Sockets Application (SOCKETS)

version 1.0

Joe Armstrong

1997-05-01

Typeset in LaTeX from SGML source using the DOCBUILDER 3.0 Document System.

Contents

SOCKETS Reference Manual

Short Summaries

sockets

No functions are exported

socket

The following functions are exported:

udp

The following functions are exported:

- `start() -> { ok, Pid }`
  [page 12]
- `send(Address, Port, Packet) -> ok | { error, Reason }`
  [page 12] Sends a packet to a specified address.
- `open(Port, Options) -> {ok, Id } | { error, Reason }`
  [page 12] Associates a UDP port number with the process calling it.
- `open(Port) -> {ok, Id } | { error, Reason }`
  [page 12] Associates a UDP port number with the process calling it.
- `close(Id) -> ok | { error, Reason }`
  [page 13] Removes the association denoted by `Id`.
- `send(Id, Address, Port, Packet) -> ok | { error, Reason }`
  [page 13] Sends a packet to a specified `Address` and `Port` (from port associated with `Id`).
- `send_same(Id, Packet) -> ok | { error, Reason }`
  [page 13] Sends `Packet` to the same address as the previous `send/4`.
- `port(Id) -> { ok, Port } | { error, Reason }`
  [page 13] Returns the UDP port number associated with `Id`.
- `gethostbyname(HostName) -> { ok, IPAddress } | { error, Reason }`
  [page 13] Returns the IP address for `HostName`.

sockets (Application)

This chapter describes the `sockets` application in the development environment. The Sockets application contains the following services:

- `socket`
- `udp`

These services are started if the configuration parameters described below are valid.

Configuration

The following configuration parameters are defined for the socket application. For more information about configuration parameters see application(3):

`start_socket = true | false` Starts the `socket` server if the parameter is `true`.
The default value is `true`.

`start_udp = true | false` Starts the `udp` server if the parameter is `true`.
The default value is `true`.

SEE ALSO

socket(3), udp(3), application(3)

socket (Module)

> Warning:
> This module should not be used in new development it will be removed in the next major release. Use `gen_tcp` in the `kernel` application instead.

Sockets are the BSD UNIX interface to communication protocols. Various protocols may be accessed through sockets.

A socket is a full duplex communications channel between two UNIX processes, either over a network to a remote machine, or locally between processes running on the same machine. A socket connects two parties, the initiator and the connector. The initiator is the UNIX process which first opens the socket. It issues a series of system calls to set up the socket and then waits for another process to create a connection to the socket. When the connector starts, it also issues a series of system calls to set up the socket. Both processes then continue to run and the communications channel is bound to a file descriptor which both processes use for reading and writing.

The module `udp` support UDP/IP sockets.

Sockets are also supported on Windows NT and Windows 95. However, for these operating systems only support the `AF_INET` protocol family and the `STREAM` protocol.

Exports

`listen(Protocol, Family, Address, Mode)`

Sets up a socket which listens to `Address`. It also binds the name specified by `Address` to the socket. `Protocol` must be either the atom `STREAM` (connection oriented), or `DGRAM` (not connection oriented). `Family` is either `AF_INET` or `AF_UNIX`.

If `AF_INET` is chosen, then the UNIX process that is to connect to the socket can run on any other accessible machine on the Internet. If this is the case, `Address` is an integer which specifies the port number to be listened to. This port number uniquely identifies the socket on the machine. If port number `0` is chosen, a free port number is automatically chosen by the UNIX kernel.

> Note:
> These port numbers are not to be confused with Erlang ports, they are UNIX socket ports.

Socket ports are used with a host name to create an end point for a socket connection.

- `listen/4` with `Protocol=STREAM` returns the tuple {`Filedescriptor`, `Portnumber`}
- `Filedescriptor` is an integer which specifies the file descriptor assigned to the socket listened to.
- `Portnumber` is an integer which specifies the port number assigned to the socket.
- If `Address` is not zero in the call to `listen`, the returned port number is equal to `Address`.
- `listen/4` with `Protocol=DGRAM` returns the tuple {`Filedescriptor`, `Portnumber, Socket`}, where `Socket` is the socket identifier to be used as described in `accept/1`.
-

For a (DGRAM) Erlang socket (nor connection oriented), the address of the first client to communicate with the initiator is established for future communication.

> **Note:**
> After establishing the client address no other client can communicate on this socket.

If `AF_UNIX` is used, the process which connects to the socket must run on the same machine. In this case, `Address` must be a file name such as `/tmp/mysocket`.

If `Protocol` is `DGRAM`, the length of the file name must not exceed 14 characters. This is useful when two processes on the same machine need to connect to each other. The file name is used as a common address for the two processes. In this case, `listen` returns the tuple {`Filedescriptor, noport`}, or {`Filedescriptor, noport, Socket`} as the concept of a port is not applicable for sockets in the `AF_UNIX` domain.

`Mode` must be one of:

```
{packet, N}
{binary_packet, N}
raw       == {packet, 0}
onebyte   == {packet, 1}
twobytes  == {packet, 2}
fourbytes == {packet, 4}
asn1
```

Valid values for `N` are `0, 1, 2`, and `4`. This parameter specifies how to read or write to the socket. If `Mode` is {`packet, N`}, then `N` bytes will be appended to the start of each series of bytes written to the socket to indicate the length of the string. These `N` bytes are in binary format, with the most significant byte first. In this way, it is possible to check that all bytes that were written are also read. For this reason, no partitioned messages will ever be delivered.

> **Note:**
> Nothing will be appended for DGRAM sockets. For a datagram socket, the user has to append the length indicator before sending the package. Datagram sockets are not reliable, so reliable delivery must be taken care of at user level.

If Mode is {binary_packet, N}, the socket is in binary mode and a bytes header of N is appended to the start of binary data. When data is delivered to a socket in binary mode, the data is delivered as a binary instead of being unpacked as a byte list. If N is 0, nothing will be appended.

If Mode is asn1, the receiving side of the connection will assume that BER-coded ASN.1 messages are sent on the socket. The header of the ASN.1 message is checked to find out the total length of the ASN.1 message. That number of bytes is then read from the socket and only one message at a time is delivered to the Erlang system.

> **Note:**
> The asn1 mode will only work if all BER encoded data uses the definite length form.

If the indefinite length form is used (the senders decision), only the tag and length bytes are received and then the connection is broken. The raw or {packet,0} mode should be used if the indefinite length form can occur (received by the Erlang system) .

For this reason, if the options {packet, N}, {binary_packet, N} (N > 0), or asn1 are set on the socket, all that is written at the sender side will be read (in one chunk) on the reader side. This can be very convenient as this result is not guaranteed in TCP. In TCP, the messages may be divided into partitions in unpredictable ways. With TCP, a STREAM of bytes is delivered and it is not a datagram protocol.

Example:

```
ListenSocket = socket:listen('STREAM', 'AF_INET', 3000, {packet, 2}).
```

ListenSocket may be bound to {3, 3000}, where 3 is a file descriptor and 3000 is the port listened to. If not successful, the process evaluating listen evaluates exit({listen, syncerror}). This happens if, for example, Portnumber is set to a number which is already occupied on the machine.

`accept(ListenSocket)`

After a listen, the incoming request to connect for a connection oriented (STREAM) socket may be accepted. This is done with the call accept. The parameter ListenSocket is the tuple returned from the previous call to listen. The call to accept suspends the caller until a connection has been established from outside. A process identifier is returned to the caller. This process is located between the user and the actual socket. All communication with the socket takes place through this process, which understands a series of messages and also sends a series of messages to the process that initiated the call to accept.

For a (DGRAM) socket, which is not connection oriented, the accept/1 function may be used to extract the Socket identifier from ListenSocket. This, however, is not required.

Example:

```
Socket = socket:accept(ListenSocket).
```

After this statement, it is possible to communicate with the socket. The messages which may be sent to the socket are:

```
Socket ! {self(), {deliver, ByteList}}.
```

or

```
Socket ! {self(), {deliver, Binary}}.
```

It causes `Binary`/`ByteList` to be written to the socket.

> **Note:**
> It is not recommended to send packages which are longer than 512 bytes on a datagram (DGRAM) socket.

```
Socket ! {self(), close}.
```

Closes the socket down in an orderly way. If the socket is not closed in this way, it will be automatically closed when the process terminates. The messages which can be received from the socket are best explained by an example:

```
receive
    {Socket, {socket_closed, normal}} ->
        ok;   %% socket close by foreign host
    {Socket, {socket_closed, Error}} ->
        notok; %% something has happened to the socket
    {Socket, {fromsocket, Bytes}} ->
        {bytes, Bytes}
end.
```

Two messages may be sent to the socket: deliver and close. The socket can send three messages back: two error messages and one message which indicates the arrival of new data. All of these are shown below.

Input to the socket:

- `{self(), {deliver, ByteList}}`
- `{self(), {deliver, Binary}}`
- `{self(), close}`

Output from the socket:

- `{Socket, {socket_closed, normal}}`
- `{Socket, {socket_closed, Error}}`
- `{Socket, {fromsocket, ByteList}}`
- `{Socket, {fromsocket, Binary}}`

Sometimes, it may be convenient to listen to several sockets at the same time. This can be achieved by assigning one Erlang process at each port number to the task of of listening.

Another common situation in network programming is a server which listens to one or more ports waiting for a connect message from the network. When it arrives, a separate process is spawned to specifically handle the connection. It returns and continues waiting for new connections from the network.

The code for this could be similar to the following example:

```
top(Port) ->
    Listen = socket:listen('STREAM', 'AF_INET', Port, {packet, 2}),
    loop(Listen).

loop(Listen) ->
    Pid = spawn(mymod, connection, [Listen, self()]),
    receive
        {Pid, ok} ->
            loop(Listen)
    end.

connection(Listen, Father) ->
    Socket = socket:accept(Listen),
    Father ! {self(), ok},
    Socket ! {self(), {deliver, "Hello there"}},
    .....
    ....
```

This code first spawns a process and lets the new process be suspended while waiting for the connection from the network. Once the new process is connected, the original process is informed by the `{self(), ok}` message. That process then spawns another, and so on.

If there is a listening function to a port and `accept/2` has been evaluated, the process is suspended and cannot be aborted. To stop accepting input, the process which makes the call receives an EXIT signal. The accept call then terminates and no more connections are accepted until a new accept call is made to the same `ListenSocket`. To achieve this, `loop(Listen)` can be modified in the following way:

```
loop(Listen) ->
    Pid = spawn(mymod, connection, [Listen, self()]),
    loop(Pid, Listen).

loop(Pid, Listen) ->
    receive
        {Pid, ok} ->
            loop(Listen);
        stop ->
            exit(Pid, abort),
            exit(normal)
    end.
```

When the code shown above has received the `stop` message and exited, there is no error in the `Listen` socket. It is still intact and can be used again in a new call to `loop/1`.

Another common situation in socket programming is a requirement to listen to an address for connections and then have all the connections handled by a single, special process which reads and writes several sockets simultaneously. The following example shows how this requirement can be coded:

```
my_accept(ListenFd, User) ->
    S = socket:accept(ListenFd),
    socket:controlling_process(S, User),
    my_accept(ListenFd, User).
```

The process `User` runs code which is similar to the following example:

```
run(Sockets) when list(Sockets) ->
    receive
        {From, {fromsocket, Bytes}} ->
            case lists:member(From, Sockets) of
                true ->  %% old socket
                    handle_input(Bytes),
                    run(Sockets);
                false ->  %% new connection
                    handle_input(Bytes),
                    run([From|Sockets])
            end;
        ..........   etc.
```

`client(Protocol, Family, Address, Mode)`

If another UNIX process already listens to a socket, the socket on the client side may be opened with this call. As before, `Protocol` must be either of the atoms `STREAM` or `DGRAM`, and `Family` can be either `AF_UNIX` or `AF_INET`. When `Family` is `AF_INET`, `Address` must be a tuple of the type {`IPAddress`, `Portnumber`}. It may be argued that users should not have to know port numbers, only names of services as in the BSD library routine `getservbyname()`. However, this idea has not been implemented in this package, so the port number has to be specified when a client is to be connected to a socket over the Internet. Examples:

```
Socket1 = socket:client('STREAM', 'AF_INET',
                        {'gin.eua.ericsson.se', 1000}, raw),
Socket2 = socket:client('DGRAM', 'AF_INET',
                        {'gin', 1001}, {packet, 2}),
Socket3 = socket:client('STREAM', 'AF_INET',
                        {'134.138.99.53', 1002}, asn1),
Socket4 = socket:client('STREAM', 'AF_INET',
                        {'gin', 1003}, {binary_packet, 4}),
```

As can be seen in the examples shown above, several formats are allowed for `Address`. The `Mode` variable in the call to `client` is the same as in the calls to `listen`.

When `Family` is `AF_UNIX`, `Address` must be a file name.

> **Note:**
> If `Protocol` is `DGRAM`, the length of the filename must not exceed 14 characters.

For example:

```
Socket4 = socket:client('STREAM', 'AF_UNIX', '/tmp/mysocket', raw),
```

`client` returns a process identifier of a process, with the same characteristics as the process described for the `accept` call above.

`controlling_process(Socket, Pid)`

The Pid of the process which performed the initiation is known by the socket when a value has been returned from the call to `accept`, or the call to `client`. All output from the socket is sent to this process. All input to the socket must also be wrapped with the Pid of the original process.

If the controlling process is to be changed, the socket must be informed. This requirement resembles the way an Erlang port needs to know the Pid of the process which opened it. The socket (and the port) must know where to send messages. The function above assigns a new controlling process to the socket. In this way, this function ensures that all output from the socket is sent to another process than the process which created the socket. It also ensures that no messages from the socket are lost while the switch takes place.

`peername(Socket)`

Returns the name of the peer to `Socket`.

If AF_UNIX is used, `peername` returns the file name used as the address of a string. If AF_INET is used, `peername` returns the tuple {`Portnumber, IPAddress`}.

`resolve()`

Returns the official name of the current host.

`resolve(IPAddress)`

Returns the official name of the host with the address `IPAddress`.

`close(Socket)`

Closes the socket. This is equivalent to sending a {`self(), close`} message to the process which controls the socket. It also operates on sockets returned by the `listen` call. This is the method used to stop listening to a socket.

`start()`

Starts the socket server.

`stop()`

Stops the socket server, and closes all open sockets.

Features

Even if a socket is opened in {packet, N} mode, it is possible to write binaries to it. The receiving part of the socket determines if data from the socket is to be unpacked as a byte list or not. That is, a sender may be in binary mode ({binary_packet, N}) and the receiver in byte list mode ({packet, N}), or the other way round. The only restriction is that the packet sizes must match.

The modes raw and twobytes are kept for backwards compatibility, and the modes onebyte and fourbytes have been added for forward compatibility.

Bugs

Some types of connections are not fully tested.

Once a client has established communication with the server side of a datagram socket, no other clients can use this socket. The reason for this is that the address binds the first clients address to the server socket.

udp (Module)

Warning:
This module should not be used in new development it will be removed in the next major release. Use `gen_udp` in the `kernel` application instead.

The `udp` module is an interface to User Datagram Protocol (`UDP`). Rather than using the socket library as an interface to sockets, the aim of the `udp` library is to provide an easy-to-use interface directly to `UDP`.

Exports

`start() -> { ok, Pid }`

Starts the `udp` server and returns `{ok, Pid}`. This function must be called before any other functions in this module are called.

`send(Address, Port, Packet) -> ok | { error, Reason }`

Types:

- Address = { integer(), integer(), integer(), integer()} | atom() | string()
- Port = integer(0..65535)
- Packet = [byte()] | binary()
- Reason = term()

Sends `Packet` to the specified address (address, port). It returns `ok`, or `{error, Reason}`. `Address` can be an IP address expressed as a tuple, for example `{192, 0, 0, 1}`. It can also be a host name expressed as an atom or a string, for example 'somehost.some.domain'. `Port` is an integer, and `Packet` is either a list of bytes or a binary.

`open(Port, Options) -> {ok, Id } | { error, Reason }`
`open(Port) -> {ok, Id } | { error, Reason }`

Types:

- Port = integer(0..65535)
- Options = [binary|list]
- Id = integer()
- Reason = term()

Associates a UDP port number (Port) with the process calling it. It returns {ok, Id}, or {error, Reason}. The returned Id is used to send packets from this port. Options is a list of options associated with this port. The list can contain either the atom binary or list.

open/1 is equivalent to open(PortNo, [list]). (This makes open/1 backwards compatible with previous versions of the udp library).

A process which calls open receives messages of the type {udp, Id, IP, InPortNo, Packet} when UDP packets arrive at that port. IP and InPortNo define the address from where Packet came. Packet is a list of bytes if the option list was used. Packet is a binary if the option binary was used.

If you set Port to 0, the underlying Operating System assigns a free UDP port. (You can find out which port by calling udp:port(Id).)

If any of the following functions are called with an Id that was not opened by the process which calls the function, they will return {error, not_owner}.

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

Removes the association Id that was performed with open. Returns ok, or {error, Reason}.

```
send(Id, Address, Port, Packet) -> ok | { error, Reason }
```

Same as send/3, except that it sends from the UDP port associated with Id. Returns ok, or {error, Reason}.

```
send_same(Id, Packet) -> ok | { error, Reason }
```

Sends the packet to the same IP address and port as the previous send/4 function which was used for the association Id. Returns ok, or {error, Reason}.

```
port(Id) -> { ok, Port } | { error, Reason }
```

Types:

- Port = integer(0..65535)

Returns the UDP port number associated with Id. Returns {ok, PortNo}, or {error, Reason}.

```
gethostbyname(HostName) -> { ok, IPAddress } | { error, Reason }
```

Types:

- HostName = atom() | string()
- IPAddress = { integer(), integer(), integer(), integer() }

Uses HostName, which is an atom or a string, and tries to find out the IP address of this host by using the gethostbyname(3) (C) library call. Returns {ok, IPAddress}, or {error, Reason}.

Bugs

Since UDP packets are carried in IP datagrams, they have a maximum size of 65508 bytes. In reality, this limit is usually lower and depends on implementation, buffer sizes, and other factors. The `udp` library does not give an error indicatation if you try to send too large packets. The packet is just dropped silently.

Most operating systems limit the number of files which one process can have open at the same time. The limitation in the `udp` library is 255, regardless of whether the OS limit is higher. This is because the `Id` is one byte.

Chapter 2

SOCKETS Release Notes

The Sockets Application (*SOCKETS*) contains code for manipulating UNIX BSD sockets.

2.1 SOCKETS Release Notes

These release notes are for the sockets application.

Sockets is replaced by gen_tcp and gen_udp in the kernel application

The functionality in the sockets application is replaced by `gen_tcp` and `gen_udp` in the `kernel` application. It will be phased out within half a year. The use of the modules `socket` and `udp` in new implementations is strongly discuraged.

sockets 1.0.5

Improvements and new features

Minor changes (in the source code structure) to allow building for an additional platform (Vxworks).

sockets 1.0.4

Improvements and new features

The fixed bugs and malfunctions in sockets 1.0.3 is now included in the Windows NT version too.

sockets 1.0.3

Fixed Bugs and malfunctions

- A problem regarding that the system runs out of UDP-ports is corrected.
  Own Id: OTP-1228
  Aux Id:AD-75861
- The return value from`socket:start()` was misspelled `allready_started` and is changed to `already_started`.
  Own Id: OTP-1188
- The `asn1` mode in socket is corrected. The documentation regarding this is also clarified.
  Own Id: OTP-1268

Incompatibilities with sockets 1.0.2

- The return value from`socket:start()` was misspelled `allready_started` and is changed to `already_started`. This is a minor API change which is not likely to break any existing code.

sockets 1.0.2

Fixed Bugs and malfunctions

- `esock.c` is using a pointer after it has been freed. This might cause file descriptor 0 (stdin) to be closed.
  Own Id: OTP-1226

sockets 1.0.1

Fixed Bugs and malfunctions

- socket:listen/4 crashes if the originating process has (or receives before listen return) a message matching {_,_} in its message queue.
  Own Id: OTP-1112

sockets 1.0

Improvements and new features

`socket` and `udp` are components in the `sockets` application. It is thus possible to configure if `socket` and/or `udp` should be started using the `start_socket` resp. `start_udp` configuration parameters; see `sockets(3)`.

Fixed bugs and malfunctions

- `socket` hangs when receiving a short `SYNC_ERROR` packet.

Index

Modules are typed in *this way*.
Functions are typed in `this way`.