Interoperability Tutorial

version 5.6

Typeset in LaTeX from SGML source using the DocBuilder-0.9.8 Document System.

Contents

1 Interoperability Tutorial **1**

1.1 Introduction . . . 1
1.1.1 Purpose . . . 1
1.1.2 Prerequisites . . . 1
1.2 Overview . . . 1
1.2.1 Built-In Mechanisms . . . 1
1.2.2 C and Java Libraries . . . 3
1.2.3 Standard Protocols . . . 4
1.2.4 IC . . . 4
1.2.5 Old Applications . . . 4
1.3 Problem Example . . . 4
1.3.1 Description . . . 4
1.4 Ports . . . 5
1.4.1 Erlang Program . . . 6
1.4.2 C Program . . . 8
1.4.3 Running the Example . . . 10
1.5 Erl_Interface . . . 10
1.5.1 Erlang Program . . . 10
1.5.2 C Program . . . 13
1.5.3 Running the Example . . . 16
1.6 Port drivers . . . 17
1.6.1 Port Drivers . . . 18
1.6.2 Erlang Program . . . 18
1.6.3 C Driver . . . 20
1.6.4 Running the Example . . . 22
1.7 C Nodes . . . 22
1.7.1 Erlang Program . . . 22
1.7.2 C Program . . . 23
1.7.3 Running the Example . . . 30

List of Figures **33**

Chapter 1

Interoperability Tutorial

1.1 Introduction

1.1.1 Purpose

The purpose of this tutorial is to give the reader an orientation of the different interoperability mechanisms that can be used when integrating a program written in Erlang with a program written in another programming language, from the Erlang programmer's point of view.

1.1.2 Prerequisites

It is assumed that the reader is a skilled Erlang programmer, familiar with concepts such as Erlang data types, processes, messages and error handling.

To illustrate the interoperability principles C programs running in a UNIX environment have been used. It is assumed that the reader has enough knowledge to be able to apply these principles to the relevant programming languages and platforms.

> **Note:**
> For the sake of readability, the example code has been kept as simple as possible. It does not include functionality such as error handling, which might be vital in a real-life system.

1.2 Overview

1.2.1 Built-In Mechanisms

There are two interoperability mechanisms built into the Erlang runtime system. One is *distributed Erlang* and the other one is *ports*. A variation of ports is *linked-in drivers*.

Distributed Erlang

An Erlang runtime system is made into a distributed Erlang node by giving it a name. A distributed Erlang node can connect to and monitor other nodes, it is also possible to spawn processes at other nodes. Message passing and error handling between processes at different nodes are transparent. There exists a number of useful `stdlib` modules intended for use in a distributed Erlang system; for example, `global` which provides global name registration. The distribution mechanism is implemented using TCP/IP sockets.

When to use: Distributed Erlang is primarily used for communication Erlang-Erlang. It can also be used for communication between Erlang and C, if the C program is implemented as a C node [page 3], see below.

Where to read more: Distributed Erlang and some distributed programming techniques are described in the Erlang book.
In the Erlang/OTP documentation there is a chapter about distributed Erlang in "Getting Started" (User's Guide).
Relevant man pages are `erlang` (describes the BIFs) and `global`, `net_adm`, `pg2`, `rpc`, `pool` and `slave`.

Ports and Linked-In Drivers

Ports provide the basic mechanism for communication with the external world, from Erlang's point of view. They provide a byte-oriented interface to an external program. When a port has been created, Erlang can communicate with it by sending and receiving lists of bytes (not Erlang terms). This means that the programmer may have to invent a suitable encoding and decoding scheme.

The actual implementation of the port mechanism depends on the platform. In the Unix case, pipes are used and the external program should as default read from standard input and write to standard output. Theoretically, the external program could be written in any programming language as long as it can handle the interprocess communication mechanism with which the port is implemented.

The external program resides in another OS process than the Erlang runtime system. In some cases this is not acceptable, consider for example drivers with very hard time requirements. It is therefore possible to write a program in C according to certain principles and dynamically link it to the Erlang runtime system, this is called a linked-in driver.

When to use: Being the basic mechanism, ports can be used for all kinds of interoperability situations where the Erlang program and the other program runs on the same machine. Programming is fairly straight-forward.
Linked-in drivers involves writing certain call-back functions in C. Very good skills are required as the code is linked to the Erlang runtime system.

> **Warning:**
> An erroneous linked-in driver will cause the entire Erlang runtime system to leak memory, hang or crash.

Where to read more: Ports are described in the "Miscellaneous Items" chapter of the Erlang book.
Linked-in drivers are described in Appendix E.
The BIF `open_port/2` is documented in the man page for `erlang`. For linked-in drivers, the programmer needs to read the information in the man page for `erl_ddll`.

*Examples:*Port example [page 5].

1.2.2 C and Java Libraries

Erl_Interface

Very often the program at the other side of a port is a C program. To help the C programmer a library called Erl_Interface has been developed. It consists of five parts:

- `erl_marshal`, `erl_eterm`, `erl_format`, `erl_malloc` Handling of the Erlang external term format.
- `erl_connect` Communication with distributed Erlang, see C nodes [page 3] below.
- `erl_error` Error print routines.
- `erl_global` Access globally registered names.
- `Registry` Store and backup of key-value pairs.

The Erlang external term format is a representation of an Erlang term as a sequence of bytes, a binary. Conversion between the two representations is done using BIFs.

```
Binary = term_to_binary(Term)
Term = binary_to_term(Binary)
```

A port can be set to use binaries instead of lists of bytes. It is then not necessary to invent any encoding/decoding scheme. Erl_Interface functions are used for unpacking the binary and convert it into a struct similar to an Erlang term. Such a struct can be manipulated in different ways and be converted to the Erlang external format and sent to Erlang.

When to use: In C code, in conjunction with Erlang binaries.

Where to read more: Read about the Erl_Interface User's Guide; Command Reference and Library Reference. In R5B and earlier versions the information can be found under the Kernel application.

*Examples:*erl_interface example [page 10].

C Nodes

A C program which uses the Erl_Interface functions for setting up a connection to and communicating with a distributed Erlang node is called a *C node*, or a *hidden node*. The main advantage with a C node is that the communication from the Erlang programmer's point of view is extremely easy, since the C program behaves as a distributed Erlang node.

When to use: C nodes can typically be used on device processors (as opposed to control processors) where C is a better choice than Erlang due to memory limitations and/or application characteristics.

Where to read more: In the `erl_connect` part of the Erl_Interface documentation, see above. The programmer also needs to be familiar with TCP/IP sockets, see below [page 4], and distributed Erlang, see above [page 1].

*Examples:*C node example [page 22].

Jinterface

In Erlang/OTP R6B, a library similar to Erl_Interface for Java was added called *jinterface*.

1.2.3 Standard Protocols

Sometimes communication between an Erlang program and another program using a standard protocol is desirable. Erlang/OTP currently supports TCP/IP and UDP *sockets*, SNMP, HTTP and IIOP (CORBA). Using one of the latter three requires good knowledge about the protocol and is not covered by this tutorial. Please refer to the documentation for the SNMP, Inets and Orber applications, respectively.

Sockets

Simply put, connection-oriented socket communication (TCP/IP) consists of an initiator socket ("server") started at a certain host with a certain port number. A connector socket ("client") aware of the initiator's host name and port number can connect to it and data can be sent between them. Connection-less socket communication (UDP) consists of an initiator socket at a certain host with a certain port number and a connector socket sending data to it. For a detailed description of the socket concept, please refer to a suitable book about network programming. A suggestion is *UNIX Network Programming, Volume 1: Networking APIs - Sockets and XTI* by W. Richard Stevens, ISBN: 013490012X.

In Erlang/OTP, access to TCP/IP and UDP sockets is provided by the Kernel modules `gen_tcp` and `gen_udp`. Both are easy to use and do not require any deeper knowledge about the socket concept.

When to use: For programs running on the same or on another machine than the Erlang program.

Where to read more: The man pages for `gen_tcp` and `gen_udp`.

1.2.4 IC

IC (IDL Compiler) is an interface generator which given an IDL interface specification automatically generates stub code in Erlang, C or Java. Please refer to the IC User's Guide and IC Reference Manual.

1.2.5 Old Applications

There are two old applications of interest when talking about interoperability: *IG* which was removed in Erlang/OTP R6B and *Jive* which was removed in Erlang/OTP R7B. Both applications have been replaced by IC and are mentioned here for reference only.

IG (Interface Generator) automatically generated code for port or socket communication between an Erlang program and a C program, given a C header file with certain keywords. Jive provided a simple interface between an Erlang program and a Java program.

1.3 Problem Example

1.3.1 Description

A common interoperability situation is when there exists a piece of code solving some complex problem, and we would like to incorporate this piece of code in our Erlang program. Suppose for example we have the following C functions that we would like to be able to call from Erlang.

```
/* complex.c */

int foo(int x) {
  return x+1;
}

int bar(int y) {
  return y*2;
}
```

(For the sake of keeping the example as simple as possible, the functions are not very complicated in this case).

Preferably we would like to able to call `foo` and `bar` without having to bother about them actually being C functions.

```
% Erlang code
...
Res = complex:foo(X),
...
```

The communication with C is hidden in the implementation of `complex.erl`. In the following chapters it is shown how this module can be implemented using the different interoperability mechanisms.

1.4 Ports

This is an example of how to solve the example problem [page 4] by using a port.

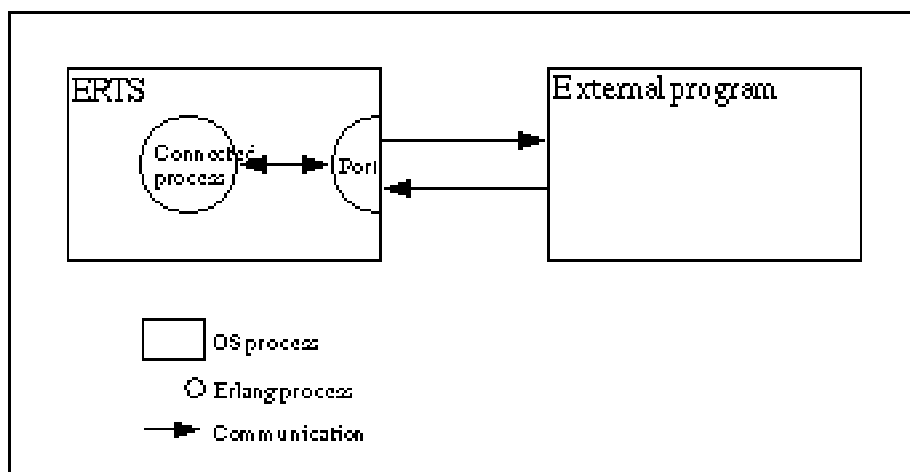

Figure 1.1: Port Communication.

1.4.1 Erlang Program

First of all communication between Erlang and C must be established by creating the port. The Erlang process which creates a port is said to be *the connected process* of the port. All communication to and from the port should go via the connected process. If the connected process terminates, so will the port (and the external program, if it is written correctly).

The port is created using the BIF `open_port/2` with `{spawn,ExtPrg}` as the first argument. The string `ExtPrg` is the name of the external program, including any command line arguments. The second argument is a list of options, in this case only `{packet,2}`. This option says that a two byte length indicator will be used to simplify the communication between C and Erlang. Adding the length indicator will be done automatically by the Erlang port, but must be done explicitly in the external C program.

The process is also set to trap exits which makes it possible to detect if the external program fails.

```
-module(complex1).
-export([start/1, init/1]).

start(ExtPrg) ->
  spawn(?MODULE, init, [ExtPrg]).

init(ExtPrg) ->
  register(complex, self()),
  process_flag(trap_exit, true),
  Port = open_port({spawn, ExtPrg}, [{packet, 2}]),
  loop(Port).
```

Now it is possible to implement `complex1:foo/1` and `complex1:bar/1`. They both send a message to the `complex` process and receive the reply.

```
foo(X) ->
  call_port({foo, X}).
bar(Y) ->
  call_port({bar, Y}).

call_port(Msg) ->
  complex ! {call, self(), Msg},
  receive
    {complex, Result} ->
      Result
  end.
```

The `complex` process encodes the message into a sequence of bytes, sends it to the port, waits for a reply, decodes the reply and sends it back to the caller.

```
loop(Port) ->
  receive
    {call, Caller, Msg} ->
      Port ! {self(), {command, encode(Msg)}},
      receive
        {Port, {data, Data}} ->
          Caller ! {complex, decode(Data)}
      end,
```

```
      loop(Port)
 end.
```

Assuming that both the arguments and the results from the C functions will be less than 256, a very simple encoding/decoding scheme is employed where `foo` is represented by the byte 1, `bar` is represented by 2, and the argument/result is represented by a single byte as well.

```
encode({foo, X}) -> [1, X];
encode({bar, Y}) -> [2, Y].

decode([Int]) -> Int.
```

The resulting Erlang program, including functionality for stopping the port and detecting port failures is shown below.

```
-module(complex1).
-export([start/1, stop/0, init/1]).
-export([foo/1, bar/1]).

start(ExtPrg) ->
    spawn(?MODULE, init, [ExtPrg]).
stop() ->
    complex ! stop.

foo(X) ->
    call_port({foo, X}).
bar(Y) ->
    call_port({bar, Y}).

call_port(Msg) ->
    complex ! {call, self(), Msg},
    receive
        {complex, Result} ->
            Result
    end.

init(ExtPrg) ->
    register(complex, self()),
    process_flag(trap_exit, true),
    Port = open_port({spawn, ExtPrg}, [{packet, 2}]),
    loop(Port).

loop(Port) ->
    receive
        {call, Caller, Msg} ->
            Port ! {self(), {command, encode(Msg)}},
            receive
                {Port, {data, Data}} ->
                    Caller ! {complex, decode(Data)}
            end,
            loop(Port);
        stop ->
```

```
            Port ! {self(), close},
            receive
                {Port, closed} ->
                    exit(normal)
            end;
        {'EXIT', Port, Reason} ->
            exit(port_terminated)
    end.

encode({foo, X}) -> [1, X];
encode({bar, Y}) -> [2, Y].

decode([Int]) -> Int.
```

1.4.2 C Program

On the C side, it is necessary to write functions for receiving and sending data with two byte length indicators from/to Erlang. By default, the C program should read from standard input (file descriptor 0) and write to standard output (file descriptor 1). Examples of such functions, `read_cmd/1` and `write_cmd/2`, are shown below.

```
/* erl_comm.c */

typedef unsigned char byte;

read_cmd(byte *buf)
{
  int len;

  if (read_exact(buf, 2) != 2)
    return(-1);
  len = (buf[0] << 8) | buf[1];
  return read_exact(buf, len);
}

write_cmd(byte *buf, int len)
{
  byte li;

  li = (len >> 8) & 0xff;
  write_exact(&li, 1);

  li = len & 0xff;
  write_exact(&li, 1);

  return write_exact(buf, len);
}

read_exact(byte *buf, int len)
{
  int i, got=0;
```

```
  do {
    if ((i = read(0, buf+got, len-got)) <= 0)
      return(i);
    got += i;
  } while (got<len);

  return(len);
}

write_exact(byte *buf, int len)
{
  int i, wrote = 0;

  do {
    if ((i = write(1, buf+wrote, len-wrote)) <= 0)
      return (i);
    wrote += i;
  } while (wrote<len);

  return (len);
}
```

Note that `stdin` and `stdout` are for buffered input/output and should not be used for the communication with Erlang!

In the `main` function, the C program should listen for a message from Erlang and, according to the selected encoding/decoding scheme, use the first byte to determine which function to call and the second byte as argument to the function. The result of calling the function should then be sent back to Erlang.

```
/* port.c */

typedef unsigned char byte;

int main() {
  int fn, arg, res;
  byte buf[100];

  while (read_cmd(buf) > 0) {
    fn = buf[0];
    arg = buf[1];

    if (fn == 1) {
      res = foo(arg);
    } else if (fn == 2) {
      res = bar(arg);
    }

    buf[0] = res;
    write_cmd(buf, 1);
  }
}
```

Note that the C program is in a `while`-loop checking for the return value of `read_cmd/1`. The reason for this is that the C program must detect when the port gets closed and terminate.

1.4.3 Running the Example

1. Compile the C code.

```
unix> gcc -o extprg complex.c erl_comm.c port.c
```

2. Start Erlang and compile the Erlang code.

```
unix> erl
Erlang (BEAM) emulator version 4.9.1.2

Eshell V4.9.1.2 (abort with ^G)
1> c(complex1).
{ok,complex1}
```

3. Run the example.

```
2> complex1:start("extprg").
<0.34.0>
3> complex1:foo(3).
4
4> complex1:bar(5).
10
5> complex1:stop().
stop
```

1.5 Erl_Interface

This is an example of how to solve the example problem [page 4] by using a port and `erl_interface`. It is necessary to read the port example [page 5] before reading this chapter.

1.5.1 Erlang Program

The example below shows an Erlang program communicating with a C program over a plain port with home made encoding.

```
-module(complex1).
-export([start/1, stop/0, init/1]).
-export([foo/1, bar/1]).

start(ExtPrg) ->
    spawn(?MODULE, init, [ExtPrg]).
stop() ->
    complex ! stop.

foo(X) ->
    call_port({foo, X}).
```

```
bar(Y) ->
    call_port({bar, Y}).

call_port(Msg) ->
    complex ! {call, self(), Msg},
    receive
        {complex, Result} ->
            Result
    end.

init(ExtPrg) ->
    register(complex, self()),
    process_flag(trap_exit, true),
    Port = open_port({spawn, ExtPrg}, [{packet, 2}]),
    loop(Port).

loop(Port) ->
    receive
        {call, Caller, Msg} ->
            Port ! {self(), {command, encode(Msg)}},
            receive
                {Port, {data, Data}} ->
                    Caller ! {complex, decode(Data)}
            end,
            loop(Port);
        stop ->
            Port ! {self(), close},
            receive
                {Port, closed} ->
                    exit(normal)
            end;
        {'EXIT', Port, Reason} ->
            exit(port_terminated)
    end.

encode({foo, X}) -> [1, X];
encode({bar, Y}) -> [2, Y].

decode([Int]) -> Int.
```

Compared to the Erlang module above used for the plain port, there are two differences when using Erl_Interface on the C side: Since Erl_Interface operates on the Erlang external term format the port must be set to use binaries and, instead of inventing an encoding/decoding scheme, the BIFs `term_to_binary/1` and `binary_to_term/1` should be used. That is:

```
open_port({spawn, ExtPrg}, [{packet, 2}])
```

is replaced with:

```
open_port({spawn, ExtPrg}, [{packet, 2}, binary])
```

And:

```
Port ! {self(), {command, encode(Msg)}},
receive
  {Port, {data, Data}} ->
    Caller ! {complex, decode(Data)}
end
```

is replaced with:

```
Port ! {self(), {command, term_to_binary(Msg)}},
receive
  {Port, {data, Data}} ->
    Caller ! {complex, binary_to_term(Data)}
end
```

The resulting Erlang program is shown below.

```
-module(complex2).
-export([start/1, stop/0, init/1]).
-export([foo/1, bar/1]).

start(ExtPrg) ->
    spawn(?MODULE, init, [ExtPrg]).
stop() ->
    complex ! stop.

foo(X) ->
    call_port({foo, X}).
bar(Y) ->
    call_port({bar, Y}).

call_port(Msg) ->
    complex ! {call, self(), Msg},
    receive
        {complex, Result} ->
            Result
    end.

init(ExtPrg) ->
    register(complex, self()),
    process_flag(trap_exit, true),
    Port = open_port({spawn, ExtPrg}, [{packet, 2}, binary]),
    loop(Port).

loop(Port) ->
    receive
        {call, Caller, Msg} ->
            Port ! {self(), {command, term_to_binary(Msg)}},
            receive
                {Port, {data, Data}} ->
                    Caller ! {complex, binary_to_term(Data)}
            end,
            loop(Port);
```

```
        stop ->
            Port ! {self(), close},
            receive
                {Port, closed} ->
                    exit(normal)
            end;
        {'EXIT', Port, Reason} ->
            exit(port_terminated)
    end.
```

Note that calling `complex2:foo/1` and `complex2:bar/1` will result in the tuple {`foo,X`} or {`bar,Y`} being sent to the `complex` process, which will code them as binaries and send them to the port. This means that the C program must be able to handle these two tuples.

1.5.2 C Program

The example below shows a C program communicating with an Erlang program over a plain port with home made encoding.

```
/* port.c */

typedef unsigned char byte;

int main() {
  int fn, arg, res;
  byte buf[100];

  while (read_cmd(buf) > 0) {
    fn = buf[0];
    arg = buf[1];

    if (fn == 1) {
      res = foo(arg);
    } else if (fn == 2) {
      res = bar(arg);
    }

    buf[0] = res;
    write_cmd(buf, 1);
  }
}
```

Compared to the C program above used for the plain port the `while`-loop must be rewritten. Messages coming from the port will be on the Erlang external term format. They should be converted into an `ETERM` struct, a C struct similar to an Erlang term. The result of calling `foo()` or `bar()` must be converted to the Erlang external term format before being sent back to the port. But before calling any other `erl_interface` function, the memory handling must be initiated.

```
erl_init(NULL, 0);
```

For reading from and writing to the port the functions `read_cmd()` and `write_cmd()` from the erl_comm.c example below can still be used.

```
/* erl_comm.c */

typedef unsigned char byte;

read_cmd(byte *buf)
{
  int len;

  if (read_exact(buf, 2) != 2)
    return(-1);
  len = (buf[0] << 8) | buf[1];
  return read_exact(buf, len);
}

write_cmd(byte *buf, int len)
{
  byte li;

  li = (len >> 8) & 0xff;
  write_exact(&li, 1);

  li = len & 0xff;
  write_exact(&li, 1);

  return write_exact(buf, len);
}

read_exact(byte *buf, int len)
{
  int i, got=0;

  do {
    if ((i = read(0, buf+got, len-got)) <= 0)
      return(i);
    got += i;
  } while (got<len);

  return(len);
}

write_exact(byte *buf, int len)
{
  int i, wrote = 0;

  do {
    if ((i = write(1, buf+wrote, len-wrote)) <= 0)
      return (i);
    wrote += i;
  } while (wrote<len);

  return (len);
}
```

The function `erl_decode()` from `erl_marshal` will convert the binary into an `ETERM` struct.

```
int main() {
  ETERM *tuplep;

  while (read_cmd(buf) > 0) {
    tuplep = erl_decode(buf);
```

In this case `tuplep` now points to an `ETERM` struct representing a tuple with two elements; the function name (atom) and the argument (integer). By using the function `erl_element()` from `erl_eterm` it is possible to extract these elements, which also must be declared as pointers to an `ETERM` struct.

```
    fnp = erl_element(1, tuplep);
    argp = erl_element(2, tuplep);
```

The macros `ERL_ATOM_PTR` and `ERL_INT_VALUE` from `erl_eterm` can be used to obtain the actual values of the atom and the integer. The atom value is represented as a string. By comparing this value with the strings "foo" and "bar" it can be decided which function to call.

```
    if (strncmp(ERL_ATOM_PTR(fnp), "foo", 3) == 0) {
      res = foo(ERL_INT_VALUE(argp));
    } else if (strncmp(ERL_ATOM_PTR(fnp), "bar", 3) == 0) {
      res = bar(ERL_INT_VALUE(argp));
    }
```

Now an `ETERM` struct representing the integer result can be constructed using the function `erl_mk_int()` from `erl_eterm`. It is also possible to use the function `erl_format()` from the module `erl_format`.

```
    intp = erl_mk_int(res);
```

The resulting `ETERM` struct is converted into the Erlang external term format using the function `erl_encode()` from `erl_marshal` and sent to Erlang using `write_cmd()`.

```
    erl_encode(intp, buf);
    write_cmd(buf, erl_eterm_len(intp));
```

Last, the memory allocated by the `ETERM` creating functions must be freed.

```
    erl_free_compound(tuplep);
    erl_free_term(fnp);
    erl_free_term(argp);
    erl_free_term(intp);
```

The resulting C program is shown below:

```
/* ei.c */

#include "erl_interface.h"
#include "ei.h"

typedef unsigned char byte;

int main() {
  ETERM *tuplep, *intp;
  ETERM *fnp, *argp;
  int res;
  byte buf[100];
  long allocated, freed;

  erl_init(NULL, 0);

  while (read_cmd(buf) > 0) {
    tuplep = erl_decode(buf);
    fnp = erl_element(1, tuplep);
    argp = erl_element(2, tuplep);

    if (strncmp(ERL_ATOM_PTR(fnp), "foo", 3) == 0) {
      res = foo(ERL_INT_VALUE(argp));
    } else if (strncmp(ERL_ATOM_PTR(fnp), "bar", 17) == 0) {
      res = bar(ERL_INT_VALUE(argp));
    }

    intp = erl_mk_int(res);
    erl_encode(intp, buf);
    write_cmd(buf, erl_term_len(intp));

    erl_free_compound(tuplep);
    erl_free_term(fnp);
    erl_free_term(argp);
    erl_free_term(intp);
  }
}
```

1.5.3 Running the Example

1. Compile the C code, providing the paths to the include files `erl_interface.h` and `ei.h`, and to the libraries `erl_interface` and `ei`.

```
unix> gcc -o extprg -I/usr/local/otp/lib/erl_interface-3.2.1/include \
      -L/usr/local/otp/lib/erl_interface-3.2.1/lib \
      complex.c erl_comm.c ei.c -lerl_interface -lei
```

In R5B and later versions of OTP, the `include` and `lib` directories are situated under `OTPROOT/lib/erl_interface-VSN`, where `OTPROOT` is the root directory of the OTP installation (`/usr/local/otp` in the example above) and `VSN` is the version of the `erl_interface` application (3.2.1 in the example above).
In R4B and earlier versions of OTP, `include` and `lib` are situated under `OTPROOT/usr`.

2. Start Erlang and compile the Erlang code.

```
unix> erl
Erlang (BEAM) emulator version 4.9.1.2

Eshell V4.9.1.2 (abort with ^G)
1> c(complex2).
{ok,complex2}
```

3. Run the example.

```
2> complex2:start("extprg").
<0.34.0>
3> complex2:foo(3).
4
4> complex2:bar(5).
10
5> complex2:bar(352).
704
6> complex2:stop().
stop
```

1.6 Port drivers

This is an example of how to solve the example problem [page 4] by using a linked in port driver.

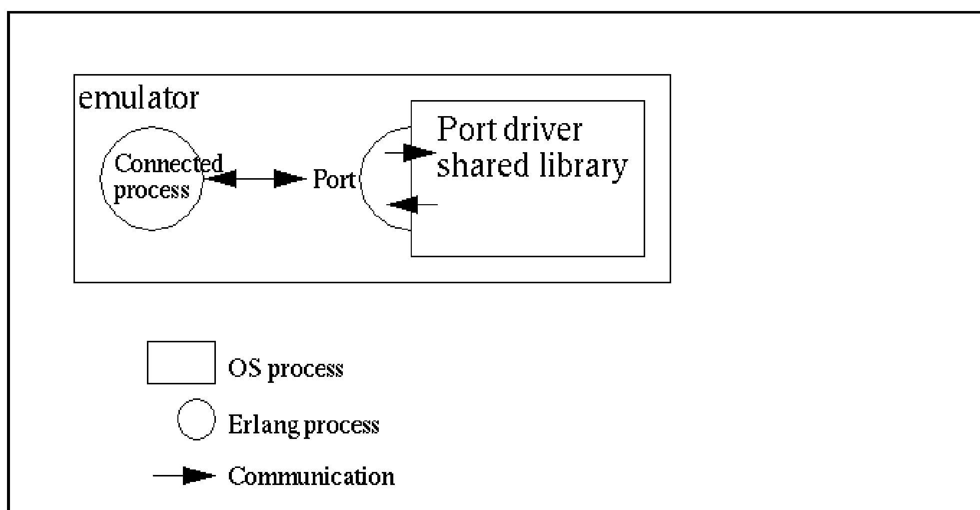

Figure 1.2: Port Driver Communication.

1.6.1 Port Drivers

A port driver is a linked in driver, that is accessible as a port from an Erlang program. It is a shared library (SO in Unix, DLL in Windows), with special entry points. The Erlang runtime calls these entry points, when the driver is started and when data is sent to the port. The port driver can also send data to Erlang.

Since a port driver is dynamically linked into the emulator process, this is the fastest way of calling C-code from Erlang. Calling functions in the port driver requires no context switches. But it is also the least safe, because a crash in the port driver brings the emulator down too.

1.6.2 Erlang Program

Just as with a port program, the port communicates with a Erlang process. All communication goes through one Erlang process that is the *connected process* of the port driver. Terminating this process closes the port driver.

Before the port is created, the driver must be loaded. This is done with the function `erl_dll:load_driver/1`, with the name of the shared library as argument.

The port is then created using the BIF `open_port/2` with the tuple `{spawn, DriverName}` as the first argument. The string `SharedLib` is the name of the port driver. The second argument is a list of options, none in this case.

```
-module(complex5).
-export([start/1, init/1]).

start(SharedLib) ->
    case erl_ddll:load_driver(".", SharedLib) of
        ok -> ok;
        {error, already_loaded} -> ok;
        _ -> exit({error, could_not_load_driver})
    end,
    spawn(?MODULE, init, [SharedLib]).

init(SharedLib) ->
  register(complex, self()),
  Port = open_port({spawn, SharedLib}, []),
  loop(Port).
```

Now it is possible to implement `complex5:foo/1` and `complex5:bar/1`. They both send a message to the `complex` process and receive the reply.

```
foo(X) ->
    call_port({foo, X}).
bar(Y) ->
    call_port({bar, Y}).

call_port(Msg) ->
    complex ! {call, self(), Msg},
    receive
        {complex, Result} ->
            Result
    end.
```

The `complex` process encodes the message into a sequence of bytes, sends it to the port, waits for a reply, decodes the reply and sends it back to the caller.

```
loop(Port) ->
    receive
        {call, Caller, Msg} ->
            Port ! {self(), {command, encode(Msg)}},
            receive
                {Port, {data, Data}} ->
                    Caller ! {complex, decode(Data)}
            end,
            loop(Port)
    end.
```

Assuming that both the arguments and the results from the C functions will be less than 256, a very simple encoding/decoding scheme is employed where `foo` is represented by the byte 1, `bar` is represented by 2, and the argument/result is represented by a single byte as well.

```
encode({foo, X}) -> [1, X];
encode({bar, Y}) -> [2, Y].

decode([Int]) -> Int.
```

The resulting Erlang program, including functionality for stopping the port and detecting port failures is shown below.

```
-module(complex5).
-export([start/1, stop/0, init/1]).
-export([foo/1, bar/1]).

start(SharedLib) ->
    case erl_ddll:load_driver(".", SharedLib) of
        ok -> ok;
        {error, already_loaded} -> ok;
        _ -> exit({error, could_not_load_driver})
    end,
    spawn(?MODULE, init, [SharedLib]).

init(SharedLib) ->
    register(complex, self()),
    Port = open_port({spawn, SharedLib}, []),
    loop(Port).

stop() ->
    complex ! stop.

foo(X) ->
    call_port({foo, X}).
bar(Y) ->
    call_port({bar, Y}).

call_port(Msg) ->
```

```
    complex ! {call, self(), Msg},
    receive
        {complex, Result} ->
            Result
    end.

loop(Port) ->
    receive
        {call, Caller, Msg} ->
            Port ! {self(), {command, encode(Msg)}},
            receive
                {Port, {data, Data}} ->
                    Caller ! {complex, decode(Data)}
            end,
            loop(Port);
        stop ->
            Port ! {self(), close},
            receive
                {Port, closed} ->
                    exit(normal)
            end;
        {'EXIT', Port, Reason} ->
            io:format("~p ~n", [Reason]),
            exit(port_terminated)
    end.

encode({foo, X}) -> [1, X];
encode({bar, Y}) -> [2, Y].

decode([Int]) -> Int.
```

1.6.3 C Driver

The C driver is a module that is compiled and linked into a shared library. It uses a driver structure, and includes the header file `erl_driver.h`.

The driver structure is filled with the driver name and function pointers. It is returned from the special entry point, declared with the macro `DRIVER_INIT(<driver_name>)`.

The functions for receiving and sending data, are combined into a function, pointed out by the driver structure. The data sent into the port is given as arguments, and the data the port sends back is sent with the C-function `driver_output`.

Since the driver is a shared module, not a program, no main function should be present. All function pointers are not used in our example, and the corresponding fields in the `driver_entry` structure are set to NULL.

All functions in the driver, takes a handle (returned from `start`), that is just passed along by the erlang process. This must in some way refer to the port driver instance.

The example_drv_start, is the only function that is called with a handle to the port instance, so we must save this. It is customary to use a allocated driver-defined structure for this one, and pass a pointer back as a reference.

It is not a good idea to use a global variable; since the port driver can be spawned by multiple Erlang processes, this driver-structure should be instantiated multiple times.

```
/* port_driver.c */

#include <stdio.h>
#include "erl_driver.h"

typedef struct {
    ErlDrvPort port;
} example_data;

static ErlDrvData example_drv_start(ErlDrvPort port, char *buff)
{
    example_data* d = (example_data*)driver_alloc(sizeof(example_data));
    d->port = port;
    return (ErlDrvData)d;
}

static void example_drv_stop(ErlDrvData handle)
{
    driver_free((char*)handle);
}

static void example_drv_output(ErlDrvData handle, char *buff, int bufflen)
{
    example_data* d = (example_data*)handle;
    char fn = buff[0], arg = buff[1], res;
    if (fn == 1) {
      res = foo(arg);
    } else if (fn == 2) {
      res = bar(arg);
    }
    driver_output(d->port, &res, 1);
}

ErlDrvEntry example_driver_entry = {
    NULL,                       /* F_PTR init, N/A */
    example_drv_start,          /* L_PTR start, called when port is opened */
    example_drv_stop,           /* F_PTR stop, called when port is closed */
    example_drv_output,         /* F_PTR output, called when erlang has sent */
    NULL,                       /* F_PTR ready_input, called when input descriptor ready */
    NULL,                       /* F_PTR ready_output, called when output descriptor ready */
    "example_drv",              /* char *driver_name, the argument to open_port */
    NULL,                       /* F_PTR finish, called when unloaded */
    NULL,                       /* F_PTR control, port_command callback */
    NULL,                       /* F_PTR timeout, reserved */
    NULL                        /* F_PTR outputv, reserved */
};

DRIVER_INIT(example_drv) /* must match name in driver_entry */
{
    return &example_driver_entry;
}
```

1.6.4 Running the Example

1. Compile the C code.

```
unix> gcc -o exampledrv -fpic -shared complex.c port_driver.c
windows> cl -LD -MD -Fe exampledrv.dll complex.c port_driver.c
```

2. Start Erlang and compile the Erlang code.

```
> erl
Erlang (BEAM) emulator version 5.1

Eshell V5.1 (abort with ^G)
1> c(complex5).
{ok,complex5}
```

3. Run the example.

```
2> complex5:start("example_drv").
<0.34.0>
3> complex5:foo(3).
4
4> complex5:bar(5).
10
5> complex5:stop().
stop
```

1.7 C Nodes

This is an example of how to solve the example problem [page 4] by using a C node. Note that a C node would not typically be used for solving a simple problem like this, a port would suffice.

1.7.1 Erlang Program

From Erlang's point of view, the C node is treated like a normal Erlang node. Therefore, calling the functions `foo` and `bar` only involves sending a message to the C node asking for the function to be called, and receiving the result. Sending a message requires a recipient; a process which can be defined using either a pid or a tuple consisting of a registered name and a node name. In this case a tuple is the only alternative as no pid is known.

```
{RegName, Node} ! Msg
```

The node name `Node` should be the name of the C node. If short node names are used, the plain name of the node will be `cN` where `N` is an integer. If long node names are used, there is no such restriction. An example of a C node name using short node names is thus `c1@idril`, an example using long node names is `cnode@idril.ericsson.se`.

The registered name `RegName` could be any atom. The name can be ignored by the C code, or it could be used for example to distinguish between different types of messages. Below is an example of what the Erlang code could look like when using short node names.

```
-module(complex3).
-export([foo/1, bar/1]).

foo(X) ->
    call_cnode({foo, X}).
bar(Y) ->
    call_cnode({bar, Y}).

call_cnode(Msg) ->
    {any, c1@idril} ! {call, self(), Msg},
    receive
        {cnode, Result} ->
            Result
    end.
```

When using long node names the code is slightly different as shown in the following example:

```
-module(complex4).
-export([foo/1, bar/1]).

foo(X) ->
    call_cnode({foo, X}).
bar(Y) ->
    call_cnode({bar, Y}).

call_cnode(Msg) ->
    {any, 'cnode@idril.du.uab.ericsson.se'} ! {call, self(), Msg},
    receive
        {cnode, Result} ->
            Result
    end.
```

1.7.2 C Program

Setting Up the Communication

Before calling any other Erl_Interface function, the memory handling must be initiated.

```
erl_init(NULL, 0);
```

Now the C node can be initiated. If short node names are used, this is done by calling `erl_connect_init()`.

```
erl_connect_init(1, "secretcookie", 0);
```

The first argument is the integer which is used to construct the node name. In the example the plain node name will be `c1`.
The second argument is a string defining the magic cookie.
The third argument is an integer which is used to identify a particular instance of a C node.

If long node node names are used, initiation is done by calling `erl_connect_xinit()`.

```
erl_connect_xinit("idril", "cnode", "cnode@idril.ericsson.se",
                  &addr, "secretcookie", 0);
```

The first three arguments are the host name, the plain node name, and the full node name. The fourth argument is a pointer to an `in_addr` struct with the IP address of the host, and the fifth and sixth arguments are the magic cookie and instance number.

The C node can act as a server or a client when setting up the communication Erlang-C. If it acts as a client, it connects to an Erlang node by calling `erl_connect()`, which will return an open file descriptor at success.

```
fd = erl_connect("e1@idril");
```

If the C node acts as a server, it must first create a socket (call `bind()` and `listen()`) listening to a certain port number `port`. It then publishes its name and port number with `epmd` (the Erlang port mapper daemon, see the man page for `epmd`).

```
erl_publish(port);
```

Now the C node server can accept connections from Erlang nodes.

```
fd = erl_accept(listen, &conn);
```

The second argument to `erl_accept` is a struct `ErlConnect` that will contain useful information when a connection has been established; for example, the name of the Erlang node.

Sending and Receiving Messages

The C node can receive a message from Erlang by calling `erl_receive msg()`. This function reads data from the open file descriptor `fd` into a buffer and puts the result in an `ErlMessage` struct `emsg`. `ErlMessage` has a field `type` defining which kind of data was received. In this case the type of interest is `ERL_REG_SEND` which indicates that Erlang sent a message to a registered process at the C node. The actual message, an `ETERM`, will be in the `msg` field.

It is also necessary to take care of the types `ERL_ERROR` (an error occurred) and `ERL_TICK` (alive check from other node, should be ignored). Other possible types indicate process events such as link/unlink and exit.

```
  while (loop) {

    got = erl_receive_msg(fd, buf, BUFSIZE, &emsg);
    if (got == ERL_TICK) {
      /* ignore */
    } else if (got == ERL_ERROR) {
      loop = 0; /* exit while loop */
    } else {
      if (emsg.type == ERL_REG_SEND) {
```

Since the message is an `ETERM` struct, Erl_Interface functions can be used to manipulate it. In this case, the message will be a 3-tuple (because that was how the Erlang code was written, see above). The second element will be the pid of the caller and the third element will be the tuple `{Function,Arg}` determining which function to call with which argument. The result of calling the function is made into an `ETERM` struct as well and sent back to Erlang using `erl_send()`, which takes the open file descriptor, a pid and a term as arguments.

```
        fromp = erl_element(2, emsg.msg);
        tuplep = erl_element(3, emsg.msg);
        fnp = erl_element(1, tuplep);
        argp = erl_element(2, tuplep);

        if (strncmp(ERL_ATOM_PTR(fnp), "foo", 3) == 0) {
          res = foo(ERL_INT_VALUE(argp));
        } else if (strncmp(ERL_ATOM_PTR(fnp), "bar", 3) == 0) {
          res = bar(ERL_INT_VALUE(argp));
        }

        resp = erl_format("{cnode, ~i}", res);
        erl_send(fd, fromp, resp);
```

Finally, the memory allocated by the ETERM creating functions (including erl_receive_msg() must be freed.

```
        erl_free_term(emsg.from); erl_free_term(emsg.msg);
        erl_free_term(fromp); erl_free_term(tuplep);
        erl_free_term(fnp); erl_free_term(argp);
        erl_free_term(resp);
```

The resulting C programs can be found in looks like the following examples. First a C node server using short node names.

```
/* cnode_s.c */

#include <stdio.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>

#include "erl_interface.h"
#include "ei.h"

#define BUFSIZE 1000

int main(int argc, char **argv) {
  int port;                                /* Listen port number */
  int listen;                              /* Listen socket */
  int fd;                                  /* fd to Erlang node */
  ErlConnect conn;                         /* Connection data */

  int loop = 1;                            /* Loop flag */
  int got;                                 /* Result of receive */
  unsigned char buf[BUFSIZE];              /* Buffer for incoming message */
  ErlMessage emsg;                         /* Incoming message */

  ETERM *fromp, *tuplep, *fnp, *argp, *resp;
  int res;

  port = atoi(argv[1]);
```

```
  erl_init(NULL, 0);

  if (erl_connect_init(1, "secretcookie", 0) == -1)
    erl_err_quit("erl_connect_init");

  /* Make a listen socket */
  if ((listen = my_listen(port)) <= 0)
    erl_err_quit("my_listen");

  if (erl_publish(port) == -1)
    erl_err_quit("erl_publish");

  if ((fd = erl_accept(listen, &conn)) == ERL_ERROR)
    erl_err_quit("erl_accept");
  fprintf(stderr, "Connected to %s
\r", conn.nodename);

  while (loop) {

    got = erl_receive_msg(fd, buf, BUFSIZE, &emsg);
    if (got == ERL_TICK) {
      /* ignore */
    } else if (got == ERL_ERROR) {
      loop = 0;
    } else {

      if (emsg.type == ERL_REG_SEND) {
        fromp = erl_element(2, emsg.msg);
        tuplep = erl_element(3, emsg.msg);
        fnp = erl_element(1, tuplep);
        argp = erl_element(2, tuplep);

        if (strncmp(ERL_ATOM_PTR(fnp), "foo", 3) == 0) {
          res = foo(ERL_INT_VALUE(argp));
        } else if (strncmp(ERL_ATOM_PTR(fnp), "bar", 3) == 0) {
          res = bar(ERL_INT_VALUE(argp));
        }

        resp = erl_format("{cnode, ~i}", res);
        erl_send(fd, fromp, resp);

        erl_free_term(emsg.from); erl_free_term(emsg.msg);
        erl_free_term(fromp); erl_free_term(tuplep);
        erl_free_term(fnp); erl_free_term(argp);
        erl_free_term(resp);
      }
    }
  } /* while */
}


int my_listen(int port) {
```

```
  int listen_fd;
  struct sockaddr_in addr;
  int on = 1;

  if ((listen_fd = socket(AF_INET, SOCK_STREAM, 0)) < 0)
    return (-1);

  setsockopt(listen_fd, SOL_SOCKET, SO_REUSEADDR, &on, sizeof(on));

  memset((void*) &addr, 0, (size_t) sizeof(addr));
  addr.sin_family = AF_INET;
  addr.sin_port = htons(port);
  addr.sin_addr.s_addr = htonl(INADDR_ANY);

  if (bind(listen_fd, (struct sockaddr*) &addr, sizeof(addr)) < 0)
    return (-1);

  listen(listen_fd, 5);
  return listen_fd;
}
```

Below follows a C node server using long node names.

```
/* cnode_s2.c */

#include <stdio.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>

#include "erl_interface.h"
#include "ei.h"

#define BUFSIZE 1000

int main(int argc, char **argv) {
  struct in_addr addr;                     /* 32-bit IP number of host */
  int port;                                /* Listen port number */
  int listen;                              /* Listen socket */
  int fd;                                  /* fd to Erlang node */
  ErlConnect conn;                         /* Connection data */

  int loop = 1;                            /* Loop flag */
  int got;                                 /* Result of receive */
  unsigned char buf[BUFSIZE];              /* Buffer for incoming message */
  ErlMessage emsg;                         /* Incoming message */

  ETERM *fromp, *tuplep, *fnp, *argp, *resp;
  int res;

  port = atoi(argv[1]);

  erl_init(NULL, 0);
```

```
  addr.s_addr = inet_addr("134.138.177.89");
  if (erl_connect_xinit("idril", "cnode", "cnode@idril.du.uab.ericsson.se",
                        &addr, "secretcookie", 0) == -1)
    erl_err_quit("erl_connect_xinit");

  /* Make a listen socket */
  if ((listen = my_listen(port)) <= 0)
    erl_err_quit("my_listen");

  if (erl_publish(port) == -1)
    erl_err_quit("erl_publish");

  if ((fd = erl_accept(listen, &conn)) == ERL_ERROR)
    erl_err_quit("erl_accept");
  fprintf(stderr, "Connected to %s
\r", conn.nodename);

  while (loop) {

    got = erl_receive_msg(fd, buf, BUFSIZE, &emsg);
    if (got == ERL_TICK) {
      /* ignore */
    } else if (got == ERL_ERROR) {
      loop = 0;
    } else {

      if (emsg.type == ERL_REG_SEND) {
        fromp = erl_element(2, emsg.msg);
        tuplep = erl_element(3, emsg.msg);
        fnp = erl_element(1, tuplep);
        argp = erl_element(2, tuplep);

        if (strncmp(ERL_ATOM_PTR(fnp), "foo", 3) == 0) {
          res = foo(ERL_INT_VALUE(argp));
        } else if (strncmp(ERL_ATOM_PTR(fnp), "bar", 3) == 0) {
          res = bar(ERL_INT_VALUE(argp));
        }

        resp = erl_format("{cnode, ~i}", res);
        erl_send(fd, fromp, resp);

        erl_free_term(emsg.from); erl_free_term(emsg.msg);
        erl_free_term(fromp); erl_free_term(tuplep);
        erl_free_term(fnp); erl_free_term(argp);
        erl_free_term(resp);
      }
    }
  }
}


int my_listen(int port) {
```

```
  int listen_fd;
  struct sockaddr_in addr;
  int on = 1;

  if ((listen_fd = socket(AF_INET, SOCK_STREAM, 0)) < 0)
    return (-1);

  setsockopt(listen_fd, SOL_SOCKET, SO_REUSEADDR, &on, sizeof(on));

  memset((void*) &addr, 0, (size_t) sizeof(addr));
  addr.sin_family = AF_INET;
  addr.sin_port = htons(port);
  addr.sin_addr.s_addr = htonl(INADDR_ANY);

  if (bind(listen_fd, (struct sockaddr*) &addr, sizeof(addr)) < 0)
    return (-1);

  listen(listen_fd, 5);
  return listen_fd;
}
```

And finally we have the code for the C node client.

```
/* cnode_c.c */

#include <stdio.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>

#include "erl_interface.h"
#include "ei.h"

#define BUFSIZE 1000

int main(int argc, char **argv) {
  int fd;                                 /* fd to Erlang node */

  int loop = 1;                           /* Loop flag */
  int got;                                /* Result of receive */
  unsigned char buf[BUFSIZE];             /* Buffer for incoming message */
  ErlMessage emsg;                        /* Incoming message */

  ETERM *fromp, *tuplep, *fnp, *argp, *resp;
  int res;

  erl_init(NULL, 0);

  if (erl_connect_init(1, "secretcookie", 0) == -1)
    erl_err_quit("erl_connect_init");

  if ((fd = erl_connect("e1@idril")) < 0)
    erl_err_quit("erl_connect");
```

```
  fprintf(stderr, "Connected to ei@idril
\r");

  while (loop) {

    got = erl_receive_msg(fd, buf, BUFSIZE, &emsg);
    if (got == ERL_TICK) {
      /* ignore */
    } else if (got == ERL_ERROR) {
      loop = 0;
    } else {

      if (emsg.type == ERL_REG_SEND) {
        fromp = erl_element(2, emsg.msg);
        tuplep = erl_element(3, emsg.msg);
        fnp = erl_element(1, tuplep);
        argp = erl_element(2, tuplep);

        if (strncmp(ERL_ATOM_PTR(fnp), "foo", 3) == 0) {
          res = foo(ERL_INT_VALUE(argp));
        } else if (strncmp(ERL_ATOM_PTR(fnp), "bar", 3) == 0) {
          res = bar(ERL_INT_VALUE(argp));
        }

        resp = erl_format("{cnode, ~i}", res);
        erl_send(fd, fromp, resp);

        erl_free_term(emsg.from); erl_free_term(emsg.msg);
        erl_free_term(fromp); erl_free_term(tuplep);
        erl_free_term(fnp); erl_free_term(argp);
        erl_free_term(resp);
      }
    }
  }
}
```

1.7.3 Running the Example

1. Compile the C code, providing the paths to the Erl_Interface include files and libraries, and to the `socket` and `nsl` libraries.

In R5B and later versions of OTP, the `include` and `lib` directories are situated under `OTPROOT/lib/erl_interface-VSN`, where `OTPROOT` is the root directory of the OTP installation (`/usr/local/otp` in the example above) and `VSN` is the version of the `erl_interface` application (3.2.1 in the example above).
In R4B and earlier versions of OTP, `include` and `lib` are situated under `OTPROOT/usr`.

```
>  gcc -o cserver \
-I/usr/local/otp/lib/erl_interface-3.2.1/include \
-L/usr/local/otp/lib/erl_interface-3.2.1/lib \
complex.c cnode_s.c \
-lerl_interface -lei -lsocket -lnsl
```

```
unix> gcc -o cserver2 \
-I/usr/local/otp/lib/erl_interface-3.2.1/include \
-L/usr/local/otp/lib/erl_interface-3.2.1/lib \
complex.c cnode_s2.c \
-lerl_interface -lei -lsocket -lnsl

unix> gcc -o cclient \
-I/usr/local/otp/lib/erl_interface-3.2.1/include \
-L/usr/local/otp/lib/erl_interface-3.2.1/lib \
complex.c cnode_c.c \
-lerl_interface -lei -lsocket -lnsl
```

2. Compile the Erlang code.

```
unix> erl -compile complex3 complex4
```

3. Run the C node server example with short node names.

Start the C program `cserver` and Erlang in different windows. `cserver` takes a port number as argument and must be started before trying to call the Erlang functions. The Erlang node should be given the short name `e1` and must be set to use the same magic cookie as the C node, `secretcookie`.

```
unix> cserver 3456

unix> erl -sname e1 -setcookie secretcookie
Erlang (BEAM) emulator version 4.9.1.2

Eshell V4.9.1.2  (abort with ^G)
(e1@idril)1> complex3:foo(3).
4
(e1@idril)2> complex3:bar(5).
10
```

4. Run the C node client example. Terminate `cserver` but not Erlang and start `cclient`. The Erlang node must be started before the C node client is.

```
unix> cclient

(e1@idril)3> complex3:foo(3).
4
(e1@idril)4> complex3:bar(5).
10
```

5. Run the C node server, long node names, example.

```
unix> cserver2 3456

unix> erl -name e1 -setcookie secretcookie
Erlang (BEAM) emulator version 4.9.1.2

Eshell V4.9.1.2  (abort with ^G)
(e1@idril.du.uab.ericsson.se)1> complex4:foo(3).
4
(e1@idril.du.uab.ericsson.se)2> complex4:bar(5).
10
```

List of Figures

1.1 Port Communication. 5
1.2 Port Driver Communication. 17