SSH

SSH 4.3.6
October 21, 2016

October 21, 2016

1 SSH User's Guide

The Erlang Secure Shell (SSH) application, ssh, implements the SSH Transport Layer Protocol and provides SSH File Transfer Protocol (SFTP) clients and servers.

1.1 Introduction

SSH is a protocol for secure remote logon and other secure network services over an insecure network.

1.1.1 Scope and Purpose

SSH provides a single, full-duplex, and byte-oriented connection between client and server. The protocol also provides privacy, integrity, server authentication, and man-in-the-middle protection.

The ssh application is an implementation of the SSH Transport, Connection and Authentication Layer Protocols in Erlang. It provides the following:

- API functions to write customized SSH clients and servers applications
- The Erlang shell available over SSH
- An SFTP client (*ssh_sftp*) and server (*ssh_sftpd*)

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 **public keys**.

1.1.3 SSH Protocol Overview

Conceptually, the SSH protocol can be partitioned into four layers:

Figure 1.1: 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. A minimum of Message Authentication Code (MAC) and encryption algorithms are supported. For details, see the *ssh(3)* manual page in ssh.

Authentication Protocol

The SSH Authentication Protocol is a general-purpose user authentication protocol run over the SSH Transport Layer Protocol. The ssh application supports user authentication as follows:

- Using public key technology. RSA and DSA, X509-certificates are not supported.
- Using keyboard-interactive authentication. This is suitable for interactive authentication methods that do not need any special software support on the client side. Instead, all authentication data is entered from the keyboard.
- Using a pure password-based authentication scheme. Here, the plain text password is encrypted before sent over the network.

Several configuration options for authentication handling are available in *ssh:connect/[3,4]* and *ssh:daemon/[2,3]*.

The public key handling can be customized by implementing the following behaviours from `ssh`:

- Module *ssh_client_key_api*.
- Module *ssh_server_key_api*.

Connection Protocol

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

Channels

All terminal sessions, forwarded connections, and so on, are channels. Multiple channels are multiplexed into a single connection. All channels are flow-controlled. This means that no data is 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. Typically, an SSH client opens a channel, sends data (commands), receives data (control information), and then closes the channel. The *ssh_channel* behaviour handles generic parts of SSH channel management. This makes it easy to write your own SSH client/server processes that use flow-control and thus opens for more focus on the application logic.

Channels come in the following three flavors:

- **Subsystem** - Named services that can be run as part of an SSH server, such as SFTP *(ssh_sftpd)*, that is built into the SSH daemon (server) by default, but it can be disabled. The Erlang `ssh` daemon can be configured to run any Erlang- implemented SSH subsystem.
- **Shell** - Interactive shell. By default the Erlang daemon runs the Erlang shell. The shell can be customized by providing your own read-eval-print loop. You can also provide your own Command-Line Interface (CLI) implementation, but that is much more work.
- **Exec** - One-time remote execution of commands. See function *ssh_connection:exec/4* for more information.

1.1.4 Where to Find More Information

For detailed information about the SSH protocol, refer to the following Request for Comments(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

1.2 Getting Started

1.2.1 General Information

The following examples use the utility function *ssh:start/0* to start 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 `ssh` application can be used. The examples are run as the user `otptest` on a local network where the user is authorized to log in over `ssh` to the host **tarlop**.

If nothing else is stated, it is presumed that the `otptest` user has an entry in the **authorized_keys** file of **tarlop** (allowed to log in over `ssh` without entering a password). Also, **tarlop** is a known host in the `known_hosts` file of the user `otptest`. This means that host-verification can be done without user-interaction.

1.2.2 Using the Erlang ssh Terminal Client

The user `otptest`, which has bash as default shell, uses the `ssh:shell/1` client to connect to the **openssh** daemon running on a host called **tarlop**:

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

1.2.3 Running an Erlang ssh Daemon

The `system_dir` option 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 directory `users ~/.ssh`.

Step 1. To run the example without root privileges, generate new keys and host keys:

```
$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
[...]
```

Step 2. Create the file `/tmp/otptest_user/.ssh/authorized_keys` and add the content of `/tmp/otptest_user/.ssh/id_rsa.pub`.

Step 3. Start the Erlang `ssh` daemon:

```
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>
```

Step 4. 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, see **Step 5a** and **Step 5b**.

Step 5a. Shut down the Erlang ssh daemon so that it stops the listener but leaves existing connections, started by the listener, operational:

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

Step 5b. Shut down the Erlang ssh daemon so that it stops the listener and all connections started by the listener:

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

1.2.4 One-Time Execution

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

> **Note:**
>
> The number of received messages in this example depends on which OS and which shell that is used on the machine running the ssh daemon. See also *ssh_connection:exec/4*.

Do a one-time execution of a remote command over ssh:

```
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>
```

```
```

Notice that 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}
...
```

1.2.5 SFTP Server

Start the Erlang `ssh` daemon with the SFTP subsystem:

```
     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>
```

1.2.6 SFTP Client

Fetch a file with the Erlang SFTP 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">>}
```

1.2.7 SFTP Client with TAR Compression and Encryption

Example of writing and then reading a tar file follows:

```
     {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 previous write and read example can be extended with encryption and decryption as follows:

```
%% First three parameters depending on which crypto type we select:
Key = <<"This is a 256 bit key. abcdefghi">>,
Ivec0 = crypto:strong_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),
```

1.2.8 Creating a Subsystem

A small ssh subsystem that echoes N bytes can be implemented as shown in the following example:

```
-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.
```

The subsystem can be run on the host **tarlop** with the generated keys, as described in Section *Running an Erlang ssh Daemon*:

```
  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).
```

See also *ssh_channel(3)*.

2 Reference Manual

The ssh application is an Erlang implementation of the Secure Shell Protocol (SSH) as defined by RFC 4250 - 4254.

SSH

Application

The ssh application is an implementation of the SSH protocol in Erlang. ssh offers API functions to write customized SSH clients and servers as well as making the Erlang shell available over SSH. An SFTP client, ssh_sftp, and server, ssh_sftpd, are also included.

DEPENDENCIES

The ssh application uses the applications *public_key* and *crypto* to handle public keys and encryption. Hence, these applications must be loaded for the ssh application to work. In an embedded environment this means that they must be started with *application:start/1,2* before the ssh application is started.

CONFIGURATION

The ssh application does not have an application- specific configuration file, as described in *application(3)*. However, by default it use the following configuration files from OpenSSH:

- known_hosts
- authorized_keys
- authorized_keys2
- id_dsa
- id_rsa
- id_ecdsa
- ssh_host_dsa_key
- ssh_host_rsa_key
- ssh_host_ecdsa_key

By default, ssh looks for id_dsa, id_rsa, id_ecdsa_key, known_hosts, and authorized_keys in ~/.ssh, and for the host key files in /etc/ssh. These locations can be changed by the options user_dir and system_dir.

Public key handling can also be customized through a callback module that implements the behaviors *ssh_client_key_api* and *ssh_server_key_api*.

Public Keys

id_dsa, id_rsa and id_ecdsa are the users private key files. Notice that the public key is part of the private key so the ssh application does not use the id_<*>.pub files. These are for the user's convenience when it is needed to convey the user's public key.

Known 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

RSA and DSA host keys are supported and are expected to be found in files named `ssh_host_rsa_key`, `ssh_host_dsa_key` and `ssh_host_ecdsa_key`.

ERROR LOGGER AND EVENT HANDLERS

The `ssh` application uses the default *OTP error logger* to log unexpected errors or print information about special events.

SUPPORTED SPECIFICATIONS AND STANDARDS

The supported SSH version is 2.0.

Algorithms

The actual set of algorithms may vary depending on which OpenSSL crypto library that is installed on the machine. For the list on a particular installation, use the command *ssh:default_algorithms/0*. The user may override the default algorithm configuration both on the server side and the client side. See the option `preferred_algorithms` in the *ssh:daemon/1,2,3* and *ssh:connect/3,4* functions.

Supported algorithms are:

Key exchange algorithms

- ecdh-sha2-nistp256
- ecdh-sha2-nistp384
- ecdh-sha2-nistp521
- diffie-hellman-group-exchange-sha1
- diffie-hellman-group-exchange-sha256
- diffie-hellman-group14-sha1
- diffie-hellman-group1-sha1

Public key algorithms

- ecdsa-sha2-nistp256
- ecdsa-sha2-nistp384
- ecdsa-sha2-nistp521
- ssh-rsa
- ssh-dss

MAC algorithms

- hmac-sha2-256
- hmac-sha2-512
- hmac-sha1

Encryption algorithms (ciphers)

- aes128-gcm@openssh.com (AEAD_AES_128_GCM)
- aes256-gcm@openssh.com (AEAD_AES_256_GCM)
- aes128-ctr
- aes192-ctr
- aes256-ctr
- aes128-cbc
- 3des-cbc

Following the internet de-facto standard, the cipher and mac algorithm AEAD_AES_128_GCM is selected when the cipher aes128-gcm@openssh.com is negotiated. The cipher and mac algorithm AEAD_AES_256_GCM is selected when the cipher aes256-gcm@openssh.com is negotiated.

See the text at the description of *the rfc 5647 further down* for more information.

Compression algorithms

- none
- zlib@openssh.com
- zlib

Unicode support

Unicode filenames are supported if the emulator and the underlaying OS support it. See section DESCRIPTION in the *file* manual page in Kernel for information about this subject.

The shell and the cli both support unicode.

Rfcs

The following rfc:s are supported:

- **RFC 4251**, The Secure Shell (SSH) Protocol Architecture.

  Except

  - 9.4.6 Host-Based Authentication
  - 9.5.2 Proxy Forwarding
  - 9.5.3 X11 Forwarding

- **RFC 4252**, The Secure Shell (SSH) Authentication Protocol.

  Except

  - 9. Host-Based Authentication: "hostbased"

- **RFC 4253**, The Secure Shell (SSH) Transport Layer Protocol.

- **RFC 4254**, The Secure Shell (SSH) Connection Protocol.

  Except

  - 6.3. X11 Forwarding
  - 7. TCP/IP Port Forwarding

- **RFC 4256**, Generic Message Exchange Authentication for the Secure Shell Protocol (SSH).

  Except

  - `num-prompts > 1`
  - password changing
  - other identification methods than userid-password

- **RFC 4419**, Diffie-Hellman Group Exchange for the Secure Shell (SSH) Transport Layer Protocol.

- **RFC 4716**, The Secure Shell (SSH) Public Key File Format.

- **RFC 5647**, AES Galois Counter Mode for the Secure Shell Transport Layer Protocol.

  There is an ambiguity in the synchronized selection of cipher and mac algorithm. This is resolved by OpenSSH in the ciphers aes128-gcm@openssh.com and aes256-gcm@openssh.com which are implemented. If the explicit

ciphers and macs AEAD_AES_128_GCM or AEAD_AES_256_GCM are needed, they could be enabled with the option preferred_algorithms.

> ### Warning:
>
> If the client or the server is not Erlang/OTP, it is the users responsibility to check that other implementation has the same interpretation of AEAD_AES_*_GCM as the Erlang/OTP SSH before enabling them. The aes*-gcm@openssh.com variants are always safe to use since they lack the ambiguity.

The second paragraph in section 5.1 is resolved as:

- If the negotiated cipher is AEAD_AES_128_GCM, the mac algorithm is set to AEAD_AES_128_GCM.
- If the negotiated cipher is AEAD_AES_256_GCM, the mac algorithm is set to AEAD_AES_256_GCM.
- If the mac algorithm is AEAD_AES_128_GCM, the cipher is set to AEAD_AES_128_GCM.
- If the mac algorithm is AEAD_AES_256_GCM, the cipher is set to AEAD_AES_256_GCM.

The first rule that matches when read in order from the top is applied

- **RFC 5656**, Elliptic Curve Algorithm Integration in the Secure Shell Transport Layer.

  Except

  - 5. ECMQV Key Exchange
  - 6.4. ECMQV Key Exchange and Verification Method Name
  - 7.2. ECMQV Message Numbers
  - 10.2. Recommended Curves

- **RFC 6668**, SHA-2 Data Integrity Verification for the Secure Shell (SSH) Transport Layer Protocol

  Comment: Defines hmac-sha2-256 and hmac-sha2-512

SEE ALSO

application(3)

ssh

Erlang module

Interface module for the `ssh` application.

See *ssh(6)* for details of supported version, algorithms and unicode support.

OPTIONS

The exact behaviour of some functions can be adjusted with the use of options which are documented together with the functions. Generally could each option be used at most one time in each function call. If given two or more times, the effect is not predictable unless explicitly documented.

The options are of different kinds:

Limits

: which alters limits in the system, for example number of simultaneous login attempts.

Timeouts

: which give some defined behaviour if too long time elapses before a given event or action, for example time to wait for an answer.

Callbacks

: which gives the caller of the function the possibility to execute own code on some events, for example calling an own logging function or to perform an own login function

Behaviour

: which changes the systems behaviour.

DATA TYPES

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

```
boolean() =
    true | false
string() =
    [byte()]
ssh_daemon_ref() =
```

opaque() - as returned by `ssh:daemon/[1,2,3]`

```
ssh_connection_ref() =
```

opaque() - as 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)*

```
key_cb() =
    atom() | {atom(), list()}
```

`atom()` - Name of the erlang module implementing the behaviours *ssh_client_key_api* or *ssh_client_key_api* as the case maybe.

`list()` - List of options that can be passed to the callback module.

```
channel_init_args() =
    list()
algs_list() =
    list( alg_entry() )
alg_entry() =
    {kex, simple_algs()} | {public_key, simple_algs()} | {cipher, double_algs()}
    | {mac, double_algs()} | {compression, double_algs()}
simple_algs() =
    list( atom() )
double_algs() =
    [{client2serverlist,simple_algs()},{server2client,simple_algs()}]        |
    simple_algs()
```

Exports

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

Types:

`ConnectionRef = ssh_connection_ref()`

Closes an SSH connection.

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

Types:

`Host = string()`

`Port = integer()`

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

`Options = [{Option, Value}]`

`Timeout = infinity | integer()`

Negotiation time-out in milli-seconds. The default value is `infinity`. For connection time-out, use option `{connect_timeout, timeout()}`.

`TcpSocket = port()`

The socket is supposed to be from *gen_tcp:connect* or *gen_tcp:accept* with option `{active,false}`

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

Options:

`{inet, inet | inet6}`

IP version to use.

`{user_dir, string()}`

Sets the user directory, that is, 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 the server to 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`. However, those optins are not always desirable to use from a security point of view.

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

Provides a fun to implement your own logging when a server disconnects the client.

`{unexpectedfun, fun(Message:term(), Peer) -> report | skip }`

Provides a fun to implement your own logging or other action when an unexpected message arrives. If the fun returns `report` the usual info report is issued but if `skip` is returned no report is generated.

`Peer` is in the format of `{Host,Port}`.

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

> **Note:**
>
> This option will be removed in OTP 20, but is kept for compatibility. It is ignored if the preferred `pref_public_key_algs` option is used.

Sets the preferred public key algorithm to use for user authentication. If the preferred algorithm fails, the other algorithm is tried. If `{public_key_alg, 'ssh-rsa'}` is set, it is translated to `{pref_public_key_algs, ['ssh-rsa','ssh-dss']}`. If it is `{public_key_alg, 'ssh-dss'}`, it is translated to `{pref_public_key_algs, ['ssh-dss','ssh-rsa']}`.

`{pref_public_key_algs, list()}`

List of user (client) public key algorithms to try to use.

The default value is `['ssh-rsa','ssh-dss','ecdsa-sha2-nistp256','ecdsa-sha2-nistp384','ecdsa-sha2-nistp521']`

If there is no public key of a specified type available, the corresponding entry is ignored.

`{preferred_algorithms, algs_list()}`

List of algorithms to use in the algorithm negotiation. The default `algs_list()` can be obtained from *default_algorithms/0*.

If an alg_entry() is missing in the algs_list(), the default value is used for that entry.

Here is an example of this option:

```
{preferred_algorithms,
 [{public_key,['ssh-rsa','ssh-dss']},
  {cipher,[{client2server,['aes128-ctr']},
           {server2client,['aes128-cbc','3des-cbc']}]},
  {mac,['hmac-sha2-256','hmac-sha1']},
  {compression,[none,zlib]}
  ]
}
```

The example specifies different algorithms in the two directions (client2server and server2client), for cipher but specifies the same algorithms for mac and compression in both directions. The kex (key exchange) is implicit but public_key is set explicitly.

> ### Warning:
>
> Changing the values can make a connection less secure. Do not change unless you know exactly what you are doing. If you do not understand the values then you are not supposed to change them.

`{dh_gex_limits,{Min=integer(),I=integer(),Max=integer()}}`

Sets the three diffie-hellman-group-exchange parameters that guides the connected server in choosing a group. See RFC 4419 for the function of thoose. The default value is `{1024, 6144, 8192}`.

`{connect_timeout, timeout()}`

Sets a time-out on the transport layer connection. For `gen_tcp` the time is in milli-seconds and the default value is `infinity`.

`{user, string()}`

Provides a username. If this option is not given, `ssh` reads from the environment (`LOGNAME` or `USER` on UNIX, `USERNAME` on Windows).

`{password, string()}`

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

`{key_cb, key_cb()}`

Module implementing the behaviour *ssh_client_key_api*. Can be used to customize the handling of public keys. If callback options are provided along with the module name, they are made available to the callback module via the options passed to it under the key 'key_cb_private'.

`{quiet_mode, atom() = boolean()}`

If `true`, the client does not print anything on authorization.

`{id_string, random | string()}`

The string that the client presents to a connected server initially. The default value is "Erlang/VSN" where VSN is the ssh application version number.

The value `random` will cause a random string to be created at each connection attempt. This is to make it a bit more difficult for a malicious peer to find the ssh software brand and version.

`{fd, file_descriptor()}`

Allows an existing file descriptor to be used (by passing it on to the transport protocol).

`{rekey_limit, integer()}`

Provides, in bytes, when rekeying is to be initiated. Defaults to once per each GB and once per hour.

`{idle_time, integer()}`

Sets a time-out on a connection when no channels are active. Defaults to `infinity`.

```
{ssh_msg_debug_fun, fun(ConnectionRef::ssh_connection_ref(),
AlwaysDisplay::boolean(), Msg::binary(), LanguageTag::binary()) -> _}
```

Provide a fun to implement your own logging of the SSH message SSH_MSG_DEBUG. The last three parameters are from the message, see RFC4253, section 11.3. The `ConnectionRef` is the reference to the connection on which the message arrived. The return value from the fun is not checked.

The default behaviour is ignore the message. To get a printout for each message with `AlwaysDisplay = true`, use for example `{ssh_msg_debug_fun, fun(_,true,M,_)-> io:format("DEBUG: ~p~n", [M]) end}`

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

Types:

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

The socket is supposed to be from *gen_tcp:connect* or *gen_tcp:accept* with option `{active,false}`

Starts a server listening for SSH connections on the given port. If the `Port` is 0, a random free port is selected. See *daemon_info/1* about how to find the selected port number.

Options:

`{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 specification is retrieved by calling `ssh_sftpd:subsystem_spec/1`. If the subsystems option is not present, the value of `[ssh_sftpd:subsystem_spec([])]` is used. The option can be set to the empty list if you do not want the daemon to run any subsystems.

```
{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. The default is to use the Erlang shell: `{shell, start, []}`

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

Provides your own CLI implementation, that is, a channel callback module that implements a shell and command execution. The shell read-eval-print loop can be customized, using the option `shell`. This means less work than implementing an own CLI channel. If set to `no_cli`, the CLI channels are disabled and only subsystem channels are allowed.

`{user_dir, string()}`

Sets the user directory. That is, 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 identify the host keys for `ssh`. Defaults to `/etc/ssh`. For security reasons, this directory is normally accessible only to the root user.

`{auth_methods, string()}`

Comma-separated string that determines which authentication methods that the server is to support and in what order they are tried. Defaults to `"publickey,keyboard-interactive,password"`

```
{auth_method_kb_interactive_data, PromptTexts}
where:
PromptTexts = kb_int_tuple() | fun(Peer::{IP::tuple(),Port::integer()},
User::string(), Service::string()) -> kb_int_tuple()
kb_int_tuple() = {Name::string(), Instruction::string(), Prompt::string(),
Echo::boolean()}
```

Sets the text strings that the daemon sends to the client for presentation to the user when using `keyboar-interactive` authentication. If the fun/3 is used, it is called when the actual authentication occurs and may therefore return dynamic data like time, remote ip etc.

The parameter `Echo` guides the client about need to hide the password.

The default value is: `{auth_method_kb_interactive_data, {"SSH server", "Enter password for \""++User++"\"", "password: ", false}>`

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

Provides passwords for password authentication. The passwords are used when someone tries to connect to the server and public key user-authentication fails. The option provides a list of valid usernames and the corresponding passwords.

`{password, string()}`

Provides a global password that authenticates any user. From a security perspective this option makes the server very vulnerable.

`{preferred_algorithms, algs_list()}`

List of algorithms to use in the algorithm negotiation. The default `algs_list()` can be obtained from *default_algorithms/0*.

If an alg_entry() is missing in the algs_list(), the default value is used for that entry.

Here is an example of this option:

```
{preferred_algorithms,
 [{public_key,['ssh-rsa','ssh-dss']},
  {cipher,[{client2server,['aes128-ctr']},
           {server2client,['aes128-cbc','3des-cbc']}]},
  {mac,['hmac-sha2-256','hmac-sha1']},
  {compression,[none,zlib]}
  ]
}
```

The example specifies different algorithms in the two directions (client2server and server2client), for cipher but specifies the same algorithms for mac and compression in both directions. The kex (key exchange) is implicit but public_key is set explicitly.

> ### Warning:
>
> Changing the values can make a connection less secure. Do not change unless you know exactly what you are doing. If you do not understand the values then you are not supposed to change them.

`{dh_gex_groups, [{Size=integer(),G=integer(),P=integer()}] | {file,filename()} {ssh_moduli_file,filename()} }`

Defines the groups the server may choose among when diffie-hellman-group-exchange is negotiated. See RFC 4419 for details. The three variants of this option are:

`{Size=integer(),G=integer(),P=integer()}`
: The groups are given explicitly in this list. There may be several elements with the same `Size`. In such a case, the server will choose one randomly in the negotiated Size.

`{file,filename()}`
: The file must have one or more three-tuples `{Size=integer(),G=integer(),P=integer()}` terminated by a dot. The file is read when the daemon starts.

`{ssh_moduli_file,filename()}`
: The file must be in *ssh-keygen moduli file format*. The file is read when the daemon starts.

The default list is fetched from the *public_key* application.

`{dh_gex_limits,{Min=integer(),Max=integer()}}`

Limits what a client can ask for in diffie-hellman-group-exchange. The limits will be `{MaxUsed = min(MaxClient,Max), MinUsed = max(MinClient,Min)}` where `MaxClient` and `MinClient` are the values proposed by a connecting client.

The default value is `{0,infinity}`.

If `MaxUsed < MinUsed` in a key exchange, it will fail with a disconnect.

See RFC 4419 for the function of the Max and Min values.

```
{pwdfun, fun(User::string(), Password::string(), PeerAddress::
{ip_adress(),port_number()}, State::any()) -> boolean() | disconnect |
{boolean(),any()} }
```

Provides a function for password validation. This could used for calling an external system or if passwords should be stored as a hash. The fun returns:

- `true` if the user and password is valid and
- `false` otherwise.

This fun can also be used to make delays in authentication tries for example by calling *timer:sleep/1*. To facilitate counting of failed tries the `State` variable could be used. This state is per connection only. The first time the pwdfun is called for a connection, the `State` variable has the value `undefined`. The pwdfun can return - in addition to the values above - a new state as:

- `{true, NewState:any()}` if the user and password is valid or
- `{false, NewState:any()}` if the user or password is invalid

A third usage is to block login attempts from a missbehaving peer. The `State` described above can be used for this. In addition to the responses above, the following return value is introduced:

- `disconnect` if the connection should be closed immediately after sending a SSH_MSG_DISCONNECT message.

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

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

This option (`{pwdfun,fun/2}`) is the same as a subset of the previous (`{pwdfun,fun/4}`). It is kept for compatibility.

`{negotiation_timeout, integer()}`

Maximum time in milliseconds for the authentication negotiation. Defaults to 120000 (2 minutes). If the client fails to log in 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. Thus, if set to `N`, and `N` clients have connected but not started the login process, connection attempt `N+1` is aborted. If `N` connections are authenticated and still logged in, no more logins are accepted until one of the existing ones log out.

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

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

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

`{max_channels, pos_integer()}`

The maximum number of channels with active remote subsystem that are accepted for each connection to this daemon

By default, this 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 at a time. If set to true, an unlimited number of login attempts are allowed simultaneously.

If the `max_sessions` option is set to `N` and `parallel_login` is set to `true`, the maximum 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, by the `max_sessions` option or a firewall configuration. If set to `true`, there is no protection against DOS attacks.

`{minimal_remote_max_packet_size, non_negative_integer()}`

The least maximum packet size that the daemon will accept in channel open requests from the client. The default value is 0.

`{id_string, random | string()}`

The string the daemon will present to a connecting peer initially. The default value is "Erlang/VSN" where VSN is the ssh application version number.

The value `random` will cause a random string to be created at each connection attempt. This is to make it a bit more difficult for a malicious peer to find the ssh software brand and version.

`{key_cb, key_cb()}`

Module implementing the behaviour *ssh_server_key_api*. Can be used to customize the handling of public keys. If callback options are provided along with the module name, they are made available to the callback module via the options passed to it under the key 'key_cb_private'.

`{profile, atom()}`

Used together with `ip-address` and `port` to uniquely identify a ssh daemon. This can be useful in a virtualized environment, where there can be more that one server that has the same `ip-address` and `port`. If this property is not explicitly set, it is assumed that the the `ip-address` and `port` uniquely identifies the SSH daemon.

`{fd, file_descriptor()}`

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

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

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

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

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

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

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

`{unexpectedfun, fun(Message:term(), Peer) -> report | skip }`

Provides a fun to implement your own logging or other action when an unexpected message arrives. If the fun returns `report` the usual info report is issued but if `skip` is returned no report is generated.

`Peer` is in the format of `{Host,Port}`.

```
{ssh_msg_debug_fun, fun(ConnectionRef::ssh_connection_ref(),
AlwaysDisplay::boolean(), Msg::binary(), LanguageTag::binary()) -> _}
```

Provide a fun to implement your own logging of the SSH message SSH_MSG_DEBUG. The last three parameters are from the message, see RFC4253, section 11.3. The `ConnectionRef` is the reference to the connection on which the message arrived. The return value from the fun is not checked.

The default behaviour is ignore the message. To get a printout for each message with `AlwaysDisplay = true`, use for example `{ssh_msg_debug_fun, fun(_,true,M,_)-> io:format("DEBUG: ~p~n", [M]) end}`

`daemon_info(Daemon) -> {ok, [{port,Port}]} | {error,Error}`

Types:

`Port = integer()`

`Error = bad_daemon_ref`

Returns a key-value list with information about the daemon. For now, only the listening port is returned. This is intended for the case the daemon is started with the port set to 0.

`default_algorithms() -> algs_list()`

Returns a key-value list, where the keys are the different types of algorithms and the values are the algorithms themselves. An example:

```
20> ssh:default_algorithms().
[{kex,['diffie-hellman-group1-sha1']},
 {public_key,['ssh-rsa','ssh-dss']},
 {cipher,[{client2server,['aes128-ctr','aes128-cbc','3des-cbc']},
          {server2client,['aes128-ctr','aes128-cbc','3des-cbc']}]},
 {mac,[{client2server,['hmac-sha2-256','hmac-sha1']},
       {server2client,['hmac-sha2-256','hmac-sha1']}]},
 {compression,[{client2server,[none,zlib]},
               {server2client,[none,zlib]}]}]
21>
```

`shell(Host) ->`

`shell(Host, Option) ->`

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

`shell(TcpSocket) -> _`

Types:

`Host = string()`

`Port = integer()`

`Options - see ssh:connect/3`

TcpSocket = port()

The socket is supposed to be from *gen_tcp:connect* or *gen_tcp:accept* with option `{active,false}`

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

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

Types:

Type = permanent | transient | temporary

Reason = term()

Utility function that starts the applications `crypto`, `public_key`, and `ssh`. Default type is `temporary`. For more information, see the *application(3)* manual page in Kernel.

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

Types:

Reason = term()

Stops the `ssh` application. For more information, see the *application(3)* manual page in Kernel.

```
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 operational.

ssh_channel

Erlang module

SSH services (clients and servers) are implemented as channels that are multiplexed over an SSH connection and communicates over the **SSH Connection Protocol**. This module provides a callback API that takes care of generic channel aspects, such as flow control and close messages. It 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 an `ssh` subsystem, use `-behaviour(ssh_daemon_channel)` instead of `-behaviour(ssh_channel)`. The reason is that 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 a limited version of the `ssh_channel` behaviour.

DATA TYPES

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

`boolean() =`

`true | false`

`string() =`

list of ASCII characters

`timeout() =`

`infinity | integer()` in milliseconds

`ssh_connection_ref() =`

opaque() -as 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 the 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 *ssh_channel: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 time-out occurs. The channel calls *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 *ssh_channel: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 calls *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 enters 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 the *proc_lib(3)* manual page in STDLIB. The user is responsible for any initialization of the process and must 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 `init` function of the callback module.

`{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 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()`

`Reply = term()`

This function can be used by a channel to 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 is 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 *ssh_connection:session_channel/[2,4]*.

`ChannelCb = atom()`

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 process that handles an SSH channel. It is called internally, by the `ssh` daemon, or explicitly by the `ssh` client implementations. The behavior sets the `trap_exit` flag to `true`.

CALLBACK TIME-OUTS

The time-out values that can be returned by the callback functions have the same semantics as in a *gen_server*. If the time-out occurs, *handle_msg/2* is 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()`

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 is to update its internal state during a release upgrade or downgrade, that is, when the instruction `{update,Module,Change,...}`, where `Change={advanced,Extra}`, is given in the `appup` file. For more information, refer to Section 9.11.6 Release Handling Instructions in the *System Documentation*.

> **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. The subsystem channels can be upgraded when upgrading the user application, if the callback functions can handle two versions of the state, but this function cannot 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 time-outs, see Section *CALLBACK TIME-OUTS*.

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

Types:

Msg = term()

From = opaque()

Is to 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 time-outs,, see Section *CALLBACK TIME-OUTS*.

`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 time-outs, see Section *CALLBACK TIME-OUTS*.

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

Types:

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

Handles other messages than SSH Connection Protocol, call, or cast messages sent to the channel.

Possible Erlang 'EXIT' messages is to be handled by this function and all channels are to handle the following message.

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

This is the first message that the channel receives. 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, 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. For details, see *ssh_connection:event()*.

The following message is taken care of by the `ssh_channel` behavior.

`{closed, ssh_channel_id()}`

The channel behavior sends a close message to the other side, if such a message has not already been sent. Then it terminates 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* is called, if it has not been called earlier. This function does 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, that is, SSH channels, to communicate over the SSH connection. The API functions in this module send SSH Connection Protocol events, which are received as messages by the remote channel. If the receiving channel is an Erlang process, the messages have the format `{ssh_cm, ssh_connection_ref(), ssh_event_msg()}`. If the *ssh_channel* behavior is used to implement the channel process, these messages are handled by *handle_ssh_msg/2*.

DATA TYPES

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

`boolean() =`

`true | false`

`string() =`

list of ASCII characters

`timeout() =`

`infinity | integer()` in milliseconds

`ssh_connection_ref() =`

opaque() -as 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 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()`

`reason() =`

`timeout | closed`

data_events()

`{data, ssh_channel_id(), ssh_data_type_code(), Data :: binary()}`

Data has arrived on the channel. This event is sent as a result of calling *ssh_connection:send/[3,4,5]*.

`{eof, ssh_channel_id()}`

Indicates that the other side sends no more data. This event is sent as a 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 do not support signals, in which case they are to ignore this message. There is currently no function to generate this event as the signals referred to are on OS-level and not something generated by an Erlang program.

```
{exit_signal, ssh_channel_id(), ExitSignal :: string(),
ErrorMsg ::string(), LanguageString :: string()}
```

A remote execution can terminate violently because of a signal. Then this message can be received. For details on valid string values, see **RFC 4254** Section 6.10, which shows a special case of these signals.

`{exit_status, ssh_channel_id(), ExitStatus :: integer()}`

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 a result of calling *ssh_connection:exit_status/3*.

`{closed, ssh_channel_id()}`

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

terminal_events()

Channels implementing a shell and command execution on the server side are to handle the following messages that can be sent by client- channel processes.

Events that include a `WantReply` expect 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(), WantReply :: boolean(), Var ::string(), Value ::
string()}
```

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

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

A pseudo-terminal has been requested for the session. `Terminal` is the value of the TERM environment variable value, that is, `vt100`. Zero dimension parameters must be ignored. The character/row dimensions override the pixel dimensions (when non-zero). Pixel dimensions refer to the drawable area of the window. `Opcode` in the `TerminalModes` list is the mnemonic name, represented as a lowercase Erlang atom, defined in **RFC 4254**, Section 8. It can 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 a result of calling *ssh_connection:ptty_alloc/4*.

`{shell, WantReply :: boolean()}`

This message requests that the user default shell is started at the other end. This event is sent as a result of calling *ssh_connection:shell/2*.

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

When the window (terminal) size changes on the client side, it **can** send a message to the server side to inform it of the new dimensions. No API function generates this event.

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

This message requests that the server starts execution of the given command. This event is sent as a 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 flow control window. This is to be done by both the client- and server-side channel processes.

> ### Note:
>
> Channels implemented with the *ssh_channel* behavior do not normally need to call this function as flow control is handled by the behavior. The behavior adjusts the window every time the callback *handle_ssh_msg/2* returns 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 is called by the `ssh_channel` behavior when the channel is terminated, see *ssh_channel(3)*. Thus, channels implemented with the behavior are not to call this function explicitly.

`exec(ConnectionRef, ChannelId, Command, TimeOut) -> ssh_request_status() | {error, reason()}`

Types:

`ConnectionRef = ssh_connection_ref()`
`ChannelId = ssh_channel_id()`
`Command = string()`
`Timeout = timeout()`

Is to be called by a client-channel process to request that the server starts executing the given command. The result is several messages according to the following pattern. The last message is 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(), Data :: binary()}}
```

The result of executing the command can 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 is to be sent.

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

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(),
ExitStatus :: integer()}}
```

It is recommended by the SSH Connection Protocol to send this message, but that is not always 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 shut down.

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

Types:

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

Is to be called by a server-channel process to send the exit status of a command to the client.

```
ptty_alloc(ConnectionRef, ChannelId, Options) ->
ptty_alloc(ConnectionRef, ChannelId, Options, Timeout) -> >
ssh_request_status() | {error, reason()}
```

Types:

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

Sends an SSH Connection Protocol `pty_req`, to allocate a pseudo-terminal. Is to be called by an SSH client process.

Options:

{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 can 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 it wants a status report, that is, `WantReply = true`. If `WantReply` is `false`, calling this function becomes a "noop". Is to 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()

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

The function *subsystem/4* and subsequent calls of `send/3,4,5` must be executed in the same process.

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

Types:

ConnectionRef = ssh_connection_ref()
ChannelId = ssh_channel_id()

Sends EOF on 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 functions in this module.

```
setenv(ConnectionRef, ChannelId, Var, Value, TimeOut) -> ssh_request_status()
| {error, reason()}
```

Types:

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

Environment variables can be passed before starting the shell/command. Is to be called by a client channel processes.

```
shell(ConnectionRef, ChannelId) -> ssh_request_status() | {error, closed}
```

Types:

ConnectionRef = ssh_connection_ref()
ChannelId = ssh_channel_id()

Is to be called by a client channel process to request that the user default shell (typically defined in /etc/passwd in Unix systems) is executed at the server end.

```
subsystem(ConnectionRef, ChannelId, Subsystem, Timeout) ->
ssh_request_status() | {error, reason()}
```

Types:

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

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

The function `subsystem/4` and subsequent calls of *send/3,4,5* must be executed in the same process.

ssh_client_key_api

Erlang module

Behavior describing the API for public key handling of an SSH client. By implementing the callbacks defined in this behavior, the public key handling of an SSH client can be customized. By default the `ssh` application implements this behavior with help of the standard OpenSSH files, see the *ssh(6)* application manual.

DATA TYPES

Type definitions that are used more than once in this module, or abstractions to indicate the intended use of the data type, or both. For more details on public key data types, refer to Section 2 Public Key Records in 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. Is to support `'ssh-rsa'| 'ssh-dss'`, but more 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. Is to support `'ssh-rsa'| 'ssh-dss'` but more algorithms can be handled.

`ConnectOptions = proplists:proplist()`

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

`PrivateKey = private_key()`

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 public key handling of an SSH server. By implementing the callbacks defined in this behavior, the public key handling of an SSH server can be customized. By default the SSH application implements this behavior with help of the standard OpenSSH files, see the *ssh(6)* application manual.

DATA TYPES

Type definitions that are used more than once in this module, or abstractions to indicate the intended use of the data type, or both. For more details on public key data types, refer to Section 2 Public Key Records in 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. Is to support `'ssh-rsa' | 'ssh-dss'`, but more algorithms can be handled.

`DaemonOptions = proplists:proplist()`

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

`Key = private_key()`

Private key of the host matching the `Algorithm`.

`Reason = term()`

Fetches the private key of the host.

`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()`

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 SSH FTP (SFTP) 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, or abstractions to indicate the intended use of the data type, or both:

`reason()`

= `atom()` A description of the reason why an operation failed.

The value is formed from the sftp error codes in the protocol-level responses as defined in **draft-ietf-secsh-filexfer-13.txt** section 9.1.

The codes are named as `SSH_FX_*` which are transformed into lowercase of the star-part. E.g. the error code `SSH_FX_NO_SUCH_FILE` will cause the `reason()` to be `no_such_file`.

`ssh_connection_ref() =`

`opaque()` - as returned by *`ssh:connect/3`*

`timeout()`

= `infinity | integer()` in milliseconds. Default infinity.

Time-outs

If the request functions for the SFTP channel return `{error, timeout}`, no answer was received from the server within the expected time.

The request may have reached the server and may have been performed. However, no answer was received from the server within the expected time.

Exports

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

Types:

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

The `apread/4` function reads from a specified position, combining the *`position/3`* and *`aread/3`* functions.

```
apwrite(ChannelPid, Handle, Position, Data) -> {async, N} | {error, reason()}
```

Types:

```
ChannelPid = pid()
Handle = term()
Position = integer()
```

Len = integer()
Data = binary()
Timeout = timeout()
N = term()

The `apwrite/4` function writes to a specified position, combining the *`position/3`* and *`awrite/3`* functions.

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

Types:

ChannelPid = pid()
Handle = term()
Position = integer()
Len = integer()
N = 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}`, `eof`, or `{error, reason()}`.

```
awrite(ChannelPid, Handle, Data) -> {async, N} | {error, reason()}
```

Types:

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

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, reason()}`.

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

Types:

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

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

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

Types:

ChannelPid = pid()
Name = string()

Timeout = timeout()

Deletes the file specified by `Name`.

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

Types:

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

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

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

Types:

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

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

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

Types:

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

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

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

Types:

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

Creates a symbolic link pointing to `Target` with the name `Name`.

```
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()
```

Opens a file on the server and returns a handle, which 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()
```

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()
```

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.

For code exampel see Section *SFTP Client with TAR Compression and Encryption* in the ssh Users Guide.

The `crypto` mode option is applied to the generated stream of bytes prior to sending them to the SFTP server. This is intended for encryption but can 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 intended for stream crypto, whereas a fixed `ChunkSize` is intended for block crypto. `ChunkSize`s can be changed in the return from the `EncryptFun` or `DecryptFun`. The value can be changed between `pos_integer()` and `undefined`.

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

Types:

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

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 eariler with `Offset` 0, that is, `{bof, 0} | {cur, 0} | {eof, 0}`.

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

Types:

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

The `pread/3,4` function reads from a specified position, combining the *`position/3`* and *`read/3,4`* functions.

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

Types:

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

The `pwrite/3,4` function writes to a specified position, combining the *`position/3`* and *`write/3,4`* functions.

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

Types:

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

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 are read and returned. If no bytes are read, `eof` is returned.

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

Types:

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

Reads a file from the server, and returns the data in a binary.

```
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()

Returns a `file_info` record from the file specified by `Name` or `Handle`. See *file:read_file_info/2* for information about the record.

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

Types:

`ChannelPid = pid()`
`Name = string()`
`Target = string()`

Reads the link target from the symbolic link specified by `name`.

```
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()`

Returns a `file_info` record from the symbolic link specified by `Name` or `Handle`. See *file:read_link_info/2* for information about the record.

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

Types:

`ChannelPid = pid()`
`OldName = string()`
`NewName = string()`
`Timeout = timeout()`

Renames a file named `OldName` and gives it the name `NewName`.

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

Types:

`Host = string()`
`ConnectionRef = ssh_connection_ref()`
`Port = integer()`

TcpSocket = port()

The socket is supposed to be from *gen_tcp:connect* or *gen_tcp:accept* with option `{active,false}`

Options = [{Option, Value}]

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 is to be used as input to all other API functions in this module.

Options:

`{timeout, timeout()}`

The time-out is passed to the `ssh_channel` start function, and defaults to `infinity`.

`{sftp_vsn, integer()}`

Desired SFTP protocol version. The actual version is 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 connection. Use *ssh:close/1* to close it.

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

Types:

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

Writes `data` to the file referenced by `Handle`. The file is to be opened with `write` or `append` flag. Returns `ok` if successful or `{error, reason()}` otherwise.

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

Types:

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

Writes a file to the server. The file is created if it does not exist but overwritten if it exists.

```
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()
```

Writes file information from a `file_info` record to the file specified by `Name`. See *file:write_file_info/[2,3]* for information about the record.

ssh_sftpd

Erlang module

Specifies a channel process to handle an 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 the *ssh_channel(3)* manual page.

```
channel_init_args() =
```

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

Exports

```
subsystem_spec(Options) -> subsystem_spec()
```

Types:

`Options = [{Option, Value}]`

Is to be used together with `ssh:daemon/[1,2,3]`

Options:

`{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`, which uses the *file* and *filelib* APIs to access the standard OTP file server. This option can 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. Then the user cannot see any files above this root. If, for example, the root directory is set to `/tmp`, then the user sees this directory as `/`. If the user then writes `cd /etc`, the user moves to `/tmp/etc`.

`{sftpd_vsn, integer()}`

Sets the SFTP version to use. Defaults to 5. Version 6 is under development and limited.