```

Types:

SshConnection = ssh_connection_ref()

ChannelId = ssh_channel_id()

As returned by cannot be defined in the return value of *ssh_connection:session_channel/[2,4]*

ChannelCb = atom()

The name of the module implementing the service specific parts of the channel.

CbInitArgs = [term()]

Argument list for the init function in the callback module.

ChannelRef = pid()

Starts a processes that handles an SSH channel. It will be called internally by the SSH daemon or explicitly by the SSH client implementations. The behavior will set the `trap_exit` flag to true.

CALLBACK TIMEOUTS

The timeout values that may be returned by the callback functions has the same semantics as in a *gen_server* If the timeout occurs *handle_msg/2* will be called as `handle_msg(timeout, State).`

Exports

```
Module:code_change(OldVsn, State, Extra) -> {ok, NewState}
```

Types:

OldVsn = term()

In the case of an upgrade, `OldVsn` is `Vsn`, and in the case of a downgrade, `OldVsn` is `{down,Vsn}`. `Vsn` is defined by the `vsn` attribute(s) of the old version of the callback module `Module`. If no such attribute is defined, the version is the checksum of the BEAM file.

State = term()

The internal state of the channel.

Extra = term()

Passed as-is from the `{advanced,Extra}` part of the update instruction.

Converts process state when code is changed.

This function is called by a client side channel when it should update its internal state during a release upgrade/downgrade, i.e. when the instruction `{update,Module,Change,...}` where `Change={advanced,Extra}` is given in the `appup` file. See *OTP Design Principles* for more information.

> **Note:**
>
> Soft upgrade according to the OTP release concept is not straight forward for the server side, as subsystem channel processes are spawned by the SSH application and hence added to its supervisor tree. It could be possible to upgrade the subsystem channels, when upgrading the user application, if the callback functions can handle two versions of the state, but this function can not be used in the normal way.

```
Module:init(Args) -> {ok, State} | {ok, State, timeout()} | {stop, Reason}
```

Types:

`Args = term()`

Last argument to ssh_channel:start_link/4.

`State = term()`

`Reason = term()`

Makes necessary initializations and returns the initial channel state if the initializations succeed.

For more detailed information on timeouts see the section *CALLBACK TIMEOUTS*.

```
Module:handle_call(Msg, From, State) -> Result
```

Types:

`Msg = term()`

`From = opaque to the user should be used as argument to ssh_channel:reply/2`

`State = term()`

`Result = {reply, Reply, NewState} | {reply, Reply, NewState, timeout()} | {noreply, NewState} | {noreply , NewState, timeout()} | {stop, Reason, Reply, NewState} | {stop, Reason, NewState}`

`Reply = term() - will be the return value of ssh_channel:call/[2,3]`

`NewState = term()`

`Reason = term()`

Handles messages sent by calling *ssh_channel:call/[2,3]*

For more detailed information on timeouts see the section *CALLBACK TIMEOUTS*.

```
Module:handle_cast(Msg, State) -> Result
```

Types:

`Msg = term()`

`State = term()`

`Result = {noreply, NewState} | {noreply, NewState, timeout()} | {stop, Reason, NewState}`

`NewState = term()`

`Reason = term()`

Handles messages sent by calling `ssh_channel:cast/2`

For more detailed information on timeouts see the section *CALLBACK TIMEOUTS*.

`Module:handle_msg(Msg, State) -> {ok, State} | {stop, ChannelId, State}`

Types:

`Msg = timeout | term()`
`ChannelId = ssh_channel_id()`
`State = term()`

Handle other messages than ssh connection protocol, call or cast messages sent to the channel.

Possible erlang 'EXIT'-messages should be handled by this function and all channels should handle the following message.

`{ssh_channel_up, ssh_channel_id(), ssh_connection_ref()}`

This is the first messages that will be received by the channel, it is sent just before the *ssh_channel:init/1* function returns successfully. This is especially useful if the server wants to send a message to the client without first receiving a message from it. If the message is not useful for your particular scenario just ignore it by immediately returning {ok, State}.

`Module:handle_ssh_msg(Msg, State) -> {ok, State} | {stop, ChannelId, State}`

Types:

`Msg =` ***`ssh_connection:event()`***
`ChannelId = ssh_channel_id()`
`State = term()`

Handles SSH connection protocol messages that may need service specific attention.

The following message is completely taken care of by the SSH channel behavior

`{closed, ssh_channel_id()}`

The channel behavior will send a close message to the other side if such a message has not already been sent and then terminate the channel with reason normal.

`Module:terminate(Reason, State) -> _`

Types:

`Reason = term()`
`State = term()`

This function is called by a channel process when it is about to terminate. Before this function is called *ssh_connection:close/2* will be called if it has not been called earlier. This function should do any necessary cleaning up. When it returns, the channel process terminates with reason `Reason`. The return value is ignored.

ssh_connection

Erlang module

The SSH Connection Protocol is used by clients and servers (i.e. SSH channels) to communicate over the SSH connection. The API functions in this module sends SSH Connection Protocol events that are received as messages by the remote channel. In the case that the receiving channel is an Erlang process the message will be on the following format `{ssh_cm, ssh_connection_ref(), ssh_event_msg()}`. If the *ssh_channel* behavior is used to implement the channel process these will be handled by *handle_ssh_msg/2* .

DATA TYPES

Type definitions that are used more than once in this module and/or abstractions to indicate the intended use of the data type:

```
boolean() = true | false
```

```
string() = list of ASCII characters
```

```
timeout() = infinity | integer() - in milliseconds.
```

```
ssh_connection_ref() - opaque to the user returned by ssh:connect/3 or sent to an SSH channel processes
```

```
ssh_channel_id() = integer()
```

```
ssh_data_type_code() = 1 ("stderr") | 0 ("normal") are currently valid values
```
see **RFC 4254** section 5.2.

```
ssh_request_status() = success | failure
```

```
event() = {ssh_cm, ssh_connection_ref(), ssh_event_msg()}
```

```
ssh_event_msg() = data_events() | status_events() | terminal_events()
```

data_events()

`{data, ssh_channel_id(), ssh_data_type_code(), binary() = Data}`
: Data has arrived on the channel. This event is sent as result of calling *ssh_connection:send/[3,4,5]*

`{eof, ssh_channel_id()}`
: Indicates that the other side will not send any more data. This event is sent as result of calling *ssh_connection:send_eof/2*

status_events()

`{signal, ssh_channel_id(), ssh_signal()}`
: A signal can be delivered to the remote process/service using the following message. Some systems will not support signals, in which case they should ignore this message. There is currently no funtion to generate this event as the signals refered to are on OS-level and not something generated by an Erlang program.

`{exit_signal, ssh_channel_id(), string() = ExitSignal, string() = ErrorMsg, string() = LanguageString}`
: A remote execution may terminate violently due to a signal then this message may be received. For details on valid string values see **RFC 4254** section 6.10. Special case of the signals mentioned above.

`{exit_status, ssh_channel_id(), integer() = ExitStatus}`
: When the command running at the other end terminates, the following message can be sent to return the exit status of the command. A zero 'exit_status' usually means that the command terminated successfully. This event is sent as result of calling *ssh_connection:exit_status/3*

`{closed, ssh_channel_id()}`

This event is sent as result of calling *ssh_connection:close/2* Both the handling of this event and sending of it will be taken care of by the *ssh_channel* behavior.

terminal_events()

Channels implementing a shell and command execution on the server side should handle the following messages that may be sent by client channel processes.

> Note:
>
> Events that includes a `WantReply` expects the event handling process to call *ssh_connection:reply_request/4* with the boolean value of `WantReply` as the second argument.

`{env, ssh_channel_id(), boolean() = WantReply, string() = Var, string() = Value}`

Environment variables may be passed to the shell/command to be started later. This event is sent as result of calling *ssh_connection:setenv/5*

`{pty, ssh_channel_id(), boolean() = WantReply, {string() = Terminal, integer() = CharWidth, integer() = RowHeight, integer() = PixelWidth, integer() = PixelHeight, [{atom() | integer() = Opcode, integer() = Value}] = TerminalModes}}`

A pseudo-terminal has been requested for the session. Terminal is the value of the TERM environment variable value (e.g., vt100). Zero dimension parameters must be ignored. The character/row dimensions override the pixel dimensions (when nonzero). Pixel dimensions refer to the drawable area of the window. The `Opcode` in the `TerminalModes` list is the mnemonic name, represented as an lowercase erlang atom, defined in **RFC 4254** section 8. It may also be an opcode if the mnemonic name is not listed in the RFC. Example `OP code: 53, mnemonic name ECHO erlang atom: echo`.This event is sent as result of calling *ssh_connection:ptty_alloc/4*

`{shell, boolean() = WantReply}`

This message will request that the user's default shell be started at the other end. This event is sent as result of calling *ssh_connection:shell/2*

`{window_change, ssh_channel_id(), integer() = CharWidth, integer() = RowHeight, integer() = PixWidth, integer() = PixHeight}`

When the window (terminal) size changes on the client side, it MAY send a message to the server side to inform it of the new dimensions. There is currently no API function to generate this event.

`{exec, ssh_channel_id(), boolean() = WantReply, string() = Cmd}`

This message will request that the server starts execution of the given command. This event is sent as result of calling *ssh_connection:exec/4*

Exports

`adjust_window(ConnectionRef, ChannelId, NumOfBytes) -> ok`

Types:

`ConnectionRef = ssh_connection_ref()`
`ChannelId = ssh_channel_id()`
`NumOfBytes = integer()`

Adjusts the SSH flowcontrol window. This shall be done by both client and server side channel processes.

> Note:
>
> Channels implemented with the *ssh_channel behavior* will normaly not need to call this function as flow control will be handled by the behavior. The behavior will adjust the window every time the callback *handle_ssh_msg/2* has returned after processing channel data

`close(ConnectionRef, ChannelId) -> ok`

Types:

`ConnectionRef = ssh_connection_ref()`

`ChannelId = ssh_channel_id()`

A server or client channel process can choose to close their session by sending a close event.

> Note:
>
> This function will be called by the ssh_channel behavior when the channel is terminated see *ssh_channel(3)* so channels implemented with the behavior should not call this function explicitly.

`exec(ConnectionRef, ChannelId, Command, TimeOut) -> ssh_request_status()`

Types:

`ConnectionRef = ssh_connection_ref()`

`ChannelId = ssh_channel_id()`

`Command = string()`

`Timeout = timeout()`

Should be called by a client channel process to request that the server starts execution of the given command, the result will be several messages according to the following pattern. Note that the last message will be a channel close message, as the exec request is a one time execution that closes the channel when it is done.

`N x {ssh_cm, ssh_connection_ref(), {data, ssh_channel_id(), ssh_data_type_code(), binary() = Data}}`

The result of executing the command may be only one line or thousands of lines depending on the command.

`0 or 1 x {ssh_cm, ssh_connection_ref(), {eof, ssh_channel_id()}}`

Indicates that no more data will be sent.

`0 or 1 x {ssh_cm, ssh_connection_ref(), {exit_signal, ssh_channel_id(), string() = ExitSignal, string() = ErrorMsg, string() = LanguageString}}`

Not all systems send signals. For details on valid string values see RFC 4254 section 6.10

`0 or 1 x {ssh_cm, ssh_connection_ref(), {exit_status, ssh_channel_id(), integer() = ExitStatus}}`

It is recommended by the `ssh connection protocol` that this message shall be sent, but that may not always be the case.

`1 x {ssh_cm, ssh_connection_ref(), {closed, ssh_channel_id()}}`

Indicates that the ssh channel started for the execution of the command has now been shutdown.

`exit_status(ConnectionRef, ChannelId, Status) -> ok`

Types:

`ConnectionRef = ssh_connection_ref()`

ChannelId = ssh_channel_id()
Status = integer()

Should be called by a server channel process to sends the exit status of a command to the client.

```
ptty_alloc(ConnectionRef, ChannelId, Options, Timeout) -> success | failure
```

Types:

ConnectionRef = ssh_connection_ref()
ChannelId = ssh_channel_id()
Options = proplists:proplist()

Sends a SSH Connection Protocol pty_req, to allocate a pseudo tty. Should be called by a SSH client process. Options are:

{term, string()}
Defaults to os:getenv("TERM") or "vt100" if it is undefined.
{width, integer()}
Defaults to 80 if pixel_width is not defined.
{height, integer()}
Defaults to 24 if pixel_height is not defined.
{pixel_width, integer()}
Is disregarded if width is defined.
{pixel_height, integer()}
Is disregarded if height is defined.
{pty_opts, [{posix_atom(), integer()}]}
Option may be an empty list, otherwise see possible POSIX names in section 8 in **RFC 4254**.

```
reply_request(ConnectionRef, WantReply, Status, ChannelId) -> ok
```

Types:

ConnectionRef = ssh_connection_ref()
WantReply = boolean()
Status = ssh_request_status()
ChannelId = ssh_channel_id()

Sends status replies to requests where the requester has stated that they want a status report e.i. `WantReply = true`, if `WantReply` is false calling this function will be a "noop". Should be called while handling an ssh connection protocol message containing a `WantReply` boolean value.

```
send(ConnectionRef, ChannelId, Data) ->
send(ConnectionRef, ChannelId, Data, Timeout) ->
send(ConnectionRef, ChannelId, Type, Data) ->
send(ConnectionRef, ChannelId, Type, Data, TimeOut) -> ok | {error, timeout}
| {error, closed}
```

Types:

ConnectionRef = ssh_connection_ref()
ChannelId = ssh_channel_id()
Data = binary()
Type = ssh_data_type_code()
Timeout = timeout()

Should be called by client- and server channel processes to send data to each other.

```
send_eof(ConnectionRef, ChannelId) -> ok | {error, closed}
```

Types:

ConnectionRef = ssh_connection_ref()
ChannelId = ssh_channel_id()

Sends eof on the channel `ChannelId`.

```
session_channel(ConnectionRef, Timeout) ->
session_channel(ConnectionRef, InitialWindowSize, MaxPacketSize, Timeout) ->
{ok, ssh_channel_id()} | {error, Reason}
```

Types:

ConnectionRef = ssh_connection_ref()
InitialWindowSize = integer()
MaxPacketSize = integer()
Timeout = timeout()
Reason = term()

Opens a channel for an SSH session. The channel id returned from this function is the id used as input to the other funtions in this module.

```
setenv(ConnectionRef, ChannelId, Var, Value, TimeOut) -> ssh_request_status()
```

Types:

ConnectionRef = ssh_connection_ref()
ChannelId = ssh_channel_id()
Var = string()
Value = string()
Timeout = timeout()

Environment variables may be passed before starting the shell/command. Should be called by a client channel processes.

```
shell(ConnectionRef, ChannelId) -> ssh_request_status()
```

Types:

ConnectionRef = ssh_connection_ref()
ChannelId = ssh_channel_id()

Should be called by a client channel process to request that the user's default shell (typically defined in /etc/passwd in UNIX systems) shall be executed at the server end.

```
subsystem(ConnectionRef, ChannelId, Subsystem, Timeout) ->
ssh_request_status()
```

Types:

ConnectionRef = ssh_connection_ref()
ChannelId = ssh_channel_id()
Subsystem = string()
Timeout = timeout()

Should be called by a client channel process for requesting to execute a predefined subsystem on the server.

ssh_client_key_api

Erlang module

Behavior describing the API for an SSH client's public key handling. By implementing the callbacks defined. in this behavior it is possible to customize the SSH client's public key handling. By default the SSH application implements this behavior with help of the standard openssh files, see *ssh(6)*.

DATA TYPES

Type definitions that are used more than once in this module and/or abstractions to indicate the intended use of the data type. For more details on public key data types see the *public_key user's guide*.

boolean() = true | false

string() = [byte()]

public_key() = #'RSAPublicKey'{}| {integer(), #'Dss-Parms'{}}| term()

private_key() = #'RSAPrivateKey'{} | #'DSAPrivateKey'{} | term()

public_key_algorithm() = 'ssh-rsa'| 'ssh-dss' | atom()

Exports

`Module:add_host_key(HostNames, Key, ConnectOptions) -> ok | {error, Reason}`

Types:

`HostNames = string()`

Description of the host that owns the `PublicKey`

`Key = public_key()`

Normally an RSA or DSA public key but handling of other public keys can be added

`ConnectOptions = proplists:proplist()`

Options provided to *ssh:connect/[3,4]*

`Reason = term()`

Adds a host key to the set of trusted host keys

`Module:is_host_key(Key, Host, Algorithm, ConnectOptions) -> Result`

Types:

`Key = public_key()`

Normally an RSA or DSA public key but handling of other public keys can be added

`Host = string()`

Description of the host

`Algorithm = public_key_algorithm()`

Host key algorithm. Should support 'ssh-rsa'| 'ssh-dss' but additional algorithms can be handled.

`ConnectOptions = proplists:proplist()`

Options provided to *ssh:connect/[3,4]*

`Result = boolean()`

Checks if a host key is trusted

`Module:user_key(Algorithm, ConnectOptions) -> {ok, PrivateKey} | {error, Reason}`

Types:

`Algorithm = public_key_algorithm()`

Host key algorithm. Should support 'ssh-rsa'| 'ssh-dss' but additional algorithms can be handled.

`ConnectOptions = proplists:proplist()`

Options provided to *ssh:connect/[3,4]*

`PrivateKey = private_key()`

The private key of the user matching the `Algorithm`

`Reason = term()`

Fetches the users "public key" matching the `Algorithm`.

> Note:
>
> The private key contains the public key

ssh_server_key_api

Erlang module

Behaviour describing the API for an SSH server's public key handling. By implementing the callbacks defined in this behavior it is possible to customize the SSH server's public key handling. By default the SSH application implements this behavior with help of the standard openssh files, see *ssh(6)*.

DATA TYPES

Type definitions that are used more than once in this module and/or abstractions to indicate the intended use of the data type. For more details on public key data types see the *public_key user's guide*.

boolean() = true | false

string() = [byte()]

public_key() = #'RSAPublicKey'{} | {integer(), #'Dss-Parms'{}} | term()

private_key() = #'RSAPrivateKey'{} | #'DSAPrivateKey'{} | term()

public_key_algorithm() = 'ssh-rsa' | 'ssh-dss' | atom()

Exports

`Module:host_key(Algorithm, DaemonOptions) -> {ok, Key} | {error, Reason}`

Types:

`Algorithm = public_key_algorithm()`

Host key algorithm. Should support 'ssh-rsa' | 'ssh-dss' but additional algorithms can be handled.

`DaemonOptions = proplists:proplist()`

Options provided to *ssh:daemon/[2,3]*

`Key = private_key()`

The private key of the host matching the `Algorithm`

`Reason = term()`

Fetches the hosts private key

`Module:is_auth_key(Key, User, DaemonOptions) -> Result`

Types:

`Key = public_key()`

Normally an RSA or DSA public key but handling of other public keys can be added

`User = string()`

The user owning the public key

`DaemonOptions = proplists:proplist()`

Options provided to *ssh:daemon/[2,3]*

`Result = boolean()`

Checks if the user key is authorized

ssh_sftp

Erlang module

This module implements an SFTP (SSH FTP) client. SFTP is a secure, encrypted file transfer service available for SSH.

DATA TYPES

Type definitions that are used more than once in this module and/or abstractions to indicate the intended use of the data type:

```
ssh_connection_ref() - opaque to the user returned by ssh:connect/3
timeout() = infinity | integer() - in milliseconds.
```

TIMEOUTS

If the request functions for the SFTP channel return {error, timeout} it does not guarantee that the request did not reach the server and was not performed, it only means that we did not receive an answer from the server within the time that was expected.

Exports

```
start_channel(ConnectionRef) ->
start_channel(ConnectionRef, Options) ->
start_channel(Host, Options) ->
start_channel(Host, Port, Options) -> {ok, Pid} | {ok, Pid, ConnectionRef} |
{error, Reason}
```

Types:

```
Host = string()
ConnectionRef = ssh_connection_ref()
Port = integer()
Options = [{Option, Value}]
Reason = term()
```

If no connection reference is provided, a connection is set up and the new connection is returned. An SSH channel process is started to handle the communication with the SFTP server. The returned pid for this process should be used as input to all other API functions in this module.

Options are:

`{timeout, timeout()}`

The timeout is passed to the ssh_channel start function, and defaults to infinity.

`{sftp_vsn, integer()}`

Desired SFTP protocol version. The actual version will be the minimum of the desired version and the maximum supported versions by the SFTP server.

All other options are directly passed to *ssh:connect/3* or ignored if a connection is already provided.

```
stop_channel(ChannelPid) -> ok
```

Types:

ChannelPid = pid()

Stops an SFTP channel. Does not close the SSH connetion. Use *ssh:close/1* to close it.

```
read_file(ChannelPid, File) ->
read_file(ChannelPid, File, Timeout) -> {ok, Data} | {error, Reason}
```

Types:

ChannelPid = pid()
File = string()
Data = binary()
Timeout = timeout()
Reason = term()

Reads a file from the server, and returns the data in a binary, like `file:read_file/1`.

```
write_file(ChannelPid, File, Iolist) ->
write_file(ChannelPid, File, Iolist, Timeout) -> ok | {error, Reason}
```

Types:

ChannelPid = pid()
File = string()
Iolist = iolist()
Timeout = timeout()
Reason = term()

Writes a file to the server, like `file:write_file/2`. The file is created if it does not exist or is owerwritten if it does.

```
list_dir(ChannelPid, Path) ->
list_dir(ChannelPid, Path, Timeout) -> {ok, Filenames} | {error, Reason}
```

Types:

ChannelPid = pid()
Path = string()
Filenames = [Filename]
Filename = string()
Timeout = timeout()
Reason = term()

Lists the given directory on the server, returning the filenames as a list of strings.

```
open(ChannelPid, File, Mode) ->
open(ChannelPid, File, Mode, Timeout) -> {ok, Handle} | {error, Reason}
```

Types:

ChannelPid = pid()
File = string()
Mode = [Modeflag]
Modeflag = read | write | creat | trunc | append | binary
Timeout = timeout()

Handle = term()
Reason = term()

Opens a file on the server, and returns a handle that can be used for reading or writing.

```
opendir(ChannelPid, Path) ->
opendir(ChannelPid, Path, Timeout) -> {ok, Handle} | {error, Reason}
```

Types:

ChannelPid = pid()
Path = string()
Timeout = timeout()
Reason = term()

Opens a handle to a directory on the server, the handle can be used for reading directory contents.

```
open_tar(ChannelPid, Path, Mode) ->
open_tar(ChannelPid, Path, Mode, Timeout) -> {ok, Handle} | {error, Reason}
```

Types:

ChannelPid = pid()
Path = string()
Mode = [read] | [write] | [read,EncryptOpt] | [write,DecryptOpt]
EncryptOpt = {crypto,{InitFun,EncryptFun,CloseFun}}
DecryptOpt = {crypto,{InitFun,DecryptFun}}
InitFun = (fun() -> {ok,CryptoState}) | (fun() ->
{ok,CryptoState,ChunkSize})
CryptoState = any()
ChunkSize = undefined | pos_integer()
EncryptFun = (fun(PlainBin,CryptoState) -> EncryptResult)
EncryptResult = {ok,EncryptedBin,CryptoState} |
{ok,EncryptedBin,CryptoState,ChunkSize}
PlainBin = binary()
EncryptedBin = binary()
DecryptFun = (fun(EncryptedBin,CryptoState) -> DecryptResult)
DecryptResult = {ok,PlainBin,CryptoState} |
{ok,PlainBin,CryptoState,ChunkSize}
CloseFun = (fun(PlainBin,CryptoState) -> {ok,EncryptedBin})
Timeout = timeout()
Reason = term()

Opens a handle to a tar file on the server associated with `ChannelPid`. The handle can be used for remote tar creation and extraction as defined by the *erl_tar:init/3* function.

An example of writing and then reading a tar file:

```
        {ok,HandleWrite} = ssh_sftp:open_tar(ChannelPid, ?tar_file_name, [write]),
        ok = erl_tar:add(HandleWrite, .... ),
        ok = erl_tar:add(HandleWrite, .... ),
            ...
```

```
ok = erl_tar:add(HandleWrite, .... ),
ok = erl_tar:close(HandleWrite),

%% And for reading
{ok,HandleRead} = ssh_sftp:open_tar(ChannelPid, ?tar_file_name, [read]),
{ok,NameValueList} = erl_tar:extract(HandleRead,[memory]),
ok = erl_tar:close(HandleRead),
```

The `crypto` mode option is applied to the generated stream of bytes just prior to sending them to the sftp server. This is intended for encryption but could of course be used for other purposes.

The `InitFun` is applied once prior to any other crypto operation. The returned `CryptoState` is then folded into repeated applications of the `EncryptFun` or `DecryptFun`. The binary returned from those Funs are sent further to the remote sftp server. Finally - if doing encryption - the `CloseFun` is applied to the last piece of data. The `CloseFun` is responsible for padding (if needed) and encryption of that last piece.

The `ChunkSize` defines the size of the `PlainBin`s that `EncodeFun` is applied to. If the `ChunkSize` is `undefined` the size of the `PlainBin`s varies because this is inteded for stream crypto while a fixed `ChunkSize` is intended for block crypto. It is possible to change the `ChunkSize`s in the return from the `EncryptFun` or `DecryptFun`. It is in fact possible to change the value between `pos_integer()` and `undefined`.

The write and read example above can be extended with encryption and decryption:

```
%% First three parameters depending on which crypto type we select:
Key = <<"This is a 256 bit key. abcdefghi">>,
Ivec0 = crypto:rand_bytes(16),
DataSize = 1024,  % DataSize rem 16 = 0 for aes_cbc

%% Initialization of the CryptoState, in this case it is the Ivector.
InitFun = fun() -> {ok, Ivec0, DataSize} end,

%% How to encrypt:
EncryptFun =
    fun(PlainBin,Ivec) ->
         EncryptedBin = crypto:block_encrypt(aes_cbc256, Key, Ivec, PlainBin),
         {ok, EncryptedBin, crypto:next_iv(aes_cbc,EncryptedBin)}
    end,

%% What to do with the very last block:
CloseFun =
    fun(PlainBin, Ivec) ->
          EncryptedBin = crypto:block_encrypt(aes_cbc256, Key, Ivec,
                                              pad(16,PlainBin) %% Last chunk
                                              ),
          {ok, EncryptedBin}
  end,

Cw = {InitFun,EncryptFun,CloseFun},
{ok,HandleWrite} = ssh_sftp:open_tar(ChannelPid, ?tar_file_name, [write,{crypto,Cw}]),
ok = erl_tar:add(HandleWrite, .... ),
ok = erl_tar:add(HandleWrite, .... ),
    ...
ok = erl_tar:add(HandleWrite, .... ),
ok = erl_tar:close(HandleWrite),

%% And for decryption (in this crypto example we could use the same InitFun
%% as for encryption):
DecryptFun =
    fun(EncryptedBin,Ivec) ->
         PlainBin = crypto:block_decrypt(aes_cbc256, Key, Ivec, EncryptedBin),
```

```
                {ok, PlainBin, crypto:next_iv(aes_cbc,EncryptedBin)}
            end,

        Cr = {InitFun,DecryptFun},
        {ok,HandleRead} = ssh_sftp:open_tar(ChannelPid, ?tar_file_name, [read,{crypto,Cw}]),
        {ok,NameValueList} = erl_tar:extract(HandleRead,[memory]),
        ok = erl_tar:close(HandleRead),
```

```
close(ChannelPid, Handle) ->
close(ChannelPid, Handle, Timeout) -> ok | {error, Reason}
```

Types:

ChannelPid = pid()
Handle = term()
Timeout = timeout()
Reason = term()

Closes a handle to an open file or directory on the server.

```
read(ChannelPid, Handle, Len) ->
read(ChannelPid, Handle, Len, Timeout) -> {ok, Data} | eof | {error, Error}
pread(ChannelPid, Handle, Position, Len) ->
pread(ChannelPid, Handle, Position, Len, Timeout) -> {ok, Data} | eof |
{error, Error}
```

Types:

ChannelPid = pid()
Handle = term()
Position = integer()
Len = integer()
Timeout = timeout()
Data = string() | binary()
Reason = term()

Reads `Len` bytes from the file referenced by `Handle`. Returns `{ok, Data}`, `eof`, or `{error, Reason}`. If the file is opened with `binary`, `Data` is a binary, otherwise it is a string.

If the file is read past eof, only the remaining bytes will be read and returned. If no bytes are read, `eof` is returned.

The `pread` function reads from a specified position, combining the `position` and `read` functions.

```
aread(ChannelPid, Handle, Len) -> {async, N} | {error, Error}
apread(ChannelPid, Handle, Position, Len) -> {async, N} | {error, Error}
```

Types:

ChannelPid = pid()
Handle = term()
Position = integer()
Len = integer()
N = term()
Reason = term()

Reads from an open file, without waiting for the result. If the handle is valid, the function returns `{async, N}`, where N is a term guaranteed to be unique between calls of `aread`. The actual data is sent as a message to the calling process. This message has the form `{async_reply, N, Result}`, where `Result` is the result from the read, either `{ok, Data}`, or `eof`, or `{error, Error}`.

The `apread` function reads from a specified position, combining the `position` and `aread` functions.

```
write(ChannelPid, Handle, Data) ->
write(ChannelPid, Handle, Data, Timeout) -> ok | {error, Error}
pwrite(ChannelPid, Handle, Position, Data) -> ok
pwrite(ChannelPid, Handle, Position, Data, Timeout) -> ok | {error, Error}
```

Types:

`ChannelPid = pid()`
`Handle = term()`
`Position = integer()`
`Data = iolist()`
`Timeout = timeout()`
`Reason = term()`

Writes`data` to the file referenced by `Handle`. The file should be opened with `write` or `append` flag. Returns `ok` if successful or S`{error, Reason}` otherwise.

Typical error reasons are:

`ebadf`

The file is not opened for writing.

`enospc`

There is a no space left on the device.

```
awrite(ChannelPid, Handle, Data) -> ok | {error, Reason}
apwrite(ChannelPid, Handle, Position, Data) -> ok | {error, Reason}
```

Types:

`ChannelPid = pid()`
`Handle = term()`
`Position = integer()`
`Len = integer()`
`Data = binary()`
`Timeout = timeout()`
`Reason = term()`

Writes to an open file, without waiting for the result. If the handle is valid, the function returns `{async, N}`, where N is a term guaranteed to be unique between calls of `awrite`. The result of the `write` operation is sent as a message to the calling process. This message has the form `{async_reply, N, Result}`, where `Result` is the result from the write, either `ok`, or `{error, Error}`.

The `apwrite` writes on a specified position, combining the `position` and `awrite` operations.

```
position(ChannelPid, Handle, Location) ->
position(ChannelPid, Handle, Location, Timeout) -> {ok, NewPosition | {error,
Error}
```

Types:

ChannelPid = pid()
Handle = term()
Location = Offset | {bof, Offset} | {cur, Offset} | {eof, Offset} | bof | cur | eof
Offset = integer()
Timeout = timeout()
NewPosition = integer()
Reason = term()

Sets the file position of the file referenced by `Handle`. Returns `{ok, NewPosition}` (as an absolute offset) if successful, otherwise `{error, Reason}`. `Location` is one of the following:

`Offset`

The same as `{bof, Offset}`.

`{bof, Offset}`

Absolute offset.

`{cur, Offset}`

Offset from the current position.

`{eof, Offset}`

Offset from the end of file.

`bof | cur | eof`

The same as above with `Offset` 0.

```
read_file_info(ChannelPid, Name) ->
read_file_info(ChannelPid, Name, Timeout) -> {ok, FileInfo} | {error, Reason}
```

Types:

ChannelPid = pid()
Name = string()
Handle = term()
Timeout = timeout()
FileInfo = record()
Reason = term()

Returns a `file_info` record from the file specified by `Name` or `Handle`, like `file:read_file_info/2`.

```
read_link_info(ChannelPid, Name) -> {ok, FileInfo} | {error, Reason}
read_link_info(ChannelPid, Name, Timeout) -> {ok, FileInfo} | {error, Reason}
```

Types:

ChannelPid = pid()
Name = string()
Handle = term()

Timeout = timeout()
FileInfo = record()
Reason = term()

Returns a `file_info` record from the symbolic link specified by `Name` or `Handle`, like `file:read_link_info/2`.

```
write_file_info(ChannelPid, Name, Info) ->
write_file_info(ChannelPid, Name, Info, Timeout) -> ok | {error, Reason}
```

Types:

ChannelPid = pid()
Name = string()
Info = record()
Timeout = timeout()
Reason = term()

Writes file information from a `file_info` record to the file specified by `Name`, like `file:write_file_info`.

```
read_link(ChannelPid, Name) ->
read_link(ChannelPid, Name, Timeout) -> {ok, Target} | {error, Reason}
```

Types:

ChannelPid = pid()
Name = string()
Target = string()
Reason = term()

Reads the link target from the symbolic link specified by `name`, like `file:read_link/1`.

```
make_symlink(ChannelPid, Name, Target) ->
make_symlink(ChannelPid, Name, Target, Timeout) -> ok | {error, Reason}
```

Types:

ChannelPid = pid()
Name = string()
Target = string()
Reason = term()

Creates a symbolic link pointing to `Target` with the name `Name`, like `file:make_symlink/2`.

```
rename(ChannelPid, OldName, NewName) ->
rename(ChannelPid, OldName, NewName, Timeout) -> ok | {error, Reason}
```

Types:

ChannelPid = pid()
OldName = string()
NewName = string()
Timeout = timeout()
Reason = term()

Renames a file named `OldName`, and gives it the name `NewName`, like `file:rename/2`

```
delete(ChannelPid, Name) ->
delete(ChannelPid, Name, Timeout) -> ok | {error, Reason}
```

Types:

ChannelPid = pid()
Name = string()
Timeout = timeout()
Reason = term()

Deletes the file specified by `Name`, like `file:delete/1`

```
make_dir(ChannelPid, Name) ->
make_dir(ChannelPid, Name, Timeout) -> ok | {error, Reason}
```

Types:

ChannelPid = pid()
Name = string()
Timeout = timeout()
Reason = term()

Creates a directory specified by `Name`. `Name` should be a full path to a new directory. The directory can only be created in an existing directory.

```
del_dir(ChannelPid, Name) ->
del_dir(ChannelPid, Name, Timeout) -> ok | {error, Reason}
```

Types:

ChannelPid = pid()
Name = string()
Timeout = timeout()
Reason = term()

Deletes a directory specified by `Name`. Note that the directory must be empty before it can be successfully deleted

ssh_sftpd

Erlang module

Specifies a channel process to handle a sftp subsystem.

DATA TYPES

`subsystem_spec() = {subsystem_name(), {channel_callback(), channel_init_args()}}`

`subsystem_name() = "sftp"`

`channel_callback() = atom()` - Name of the erlang module implementing the subsystem using the ssh_channel behavior see *ssh_channel(3)*

`channel_init_args() = list()` - The one given as argument to function `subsystem_spec/1`.

Exports

`subsystem_spec(Options) -> subsystem_spec()`

Types:

`Options = [{Option, Value}]`

Should be used together with ssh:daemon/[1,2,3]

Options are:

`{cwd, String}`

Sets the initial current working directory for the server.

`{file_handler, CallbackModule}`

Determines which module to call for accessing the file server. The default value is `ssh_sftpd_file` that uses the *file* and *filelib* API:s to access the standard OTP file server. This option may be used to plug in other file servers.

`{max_files, Integer}`

The default value is `0`, which means that there is no upper limit. If supplied, the number of filenames returned to the sftp client per `READDIR` request is limited to at most the given value.

`{root, String}`

Sets the sftp root directory. The user will then not be able to see any files above this root. If for instance the root is set to `/tmp` the user will see this directory as `/` and if the user does cd `/etc` the user will end up in `/tmp/etc`.

`{sftpd_vsn, integer()}`

Sets the sftp version to use, defaults to 5. Version 6 is under development and limited.