SSH

SSH 2.1.1
September 10 2012

September 10 2012

1 Reference Manual

The SSH application is an erlang implementation of the secure shell protocol.

ssh

Erlang module

Interface module for the SSH application.

SSH

- ssh requires the crypto and public_key applications.
- Supported SSH-version is 2.0
- Currently supports only a minimum of mac and encryption algorithms i.e. hmac-sha1, and aes128-cb and 3des-cbc.

COMMON DATA TYPES

Type definitions that are used more than once in this module:

`boolean() = true | false`

`string() = list of ASCII characters`

`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() - {N1,N2,N3,N4} % IPv4 | {K1,K2,K3,K4,K5,K6,K7,K8} % IPv6`

`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 a 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 registered port for SSH.

`Options = [{Option, Value}]`

`Timeout = infinity | integer(milliseconds)`

Connects to an SSH server. No channel is started this is done by calling ssh_connect:session_channel/2.

Options are:

`{user_dir, string()}`

Sets the user directory e.i. 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 pass phrase it can be supplied with this option.

`{rsa_pass_phrase, string()}`

If the user rsa key is protected by a pass phrase 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_dsa}`

Sets the preferred public key algorithm to use for user authentication. If the the preferred algorithm fails of some reason, the other algorithm is tried. The default is to try `ssh_rsa` first.

`{connect_timeout, timeout()}`

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

`{user, String}`

Provide 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.

`{user_auth, Fun/3}`

Provide a fun for password authentication. The fun will be called as `fun(User, Password, Opts)` and should return `true` or `false`.

`{key_cb, atom() = KeyCallbackModule}`

Provide a special call-back module for key handling. The call-back module should be modeled after the `ssh_file` module. The functions that must be exported are: `private_host_rsa_key/2`, `private_host_dsa_key/2`, `lookup_host_key/3` and `add_host_key/3`. This is considered somewhat experimental and will be better documented later on.

`{fd, file_descriptor()}`

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

`{ip_v6_disabled, boolean()}`

Determines if SSH shall use IPv6 or not.

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

Types:

Option = client_version | server_version | peer

Value = term()

Retrieves information about a connection.

daemon(Port) ->

daemon(Port, Options) ->

daemon(HostAddress, Port, Options) -> ssh_daemon_ref()

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:

`{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. Example use the erlang shell: `{shell, start, []}` which is the default behavior.

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

Provide your own cli implementation, e.i. 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.

`{user_dir, String}`

Sets the user directory e.i. 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 files that identifies the host for ssh. The default is `/etc/ssh`, note that SSH normally requires the host files there to be readable only by root.

`{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/2}`

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.

`{fd, file_descriptor()}`

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

`{ip_v6_disabled, boolean()}`

Determines if SSH shall use IPv6 or not (only used when HostAddress is set to any).

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

Types:

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

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

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

Types:

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

Starts the Ssh application. Default type is temporary. See also *application(3)* Requires that the crypto application has been started.

stop() -> ok

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 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 specific parts.

COMMON 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 a 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 `CallbackModule: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 `CallbackModule:handle_cast/2` to handle the message.

enter_loop(State) -> _

Types:

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

Makes an existing process into a `ssh_channel` process. Does not return, instead the calling process will enter the `ssh_channel` process receive loop and become a `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}]

The following options must be present:

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

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

`{cm, connection_ref()}`
: Reference to the ssh connection.

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

> Note:
>
> This function is normally not called by the user, it is only needed if for some reason the channel process needs to be started with help of `proc_lib` instead 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 `CallbackModule: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 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 a ssh channel. Will be called internally by the ssh daemon or explicitly by the ssh client implementations. A channel process traps exit signals by default.

CALLBACK FUNCTIONS

The functions init/1, terminate/2, handle_ssh_msg/2 and handle_msg/2 are the functions that are required to provide the implementation for a server side channel, such as a ssh subsystem channel that can be plugged into the erlang ssh daemon see *ssh:daemon/[2, 3]*. The handle_call/3, handle_cast/2 code_change/3 and enter_loop/1 functions are only relevant when implementing a client side channel.

CALLBACK TIMEOUTS

If an integer timeout value is provided in a return value of one of the callback functions, a timeout will occur unless a message is received within `Timeout` milliseconds. A timeout is represented by the atom `timeout` which should be handled by the *handle_msg/2* callback function. The atom infinity can be used to wait indefinitely, this is the default value.

Exports

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

Types:

`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. Any new connection will benefit from a server side upgrade but already started connections on the server side will not be affected.

> **Note:**
>
> If there are long lived ssh connections and more than one upgrade in a short time this may cause the old connections to fail as only two versions of the code may be loaded simultaneously.

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` is the internal state of the channel.

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

The function should return the updated internal state.

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

Types:

`Args = term()`

Last argument to ssh_channel:start_link/4.

State = term()

Timeout = timeout()

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*.

CallbackModule: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]

Timeout = timeout()

NewState = term() - a possible updated version of State

Reason = term()

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

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

CallbackModule:handle_cast(Msg, State) -> Result

Types:

Msg = term()

State = term()

Result = {noreply, NewState} | {noreply, NewState, Timeout} | {stop, Reason, NewState}

NewState = term() - a possible updated version of State

Timeout = timeout()

Reason = term()

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

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

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

Types:

Msg = timeout | term()

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 the client. If the message is not useful for your particular problem just ignore it by immediately returning {ok, State}.

`CallbackModule:handle_ssh_msg(Msg, State) -> {ok, State} | {stop, ssh_channel_id(), State}`

Types:

`Msg = {ssh_cm, ssh_connection_ref(), SshMsg}`

`SshMsg = tuple() - see message list below`

`State = term()`

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

All channels should handle the following messages. For channels implementing subsystems the handle_ssh_msg-callback will not be called for any other messages.

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

Data has arrived on the channel. When the callback for this message returns the channel behavior will adjust the ssh flow control window.

`{ssh_cm, ssh_connection_ref(), {eof, ssh_channel_id()}}`

Indicteas that the other side will not send any more data.

`{ssh_cm, ssh_connection_ref(), {signal, ssh_channel_id(), ssh_signal()}}`

A signal can be delivered to the remote process/service using the following message. Some systems may not implement signals, in which case they should ignore this message.

`{ssh_cm, ssh_connection_ref(), {exit_signal, ssh_channel_id(), string() = exit_signal, 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

`{ssh_cm, ssh_connection_ref(), {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.

Channels implementing a shell and command execution on the server side should also handle the following messages.

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

Environment variables may be passed to the shell/command to be started later. Note that before the callback returns it should call the function ssh_connection:reply_request/4 with the boolean value of `WantReply` as the second argument.

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

This message will request that the server start the execution of the given command. Note that before the callback returns it should call the function ssh_connection:reply_request/4 with the boolean value of `WantReply` as the second argument.

`{ssh_cm, ssh_connection_ref(), {pty, ssh_channel_id(), boolean() = WantReply, {string() = Terminal, integer() = CharWidth, integer() = RowHeight, integer() = PixelWidth, integer() = PixelHight, [{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, or the opcode if the mnemonic name is not listed in the RFC. Example `OP code: 53, mnemonic name ECHO erlang atom: echo`. Note that before the callback returns it should call the function ssh_connection:reply_request/4 with the boolean value of `WantReply` as the second argument.

`{ssh_cm, ConnectionRef, {shell, boolean() = WantReply}}`

This message will request that the user's default shell be started at the other end. Note that before the callback returns it should call the function ssh_connection:reply_request/4 with the value of `WantReply` as the second argument.

```
{ssh_cm, ssh_connection_ref(), {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 other side to inform it of the new dimensions.

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

`{ssh_cm, ssh_connection_ref(), {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.

`CallbackModule: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 be the opposite of `CallbackModule:init/1` and do any necessary cleaning up. When it returns, the channel process terminates with reason `Reason`. The return value is ignored.

ssh_connection

Erlang module

This module provides an API to the ssh connection protocol. Not all features of the connection protocol are officially supported yet. Only the ones supported are documented here.

COMMON 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 a 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
```

MESSAGES SENT TO CHANNEL PROCESSES

As a result of the ssh connection protocol messages on the form `{ssh_cm, ssh_connection_ref(), term()}` will be sent to a channel process. The term will contain information regarding the ssh connection protocol event, for details see the ssh channel behavior callback *handle_ssh_msg/2*

Exports

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

Types:

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

Adjusts the ssh flowcontrol window.

> ### Note:
>
> This will be taken care of by the ssh_channel behavior when the callback *handle_ssh_msg/2* has returned after processing a {ssh_cm, ssh_connection_ref(), {data, ssh_channel_id(), ssh_data_type_code(), binary()}} message, and should normally not be called explicitly.

`close(ConnectionRef, ChannelId) -> ok`

Types:

ConnectionRef = ssh_connection_ref()
ChannelId = ssh_channel_id()

Sends a close message on the channel `ChannelId`

> Note:
>
> This function will be called by the ssh channel behavior when the channel is terminated see *ssh_channel(3)* and should normally not be called explicitly.

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

Types:

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

Will request that the server start the execution of the given command, the result will be received as:

`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.

`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.

These message should be handled by the client. The *ssh channel behavior* can be used when writing a client.

exit_status(ConnectionRef, ChannelId, Status) -> ok

Types:

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

Sends the exit status of a command to the client.

reply_request(ConnectionRef, WantReply, Status, CannelId) -> 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 after handling an ssh connection protocol message containing a `WantReply` boolean value. See the ssh_channel behavior callback *handle_ssh_msg/2*

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

Types:

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

Sends channel data.

send_eof(ConnectionRef, ChannelId) -> ok

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 a ssh session. A session is a remote execution of a program. The program may be a shell, an application, a system command, or some built-in subsystem.

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 to the shell/command to be started later.

shell(ConnectionRef, ChannelId) -> ssh_request_status()

Types:

ConnectionRef = ssh_connection_ref()

ChannelId = ssh_channel_id()

Will request that the user's default shell (typically defined in /etc/passwd in UNIX systems) be started at the other end.

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

Types:

ConnectionRef = ssh_connection_ref()

ChannelId = ssh_channel_id()

Subsystem = string()

Timeout = timeout()

Sends a request to execute a predefined subsystem.

ssh_sftp

Erlang module

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

COMMON 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 mean 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 not provided, setups a ssh connection in this case a connection reference will be returned too. A 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.

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 a sftp channel. If the ssh connection should be closed call *ssh:close/1*.

```
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's not there.

```
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 is 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 is used for reading directory contents.

```
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}`, or `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()
```

Write `data` to the file referenced by `Handle`. The file should be opened with `write` or `append` flag. Returns `ok` if successful and `{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 = int()
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()
```

Read 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`. The directory should be empty.

ssh_sftpd

Erlang module

Specifies a channel process to handle a sftp subsystem.

COMMON 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 communicating with the file server. 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 the use of 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`.