Erlang/OTP System Documentation

Erlang/OTP System Documentation 5.7.5
August 2 2010

August 2 2010

1 User's Guide

How to install Erlang/OTP on UNIX or Windows.

1.1 Installation

1.1.1 Source

This document describes installation procedures for binary releases. Documentation of build and installation procedures for the source release can be found in the source tree at the following locations:

Building and Installing Erlang/OTP
`$ERL_TOP/INSTALL.md`
Cross Compiling Erlang/OTP
`$ERL_TOP/INSTALL-CROSS.md`
How to Build Erlang/OTP on Windows
`$ERL_TOP/INSTALL-WIN32.md`

Where `$ERL_TOP` is the top directory in the source tree.

1.1.2 UNIX

Introduction

The system is delivered as a single compressed tar file.

To browse the on-line HTML documentation, Netscape or an equivalent browser supporting frames is needed.

Installation Procedure

When installed, the entire system, except for a small start-up script, resides in a single directory tree. The location of this directory tree can be chosen arbitrarily by the installer, and it does not need to be in the user's `$PATH`. The only requirements are that the file system where it is placed has enough free space, and that the users who run Erlang/OTP have read access to it. In the example below, the directory tree is assumed to be located at `/usr/local/erlang`, which is here called the *top-level directory*.

It is assumed that you have the compressed tar file, the name of which is `<PREFIX>.tar.gz`, where `<PREFIX>` is a string denoting the particular Erlang/OTP release, e.g. `otp_LXA_11930_sunos5_R9B`.

Wherever the string `<PREFIX>` is used below, it should be replaced by the actual name prefix of the compressed tar file.

The tape archive file does not have one single directory in which all other files are rooted. Therefore the tape archive file must be extracted into an empty (newly created) directory.

- If the *top-level directory* does not already exist, create it:

```
mkdir /usr/local/erlang
```

- Change the current directory to the *top level directory*:

```
```

```
cd /usr/local/erlang
```

- Create the *installation directory* with an appropriate name. For example:

```
mkdir otp_r7b
```

- Change to the *installation directory*, e.g.

```
cd otp_r7b
```

- Assuming the compressed tar file resides in the directory <SOME-DIR>,. extract the compressed tar file into the current directory:

```
gunzip -c <SOME-DIR>/<PREFIX>.tar.gz | tar xfp -
```

- Read the README file in the installation directory for last minute updates, before proceeding.
- Run the Install script in the installation directory, with the absolute path of the installation directory as argument,

```
./Install /usr/local/erlang/otp_r7b
```

  and supply answers to the prompts.

  In most cases, there is a default answer in square brackets ([]). If the default is satisfactory, just press <Return>. In general you are only prompted for one thing:

  - "Do you want to use a minimal system startup instead of the SASL startup?"
    In a minimal system, only the Kernel and STDLIB applications are loaded and started. If the SASL startup is used, the SASL application is included as well. Normally, the minimal system is enough.
- Make Erlang/OTP available for users, either by putting the path /usr/local/erlang/otp_r7b/bin in users $PATH variable, or link the executable /usr/local/erlang/otp_r7b/bin/erl accordingly, e.g.:

```
ln -s /usr/local/erlang/otp_r7b/bin/erl /usr/local/bin/erl
```

1.1.3 Windows

Introduction

The system is delivered as a single .exe file.

To browse the on-line HTML documentation, Netscape or an equivalent browser supporting frames is needed.

Installation Procedure

The installation procedure is is automated. Double-click the .exe file icon and follow the instructions.

1.2 Installation Verification

This chapter is about verifying your installation by performing a few simple tests to see that your system is properly installed.

1.2.1 UNIX

- Start Erlang/OTP from the command line,

```
unix> erl
```

  Expect the following output:

```
Erlang (BEAM) emulator version 5.0.1 [threads]

Eshell V5.0.1  (abort with ^G)
1>
```

- Start the GS-based toolbar from the Erlang shell,

```
1> toolbar:start().
```

  and check that the toolbar window pops up.

  Note: The trailing full stop (`"."`) is an end marker for all commands in the Erlang shell, and must be entered for a command to begin execution.

- Exit by entering the command `halt()`,

```
2> halt().
```

  which should end both the toolbar window and the command line window.

1.2.2 Windows

- Start Erlang/OTP by double-clicking on the Erlang shortcut icon on the desktop.

  Expect a command line window to pop up with the following output,

```
Erlang (BEAM) emulator version 5.0.1 [threads]

Eshell V5.0.1  (abort with ^G)
1>
```

- Start the GS-based toolbar from the Erlang shell,

```
1> toolbar:start().
```

  and check that the toolbar window pops up.

  Note: The trailing full stop (`"."`) is an end marker for all commands in the Erlang shell, and must be entered for a command to begin execution.

- Exit by entering the command `halt()`,

```
2> halt().
```

which should end both the toolbar window and the command line window.

2 User's Guide

2.1 System Principles

2.1.1 Starting the System

An Erlang runtime system is started with the command `erl`:

```
% erl
Erlang (BEAM) emulator version 5.2.3.5 [hipe] [threads:0]

Eshell V5.2.3.5  (abort with ^G)
1>
```

`erl` understands a number of command line arguments, see `erl(1)`. A number of them are also described in this chapter.

Application programs can access the values of the command line arguments by calling one of the functions `init:get_argument(Key)`, or `init:get_arguments()`. See `init(3)`.

2.1.2 Restarting and Stopping the System

The runtime system can be halted by calling `halt/0,1`. See `erlang(3)`.

The module `init` contains function for restarting, rebooting and stopping the runtime system. See `init(3)`.

```
init:restart()
init:reboot()
init:stop()
```

Also, the runtime system will terminate if the Erlang shell is terminated.

2.1.3 Boot Scripts

The runtime system is started using a *boot script*. The boot script contains instructions on which code to load and which processes and applications to start.

A boot script file has the extension `.script`. The runtime system uses a binary version of the script. This *binary boot script* file has the extension `.boot`.

Which boot script to use is specified by the command line flag `-boot`. The extension `.boot` should be omitted. Example, using the boot script `start_all.boot`:

```
% erl -boot start_all
```

If no boot script is specified, it defaults to `ROOT/bin/start`, see Default Boot Scripts below.

The command line flag `-init_debug` makes the `init` process write some debug information while interpreting the boot script:

```
% erl -init_debug
{progress,preloaded}
{progress,kernel_load_completed}
{progress,modules_loaded}
{start,heart}
{start,error_logger}
...
```

See `script(4)` for a detailed description of the syntax and contents of the boot script.

Default Boot Scripts

Erlang/OTP comes with two boot scripts:

`start_clean.boot`

Loads the code for and starts the applications Kernel and STDLIB.

`start_sasl.boot`

Loads the code for and starts the applications Kernel, STDLIB and SASL.

Which of `start_clean` and `start_sasl` to use as default is decided by the user when installing Erlang/OTP using `Install`. The user is asked "Do you want to use a minimal system startup instead of the SASL startup". If the answer is yes, then `start_clean` is used, otherwise `start_sasl` is used. A copy of the selected boot script is made, named `start.boot` and placed in the `ROOT/bin` directory.

User-Defined Boot Scripts

It is sometimes useful or necessary to create a user-defined boot script. This is true especially when running Erlang in embedded mode, see *Code Loading Strategy*.

It is possible to write a boot script manually. The recommended way to create a boot script, however, is to generate the boot script from a release resource file `Name.rel`, using the function `systools:make_script/1,2`. This requires that the source code is structured as applications according to the OTP design principles. (The program does not have to be started in terms of OTP applications but can be plain Erlang).

Read more about `.rel` files in OTP Design Principles and `rel(4)`.

The binary boot script file `Name.boot` is generated from the boot script file `Name.script` using the function `systools:script2boot(File)`.

2.1.4 Code Loading Strategy

The runtime system can be started in either *embedded* or *interactive* mode. Which one is decided by the command line flag `-mode`.

```
% erl -mode embedded
```

Default mode is `interactive`.

- In embedded mode, all code is loaded during system start-up according to the boot script. (Code can also be loaded later by explicitly ordering the code server to do so).

- In interactive mode, code is dynamically loaded when first referenced. When a call to a function in a module is made, and the module is not loaded, the code server searches the code path and loads the module into the system.

Initially, the code path consists of the current working directory and all object code directories under `ROOT/lib`, where `ROOT` is the installation directory of Erlang/OTP. Directories can be named `Name[-Vsn]` and the code server, by default, chooses the directory with the highest version number among those which have the same `Name`. The `-Vsn` suffix is optional. If an `ebin` directory exists under the `Name[-Vsn]` directory, it is this directory which is added to the code path.

The code path can be extended by using the command line flags `-pa Directories` and `-pz Directories`. These will add `Directories` to the head or end of the code path, respectively. Example

```
% erl -pa /home/arne/mycode
```

The code server module `code` contains a number of functions for modifying and checking the search path, see `code(3)`.

2.1.5 File Types

The following file types are defined in Erlang/OTP:

File Type	*File Name/Extension*	*Documented in*
module	`.erl`	Erlang Reference Manual
include file	`.hrl`	Erlang Reference Manual
release resource file	`.rel`	`rel(4)`
application resource file	`.app`	`app(4)`
boot script	`.script`	`script(4)`
binary boot script	`.boot`	-
configuration file	`.config`	`config(4)`
application upgrade file	`.appup`	`appup(4)`
release upgrade file	`relup`	`relup(4)`

Table 1.1: File Types

2.2 Error Logging

2.2.1 Error Information From the Runtime System

Error information from the runtime system, that is, information about a process terminating due to an uncaught error exception, is by default written to terminal (tty):

```
=ERROR REPORT==== 9-Dec-2003::13:25:02 ===
Error in process <0.27.0> with exit value: {{badmatch,[1,2,3]},[{m,f,1},{shell,eval_loop,2}]}
```

The error information is handled by the *error logger*, a system process registered as `error_logger`. This process receives all error messages from the Erlang runtime system and also from the standard behaviours and different Erlang/OTP applications.

The exit reasons (such as `badarg` above) used by the runtime system are described in *Errors and Error Handling* in the Erlang Reference Manual.

The process `error_logger` and its user interface (with the same name) are described in *error_logger(3)*. It is possible to configure the system so that error information is written to file instead/as well as tty. Also, it is possible for user defined applications to send and format error information using `error_logger`.

2.2.2 SASL Error Logging

The standard behaviors (`supervisor`, `gen_server`, etc.) sends progress and error information to `error_logger`. If the SASL application is started, this information is written to tty as well. See *SASL Error Logging* in the SASL User's Guide for further information.

```
% erl -boot start_sasl
Erlang (BEAM) emulator version 5.4.13 [hipe] [threads:0] [kernel-poll]


=PROGRESS REPORT==== 31-Mar-2006::12:45:58 ===
          supervisor: {local,sasl_safe_sup}
             started: [{pid,<0.33.0>},
                       {name,alarm_handler},
                       {mfa,{alarm_handler,start_link,[]}},
                       {restart_type,permanent},
                       {shutdown,2000},
                       {child_type,worker}]

=PROGRESS REPORT==== 31-Mar-2006::12:45:58 ===
          supervisor: {local,sasl_safe_sup}
             started: [{pid,<0.34.0>},
                       {name,overload},
                       {mfa,{overload,start_link,[]}},
                       {restart_type,permanent},
                       {shutdown,2000},
                       {child_type,worker}]

=PROGRESS REPORT==== 31-Mar-2006::12:45:58 ===
          supervisor: {local,sasl_sup}
             started: [{pid,<0.32.0>},
                       {name,sasl_safe_sup},
                       {mfa,{supervisor,
                                start_link,
                                [{local,sasl_safe_sup},sasl,safe]}},
                       {restart_type,permanent},
                       {shutdown,infinity},
                       {child_type,supervisor}]

=PROGRESS REPORT==== 31-Mar-2006::12:45:58 ===
          supervisor: {local,sasl_sup}
             started: [{pid,<0.35.0>},
                       {name,release_handler},
                       {mfa,{release_handler,start_link,[]}},
                       {restart_type,permanent},
                       {shutdown,2000},
                       {child_type,worker}]
```

```
=PROGRESS REPORT==== 31-Mar-2006::12:45:58 ===
         application: sasl
          started_at: nonode@nohost
Eshell V5.4.13  (abort with ^G)
1>
```

2.3 Creating a First Target System

2.3.1 Introduction

When creating a system using Erlang/OTP, the most simple way is to install Erlang/OTP somewhere, install the application specific code somewhere else, and then start the Erlang runtime system, making sure the code path includes the application specific code.

Often it is not desirable to use an Erlang/OTP system as is. A developer may create new Erlang/OTP compliant applications for a particular purpose, and several original Erlang/OTP applications may be irrelevant for the purpose in question. Thus, there is a need to be able to create a new system based on a given Erlang/OTP system, where dispensable applications are removed, and a set of new applications that are included in the new system. Documentation and source code is irrelevant and is therefore not included in the new system.

This chapter is about creating such a system, which we call a *target system*.

In the following sections we consider creating target systems with different requirements of functionality:

- a *basic target system* that can be started by calling the ordinary `erl` script,
- a *simple target system* where also code replacement in run-time can be performed, and
- an *embedded target system* where there is also support for logging output from the system to file for later inspection, and where the system can be started automatically at boot time.

We only consider the case when Erlang/OTP is running on a UNIX system.

There is an example Erlang module `target_system.erl` that contains functions for creating and installing a target system. That module is used in the examples below. The source code of the module is listed at the end of this chapter.

2.3.2 Creating a Target System

It is assumed that you have a working Erlang/OTP system structured according to the OTP Design Principles.

Step 1. First create a `.rel` file (see `rel(4)`) that specifies the `erts` version and lists all applications that should be included in the new basic target system. An example is the following `mysystem.rel` file:

```
%% mysystem.rel
{release,
 {"MYSYSTEM", "FIRST"},
 {erts, "5.1"},
 [{kernel, "2.7"},
  {stdlib, "1.10"},
  {sasl, "1.9.3"},
  {pea, "1.0"}]}.
```

The listed applications are not only original Erlang/OTP applications but possibly also new applications that you have written yourself (here examplified by the application `pea`).

Step 2. From the directory where the `mysystem.rel` file reside, start the Erlang/OTP system:

```

```

```
os> erl -pa /home/user/target_system/myapps/pea-1.0/ebin
```

where also the path to the `pea-1.0` ebin directory is provided.

Step 3. Now create the target system:

```
1> target_system:create("mysystem").
```

The `target_system:create/1` function does the following:

- Reads the `mysystem.rel` file, and creates a new file `plain.rel` which is identical to former, except that it only lists the `kernel` and `stdlib` applications.
- From the `mysystem.rel` and `plain.rel` files creates the files `mysystem.script`, `mysystem.boot`, `plain.script`, and `plain.boot` through a call to `systools:make_script/2`.
- Creates the file `mysystem.tar.gz` by a call to `systools:make_tar/2`. That file has the following contents:

  ```
  erts-5.1/bin/
  releases/FIRST/start.boot
  releases/mysystem.rel
  lib/kernel-2.7/
  lib/stdlib-1.10/
  lib/sasl-1.9.3/
  lib/pea-1.0/
  ```

  The file `releases/FIRST/start.boot` is a copy of our `mysystem.boot`, and a copy of the original `mysystem.rel` has been put in the `releases` directory.
- Creates the temporary directory `tmp` and extracts the tar file `mysystem.tar.gz` into that directory.
- Deletes the `erl` and `start` files from `tmp/erts-5.1/bin`. XXX Why.
- Creates the directory `tmp/bin`.
- Copies the previously creates file `plain.boot` to `tmp/bin/start.boot`.
- Copies the files `epmd`, `run_erl`, and `to_erl` from the directory `tmp/erts-5.1/bin` to the directory `tmp/bin`.
- Creates the file `tmp/releases/start_erl.data` with the contents "5.1 FIRST".
- Recreates the file `mysystem.tar.gz` from the directories in the directory `tmp`, and removes `tmp`.

2.3.3 Installing a Target System

Step 4. Install the created target system in a suitable directory.

```
2> target_system:install("mysystem", "/usr/local/erl-target").
```

The function `target_system:install/2` does the following:

- Extracts the tar file `mysystem.tar.gz` into the target directory `/usr/local/erl-target`.
- In the target directory reads the file `releases/start_erl.data` in order to find the Erlang runtime system version ("5.1").

- Substitutes %FINAL_ROOTDIR% and %EMU% for /usr/local/erl-target and beam, respectively, in the files erl.src, start.src, and start_erl.src of the target erts-5.1/bin directory, and puts the resulting files erl, start, and run_erl in the target bin directory.
- Finally the target releases/RELEASES file is created from data in the releases/mysystem.rel file.

2.3.4 Starting a Target System

Now we have a target system that can be started in various ways.

We start it as a *basic target system* by invoking

```
os> /usr/local/erl-target/bin/erl
```

where only the kernel and stdlib applications are started, i.e. the system is started as an ordinary development system. There are only two files needed for all this to work: bin/erl file (obtained from erts-5.1/bin/erl.src) and the bin/start.boot file (a copy of plain.boot).

We can also start a distributed system (requires bin/epmd).

To start all applications specified in the original mysystem.rel file, use the -boot flag as follows:

```
os> /usr/local/erl-target/bin/erl -boot /usr/local/erl-target/releases/FIRST/start
```

We start a *simple target system* as above. The only difference is that also the file releases/RELEASES is present for code replacement in run-time to work.

To start an *embedded target system* the shell script bin/start is used. That shell script calls bin/run_erl, which in turn calls bin/start_erl (roughly, start_erl is an embedded variant of erl).

The shell script start is only an example. You should edit it to suite your needs. Typically it is executed when the UNIX system boots.

run_erl is a wrapper that provides logging of output from the run-time system to file. It also provides a simple mechanism for attaching to the Erlang shell (to_erl).

start_erl requires the root directory ("/usr/local/erl-target"), the releases directory ("/usr/local/erl-target/releases"), and the location of the start_erl.data file. It reads the run-time system version ("5.1") and release version ("FIRST") from the start_erl.data file, starts the run-time system of the version found, and provides -boot flag specifying the boot file of the release version found ("releases/FIRST/start.boot").

start_erl also assumes that there is sys.config in release version directory ("releases/FIRST/sys.config). That is the topic of the next section (see below).

The start_erl shell script should normally not be altered by the user.

2.3.5 System Configuration Parameters

As was pointed out above start_erl requires a sys.config in the release version directory ("releases/FIRST/sys.config"). If there is no such a file, the system start will fail. Hence such a file has to added as well.

If you have system configuration data that are neither file location dependent nor site dependent, it may be convenient to create the sys.config early, so that it becomes a part of the target system tar file created by target_system:create/1. In fact, if you create, in the current directory, not only the mysystem.rel file, but also a sys.config file, that latter file will be tacitly put in the apropriate directory.

2.3.6 Differences from the Install Script

The above `install/2` procedure differs somewhat from that of the ordinary `Install` shell script. In fact, `create/1` makes the release package as complete as possible, and leave to the `install/2` procedure to finish by only considering location dependent files.

2.3.7 Listing of target_system.erl

```
-module(target_system).
-include_lib("kernel/include/file.hrl").
-export([create/1, install/2]).
-define(BUFSIZE, 8192).

%% Note: RelFileName below is the *stem* without trailing .rel,
%% .script etc.
%%

%% create(RelFileName)
%%
create(RelFileName) ->
    RelFile = RelFileName ++ ".rel",
    io:fwrite("Reading file: \"~s\" ...~n", [RelFile]),
    {ok, [RelSpec]} = file:consult(RelFile),
    io:fwrite("Creating file: \"~s\" from \"~s\" ...~n",
              ["plain.rel", RelFile]),
    {release,
     {RelName, RelVsn},
     {erts, ErtsVsn},
     AppVsns} = RelSpec,
    PlainRelSpec = {release,
                    {RelName, RelVsn},
                    {erts, ErtsVsn},
                    lists:filter(fun({kernel, _}) ->
                                         true;
                                    ({stdlib, _}) ->
                                         true;
                                    (_) ->
                                         false
                                 end, AppVsns)
                   },
    {ok, Fd} = file:open("plain.rel", [write]),
    io:fwrite(Fd, "~p.~n", [PlainRelSpec]),
    file:close(Fd),

    io:fwrite("Making \"plain.script\" and \"plain.boot\" files ...~n"),
    make_script("plain"),

    io:fwrite("Making \"~s.script\" and \"~s.boot\" files ...~n",
              [RelFileName, RelFileName]),
    make_script(RelFileName),

    TarFileName = io_lib:fwrite("~s.tar.gz", [RelFileName]),
    io:fwrite("Creating tar file \"~s\" ...~n", [TarFileName]),
    make_tar(RelFileName),

    io:fwrite("Creating directory \"tmp\" ...~n"),
    file:make_dir("tmp"),

    io:fwrite("Extracting \"~s\" into directory \"tmp\" ...~n", [TarFileName]),
    extract_tar(TarFileName, "tmp"),

    TmpBinDir = filename:join(["tmp", "bin"]),
```

```
    ErtsBinDir = filename:join(["tmp", "erts-" ++ ErtsVsn, "bin"]),
    io:fwrite("Deleting \"erl\" and \"start\" in directory \"~s\" ...~n",
              [ErtsBinDir]),
    file:delete(filename:join([ErtsBinDir, "erl"])),
    file:delete(filename:join([ErtsBinDir, "start"])),

    io:fwrite("Creating temporary directory \"~s\" ...~n", [TmpBinDir]),
    file:make_dir(TmpBinDir),

    io:fwrite("Copying file \"plain.boot\" to \"~s\" ...~n",
              [filename:join([TmpBinDir, "start.boot"])]),
    copy_file("plain.boot", filename:join([TmpBinDir, "start.boot"])),

    io:fwrite("Copying files \"epmd\", \"run_erl\" and \"to_erl\" from \n"
              "\"~s\" to \"~s\" ...~n",
              [ErtsBinDir, TmpBinDir]),
    copy_file(filename:join([ErtsBinDir, "epmd"]),
              filename:join([TmpBinDir, "epmd"]), [preserve]),
    copy_file(filename:join([ErtsBinDir, "run_erl"]),
              filename:join([TmpBinDir, "run_erl"]), [preserve]),
    copy_file(filename:join([ErtsBinDir, "to_erl"]),
              filename:join([TmpBinDir, "to_erl"]), [preserve]),

    StartErlDataFile = filename:join(["tmp", "releases", "start_erl.data"]),
    io:fwrite("Creating \"~s\" ...~n", [StartErlDataFile]),
    StartErlData = io_lib:fwrite("~s ~s~n", [ErtsVsn, RelVsn]),
    write_file(StartErlDataFile, StartErlData),

    io:fwrite("Recreating tar file \"~s\" from contents in directory "
              "\"tmp\" ...~n", [TarFileName]),
    {ok, Tar} = erl_tar:open(TarFileName, [write, compressed]),
    {ok, Cwd} = file:get_cwd(),
    file:set_cwd("tmp"),
    erl_tar:add(Tar, "bin", []),
    erl_tar:add(Tar, "erts-" ++ ErtsVsn, []),
    erl_tar:add(Tar, "releases", []),
    erl_tar:add(Tar, "lib", []),
    erl_tar:close(Tar),
    file:set_cwd(Cwd),
    io:fwrite("Removing directory \"tmp\" ...~n"),
    remove_dir_tree("tmp"),
    ok.


install(RelFileName, RootDir) ->
    TarFile = RelFileName ++ ".tar.gz",
    io:fwrite("Extracting ~s ...~n", [TarFile]),
    extract_tar(TarFile, RootDir),
    StartErlDataFile = filename:join([RootDir, "releases", "start_erl.data"]),
    {ok, StartErlData} = read_txt_file(StartErlDataFile),
    [ErlVsn, RelVsn| _] = string:tokens(StartErlData, " \n"),
    ErtsBinDir = filename:join([RootDir, "erts-" ++ ErlVsn, "bin"]),
    BinDir = filename:join([RootDir, "bin"]),
    io:fwrite("Substituting in erl.src, start.src and start_erl.src to\n"
              "form erl, start and start_erl ...\n"),
    subst_src_scripts(["erl", "start", "start_erl"], ErtsBinDir, BinDir,
                      [{"FINAL_ROOTDIR", RootDir}, {"EMU", "beam"}],
                      [preserve]),
    io:fwrite("Creating the RELEASES file ...\n"),
    create_RELEASES(RootDir,
                    filename:join([RootDir, "releases", RelFileName])).

%% LOCALS

%% make_script(RelFileName)
```

```
%%
make_script(RelFileName) ->
    Opts = [no_module_tests],
    systools:make_script(RelFileName, Opts).

%% make_tar(RelFileName)
%%
make_tar(RelFileName) ->
    RootDir = code:root_dir(),
    systools:make_tar(RelFileName, [{erts, RootDir}]).

%% extract_tar(TarFile, DestDir)
%%
extract_tar(TarFile, DestDir) ->
    erl_tar:extract(TarFile, [{cwd, DestDir}, compressed]).

create_RELEASES(DestDir, RelFileName) ->
    release_handler:create_RELEASES(DestDir, RelFileName ++ ".rel").

subst_src_scripts(Scripts, SrcDir, DestDir, Vars, Opts) ->
    lists:foreach(fun(Script) ->
                          subst_src_script(Script, SrcDir, DestDir,
                                           Vars, Opts)
                  end, Scripts).

subst_src_script(Script, SrcDir, DestDir, Vars, Opts) ->
    subst_file(filename:join([SrcDir, Script ++ ".src"]),
               filename:join([DestDir, Script]),
               Vars, Opts).

subst_file(Src, Dest, Vars, Opts) ->
    {ok, Conts} = read_txt_file(Src),
    NConts = subst(Conts, Vars),
    write_file(Dest, NConts),
    case lists:member(preserve, Opts) of
        true ->
            {ok, FileInfo} = file:read_file_info(Src),
            file:write_file_info(Dest, FileInfo);
        false ->
            ok
    end.

%% subst(Str, Vars)
%% Vars = [{Var, Val}]
%% Var = Val = string()
%% Substitute all occurrences of %Var% for Val in Str, using the list
%% of variables in Vars.
%%
subst(Str, Vars) ->
    subst(Str, Vars, []).

subst([$%, C| Rest], Vars, Result) when $A =< C, C =< $Z ->
    subst_var([C| Rest], Vars, Result, []);
subst([$%, C| Rest], Vars, Result) when $a =< C, C =< $z ->
    subst_var([C| Rest], Vars, Result, []);
subst([$%, C| Rest], Vars, Result) when  C == $_ ->
    subst_var([C| Rest], Vars, Result, []);
subst([C| Rest], Vars, Result) ->
    subst(Rest, Vars, [C| Result]);
subst([], _Vars, Result) ->
    lists:reverse(Result).

subst_var([$%| Rest], Vars, Result, VarAcc) ->
    Key = lists:reverse(VarAcc),
    case lists:keysearch(Key, 1, Vars) of
```

```
        {value, {Key, Value}} ->
            subst(Rest, Vars, lists:reverse(Value, Result));
        false ->
            subst(Rest, Vars, [$%| VarAcc ++ [$%| Result]])
    end;
subst_var([C| Rest], Vars, Result, VarAcc) ->
    subst_var(Rest, Vars, Result, [C| VarAcc]);
subst_var([], Vars, Result, VarAcc) ->
    subst([], Vars, [VarAcc ++ [$%| Result]]).

copy_file(Src, Dest) ->
    copy_file(Src, Dest, []).

copy_file(Src, Dest, Opts) ->
    {ok, InFd} = file:open(Src, [raw, binary, read]),
    {ok, OutFd} = file:open(Dest, [raw, binary, write]),
    do_copy_file(InFd, OutFd),
    file:close(InFd),
    file:close(OutFd),
    case lists:member(preserve, Opts) of
        true ->
            {ok, FileInfo} = file:read_file_info(Src),
            file:write_file_info(Dest, FileInfo);
        false ->
            ok
    end.

do_copy_file(InFd, OutFd) ->
    case file:read(InFd, ?BUFSIZE) of
        {ok, Bin} ->
            file:write(OutFd, Bin),
            do_copy_file(InFd, OutFd);
        eof  ->
            ok
    end.

write_file(FName, Conts) ->
    {ok, Fd} = file:open(FName, [write]),
    file:write(Fd, Conts),
    file:close(Fd).

read_txt_file(File) ->
    {ok, Bin} = file:read_file(File),
    {ok, binary_to_list(Bin)}.

remove_dir_tree(Dir) ->
    remove_all_files(".", [Dir]).

remove_all_files(Dir, Files) ->
    lists:foreach(fun(File) ->
                          FilePath = filename:join([Dir, File]),
                          {ok, FileInfo} = file:read_file_info(FilePath),
                          case FileInfo#file_info.type of
                              directory ->
                                  {ok, DirFiles} = file:list_dir(FilePath),
                                  remove_all_files(FilePath, DirFiles),
                                  file:del_dir(FilePath);
                              _ ->
                                  file:delete(FilePath)
                          end
                  end, Files).
```

3 User's Guide

This manual describes the issues that are specific for running Erlang on an embedded system. It describes the differences in installing and starting Erlang compared to how it is done for a non-embedded system.

Note that this is a supplementary document. You still need to read the Installation Guide.

There is also target architecture specific information in the top level README file of the Erlang distribution.

3.1 Embedded Solaris

This chapter describes the OS specific parts of OTP which relate to Solaris.

3.1.1 Memory Usage

Solaris takes about 17 Mbyte of RAM on a system with 64 Mbyte of total RAM. This leaves about 47 Mbyte for the applications. If the system utilizes swapping, these figures cannot be improved because unnecessary daemon processes are swapped out. However, if swapping is disabled, or if the swap space is of limited resource in the system, it becomes necessary to kill off unnecessary daemon processes.

3.1.2 Disk Space Usage

The disk space required by Solaris can be minimized by using the Core User support installation. It requires about 80 Mbyte of disk space. This installs only the minimum software required to boot and run Solaris. The disk space can be further reduced by deleting unnecessary individual files. However, unless disk space is a critical resource the effort required and the risks involved may not be justified.

3.1.3 Installation

This section is about installing an embedded system. The following topics are considered,

- Creation of user and installation directory,
- Installation of embedded system,
- Configuration for automatic start at reboot,
- Making a hardware watchdog available,
- Changing permission for reboot,
- Patches,
- Configuration of the OS_Mon application.

Several of the procedures described below require expert knowledge of the Solaris 2 operating system. For most of them super user privilege is needed.

Creation of User and Installation Directory

It is recommended that the Embedded Environment is run by an ordinary user, i.e. a user who does not have super user privileges.

Throughout this section we assume that the user name is `otpuser`, and that the home directory of that user is,

```
        /export/home/otpuser
```

Furthermore, we assume that in the home directory of `otpuser`, there is a directory named `otp`, the full path of which is,

```
        /export/home/otpuser/otp
```

This directory is the *installation directory* of the Embedded Environment.

Installation of an Embedded System

The procedure for installation of an embedded system does not differ from that of an ordinary system (see the *Installation Guide*), except for the following:

- the (compressed) tape archive file should be extracted in the installation directory as defined above, and,
- there is no need to link the start script to a standard directory like `/usr/local/bin`.

Configuration for Automatic Start at Boot

A true embedded system has to start when the system boots. This section accounts for the necessary configurations needed to achieve that.

The embedded system and all the applications will start automatically if the script file shown below is added to the `/etc/rc3.d` directory. The file must be owned and readable by `root`, and its name cannot be arbitrarily assigned. The following name is recommended,

```
        S75otp.system
```

For further details on initialization (and termination) scripts, and naming thereof, see the Solaris documentation.

```
#!/bin/sh
#
#  File name:  S75otp.system
#  Purpose:    Automatically starts Erlang and applications when the
#              system starts
#  Author:     janne@erlang.ericsson.se
#  Resides in: /etc/rc3.d
#

if [ ! -d /usr/bin ]
then                    # /usr not mounted
        exit
fi

killproc() {            # kill the named process(es)
        pid=`/usr/bin/ps -e |
             /usr/bin/grep -w $1 |
             /usr/bin/sed -e 's/^  *//' -e 's/ .*//'`
        [ "$pid" != "" ] && kill $pid
}

# Start/stop processes required for Erlang

case "$1" in
'start')
        # Start the Erlang emulator
```

```
        #
        su - otpuser -c "/export/home/otpuser/otp/bin/start" &
        ;;
'stop')
        killproc beam
        ;;
*)
        echo "Usage: $0 { start | stop }"
        ;;
esac
```

The file `/export/home/otpuser/otp/bin/start` referred to in the above script, is precisely the script `start` described in the section *Starting Erlang* below. The script variable `OTP_ROOT` in that `start` script corresponds to the example path

```
        /export/home/otpuser/otp
```

used in this section. The `start` script should be edited accordingly.

Use of the `killproc` procedure in the above script could be combined with a call to `erl_call`, e.g.

```
        $SOME_PATH/erl_call -n Node init stop
```

In order to take Erlang down gracefully see the `erl_call(1)` reference manual page for further details on the use of `erl_call`. That however requires that Erlang runs as a distributed node which is not always the case.

The `killproc` procedure should not be removed: the purpose is here to move from run level 3 (multi-user mode with networking resources) to run level 2 (multi-user mode without such resources), in which Erlang should not run.

Hardware Watchdog

For Solaris running on VME boards from Force Computers, there is a possibility to activate the onboard hardware watchdog, provided a VME bus driver is added to the operating system (see also *Installation Problems* below).

See also the `heart(3)` reference manual page in *Kernel*.

Changing Permissions for Reboot

If the `HEART_COMMAND` environment variable is to be set in the `start` script in the section, *Starting Erlang*, and if the value shall be set to the path of the Solaris `reboot` command, i.e.

```
        HEART_COMMAND=/usr/sbin/reboot
```

the ownership and file permissions for `/usr/sbin/reboot` must be changed as follows,

```
        chown 0 /usr/sbin/reboot
        chmod 4755 /usr/sbin/reboot
```

See also the `heart(3)` reference manual page in *Kernel*.

The TERM Environment Variable

When the Erlang runtime system is automatically started from the `S75otp.system` script the `TERM` environment variable has to be set. The following is a minimal setting,

```
TERM=sun
```

which should be added to the `start` script described in the section.

Patches

For proper functioning of flushing file system data to disk on Solaris 2.5.1, the version specific patch with number 103640-02 must be added to the operating system. There may be other patches needed, see the release README file `<ERL_INSTALL_DIR>/README`.

Installation of Module os_sup in Application OS_Mon

The following four installation procedures require super user privilege.

- *Make a copy the Solaris standard configuration file for syslogd.*
  - Make a copy the Solaris standard configuration file for syslogd. This file is usually named `syslog.conf` and found in the `/etc` directory.
  - The file name of the copy must be `syslog.conf.ORIG` but the directory location is optional. Usually it is `/etc`.

    A simple way to do this is to issue the command

    ```
    cp /etc/syslog.conf /etc/syslog.conf.ORIG
    ```
- *Make an Erlang specific configuration file for syslogd.*
  - Make an edited copy of the back-up copy previously made.
  - The file name must be `syslog.conf.OTP` and the path must be the same as the back-up copy.
  - The format of the configuration file is found in the man page for `syslog.conf(5)`, by issuing the command `man syslog.conf`.
  - Usually a line is added which should state:
    - which types of information that will be supervised by Erlang,
    - the name of the file (actually a named pipe) that should receive the information.
  - If e.g. only information originating from the unix-kernel should be supervised, the line should begin with `kern.LEVEL` (for the possible values of `LEVEL` see `syslog.conf(5)`).
  - After at least one tab-character, the line added should contain the full name of the named pipe where syslogd writes its information. The path must be the same as for the `syslog.conf.ORIG` and `syslog.conf.OTP` files. The file name must be `syslog.otp`.
  - If the directory for the `syslog.conf.ORIG` and `syslog.conf.OTP` files is `/etc` the line in `syslog.conf.OTP` will look like:

    ```
    kern.LEVEL                /etc/syslog.otp
    ```
- *Check the file privileges of the configuration files.*
  - The configuration files should have `rw-r--r--` file privileges and be owned by root.

- A simple way to do this is to issue the commands

```
chmod 644 /etc/syslog.conf
chmod 644 /etc/syslog.conf.ORIG
chmod 644 /etc/syslog.conf.OTP
```

- *Note:* If the `syslog.conf.ORIG` and `syslog.conf.OTP` files are not in the `/etc` directory, the file path in the second and third command must be modified.

- *Modify file privileges and ownership of the mod_syslog utility.*
  - The file privileges and ownership of the `mod_syslog` utility must be modified.
  - The full name of the binary executable file is derived from the position of the `os_mon` application if the file system by adding `/priv/bin/mod_syslog`. The generic full name of the binary executable file is thus

```
<OTP_ROOT>/lib/os_mon-<REV>/priv/bin/mod_syslog
```

    Example: If the path to the otp-root is `/usr/otp`, thus the path to the `os_mon` application is `/usr/otp/lib/os_mon-1.0` (assuming revision 1.0) and the full name of the binary executable file is `/usr/otp/lib/os_mon-1.0/priv/bin/mod_syslog`.
  - The binary executable file must be owned by root, have `rwsr-xr-x` file privileges, in particular the setuid bit of user must be set.
  - A simple way to do this is to issue the commands

```
cd <OTP_ROOT>/lib/os_mon-<REV>/priv/bin/mod_syslog
chmod 4755 mod_syslog
chown root mod_syslog
```

The following procedure does not require root privilege.

- Ensure that the configuration parameters for the `os_sup` module in the `os_mon` application are correct.
- Browse the application configuration file (do *not* edit it). The full name of the application configuration file is derived from the position of the OS_Mon application if the file system by adding `/ebin/os_mon.app`.

  The generic full name of the file is thus

```
<OTP_ROOT>/lib/os_mon-<REV>/ebin/os_mon.app.
```

  Example: If the path to the otp-root is `/usr/otp`, thus the path to the `os_mon` application is `/usr/otp/lib/os_mon-1.0` (assuming revision 1.0) and the full name of the binary executable file is `/usr/otp/lib/os_mon-1.0/ebin/os_mon.app`.
- Ensure that the following configuration parameters are bound to the correct values.

Parameter	*Function*	*Standard value*
start_os_sup	Specifies if os_sup will be started or not.	`true`for the first instance on the hardware; `false`for the other instances.

os_sup_own	The directory for (1)the back-up copy, (2) the Erlang specific configuration file for syslogd.	`"/etc"`
os_sup_syslogconf	The full name for the Solaris standard configuration file for syslogd	`"/etc/syslog.conf"`
error_tag	The tag for the messages that are sent to the error logger in the Erlang runtime system.	`std_error`

Table 1.1: Configuration Parameters

If the values listed in the `os_mon.app` do not suit your needs, you should `not` edit that file. Instead you should *override* values in a *system configuration file*, the full pathname of which is given on the command line to `erl`.

Example: The following is an example of the contents of an application configuration file.

```
[{os_mon, [{start_os_sup, true}, {os_sup_own, "/etc"},
{os_sup_syslogconf, "/etc/syslog.conf"}, {os_sup_errortag, std_error}]}].
```

See also the `os_mon(3)`, `application(3)` and `erl(1)` reference manual pages.

Installation Problems

The hardware watchdog timer which is controlled by the `heart` port program requires the `FORCEvme` package, which contains the VME bus driver, to be installed. This driver, however, may clash with the Sun `mcp` driver and cause the system to completely refuse to boot. To cure this problem, the following lines should be added to `/etc/system`:

- `exclude: drv/mcp`
- `exclude: drv/mcpzsa`
- `exclude: drv/mcpp`

> **Warning:**
>
> It is recommended that these lines be added to avoid the clash described, which may make it completely impossible to boot the system.

3.1.4 Starting Erlang

This section describes how an embedded system is started. There are four programs involved, and they all normally reside in the directory `<ERL_INSTALL_DIR>/bin`. The only exception is the program `start`, which may be located anywhere, and also is the only program that must be modified by the user.

In an embedded system there usually is no interactive shell. However, it is possible for an operator to attach to the Erlang system by giving the command `to_erl`. He is then connected to the Erlang shell, and may give ordinary Erlang commands. All interaction with the system through this shell is logged in a special directory.

Basically, the procedure is as follows. The program `start` is called when the machine is started. It calls `run_erl`, which sets things up so the operator can attach to the system. It calls `start_erl` which calls the correct version of

erlexec (which is located in <ERL_INSTALL_DIR>/erts-EVsn/bin) with the correct boot and config files.

3.1.5 Programs

start

This program is called when the machine is started. It may be modified or re-written to suit a special system. By default, it must be called start and reside in <ERL_INSTALL_DIR>/bin. Another start program can be used, by using the configuration parameter start_prg in the application sasl.

The start program must call run_erl as shown below. It must also take an optional parameter which defaults to <ERL_INSTALL_DIR>/releases/start_erl.data.

This program should set static parameters and environment variables such as -sname Name and HEART_COMMAND to reboot the machine.

The <RELDIR> directory is where new release packets are installed, and where the release handler keeps information about releases. See release_handler(3) in the application sasl for further information.

The following script illustrates the default behaviour of the program.

```
#!/bin/sh
# Usage: start [DataFile]
#
ROOTDIR=/usr/local/otp

if [ -z "$RELDIR" ]
then
   RELDIR=$ROOTDIR/releases
fi

START_ERL_DATA=${1:-$RELDIR/start_erl.data}

$ROOTDIR/bin/run_erl /tmp/ $ROOTDIR/log "exec $ROOTDIR/bin/start_erl \
                     $ROOTDIR $RELDIR $START_ERL_DATA" > /dev/null 2>&1 &
```

The following script illustrates a modification where the node is given the name cp1, and the environment variables HEART_COMMAND and TERM have been added to the above script.

```
#!/bin/sh
# Usage: start [DataFile]
#
HEART_COMMAND=/usr/sbin/reboot
TERM=sun
export HEART_COMMAND TERM

ROOTDIR=/usr/local/otp

if [ -z "$RELDIR" ]
then
   RELDIR=$ROOTDIR/releases
fi

START_ERL_DATA=${1:-$RELDIR/start_erl.data}

$ROOTDIR/bin/run_erl /tmp/ $ROOTDIR/log "exec $ROOTDIR/bin/start_erl \
      $ROOTDIR $RELDIR $START_ERL_DATA -heart -sname cp1" > /dev/null 2>&1 &
```

If a diskless and/or read-only client node is about to start the `start_erl.data` file is located in the client directory at the master node. Thus, the `START_ERL_DATA` line should look like:

```
CLIENTDIR=$ROOTDIR/clients/clientname
START_ERL_DATA=${1:-$CLIENTDIR/bin/start_erl.data}
```

run_erl

This program is used to start the emulator, but you will not be connected to the shell. `to_erl` is used to connect to the Erlang shell.

```
Usage: run_erl pipe_dir/ log_dir "exec command [parameters ...]"
```

Where `pipe_dir/` should be `/tmp/` (`to_erl` uses this name by default) and `log_dir` is where the log files are written. `command [parameters]` is executed, and everything written to stdin and stdout is logged in the `log_dir`.

In the `log_dir`, log files are written. Each logfile has a name of the form: `erlang.log.N` where N is a generation number, ranging from 1 to 5. Each logfile holds up to 100kB text. As time goes by the following logfiles will be found in the logfile directory

```
erlang.log.1
erlang.log.1, erlang.log.2
erlang.log.1, erlang.log.2, erlang.log.3
erlang.log.1, erlang.log.2, erlang.log.3, erlang.log.4
erlang.log.2, erlang.log.3, erlang.log.4, erlang.log.5
erlang.log.3, erlang.log.4, erlang.log.5, erlang.log.1
...
```

with the most recent logfile being the right most in each row of the above list. That is, the most recent file is the one with the highest number, or if there are already four files, the one before the skip.

When a logfile is opened (for appending or created) a time stamp is written to the file. If nothing has been written to the log files for 15 minutes, a record is inserted that says that we're still alive.

to_erl

This program is used to attach to a running Erlang runtime system, started with `run_erl`.

```
Usage: to_erl [pipe_name | pipe_dir]
```

Where `pipe_name` defaults to `/tmp/erlang.pipe.N`.

To disconnect from the shell without exiting the Erlang system, type `Ctrl-D`.

start_erl

This program starts the Erlang emulator with parameters `-boot` and `-config` set. It reads data about where these files are located from a file called `start_erl.data` which is located in the `<RELDIR>`. Each new release introduces a new data file. This file is automatically generated by the release handler in Erlang.

The following script illustrates the behaviour of the program.

```
#!/bin/sh
#
# This program is called by run_erl.  It starts
# the Erlang emulator and sets -boot and -config parameters.
# It should only be used at an embedded target system.
#
# Usage: start_erl RootDir RelDir DataFile [ErlFlags ...]
#
ROOTDIR=$1
shift
RELDIR=$1
shift
DataFile=$1
shift

ERTS_VSN=`awk '{print $1}' $DataFile`
VSN=`awk '{print $2}' $DataFile`

BINDIR=$ROOTDIR/erts-$ERTS_VSN/bin
EMU=beam
PROGNAME=`echo $0 | sed 's/.*\///'`
export EMU
export ROOTDIR
export BINDIR
export PROGNAME
export RELDIR

exec $BINDIR/erlexec -boot $RELDIR/$VSN/start -config $RELDIR/$VSN/sys $*
```

If a diskless and/or read-only client node with the `sasl` configuration parameter `static_emulator` set to `true` is about to start the `-boot` and `-config` flags must be changed. As such a client cannot read a new `start_erl.data` file (the file is not possible to change dynamically) the boot and config files are always fetched from the same place (but with new contents if a new release has been installed). The `release_handler` copies this files to the `bin` directory in the client directory at the master nodes whenever a new release is made permanent.

Assuming the same `CLIENTDIR` as above the last line should look like:

```
exec $BINDIR/erlexec -boot $CLIENTDIR/bin/start \
     -config $CLIENTDIR/bin/sys $*
```

3.2 Windows NT

This chapter describes the OS specific parts of OTP which relate to Windows NT.

3.2.1 Introduction

A normal installation of NT 4.0, with service pack 4 or later, is required for an embedded Windows NT running OTP.

3.2.2 Memory Usage

RAM memory of 96 MBytes is recommended to run OTP on NT. A system with less than 64 Mbytes of RAM is not recommended.

3.2.3 Disk Space Usage

A minimum NT installation with networking needs 250 MB, and an additional 130 MB for the swap file.

3.2.4 Installation

Normal NT installation is performed. No additional application programs are needed, such as Internet explorer or web server. Networking with TCP/IP is required.
Service pack 4 or later must be installed.

Hardware Watchdog

For Windows NT running on standard PCs with ISA and/or PCI bus there is a possibility to install an extension card with a hardware watchdog.

See also the `heart(3)` reference manual page in *Kernel*.

3.2.5 Starting Erlang

On an embedded system, the `erlsrv` module should be used, to install the erlang process as a Windows system service. This service can start after NT has booted. See documentation for `erlsrv`.

3.3 VxWorks

This chapter describes the OS specific parts of OTP which relate to VxWorks.

3.3.1 Introduction

The Erlang/OTP distribution for VxWorks is limited to what Switchboard requires (Switchboard is a general purpose switching hardware developed by Ericsson).

Please consult the README file, included at root level in the installation, for latest information on the distribution.

3.3.2 Memory Usage

Memory required is 32 Mbyte.

3.3.3 Disk Usage

The disk space required is 22 Mbyte, the documentation included.

3.3.4 Installation

OTP/VxWorks is supplied in a distribution file named `<PREFIX>.tar.gz`; i.e. a tar archive that is compressed with gzip. `<PREFIX>` represents the name of the release, e.g. `otp_LXA12345_vxworks_cpu32_R42A`. Assuming you are installing to a Solaris file system, the installation is performed by following these steps: <

- Change to the directory where you want to install OTP/VxWorks (`<ROOTDIR>`): `cd <ROOTDIR>`
- Make a directory to put OTP/VxWorks in: `mkdir otp_vxworks_cpu32` (or whatever you want to call it)
- Change directory to the newly created one: `cd otp_vxworks_cpu32`
- Copy the distribution file there from where it is located (`<RELDIR>`): `cp <RELDIR>/ <PREFIX>.tar.gz .`
- Unzip the distribution file: `gunzip <PREFIX>.tar.gz`
- Untar `<PREFIX>.tar`: `tar xvf <PREFIX>.tar`
- Create a bin directory: `mkdir bin`
- Copy the VxWorks Erlang/OTP start-up script to the bin directory: `cp erts-Vsn/bin/erl bin/.`
- Copy the example start scripts to the bin directory: `cp releases/R42A/*.boot bin/.`

If you use VxWorks nfs mounting facility to mount the Solaris file system, this installation may be directly used. An other possibility is to copy the installation to a local VxWorks DOS file system, from where it is used.

3.3.5 OS Specific Functionality/Information

There are a couple of files that are unique to the VxWorks distribution of Erlang/OTP, these files are described here.

- README - this files has some information on VxWorks specifics that you are advised to consult. This includes the latest information on what parts of OTP are included in the VxWorks distribution of Erlang/OTP. If you want us to include more parts, please contact us to discuss this.
- erts-Vsn/bin/resolv.conf - A resolver configuration EXAMPLE file. You have to edit this file.
- erts-Vsn/bin/erl - This is an EXAMPLE start script for VxWorks. You have to edit this file to suit your needs.
- erts-Vsn/bin/erl_io - One possible solution to the problem of competing Erlang and VxWorks shell. Contains the function 'start_erl' called by the erl script. Also contains the function 'to_erl' to be used when connecting to the Erlang shell from VxWorks' shell.
- erts-Vsn/bin/erl_exec - Rearranges command line arguments and starts Erlang.
- erts-Vsn/bin/vxcall - Allows spawning of standard VxWorks shell functions (which is just about any function in the system...) from open_port/2. E.g. open_port({spawn, 'vxcall func arg1 arg2'}, []) will cause the output that 'func arg1, arg2' would have given in the shell to be received from the port.
- erts-Vsn/bin/rdate - Set the time from a networked host, like the SunOS command. Nothing Erlang-specific, but nice if you want date/0 and time/0 to give meaningful values (you also need a TIMEZONE environment setting if GMT isn't acceptable). For example: `putenv "TIMEZONE=CET::-60:033002:102603"` sets central european time.
- erts-Vsn/src - Contains source for the above files, and additionally config.c, driver.h, preload.c and reclaim.h. Reclaim.h defines the interface to a simple mechanism for "resource reclamation" that is part of the Erlang runtime system - may be useful to "port program" writers (and possibly others). Take careful note of the caveats listed in the file!

3.3.6 Starting Erlang

Start (and restart) of the system depends on what file system is used. To be able to start the system from a nfs mounted file system you can use VxWorks start script facility to run a start script similar to the example below. Note that the Erlang/OTP start-up script is run at the end of this script.

```
#  start.script v1.0 1997/09/08 patrik
#
#  File name:  start.script
#  Purpose:    Starting the VxWorks/cpu32 erlang/OTP
#  Author:     patrik@erix.ericsson.se
#  Resides in: ~tornado/wind/target/config/ads360/

#
# Set shell prompt
#
shellPromptSet("sauron-> ")

#
# Set default gateway
#
hostAdd "router-20","150.236.20.251"
routeAdd "0","router-20"

#
# Mount /home from gandalf
#
```

```
hostAdd "gandalf","150.236.20.16"
usergroup=10
nfsAuthUnixSet("gandalf", 452, 10, 1, &usergroup)
nfsMount("gandalf", "/export/home", "/home")

#
# Load and run rdate.o to set correct date on the target
#
ld < /home/gandalf/tornado/wind/target/config/ads360/rdate.o
rdate("gandalf")

#
# Setup timezone information (Central European time)
#
putenv "TIMEZONE=CET::-60:033002:102603"

#
# Run the Erlang/OTP start script
#
cd "/home/gandalf/tornado/wind/target/erlang_cpu32_R42A/bin"
<erl
```

4 User's Guide

4.1 Introduction

4.1.1 Introduction

This is a "kick start" tutorial to get you started with Erlang. Everything here is true, but only part of the truth. For example, I'll only tell you the simplest form of the syntax, not all esoteric forms. Where I've greatly oversimplified things I'll write *manual* which means there is lots more information to be found in the Erlang book or in the *Erlang Reference Manual*.

I also assume that this isn't the first time you have touched a computer and you have a basic idea about how they are programmed. Don't worry, I won't assume you're a wizard programmer.

4.1.2 Things Left Out

In particular the following has been omitted:

- References
- Local error handling (catch/throw)
- Single direction links (monitor)
- Handling of binary data (binaries / bit syntax)
- List comprehensions
- How to communicate with the outside world and/or software written in other languages (ports). There is however a separate tutorial for this, *Interoperability Tutorial*
- Very few of the Erlang libraries have been touched on (for example file handling)
- OTP has been totally skipped and in consequence the Mnesia database has been skipped.
- Hash tables for Erlang terms (ETS)
- Changing code in running systems

4.2 Sequential Programming

4.2.1 The Erlang Shell

Most operating systems have a command interpreter or shell, Unix and Linux have many, Windows has the Command Prompt. Erlang has its own shell where you can directly write bits of Erlang code and evaluate (run) them to see what happens (see *shell(3)*). Start the Erlang shell (in Linux or UNIX) by starting a shell or command interpreter in your operating system and typing `erl`, you will see something like this.

```
% erl
Erlang (BEAM) emulator version 5.2 [source] [hipe]

Eshell V5.2  (abort with ^G)
1>
```

Now type in "2 + 5." as shown below.

```
1> 2 + 5.
7
2>
```

In Windows, the shell is started by double-clicking on the Erlang shell icon.

You'll notice that the Erlang shell has numbered the lines that can be entered, (as 1> 2>) and that it has correctly told you that 2 + 5 is 7! Also notice that you have to tell it you are done entering code by finishing with a full stop "." and a carriage return. If you make mistakes writing things in the shell, you can delete things by using the backspace key as in most shells. There are many more editing commands in the shell (See the chapter *"tty - A command line interface"* in ERTS User's Guide).

(Note: you will find a lot of line numbers given by the shell out of sequence in this tutorial as it was written and the code tested in several sessions).

Now let's try a more complex calculation.

```
2> (42 + 77) * 66 / 3.
2618.0
```

Here you can see the use of brackets and the multiplication operator "*" and division operator "/", just as in normal arithmetic (see the chapter *"Arithmetic Expressions"* in the Erlang Reference Manual).

To shutdown the Erlang system and the Erlang shell type Control-C. You will see the following output:

```
BREAK: (a)bort (c)ontinue (p)roc info (i)nfo (l)oaded
       (v)ersion (k)ill (D)b-tables (d)istribution
a
%
```

Type "a" to leave the Erlang system.

Another way to shutdown the Erlang system is by entering `halt()`:

```
3> halt().
%
```

4.2.2 Modules and Functions

A programming language isn't much use if you can just run code from the shell. So here is a small Erlang program. Enter it into a file called `tut.erl` (the file name `tut.erl` is important, also make sure that it is in the same directory as the one where you started `erl`) using a suitable text editor. If you are lucky your editor will have an Erlang mode which will make it easier for you to enter and format your code nicely (see the chapter *"The Erlang mode for Emacs"* in Tools User's Guide), but you can manage perfectly well without. Here's the code to enter:

```
-module(tut).
-export([double/1]).

double(X) ->
    2 * X.
```

It's not hard to guess that this "program" doubles the value of numbers. I'll get back to the first two lines later. Let's compile the program. This can be done in your Erlang shell as shown below:

```
3> c(tut).
{ok,tut}
```

The `{ok,tut}` tells you that the compilation was OK. If it said "error" instead, you have made some mistake in the text you entered and there will also be error messages to give you some idea as to what has gone wrong so you can change what you have written and try again.

Now lets run the program.

```
4> tut:double(10).
20
```

As expected double of 10 is 20.

Now let's get back to the first two lines. Erlang programs are written in files. Each file contains what we call an Erlang *module*. The first line of code in the module tells us the name of the module (see the chapter *"Modules"* in the Erlang Reference Manual).

```
-module(tut).
```

This tells us that the module is called *tut*. Note the "." at the end of the line. The files which are used to store the module must have the same name as the module but with the extension ".erl". In our case the file name is `tut.erl`. When we use a function in another module, we use the syntax, `module_name:function_name(arguments)`. So

```
4> tut:double(10).
```

means call function `double` in module `tut` with argument "10".

The second line:

```
-export([double/1]).
```

says that the module `tut` contains a function called `double` which takes one argument (`X` in our example) and that this function can be called from outside the module `tut`. More about this later. Again note the "." at the end of the line.

Now for a more complicated example, the factorial of a number (e.g. factorial of 4 is 4 * 3 * 2 * 1). Enter the following code in a file called `tut1.erl`.

```
-module(tut1).
-export([fac/1]).

fac(1) ->
    1;
fac(N) ->
    N * fac(N - 1).
```

Compile the file

```
5> c(tut1).
{ok,tut1}
```

And now calculate the factorial of 4.

```
6> tut1:fac(4).
24
```

The first part:

```
fac(1) ->
    1;
```

says that the factorial of 1 is 1. Note that we end this part with a ";" which indicates that there is more of this function to come. The second part:

```
fac(N) ->
    N * fac(N - 1).
```

says that the factorial of N is N multiplied by the factorial of N - 1. Note that this part ends with a "." saying that there are no more parts of this function.

A function can have many arguments. Let's expand the module `tut1` with the rather stupid function to multiply two numbers:

```
-module(tut1).
-export([fac/1, mult/2]).

fac(1) ->
    1;
fac(N) ->
    N * fac(N - 1).

mult(X, Y) ->
    X * Y.
```

Note that we have also had to expand the `-export` line with the information that there is another function `mult` with two arguments.

Compile:

```
7> c(tut1).
{ok,tut1}
```

and try it out:

```
8> tut1:mult(3,4).
12
```

In the example above the numbers are integers and the arguments in the functions in the code, `N`, `X`, `Y` are called variables. Variables must start with a capital letter (see the chapter *"Variables"* in the Erlang Reference Manual). Examples of variable could be `Number`, `ShoeSize`, `Age` etc.

4.2.3 Atoms

Atoms are another data type in Erlang. Atoms start with a small letter ((see the chapter *"Atom"* in the Erlang Reference Manual)), for example: `charles`, `centimeter`, `inch`. Atoms are simply names, nothing else. They are not like variables which can have a value.

Enter the next program (file: `tut2.erl`) which could be useful for converting from inches to centimeters and vice versa:

```
-module(tut2).
-export([convert/2]).

convert(M, inch) ->
    M / 2.54;

convert(N, centimeter) ->
    N * 2.54.
```

Compile and test:

```
9> c(tut2).
{ok,tut2}
10> tut2:convert(3, inch).
1.1811023622047243
11> tut2:convert(7, centimeter).
17.78
```

Notice that I have introduced decimals (floating point numbers) without any explanation, but I guess you can cope with that.

See what happens if I enter something other than centimeter or inch in the convert function:

```
12> tut2:convert(3, miles).
** exception error: no function clause matching tut2:convert(3,miles)
```

The two parts of the `convert` function are called its clauses. Here we see that "miles" is not part of either of the clauses. The Erlang system can't *match* either of the clauses so we get an error message `function_clause`. The shell formats the error message nicely, but the error tuple is saved in the shell's history list and can be output by the shell command `v/1`:

```
13> v(12).
{'EXIT',{function_clause,[{tut2,convert,[3,miles]},
                          {erl_eval,do_apply,5},
                          {shell,exprs,6},
```

```
                          {shell,eval_exprs,6},
                          {shell,eval_loop,3}]}}
```

4.2.4 Tuples

Now the `tut2` program is hardly good programming style. Consider:

```
tut2:convert(3, inch).
```

Does this mean that 3 is in inches? or that 3 is in centimeters and we want to convert it to inches? So Erlang has a way to group things together to make things more understandable. We call these *tuples*. Tuples are surrounded by "{" and "}".

So we can write `{inch,3}` to denote 3 inches and `{centimeter,5}` to denote 5 centimeters. Now let's write a new program which converts centimeters to inches and vice versa. (file `tut3.erl`).

```
-module(tut3).
-export([convert_length/1]).

convert_length({centimeter, X}) ->
    {inch, X / 2.54};
convert_length({inch, Y}) ->
    {centimeter, Y * 2.54}.
```

Compile and test:

```
14> c(tut3).
{ok,tut3}
15> tut3:convert_length({inch, 5}).
{centimeter,12.7}
16> tut3:convert_length(tut3:convert_length({inch, 5})).
{inch,5.0}
```

Note on line 16 we convert 5 inches to centimeters and back again and reassuringly get back to the original value. I.e the argument to a function can be the result of another function. Pause for a moment and consider how line 16 (above) works. The argument we have given the function `{inch,5}` is first matched against the first head clause of `convert_length` i.e. `convert_length({centimeter,X})` where it can be seen that `{centimeter,X}` does not match `{inch,5}` (the head is the bit before the "->"). This having failed, we try the head of the next clause i.e. `convert_length({inch,Y})`, this matches and `Y` get the value 5.

We have shown tuples with two parts above, but tuples can have as many parts as we want and contain any valid Erlang *term*. For example, to represent the temperature of various cities of the world we could write

```
{moscow, {c, -10}}
{cape_town, {f, 70}}
{paris, {f, 28}}
```

Tuples have a fixed number of things in them. We call each thing in a tuple an element. So in the tuple `{moscow, {c,-10}}`, element 1 is `moscow` and element 2 is `{c,-10}`. I have chosen `c` meaning Centigrade (or Celsius) and `f` meaning Fahrenheit.

4.2.5 Lists

Whereas tuples group things together, we also want to be able to represent lists of things. Lists in Erlang are surrounded by "[" and "]". For example a list of the temperatures of various cities in the world could be:

```
[{moscow, {c, -10}}, {cape_town, {f, 70}}, {stockholm, {c, -4}},
 {paris, {f, 28}}, {london, {f, 36}}]
```

Note that this list was so long that it didn't fit on one line. This doesn't matter, Erlang allows line breaks at all "sensible places" but not, for example, in the middle of atoms, integers etc.

A very useful way of looking at parts of lists, is by using "|". This is best explained by an example using the shell.

```
17> [First |TheRest] = [1,2,3,4,5].
[1,2,3,4,5]
18> First.
1
19> TheRest.
[2,3,4,5]
```

We use | to separate the first elements of the list from the rest of the list. (`First` has got value 1 and `TheRest` value [2,3,4,5]).

Another example:

```
20> [E1, E2 | R] = [1,2,3,4,5,6,7].
[1,2,3,4,5,6,7]
21> E1.
1
22> E2.
2
23> R.
[3,4,5,6,7]
```

Here we see the use of | to get the first two elements from the list. Of course if we try to get more elements from the list than there are elements in the list we will get an error. Note also the special case of the list with no elements [].

```
24> [A, B | C] = [1, 2].
[1,2]
25> A.
1
26> B.
2
27> C.
[]
```

In all the examples above, I have been using new variable names, not reusing the old ones: `First`, `TheRest`, `E1`, `E2`, `R`, `A`, `B`, `C`. The reason for this is that a variable can only be given a value once in its context (scope). I'll get back to this later, it isn't so peculiar as it sounds!

The following example shows how we find the length of a list:

```
```

```
-module(tut4).

-export([list_length/1]).

list_length([]) ->
    0;
list_length([First | Rest]) ->
    1 + list_length(Rest).
```

Compile (file `tut4.erl`) and test:

```
28> c(tut4).
{ok,tut4}
29> tut4:list_length([1,2,3,4,5,6,7]).
7
```

Explanation:

```
list_length([]) ->
    0;
```

The length of an empty list is obviously 0.

```
list_length([First | Rest]) ->
    1 + list_length(Rest).
```

The length of a list with the first element `First` and the remaining elements `Rest` is 1 + the length of `Rest`.

(Advanced readers only: This is not tail recursive, there is a better way to write this function).

In general we can say we use tuples where we would use "records" or "structs" in other languages and we use lists when we want to represent things which have varying sizes, (i.e. where we would use linked lists in other languages).

Erlang does not have a string date type, instead strings can be represented by lists of ASCII characters. So the list `[97,98,99]` is equivalent to "abc". The Erlang shell is "clever" and guesses the what sort of list we mean and outputs it in what it thinks is the most appropriate form, for example:

```
30> [97,98,99].
"abc"
```

4.2.6 Standard Modules and Manual Pages

Erlang has a lot of standard modules to help you do things. For example, the module `io` contains a lot of functions to help you do formatted input/output. To look up information about standard modules, the command `erl -man` can be used at the operating shell or command prompt (i.e. at the same place as that where you started `erl`). Try the operating system shell command:

```
% erl -man io
ERLANG MODULE DEFINITION                                io(3)

MODULE
```

```
     io - Standard I/O Server Interface Functions

DESCRIPTION
     This module provides an  interface  to  standard  Erlang  IO
     servers. The output functions all return ok if they are suc-
     ...
```

If this doesn't work on your system, the documentation is included as HTML in the Erlang/OTP release, or you can read the documentation as HTML or download it as PDF from either of the sites www.erlang.se (commercial Erlang) or www.erlang.org (open source), for example for release R9B:

```
http://www.erlang.org/doc/r9b/doc/index.html
```

4.2.7 Writing Output to a Terminal

It's nice to be able to do formatted output in these example, so the next example shows a simple way to use to use the `io:format` function. Of course, just like all other exported functions, you can test the `io:format` function in the shell:

```
31> io:format("hello world~n", []).
hello world
ok
32> io:format("this outputs one Erlang term: ~w~n", [hello]).
this outputs one Erlang term: hello
ok
33> io:format("this outputs two Erlang terms: ~w~w~n", [hello, world]).
this outputs two Erlang terms: helloworld
ok
34> io:format("this outputs two Erlang terms: ~w ~w~n", [hello, world]).
this outputs two Erlang terms: hello world
ok
```

The function `format/2` (i.e. `format` with two arguments) takes two lists. The first one is nearly always a list written between " ". This list is printed out as it stands, except that each ~w is replaced by a term taken in order from the second list. Each ~n is replaced by a new line. The `io:format/2` function itself returns the atom `ok` if everything goes as planned. Like other functions in Erlang, it crashes if an error occurs. This is not a fault in Erlang, it is a deliberate policy. Erlang has sophisticated mechanisms to handle errors which we will show later. As an exercise, try to make `io:format` crash, it shouldn't be difficult. But notice that although `io:format` crashes, the Erlang shell itself does not crash.

4.2.8 A Larger Example

Now for a larger example to consolidate what we have learnt so far. Assume we have a list of temperature readings from a number of cities in the world. Some of them are in Celsius (Centigrade) and some in Fahrenheit (as in the previous list). First let's convert them all to Celsius, then let's print out the data neatly.

```
%% This module is in file tut5.erl

-module(tut5).
-export([format_temps/1]).

%% Only this function is exported
format_temps([])->                      % No output for an empty list
```

```
    ok;
format_temps([City | Rest]) ->
    print_temp(convert_to_celsius(City)),
    format_temps(Rest).

convert_to_celsius({Name, {c, Temp}}) ->  % No conversion needed
    {Name, {c, Temp}};
convert_to_celsius({Name, {f, Temp}}) ->  % Do the conversion
    {Name, {c, (Temp - 32) * 5 / 9}}.

print_temp({Name, {c, Temp}}) ->
    io:format("~-15w ~w c~n", [Name, Temp]).
```

```
35> c(tut5).
{ok,tut5}
36> tut5:format_temps([{moscow, {c, -10}}, {cape_town, {f, 70}},
{stockholm, {c, -4}}, {paris, {f, 28}}, {london, {f, 36}}]).
moscow          -10 c
cape_town       21.11111111111111 c
stockholm       -4 c
paris           -2.2222222222222223 c
london          2.2222222222222223 c
ok
```

Before we look at how this program works, notice that we have added a few comments to the code. A comment starts with a % character and goes on to the end of the line. Note as well that the `-export([format_temps/1]).` line only includes the function `format_temps/1`, the other functions are *local* functions, i.e. they are not visible from outside the module `tut5`.

Note as well that when testing the program from the shell, I had to spread the input over two lines as the line was too long.

When we call `format_temps` the first time, `City` gets the value `{moscow,{c,-10}}` and `Rest` is the rest of the list. So we call the function `print_temp(convert_to_celsius({moscow,{c,-10}}))`.

Here we see a function call as `convert_to_celsius({moscow,{c,-10}})` as the argument to the function `print_temp`. When we *nest* function calls like this we execute (evaluate) them from the inside out. I.e. we first evaluate `convert_to_celsius({moscow,{c,-10}})` which gives the value `{moscow,{c,-10}}` as the temperature is already in Celsius and then we evaluate `print_temp({moscow,{c,-10}})`. The function `convert_to_celsius` works in a similar way to the `convert_length` function in the previous example.

`print_temp` simply calls `io:format` in a similar way to what has been described above. Note that ~-15w says to print the "term" with a field length (width) of 15 and left justify it. (*io(3)*).

Now we call `format_temps(Rest)` with the rest of the list as an argument. This way of doing things is similar to the loop constructs in other languages. (Yes, this is recursion, but don't let that worry you). So the same `format_temps` function is called again, this time `City` gets the value `{cape_town,{f,70}}` and we repeat the same procedure as before. We go on doing this until the list becomes empty, i.e. [], which causes the first clause `format_temps([])` to match. This simply returns (results in) the atom `ok`, so the program ends.

4.2.9 Matching, Guards and Scope of Variables

It could be useful to find the maximum and minimum temperature in lists like this. Before extending the program to do this, let's look at functions for finding the maximum value of the elements in a list:

```
-module(tut6).
-export([list_max/1]).
```

```
list_max([Head|Rest]) ->
   list_max(Rest, Head).

list_max([], Res) ->
    Res;
list_max([Head|Rest], Result_so_far) when Head > Result_so_far ->
    list_max(Rest, Head);
list_max([Head|Rest], Result_so_far)  ->
    list_max(Rest, Result_so_far).
```

```
37> c(tut6).
{ok,tut6}
38> tut6:list_max([1,2,3,4,5,7,4,3,2,1]).
7
```

First note that we have two functions here with the same name `list_max`. However each of these takes a different number of arguments (parameters). In Erlang these are regarded as completely different functions. Where we need to distinguish between these functions we write `name/arity`, where `name` is the name of the function and `arity` is the number of arguments, in this case `list_max/1` and `list_max/2`.

This is an example where we walk through a list "carrying" a value with us, in this case `Result_so_far`. `list_max/1` simply assumes that the max value of the list is the head of the list and calls `list_max/2` with the rest of the list and the value of the head of the list, in the above this would be `list_max([2,3,4,5,7,4,3,2,1],1)`. If we tried to use `list_max/1` with an empty list or tried to use it with something which isn't a list at all, we would cause an error. Note that the Erlang philosophy is not to handle errors of this type in the function they occur, but to do so elsewhere. More about this later.

In `list_max/2` we walk down the list and use `Head` instead of `Result_so_far` when `Head` > `Result_so_far`. `when` is a special word we use before the -> in the function to say that we should only use this part of the function if the test which follows is true. We call tests of this type a *guard*. If the guard isn't true (we say the guard fails), we try the next part of the function. In this case if `Head` isn't greater than `Result_so_far` then it must be smaller or equal to is, so we don't need a guard on the next part of the function.

Some useful operators in guards are, < less than, > greater than, == equal, >= greater or equal, =< less or equal, /= not equal. (see the chapter *"Guard Sequences"* in the Erlang Reference Manual).

To change the above program to one which works out the minimum value of the element in a list, all we would need to do is to write < instead of >. (But it would be wise to change the name of the function to `list_min` :-).

Remember that I mentioned earlier that a variable could only be given a value once in its scope? In the above we see, for example, that `Result_so_far` has been given several values. This is OK since every time we call `list_max/2` we create a new scope and one can regard the `Result_so_far` as a completely different variable in each scope.

Another way of creating and giving a variable a value is by using the match operator = . So if I write `M = 5`, a variable called `M` will be created and given the value 5. If, in the same scope I then write `M = 6`, I'll get an error. Try this out in the shell:

```
39> M = 5.
5
40> M = 6.
** exception error: no match of right hand side value 6
41> M = M + 1.
** exception error: no match of right hand side value 6
42> N = M + 1.
6
```

The use of the match operator is particularly useful for pulling apart Erlang terms and creating new ones.

```
43> {X, Y} = {paris, {f, 28}}.
{paris,{f,28}}
44> X.
paris
45> Y.
{f,28}
```

Here we see that `X` gets the value `paris` and `Y{f,28}`.

Of course if we try to do the same again with another city, we get an error:

```
46> {X, Y} = {london, {f, 36}}.
** exception error: no match of right hand side value {london,{f,36}}
```

Variables can also be used to improve the readability of programs, for example, in the `list_max/2` function above, we could write:

```
list_max([Head|Rest], Result_so_far) when Head > Result_so_far ->
    New_result_far = Head,
    list_max(Rest, New_result_far);
```

which is possibly a little clearer.

4.2.10 More About Lists

Remember that the | operator can be used to get the head of a list:

```
47> [M1|T1] = [paris, london, rome].
[paris,london,rome]
48> M1.
paris
49> T1.
[london,rome]
```

The | operator can also be used to add a head to a list:

```
50> L1 = [madrid | T1].
[madrid,london,rome]
51> L1.
[madrid,london,rome]
```

Now an example of this when working with lists - reversing the order of a list:

```
-module(tut8).

-export([reverse/1]).

reverse(List) ->
```

```
    reverse(List, []).

reverse([Head | Rest], Reversed_List) ->
    reverse(Rest, [Head | Reversed_List]);
reverse([], Reversed_List) ->
    Reversed_List.
```

```
52> c(tut8).
{ok,tut8}
53> tut8:reverse([1,2,3]).
[3,2,1]
```

Consider how `Reversed_List` is built. It starts as [], we then successively take off the heads of the list to be reversed and add them to the the `Reversed_List`, as shown in the following:

```
reverse([1|2,3], []) =>
    reverse([2,3], [1|[]])

reverse([2|3], [1]) =>
    reverse([3], [2|[1])

reverse([3|[]], [2,1]) =>
    reverse([], [3|[2,1]])

reverse([], [3,2,1]) =>
    [3,2,1]
```

The module `lists` contains a lot of functions for manipulating lists, for example for reversing them, so before you write a list manipulating function it is a good idea to check that one isn't already written for you. (see *lists(3)*).

Now lets get back to the cities and temperatures, but take a more structured approach this time. First let's convert the whole list to Celsius as follows and test the function:

```
-module(tut7).
-export([format_temps/1]).

format_temps(List_of_cities) ->
    convert_list_to_c(List_of_cities).

convert_list_to_c([{Name, {f, F}} | Rest]) ->
    Converted_City = {Name, {c, (F -32)* 5 / 9}},
    [Converted_City | convert_list_to_c(Rest)];

convert_list_to_c([City | Rest]) ->
    [City | convert_list_to_c(Rest)];

convert_list_to_c([]) ->
    [].
```

```
54> c(tut7).
{ok, tut7}.
55> tut7:format_temps([{moscow, {c, -10}}, {cape_town, {f, 70}},
{stockholm, {c, -4}}, {paris, {f, 28}}, {london, {f, 36}}]).
[{moscow,{c,-10}},
```

```
 {cape_town,{c,21.11111111111111}},
 {stockholm,{c,-4}},
 {paris,{c,-2.2222222222222223}},
 {london,{c,2.2222222222222223}}]
```

Looking at this bit by bit:

```
format_temps(List_of_cities) ->
    convert_list_to_c(List_of_cities).
```

Here we see that `format_temps/1` calls `convert_list_to_c/1`. `convert_list_to_c/1` takes off the head of the `List_of_cities`, converts it to Celsius if needed. The | operator is used to add the (maybe) converted to the converted rest of the list:

```
[Converted_City | convert_list_to_c(Rest)];
```

or

```
[City | convert_list_to_c(Rest)];
```

We go on doing this until we get to the end of the list (i.e. the list is empty:

```
convert_list_to_c([]) ->
    [].
```

Now we have converted the list, we add a function to print it:

```
-module(tut7).
-export([format_temps/1]).

format_temps(List_of_cities) ->
    Converted_List = convert_list_to_c(List_of_cities),
    print_temp(Converted_List).

convert_list_to_c([{Name, {f, F}} | Rest]) ->
    Converted_City = {Name, {c, (F -32)* 5 / 9}},
    [Converted_City | convert_list_to_c(Rest)];

convert_list_to_c([City | Rest]) ->
    [City | convert_list_to_c(Rest)];

convert_list_to_c([]) ->
    [].

print_temp([{Name, {c, Temp}} | Rest]) ->
    io:format("~-15w ~w c~n", [Name, Temp]),
    print_temp(Rest);
print_temp([]) ->
    ok.
```

```
56> c(tut7).
{ok,tut7}
57> tut7:format_temps([{moscow, {c, -10}}, {cape_town, {f, 70}},
{stockholm, {c, -4}}, {paris, {f, 28}}, {london, {f, 36}}]).
moscow          -10 c
cape_town       21.11111111111111 c
stockholm       -4 c
paris           -2.2222222222222223 c
london          2.2222222222222223 c
ok
```

We now have to add a function to find the cities with the maximum and minimum temperatures. The program below isn't the most efficient way of doing this as we walk through the list of cities four times. But it is better to first strive for clarity and correctness and to make programs efficient only if really needed.

```
-module(tut7).
-export([format_temps/1]).

format_temps(List_of_cities) ->
    Converted_List = convert_list_to_c(List_of_cities),
    print_temp(Converted_List),
    {Max_city, Min_city} = find_max_and_min(Converted_List),
    print_max_and_min(Max_city, Min_city).

convert_list_to_c([{Name, {f, Temp}} | Rest]) ->
    Converted_City = {Name, {c, (Temp -32)* 5 / 9}},
    [Converted_City | convert_list_to_c(Rest)];

convert_list_to_c([City | Rest]) ->
    [City | convert_list_to_c(Rest)];

convert_list_to_c([]) ->
    [].

print_temp([{Name, {c, Temp}} | Rest]) ->
    io:format("~-15w ~w c~n", [Name, Temp]),
    print_temp(Rest);
print_temp([]) ->
    ok.

find_max_and_min([City | Rest]) ->
    find_max_and_min(Rest, City, City).

find_max_and_min([{Name, {c, Temp}} | Rest],
         {Max_Name, {c, Max_Temp}},
         {Min_Name, {c, Min_Temp}}) ->
    if
        Temp > Max_Temp ->
            Max_City = {Name, {c, Temp}};           % Change
        true ->
            Max_City = {Max_Name, {c, Max_Temp}} % Unchanged
    end,
    if
         Temp < Min_Temp ->
            Min_City = {Name, {c, Temp}};           % Change
        true ->
            Min_City = {Min_Name, {c, Min_Temp}} % Unchanged
    end,
    find_max_and_min(Rest, Max_City, Min_City);

find_max_and_min([], Max_City, Min_City) ->
```

```
    {Max_City, Min_City}.

print_max_and_min({Max_name, {c, Max_temp}}, {Min_name, {c, Min_temp}}) ->
    io:format("Max temperature was ~w c in ~w~n", [Max_temp, Max_name]),
    io:format("Min temperature was ~w c in ~w~n", [Min_temp, Min_name]).
```

```
58> c(tut7).
{ok, tut7}
59> tut7:format_temps([{moscow, {c, -10}}, {cape_town, {f, 70}},
{stockholm, {c, -4}}, {paris, {f, 28}}, {london, {f, 36}}]).
moscow          -10 c
cape_town       21.11111111111111 c
stockholm       -4 c
paris           -2.2222222222222223 c
london          2.2222222222222223 c
Max temperature was 21.11111111111111 c in cape_town
Min temperature was -10 c in moscow
ok
```

4.2.11 If and Case

The function `find_max_and_min` works out the maximum and minimum temperature. We have introduced a new construct here `if`. If works as follows:

```
if
    Condition 1 ->
        Action 1;
    Condition 2 ->
        Action 2;
    Condition 3 ->
        Action 3;
    Condition 4 ->
        Action 4
end
```

Note there is no ";" before `end`! Conditions are the same as guards, tests which succeed or fail. Erlang starts at the top until it finds a condition which succeeds and then it evaluates (performs) the action following the condition and ignores all other conditions and action before the `end`. If no condition matches, there will be a run-time failure. A condition which always is succeeds is the atom, `true` and this is often used last in an `if` meaning do the action following the `true` if all other conditions have failed.

The following is a short program to show the workings of `if`.

```
-module(tut9).
-export([test_if/2]).

test_if(A, B) ->
    if
        A == 5 ->
            io:format("A == 5~n", []),
            a_equals_5;
        B == 6 ->
            io:format("B == 6~n", []),
            b_equals_6;
        A == 2, B == 3 ->                       %i.e. A equals 2 and B equals 3
            io:format("A == 2, B == 3~n", []),
```

```
            a_equals_2_b_equals_3;
        A == 1 ; B == 7 ->                      %i.e. A equals 1 or B equals 7
            io:format("A == 1 ; B == 7~n", []),
            a_equals_1_or_b_equals_7
    end.
```

Testing this program gives:

```
60> c(tut9).
{ok,tut9}
61> tut9:test_if(5,33).
A == 5
a_equals_5
62> tut9:test_if(33,6).
B == 6
b_equals_6
63> tut9:test_if(2, 3).
A == 2, B == 3
a_equals_2_b_equals_3
64> tut9:test_if(1, 33).
A == 1 ; B == 7
a_equals_1_or_b_equals_7
65> tut9:test_if(33, 7).
A == 1 ; B == 7
a_equals_1_or_b_equals_7
66> tut9:test_if(33, 33).
** exception error: no true branch found when evaluating an if expression
     in function  tut9:test_if/2
```

Notice that `tut9:test_if(33,33)` did not cause any condition to succeed so we got the run time error `if_clause`, here nicely formatted by the shell. See the chapter *"Guard Sequences"* in the Erlang Reference Manual for details of the many guard tests available. `case` is another construct in Erlang. Recall that we wrote the `convert_length` function as:

```
convert_length({centimeter, X}) ->
    {inch, X / 2.54};
convert_length({inch, Y}) ->
    {centimeter, Y * 2.54}.
```

We could also write the same program as:

```
-module(tut10).
-export([convert_length/1]).

convert_length(Length) ->
    case Length of
        {centimeter, X} ->
            {inch, X / 2.54};
        {inch, Y} ->
            {centimeter, Y * 2.54}
    end.
```

```
67> c(tut10).
{ok,tut10}
```

```
68> tut10:convert_length({inch, 6}).
{centimeter,15.24}
69> tut10:convert_length({centimeter, 2.5}).
{inch,0.984251968503937}
```

Notice that both `case` and `if` have *return values*, i.e. in the above example `case` returned either `{inch,X/2.54}` or `{centimeter,Y*2.54}`. The behaviour of `case` can also be modified by using guards. An example should hopefully clarify this. The following example tells us the length of a month, given the year. We need to know the year of course, since February has 29 days in a leap year.

```
-module(tut11).
-export([month_length/2]).

month_length(Year, Month) ->
    %% All years divisible by 400 are leap
    %% Years divisible by 100 are not leap (except the 400 rule above)
    %% Years divisible by 4 are leap (except the 100 rule above)
    Leap = if
        trunc(Year / 400) * 400 == Year ->
            leap;
        trunc(Year / 100) * 100 == Year ->
            not_leap;
        trunc(Year / 4) * 4 == Year ->
            leap;
        true ->
            not_leap
    end,
    case Month of
        sep -> 30;
        apr -> 30;
        jun -> 30;
        nov -> 30;
        feb when Leap == leap -> 29;
        feb -> 28;
        jan -> 31;
        mar -> 31;
        may -> 31;
        jul -> 31;
        aug -> 31;
        oct -> 31;
        dec -> 31
    end.
```

```
70> c(tut11).
{ok,tut11}
71> tut11:month_length(2004, feb).
29
72> tut11:month_length(2003, feb).
28
73> tut11:month_length(1947, aug).
31
```

4.2.12 Built In Functions (BIFs)

Built in functions BIFs are functions which for some reason is built in to the Erlang virtual machine. BIFs often implement functionality that is impossible to implement in Erlang or is to inefficient to implement in Erlang. Some

BIFs can be called by use of the function name only but they are by default belonging to the erlang module so for example the call to the BIF `trunc` below is equivalent with a call to `erlang:trunc`.

As you can see, we first find out if a year is leap or not. If a year is divisible by 400, it is a leap year. To find this out we first divide the year by 400 and use the built in function `trunc` (more later) to cut off any decimals. We then multiply by 400 again and see if we get back the same value. For example, year 2004:

```
2004 / 400 = 5.01
trunc(5.01) = 5
5 * 400 = 2000
```

and we can see that we got back 2000 which is not the same as 2004, so 2004 isn't divisible by 400. Year 2000:

```
2000 / 400 = 5.0
trunc(5.0) = 5
5 * 400 = 2000
```

so we have a leap year. The next two tests if the year is divisible by 100 or 4 are done in the same way. The first `if` returns `leap` or `not_leap` which lands up in the variable `Leap`. We use this variable in the guard for `feb` in the following `case` which tells us how long the month is.

This example showed the use of `trunc`, an easier way would be to use the Erlang operator `rem` which gives the remainder after division. For example:

```
74> 2004 rem 400.
4
```

so instead of writing

```
trunc(Year / 400) * 400 == Year ->
    leap;
```

we could write

```
Year rem 400 == 0 ->
    leap;
```

There are many other built in functions (BIF) such as `trunc`. Only a few built in functions can be used in guards, and you cannot use functions you have defined yourself in guards. (see the chapter *"Guard Sequences"* in the Erlang Reference Manual) (Aside for advanced readers: This is to ensure that guards don't have side effects). Let's play with a few of these functions in the shell:

```
75> trunc(5.6).
5
76> round(5.6).
6
77> length([a,b,c,d]).
4
```

```
78> float(5).
5.0
79> is_atom(hello).
true
80> is_atom("hello").
false
81> is_tuple({paris, {c, 30}}).
true
82> is_tuple([paris, {c, 30}]).
false
```

All the above can be used in guards. Now for some which can't be used in guards:

```
83> atom_to_list(hello).
"hello"
84> list_to_atom("goodbye").
goodbye
85> integer_to_list(22).
"22"
```

The 3 BIFs above do conversions which would be difficult (or impossible) to do in Erlang.

4.2.13 Higher Order Functions (Funs)

Erlang, like most modern functional programming languages, has higher order functions. We start with an example using the shell:

```
86> Xf = fun(X) -> X * 2 end.
#Fun<erl_eval.5.123085357>
87> Xf(5).
10
```

What we have done here is to define a function which doubles the value of number and assign this function to a variable. Thus `Xf(5)` returned the value 10. Two useful functions when working with lists are `foreach` and `map`, which are defined as follows:

```
foreach(Fun, [First|Rest]) ->
    Fun(First),
    foreach(Fun, Rest);
foreach(Fun, []) ->
    ok.

map(Fun, [First|Rest]) ->
    [Fun(First)|map(Fun,Rest)];
map(Fun, []) ->
    [].
```

These two functions are provided in the standard module `lists`. `foreach` takes a list and applies a fun to every element in the list, `map` creates a new list by applying a fun to every element in a list. Going back to the shell, we start by using `map` and a fun to add 3 to every element of a list:

```
88> Add_3 = fun(X) -> X + 3 end.
#Fun<erl_eval.5.123085357>
```

```
89> lists:map(Add_3, [1,2,3]).
[4,5,6]
```

Now lets print out the temperatures in a list of cities (yet again):

```
90> Print_City = fun({City, {X, Temp}}) -> io:format("~-15w ~w ~w~n",
[City, X, Temp]) end.
#Fun<erl_eval.5.123085357>
91> lists:foreach(Print_City, [{moscow, {c, -10}}, {cape_town, {f, 70}},
{stockholm, {c, -4}}, {paris, {f, 28}}, {london, {f, 36}}]).
moscow          c -10
cape_town       f 70
stockholm       c -4
paris           f 28
london          f 36
ok
```

We will now define a fun which can be used to go through a list of cities and temperatures and transform them all to Celsius.

```
-module(tut13).

-export([convert_list_to_c/1]).

convert_to_c({Name, {f, Temp}}) ->
    {Name, {c, trunc((Temp - 32) * 5 / 9)}};
convert_to_c({Name, {c, Temp}}) ->
    {Name, {c, Temp}}.

convert_list_to_c(List) ->
    lists:map(fun convert_to_c/1, List).
```

```
92> tut13:convert_list_to_c([{moscow, {c, -10}}, {cape_town, {f, 70}},
{stockholm, {c, -4}}, {paris, {f, 28}}, {london, {f, 36}}]).
[{moscow,{c,-10}},
 {cape_town,{c,21}},
 {stockholm,{c,-4}},
 {paris,{c,-2}},
 {london,{c,2}}]
```

The `convert_to_c` function is the same as before, but we use it as a fun:

```
lists:map(fun convert_to_c/1, List)
```

When we use a function defined elsewhere as a fun we can refer to it as `Function/Arity` (remember that `Arity` = number of arguments). So in the `map` call we write `lists:map(fun convert_to_c/1, List)`. As you can see `convert_list_to_c` becomes much shorter and easier to understand.

The standard module `lists` also contains a function `sort(Fun, List)` where `Fun` is a fun with two arguments. This fun should return `true` if the the first argument is less than the second argument, or else `false`. We add sorting to the `convert_list_to_c`:

```
-module(tut13).

-export([convert_list_to_c/1]).

convert_to_c({Name, {f, Temp}}) ->
    {Name, {c, trunc((Temp - 32) * 5 / 9)}};
convert_to_c({Name, {c, Temp}}) ->
    {Name, {c, Temp}}.

convert_list_to_c(List) ->
    New_list = lists:map(fun convert_to_c/1, List),
    lists:sort(fun({_, {c, Temp1}}, {_, {c, Temp2}}) ->
                       Temp1 < Temp2 end, New_list).
```

```
93> c(tut13).
{ok,tut13}
94> tut13:convert_list_to_c([{moscow, {c, -10}}, {cape_town, {f, 70}},
{stockholm, {c, -4}}, {paris, {f, 28}}, {london, {f, 36}}]).
[{moscow,{c,-10}},
 {stockholm,{c,-4}},
 {paris,{c,-2}},
 {london,{c,2}},
 {cape_town,{c,21}}]
```

In `sort` we use the fun:

```
fun({_, {c, Temp1}}, {_, {c, Temp2}}) -> Temp1 < Temp2 end,
```

Here we introduce the concept of an *anonymous variable* "_". This is simply shorthand for a variable which is going to get a value, but we will ignore the value. This can be used anywhere suitable, not just in fun's. `Temp1 < Temp2` returns `true` if `Temp1` is less than `Temp2`.

4.3 Concurrent Programming

4.3.1 Processes

One of the main reasons for using Erlang instead of other functional languages is Erlang's ability to handle concurrency and distributed programming. By concurrency we mean programs which can handle several threads of execution at the same time. For example, modern operating systems would allow you to use a word processor, a spreadsheet, a mail client and a print job all running at the same time. Of course each processor (CPU) in the system is probably only handling one thread (or job) at a time, but it swaps between the jobs a such a rate that it gives the illusion of running them all at the same time. It is easy to create parallel threads of execution in an Erlang program and it is easy to allow these threads to communicate with each other. In Erlang we call each thread of execution a *process*.

(Aside: the term "process" is usually used when the threads of execution share no data with each other and the term "thread" when they share data in some way. Threads of execution in Erlang share no data, that's why we call them processes).

The Erlang BIF `spawn` is used to create a new process: `spawn(Module, Exported_Function, List of Arguments)`. Consider the following module:

```
-module(tut14).
```

```
-export([start/0, say_something/2]).

say_something(What, 0) ->
    done;
say_something(What, Times) ->
    io:format("~p~n", [What]),
    say_something(What, Times - 1).

start() ->
    spawn(tut14, say_something, [hello, 3]),
    spawn(tut14, say_something, [goodbye, 3]).
```

```
5> c(tut14).
{ok,tut14}
6> tut14:say_something(hello, 3).
hello
hello
hello
done
```

We can see that function `say_something` writes its first argument the number of times specified by second argument. Now look at the function `start`. It starts two Erlang processes, one which writes "hello" three times and one which writes "goodbye" three times. Both of these processes use the function `say_something`. Note that a function used in this way by `spawn` to start a process must be exported from the module (i.e. in the `-export` at the start of the module).

```
9> tut14:start().
hello
goodbye
<0.63.0>
hello
goodbye
hello
goodbye
```

Notice that it didn't write "hello" three times and then "goodbye" three times, but the first process wrote a "hello", the second a "goodbye", the first another "hello" and so forth. But where did the <0.63.0> come from? The return value of a function is of course the return value of the last "thing" in the function. The last thing in the function `start` is

```
spawn(tut14, say_something, [goodbye, 3]).
```

`spawn` returns a *process identifier*, or *pid*, which uniquely identifies the process. So <0.63.0> is the pid of the `spawn` function call above. We will see how to use pids in the next example.

Note as well that we have used ~p instead of ~w in `io:format`. To quote the manual: "~p Writes the data with standard syntax in the same way as ~w, but breaks terms whose printed representation is longer than one line into many lines and indents each line sensibly. It also tries to detect lists of printable characters and to output these as strings".

4.3.2 Message Passing

In the following example we create two processes which send messages to each other a number of times.

```
-module(tut15).

-export([start/0, ping/2, pong/0]).

ping(0, Pong_PID) ->
    Pong_PID ! finished,
    io:format("ping finished~n", []);

ping(N, Pong_PID) ->
    Pong_PID ! {ping, self()},
    receive
        pong ->
            io:format("Ping received pong~n", [])
    end,
    ping(N - 1, Pong_PID).

pong() ->
    receive
        finished ->
            io:format("Pong finished~n", []);
        {ping, Ping_PID} ->
            io:format("Pong received ping~n", []),
            Ping_PID ! pong,
            pong()
    end.

start() ->
    Pong_PID = spawn(tut15, pong, []),
    spawn(tut15, ping, [3, Pong_PID]).
```

```
1> c(tut15).
{ok,tut15}
2> tut15: start().
<0.36.0>
Pong received ping
Ping received pong
Pong received ping
Ping received pong
Pong received ping
Ping received pong
ping finished
Pong finished
```

The function `start` first creates a process, let's call it "pong":

```
Pong_PID = spawn(tut15, pong, [])
```

This process executes `tut15:pong()`. `Pong_PID` is the process identity of the "pong" process. The function `start` now creates another process "ping".

```
spawn(tut15, ping, [3, Pong_PID]),
```

this process executes

```
```

```
tut15:ping(3, Pong_PID)
```

<0.36.0> is the return value from the `start` function.

The process "pong" now does:

```
receive
    finished ->
        io:format("Pong finished~n", []);
    {ping, Ping_PID} ->
        io:format("Pong received ping~n", []),
        Ping_PID ! pong,
        pong()
end.
```

The `receive` construct is used to allow processes to wait for messages from other processes. It has the format:

```
receive
   pattern1 ->
       actions1;
   pattern2 ->
       actions2;
   ....
   patternN
       actionsN
end.
```

Note: no ";" before the `end`.

Messages between Erlang processes are simply valid Erlang terms. I.e. they can be lists, tuples, integers, atoms, pids etc.

Each process has its own input queue for messages it receives. New messages received are put at the end of the queue. When a process executes a `receive`, the first message in the queue is matched against the first pattern in the `receive`, if this matches, the message is removed from the queue and the actions corresponding to the the pattern are executed.

However, if the first pattern does not match, the second pattern is tested, if this matches the message is removed from the queue and the actions corresponding to the second pattern are executed. If the second pattern does not match the third is tried and so on until there are no more pattern to test. If there are no more patterns to test, the first message is kept in the queue and we try the second message instead. If this matches any pattern, the appropriate actions are executed and the second message is removed from the queue (keeping the first message and any other messages in the queue). If the second message does not match we try the third message and so on until we reach the end of the queue. If we reach the end of the queue, the process blocks (stops execution) and waits until a new message is received and this procedure is repeated.

Of course the Erlang implementation is "clever" and minimizes the number of times each message is tested against the patterns in each `receive`.

Now back to the ping pong example.

"Pong" is waiting for messages. If the atom `finished` is received, "pong" writes "Pong finished" to the output and as it has nothing more to do, terminates. If it receives a message with the format:

```
{ping, Ping_PID}
```

it writes "Pong received ping" to the output and sends the atom `pong` to the process "ping":

```
Ping_PID ! pong
```

Note how the operator "!" is used to send messages. The syntax of "!" is:

```
Pid ! Message
```

I.e. `Message` (any Erlang term) is sent to the process with identity `Pid`.

After sending the message `pong`, to the process "ping", "pong" calls the `pong` function again, which causes it to get back to the `receive` again and wait for another message. Now let's look at the process "ping". Recall that it was started by executing:

```
tut15:ping(3, Pong_PID)
```

Looking at the function `ping/2` we see that the second clause of `ping/2` is executed since the value of the first argument is 3 (not 0) (first clause head is `ping(0,Pong_PID)`, second clause head is `ping(N,Pong_PID)`, so `N` becomes 3).

The second clause sends a message to "pong":

```
Pong_PID ! {ping, self()},
```

`self()` returns the pid of the process which executes `self()`, in this case the pid of "ping". (Recall the code for "pong", this will land up in the variable `Ping_PID` in the `receive` previously explained).

"Ping" now waits for a reply from "pong":

```
receive
    pong ->
        io:format("Ping received pong~n", [])
end,
```

and writes "Ping received pong" when this reply arrives, after which "ping" calls the `ping` function again.

```
ping(N - 1, Pong_PID)
```

`N-1` causes the first argument to be decremented until it becomes 0. When this occurs, the first clause of `ping/2` will be executed:

```
ping(0, Pong_PID) ->
    Pong_PID !  finished,
    io:format("ping finished~n", []);
```

The atom `finished` is sent to "pong" (causing it to terminate as described above) and "ping finished" is written to the output. "Ping" then itself terminates as it has nothing left to do.

4.3.3 Registered Process Names

In the above example, we first created "pong" so as to be able to give the identity of "pong" when we started "ping". I.e. in some way "ping" must be able to know the identity of "pong" in order to be able to send a message to it. Sometimes processes which need to know each others identities are started completely independently of each other. Erlang thus provides a mechanism for processes to be given names so that these names can be used as identities instead of pids. This is done by using the `register` BIF:

```
register(some_atom, Pid)
```

We will now re-write the ping pong example using this and giving the name `pong` to the "pong" process:

```
-module(tut16).

-export([start/0, ping/1, pong/0]).

ping(0) ->
    pong ! finished,
    io:format("ping finished~n", []);

ping(N) ->
    pong ! {ping, self()},
    receive
        pong ->
            io:format("Ping received pong~n", [])
    end,
    ping(N - 1).

pong() ->
    receive
        finished ->
            io:format("Pong finished~n", []);
        {ping, Ping_PID} ->
            io:format("Pong received ping~n", []),
            Ping_PID ! pong,
            pong()
    end.

start() ->
    register(pong, spawn(tut16, pong, [])),
    spawn(tut16, ping, [3]).
```

```
2> c(tut16).
{ok, tut16}
3> tut16:start().
<0.38.0>
Pong received ping
Ping received pong
Pong received ping
Ping received pong
Pong received ping
Ping received pong
ping finished
```

```
Pong finished
```

In the `start/0` function,

```
register(pong, spawn(tut16, pong, [])),
```

both spawns the "pong" process and gives it the name `pong`. In the "ping" process we can now send messages to `pong` by:

```
pong ! {ping, self()},
```

so that `ping/2` now becomes `ping/1` as we don't have to use the argument `Pong_PID`.

4.3.4 Distributed Programming

Now let's re-write the ping pong program with "ping" and "pong" on different computers. Before we do this, there are a few things we need to set up to get this to work. The distributed Erlang implementation provides a basic security mechanism to prevent unauthorized access to an Erlang system on another computer (*manual*). Erlang systems which talk to each other must have the same *magic cookie*. The easiest way to achieve this is by having a file called `.erlang.cookie` in your home directory on all machines which on which you are going to run Erlang systems communicating with each other (on Windows systems the home directory is the directory where pointed to by the $HOME environment variable - you may need to set this. On Linux or Unix you can safely ignore this and simply create a file called `.erlang.cookie` in the directory you get to after executing the command `cd` without any argument). The `.erlang.cookie` file should contain on line with the same atom. For example on Linux or Unix in the OS shell:

```
$ cd
$ cat > .erlang.cookie
this_is_very_secret
$ chmod 400 .erlang.cookie
```

The `chmod` above make the `.erlang.cookie` file accessible only by the owner of the file. This is a requirement.

When you start an Erlang system which is going to talk to other Erlang systems, you must give it a name, eg:

```
$ erl -sname my_name
```

We will see more details of this later (*manual*). If you want to experiment with distributed Erlang, but you only have one computer to work on, you can start two separate Erlang systems on the same computer but give them different names. Each Erlang system running on a computer is called an Erlang node.

(Note: `erl -sname` assumes that all nodes are in the same IP domain and we can use only the first component of the IP address, if we want to use nodes in different domains we use `-name` instead, but then all IP address must be given in full (*manual*).

Here is the ping pong example modified to run on two separate nodes:

```
-module(tut17).

-export([start_ping/1, start_pong/0,  ping/2, pong/0]).
```

```
ping(0, Pong_Node) ->
    {pong, Pong_Node} ! finished,
    io:format("ping finished~n", []);

ping(N, Pong_Node) ->
    {pong, Pong_Node} ! {ping, self()},
    receive
        pong ->
            io:format("Ping received pong~n", [])
    end,
    ping(N - 1, Pong_Node).

pong() ->
    receive
        finished ->
            io:format("Pong finished~n", []);
        {ping, Ping_PID} ->
            io:format("Pong received ping~n", []),
            Ping_PID ! pong,
            pong()
    end.

start_pong() ->
    register(pong, spawn(tut17, pong, [])).

start_ping(Pong_Node) ->
    spawn(tut17, ping, [3, Pong_Node]).
```

Let us assume we have two computers called gollum and kosken. We will start a node on kosken called ping and then a node on gollum called pong.

On kosken (on a Linux/Unix system):

```
kosken> erl -sname ping
Erlang (BEAM) emulator version 5.2.3.7 [hipe] [threads:0]

Eshell V5.2.3.7  (abort with ^G)
(ping@kosken)1>
```

On gollum:

```
gollum> erl -sname pong
Erlang (BEAM) emulator version 5.2.3.7 [hipe] [threads:0]

Eshell V5.2.3.7  (abort with ^G)
(pong@gollum)1>
```

Now we start the "pong" process on gollum:

```
(pong@gollum)1> tut17:start_pong().
true
```

and start the "ping" process on kosken (from the code above you will see that a parameter of the `start_ping` function is the node name of the Erlang system where "pong" is running):

```
(ping@kosken)1> tut17:start_ping(pong@gollum).
<0.37.0>
Ping received pong
Ping received pong
Ping received pong
ping finished
```

Here we see that the ping pong program has run, on the "pong" side we see:

```
(pong@gollum)2>
Pong received ping
Pong received ping
Pong received ping
Pong finished
(pong@gollum)2>
```

Looking at the `tut17` code we see that the `pong` function itself is unchanged, the lines:

```
{ping, Ping_PID} ->
    io:format("Pong received ping~n", []),
    Ping_PID ! pong,
```

work in the same way irrespective of on which node the "ping" process is executing. Thus Erlang pids contain information about where the process executes so if you know the pid of a process, the "!" operator can be used to send it a message if the process is on the same node or on a different node.

A difference is how we send messages to a registered process on another node:

```
{pong, Pong_Node} ! {ping, self()},
```

We use a tuple `{registered_name,node_name}` instead of just the `registered_name`.

In the previous example, we started "ping" and "pong" from the shells of two separate Erlang nodes. `spawn` can also be used to start processes in other nodes. The next example is the ping pong program, yet again, but this time we will start "ping" in another node:

```
-module(tut18).

-export([start/1,  ping/2, pong/0]).

ping(0, Pong_Node) ->
    {pong, Pong_Node} ! finished,
    io:format("ping finished~n", []);

ping(N, Pong_Node) ->
    {pong, Pong_Node} ! {ping, self()},
    receive
        pong ->
            io:format("Ping received pong~n", [])
    end,
    ping(N - 1, Pong_Node).

pong() ->
```

```
    receive
        finished ->
            io:format("Pong finished~n", []);
        {ping, Ping_PID} ->
            io:format("Pong received ping~n", []),
            Ping_PID ! pong,
            pong()
    end.

start(Ping_Node) ->
    register(pong, spawn(tut18, pong, [])),
    spawn(Ping_Node, tut18, ping, [3, node()]).
```

Assuming an Erlang system called ping (but not the "ping" process) has already been started on kosken, then on gollum we do:

```
(pong@gollum)1> tut18:start(ping@kosken).
<3934.39.0>
Pong received ping
Ping received pong
Pong received ping
Ping received pong
Pong received ping
Ping received pong
Pong finished
ping finished
```

Notice we get all the output on gollum. This is because the io system finds out where the process is spawned from and sends all output there.

4.3.5 A Larger Example

Now for a larger example. We will make an extremely simple "messenger". The messenger is a program which allows users to log in on different nodes and send simple messages to each other.

Before we start, let's note the following:

- This example will just show the message passing logic no attempt at all has been made to provide a nice graphical user interface - this can of course also be done in Erlang - but that's another tutorial.
- This sort of problem can be solved more easily if you use the facilities in OTP, which will also provide methods for updating code on the fly etc. But again, that's another tutorial.
- The first program we write will contain some inadequacies as regards handling of nodes which disappear, we will correct these in a later version of the program.

We will set up the messenger by allowing "clients" to connect to a central server and say who and where they are. I.e. a user won't need to know the name of the Erlang node where another user is located to send a message.

File `messenger.erl`:

```
%%% Message passing utility.
%%% User interface:
%%% logon(Name)
%%%     One user at a time can log in from each Erlang node in the
%%%     system messenger: and choose a suitable Name. If the Name
%%%     is already logged in at another node or if someone else is
%%%     already logged in at the same node, login will be rejected
%%%     with a suitable error message.
```

```
%%% logoff()
%%%     Logs off anybody at at node
%%% message(ToName, Message)
%%%     sends Message to ToName. Error messages if the user of this
%%%     function is not logged on or if ToName is not logged on at
%%%     any node.
%%%
%%% One node in the network of Erlang nodes runs a server which maintains
%%% data about the logged on users. The server is registered as "messenger"
%%% Each node where there is a user logged on runs a client process registered
%%% as "mess_client"
%%%
%%% Protocol between the client processes and the server
%%% ----------------------------------------------------
%%%
%%% To server: {ClientPid, logon, UserName}
%%% Reply {messenger, stop, user_exists_at_other_node} stops the client
%%% Reply {messenger, logged_on} logon was successful
%%%
%%% To server: {ClientPid, logoff}
%%% Reply: {messenger, logged_off}
%%%
%%% To server: {ClientPid, logoff}
%%% Reply: no reply
%%%
%%% To server: {ClientPid, message_to, ToName, Message} send a message
%%% Reply: {messenger, stop, you_are_not_logged_on} stops the client
%%% Reply: {messenger, receiver_not_found} no user with this name logged on
%%% Reply: {messenger, sent} Message has been sent (but no guarantee)
%%%
%%% To client: {message_from, Name, Message},
%%%
%%% Protocol between the "commands" and the client
%%% ----------------------------------------------
%%%
%%% Started: messenger:client(Server_Node, Name)
%%% To client: logoff
%%% To client: {message_to, ToName, Message}
%%%
%%% Configuration: change the server_node() function to return the
%%% name of the node where the messenger server runs

-module(messenger).
-export([start_server/0, server/1, logon/1, logoff/0, message/2, client/2]).

%%% Change the function below to return the name of the node where the
%%% messenger server runs
server_node() ->
    messenger@bill.

%%% This is the server process for the "messenger"
%%% the user list has the format [{ClientPid1, Name1},{ClientPid22, Name2},...]
server(User_List) ->
    receive
        {From, logon, Name} ->
            New_User_List = server_logon(From, Name, User_List),
            server(New_User_List);
        {From, logoff} ->
            New_User_List = server_logoff(From, User_List),
            server(New_User_List);
        {From, message_to, To, Message} ->
            server_transfer(From, To, Message, User_List),
            io:format("list is now: ~p~n", [User_List]),
            server(User_List)
    end.
```

```
%%% Start the server
start_server() ->
    register(messenger, spawn(messenger, server, [[]])).


%%% Server adds a new user to the user list
server_logon(From, Name, User_List) ->
    %% check if logged on anywhere else
    case lists:keymember(Name, 2, User_List) of
        true ->
            From ! {messenger, stop, user_exists_at_other_node},  %reject logon
            User_List;
        false ->
            From ! {messenger, logged_on},
            [{From, Name} | User_List]        %add user to the list
    end.

%%% Server deletes a user from the user list
server_logoff(From, User_List) ->
    lists:keydelete(From, 1, User_List).


%%% Server transfers a message between user
server_transfer(From, To, Message, User_List) ->
    %% check that the user is logged on and who he is
    case lists:keysearch(From, 1, User_List) of
        false ->
            From ! {messenger, stop, you_are_not_logged_on};
        {value, {From, Name}} ->
            server_transfer(From, Name, To, Message, User_List)
    end.
%%% If the user exists, send the message
server_transfer(From, Name, To, Message, User_List) ->
    %% Find the receiver and send the message
    case lists:keysearch(To, 2, User_List) of
        false ->
            From ! {messenger, receiver_not_found};
        {value, {ToPid, To}} ->
            ToPid ! {message_from, Name, Message},
            From ! {messenger, sent}
    end.


%%% User Commands
logon(Name) ->
    case whereis(mess_client) of
        undefined ->
            register(mess_client,
                     spawn(messenger, client, [server_node(), Name]));
        _ -> already_logged_on
    end.

logoff() ->
    mess_client ! logoff.

message(ToName, Message) ->
    case whereis(mess_client) of % Test if the client is running
        undefined ->
            not_logged_on;
        _ -> mess_client ! {message_to, ToName, Message},
             ok
end.
```

```
%%% The client process which runs on each server node
client(Server_Node, Name) ->
    {messenger, Server_Node} ! {self(), logon, Name},
    await_result(),
    client(Server_Node).

client(Server_Node) ->
    receive
        logoff ->
            {messenger, Server_Node} ! {self(), logoff},
            exit(normal);
        {message_to, ToName, Message} ->
            {messenger, Server_Node} ! {self(), message_to, ToName, Message},
            await_result();
        {message_from, FromName, Message} ->
            io:format("Message from ~p: ~p~n", [FromName, Message])
    end,
    client(Server_Node).

%%% wait for a response from the server
await_result() ->
    receive
        {messenger, stop, Why} -> % Stop the client
            io:format("~p~n", [Why]),
            exit(normal);
        {messenger, What} ->  % Normal response
            io:format("~p~n", [What])
    end.
```

To use this program you need to:

- configure the `server_node()` function
- copy the compiled code (`messenger.beam`) to the directory on each computer where you start Erlang.

In the following example of use of this program, I have started nodes on four different computers, but if you don't have that many machines available on your network, you could start up several nodes on the same machine.

We start up four Erlang nodes, messenger@super, c1@bilbo, c2@kosken, c3@gollum.

First we start up a the server at messenger@super:

```
(messenger@super)1> messenger:start_server().
true
```

Now Peter logs on at c1@bilbo:

```
(c1@bilbo)1> messenger:logon(peter).
true
logged_on
```

James logs on at c2@kosken:

```
(c2@kosken)1> messenger:logon(james).
true
logged_on
```

and Fred logs on at c3@gollum:

```
(c3@gollum)1> messenger:logon(fred).
true
logged_on
```

Now Peter sends Fred a message:

```
(c1@bilbo)2> messenger:message(fred, "hello").
ok
sent
```

And Fred receives the message and sends a message to Peter and logs off:

```
Message from peter: "hello"
(c3@gollum)2> messenger:message(peter, "go away, I'm busy").
ok
sent
(c3@gollum)3> messenger:logoff().
logoff
```

James now tries to send a message to Fred:

```
(c2@kosken)2> messenger:message(fred, "peter doesn't like you").
ok
receiver_not_found
```

But this fails as Fred has already logged off.

First let's look at some of the new concepts we have introduced.

There are two versions of the `server_transfer` function, one with four arguments (`server_transfer/4`) and one with five (`server_transfer/5`). These are regarded by Erlang as two separate functions.

Note how we write the `server` function so that it calls itself, `server(User_List)` and thus creates a loop. The Erlang compiler is "clever" and optimizes the code so that this really is a sort of loop and not a proper function call. But this only works if there is no code after the call, otherwise the compiler will expect the call to return and make a proper function call. This would result in the process getting bigger and bigger for every loop.

We use functions in the `lists` module. This is a very useful module and a study of the manual page is recommended (`erl -man lists`). `lists:keymember(Key,Position,Lists)` looks through a list of tuples and looks at `Position` in each tuple to see if it is the same as `Key`. The first element is position 1. If it finds a tuple where the element at `Position` is the same as Key, it returns `true`, otherwise `false`.

```
3> lists:keymember(a, 2, [{x,y,z},{b,b,b},{b,a,c},{q,r,s}]).
true
4> lists:keymember(p, 2, [{x,y,z},{b,b,b},{b,a,c},{q,r,s}]).
false
```

`lists:keydelete` works in the same way but deletes the first tuple found (if any) and returns the remaining list:

```
5> lists:keydelete(a, 2, [{x,y,z},{b,b,b},{b,a,c},{q,r,s}]).
```

```
[{x,y,z},{b,b,b},{q,r,s}]
```

`lists:keysearch` is like `lists:keymember`, but it returns `{value,Tuple_Found}` or the atom `false`.

There are a lot more very useful functions in the `lists` module.

An Erlang process will (conceptually) run until it does a `receive` and there is no message which it wants to receive in the message queue. I say "conceptually" because the Erlang system shares the CPU time between the active processes in the system.

A process terminates when there is nothing more for it to do, i.e. the last function it calls simply returns and doesn't call another function. Another way for a process to terminate is for it to call `exit/1`. The argument to `exit/1` has a special meaning which we will look at later. In this example we will do `exit(normal)` which has the same effect as a process running out of functions to call.

The BIF `whereis(RegisteredName)` checks if a registered process of name `RegisteredName` exists and return the pid of the process if it does exist or the atom `undefined` if it does not.

You should by now be able to understand most of the code above so I'll just go through one case: a message is sent from one user to another.

The first user "sends" the message in the example above by:

```
messenger:message(fred, "hello")
```

After testing that the client process exists:

```
whereis(mess_client)
```

and a message is sent to `mess_client`:

```
mess_client ! {message_to, fred, "hello"}
```

The client sends the message to the server by:

```
{messenger, messenger@super} ! {self(), message_to, fred, "hello"},
```

and waits for a reply from the server.

The server receives this message and calls:

```
server_transfer(From, fred, "hello", User_List),
```

which checks that the pid `From` is in the `User_List`:

```
lists:keysearch(From, 1, User_List)
```

If `keysearch` returns the atom `false`, some sort of error has occurred and the server sends back the message:

```
From ! {messenger, stop, you_are_not_logged_on}
```

which is received by the client which in turn does `exit(normal)` and terminates. If `keysearch` returns `{value, {From,Name}}` we know that the user is logged on and is his name (peter) is in variable `Name`. We now call:

```
server_transfer(From, peter, fred, "hello", User_List)
```

Note that as this is `server_transfer/5` it is not the same as the previous function `server_transfer/4`. We do another `keysearch` on `User_List` to find the pid of the client corresponding to fred:

```
lists:keysearch(fred, 2, User_List)
```

This time we use argument 2 which is the second element in the tuple. If this returns the atom `false` we know that fred is not logged on and we send the message:

```
From ! {messenger, receiver_not_found};
```

which is received by the client, if `keysearch` returns:

```
{value, {ToPid, fred}}
```

we send the message:

```
ToPid ! {message_from, peter, "hello"},
```

to fred's client and the message:

```
From ! {messenger, sent}
```

to peter's client.

Fred's client receives the message and prints it:

```
{message_from, peter, "hello"} ->
    io:format("Message from ~p: ~p~n", [peter, "hello"])
```

and peter's client receives the message in the `await_result` function.

4.4 Robustness

There are several things which are wrong with the *messenger example* from the previous chapter. For example if a node where a user is logged on goes down without doing a log off, the user will remain in the server's `User_List`

but the client will disappear thus making it impossible for the user to log on again as the server thinks the user already logged on.

Or what happens if the server goes down in the middle of sending a message leaving the sending client hanging for ever in the `await_result` function?

4.4.1 Timeouts

Before improving the messenger program we will look into some general principles, using the ping pong program as an example. Recall that when "ping" finishes, it tells "pong" that it has done so by sending the atom `finished` as a message to "pong" so that "pong" could also finish. Another way to let "pong" finish, is to make "pong" exit if it does not receive a message from ping within a certain time, this can be done by adding a *timeout* to `pong` as shown in the following example:

```
-module(tut19).

-export([start_ping/1, start_pong/0,  ping/2, pong/0]).

ping(0, Pong_Node) ->
    io:format("ping finished~n", []);

ping(N, Pong_Node) ->
    {pong, Pong_Node} ! {ping, self()},
    receive
        pong ->
            io:format("Ping received pong~n", [])
    end,
    ping(N - 1, Pong_Node).

pong() ->
    receive
        {ping, Ping_PID} ->
            io:format("Pong received ping~n", []),
            Ping_PID ! pong,
            pong()
    after 5000 ->
            io:format("Pong timed out~n", [])
    end.

start_pong() ->
    register(pong, spawn(tut19, pong, [])).

start_ping(Pong_Node) ->
    spawn(tut19, ping, [3, Pong_Node]).
```

After we have compiled this and copied the `tut19.beam` file to the necessary directories:

On (pong@kosken):

```
(pong@kosken)1> tut19:start_pong().
true
Pong received ping
Pong received ping
Pong received ping
Pong timed out
```

On (ping@gollum):

```
(ping@gollum)1> tut19:start_ping(pong@kosken).
<0.36.0>
Ping received pong
Ping received pong
Ping received pong
ping finished
```

(The timeout is set in:

```
pong() ->
    receive
        {ping, Ping_PID} ->
            io:format("Pong received ping~n", []),
            Ping_PID ! pong,
            pong()
    after 5000 ->
            io:format("Pong timed out~n", [])
    end.
```

We start the timeout (`after 5000`) when we enter `receive`. The timeout is canceled if `{ping,Ping_PID}` is received. If `{ping,Ping_PID}` is not received, the actions following the timeout will be done after 5000 milliseconds. `after` must be last in the `receive`, i.e. preceded by all other message reception specifications in the `receive`. Of course we could also call a function which returned an integer for the timeout:

```
after pong_timeout() ->
```

In general, there are better ways than using timeouts to supervise parts of a distributed Erlang system. Timeouts are usually appropriate to supervise external events, for example if you have expected a message from some external system within a specified time. For example, we could use a timeout to log a user out of the messenger system if they have not accessed it, for example, in ten minutes.

4.4.2 Error Handling

Before we go into details of the supervision and error handling in an Erlang system, we need see how Erlang processes terminate, or in Erlang terminology, *exit*.

A process which executes `exit(normal)` or simply runs out of things to do has a *normal* exit.

A process which encounters a runtime error (e.g. divide by zero, bad match, trying to call a function which doesn't exist etc) exits with an error, i.e. has an *abnormal* exit. A process which executes *exit(Reason)* where `Reason` is any Erlang term except the atom `normal`, also has an abnormal exit.

An Erlang process can set up links to other Erlang processes. If a process calls *link(Other_Pid)* it sets up a bidirectional link between itself and the process called `Other_Pid`. When a process terminates its sends something called a *signal* to all the processes it has links to.

The signal carries information about the pid it was sent from and the exit reason.

The default behaviour of a process which receives a normal exit is to ignore the signal.

The default behaviour in the two other cases (i.e. abnormal exit) above is to bypass all messages to the receiving process and to kill it and to propagate the same error signal to the killed process' links. In this way you can connect all processes in a transaction together using links and if one of the processes exits abnormally, all the processes in the transaction will be killed. As we often want to create a process and link to it at the same time, there is a special BIF, *spawn_link* which does the same as `spawn`, but also creates a link to the spawned process.

Now an example of the ping pong example using links to terminate "pong":

```
-module(tut20).

-export([start/1,  ping/2, pong/0]).

ping(N, Pong_Pid) ->
    link(Pong_Pid),
    ping1(N, Pong_Pid).

ping1(0, _) ->
    exit(ping);

ping1(N, Pong_Pid) ->
    Pong_Pid ! {ping, self()},
    receive
        pong ->
            io:format("Ping received pong~n", [])
    end,
    ping1(N - 1, Pong_Pid).

pong() ->
    receive
        {ping, Ping_PID} ->
            io:format("Pong received ping~n", []),
            Ping_PID ! pong,
            pong()
    end.

start(Ping_Node) ->
    PongPID = spawn(tut20, pong, []),
    spawn(Ping_Node, tut20, ping, [3, PongPID]).
```

```
(s1@bill)3> tut20:start(s2@kosken).
Pong received ping
<3820.41.0>
Ping received pong
Pong received ping
Ping received pong
Pong received ping
Ping received pong
```

This is a slight modification of the ping pong program where both processes are spawned from the same `start/1` function, where the "ping" process can be spawned on a separate node. Note the use of the `link` BIF. "Ping" calls `exit(ping)` when it finishes and this will cause an exit signal to be sent to "pong" which will also terminate.

It is possible to modify the default behaviour of a process so that it does not get killed when it receives abnormal exit signals, but all signals will be turned into normal messages on the format `{'EXIT',FromPID,Reason}` and added to the end of the receiving processes message queue. This behaviour is set by:

```
process_flag(trap_exit, true)
```

There are several other process flags, see *erlang(3)*. Changing the default behaviour of a process in this way is usually not done in standard user programs, but is left to the supervisory programs in OTP (but that's another tutorial). However we will modify the ping pong program to illustrate exit trapping.

```
-module(tut21).

-export([start/1,  ping/2, pong/0]).

ping(N, Pong_Pid) ->
    link(Pong_Pid),
    ping1(N, Pong_Pid).

ping1(0, _) ->
    exit(ping);

ping1(N, Pong_Pid) ->
    Pong_Pid ! {ping, self()},
    receive
        pong ->
            io:format("Ping received pong~n", [])
    end,
    ping1(N - 1, Pong_Pid).

pong() ->
    process_flag(trap_exit, true),
    pong1().

pong1() ->
    receive
        {ping, Ping_PID} ->
            io:format("Pong received ping~n", []),
            Ping_PID ! pong,
            pong1();
        {'EXIT', From, Reason} ->
            io:format("pong exiting, got ~p~n", [{'EXIT', From, Reason}])
    end.

start(Ping_Node) ->
    PongPID = spawn(tut21, pong, []),
    spawn(Ping_Node, tut21, ping, [3, PongPID]).
```

```
(s1@bill)1> tut21:start(s2@gollum).
<3820.39.0>
Pong received ping
Ping received pong
Pong received ping
Ping received pong
Pong received ping
Ping received pong
pong exiting, got {'EXIT',<3820.39.0>,ping}
```

4.4.3 The Larger Example with Robustness Added

Now we return to the messenger program and add changes which make it more robust:

```
%%% Message passing utility.
%%% User interface:
%%% login(Name)
%%%     One user at a time can log in from each Erlang node in the
%%%     system messenger: and choose a suitable Name. If the Name
%%%     is already logged in at another node or if someone else is
%%%     already logged in at the same node, login will be rejected
%%%     with a suitable error message.
```

```
%%% logoff()
%%%     Logs off anybody at at node
%%% message(ToName, Message)
%%%     sends Message to ToName. Error messages if the user of this
%%%     function is not logged on or if ToName is not logged on at
%%%     any node.
%%%
%%% One node in the network of Erlang nodes runs a server which maintains
%%% data about the logged on users. The server is registered as "messenger"
%%% Each node where there is a user logged on runs a client process registered
%%% as "mess_client"
%%%
%%% Protocol between the client processes and the server
%%% ----------------------------------------------------
%%%
%%% To server: {ClientPid, logon, UserName}
%%% Reply {messenger, stop, user_exists_at_other_node} stops the client
%%% Reply {messenger, logged_on} logon was successful
%%%
%%% When the client terminates for some reason
%%% To server: {'EXIT', ClientPid, Reason}
%%%
%%% To server: {ClientPid, message_to, ToName, Message} send a message
%%% Reply: {messenger, stop, you_are_not_logged_on} stops the client
%%% Reply: {messenger, receiver_not_found} no user with this name logged on
%%% Reply: {messenger, sent} Message has been sent (but no guarantee)
%%%
%%% To client: {message_from, Name, Message},
%%%
%%% Protocol between the "commands" and the client
%%% ----------------------------------------------
%%%
%%% Started: messenger:client(Server_Node, Name)
%%% To client: logoff
%%% To client: {message_to, ToName, Message}
%%%
%%% Configuration: change the server_node() function to return the
%%% name of the node where the messenger server runs

-module(messenger).
-export([start_server/0, server/0,
         logon/1, logoff/0, message/2, client/2]).

%%% Change the function below to return the name of the node where the
%%% messenger server runs
server_node() ->
    messenger@super.

%%% This is the server process for the "messenger"
%%% the user list has the format [{ClientPid1, Name1},{ClientPid22, Name2},...]
server() ->
    process_flag(trap_exit, true),
    server([]).

server(User_List) ->
    receive
        {From, logon, Name} ->
            New_User_List = server_logon(From, Name, User_List),
            server(New_User_List);
        {'EXIT', From, _} ->
            New_User_List = server_logoff(From, User_List),
            server(New_User_List);
        {From, message_to, To, Message} ->
            server_transfer(From, To, Message, User_List),
            io:format("list is now: ~p~n", [User_List]),
```

```
            server(User_List)
    end.

%%% Start the server
start_server() ->
    register(messenger, spawn(messenger, server, [])).

%%% Server adds a new user to the user list
server_logon(From, Name, User_List) ->
    %% check if logged on anywhere else
    case lists:keymember(Name, 2, User_List) of
        true ->
            From ! {messenger, stop, user_exists_at_other_node},  %reject logon
            User_List;
        false ->
            From ! {messenger, logged_on},
            link(From),
            [{From, Name} | User_List]        %add user to the list
    end.

%%% Server deletes a user from the user list
server_logoff(From, User_List) ->
    lists:keydelete(From, 1, User_List).


%%% Server transfers a message between user
server_transfer(From, To, Message, User_List) ->
    %% check that the user is logged on and who he is
    case lists:keysearch(From, 1, User_List) of
        false ->
            From ! {messenger, stop, you_are_not_logged_on};
        {value, {_, Name}} ->
            server_transfer(From, Name, To, Message, User_List)
    end.

%%% If the user exists, send the message
server_transfer(From, Name, To, Message, User_List) ->
    %% Find the receiver and send the message
    case lists:keysearch(To, 2, User_List) of
        false ->
            From ! {messenger, receiver_not_found};
        {value, {ToPid, To}} ->
            ToPid ! {message_from, Name, Message},
            From ! {messenger, sent}
    end.

%%% User Commands
logon(Name) ->
    case whereis(mess_client) of
        undefined ->
            register(mess_client,
                     spawn(messenger, client, [server_node(), Name]));
        _ -> already_logged_on
    end.

logoff() ->
    mess_client ! logoff.

message(ToName, Message) ->
    case whereis(mess_client) of % Test if the client is running
        undefined ->
            not_logged_on;
        _ -> mess_client ! {message_to, ToName, Message},
             ok
end.
```

```
%%% The client process which runs on each user node
client(Server_Node, Name) ->
    {messenger, Server_Node} ! {self(), logon, Name},
    await_result(),
    client(Server_Node).

client(Server_Node) ->
    receive
        logoff ->
            exit(normal);
        {message_to, ToName, Message} ->
            {messenger, Server_Node} ! {self(), message_to, ToName, Message},
            await_result();
        {message_from, FromName, Message} ->
            io:format("Message from ~p: ~p~n", [FromName, Message])
    end,
    client(Server_Node).

%%% wait for a response from the server
await_result() ->
    receive
        {messenger, stop, Why} -> % Stop the client
            io:format("~p~n", [Why]),
            exit(normal);
        {messenger, What} ->  % Normal response
            io:format("~p~n", [What])
    after 5000 ->
            io:format("No response from server~n", []),
            exit(timeout)
    end.
```

We have added the following changes:

The messenger server traps exits. If it receives an exit signal, `{'EXIT',From,Reason}` this means that a client process has terminated or is unreachable because:

- the user has logged off (we have removed the "logoff" message),
- the network connection to the client is broken,
- the node on which the client process resides has gone down, or
- the client processes has done some illegal operation.

If we receive an exit signal as above, we delete the tuple, `{From,Name}` from the servers `User_List` using the `server_logoff` function. If the node on which the server runs goes down, an exit signal (automatically generated by the system), will be sent to all of the client processes: `{'EXIT',MessengerPID,noconnection}` causing all the client processes to terminate.

We have also introduced a timeout of five seconds in the `await_result` function. I.e. if the server does not reply within five seconds (5000 ms), the client terminates. This is really only needed in the logon sequence before the client and server are linked.

An interesting case is if the client was to terminate before the server links to it. This is taken care of because linking to a non-existent process causes an exit signal, `{'EXIT',From,noproc}`, to be automatically generated as if the process terminated immediately after the link operation.

4.5 Records and Macros

Larger programs are usually written as a collection of files with a well defined interface between the various parts.

4.5.1 The Larger Example Divided into Several Files

To illustrate this, we will divide the messenger example from the previous chapter into five files.

`mess_config.hrl`
: header file for configuration data

`mess_interface.hrl`
: interface definitions between the client and the messenger

`user_interface.erl`
: functions for the user interface

`mess_client.erl`
: functions for the client side of the messenger

`mess_server.erl`
: functions for the server side of the messenger

While doing this we will also clean up the message passing interface between the shell, the client and the server and define it using *records*, we will also introduce *macros*.

```
%%%----FILE mess_config.hrl----

%%% Configure the location of the server node,
-define(server_node, messenger@super).

%%%----END FILE----
```

```
%%%----FILE mess_interface.hrl----

%%% Message interface between client and server and client shell for
%%% messenger program

%%%Messages from Client to server received in server/1 function.
-record(logon,{client_pid, username}).
-record(message,{client_pid, to_name, message}).
%%% {'EXIT', ClientPid, Reason}  (client terminated or unreachable.

%%% Messages from Server to Client, received in await_result/0 function
-record(abort_client,{message}).
%%% Messages are: user_exists_at_other_node,
%%%               you_are_not_logged_on
-record(server_reply,{message}).
%%% Messages are: logged_on
%%%               receiver_not_found
%%%               sent  (Message has been sent (no guarantee)
%%% Messages from Server to Client received in client/1 function
-record(message_from,{from_name, message}).

%%% Messages from shell to Client received in client/1 function
%%% spawn(mess_client, client, [server_node(), Name])
-record(message_to,{to_name, message}).
%%% logoff

%%%----END FILE----
```

```
%%%----FILE user_interface.erl----

%%% User interface to the messenger program
```

```
%%% login(Name)
%%%     One user at a time can log in from each Erlang node in the
%%%     system messenger: and choose a suitable Name. If the Name
%%%     is already logged in at another node or if someone else is
%%%     already logged in at the same node, login will be rejected
%%%     with a suitable error message.

%%% logoff()
%%%     Logs off anybody at at node

%%% message(ToName, Message)
%%%     sends Message to ToName. Error messages if the user of this
%%%     function is not logged on or if ToName is not logged on at
%%%     any node.

-module(user_interface).
-export([logon/1, logoff/0, message/2]).
-include("mess_interface.hrl").
-include("mess_config.hrl").

logon(Name) ->
    case whereis(mess_client) of
        undefined ->
            register(mess_client,
                     spawn(mess_client, client, [?server_node, Name]));
        _ -> already_logged_on
    end.

logoff() ->
    mess_client ! logoff.

message(ToName, Message) ->
    case whereis(mess_client) of % Test if the client is running
        undefined ->
            not_logged_on;
        _ -> mess_client ! #message_to{to_name=ToName, message=Message},
             ok
end.

%%%----END FILE----
```

```
%%%----FILE mess_client.erl----

%%% The client process which runs on each user node

-module(mess_client).
-export([client/2]).
-include("mess_interface.hrl").

client(Server_Node, Name) ->
    {messenger, Server_Node} ! #logon{client_pid=self(), username=Name},
    await_result(),
    client(Server_Node).

client(Server_Node) ->
    receive
        logoff ->
            exit(normal);
        #message_to{to_name=ToName, message=Message} ->
            {messenger, Server_Node} !
                #message{client_pid=self(), to_name=ToName, message=Message},
            await_result();
        {message_from, FromName, Message} ->
```

```
            io:format("Message from ~p: ~p~n", [FromName, Message])
    end,
    client(Server_Node).

%%% wait for a response from the server
await_result() ->
    receive
        #abort_client{message=Why} ->
            io:format("~p~n", [Why]),
            exit(normal);
        #server_reply{message=What} ->
            io:format("~p~n", [What])
    after 5000 ->
            io:format("No response from server~n", []),
            exit(timeout)
    end.

%%%----END FILE---
```

```
%%%----FILE mess_server.erl----

%%% This is the server process of the messenger service

-module(mess_server).
-export([start_server/0, server/0]).
-include("mess_interface.hrl").

server() ->
    process_flag(trap_exit, true),
    server([]).

%%% the user list has the format [{ClientPid1, Name1},{ClientPid22, Name2},...]
server(User_List) ->
    io:format("User list = ~p~n", [User_List]),
    receive
        #logon{client_pid=From, username=Name} ->
            New_User_List = server_logon(From, Name, User_List),
            server(New_User_List);
        {'EXIT', From, _} ->
            New_User_List = server_logoff(From, User_List),
            server(New_User_List);
        #message{client_pid=From, to_name=To, message=Message} ->
            server_transfer(From, To, Message, User_List),
            server(User_List)
    end.

%%% Start the server
start_server() ->
    register(messenger, spawn(?MODULE, server, [])).

%%% Server adds a new user to the user list
server_logon(From, Name, User_List) ->
    %% check if logged on anywhere else
    case lists:keymember(Name, 2, User_List) of
        true ->
            From ! #abort_client{message=user_exists_at_other_node},
            User_List;
        false ->
            From ! #server_reply{message=logged_on},
            link(From),
            [{From, Name} | User_List]        %add user to the list
    end.
```

```
%%% Server deletes a user from the user list
server_logoff(From, User_List) ->
    lists:keydelete(From, 1, User_List).

%%% Server transfers a message between user
server_transfer(From, To, Message, User_List) ->
    %% check that the user is logged on and who he is
    case lists:keysearch(From, 1, User_List) of
        false ->
            From ! #abort_client{message=you_are_not_logged_on};
        {value, {_, Name}} ->
            server_transfer(From, Name, To, Message, User_List)
    end.
%%% If the user exists, send the message
server_transfer(From, Name, To, Message, User_List) ->
    %% Find the receiver and send the message
    case lists:keysearch(To, 2, User_List) of
        false ->
            From ! #server_reply{message=receiver_not_found};
        {value, {ToPid, To}} ->
            ToPid ! #message_from{from_name=Name, message=Message},
            From !  #server_reply{message=sent}
    end.

%%%----END FILE---
```

4.5.2 Header Files

You will see some files above with extension `.hrl`. These are header files which are included in the `.erl` files by:

```
-include("File_Name").
```

for example:

```
-include("mess_interface.hrl").
```

In our case above the file is fetched from the same directory as all the other files in the messenger example. (*manual*).

.hrl files can contain any valid Erlang code but are most often used for record and macro definitions.

4.5.3 Records

A record is defined as:

```
-record(name_of_record,{field_name1, field_name2, field_name3, ......}).
```

For example:

```
-record(message_to,{to_name, message}).
```

This is exactly equivalent to:

```
{message_to, To_Name, Message}
```

Creating record, is best illustrated by an example:

```
#message_to{message="hello", to_name=fred)
```

This will create:

```
{message_to, fred, "hello"}
```

Note that you don't have to worry about the order you assign values to the various parts of the records when you create it. The advantage of using records is that by placing their definitions in header files you can conveniently define interfaces which are easy to change. For example, if you want to add a new field to the record, you will only have to change the code where the new field is used and not at every place the record is referred to. If you leave out a field when creating a record, it will get the value of the atom undefined. (*manual*)

Pattern matching with records is very similar to creating records. For example inside a `case` or `receive`:

```
#message_to{to_name=ToName, message=Message} ->
```

is the same as:

```
{message_to, ToName, Message}
```

4.5.4 Macros

The other thing we have added to the messenger is a macro. The file `mess_config.hrl` contains the definition:

```
%%% Configure the location of the server node,
-define(server_node, messenger@super).
```

We include this file in mess_server.erl:

```
-include("mess_config.hrl").
```

Every occurrence of `?server_node` in `mess_server.erl` will now be replaced by `messenger@super`.

The other place a macro is used is when we spawn the server process:

```
spawn(?MODULE, server, [])
```

This is a standard macro (i.e. defined by the system, not the user). `?MODULE` is always replaced by the name of current module (i.e. the `-module` definition near the start of the file). There are more advanced ways of using macros with, for example parameters (*manual*).

The three Erlang (`.erl`) files in the messenger example are individually compiled into object code file (`.beam`). The Erlang system loads and links these files into the system when they are referred to during execution of the code. In our case we simply have put them in the same directory which is our current working directory (i.e. the place we have done "cd" to). There are ways of putting the `.beam` files in other directories.

In the messenger example, no assumptions have been made about what the message being sent is. It could be any valid Erlang term.

5 User's Guide

5.1 Introduction

5.1.1 Purpose

This reference manual describes the Erlang programming language. The focus is on the language itself, not the implementation. The language constructs are described in text and with examples rather than formally specified, with the intention to make the manual more readable. The manual is not intended as a tutorial.

Information about this implementation of Erlang can be found, for example, in *System Principles* (starting and stopping, boot scripts, code loading, error logging, creating target systems), *Efficiency Guide* (memory consumption, system limits) and *ERTS User's Guide* (crash dumps, drivers).

5.1.2 Prerequisites

It is assumed that the reader has done some programming and is familiar with concepts such as data types and programming language syntax.

5.1.3 Document Conventions

In the document, the following terminology is used:

- A *sequence* is one or more items. For example, a clause body consists of a sequence of expressions. This means that there must be at least one expression.
- A *list* is any number of items. For example, an argument list can consist of zero, one or more arguments.

If a feature has been added recently, in Erlang 5.0/OTP R7 or later, this is mentioned in the text.

5.1.4 Complete List of BIFs

For a complete list of BIFs, their arguments and return values, refer to `erlang(3)`.

5.1.5 Reserved Words

The following are reserved words in Erlang:

after and andalso band begin bnot bor bsl bsr bxor case catch cond div end fun if let not of or orelse query receive rem try when xor

5.1.6 Character Set

In Erlang 4.8/OTP R5A the syntax of Erlang tokens was extended to allow the use of the full ISO-8859-1 (Latin-1) character set. This is noticeable in the following ways:

- All the Latin-1 printable characters can be used and are shown without the escape backslash convention.
- Atoms and variables can use all Latin-1 letters.

Octal	*Decimal*		*Class*
200 - 237	128 - 159		Control characters

240 - 277	160 - 191	- ¿	Punctuation characters
300 - 326	192 - 214	À - Ö	Uppercase letters
327	215	×	Punctuation character
330 - 336	216 - 222	Ø - Þ	Uppercase letters
337 - 366	223 - 246	ß - ö	Lowercase letters
367	247	÷	Punctuation character
370 - 377	248 - 255	ø - ÿ	Lowercase letters

Table 1.1: Character Classes.

5.2 Data Types

5.2.1 Terms

Erlang provides a number of data types which are listed in this chapter. A piece of data of any data type is called a *term*.

5.2.2 Number

There are two types of numeric literals, *integers* and *floats*. Besides the conventional notation, there are two Erlang-specific notations:

- `$char`
  ASCII value of the character `char`.
- `base#value`
  Integer with the base `base`, which must be an integer in the range 2..36.
  In Erlang 5.2/OTP R9B and earlier versions, the allowed range is 2..16.

Examples:

```
1> 42.
42
2> $A.
65
3> $\n.
10
4> 2#101.
5
5> 16#1f.
31
6> 2.3.
2.3
7> 2.3e3.
2.3e3
8> 2.3e-3.
0.0023
```

5.2.3 Atom

An atom is a literal, a constant with name. An atom should be enclosed in single quotes (') if it does not begin with a lower-case letter or if it contains other characters than alphanumeric characters, underscore (_), or @.

Examples:

```
hello
phone_number
'Monday'
'phone number'
```

5.2.4 Bit Strings and Binaries

A bit string is used to store an area of untyped memory.

Bit Strings are expressed using the *bit syntax*.

Bit Strings which consists of a number of bits which is evenly divisible by eight are called Binaries

Examples:

```
1> <<10,20>>.
<<10,20>>
2> <<"ABC">>.
<<"ABC">>
1> <<1:1,0:1>>.
<<2:2>>
```

More examples can be found in Programming Examples.

5.2.5 Reference

A reference is a term which is unique in an Erlang runtime system, created by calling `make_ref/0`.

5.2.6 Fun

A fun is a functional object. Funs make it possible to create an anonymous function and pass the function itself -- not its name -- as argument to other functions.

Example:

```
1> Fun1 = fun (X) -> X+1 end.
#Fun<erl_eval.6.39074546>
2> Fun1(2).
3
```

Read more about funs in *Fun Expressions*. More examples can be found in Programming Examples.

5.2.7 Port Identifier

A port identifier identifies an Erlang port. `open_port/2`, which is used to create ports, will return a value of this type.

Read more about ports in *Ports and Port Drivers*.

5.2.8 Pid

A process identifier, pid, identifies a process. `spawn/1,2,3,4`, `spawn_link/1,2,3,4` and `spawn_opt/4`, which are used to create processes, return values of this type. Example:

```
1> spawn(m, f, []).
<0.51.0>
```

The BIF `self()` returns the pid of the calling process. Example:

```
-module(m).
-export([loop/0]).

loop() ->
    receive
        who_are_you ->
            io:format("I am ~p~n", [self()]),
            loop()
    end.

1> P = spawn(m, loop, []).
<0.58.0>
2> P ! who_are_you.
I am <0.58.0>
who_are_you
```

Read more about processes in *Processes*.

5.2.9 Tuple

Compound data type with a fixed number of terms:

```
{Term1,...,TermN}
```

Each term `Term` in the tuple is called an *element*. The number of elements is said to be the *size* of the tuple.

There exists a number of BIFs to manipulate tuples.

Examples:

```
1> P = {adam,24,{july,29}}.
{adam,24,{july,29}}
2> element(1,P).
adam
3> element(3,P).
{july,29}
4> P2 = setelement(2,P,25).
{adam,25,{july,29}}
5> tuple_size(P).
3
6> tuple_size({}).
0
```

5.2.10 List

Compound data type with a variable number of terms.

```
[Term1,...,TermN]
```

Each term `Term` in the list is called an *element*. The number of elements is said to be the *length* of the list.

Formally, a list is either the empty list `[]` or consists of a *head* (first element) and a *tail* (remainder of the list) which is also a list. The latter can be expressed as `[H|T]`. The notation `[Term1,...,TermN]` above is actually shorthand for the list `[Term1|[...|[TermN|[]]]]`.

Example:
`[]` is a list, thus
`[c|[]]` is a list, thus
`[b|[c|[]]]` is a list, thus
`[a|[b|[c|[]]]]` is a list, or in short `[a,b,c]`.

A list where the tail is a list is sometimes called a *proper list*. It is allowed to have a list where the tail is not a list, for example `[a|b]`. However, this type of list is of little practical use.

Examples:

```
1> L1 = [a,2,{c,4}].
[a,2,{c,4}]
2> [H|T] = L1.
[a,2,{c,4}]
3> H.
a
4> T.
[2,{c,4}]
5> L2 = [d|T].
[d,2,{c,4}]
6> length(L1).
3
7> length([]).
0
```

A collection of list processing functions can be found in the STDLIB module `lists`.

5.2.11 String

Strings are enclosed in double quotes ("), but is not a data type in Erlang. Instead a string `"hello"` is shorthand for the list `[$h,$e,$l,$l,$o]`, that is `[104,101,108,108,111]`.

Two adjacent string literals are concatenated into one. This is done at compile-time and does not incur any runtime overhead. Example:

```
"string" "42"
```

is equivalent to

```
```

```
"string42"
```

5.2.12 Record

A record is a data structure for storing a fixed number of elements. It has named fields and is similar to a struct in C. However, record is not a true data type. Instead record expressions are translated to tuple expressions during compilation. Therefore, record expressions are not understood by the shell unless special actions are taken. See `shell(3)` for details.

Examples:

```
-module(person).
-export([new/2]).

-record(person, {name, age}).

new(Name, Age) ->
    #person{name=Name, age=Age}.

1> person:new(ernie, 44).
{person,ernie,44}
```

Read more about records in *Records*. More examples can be found in Programming Examples.

5.2.13 Boolean

There is no Boolean data type in Erlang. Instead the atoms `true` and `false` are used to denote Boolean values.

Examples:

```
1> 2 =< 3.
true
2> true or false.
true
```

5.2.14 Escape Sequences

Within strings and quoted atoms, the following escape sequences are recognized:

Sequence	*Description*
\b	backspace
\d	delete
\e	escape
\f	form feed
\n	newline
\r	carriage return

\s	space
\t	tab
\v	vertical tab
\XYZ, \YZ, \Z	character with octal representation XYZ, YZ or Z
\xXY	character with hexadecimal representation XY
\x{X...}	character with hexadecimal representation; X... is one or more hexadecimal characters
\^a...\^z \^A...\^Z	control A to control Z
\'	single quote
\"	double quote
\\	backslash

Table 2.1: Recognized Escape Sequences.

5.2.15 Type Conversions

There are a number of BIFs for type conversions. Examples:

```
1> atom_to_list(hello).
"hello"
2> list_to_atom("hello").
hello
3> binary_to_list(<<"hello">>).
"hello"
4> binary_to_list(<<104,101,108,108,111>>).
"hello"
5> list_to_binary("hello").
<<104,101,108,108,111>>
6> float_to_list(7.0).
"7.00000000000000000000e+00"
7> list_to_float("7.000e+00").
7.0
8> integer_to_list(77).
"77"
9> list_to_integer("77").
77
10> tuple_to_list({a,b,c}).
[a,b,c]
11> list_to_tuple([a,b,c]).
{a,b,c}
12> term_to_binary({a,b,c}).
<<131,104,3,100,0,1,97,100,0,1,98,100,0,1,99>>
13> binary_to_term(<<131,104,3,100,0,1,97,100,0,1,98,100,0,1,99>>).
{a,b,c}
```

5.3 Pattern Matching

5.3.1 Pattern Matching

Variables are bound to values through the *pattern matching* mechanism. Pattern matching occurs when evaluating a function call, `case`- `receive`- `try`- expressions and match operator (=) expressions.

In a pattern matching, a left-hand side *pattern* is matched against a right-hand side *term*. If the matching succeeds, any unbound variables in the pattern become bound. If the matching fails, a run-time error occurs.

Examples:

```
1> X.
** 1: variable 'X' is unbound **
2> X = 2.
2
3> X + 1.
3
4> {X, Y} = {1, 2}.
** exception error: no match of right hand side value {1,2}
5> {X, Y} = {2, 3}.
{2,3}
6> Y.
3
```

5.4 Modules

5.4.1 Module Syntax

Erlang code is divided into *modules*. A module consists of a sequence of attributes and function declarations, each terminated by period (.). Example:

```
-module(m).          % module attribute
-export([fact/1]).   % module attribute

fact(N) when N>0 ->  % beginning of function declaration
    N * fact(N-1);   %  |
fact(0) ->           %  |
    1.               % end of function declaration
```

See the *Functions* chapter for a description of function declarations.

5.4.2 Module Attributes

A *module attribute* defines a certain property of a module. A module attribute consists of a tag and a value.

```
-Tag(Value).
```

`Tag` must be an atom, while `Value` must be a literal term. As a convenience in user-defined attributes, the literal term `Value` the syntax `Name/Arity` (where `Name` is an atom and `Arity` a positive integer) will be translated to `{Name,Arity}`.

Any module attribute can be specified. The attributes are stored in the compiled code and can be retrieved by calling `Module:module_info(attributes)` or by using *beam_lib(3)*.

There are several module attributes with predefined meanings, some of which have arity two, but user-defined module attributes must have arity one.

Pre-Defined Module Attributes

Pre-defined module attributes should be placed before any function declaration.

`-module(Module).`

Module declaration, defining the name of the module. The name `Module`, an atom, should be the same as the file name minus the extension `erl`. Otherwise *code loading* will not work as intended.

This attribute should be specified first and is the only attribute which is mandatory.

`-export(Functions).`

Exported functions. Specifies which of the functions defined within the module that are visible outside the module.

`Functions` is a list `[Name1/Arity1, ..., NameN/ArityN]`, where each `NameI` is an atom and `ArityI` an integer.

`-import(Module,Functions).`

Imported functions. Imported functions can be called the same way as local functions, that is without any module prefix.

`Module`, an atom, specifies which module to import functions from. `Functions` is a list similar as for `export` above.

`-compile(Options).`

Compiler options. `Options`, which is a single option or a list of options, will be added to the option list when compiling the module. See `compile(3)`.

`-vsn(Vsn).`

Module version. `Vsn` is any literal term and can be retrieved using `beam_lib:version/1`, see *beam_lib(3)*.

If this attribute is not specified, the version defaults to the MD5 checksum of the module.

Behaviour Module Attribute

It is possible to specify that the module is the callback module for a *behaviour*:

```
-behaviour(Behaviour).
```

The atom `Behaviour` gives the name of the behaviour, which can be a user defined behaviour or one of the OTP standard behaviours `gen_server`, `gen_fsm`, `gen_event` or `supervisor`.

The spelling `behavior` is also accepted.

Read more about behaviours and callback modules in OTP Design Principles.

Record Definitions

The same syntax as for module attributes is used by for record definitions:

```
-record(Record,Fields).
```

Record definitions are allowed anywhere in a module, also among the function declarations. Read more in *Records*.

The Preprocessor

The same syntax as for module attributes is used by the preprocessor, which supports file inclusion, macros, and conditional compilation:

```
-include("SomeFile.hrl").
-define(Macro,Replacement).
```

Read more in *The Preprocessor*.

Setting File and Line

The same syntax as for module attributes is used for changing the pre-defined macros `?FILE` and `?LINE`:

```
-file(File, Line).
```

This attribute is used by tools such as Yecc to inform the compiler that the source program was generated by another tool and indicates the correspondence of source files to lines of the original user-written file from which the source program was produced.

Types and function specifications

A similar syntax as for module attributes is used for specifying types and function specifications.

```
-type my_type() :: atom() | integer().
-spec my_function(integer()) -> integer().
```

Read more in *Types and Function specifications*.

The desciption is based on **EEP8 - Types and function specifications** which will not be further updated.

5.4.3 Comments

Comments may be placed anywhere in a module except within strings and quoted atoms. The comment begins with the character "%", continues up to, but does not include the next end-of-line, and has no effect. Note that the terminating end-of-line has the effect of white space.

5.4.4 The module_info/0 and module_info/1 functions

The compiler automatically inserts the two special, exported functions into each module: `Module:module_info/0` and `Module:module_info/1`. These functions can be called to retrieve information about the module.

module_info/0

The `module_info/0` function in each module returns a list of `{Key,Value}` tuples with information about the module. Currently, the list contain tuples with the following `Key`s: `attributes`, `compile`, `exports`, and `imports`. The order and number of tuples may change without prior notice.

> Warning:
>
> The `{imports,Value}` tuple may be removed in a future release because `Value` is always an empty list. Do not write code that depends on it being present.

module_info/1

The call `module_info(Key)`, where key is an atom, returns a single piece of information about the module.

The following values are allowed for `Key`:

`attributes`

Return a list of `{AttributeName,ValueList}` tuples, where `AttributeName` is the name of an attribute, and `ValueList` is a list of values. Note: a given attribute may occur more than once in the list with different values if the attribute occurs more than once in the module.

The list of attributes will be empty if the module has been stripped with *beam_lib(3)*.

`compile`

Return a list of tuples containing information about how the module was compiled. This list will be empty if the module has been stripped with *beam_lib(3)*.

`imports`

Always return an empty list. The `imports` key may not be supported in future release.

`exports`

Return a list of `{Name,Arity}` tuples with all exported functions in the module.

`functions`

Return a list of `{Name,Arity}` tuples with all functions in the module.

5.5 Functions

5.5.1 Function Declaration Syntax

A *function declaration* is a sequence of function clauses separated by semicolons, and terminated by period (.).

A *function clause* consists of a clause head and a clause body, separated by `->`.

A clause *head* consists of the function name, an argument list, and an optional guard sequence beginning with the keyword `when`.

```
Name(Pattern11,...,Pattern1N) [when GuardSeq1] ->
    Body1;
...;
Name(PatternK1,...,PatternKN) [when GuardSeqK] ->
    BodyK.
```

The function name is an atom. Each argument is a pattern.

The number of arguments `N` is the *arity* of the function. A function is uniquely defined by the module name, function name and arity. That is, two functions with the same name and in the same module, but with different arities are two completely different functions.

A function named `f` in the module `m` and with arity `N` is often denoted as `m:f/N`.

A clause *body* consists of a sequence of expressions separated by comma (,):

```
Expr1,
...,
ExprN
```

Valid Erlang expressions and guard sequences are described in *Erlang Expressions*.

Example:

```
fact(N) when N>0 ->  % first clause head
    N * fact(N-1);   % first clause body

fact(0) ->           % second clause head
    1.               % second clause body
```

5.5.2 Function Evaluation

When a function `m:f/N` is called, first the code for the function is located. If the function cannot be found, an `undef` run-time error will occur. Note that the function must be exported to be visible outside the module it is defined in.

If the function is found, the function clauses are scanned sequentially until a clause is found that fulfills the following two conditions:

- the patterns in the clause head can be successfully matched against the given arguments, and
- the guard sequence, if any, is true.

If such a clause cannot be found, a `function_clause` run-time error will occur.

If such a clause is found, the corresponding clause body is evaluated. That is, the expressions in the body are evaluated sequentially and the value of the last expression is returned.

Example: Consider the function `fact`:

```
-module(m).
-export([fact/1]).

fact(N) when N>0 ->
    N * fact(N-1);
fact(0) ->
    1.
```

Assume we want to calculate factorial for 1:

```
1> m:fact(1).
```

Evaluation starts at the first clause. The pattern `N` is matched against the argument 1. The matching succeeds and the guard (`N>0`) is true, thus `N` is bound to 1 and the corresponding body is evaluated:

```
N * fact(N-1) => (N is bound to 1)
1 * fact(0)
```

Now `fact(0)` is called and the function clauses are scanned sequentially again. First, the pattern `N` is matched against 0. The matching succeeds, but the guard (`N>0`) is false. Second, the pattern 0 is matched against 0. The matching succeeds and the body is evaluated:

```
1 * fact(0) =>
1 * 1 =>
1
```

Evaluation has succeed and `m:fact(1)` returns 1.

If `m:fact/1` is called with a negative number as argument, no clause head will match. A `function_clause` run-time error will occur.

5.5.3 Tail recursion

If the last expression of a function body is a function call, a *tail recursive* call is done so that no system resources for example call stack are consumed. This means that an infinite loop can be done if it uses tail recursive calls.

Example:

```
loop(N) ->
    io:format("~w~n", [N]),
    loop(N+1).
```

As a counter-example see the factorial example above that is not tail recursive since a multiplication is done on the result of the recursive call to `fact(N-1)`.

5.5.4 Built-In Functions, BIFs

Built-in functions, BIFs, are implemented in C code in the runtime system and do things that are difficult or impossible to implement in Erlang. Most of the built-in functions belong to the module `erlang` but there are also built-in functions belonging to a few other modules, for example `lists` and `ets`.

The most commonly used BIFs belonging to `erlang` are *auto-imported*, they do not need to be prefixed with the module name. Which BIFs are auto-imported is specified in `erlang(3)`. For example, standard type conversion BIFs like `atom_to_list` and BIFs allowed in guards can be called without specifying the module name. Examples:

```
1> tuple_size({a,b,c}).
3
2> atom_to_list('Erlang').
"Erlang"
```

Note that normally it is the set of auto-imported built-in functions that is referred to when talking about 'BIFs'.

5.6 Types and Function Specifications

5.6.1 Introduction of Types

Although Erlang is a dynamically typed language this section describes an extension to the Erlang language for declaring sets of Erlang terms to form a particular type, effectively forming a specific sub-type of the set of all Erlang terms.

Subsequently, these types can be used to specify types of record fields and the argument and return types of functions.

Type information can be used to document function interfaces, provide more information for bug detection tools such as `Dialyzer`, and can be exploited by documentation tools such as `Edoc` for generating program documentation of various forms. It is expected that the type language described in this document will supersede and replace the purely comment-based `@type` and `@spec` declarations used by `Edoc`.

> Warning:
> The syntax and semantics described here is still preliminary and might be slightly changed and extended before it becomes officially supported. The plan is that this will happen in R14B.

5.6.2 Types and their Syntax

Types describe sets of Erlang terms. Types consist and are built from a set of predefined types (e.g. `integer()`, `atom()`, `pid()`, ...) described below. Predefined types represent a typically infinite set of Erlang terms which belong to this type. For example, the type `atom()` stands for the set of all Erlang atoms.

For integers and atoms, we allow for singleton types (e.g. the integers `-1` and `42` or the atoms `'foo'` and `'bar'`). All other types are built using unions of either predefined types or singleton types. In a type union between a type and one of its sub-types the sub-type is absorbed by the super-type and the union is subsequently treated as if the sub-type was not a constituent of the union. For example, the type union:

```
atom() | 'bar' | integer() | 42
```

describes the same set of terms as the type union:

```
atom() | integer()
```

Because of sub-type relations that exist between types, types form a lattice where the topmost element, any(), denotes the set of all Erlang terms and the bottom-most element, none(), denotes the empty set of terms.

The set of predefined types and the syntax for types is given below:

```
Type :: any()            %% The top type, the set of all Erlang terms.
      | none()           %% The bottom type, contains no terms.
      | pid()
      | port()
      | ref()
      | []               %% nil
      | Atom
      | Binary
      | float()
      | Fun
      | Integer
      | List
      | Tuple
      | Union
      | UserDefined      %% described in Section 2

Union :: Type1 | Type2

Atom :: atom()
```

```
      | Erlang_Atom      %% 'foo', 'bar', ...

Binary :: binary()                          %% <<_:_ * 8>>
        | <<>>
        | <<_:Erlang_Integer>>              %% Base size
        | <<_:_*Erlang_Integer>>            %% Unit size
        | <<_:Erlang_Integer, _:_*Erlang_Integer>>

Fun :: fun()                                %% any function
     | fun((...) -> Type)                   %% any arity, returning Type
     | fun(() -> Type)
     | fun((TList) -> Type)

Integer :: integer()
         | Erlang_Integer                   %% ..., -1, 0, 1, ... 42 ...
         | Erlang_Integer..Erlang_Integer   %% specifies an integer range

List :: list(Type)                          %% Proper list ([]-terminated)
      | improper_list(Type1, Type2)         %% Type1=contents, Type2=termination
      | maybe_improper_list(Type1, Type2)   %% Type1 and Type2 as above

Tuple :: tuple()                            %% stands for a tuple of any size
       | {}
       | {TList}

TList :: Type
       | Type, TList
```

Because lists are commonly used, they have shorthand type notations. The type `list(T)` has the shorthand `[T]`. The shorthand `[T,...]` stands for the set of non-empty proper lists whose elements are of type `T`. The only difference between the two shorthands is that `[T]` may be an empty list but `[T,...]` may not.

Notice that the shorthand for `list()`, i.e. the list of elements of unknown type, is `[_]` (or `[any()]`), not `[]`. The notation `[]` specifies the singleton type for the empty list.

For convenience, the following types are also built-in. They can be thought as predefined aliases for the type unions also shown in the table. (Some type unions below slightly abuse the syntax of types.)

Built-in type	Stands for
`term()`	`any()`
`bool()`	`'false' | 'true'`
`byte()`	`0..255`
`char()`	`0..16#10ffff`
`non_neg_integer()`	`0..`
`pos_integer()`	`1..`
`neg_integer()`	`..-1`
`number()`	`integer() | float()`
`list()`	`[any()]`

`maybe_improper_list()`	`maybe_improper_list(any(), any())`
`maybe_improper_list(T)`	`maybe_improper_list(T, any())`
`string()`	`[char()]`
`nonempty_string()`	`[char(),...]`
`iolist()`	`maybe_improper_list( char() | binary() | iolist(), binary() | [])`
`module()`	`atom()`
`mfa()`	`{atom(),atom(),byte()}`
`node()`	`atom()`
`timeout()`	`'infinity' | non_neg_integer()`
`no_return()`	`none()`

Users are not allowed to define types with the same names as the predefined or built-in ones. This is checked by the compiler and its violation results in a compilation error. (For bootstrapping purposes, it can also result to just a warning if this involves a built-in type which has just been introduced.)

> Note:
> The following built-in list types also exist, but they are expected to be rarely used. Hence, they have long names:

```
nonempty_maybe_improper_list(Type) :: nonempty_maybe_improper_list(Type, any())
nonempty_maybe_improper_list() :: nonempty_maybe_improper_list(any())
```

where the following two types define the set of Erlang terms one would expect:

```
nonempty_improper_list(Type1, Type2)
nonempty_maybe_improper_list(Type1, Type2)
```

Also for convenience, we allow for record notation to be used. Records are just shorthands for the corresponding tuples.

```
Record :: #Erlang_Atom{}
        | #Erlang_Atom{Fields}
```

Records have been extended to possibly contain type information. This is described in the sub-section *"Type information in record declarations"* below.

5.6.3 Type declarations of user-defined types

As seen, the basic syntax of a type is an atom followed by closed parentheses. New types are declared using '-type' compiler attributes as in the following:

```
-type my_type() :: Type.
```

where the type name is an atom (`'my_type'` in the above) followed by parenthesis. Type is a type as defined in the previous section. A current restriction is that Type can contain only predefined types or user-defined types which have been previously defined. This restriction is enforced by the compiler and results in a compilation error. (A similar restriction currently exists for records).

This means that currently general recursive types cannot be defined. Lifting this restriction is future work.

Type declarations can also be parameterized by including type variables between the parentheses. The syntax of type variables is the same as Erlang variables (starts with an upper case letter). Naturally, these variables can - and should - appear on the RHS of the definition. A concrete example appears below:

```
-type orddict(Key, Val) :: [{Key, Val}].
```

5.6.4 Type information in record declarations

The types of record fields can be specified in the declaration of the record. The syntax for this is:

```
-record(rec, {field1 :: Type1, field2, field3 :: Type3}).
```

For fields without type annotations, their type defaults to any(). I.e., the above is a shorthand for:

```
-record(rec, {field1 :: Type1, field2 :: any(), field3 :: Type3}).
```

In the presence of initial values for fields, the type must be declared after the initialization as in the following:

```
-record(rec, {field1 = [] :: Type1, field2, field3 = 42 :: Type3}).
```

Naturally, the initial values for fields should be compatible with (i.e. a member of) the corresponding types. This is checked by the compiler and results in a compilation error if a violation is detected. For fields without initial values, the singleton type `'undefined'` is added to all declared types. In other words, the following two record declarations have identical effects:

```
-record(rec, {f1 = 42 :: integer(),
              f2      :: float(),
              f3      :: 'a' | 'b').
```

```
-record(rec, {f1 = 42 :: integer(),
              f2      :: 'undefined' | float(),
              f3      :: 'undefined' | 'a' | 'b').
```

For this reason, it is recommended that records contain initializers, whenever possible.

Any record, containing type information or not, once defined, can be used as a type using the syntax:

```
#rec{}
```

In addition, the record fields can be further specified when using a record type by adding type information about the field in the following manner:

```
#rec{some_field :: Type}
```

Any unspecified fields are assumed to have the type in the original record declaration.

5.6.5 Specifications (contracts) for functions

A contract (or specification) for a function is given using the new compiler attribute `'-spec'`. The basic format is as follows:

```
-spec Module:Function(ArgType1, ..., ArgTypeN) -> ReturnType.
```

The arity of the function has to match the number of arguments, or else a compilation error occurs.

This form can also be used in header files (.hrl) to declare type information for exported functions. Then these header files can be included in files that (implicitly or explicitly) import these functions.

For most uses within a given module, the following shorthand is allowed:

```
-spec Function(ArgType1, ..., ArgTypeN) -> ReturnType.
```

Also, for documentation purposes, argument names can be given:

```
-spec Function(ArgName1 :: Type1, ..., ArgNameN :: TypeN) -> RT.
```

A function specification can be overloaded. That is, it can have several types, separated by a semicolon (`;`):

```
-spec foo(T1, T2) -> T3
       ; (T4, T5) -> T6.
```

A current restriction, which currently results in a warning (OBS: not an error) by the compiler, is that the domains of the argument types cannot be overlapping. For example, the following specification results in a warning:

```
-spec foo(pos_integer()) -> pos_integer()
      ; (integer()) -> integer().
```

Type variables can be used in specifications to specify relations for the input and output arguments of a function. For example, the following specification defines the type of a polymorphic identity function:

```
-spec id(X) -> X.
```

However, note that the above specification does not restrict the input and output type in any way. We can constrain these types by guard-like subtype constraints:

```
-spec id(X) -> X when is_subtype(X, tuple()).
```

and provide bounded quantification. Currently, the `is_subtype/2` guard is the only guard which can be used in a `'-spec'` attribute.

The scope of an `is_subtype/2` constraint is the `(...) -> RetType` specification after which it appears. To avoid confusion, we suggest that different variables are used in different constituents of an overloaded contract as in the example below:

```
-spec foo({X, integer()}) -> X when is_subtype(X, atom())
      ; ([Y]) -> Y when is_subtype(Y, number()).
```

Some functions in Erlang are not meant to return; either because they define servers or because they are used to throw exceptions as the function below:

```
my_error(Err) -> erlang:throw({error, Err}).
```

For such functions we recommend the use of the special no_return() type for their "return", via a contract of the form:

```
-spec my_error(term()) -> no_return().
```

5.7 Expressions

In this chapter, all valid Erlang expressions are listed. When writing Erlang programs, it is also allowed to use macro- and record expressions. However, these expressions are expanded during compilation and are in that sense not true Erlang expressions. Macro- and record expressions are covered in separate chapters: *Macros* and *Records*.

5.7.1 Expression Evaluation

All subexpressions are evaluated before an expression itself is evaluated, unless explicitly stated otherwise. For example, consider the expression:

```
Expr1 + Expr2
```

`Expr1` and `Expr2`, which are also expressions, are evaluated first - in any order - before the addition is performed.

Many of the operators can only be applied to arguments of a certain type. For example, arithmetic operators can only be applied to numbers. An argument of the wrong type will cause a `badarg` run-time error.

5.7.2 Terms

The simplest form of expression is a term, that is an integer, float, atom, string, list or tuple. The return value is the term itself.

5.7.3 Variables

A variable is an expression. If a variable is bound to a value, the return value is this value. Unbound variables are only allowed in patterns.

Variables start with an uppercase letter or underscore (_) and may contain alphanumeric characters, underscore and @. Examples:

```
X
Name1
PhoneNumber
Phone_number
_
_Height
```

Variables are bound to values using *pattern matching*. Erlang uses *single assignment*, a variable can only be bound once.

The *anonymous variable* is denoted by underscore (_) and can be used when a variable is required but its value can be ignored. Example:

```
[H|_] = [1,2,3]
```

Variables starting with underscore (_), for example `_Height`, are normal variables, not anonymous. They are however ignored by the compiler in the sense that they will not generate any warnings for unused variables. Example: The following code

```
member(_, []) ->
    [].
```

can be rewritten to be more readable:

```
member(Elem, []) ->
```

```
    [].
```

This will however cause a warning for an unused variable `Elem`, if the code is compiled with the flag `warn_unused_vars` set. Instead, the code can be rewritten to:

```
member(_Elem, []) ->
    [].
```

Note that since variables starting with an underscore are not anonymous, this will match:

```
{_,_} = {1,2}
```

But this will fail:

```
{_N,_N} = {1,2}
```

The scope for a variable is its function clause. Variables bound in a branch of an `if`, `case`, or `receive` expression must be bound in all branches to have a value outside the expression, otherwise they will be regarded as 'unsafe' outside the expression.

For the `try` expression introduced in Erlang 5.4/OTP-R10B, variable scoping is limited so that variables bound in the expression are always 'unsafe' outside the expression. This will be improved.

5.7.4 Patterns

A pattern has the same structure as a term but may contain unbound variables. Example:

```
Name1
[H|T]
{error,Reason}
```

Patterns are allowed in clause heads, `case` and `receive` expressions, and match expressions.

Match Operator = in Patterns

If `Pattern1` and `Pattern2` are valid patterns, then the following is also a valid pattern:

```
Pattern1 = Pattern2
```

When matched against a term, both `Pattern1` and `Pattern2` will be matched against the term. The idea behind this feature is to avoid reconstruction of terms. Example:

```
f({connect,From,To,Number,Options}, To) ->
    Signal = {connect,From,To,Number,Options},
    ...;
f(Signal, To) ->
    ignore.
```

can instead be written as

```
f({connect,_,To,_,_} = Signal, To) ->
    ...;
f(Signal, To) ->
    ignore.
```

String Prefix in Patterns

When matching strings, the following is a valid pattern:

```
f("prefix" ++ Str) -> ...
```

This is syntactic sugar for the equivalent, but harder to read

```
f([$p,$r,$e,$f,$i,$x | Str]) -> ...
```

Expressions in Patterns

An arithmetic expression can be used within a pattern, if it uses only numeric or bitwise operators, and if its value can be evaluated to a constant at compile-time. Example:

```
case {Value, Result} of
    {?THRESHOLD+1, ok} -> ...
```

This feature was added in Erlang 5.0/OTP R7.

5.7.5 Match

```
Expr1 = Expr2
```

Matches `Expr1`, a pattern, against `Expr2`. If the matching succeeds, any unbound variable in the pattern becomes bound and the value of `Expr2` is returned.

If the matching fails, a `badmatch` run-time error will occur.

Examples:

```
1> {A, B} = {answer, 42}.
{answer,42}
2> A.
answer
3> {C, D} = [1, 2].
** exception error: no match of right hand side value [1,2]
```

5.7.6 Function Calls

```
ExprF(Expr1,...,ExprN)
ExprM:ExprF(Expr1,...,ExprN)
```

In the first form of function calls, `ExprM:ExprF(Expr1,...,ExprN)`, each of `ExprM` and `ExprF` must be an atom or an expression that evaluates to an atom. The function is said to be called by using the *fully qualified function name*. This is often referred to as a *remote* or *external function call*. Example:

```
lists:keysearch(Name, 1, List)
```

In the second form of function calls, `ExprF(Expr1,...,ExprN)`, `ExprF` must be an atom or evaluate to a fun.

If `ExprF` is an atom the function is said to be called by using the *implicitly qualified function name*. If `ExprF/N` is the name of a function explicitly or automatically imported from module `M`, then the call is short for `M:ExprF(Expr1,...,ExprN)`. Otherwise, `ExprF/N` must be a locally defined function. Examples:

```
handle(Msg, State)
spawn(m, init, [])
```

Examples where ExprF is a fun:

```
Fun1 = fun(X) -> X+1 end
Fun1(3)
=> 4

Fun2 = {lists,append}
Fun2([1,2], [3,4])
=> [1,2,3,4]

fun lists:append/2([1,2], [3,4])
=> [1,2,3,4]
```

To avoid possible ambiguities, the fully qualified function name must be used when calling a function with the same name as a BIF, and the compiler does not allow defining a function with the same name as an explicitly imported function.

Note that when calling a local function, there is a difference between using the implicitly or fully qualified function name, as the latter always refers to the latest version of the module. See *Compilation and Code Loading*.

See also the chapter about *Function Evaluation*.

5.7.7 If

```
if
    GuardSeq1 ->
        Body1;
    ...;
    GuardSeqN ->
        BodyN
```

```
end
```

The branches of an `if`-expression are scanned sequentially until a guard sequence `GuardSeq` which evaluates to true is found. Then the corresponding `Body` (sequence of expressions separated by ',') is evaluated.

The return value of `Body` is the return value of the `if` expression.

If no guard sequence is true, an `if_clause` run-time error will occur. If necessary, the guard expression `true` can be used in the last branch, as that guard sequence is always true.

Example:

```
is_greater_than(X, Y) ->
    if
        X>Y ->
            true;
        true -> % works as an 'else' branch
            false
    end
```

5.7.8 Case

```
case Expr of
    Pattern1 [when GuardSeq1] ->
        Body1;
    ...;
    PatternN [when GuardSeqN] ->
        BodyN
end
```

The expression `Expr` is evaluated and the patterns `Pattern` are sequentially matched against the result. If a match succeeds and the optional guard sequence `GuardSeq` is true, the corresponding `Body` is evaluated.

The return value of `Body` is the return value of the `case` expression.

If there is no matching pattern with a true guard sequence, a `case_clause` run-time error will occur.

Example:

```
is_valid_signal(Signal) ->
    case Signal of
        {signal, _What, _From, _To} ->
            true;
        {signal, _What, _To} ->
            true;
        _Else ->
            false
    end.
```

5.7.9 Send

```
Expr1 ! Expr2
```

Sends the value of `Expr2` as a message to the process specified by `Expr1`. The value of `Expr2` is also the return value of the expression.

`Expr1` must evaluate to a pid, a registered name (atom) or a tuple `{Name,Node}`, where `Name` is an atom and `Node` a node name, also an atom.

- If `Expr1` evaluates to a name, but this name is not registered, a `badarg` run-time error will occur.
- Sending a message to a pid never fails, even if the pid identifies a non-existing process.
- Distributed message sending, that is if `Expr1` evaluates to a tuple `{Name,Node}` (or a pid located at another node), also never fails.

5.7.10 Receive

```
receive
    Pattern1 [when GuardSeq1] ->
        Body1;
    ...;
    PatternN [when GuardSeqN] ->
        BodyN
end
```

Receives messages sent to the process using the send operator (!). The patterns `Pattern` are sequentially matched against the first message in time order in the mailbox, then the second, and so on. If a match succeeds and the optional guard sequence `GuardSeq` is true, the corresponding `Body` is evaluated. The matching message is consumed, that is removed from the mailbox, while any other messages in the mailbox remain unchanged.

The return value of `Body` is the return value of the `receive` expression.

`receive` never fails. Execution is suspended, possibly indefinitely, until a message arrives that does match one of the patterns and with a true guard sequence.

Example:

```
wait_for_onhook() ->
    receive
        onhook ->
            disconnect(),
            idle();
        {connect, B} ->
            B ! {busy, self()},
            wait_for_onhook()
    end.
```

It is possible to augment the `receive` expression with a timeout:

```
receive
    Pattern1 [when GuardSeq1] ->
        Body1;
    ...;
    PatternN [when GuardSeqN] ->
        BodyN
after
    ExprT ->
        BodyT
end
```

`ExprT` should evaluate to an integer. The highest allowed value is 16#ffffffff, that is, the value must fit in 32 bits. `receive..after` works exactly as `receive`, except that if no matching message has arrived within `ExprT` milliseconds, then `BodyT` is evaluated instead and its return value becomes the return value of the `receive..after` expression.

Example:

```
wait_for_onhook() ->
    receive
        onhook ->
            disconnect(),
            idle();
        {connect, B} ->
            B ! {busy, self()},
            wait_for_onhook()
    after
        60000 ->
            disconnect(),
            error()
    end.
```

It is legal to use a `receive..after` expression with no branches:

```
receive
after
    ExprT ->
        BodyT
end
```

This construction will not consume any messages, only suspend execution in the process for `ExprT` milliseconds and can be used to implement simple timers.

Example:

```
timer() ->
    spawn(m, timer, [self()]).

timer(Pid) ->
    receive
    after
        5000 ->
            Pid ! timeout
    end.
```

There are two special cases for the timeout value `ExprT`:

`infinity`
: The process should wait indefinitely for a matching message -- this is the same as not using a timeout. Can be useful for timeout values that are calculated at run-time.

0
: If there is no matching message in the mailbox, the timeout will occur immediately.

5.7.11 Term Comparisons

```
Expr1 op Expr2
```

op	*Description*
==	equal to
/=	not equal to
=<	less than or equal to
<	less than
>=	greater than or equal to
>	greater than
=:=	exactly equal to
=/=	exactly not equal to

Table 7.1: Term Comparison Operators.

The arguments may be of different data types. The following order is defined:

```
number < atom < reference < fun < port < pid < tuple < list < bit string
```

Lists are compared element by element. Tuples are ordered by size, two tuples with the same size are compared element by element.

If one of the compared terms is an integer and the other a float, the integer is first converted into a float, unless the operator is one of =:= and =/=. If the integer is too big to fit in a float no conversion is done, but the order is determined by inspecting the sign of the numbers.

Returns the Boolean value of the expression, `true` or `false`.

Examples:

```
1> 1==1.0.
true
2> 1=:=1.0.
false
3> 1 > a.
false
```

5.7.12 Arithmetic Expressions

```
op Expr
Expr1 op Expr2
```

op	*Description*	*Argument type*
+	unary +	number
-	unary -	number
+		number
-		number
*		number
/	floating point division	number
bnot	unary bitwise not	integer
div	integer division	integer
rem	integer remainder of X/Y	integer
band	bitwise and	integer
bor	bitwise or	integer
bxor	arithmetic bitwise xor	integer
bsl	arithmetic bitshift left	integer
bsr	bitshift right	integer

Table 7.2: Arithmetic Operators.

Examples:

```
1> +1.
1
2> -1.
-1
3> 1+1.
2
4> 4/2.
2.0
5> 5 div 2.
2
6> 5 rem 2.
1
7> 2#10 band 2#01.
0
8> 2#10 bor 2#01.
3
9> a + 10.
```

```
** exception error: bad argument in an arithmetic expression
     in operator  +/2
        called as a + 10
10> 1 bsl (1 bsl 64).
** exception error: a system limit has been reached
     in operator  bsl/2
        called as 1 bsl 18446744073709551616
```

5.7.13 Boolean Expressions

```
op Expr
Expr1 op Expr2
```

op	*Description*
not	unary logical not
and	logical and
or	logical or
xor	logical xor

Table 7.3: Logical Operators.

Examples:

```
1> not true.
false
2> true and false.
false
3> true xor false.
true
4> true or garbage.
** exception error: bad argument
     in operator  or/2
        called as true or garbage
```

5.7.14 Short-Circuit Expressions

```
Expr1 orelse Expr2
Expr1 andalso Expr2
```

Expressions where `Expr2` is evaluated only if necessary. That is, `Expr2` is evaluated only if `Expr1` evaluates to `false` in an `orelse` expression, or only if `Expr1` evaluates to `true` in an `andalso` expression. Returns either the value of `Expr1` (that is, `true` or `false`) or the value of `Expr2` (if `Expr2` was evaluated).

Example 1:

```
case A >= -1.0 andalso math:sqrt(A+1) > B of
```

This will work even if `A` is less than `-1.0`, since in that case, `math:sqrt/1` is never evaluated.

Example 2:

```
OnlyOne = is_atom(L) orelse
          (is_list(L) andalso length(L) == 1),
```

From R13A, `Expr2` is no longer required to evaluate to a boolean value. As a consequence, `andalso` and `orelse` are now tail-recursive. For instance, the following function is tail-recursive in R13A and later:

```
all(Pred, [Hd|Tail]) ->
    Pred(Hd) andalso all(Pred, Tail);
all(_, []) ->
    true.
```

5.7.15 List Operations

```
Expr1 ++ Expr2
Expr1 -- Expr2
```

The list concatenation operator `++` appends its second argument to its first and returns the resulting list.

The list subtraction operator `--` produces a list which is a copy of the first argument, subjected to the following procedure: for each element in the second argument, the first occurrence of this element (if any) is removed.

Example:

```
1> [1,2,3]++[4,5].
[1,2,3,4,5]
2> [1,2,3,2,1,2]--[2,1,2].
[3,1,2]
```

> ### Warning:
>
> The complexity of `A -- B` is proportional to `length(A)*length(B)`, meaning that it will be very slow if both `A` and `B` are long lists.

5.7.16 Bit Syntax Expressions

```
<<>>
<<E1,...,En>>
```

Each element `Ei` specifies a *segment* of the bit string. Each element `Ei` is a value, followed by an optional *size expression* and an optional *type specifier list*.

```
Ei = Value |
     Value:Size |
     Value/TypeSpecifierList |
     Value:Size/TypeSpecifierList
```

Used in a bit string construction, `Value` is an expression which should evaluate to an integer, float or bit string. If the expression is something else than a single literal or variable, it should be enclosed in parenthesis.

Used in a bit string matching, `Value` must be a variable, or an integer, float or string.

Note that, for example, using a string literal as in `<<"abc">>` is syntactic sugar for `<<$a,$b,$c>>`.

Used in a bit string construction, `Size` is an expression which should evaluate to an integer.

Used in a bit string matching, `Size` must be an integer or a variable bound to an integer.

The value of `Size` specifies the size of the segment in units (see below). The default value depends on the type (see below). For `integer` it is 8, for `float` it is 64, for `binary` and `bitstring` it is the whole binary or bit string. In matching, this default value is only valid for the very last element. All other bit string or binary elements in the matching must have a size specification.

For the `utf8`, `utf16`, and `utf32` types, `Size` must not be given. The size of the segment is implicitly determined by the type and value itself.

`TypeSpecifierList` is a list of type specifiers, in any order, separated by hyphens (-). Default values are used for any omitted type specifiers.

`Type`= `integer` | `float` | `binary` | `bytes` | `bitstring` | `bits` | `utf8` | `utf16` | `utf32`
: The default is `integer`. `bytes` is a shorthand for `binary` and `bits` is a shorthand for `bitstring`. See below for more information about the `utf` types.

`Signedness`= `signed` | `unsigned`
: Only matters for matching and when the type is `integer`. The default is `unsigned`.

`Endianness`= `big` | `little` | `native`
: Native-endian means that the endianness will be resolved at load time to be either big-endian or little-endian, depending on what is native for the CPU that the Erlang machine is run on. Endianness only matters when the Type is either `integer`, `utf16`, `utf32`, or `float`. The default is `big`.

`Unit`= `unit:IntegerLiteral`
: The allowed range is 1..256. Defaults to 1 for `integer`, `float` and `bitstring`, and to 8 for `binary`. No unit specifier must be given for the types `utf8`, `utf16`, and `utf32`.

The value of `Size` multiplied with the unit gives the number of bits. A segment of type `binary` must have a size that is evenly divisible by 8.

> **Note:**
>
> When constructing binaries, if the size `N` of an integer segment is too small to contain the given integer, the most significant bits of the integer will be silently discarded and only the `N` least significant bits will be put into the binary.

The types `utf8`, `utf16`, and `utf32` specifies encoding/decoding of the *Unicode Transformation Format*s UTF-8, UTF-16, and UTF-32, respectively.

When constructing a segment of a `utf` type, `Value` must be an integer in one of the ranges 0..16#D7FF, 16#E000..16#FFFD, or 16#10000..16#10FFFF (i.e. a valid Unicode code point). Construction will fail with a `badarg` exception if `Value` is outside the allowed ranges. The size of the resulting binary segment depends on the type and/

or `Value`. For `utf8`, `Value` will be encoded in 1 through 4 bytes. For `utf16`, `Value` will be encoded in 2 or 4 bytes. Finally, for `utf32`, `Value` will always be encoded in 4 bytes.

When constructing, a literal string may be given followed by one of the UTF types, for example: `<<"abc"/utf8>>` which is syntatic sugar for `<<$a/utf8,$b/utf8,$c/utf8>>`.

A successful match of a segment of a `utf` type results in an integer in one of the ranges 0..16#D7FF, 16#E000..16#FFFD, or 16#10000..16#10FFFF (i.e. a valid Unicode code point). The match will fail if returned value would fall outside those ranges.

A segment of type `utf8` will match 1 to 4 bytes in the binary, if the binary at the match position contains a valid UTF-8 sequence. (See RFC-2279 or the Unicode standard.)

A segment of type `utf16` may match 2 or 4 bytes in the binary. The match will fail if the binary at the match position does not contain a legal UTF-16 encoding of a Unicode code point. (See RFC-2781 or the Unicode standard.)

A segment of type `utf32` may match 4 bytes in the binary in the same way as an `integer` segment matching 32 bits. The match will fail if the resulting integer is outside the legal ranges mentioned above.

Examples:

```
1> Bin1 = <<1,17,42>>.
<<1,17,42>>
2> Bin2 = <<"abc">>.
<<97,98,99>>
3> Bin3 = <<1,17,42:16>>.
<<1,17,0,42>>
4> <<A,B,C:16>> = <<1,17,42:16>>.
<<1,17,0,42>>
5> C.
42
6> <<D:16,E,F>> = <<1,17,42:16>>.
<<1,17,0,42>>
7> D.
273
8> F.
42
9> <<G,H/binary>> = <<1,17,42:16>>.
<<1,17,0,42>>
10> H.
<<17,0,42>>
11> <<G,H/bitstring>> = <<1,17,42:12>>.
<<1,17,1,10:4>>
12> H.
<<17,1,10:4>>
13> <<1024/utf8>>.
<<208,128>>
```

Note that bit string patterns cannot be nested.

Note also that "`B=<<1>>`" is interpreted as "`B =<<1>>`" which is a syntax error. The correct way is to write a space after '=': "`B= <<1>>`.

More examples can be found in *Programming Examples*.

5.7.17 Fun Expressions

```
fun
    (Pattern11,...,Pattern1N) [when GuardSeq1] ->
        Body1;
```

```
    ...;
    (PatternK1,...,PatternKN) [when GuardSeqK] ->
        BodyK
end
```

A fun expression begins with the keyword `fun` and ends with the keyword `end`. Between them should be a function declaration, similar to a *regular function declaration*, except that no function name is specified.

Variables in a fun head shadow variables in the function clause surrounding the fun expression, and variables bound in a fun body are local to the fun body.

The return value of the expression is the resulting fun.

Examples:

```
1> Fun1 = fun (X) -> X+1 end.
#Fun<erl_eval.6.39074546>
2> Fun1(2).
3
3> Fun2 = fun (X) when X>=5 -> gt; (X) -> lt end.
#Fun<erl_eval.6.39074546>
4> Fun2(7).
gt
```

The following fun expressions are also allowed:

```
fun Name/Arity
fun Module:Name/Arity
```

In `Name/Arity`, `Name` is an atom and `Arity` is an integer. `Name/Arity` must specify an existing local function. The expression is syntactic sugar for:

```
fun (Arg1,...,ArgN) -> Name(Arg1,...,ArgN) end
```

In `Module:Name/Arity`, `Module` and `Name` are atoms and `Arity` is an integer. A fun defined in this way will refer to the function `Name` with arity `Arity` in the *latest* version of module `Module`.

When applied to a number N of arguments, a tuple `{Module,FunctionName}` is interpreted as a fun, referring to the function `FunctionName` with arity N in the module `Module`. The function must be exported. *This usage is deprecated.* See *Function Calls* for an example.

More examples can be found in *Programming Examples*.

5.7.18 Catch and Throw

```
catch Expr
```

Returns the value of `Expr` unless an exception occurs during the evaluation. In that case, the exception is caught. For exceptions of class `error`, that is run-time errors: `{'EXIT',{Reason,Stack}}` is returned. For exceptions of class `exit`, that is the code called `exit(Term)`: `{'EXIT',Term}` is returned. For exceptions of class `throw`, that is the code called `throw(Term)`: `Term` is returned.

`Reason` depends on the type of error that occurred, and `Stack` is the stack of recent function calls, see *Errors and Error Handling*.

Examples:

```
1> catch 1+2.
3
2> catch 1+a.
{'EXIT',{badarith,[...]}}
```

Note that `catch` has low precedence and catch subexpressions often needs to be enclosed in a block expression or in parenthesis:

```
3> A = catch 1+2.
** 1: syntax error before: 'catch' **
4> A = (catch 1+2).
3
```

The BIF `throw(Any)` can be used for non-local return from a function. It must be evaluated within a `catch`, which will return the value `Any`. Example:

```
5> catch throw(hello).
hello
```

If `throw/1` is not evaluated within a catch, a `nocatch` run-time error will occur.

5.7.19 Try

```
try Exprs
catch
    [Class1:]ExceptionPattern1 [when ExceptionGuardSeq1] ->
        ExceptionBody1;
    [ClassN:]ExceptionPatternN [when ExceptionGuardSeqN] ->
        ExceptionBodyN
end
```

This is an enhancement of *catch* that appeared in Erlang 5.4/OTP-R10B. It gives the possibility do distinguish between different exception classes, and to choose to handle only the desired ones, passing the others on to an enclosing `try` or `catch` or to default error handling.

Note that although the keyword `catch` is used in the `try` expression, there is not a `catch` expression within the `try` expression.

Returns the value of `Exprs` (a sequence of expressions `Expr1, ..., ExprN`) unless an exception occurs during the evaluation. In that case the exception is caught and the patterns `ExceptionPattern` with the right exception class `Class` are sequentially matched against the caught exception. An omitted `Class` is shorthand for `throw`. If a match succeeds and the optional guard sequence `ExceptionGuardSeq` is true, the corresponding `ExceptionBody` is evaluated to become the return value.

If an exception occurs during evaluation of `Exprs` but there is no matching `ExceptionPattern` of the right `Class` with a true guard sequence, the exception is passed on as if `Exprs` had not been enclosed in a `try` expression.

If an exception occurs during evaluation of `ExceptionBody` it is not caught.

The `try` expression can have an `of` section:

```
try Exprs of
    Pattern1 [when GuardSeq1] ->
        Body1;
    ...;
    PatternN [when GuardSeqN] ->
        BodyN
catch
    [Class1:]ExceptionPattern1 [when ExceptionGuardSeq1] ->
        ExceptionBody1;
    ...;
    [ClassN:]ExceptionPatternN [when ExceptionGuardSeqN] ->
        ExceptionBodyN
end
```

If the evaluation of `Exprs` succeeds without an exception, the patterns `Pattern` are sequentially matched against the result in the same way as for a *case* expression, except that if the matching fails, a `try_clause` run-time error will occur.

An exception occurring during the evaluation of `Body` is not caught.

The `try` expression can also be augmented with an `after` section, intended to be used for cleanup with side effects:

```
try Exprs of
    Pattern1 [when GuardSeq1] ->
        Body1;
    ...;
    PatternN [when GuardSeqN] ->
        BodyN
catch
    [Class1:]ExceptionPattern1 [when ExceptionGuardSeq1] ->
        ExceptionBody1;
    ...;
    [ClassN:]ExceptionPatternN [when ExceptionGuardSeqN] ->
        ExceptionBodyN
after
    AfterBody
end
```

`AfterBody` is evaluated after either `Body` or `ExceptionBody` no matter which one. The evaluated value of `AfterBody` is lost; the return value of the `try` expression is the same with an `after` section as without.

Even if an exception occurs during evaluation of `Body` or `ExceptionBody`, `AfterBody` is evaluated. In this case the exception is passed on after `AfterBody` has been evaluated, so the exception from the `try` expression is the same with an `after` section as without.

If an exception occurs during evaluation of `AfterBody` itself it is not caught, so if `AfterBody` is evaluated after an exception in `Exprs`, `Body` or `ExceptionBody`, that exception is lost and masked by the exception in `AfterBody`.

The `of`, `catch` and `after` sections are all optional, as long as there is at least a `catch` or an `after` section, so the following are valid `try` expressions:

```
try Exprs of
    Pattern when GuardSeq ->
        Body
```

```
after
    AfterBody
end

try Exprs
catch
    ExpressionPattern ->
        ExpressionBody
after
    AfterBody
end

try Exprs after AfterBody end
```

Example of using `after`, this code will close the file even in the event of exceptions in `file:read/2` or in `binary_to_term/1`, and exceptions will be the same as without the `try`...`after`...`end` expression:

```
termize_file(Name) ->
    {ok,F} = file:open(Name, [read,binary]),
    try
        {ok,Bin} = file:read(F, 1024*1024),
        binary_to_term(Bin)
    after
        file:close(F)
    end.
```

Example: Using `try` to emulate `catch Expr`.

```
try Expr
catch
    throw:Term -> Term;
    exit:Reason -> {'EXIT',Reason}
    error:Reason -> {'EXIT',{Reason,erlang:get_stacktrace()}}
end
```

5.7.20 Parenthesized Expressions

```
(Expr)
```

Parenthesized expressions are useful to override *operator precedences*, for example in arithmetic expressions:

```
1> 1 + 2 * 3.
7
2> (1 + 2) * 3.
9
```

5.7.21 Block Expressions

```
begin
   Expr1,
   ...,
```

```
   ExprN
end
```

Block expressions provide a way to group a sequence of expressions, similar to a clause body. The return value is the value of the last expression `ExprN`.

5.7.22 List Comprehensions

List comprehensions are a feature of many modern functional programming languages. Subject to certain rules, they provide a succinct notation for generating elements in a list.

List comprehensions are analogous to set comprehensions in Zermelo-Frankel set theory and are called ZF expressions in Miranda. They are analogous to the `setof` and `findall` predicates in Prolog.

List comprehensions are written with the following syntax:

```
[Expr || Qualifier1,...,QualifierN]
```

`Expr` is an arbitrary expression, and each `Qualifier` is either a generator or a filter.

- A *generator* is written as:
  `Pattern <- ListExpr`.
  `ListExpr` must be an expression which evaluates to a list of terms.
- A *bit string generator* is written as:
  `BitstringPattern <= BitStringExpr`.
  `BitStringExpr` must be an expression which evaluates to a bitstring.
- A *filter* is an expression which evaluates to `true` or `false`.

The variables in the generator patterns shadow variables in the function clause surrounding the list comprehensions.

A list comprehension returns a list, where the elements are the result of evaluating `Expr` for each combination of generator list elements and bit string generator elements for which all filters are true.

Example:

```
1> [X*2 || X <- [1,2,3]].
[2,4,6]
```

More examples can be found in *Programming Examples*.

5.7.23 Bit String Comprehensions

Bit string comprehensions are analogous to List Comprehensions. They are used to generate bit strings efficiently and succinctly.

Bit string comprehensions are written with the following syntax:

```
<< BitString || Qualifier1,...,QualifierN >>
```

`BitString` is a bit string expression, and each `Qualifier` is either a generator, a bit string generator or a filter.

- A *generator* is written as:
  `Pattern <- ListExpr`.

`ListExpr` must be an expression which evaluates to a list of terms.

- A *bit string generator* is written as:
  `BitstringPattern <= BitStringExpr.`
  `BitStringExpr` must be an expression which evaluates to a bitstring.
- A *filter* is an expression which evaluates to `true` or `false`.

The variables in the generator patterns shadow variables in the function clause surrounding the bit string comprehensions.

A bit string comprehension returns a bit string, which is created by concatenating the results of evaluating `BitString` for each combination of bit string generator elements for which all filters are true.

Example:

```
1> << << (X*2) >> ||
<<X>> <= << 1,2,3 >> >>.
<<2,4,6>>
```

More examples can be found in *Programming Examples*.

5.7.24 Guard Sequences

A *guard sequence* is a sequence of guards, separated by semicolon (;). The guard sequence is true if at least one of the guards is true. (The remaining guards, if any, will not be evaluated.)
`Guard1;...;GuardK`

A *guard* is a sequence of guard expressions, separated by comma (,). The guard is true if all guard expressions evaluate to `true`.
`GuardExpr1,...,GuardExprN`

The set of valid *guard expressions* (sometimes called guard tests) is a subset of the set of valid Erlang expressions. The reason for restricting the set of valid expressions is that evaluation of a guard expression must be guaranteed to be free of side effects. Valid guard expressions are:

- the atom `true`,
- other constants (terms and bound variables), all regarded as false,
- calls to the BIFs specified below,
- term comparisons,
- arithmetic expressions,
- boolean expressions, and
- short-circuit expressions (`andalso`/`orelse`).

`is_atom/1`
`is_binary/1`
`is_bitstring/1`
`is_float/1`
`is_function/1`
`is_function/2`

is_integer/1
is_list/1
is_number/1
is_pid/1
is_port/1
is_record/2
is_record/3
is_reference/1
is_tuple/1

Table 7.4: Type Test BIFs.

Note that most type test BIFs have older equivalents, without the `is_` prefix. These old BIFs are retained for backwards compatibility only and should not be used in new code. They are also only allowed at top level. For example, they are not allowed in boolean expressions in guards.

abs(Number)
bit_size(Bitstring)
byte_size(Bitstring)
element(N, Tuple)
float(Term)
hd(List)
length(List)
node()
node(Pid|Ref|Port)
round(Number)
self()
size(Tuple|Bitstring)
tl(List)
trunc(Number)

`tuple_size(Tuple)`

Table 7.5: Other BIFs Allowed in Guard Expressions.

If an arithmetic expression, a boolean expression, a short-circuit expression, or a call to a guard BIF fails (because of invalid arguments), the entire guard fails. If the guard was part of a guard sequence, the next guard in the sequence (that is, the guard following the next semicolon) will be evaluated.

5.7.25 Operator Precedence

Operator precedence in falling priority:

:	
#	
Unary + - bnot not	
/ * div rem band and	Left associative
+ - bor bxor bsl bsr or xor	Left associative
++ --	Right associative
== /= =< < >= > =:= =/=	
andalso	
orelse	
= !	Right associative
catch	

Table 7.6: Operator Precedence.

When evaluating an expression, the operator with the highest priority is evaluated first. Operators with the same priority are evaluated according to their associativity. Example: The left associative arithmetic operators are evaluated left to right:

```
6 + 5 * 4 - 3 / 2 evaluates to
6 + 20 - 1.5 evaluates to
26 - 1.5 evaluates to
24.5
```

5.8 The Preprocessor

5.8.1 File Inclusion

A file can be included in the following way:

```
-include(File).
-include_lib(File).
```

`File`, a string, should point out a file. The contents of this file are included as-is, at the position of the directive.

Include files are typically used for record and macro definitions that are shared by several modules. It is recommended that the file name extension `.hrl` be used for include files.

`File` may start with a path component `$VAR`, for some string `VAR`. If that is the case, the value of the environment variable `VAR` as returned by `os:getenv(VAR)` is substituted for `$VAR`. If `os:getenv(VAR)` returns `false`, `$VAR` is left as is.

If the filename `File` is absolute (possibly after variable substitution), the include file with that name is included. Otherwise, the specified file is searched for in the current working directory, in the same directory as the module being compiled, and in the directories given by the `include` option, in that order. See `erlc(1)` and `compile(3)` for details.

Examples:

```
-include("my_records.hrl").
-include("incdir/my_records.hrl").
-include("/home/user/proj/my_records.hrl").
-include("$PROJ_ROOT/my_records.hrl").
```

`include_lib` is similar to `include`, but should not point out an absolute file. Instead, the first path component (possibly after variable substitution) is assumed to be the name of an application. Example:

```
-include_lib("kernel/include/file.hrl").
```

The code server uses `code:lib_dir(kernel)` to find the directory of the current (latest) version of Kernel, and then the subdirectory `include` is searched for the file `file.hrl`.

5.8.2 Defining and Using Macros

A macro is defined the following way:

```
-define(Const, Replacement).
-define(Func(Var1,...,VarN), Replacement).
```

A macro definition can be placed anywhere among the attributes and function declarations of a module, but the definition must come before any usage of the macro.

If a macro is used in several modules, it is recommended that the macro definition is placed in an include file.

A macro is used the following way:

```
?Const
?Func(Arg1,...,ArgN)
```

Macros are expanded during compilation. A simple macro `?Const` will be replaced with `Replacement`. Example:

```
-define(TIMEOUT, 200).
...
call(Request) ->
    server:call(refserver, Request, ?TIMEOUT).
```

This will be expanded to:

```
call(Request) ->
    server:call(refserver, Request, 200).
```

A macro `?Func(Arg1,...,ArgN)` will be replaced with `Replacement`, where all occurrences of a variable `Var` from the macro definition are replaced with the corresponding argument `Arg`. Example:

```
-define(MACRO1(X, Y), {a, X, b, Y}).
...
bar(X) ->
    ?MACRO1(a, b),
    ?MACRO1(X, 123)
```

This will be expanded to:

```
bar(X) ->
    {a,a,b,b},
    {a,X,b,123}.
```

It is good programming practice, but not mandatory, to ensure that a macro definition is a valid Erlang syntactic form.

To view the result of macro expansion, a module can be compiled with the `'P'` option. `compile:file(File, ['P'])`. This produces a listing of the parsed code after preprocessing and parse transforms, in the file `File.P`.

5.8.3 Predefined Macros

The following macros are predefined:

`?MODULE`
: The name of the current module.

`?MODULE_STRING`.
: The name of the current module, as a string.

`?FILE`.
: The file name of the current module.

`?LINE`.
: The current line number.

`?MACHINE`.
: The machine name, `'BEAM'`.

5.8.4 Macros Overloading

It is possible to overload macros, except for predefined macros. An overloaded macro has more than one definition, each with a different number of arguments.

The feature was added in Erlang 5.7.5/OTP R13B04.

A macro `?Func(Arg1,...,ArgN)` with a (possibly empty) list of arguments results in an error message if there is at least one definition of `Func` with arguments, but none with N arguments.

Assuming these definitions:

```
-define(F0(), c).
-define(F1(A), A).
-define(C, m:f).
```

the following will not work:

```
f0() ->
    ?F0. % No, an empty list of arguments expected.

f1(A) ->
    ?F1(A, A). % No, exactly one argument expected.
```

On the other hand,

```
f() ->
    ?C().
```

will expand to

```
f() ->
    m:f().
```

5.8.5 Flow Control in Macros

The following macro directives are supplied:

`-undef(Macro).`
: Causes the macro to behave as if it had never been defined.

`-ifdef(Macro).`
: Evaluate the following lines only if `Macro` is defined.

`-ifndef(Macro).`
: Evaluate the following lines only if `Macro` is not defined.

`-else.`
: Only allowed after an `ifdef` or `ifndef` directive. If that condition was false, the lines following `else` are evaluated instead.

`-endif.`
: Specifies the end of an `ifdef` or `ifndef` directive.

> **Note:**
>
> The macro directives cannot be used inside functions.

Example:

```
-module(m).
...

-ifdef(debug).
-define(LOG(X), io:format("{~p,~p}: ~p~n", [?MODULE,?LINE,X])).
-else.
-define(LOG(X), true).
-endif.

...
```

When trace output is desired, `debug` should be defined when the module `m` is compiled:

```
% erlc -Ddebug m.erl

or

1> c(m, {d, debug}).
{ok,m}
```

`?LOG(Arg)` will then expand to a call to `io:format/2` and provide the user with some simple trace output.

5.8.6 Stringifying Macro Arguments

The construction `??Arg`, where `Arg` is a macro argument, will be expanded to a string containing the tokens of the argument. This is similar to the `#arg` stringifying construction in C.

The feature was added in Erlang 5.0/OTP R7.

Example:

```
-define(TESTCALL(Call), io:format("Call ~s: ~w~n", [??Call, Call])).

?TESTCALL(myfunction(1,2)),
?TESTCALL(you:function(2,1)).
```

results in

```
io:format("Call ~s: ~w~n",["myfunction ( 1 , 2 )",m:myfunction(1,2)]),
io:format("Call ~s: ~w~n",["you : function ( 2 , 1 )",you:function(2,1)]).
```

That is, a trace output with both the function called and the resulting value.

5.9 Records

A record is a data structure for storing a fixed number of elements. It has named fields and is similar to a struct in C. Record expressions are translated to tuple expressions during compilation. Therefore, record expressions are not understood by the shell unless special actions are taken. See `shell(3)` for details.

More record examples can be found in *Programming Examples*.

5.9.1 Defining Records

A record definition consists of the name of the record, followed by the field names of the record. Record and field names must be atoms. Each field can be given an optional default value. If no default value is supplied, `undefined` will be used.

```
-record(Name, {Field1 [= Value1],
               ...
               FieldN [= ValueN]}).
```

A record definition can be placed anywhere among the attributes and function declarations of a module, but the definition must come before any usage of the record.

If a record is used in several modules, it is recommended that the record definition is placed in an include file.

5.9.2 Creating Records

The following expression creates a new `Name` record where the value of each field `FieldI` is the value of evaluating the corresponding expression `ExprI`:

```
#Name{Field1=Expr1,...,FieldK=ExprK}
```

The fields may be in any order, not necessarily the same order as in the record definition, and fields can be omitted. Omitted fields will get their respective default value instead.

If several fields should be assigned the same value, the following construction can be used:

```
#Name{Field1=Expr1,...,FieldK=ExprK, _=ExprL}
```

Omitted fields will then get the value of evaluating `ExprL` instead of their default values. This feature was added in Erlang 5.1/OTP R8 and is primarily intended to be used to create patterns for ETS and Mnesia match functions. Example:

```
-record(person, {name, phone, address}).

...

lookup(Name, Tab) ->
    ets:match_object(Tab, #person{name=Name, _='_'}).
```

5.9.3 Accessing Record Fields

```
Expr#Name.Field
```

Returns the value of the specified field. `Expr` should evaluate to a `Name` record.

The following expression returns the position of the specified field in the tuple representation of the record:

```
#Name.Field
```

Example:

```
-record(person, {name, phone, address}).

...

lookup(Name, List) ->
    lists:keysearch(Name, #person.name, List).
```

5.9.4 Updating Records

```
Expr#Name{Field1=Expr1,...,FieldK=ExprK}
```

`Expr` should evaluate to a `Name` record. Returns a copy of this record, with the value of each specified field `FieldI` changed to the value of evaluating the corresponding expression `ExprI`. All other fields retain their old values.

5.9.5 Records in Guards

Since record expressions are expanded to tuple expressions, creating records and accessing record fields are allowed in guards. However all subexpressions, for example for field initiations, must of course be valid guard expressions as well. Examples:

```
handle(Msg, State) when Msg==#msg{to=void, no=3} ->
    ...

handle(Msg, State) when State#state.running==true ->
    ...
```

There is also a type test BIF `is_record(Term, RecordTag)`. Example:

```
is_person(P) when is_record(P, person) ->
    true;
is_person(_P) ->
    false.
```

5.9.6 Records in Patterns

A pattern that will match a certain record is created the same way as a record is created:

```
#Name{Field1=Expr1,...,FieldK=ExprK}
```

In this case, one or more of `Expr1`...`ExprK` may be unbound variables.

5.9.7 Internal Representation of Records

Record expressions are translated to tuple expressions during compilation. A record defined as

```
-record(Name, {Field1,...,FieldN}).
```

is internally represented by the tuple

```
{Name,Value1,...,ValueN}
```

where each `ValueI` is the default value for `FieldI`.

To each module using records, a pseudo function is added during compilation to obtain information about records:

```
record_info(fields, Record) -> [Field]
record_info(size, Record) -> Size
```

`Size` is the size of the tuple representation, that is one more than the number of fields.

In addition, `#Record.Name` returns the index in the tuple representation of `Name` of the record `Record`. `Name` must be an atom.

5.10 Errors and Error Handling

5.10.1 Terminology

Errors can roughly be divided into four different types:

- Compile-time errors
- Logical errors
- Run-time errors
- Generated errors

A compile-time error, for example a syntax error, should not cause much trouble as it is caught by the compiler.

A logical error is when a program does not behave as intended, but does not crash. An example could be that nothing happens when a button in a graphical user interface is clicked.

A run-time error is when a crash occurs. An example could be when an operator is applied to arguments of the wrong type. The Erlang programming language has built-in features for handling of run-time errors.

A run-time error can also be emulated by calling `erlang:error(Reason)`, `erlang:error(Reason, Args)` (those appeared in Erlang 5.4/OTP-R10), `erlang:fault(Reason)` or `erlang:fault(Reason, Args)` (old equivalents).

A run-time error is another name for an exception of class `error`.

A generated error is when the code itself calls `exit/1` or `throw/1`. Note that emulated run-time errors are not denoted as generated errors here.

Generated errors are exceptions of classes `exit` and `throw`.

When a run-time error or generated error occurs in Erlang, execution for the process which evaluated the erroneous expression is stopped. This is referred to as a *failure*, that execution or evaluation *fails*, or that the process *fails*, *terminates* or *exits*. Note that a process may terminate/exit for other reasons than a failure.

A process that terminates will emit an *exit signal* with an *exit reason* that says something about which error has occurred. Normally, some information about the error will be printed to the terminal.

5.10.2 Exceptions

Exceptions are run-time errors or generated errors and are of three different classes, with different origins. The *try* expression (appeared in Erlang 5.4/OTP-R10B) can distinguish between the different classes, whereas the *catch* expression can not. They are described in the Expressions chapter.

Class	*Origin*
`error`	Run-time error for example `1+a`, or the process called `erlang:error/1,2` (appeared in Erlang 5.4/OTP-R10B) or `erlang:fault/1,2` (old equivalent)
`exit`	The process called `exit/1`
`throw`	The process called `throw/1`

Table 10.1: Exception Classes.

An exception consists of its class, an exit reason (the *Exit Reason*), and a stack trace (that aids in finding the code location of the exception).

The stack trace can be retrieved using `erlang:get_stacktrace/0` (new in Erlang 5.4/OTP-R10B from within a `try` expression, and is returned for exceptions of class `error` from a `catch` expression.

An exception of class `error` is also known as a run-time error.

5.10.3 Handling of Run-Time Errors in Erlang

Error Handling Within Processes

It is possible to prevent run-time errors and other exceptions from causing the process to terminate by using `catch` or `try`, see the Expressions chapter about *Catch* and *Try*.

Error Handling Between Processes

Processes can monitor other processes and detect process terminations, see the *Processes* chapter.

5.10.4 Exit Reasons

When a run-time error occurs, that is an exception of class `error`, the exit reason is a tuple `{Reason,Stack}`. `Reason` is a term indicating the type of error:

Reason	*Type of error*
`badarg`	Bad argument. The argument is of wrong data type, or is otherwise badly formed.
`badarith`	Bad argument in an arithmetic expression.
`{badmatch,V}`	Evaluation of a match expression failed. The value `V` did not match.
`function_clause`	No matching function clause is found when evaluating a function call.
`{case_clause,V}`	No matching branch is found when evaluating a `case` expression. The value `V` did not match.
`if_clause`	No true branch is found when evaluating an `if` expression.
`{try_clause,V}`	No matching branch is found when evaluating the of-section of a `try` expression. The value `V` did not match.
`undef`	The function cannot be found when evaluating a function call.
`{badfun,F}`	There is something wrong with a fun `F`.
`{badarity,F}`	A fun is applied to the wrong number of arguments. `F` describes the fun and the arguments.
`timeout_value`	The timeout value in a `receive..after` expression is evaluated to something else than an integer or `infinity`.
`noproc`	Trying to link to a non-existing process.
`{nocatch,V}`	Trying to evaluate a `throw` outside a `catch`. `V` is the thrown term.
`system_limit`	A system limit has been reached. See Efficiency Guide for information about system limits.

Table 10.2: Exit Reasons.

`Stack` is the stack of function calls being evaluated when the error occurred, given as a list of tuples `{Module,Name,Arity}` with the most recent function call first. The most recent function call tuple may in some cases be `{Module,Name,[Arg]}`.

5.11 Processes

5.11.1 Processes

Erlang is designed for massive concurrency. Erlang processes are light-weight (grow and shrink dynamically) with small memory footprint, fast to create and terminate and the scheduling overhead is low.

5.11.2 Process Creation

A process is created by calling `spawn`:

```
spawn(Module, Name, Args) -> pid()
  Module = Name = atom()
  Args = [Arg1,...,ArgN]
    ArgI = term()
```

`spawn` creates a new process and returns the pid.

The new process will start executing in `Module:Name(Arg1,...,ArgN)` where the arguments is the elements of the (possible empty) `Args` argument list.

There exist a number of other `spawn` BIFs, for example `spawn/4` for spawning a process at another node.

5.11.3 Registered Processes

Besides addressing a process by using its pid, there are also BIFs for registering a process under a name. The name must be an atom and is automatically unregistered if the process terminates:

`register(Name, Pid)`	Associates the name `Name`, an atom, with the process `Pid`.
`registered()`	Returns a list of names which have been registered using`register/2`.
`whereis(Name)`	Returns the pid registered under `Name`, or`undefined`if the name is not registered.

Table 11.1: Name Registration BIFs.

5.11.4 Process Termination

When a process terminates, it always terminates with an *exit reason*. The reason may be any term.

A process is said to terminate *normally*, if the exit reason is the atom `normal`. A process with no more code to execute terminates normally.

A process terminates with exit reason `{Reason,Stack}` when a run-time error occurs. See *Error and Error Handling*.

A process can terminate itself by calling one of the BIFs `exit(Reason)`, `erlang:error(Reason)`, `erlang:error(Reason, Args)`, `erlang:fault(Reason)` or `erlang:fault(Reason, Args)`. The process then terminates with reason `Reason` for `exit/1` or `{Reason,Stack} for the others`.

A process may also be terminated if it receives an exit signal with another exit reason than `normal`, see *Error Handling* below.

5.11.5 Message Sending

Processes communicate by sending and receiving messages. Messages are sent by using the *send operator !* and received by calling *receive*.

Message sending is asynchronous and safe, the message is guaranteed to eventually reach the recipient, provided that the recipient exists.

5.11.6 Links

Two processes can be *linked* to each other. A link between two processes `Pid1` and `Pid2` is created by `Pid1` calling the BIF `link(Pid2)` (or vice versa). There also exists a number a `spawn_link` BIFs, which spawns and links to a process in one operation.

Links are bidirectional and there can only be one link between two processes. Repeated calls to `link(Pid)` have no effect.

A link can be removed by calling the BIF `unlink(Pid)`.

Links are used to monitor the behaviour of other processes, see *Error Handling* below.

5.11.7 Error Handling

Erlang has a built-in feature for error handling between processes. Terminating processes will emit exit signals to all linked processes, which may terminate as well or handle the exit in some way. This feature can be used to build hierarchical program structures where some processes are supervising other processes, for example restarting them if they terminate abnormally.

Refer to OTP Design Principles for more information about OTP supervision trees, which uses this feature.

Emitting Exit Signals

When a process terminates, it will terminate with an *exit reason* as explained in *Process Termination* above. This exit reason is emitted in an *exit signal* to all linked processes.

A process can also call the function `exit(Pid,Reason)`. This will result in an exit signal with exit reason `Reason` being emitted to `Pid`, but does not affect the calling process.

Receiving Exit Signals

The default behaviour when a process receives an exit signal with an exit reason other than `normal`, is to terminate and in turn emit exit signals with the same exit reason to its linked processes. An exit signal with reason `normal` is ignored.

A process can be set to trap exit signals by calling:

```
process_flag(trap_exit, true)
```

When a process is trapping exits, it will not terminate when an exit signal is received. Instead, the signal is transformed into a message `{'EXIT',FromPid,Reason}` which is put into the mailbox of the process just like a regular message.

An exception to the above is if the exit reason is `kill`, that is if `exit(Pid,kill)` has been called. This will unconditionally terminate the process, regardless of if it is trapping exit signals or not.

5.11.8 Monitors

An alternative to links are *monitors*. A process `Pid1` can create a monitor for `Pid2` by calling the BIF `erlang:monitor(process, Pid2)`. The function returns a reference `Ref`.

If `Pid2` terminates with exit reason `Reason`, a 'DOWN' message is sent to `Pid1`:

```
{'DOWN', Ref, process, Pid2, Reason}
```

If `Pid2` does not exist, the 'DOWN' message is sent immediately with `Reason` set to `noproc`.

Monitors are unidirectional. Repeated calls to `erlang:monitor(process, Pid)` will create several, independent monitors and each one will send a 'DOWN' message when `Pid` terminates.

A monitor can be removed by calling `erlang:demonitor(Ref)`.

It is possible to create monitors for processes with registered names, also at other nodes.

5.11.9 Process Dictionary

Each process has its own process dictionary, accessed by calling the following BIFs:

```
put(Key, Value)
get(Key)
get()
get_keys(Value)
erase(Key)
erase()
```

5.12 Distributed Erlang

5.12.1 Distributed Erlang System

A *distributed Erlang system* consists of a number of Erlang runtime systems communicating with each other. Each such runtime system is called a *node*. Message passing between processes at different nodes, as well as links and monitors, are transparent when pids are used. Registered names, however, are local to each node. This means the node must be specified as well when sending messages etc. using registered names.

The distribution mechanism is implemented using TCP/IP sockets. How to implement an alternative carrier is described in *ERTS User's Guide*.

5.12.2 Nodes

A *node* is an executing Erlang runtime system which has been given a name, using the command line flag `-name` (long names) or `-sname` (short names).

The format of the node name is an atom `name@host` where `name` is the name given by the user and `host` is the full host name if long names are used, or the first part of the host name if short names are used. `node()` returns the name of the node. Example:

```
% erl -name dilbert
(dilbert@uab.ericsson.se)1> node().
'dilbert@uab.ericsson.se'
```

```
% erl -sname dilbert
(dilbert@uab)1> node().
dilbert@uab
```

> Note:
>
> A node with a long node name cannot communicate with a node with a short node name.

5.12.3 Node Connections

The nodes in a distributed Erlang system are loosely connected. The first time the name of another node is used, for example if `spawn(Node,M,F,A)` or `net_adm:ping(Node)` is called, a connection attempt to that node will be made.

Connections are by default transitive. If a node A connects to node B, and node B has a connection to node C, then node A will also try to connect to node C. This feature can be turned off by using the command line flag `-connect_all false`, see `erl(1)`.

If a node goes down, all connections to that node are removed. Calling `erlang:disconnect(Node)` will force disconnection of a node.

The list of (visible) nodes currently connected to is returned by `nodes()`.

5.12.4 epmd

The Erlang Port Mapper Daemon *epmd* is automatically started at every host where an Erlang node is started. It is responsible for mapping the symbolic node names to machine addresses. See `epmd(1)`.

5.12.5 Hidden Nodes

In a distributed Erlang system, it is sometimes useful to connect to a node without also connecting to all other nodes. An example could be some kind of O&M functionality used to inspect the status of a system without disturbing it. For this purpose, a *hidden node* may be used.

A hidden node is a node started with the command line flag `-hidden`. Connections between hidden nodes and other nodes are not transitive, they must be set up explicitly. Also, hidden nodes does not show up in the list of nodes returned by `nodes()`. Instead, `nodes(hidden)` or `nodes(connected)` must be used. This means, for example, that the hidden node will not be added to the set of nodes that `global` is keeping track of.

This feature was added in Erlang 5.0/OTP R7.

5.12.6 C Nodes

A *C node* is a C program written to act as a hidden node in a distributed Erlang system. The library *Erl_Interface* contains functions for this purpose. Refer to the documentation for Erl_Interface and *Interoperability Tutorial* for more information about C nodes.

5.12.7 Security

Authentication determines which nodes are allowed to communicate with each other. In a network of different Erlang nodes, it is built into the system at the lowest possible level. Each node has its own *magic cookie*, which is an Erlang atom.

When a nodes tries to connect to another node, the magic cookies are compared. If they do not match, the connected node rejects the connection.

At start-up, a node has a random atom assigned as its magic cookie and the cookie of other nodes is assumed to be `nocookie`. The first action of the Erlang network authentication server (`auth`) is then to read a file named `$HOME/.erlang.cookie`. If the file does not exist, it is created. The UNIX permissions mode of the file is set to octal 400 (read-only by user) and its contents are a random string. An atom `Cookie` is created from the contents of the file and the cookie of the local node is set to this using `erlang:set_cookie(node(), Cookie)`. This also makes the local node assume that all other nodes have the same cookie `Cookie`.

Thus, groups of users with identical cookie files get Erlang nodes which can communicate freely and without interference from the magic cookie system. Users who want run nodes on separate file systems must make certain that their cookie files are identical on the different file systems.

For a node `Node1` with magic cookie `Cookie` to be able to connect to, or accept a connection from, another node `Node2` with a different cookie `DiffCookie`, the function `erlang:set_cookie(Node2, DiffCookie)` must first be called at `Node1`. Distributed systems with multiple user IDs can be handled in this way.

The default when a connection is established between two nodes, is to immediately connect all other visible nodes as well. This way, there is always a fully connected network. If there are nodes with different cookies, this method might be inappropriate and the command line flag `-connect_all false` must be set, see *erl(1)*.

The magic cookie of the local node is retrieved by calling `erlang:get_cookie()`.

5.12.8 Distribution BIFs

Some useful BIFs for distributed programming, see `erlang(3)` for more information:

`erlang:disconnect_node(Node)`	Forces the disconnection of a node.
`erlang:get_cookie()`	Returns the magic cookie of the current node.
`is_alive()`	Returns `true` if the runtime system is a node and can connect to other nodes, `false` otherwise.
`monitor_node(Node, true|false)`	Monitor the status of `Node`. A message `{nodedown, Node}` is received if the connection to it is lost.
`node()`	Returns the name of the current node. Allowed in guards.
`node(Arg)`	Returns the node where `Arg`, a pid, reference, or port, is located.
`nodes()`	Returns a list of all visible nodes this node is connected to.
`nodes(Arg)`	Depending on `Arg`, this function can return a list not only of visible nodes, but also hidden nodes and previously known nodes, etc.
`set_cookie(Node, Cookie)`	Sets the magic cookie used when connecting to `Node`. If `Node` is the current node, `Cookie` will be used when connecting to all new nodes.
`spawn[_link|_opt](Node, Fun)`	Creates a process at a remote node.

`spawn[_link|opt](Node, Module, FunctionName, Args)`	Creates a process at a remote node.

Table 12.1: Distribution BIFs.

5.12.9 Distribution Command Line Flags

Examples of command line flags used for distributed programming, see `erl(1)` for more information:

`-connect_all false`	Only explicit connection set-ups will be used.
`-hidden`	Makes a node into a hidden node.
`-name Name`	Makes a runtime system into a node, using long node names.
`-setcookie Cookie`	Same as calling `erlang:set_cookie(node(), Cookie)`.
`-sname Name`	Makes a runtime system into a node, using short node names.

Table 12.2: Distribution Command Line Flags.

5.12.10 Distribution Modules

Examples of modules useful for distributed programming:

In Kernel:

`global`	A global name registration facility.
`global_group`	Grouping nodes to global name registration groups.
`net_adm`	Various Erlang net administration routines.
`net_kernel`	Erlang networking kernel.

Table 12.3: Kernel Modules Useful For Distribution.

In STDLIB:

`slave`	Start and control of slave nodes.

Table 12.4: STDLIB Modules Useful For Distribution.

5.13 Compilation and Code Loading

How code is compiled and loaded is not a language issue, but is system dependent. This chapter describes compilation and code loading in Erlang/OTP with pointers to relevant parts of the documentation.

5.13.1 Compilation

Erlang programs must be *compiled* to object code. The compiler can generate a new file which contains the object code. The current abstract machine which runs the object code is called BEAM, therefore the object files get the suffix `.beam`. The compiler can also generate a binary which can be loaded directly.

The compiler is located in the Kernel module `compile`, see `compile(3)`.

```
compile:file(Module)
compile:file(Module, Options)
```

The Erlang shell understands the command `c(Module)` which both compiles and loads `Module`.

There is also a module `make` which provides a set of functions similar to the UNIX type Make functions, see `make(3)`.

The compiler can also be accessed from the OS prompt, see `erl(1)`.

```
% erl -compile Module1...ModuleN
% erl -make
```

The `erlc` program provides an even better way to compile modules from the shell, see `erlc(1)`. It understands a number of flags that can be used to define macros, add search paths for include files, and more.

```
% erlc <flags> File1.erl...FileN.erl
```

5.13.2 Code Loading

The object code must be *loaded* into the Erlang runtime system. This is handled by the *code server*, see `code(3)`.

The code server loads code according to a code loading strategy which is either *interactive* (default) or *embedded*. In interactive mode, code are searched for in a *code path* and loaded when first referenced. In embedded mode, code is loaded at start-up according to a *boot script*. This is described in *System Principles*.

5.13.3 Code Replacement

Erlang supports change of code in a running system. Code replacement is done on module level.

The code of a module can exist in two variants in a system: *current* and *old*. When a module is loaded into the system for the first time, the code becomes 'current'. If then a new instance of the module is loaded, the code of the previous instance becomes 'old' and the new instance becomes 'current'.

Both old and current code is valid, and may be evaluated concurrently. Fully qualified function calls always refer to current code. Old code may still be evaluated because of processes lingering in the old code.

If a third instance of the module is loaded, the code server will remove (purge) the old code and any processes lingering in it will be terminated. Then the third instance becomes 'current' and the previously current code becomes 'old'.

To change from old code to current code, a process must make a fully qualified function call. Example:

```
-module(m).
-export([loop/0]).

loop() ->
    receive
        code_switch ->
            m:loop();
        Msg ->
            ...
            loop()
    end.
```

To make the process change code, send the message `code_switch` to it. The process then will make a fully qualified call to `m:loop()` and change to current code. Note that `m:loop/0` must be exported.

For code replacement of funs to work, the tuple syntax `{Module,FunctionName}` must be used to represent the fun.

5.13.4 Running a function when a module is loaded

> **Warning:**
>
> This section describes an experimental feature that was introduced in R13B03, and changed in a backwards-incompatible way in R13B04. There may be more backward-incompatible changes in future releases.

The `-on_load()` directive names a function that should be run automatically when a module a loaded. Its syntax is:

```
-on_load(Name/0).
```

It is not necessary to export the function. It will be called in a freshly spawned process (which will be terminated as soon as the function returns). The function must return `ok` if the module is to be remained loaded and become callable, or any other value if the module is to be unloaded. Generating an exception will also cause the module to be unloaded. If the return value is not an atom, a warning error report will be sent to the error logger.

A process that calls any function in a module whose `on_load` function has not yet returned will be suspended until the `on_load` function has returned.

In embedded mode, all modules will be loaded first and then will all on_load functions be called. The system will be terminated unless all of the on_load functions return `ok`

Example:

```
-module(m).
-on_load(load_my_nifs/0).

load_my_nifs() ->
    NifPath = ...,    %Set up the path to the NIF library.
    Info = ...,       %Initialize the Info term
    erlang:load_nif(NifPath, Info).
```

If the call to `erlang:load_nif/2` fails, the module will be unloaded and there will be warning report sent to the error loader.

5.14 Ports and Port Drivers

Examples of how to use ports and port drivers can be found in *Interoperability Tutorial*. The BIFs mentioned are as usual documented in `erlang(3)`.

5.14.1 Ports

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, including binaries.

The Erlang process which creates a port is said to be the *port owner*, or the *connected process* of the port. All communication to and from the port should go via the port owner. If the port owner terminates, so will the port (and the external program, if it is written correctly).

The external program resides in another OS process. By default, it should read from standard input (file descriptor 0) and write to standard output (file descriptor 1). The external program should terminate when the port is closed.

5.14.2 Port Drivers

It is also possible to write a driver in C according to certain principles and dynamically link it to the Erlang runtime system. The linked-in driver looks like a port from the Erlang programmer's point of view and is called a *port driver*.

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

Port drivers are documented in `erl_driver(4)`, `driver_entry(1)` and `erl_ddll(3)`.

5.14.3 Port BIFs

To create a port:

`open_port(PortName, PortSettings`	Returns a port identifier `Port`as the result of opening a new Erlang port. Messages can be sent to and received from a port identifier, just like a pid. Port identifiers can also be linked to or registered under a name using `link/1`and `register/2`.

Table 14.1: Port Creation BIF.

`PortName` is usually a tuple `{spawn,Command}`, where the string `Command` is the name of the external program. The external program runs outside the Erlang workspace unless a port driver with the name `Command` is found. If found, that driver is started.

`PortSettings` is a list of settings (options) for the port. The list typically contains at least a tuple `{packet,N}` which specifies that data sent between the port and the external program are preceded by an N-byte length indicator. Valid values for N are 1, 2 or 4. If binaries should be used instead of lists of bytes, the option `binary` must be included.

The port owner `Pid` can communicate with the port `Port` by sending and receiving messages. (In fact, any process can send the messages to the port, but the messages from the port always go to the port owner).

Below, `Data` must be an I/O list. An I/O list is a binary or a (possibly deep) list of binaries or integers in the range 0..255.

`{Pid,{command,Data}}`	Sends `Data` to the port.
`{Pid,close}`	Closes the port. Unless the port is already closed, the port replies with `{Port,closed}` when all buffers have been flushed and the port really closes.
`{Pid,{connect,NewPid}}`	Sets the port owner of `Port` to `NewPid`. Unless the port is already closed, the port replies with `{Port,connected}` to the old port owner. Note that the old port owner is still linked to the port, but the new port owner is not.

Table 14.2: Messages Sent To a Port.

`{Port,{data,Data}}`	`Data` is received from the external program.
`{Port,closed}`	Reply to `Port ! {Pid,close}`.
`{Port,connected}`	Reply to `Port ! {Pid,{connect,NewPid}}`
`{'EXIT',Port,Reason}`	If the port has terminated for some reason.

Table 14.3: Messages Received From a Port.

Instead of sending and receiving messages, there are also a number of BIFs that can be used. These can be called by any process, not only the port owner.

`port_command(Port,Data)`	Sends `Data` to the port.
`port_close(Port)`	Closes the port.
`port_connect(Port,NewPid)`	Sets the port owner of `Port` to `NewPid`. The old port owner `Pid` stays linked to the port and have to call `unlink(Port)` if this is not desired.
`erlang:port_info(Port,Item)`	Returns information as specified by `Item`.
`erlang:ports()`	Returns a list of all ports on the current node.

Table 14.4: Port BIFs.

There are some additional BIFs that only apply to port drivers: `port_control/3` and `erlang:port_call/3`.

6 User's Guide

This chapter contains examples on using records, funs, list comprehensions and the bit syntax.

6.1 Records

6.1.1 Records vs Tuples

The main advantage of using records instead of tuples is that fields in a record are accessed by name, whereas fields in a tuple are accessed by position. To illustrate these differences, suppose that we want to represent a person with the tuple `{Name, Address, Phone}`.

We must remember that the `Name` field is the first element of the tuple, the `Address` field is the second element, and so on, in order to write functions which manipulate this data. For example, to extract data from a variable `P` which contains such a tuple we might write the following code and then use pattern matching to extract the relevant fields.

```
Name = element(1, P),
Address = element(2, P),
...
```

Code like this is difficult to read and understand and errors occur if we get the numbering of the elements in the tuple wrong. If we change the data representation by re-ordering the fields, or by adding or removing a field, then all references to the person tuple, wherever they occur, must be checked and possibly modified.

Records allow us to refer to the fields by name and not position. We use a record instead of a tuple to store the data. If we write a record definition of the type shown below, we can then refer to the fields of the record by name.

```
-record(person, {name, phone, address}).
```

For example, if `P` is now a variable whose value is a `person` record, we can code as follows in order to access the name and address fields of the records.

```
Name = P#person.name,
Address = P#person.address,
...
```

Internally, records are represented using tagged tuples:

```
{person, Name, Phone, Address}
```

6.1.2 Defining a Record

This definition of a person will be used in many of the examples which follow. It contains three fields, `name`, `phone` and `address`. The default values for `name` and `phone` is "" and [], respectively. The default value for `address` is the atom `undefined`, since no default value is supplied for this field:

```
-record(person, {name = "", phone = [], address}).
```

We have to define the record in the shell in order to be able use the record syntax in the examples:

```
> rd(person, {name = "", phone = [], address}).
person
```

This is due to the fact that record definitions are available at compile time only, not at runtime. See `shell(3)` for details on records in the shell.

6.1.3 Creating a Record

A new `person` record is created as follows:

```
> #person{phone=[0,8,2,3,4,3,1,2], name="Robert"}.
#person{name = "Robert",phone = [0,8,2,3,4,3,1,2],address = undefined}
```

Since the `address` field was omitted, its default value is used.

There is a new feature introduced in Erlang 5.1/OTP R8B, with which you can set a value to all fields in a record, overriding the defaults in the record specification. The special field _, means "all fields not explicitly specified".

```
> #person{name = "Jakob", _ = '_'}.
#person{name = "Jakob",phone = '_',address = '_'}
```

It is primarily intended to be used in `ets:match/2` and `mnesia:match_object/3`, to set record fields to the atom `'_'`. (This is a wildcard in `ets:match/2`.)

6.1.4 Accessing a Record Field

```
> P = #person{name = "Joe", phone = [0,8,2,3,4,3,1,2]}.
#person{name = "Joe",phone = [0,8,2,3,4,3,1,2],address = undefined}
> P#person.name.
"Joe"
```

6.1.5 Updating a Record

```
> P1 = #person{name="Joe", phone=[1,2,3], address="A street"}.
#person{name = "Joe",phone = [1,2,3],address = "A street"}
> P2 = P1#person{name="Robert"}.
```

```
#person{name = "Robert",phone = [1,2,3],address = "A street"}
```

6.1.6 Type Testing

The following example shows that the guard succeeds if `P` is record of type `person`.

```
foo(P) when is_record(P, person) -> a_person;
foo(_) -> not_a_person.
```

6.1.7 Pattern Matching

Matching can be used in combination with records as shown in the following example:

```
> P3 = #person{name="Joe", phone=[0,0,7], address="A street"}.
#person{name = "Joe",phone = [0,0,7],address = "A street"}
> #person{name = Name} = P3, Name.
"Joe"
```

The following function takes a list of `person` records and searches for the phone number of a person with a particular name:

```
find_phone([#person{name=Name, phone=Phone} | _], Name) ->
    {found,  Phone};
find_phone([_| T], Name) ->
    find_phone(T, Name);
find_phone([], Name) ->
    not_found.
```

The fields referred to in the pattern can be given in any order.

6.1.8 Nested Records

The value of a field in a record might be an instance of a record. Retrieval of nested data can be done stepwise, or in a single step, as shown in the following example:

```
-record(name, {first = "Robert", last = "Ericsson"}).
-record(person, {name = #name{}, phone}).

demo() ->
  P = #person{name= #name{first="Robert",last="Virding"}, phone=123},
  First = (P#person.name)#name.first.
```

In this example, `demo()` evaluates to `"Robert"`.

6.1.9 Example

```
%% File: person.hrl

%%---------------------------------------------------------
```

```
%% Data Type: person
%% where:
%%    name:  A string (default is undefined).
%%    age:   An integer (default is undefined).
%%    phone: A list of integers (default is []).
%%    dict:  A dictionary containing various information
%%           about the person.
%%           A {Key, Value} list (default is the empty list).
%%----------------------------------------------------------
-record(person, {name, age, phone = [], dict = []}).
```

```
-module(person).
-include("person.hrl").
-compile(export_all). % For test purposes only.

%% This creates an instance of a person.
%%   Note: The phone number is not supplied so the
%%         default value [] will be used.

make_hacker_without_phone(Name, Age) ->
   #person{name = Name, age = Age,
           dict = [{computer_knowledge, excellent},
                   {drinks, coke}]}.

%% This demonstrates matching in arguments

print(#person{name = Name, age = Age,
              phone = Phone, dict = Dict}) ->
  io:format("Name: ~s, Age: ~w, Phone: ~w ~n"
            "Dictionary: ~w.~n", [Name, Age, Phone, Dict]).

%% Demonstrates type testing, selector, updating.

birthday(P) when record(P, person) ->
   P#person{age = P#person.age + 1}.

register_two_hackers() ->
   Hacker1 = make_hacker_without_phone("Joe", 29),
   OldHacker = birthday(Hacker1),
   % The central_register_server should have
   % an interface function for this.
   central_register_server ! {register_person, Hacker1},
   central_register_server ! {register_person,
             OldHacker#person{name = "Robert",
                              phone = [0,8,3,2,4,5,3,1]}}.
```

6.2 Funs

6.2.1 Example 1 - map

If we want to double every element in a list, we could write a function named `double`:

```
double([H|T]) -> [2*H|double(T)];
double([])    -> [].
```

This function obviously doubles the argument entered as input as follows:

```
> double([1,2,3,4]).
[2,4,6,8]
```

We now add the function `add_one`, which adds one to every element in a list:

```
add_one([H|T]) -> [H+1|add_one(T)];
add_one([])    -> [].
```

These functions, `double` and `add_one`, have a very similar structure. We can exploit this fact and write a function `map` which expresses this similarity:

```
map(F, [H|T]) -> [F(H)|map(F, T)];
map(F, [])    -> [].
```

We can now express the functions `double` and `add_one` in terms of `map` as follows:

```
double(L)  -> map(fun(X) -> 2*X end, L).
add_one(L) -> map(fun(X) -> 1 + X end, L).
```

`map(F, List)` is a function which takes a function `F` and a list `L` as arguments and returns the new list which is obtained by applying `F` to each of the elements in `L`.

The process of abstracting out the common features of a number of different programs is called procedural abstraction. Procedural abstraction can be used in order to write several different functions which have a similar structure, but differ only in some minor detail. This is done as follows:

- write one function which represents the common features of these functions
- parameterize the difference in terms of functions which are passed as arguments to the common function.

6.2.2 Example 2 - foreach

This example illustrates procedural abstraction. Initially, we show the following two examples written as conventional functions:

- all elements of a list are printed onto a stream
- a message is broadcast to a list of processes.

```
print_list(Stream, [H|T]) ->
    io:format(Stream, "~p~n", [H]),
    print_list(Stream, T);
print_list(Stream, []) ->
    true.
```

```
broadcast(Msg, [Pid|Pids]) ->
    Pid ! Msg,
    broadcast(Msg, Pids);
broadcast(_, []) ->
```

```
    true.
```

Both these functions have a very similar structure. They both iterate over a list doing something to each element in the list. The "something" has to be carried round as an extra argument to the function which does this.

The function `foreach` expresses this similarity:

```
foreach(F, [H|T]) ->
    F(H),
    foreach(F, T);
foreach(F, []) ->
    ok.
```

Using `foreach`, `print_list` becomes:

```
foreach(fun(H) -> io:format(S, "~p~n",[H]) end, L)
```

`broadcast` becomes:

```
foreach(fun(Pid) -> Pid ! M end, L)
```

`foreach` is evaluated for its side-effect and not its value. `foreach(Fun ,L)` calls `Fun(X)` for each element `X` in `L` and the processing occurs in the order in which the elements were defined in `L`. `map` does not define the order in which its elements are processed.

6.2.3 The Syntax of Funs

Funs are written with the syntax:

```
F = fun (Arg1, Arg2, ... ArgN) ->
        ...
    end
```

This creates an anonymous function of `N` arguments and binds it to the variable `F`.

If we have already written a function in the same module and wish to pass this function as an argument, we can use the following syntax:

```
F = fun FunctionName/Arity
```

With this form of function reference, the function which is referred to does not need to be exported from the module.

We can also refer to a function defined in a different module with the following syntax:

```
F = {Module, FunctionName}
```

In this case, the function must be exported from the module in question.

The follow program illustrates the different ways of creating funs:

```
-module(fun_test).
-export([t1/0, t2/0, t3/0, t4/0, double/1]).
-import(lists, [map/2]).

t1() -> map(fun(X) -> 2 * X end, [1,2,3,4,5]).

t2() -> map(fun double/1, [1,2,3,4,5]).

t3() -> map({?MODULE, double}, [1,2,3,4,5]).

double(X) -> X * 2.
```

We can evaluate the fun F with the syntax:

```
F(Arg1, Arg2, ..., Argn)
```

To check whether a term is a fun, use the test `is_function/1` in a guard. Example:

```
f(F, Args) when is_function(F) ->
   apply(F, Args);
f(N, _) when is_integer(N) ->
   N.
```

Funs are a distinct type. The BIFs erlang:fun_info/1,2 can be used to retrieve information about a fun, and the BIF erlang:fun_to_list/1 returns a textual representation of a fun. The check_process_code/2 BIF returns true if the process contains funs that depend on the old version of a module.

> **Note:**
>
> In OTP R5 and earlier releases, funs were represented using tuples.

6.2.4 Variable Bindings Within a Fun

The scope rules for variables which occur in funs are as follows:

- All variables which occur in the head of a fun are assumed to be "fresh" variables.
- Variables which are defined before the fun, and which occur in function calls or guard tests within the fun, have the values they had outside the fun.
- No variables may be exported from a fun.

The following examples illustrate these rules:

```
print_list(File, List) ->
    {ok, Stream} = file:open(File, write),
    foreach(fun(X) -> io:format(Stream,"~p~n",[X]) end, List),
    file:close(Stream).
```

In the above example, the variable X which is defined in the head of the fun is a new variable. The value of the variable Stream which is used within within the fun gets its value from the file:open line.

Since any variable which occurs in the head of a fun is considered a new variable it would be equally valid to write:

```
print_list(File, List) ->
    {ok, Stream} = file:open(File, write),
    foreach(fun(File) ->
                io:format(Stream,"~p~n",[File])
            end, List),
    file:close(Stream).
```

In this example, File is used as the new variable instead of X. This is rather silly since code in the body of the fun cannot refer to the variable File which is defined outside the fun. Compiling this example will yield the diagnostic:

```
./FileName.erl:Line: Warning: variable 'File'
      shadowed in 'lambda head'
```

This reminds us that the variable File which is defined inside the fun collides with the variable File which is defined outside the fun.

The rules for importing variables into a fun has the consequence that certain pattern matching operations have to be moved into guard expressions and cannot be written in the head of the fun. For example, we might write the following code if we intend the first clause of F to be evaluated when the value of its argument is Y:

```
f(...) ->
    Y = ...
    map(fun(X) when X == Y ->
               ;
           (_) ->
             ...
        end, ...)
    ...
```

instead of

```
f(...) ->
    Y = ...
    map(fun(Y) ->
               ;
           (_) ->
             ...
        end, ...)
    ...
```

6.2.5 Funs and the Module Lists

The following examples show a dialogue with the Erlang shell. All the higher order functions discussed are exported from the module lists.

map

```
map(F, [H|T]) -> [F(H)|map(F, T)];
map(F, [])    -> [].
```

`map` takes a function of one argument and a list of terms. It returns the list obtained by applying the function to every argument in the list.

```
> Double = fun(X) -> 2 * X end.
#Fun<erl_eval.6.72228031>
> lists:map(Double, [1,2,3,4,5]).
[2,4,6,8,10]
```

When a new fun is defined in the shell, the value of the Fun is printed as `Fun#<erl_eval>`.

any

```
any(Pred, [H|T]) ->
    case Pred(H) of
        true  ->  true;
        false ->  any(Pred, T)
    end;
any(Pred, []) ->
    false.
```

`any` takes a predicate `P` of one argument and a list of terms. A predicate is a function which returns `true` or `false`. `any` is true if there is a term `X` in the list such that `P(X)` is `true`.

We define a predicate `Big(X)` which is `true` if its argument is greater that 10.

```
> Big =  fun(X) -> if X > 10 -> true; true -> false end end.
#Fun<erl_eval.6.72228031>
> lists:any(Big, [1,2,3,4]).
false
> lists:any(Big, [1,2,3,12,5]).
true
```

all

```
all(Pred, [H|T]) ->
    case Pred(H) of
        true  ->  all(Pred, T);
        false ->  false
    end;
all(Pred, []) ->
    true.
```

`all` has the same arguments as `any`. It is true if the predicate applied to all elements in the list is true.

```
> lists:all(Big, [1,2,3,4,12,6]).
false
> lists:all(Big, [12,13,14,15]).
true
```

foreach

```
foreach(F, [H|T]) ->
    F(H),
    foreach(F, T);
foreach(F, []) ->
    ok.
```

`foreach` takes a function of one argument and a list of terms. The function is applied to each argument in the list. `foreach` returns `ok`. It is used for its side-effect only.

```
> lists:foreach(fun(X) -> io:format("~w~n",[X]) end, [1,2,3,4]).
1
2
3
4
ok
```

foldl

```
foldl(F, Accu, [Hd|Tail]) ->
    foldl(F, F(Hd, Accu), Tail);
foldl(F, Accu, []) -> Accu.
```

`foldl` takes a function of two arguments, an accumulator and a list. The function is called with two arguments. The first argument is the successive elements in the list, the second argument is the accumulator. The function must return a new accumulator which is used the next time the function is called.

If we have a list of lists `L = ["I","like","Erlang"]`, then we can sum the lengths of all the strings in `L` as follows:

```
> L = ["I","like","Erlang"].
["I","like","Erlang"]
10> lists:foldl(fun(X, Sum) -> length(X) + Sum end, 0, L).
11
```

`foldl` works like a `while` loop in an imperative language:

```
L =  ["I","like","Erlang"],
Sum = 0,
while( L != []){
    Sum += length(head(L)),
    L = tail(L)
```

```
end
```

mapfoldl

```
mapfoldl(F, Accu0, [Hd|Tail]) ->
    {R,Accu1} = F(Hd, Accu0),
    {Rs,Accu2} = mapfoldl(F, Accu1, Tail),
    {[R|Rs], Accu2};
mapfoldl(F, Accu, []) -> {[], Accu}.
```

`mapfoldl` simultaneously maps and folds over a list. The following example shows how to change all letters in `L` to upper case and count them.

First upcase:

```
> Upcase =  fun(X) when $a =< X,  X =< $z -> X + $A - $a;
(X) -> X
end.
#Fun<erl_eval.6.72228031>
> Upcase_word =
fun(X) ->
lists:map(Upcase, X)
end.
#Fun<erl_eval.6.72228031>
> Upcase_word("Erlang").
"ERLANG"
> lists:map(Upcase_word, L).
["I","LIKE","ERLANG"]
```

Now we can do the fold and the map at the same time:

```
> lists:mapfoldl(fun(Word, Sum) ->
{Upcase_word(Word), Sum + length(Word)}
end, 0, L).
{["I","LIKE","ERLANG"],11}
```

filter

```
filter(F, [H|T]) ->
    case F(H) of
        true  -> [H|filter(F, T)];
        false -> filter(F, T)
    end;
filter(F, []) -> [].
```

`filter` takes a predicate of one argument and a list and returns all element in the list which satisfy the predicate.

```
> lists:filter(Big, [500,12,2,45,6,7]).
[500,12,45]
```

When we combine maps and filters we can write very succinct code. For example, suppose we want to define a set difference function. We want to define `diff(L1, L2)` to be the difference between the lists `L1` and `L2`. This is the list of all elements in L1 which are not contained in L2. This code can be written as follows:

```
diff(L1, L2) ->
    filter(fun(X) -> not member(X, L2) end, L1).
```

The AND intersection of the list `L1` and `L2` is also easily defined:

```
intersection(L1,L2) -> filter(fun(X) -> member(X,L1) end, L2).
```

takewhile

```
takewhile(Pred, [H|T]) ->
    case Pred(H) of
        true  -> [H|takewhile(Pred, T)];
        false -> []
    end;
takewhile(Pred, []) ->
    [].
```

`takewhile(P, L)` takes elements `X` from a list `L` as long as the predicate `P(X)` is true.

```
> lists:takewhile(Big, [200,500,45,5,3,45,6]).
[200,500,45]
```

dropwhile

```
dropwhile(Pred, [H|T]) ->
    case Pred(H) of
        true  -> dropwhile(Pred, T);
        false -> [H|T]
    end;
dropwhile(Pred, []) ->
    [].
```

`dropwhile` is the complement of `takewhile`.

```
> lists:dropwhile(Big, [200,500,45,5,3,45,6]).
[5,3,45,6]
```

splitwith

```
```

```
splitwith(Pred, L) ->
    splitwith(Pred, L, []).

splitwith(Pred, [H|T], L) ->
    case Pred(H) of
        true  -> splitwith(Pred, T, [H|L]);
        false -> {reverse(L), [H|T]}
    end;
splitwith(Pred, [], L) ->
    {reverse(L), []}.
```

`splitwith(P, L)` splits the list `L` into the two sub-lists `{L1, L2}`, where `L = takewhile(P, L)` and `L2 = dropwhile(P, L)`.

```
> lists:splitwith(Big, [200,500,45,5,3,45,6]).
{[200,500,45],[5,3,45,6]}
```

6.2.6 Funs Which Return Funs

So far, this section has only described functions which take funs as arguments. It is also possible to write more powerful functions which themselves return funs. The following examples illustrate these type of functions.

Simple Higher Order Functions

`Adder(X)` is a function which, given `X`, returns a new function `G` such that `G(K)` returns `K + X`.

```
> Adder = fun(X) -> fun(Y) -> X + Y end end.
#Fun<erl_eval.6.72228031>
> Add6 = Adder(6).
#Fun<erl_eval.6.72228031>
> Add6(10).
16
```

Infinite Lists

The idea is to write something like:

```
-module(lazy).
-export([ints_from/1]).
ints_from(N) ->
    fun() ->
            [N|ints_from(N+1)]
    end.
```

Then we can proceed as follows:

```
> XX = lazy:ints_from(1).
#Fun<lazy.0.29874839>
> XX().
[1|#Fun<lazy.0.29874839>]
> hd(XX()).
1
> Y = tl(XX()).
#Fun<lazy.0.29874839>
```

```
> hd(Y()).
2
```

etc. - this is an example of "lazy embedding".

Parsing

The following examples show parsers of the following type:

```
Parser(Toks) -> {ok, Tree, Toks1} | fail
```

`Toks` is the list of tokens to be parsed. A successful parse returns `{ok, Tree, Toks1}`, where `Tree` is a parse tree and `Toks1` is a tail of `Tree` which contains symbols encountered after the structure which was correctly parsed. Otherwise `fail` is returned.

The example which follows illustrates a simple, functional parser which parses the grammar:

```
(a | b) & (c | d)
```

The following code defines a function `pconst(X)` in the module `funparse`, which returns a fun which parses a list of tokens.

```
pconst(X) ->
    fun (T) ->
       case T of
           [X|T1] -> {ok, {const, X}, T1};
           _      -> fail
       end
    end.
```

This function can be used as follows:

```
> P1 = funparse:pconst(a).
#Fun<funparse.0.22674075>
> P1([a,b,c]).
{ok,{const,a},[b,c]}
> P1([x,y,z]).
fail
```

Next, we define the two higher order functions `pand` and `por` which combine primitive parsers to produce more complex parsers. Firstly `pand`:

```
pand(P1, P2) ->
    fun (T) ->
        case P1(T) of
            {ok, R1, T1} ->
                case P2(T1) of
                    {ok, R2, T2} ->
                        {ok, {'and', R1, R2}};
```

```
                    fail ->
                        fail
                end;
            fail ->
                fail
        end
    end.
```

Given a parser `P1` for grammar `G1`, and a parser `P2` for grammar `G2`, `pand(P1, P2)` returns a parser for the grammar which consists of sequences of tokens which satisfy `G1` followed by sequences of tokens which satisfy `G2`.

`por(P1, P2)` returns a parser for the language described by the grammar `G1` or `G2`.

```
por(P1, P2) ->
    fun (T) ->
        case P1(T) of
            {ok, R, T1} ->
                {ok, {'or',1,R}, T1};
            fail ->
                case P2(T) of
                    {ok, R1, T1} ->
                        {ok, {'or',2,R1}, T1};
                    fail ->
                        fail
                end
        end
    end.
```

The original problem was to parse the grammar `(a | b) & (c | d)`. The following code addresses this problem:

```
grammar() ->
    pand(
         por(pconst(a), pconst(b)),
         por(pconst(c), pconst(d))).
```

The following code adds a parser interface to the grammar:

```
parse(List) ->
    (grammar())(List).
```

We can test this parser as follows:

```
> funparse:parse([a,c]).
{ok,{'and',{'or',1,{const,a}},{'or',1,{const,c}}}}
> funparse:parse([a,d]).
{ok,{'and',{'or',1,{const,a}},{'or',2,{const,d}}}}
> funparse:parse([b,c]).
{ok,{'and',{'or',2,{const,b}},{'or',1,{const,c}}}}
> funparse:parse([b,d]).
{ok,{'and',{'or',2,{const,b}},{'or',2,{const,d}}}}
> funparse:parse([a,b]).
```

```
fail
```

6.3 List Comprehensions

6.3.1 Simple Examples

We start with a simple example:

```
> [X || X <- [1,2,a,3,4,b,5,6], X > 3].
[a,4,b,5,6]
```

This should be read as follows:

The list of X such that X is taken from the list `[1,2,a,...]` and X is greater than 3.

The notation `X <- [1,2,a,...]` is a generator and the expression `X > 3` is a filter.

An additional filter can be added in order to restrict the result to integers:

```
> [X || X <- [1,2,a,3,4,b,5,6], integer(X), X > 3].
[4,5,6]
```

Generators can be combined. For example, the Cartesian product of two lists can be written as follows:

```
> [{X, Y} || X <- [1,2,3], Y <- [a,b]].
[{1,a},{1,b},{2,a},{2,b},{3,a},{3,b}]
```

6.3.2 Quick Sort

The well known quick sort routine can be written as follows:

```
sort([Pivot|T]) ->
    sort([ X || X <- T, X < Pivot]) ++
    [Pivot] ++
    sort([ X || X <- T, X >= Pivot]);
sort([]) -> [].
```

The expression `[X || X <- T, X < Pivot]` is the list of all elements in `T`, which are less than `Pivot`.

`[X || X <- T, X >= Pivot]` is the list of all elements in `T`, which are greater or equal to `Pivot`.

To sort a list, we isolate the first element in the list and split the list into two sub-lists. The first sub-list contains all elements which are smaller than the first element in the list, the second contains all elements which are greater than or equal to the first element in the list. We then sort the sub-lists and combine the results.

6.3.3 Permutations

The following example generates all permutations of the elements in a list:

```
perms([]) -> [[]];
```

```
perms(L)  -> [[H|T] || H <- L, T <- perms(L--[H])].
```

We take take `H` from `L` in all possible ways. The result is the set of all lists `[H|T]`, where `T` is the set of all possible permutations of `L` with `H` removed.

```
> perms([b,u,g]).
[[b,u,g],[b,g,u],[u,b,g],[u,g,b],[g,b,u],[g,u,b]]
```

6.3.4 Pythagorean Triplets

Pythagorean triplets are sets of integers `{A,B,C}` such that `A**2 + B**2 = C**2`.

The function `pyth(N)` generates a list of all integers `{A,B,C}` such that `A**2 + B**2 = C**2` and where the sum of the sides is equal to or less than `N`.

```
pyth(N) ->
    [ {A,B,C} ||
        A <- lists:seq(1,N),
        B <- lists:seq(1,N),
        C <- lists:seq(1,N),
        A+B+C =< N,
        A*A+B*B == C*C
    ].
```

```
> pyth(3).
[].
> pyth(11).
[].
> pyth(12).
[{3,4,5},{4,3,5}]
> pyth(50).
[{3,4,5},
 {4,3,5},
 {5,12,13},
 {6,8,10},
 {8,6,10},
 {8,15,17},
 {9,12,15},
 {12,5,13},
 {12,9,15},
 {12,16,20},
 {15,8,17},
 {16,12,20}]
```

The following code reduces the search space and is more efficient:

```
pyth1(N) ->
   [{A,B,C} ||
       A <- lists:seq(1,N-2),
       B <- lists:seq(A+1,N-1),
       C <- lists:seq(B+1,N),
       A+B+C =< N,
       A*A+B*B == C*C ].
```

6.3.5 Simplifications with List Comprehensions

As an example, list comprehensions can be used to simplify some of the functions in `lists.erl`:

```
append(L)   ->  [X || L1 <- L, X <- L1].
map(Fun, L) -> [Fun(X) || X <- L].
filter(Pred, L) -> [X || X <- L, Pred(X)].
```

6.3.6 Variable Bindings in List Comprehensions

The scope rules for variables which occur in list comprehensions are as follows:

- all variables which occur in a generator pattern are assumed to be "fresh" variables
- any variables which are defined before the list comprehension and which are used in filters have the values they had before the list comprehension
- no variables may be exported from a list comprehension.

As an example of these rules, suppose we want to write the function `select`, which selects certain elements from a list of tuples. We might write `select(X, L) -> [Y || {X, Y} <- L].` with the intention of extracting all tuples from `L` where the first item is `X`.

Compiling this yields the following diagnostic:

```
./FileName.erl:Line: Warning: variable 'X' shadowed in generate
```

This diagnostic warns us that the variable `X` in the pattern is not the same variable as the variable `X` which occurs in the function head.

Evaluating `select` yields the following result:

```
> select(b,[{a,1},{b,2},{c,3},{b,7}]).
[1,2,3,7]
```

This result is not what we wanted. To achieve the desired effect we must write `select` as follows:

```
select(X, L) ->  [Y || {X1, Y} <- L, X == X1].
```

The generator now contains unbound variables and the test has been moved into the filter. This now works as expected:

```
> select(b,[{a,1},{b,2},{c,3},{b,7}]).
[2,7]
```

One consequence of the rules for importing variables into a list comprehensions is that certain pattern matching operations have to be moved into the filters and cannot be written directly in the generators. To illustrate this, do not write as follows:

```
f(...) ->
    Y = ...
```

```
    [ Expression || PatternInvolving Y  <- Expr, ...]
    ...
```

Instead, write as follows:

```
f(...) ->
    Y = ...
    [ Expression || PatternInvolving Y1  <- Expr, Y == Y1, ...]
    ...
```

6.4 Bit Syntax

6.4.1 Introduction

In Erlang a Bin is used for constructing binaries and matching binary patterns. A Bin is written with the following syntax:

```
     <<E1, E2, ... En>>
```

A Bin is a low-level sequence of bits or bytes. The purpose of a Bin is to be able to, from a high level, construct a binary,

```
Bin = <<E1, E2, ... En>>
```

in which case all elements must be bound, or to match a binary,

```
<<E1, E2, ... En>> = Bin
```

where `Bin` is bound, and where the elements are bound or unbound, as in any match.

In R12B, a Bin need not consist of a whole number of bytes.

A *bitstring* is a sequence of zero or more bits, where the number of bits doesn't need to be divisible by 8. If the number of bits is divisible by 8, the bitstring is also a binary.

Each element specifies a certain *segment* of the bitstring. A segment is a set of contiguous bits of the binary (not necessarily on a byte boundary). The first element specifies the initial segment, the second element specifies the following segment etc.

The following examples illustrate how binaries are constructed or matched, and how elements and tails are specified.

Examples

Example 1: A binary can be constructed from a set of constants or a string literal:

```
Bin11 = <<1, 17, 42>>,
Bin12 = <<"abc">>
```

yields binaries of size 3; `binary_to_list(Bin11)` evaluates to `[1, 17, 42]`, and `binary_to_list(Bin12)` evaluates to `[97, 98, 99]`.

Example 2: Similarly, a binary can be constructed from a set of bound variables:

```
A = 1, B = 17, C = 42,
Bin2 = <<A, B, C:16>>
```

yields a binary of size 4, and `binary_to_list(Bin2)` evaluates to `[1, 17, 00, 42]` too. Here we used a *size expression* for the variable `C` in order to specify a 16-bits segment of `Bin2`.

Example 3: A Bin can also be used for matching: if `D`, `E`, and `F` are unbound variables, and `Bin2` is bound as in the former example,

```
<<D:16, E, F/binary>> = Bin2
```

yields `D = 273`, `E = 00`, and F binds to a binary of size 1: `binary_to_list(F) = [42]`.

Example 4: The following is a more elaborate example of matching, where `Dgram` is bound to the consecutive bytes of an IP datagram of IP protocol version 4, and where we want to extract the header and the data of the datagram:

```
-define(IP_VERSION, 4).
-define(IP_MIN_HDR_LEN, 5).

DgramSize = byte_size(Dgram),
case Dgram of
    <<?IP_VERSION:4, HLen:4, SrvcType:8, TotLen:16,
      ID:16, Flgs:3, FragOff:13,
      TTL:8, Proto:8, HdrChkSum:16,
      SrcIP:32,
      DestIP:32, RestDgram/binary>> when HLen>=5, 4*HLen=<DgramSize ->
        OptsLen = 4*(HLen - ?IP_MIN_HDR_LEN),
        <<Opts:OptsLen/binary,Data/binary>> = RestDgram,
    ...
end.
```

Here the segment corresponding to the `Opts` variable has a *type modifier* specifying that `Opts` should bind to a binary. All other variables have the default type equal to unsigned integer.

An IP datagram header is of variable length, and its length - measured in the number of 32-bit words - is given in the segment corresponding to `HLen`, the minimum value of which is 5. It is the segment corresponding to `Opts` that is variable: if `HLen` is equal to 5, `Opts` will be an empty binary.

The tail variables `RestDgram` and `Data` bind to binaries, as all tail variables do. Both may bind to empty binaries.

If the first 4-bits segment of `Dgram` is not equal to 4, or if `HLen` is less than 5, or if the size of `Dgram` is less than `4*HLen`, the match of `Dgram` fails.

6.4.2 A Lexical Note

Note that "`B=<<1>>`" will be interpreted as "`B =< <1>>`", which is a syntax error. The correct way to write the expression is "`B = <<1>>`".

6.4.3 Segments

Each segment has the following general syntax:

`Value:Size/TypeSpecifierList`

Both the `Size` and the `TypeSpecifier` or both may be omitted; thus the following variations are allowed:

`Value`

`Value:Size`

`Value/TypeSpecifierList`

Default values will be used for missing specifications. The default values are described in the section *Defaults*.

Used in binary construction, the `Value` part is any expression. Used in binary matching, the `Value` part must be a literal or variable. You can read more about the `Value` part in the section about constructing binaries and matching binaries.

The `Size` part of the segment multiplied by the unit in the `TypeSpecifierList` (described below) gives the number of bits for the segment. In construction, `Size` is any expression that evaluates to an integer. In matching, `Size` must be a constant expression or a variable.

The `TypeSpecifierList` is a list of type specifiers separated by hyphens.

Type
: The type can be `integer`, `float`, or `binary`.

Signedness
: The signedness specification can be either `signed` or `unsigned`. Note that signedness only matters for matching.

Endianness
: The endianness specification can be either `big`, `little`, or `native`. Native-endian means that the endian will be resolved at load time to be either big-endian or little-endian, depending on what is "native" for the CPU that the Erlang machine is run on.

Unit
: The unit size is given as `unit:IntegerLiteral`. The allowed range is 1-256. It will be multiplied by the `Size` specifier to give the effective size of the segment. In R12B, the unit size specifies the alignment for binary segments without size (examples will follow).

Example:

```
X:4/little-signed-integer-unit:8
```

This element has a total size of 4*8 = 32 bits, and it contains a signed integer in little-endian order.

6.4.4 Defaults

The default type for a segment is integer. The default type does not depend on the value, even if the value is a literal. For instance, the default type in '`<<3.14>>`' is integer, not float.

The default `Size` depends on the type. For integer it is 8. For float it is 64. For binary it is all of the binary. In matching, this default value is only valid for the very last element. All other binary elements in matching must have a size specification.

The default unit depends on the the type. For `integer`, `float`, and `bitstring` it is 1. For binary it is 8.

The default signedness is `unsigned`.

The default endianness is `big`.

6.4.5 Constructing Binaries and Bitstrings

This section describes the rules for constructing binaries using the bit syntax. Unlike when constructing lists or tuples, the construction of a binary can fail with a `badarg` exception.

There can be zero or more segments in a binary to be constructed. The expression '<<>>' constructs a zero length binary.

Each segment in a binary can consist of zero or more bits. There are no alignment rules for individual segments of type `integer` and `float`. For binaries and bitstrings without size, the unit specifies the alignment. Since the default alignment for the `binary` type is 8, the size of a binary segment must be a multiple of 8 bits (i.e. only whole bytes). Example:

```
<<Bin/binary,Bitstring/bitstring>>
```

The variable `Bin` must contain a whole number of bytes, because the `binary` type defaults to `unit:8`. A `badarg` exception will be generated if `Bin` would consist of (for instance) 17 bits.

On the other hand, the variable `Bitstring` may consist of any number of bits, for instance 0, 1, 8, 11, 17, 42, and so on, because the default `unit` for bitstrings is 1.

> ### Warning:
>
> For clarity, it is recommended not to change the unit size for binaries, but to use `binary` when you need byte alignment, and `bitstring` when you need bit alignment.

The following example

```
<<X:1,Y:6>>
```

will successfully construct a bitstring of 7 bits. (Provided that all of X and Y are integers.)

As noted earlier, segments have the following general syntax:

`Value:Size/TypeSpecifierList`

When constructing binaries, `Value` and `Size` can be any Erlang expression. However, for syntactical reasons, both `Value` and `Size` must be enclosed in parenthesis if the expression consists of anything more than a single literal or variable. The following gives a compiler syntax error:

```
<<X+1:8>>
```

This expression must be rewritten to

```
<<(X+1):8>>
```

in order to be accepted by the compiler.

Including Literal Strings

As syntactic sugar, an literal string may be written instead of a element.

```
```

```
<<"hello">>
```

which is syntactic sugar for

```
<<$h,$e,$l,$l,$o>>
```

6.4.6 Matching Binaries

This section describes the rules for matching binaries using the bit syntax.

There can be zero or more segments in a binary pattern. A binary pattern can occur in every place patterns are allowed, also inside other patterns. Binary patterns cannot be nested.

The pattern '`<<>>`' matches a zero length binary.

Each segment in a binary can consist of zero or more bits.

A segment of type `binary` must have a size evenly divisible by 8 (or divisible by the unit size, if the unit size has been changed).

A segment of type `bitstring` has no restrictions on the size.

As noted earlier, segments have the following general syntax:

`Value:Size/TypeSpecifierList`

When matching `Value` value must be either a variable or an integer or floating point literal. Expressions are not allowed.

`Size` must be an integer literal, or a previously bound variable. Note that the following is not allowed:

```
foo(N, <<X:N,T/binary>>) ->
   {X,T}.
```

The two occurrences of `N` are not related. The compiler will complain that the `N` in the size field is unbound.

The correct way to write this example is like this:

```
foo(N, Bin) ->
   <<X:N,T/binary>> = Bin,
   {X,T}.
```

Getting the Rest of the Binary or Bitstring

To match out the rest of a binary, specify a binary field without size:

```
foo(<<A:8,Rest/binary>>) ->
```

The size of the tail must be evenly divisible by 8.

To match out the rest of a bitstring, specify a field without size:

```
foo(<<A:8,Rest/bitstring>>) ->
```

There is no restriction on the number of bits in the tail.

6.4.7 Appending to a Binary

In R12B, the following function for creating a binary out of a list of triples of integers is now efficient:

```
triples_to_bin(T) ->
    triples_to_bin(T, <<>>).

triples_to_bin([{X,Y,Z} | T], Acc) ->
    triples_to_bin(T, <<Acc/binary,X:32,Y:32,Z:32>>);   % inefficient before R12B
triples_to_bin([], Acc) ->
    Acc.
```

In previous releases, this function was highly inefficient, because the binary constructed so far (`Acc`) was copied in each recursion step. That is no longer the case. See the Efficiency Guide for more information.

7 User's Guide

7.1 Introduction

7.1.1 Purpose

Premature optimization is the root of all evil. -- D.E. Knuth

Efficient code can be well-structured and clean code, based on on a sound overall architecture and sound algorithms. Efficient code can be highly implementation-code that bypasses documented interfaces and takes advantage of obscure quirks in the current implementation.

Ideally, your code should only contain the first kind of efficient code. If that turns out to be too slow, you should profile the application to find out where the performance bottlenecks are and optimize only the bottlenecks. Other code should stay as clean as possible.

Fortunately, compiler and run-time optimizations introduced in R12B makes it easier to write code that is both clean and efficient. For instance, the ugly workarounds needed in R11B and earlier releases to get the most speed out of binary pattern matching are no longer necessary. In fact, the ugly code is slower than the clean code (because the clean code has become faster, not because the uglier code has become slower).

This Efficiency Guide cannot really learn you how to write efficient code. It can give you a few pointers about what to avoid and what to use, and some understanding of how certain language features are implemented. We have generally not included general tips about optimization that will work in any language, such as moving common calculations out of loops.

7.1.2 Prerequisites

It is assumed that the reader is familiar with the Erlang programming language and concepts of OTP.

7.2 The Eight Myths of Erlang Performance

Some truths seem to live on well beyond their best-before date, perhaps because "information" spreads more rapidly from person-to-person faster than a single release note that notes, for instance, that funs have become faster.

Here we try to kill the old truths (or semi-truths) that have become myths.

7.2.1 Myth: Funs are slow

Yes, funs used to be slow. Very slow. Slower than `apply/3`. Originally, funs were implemented using nothing more than compiler trickery, ordinary tuples, `apply/3`, and a great deal of ingenuity.

But that is ancient history. Funs was given its own data type in the R6B release and was further optimized in the R7B release. Now the cost for a fun call falls roughly between the cost for a call to local function and `apply/3`.

7.2.2 Myth: List comprehensions are slow

List comprehensions used to be implemented using funs, and in the bad old days funs were really slow.

Nowadays the compiler rewrites list comprehensions into an ordinary recursive function. Of course, using a tail-recursive function with a reverse at the end would be still faster. Or would it? That leads us to the next myth.

7.2.3 Myth: Tail-recursive functions are MUCH faster than recursive functions

According to the myth, recursive functions leave references to dead terms on the stack and the garbage collector will have to copy all those dead terms, while tail-recursive functions immediately discard those terms.

That used to be true before R7B. In R7B, the compiler started to generate code that overwrites references to terms that will never be used with an empty list, so that the garbage collector would not keep dead values any longer than necessary.

Even after that optimization, a tail-recursive function would still most of the time be faster than a body-recursive function. Why?

It has to do with how many words of stack that are used in each recursive call. In most cases, a recursive function would use more words on the stack for each recursion than the number of words a tail-recursive would allocate on the heap. Since more memory is used, the garbage collector will be invoked more frequently, and it will have more work traversing the stack.

In R12B and later releases, there is an optimization that will in many cases reduces the number of words used on the stack in body-recursive calls, so that a body-recursive list function and tail-recursive function that calls *lists:reverse/1* at the end will use exactly the same amount of memory. `lists:map/2`, `lists:filter/2`, list comprehensions, and many other recursive functions now use the same amount of space as their tail-recursive equivalents.

So which is faster?

It depends. On Solaris/Sparc, the body-recursive function seems to be slightly faster, even for lists with very many elements. On the x86 architecture, tail-recursion was up to about 30 percent faster.

So the choice is now mostly a matter of taste. If you really do need the utmost speed, you must *measure*. You can no longer be absolutely sure that the tail-recursive list function will be the fastest in all circumstances.

Note: A tail-recursive function that does not need to reverse the list at the end is, of course, faster than a body-recursive function, as are tail-recursive functions that do not construct any terms at all (for instance, a function that sums all integers in a list).

7.2.4 Myth: '++' is always bad

The ++ operator has, somewhat undeservedly, got a very bad reputation. It probably has something to do with code like

DO NOT

```
naive_reverse([H|T]) ->
    naive_reverse(T)++[H];
naive_reverse([]) ->
    [].
```

which is the most inefficient way there is to reverse a list. Since the ++ operator copies its left operand, the result will be copied again and again and again... leading to quadratic complexity.

On the other hand, using ++ like this

OK

```
naive_but_ok_reverse([H|T], Acc) ->
    naive_but_ok_reverse(T, [H]++Acc);
naive_but_ok_reverse([], Acc) ->
    Acc.
```

is not bad. Each list element will only be copied once. The growing result `Acc` is the right operand for the `++` operator, and it will *not* be copied.

Of course, experienced Erlang programmers would actually write

DO

```
vanilla_reverse([H|T], Acc) ->
    vanilla_reverse(T, [H|Acc]);
vanilla_reverse([], Acc) ->
    Acc.
```

which is slightly more efficient because you don't build a list element only to directly copy it. (Or it would be more efficient if the the compiler did not automatically rewrite `[H]++Acc` to `[H|Acc]`.)

7.2.5 Myth: Strings are slow

Actually, string handling could be slow if done improperly. In Erlang, you'll have to think a little more about how the strings are used and choose an appropriate representation and use the *re* instead of the obsolete `regexp` module if you are going to use regualr expressions.

7.2.6 Myth: Repairing a Dets file is very slow

The repair time is still proportional to the number of records in the file, but Dets repairs used to be much, much slower in the past. Dets has been massively rewritten and improved.

7.2.7 Myth: BEAM is a stack-based byte-code virtual machine (and therefore slow)

BEAM is a register-based virtual machine. It has 1024 virtual registers that are used for holding temporary values and for passing arguments when calling functions. Variables that need to survive a function call are saved to the stack.

BEAM is a threaded-code interpreter. Each instruction is word pointing directly to executable C-code, making instruction dispatching very fast.

7.2.8 Myth: Use '_' to speed up your program when a variable is not used

That was once true, but since R6B the BEAM compiler is quite capable of seeing itself that a variable is not used.

7.3 Common Caveats

Here we list a few modules and BIFs to watch out for, and not only from a performance point of view.

7.3.1 The regexp module

The regular expression functions in the *regexp* module are written in Erlang, not in C, and were meant for occasional use on small amounts of data, for instance for validation of configuration files when starting an application.

Use the *re* module (introduced in R13A) instead, especially in time-critical code.

7.3.2 The timer module

Creating timers using *erlang:send_after/3* and *erlang:start_timer/3* is much more efficient than using the timers provided by the *timer* module. The `timer` module uses a separate process to manage the timers, and that process

can easily become overloaded if many processes create and cancel timers frequently (especially when using the SMP emulator).

The functions in the `timer` module that do not manage timers (such as `timer:tc/3` or `timer:sleep/1`), do not call the timer-server process and are therefore harmless.

7.3.3 list_to_atom/1

Atoms are not garbage-collected. Once an atom is created, it will never be removed. The emulator will terminate if the limit for the number of atoms (1048576) is reached.

Therefore, converting arbitrary input strings to atoms could be dangerous in a system that will run continuously. If only certain well-defined atoms are allowed as input, you can use *list_to_existing_atom/1* to guard against a denial-of-service attack. (All atoms that are allowed must have been created earlier, for instance by simply using all of them in a module and loading that module.)

Using `list_to_atom/1` to construct an atom that is passed to `apply/3` like this

```
apply(list_to_atom("some_prefix"++Var), foo, Args)
```

is quite expensive and is not recommended in time-critical code.

7.3.4 length/1

The time for calculating the length of a list is proportional to the length of the list, as opposed to `tuple_size/1`, `byte_size/1`, and `bit_size/1`, which all execute in constant time.

Normally you don't have to worry about the speed of `length/1`, because it is efficiently implemented in C. In time critical-code, though, you might want to avoid it if the input list could potentially be very long.

Some uses of `length/1` can be replaced by matching. For instance, this code

```
foo(L) when length(L) >= 3 ->
    ...
```

can be rewritten to

```
foo([_,_,_|_]=L) ->
   ...
```

(One slight difference is that `length(L)` will fail if the `L` is an improper list, while the pattern in the second code fragment will accept an improper list.)

7.3.5 setelement/3

setelement/3 copies the tuple it modifies. Therefore, updating a tuple in a loop using `setelement/3` will create a new copy of the tuple every time.

There is one exception to the rule that the tuple is copied. If the compiler clearly can see that destructively updating the tuple would give exactly the same result as if the tuple was copied, the call to `setelement/3` will be replaced with a special destructive setelement instruction. In the following code sequence

```
```

```
multiple_setelement(T0) ->
    T1 = setelement(9, T0, bar),
    T2 = setelement(7, T1, foobar),
    setelement(5, T2, new_value).
```

the first `setelement/3` call will copy the tuple and modify the ninth element. The two following `setelement/3` calls will modify the tuple in place.

For the optimization to be applied, *all* of the followings conditions must be true:

- The indices must be integer literals, not variables or expressions.
- The indices must be given in descending order.
- There must be no calls to other function in between the calls to `setelement/3`.
- The tuple returned from one `setelement/3` call must only be used in the subsequent call to `setelement/3`.

If it is not possible to structure the code as in the `multiple_setelement/1` example, the best way to modify multiple elements in a large tuple is to convert the tuple to a list, modify the list, and convert the list back to a tuple.

7.3.6 size/1

`size/1` returns the size for both tuples and binary.

Using the new BIFs `tuple_size/1` and `byte_size/1` introduced in R12B gives the compiler and run-time system more opportunities for optimization. A further advantage is that the new BIFs could help Dialyzer find more bugs in your program.

7.3.7 split_binary/2

It is usually more efficient to split a binary using matching instead of calling the `split_binary/2` function. Furthermore, mixing bit syntax matching and `split_binary/2` may prevent some optimizations of bit syntax matching.

DO

```
      <<Bin1:Num/binary,Bin2/binary>> = Bin,
```

DO NOT

```
      {Bin1,Bin2} = split_binary(Bin, Num)
```

7.3.8 The '--' operator

Note that the `--` operator has a complexity proportional to the product of the length of its operands, meaning that it will be very slow if both of its operands are long lists:

DO NOT

```
      HugeList1 -- HugeList2
```

Instead use the *ordsets* module:

DO

```
HugeSet1 = ordsets:from_list(HugeList1),
HugeSet2 = ordsets:from_list(HugeList2),
ordsets:subtract(HugeSet1, HugeSet2)
```

Obviously, that code will not work if the original order of the list is important. If the order of the list must be preserved, do like this:

DO

```
Set = gb_sets:from_list(HugeList2),
[E || E <- HugeList1, not gb_sets:is_element(E, Set)]
```

Subtle note 1: This code behaves differently from '`--`' if the lists contain duplicate elements. (One occurrence of an element in HugeList2 will remove *all* occurrences in HugeList1.)

Subtle note 2: This code compares lists elements using the '`==`' operator, while '`--`' uses the '`=:=`'. If that difference is important, `sets` can be used instead of `gb_sets`, but note that `sets:from_list/1` is much slower than `gb_sets:from_list/1` for long lists.

Using the '`--`' operator to delete an element from a list is not a performance problem:

OK

```
HugeList1 -- [Element]
```

7.4 Constructing and matching binaries

In R12B, the most natural way to write binary construction and matching is now significantly faster than in earlier releases.

To construct at binary, you can simply write

DO (in R12B) / *REALLY DO NOT* (in earlier releases)

```
my_list_to_binary(List) ->
    my_list_to_binary(List, <<>>).

my_list_to_binary([H|T], Acc) ->
    my_list_to_binary(T, <<Acc/binary,H>>);
my_list_to_binary([], Acc) ->
    Acc.
```

In releases before R12B, `Acc` would be copied in every iteration. In R12B, `Acc` will be copied only in the first iteration and extra space will be allocated at the end of the copied binary. In the next iteration, `H` will be written in to the extra space. When the extra space runs out, the binary will be reallocated with more extra space.

The extra space allocated (or reallocated) will be twice the size of the existing binary data, or 256, whichever is larger.

The most natural way to match binaries is now the fastest:

DO (in R12B)

```
my_binary_to_list(<<H,T/binary>>) ->
    [H|my_binary_to_list(T)];
my_binary_to_list(<<>>) -> [].
```

7.4.1 How binaries are implemented

Internally, binaries and bitstrings are implemented in the same way. In this section, we will call them *binaries* since that is what they are called in the emulator source code.

There are four types of binary objects internally. Two of them are containers for binary data and two of them are merely references to a part of a binary.

The binary containers are called *refc binaries* (short for *reference-counted binaries*) and *heap binaries*.

Refc binaries consist of two parts: an object stored on the process heap, called a *ProcBin*, and the binary object itself stored outside all process heaps.

The binary object can be referenced by any number of ProcBins from any number of processes; the object contains a reference counter to keep track of the number of references, so that it can be removed when the last reference disappears.

All ProcBin objects in a process are part of a linked list, so that the garbage collector can keep track of them and decrement the reference counters in the binary when a ProcBin disappears.

Heap binaries are small binaries, up to 64 bytes, that are stored directly on the process heap. They will be copied when the process is garbage collected and when they are sent as a message. They don't require any special handling by the garbage collector.

There are two types of reference objects that can reference part of a refc binary or heap binary. They are called *sub binaries* and *match contexts*.

A *sub binary* is created by `split_binary/2` and when a binary is matched out in a binary pattern. A sub binary is a reference into a part of another binary (refc or heap binary, never into a another sub binary). Therefore, matching out a binary is relatively cheap because the actual binary data is never copied.

A *match context* is similar to a sub binary, but is optimized for binary matching; for instance, it contains a direct pointer to the binary data. For each field that is matched out of a binary, the position in the match context will be incremented.

In R11B, a match context was only using during a binary matching operation.

In R12B, the compiler tries to avoid generating code that creates a sub binary, only to shortly afterwards create a new match context and discard the sub binary. Instead of creating a sub binary, the match context is kept.

The compiler can only do this optimization if it can know for sure that the match context will not be shared. If it would be shared, the functional properties (also called referential transparency) of Erlang would break.

7.4.2 Constructing binaries

In R12B, appending to a binary or bitstring

```
<<Binary/binary, ...>>
<<Binary/bitstring, ...>>
```

is specially optimized by the *run-time system*. Because the run-time system handles the optimization (instead of the compiler), there are very few circumstances in which the optimization will not work.

To explain how it works, we will go through this code

```
Bin0 = <<0>>,                    %% 1
Bin1 = <<Bin0/binary,1,2,3>>,    %% 2
Bin2 = <<Bin1/binary,4,5,6>>,    %% 3
Bin3 = <<Bin2/binary,7,8,9>>,    %% 4
Bin4 = <<Bin1/binary,17>>,       %% 5 !!!
{Bin4,Bin3}                      %% 6
```

line by line.

The first line (marked with the `%% 1` comment), assigns a *heap binary* to the variable `Bin0`.

The second line is an append operation. Since `Bin0` has not been involved in an append operation, a new *refc binary* will be created and the contents of `Bin0` will be copied into it. The *ProcBin* part of the refc binary will have its size set to the size of the data stored in the binary, while the binary object will have extra space allocated. The size of the binary object will be either twice the size of `Bin0` or 256, whichever is larger. In this case it will be 256.

It gets more interesting in the third line. `Bin1` *has* been used in an append operation, and it has 255 bytes of unused storage at the end, so the three new bytes will be stored there.

Same thing in the fourth line. There are 252 bytes left, so there is no problem storing another three bytes.

But in the fifth line something *interesting* happens. Note that we don't append to the previous result in `Bin3`, but to `Bin1`. We expect that `Bin4` will be assigned the value `<<0,1,2,3,17>>`. We also expect that `Bin3` will retain its value (`<<0,1,2,3,4,5,6,7,8,9>>`). Clearly, the run-time system cannot write the byte `17` into the binary, because that would change the value of `Bin3` to `<<0,1,2,3,4,17,6,7,8,9>>`.

What will happen?

The run-time system will see that `Bin1` is the result from a previous append operation (not from the latest append operation), so it will *copy* the contents of `Bin1` to a new binary and reserve extra storage and so on. (We will not explain here how the run-time system can know that it is not allowed to write into `Bin1`; it is left as an exercise to the curious reader to figure out how it is done by reading the emulator sources, primarily `erl_bits.c`.)

Circumstances that force copying

The optimization of the binary append operation requires that there is a *single* ProcBin and a *single reference* to the ProcBin for the binary. The reason is that the binary object can be moved (reallocated) during an append operation, and when that happens the pointer in the ProcBin must be updated. If there would be more than on ProcBin pointing to the binary object, it would not be possible to find and update all of them.

Therefore, certain operations on a binary will mark it so that any future append operation will be forced to copy the binary. In most cases, the binary object will be shrunk at the same time to reclaim the extra space allocated for growing.

When appending to a binary

```
Bin = <<Bin0,...>>
```

only the binary returned from the latest append operation will support further cheap append operations. In the code fragment above, appending to `Bin` will be cheap, while appending to `Bin0` will force the creation of a new binary and copying of the contents of `Bin0`.

If a binary is sent as a message to a process or port, the binary will be shrunk and any further append operation will copy the binary data into a new binary. For instance, in the following code fragment

```
Bin1 = <<Bin0,...>>,
PortOrPid ! Bin1,
```

```
Bin = <<Bin1,...>>  %% Bin1 will be COPIED
```

`Bin1` will be copied in the third line.

The same thing happens if you insert a binary into an *ets* table or send it to a port using `erlang:port_command/2`.

Matching a binary will also cause it to shrink and the next append operation will copy the binary data:

```
Bin1 = <<Bin0,...>>,
<<X,Y,Z,T/binary>> = Bin1,
Bin = <<Bin1,...>>  %% Bin1 will be COPIED
```

The reason is that a *match context* contains a direct pointer to the binary data.

If a process simply keeps binaries (either in "loop data" or in the process dictionary), the garbage collector may eventually shrink the binaries. If only one such binary is kept, it will not be shrunk. If the process later appends to a binary that has been shrunk, the binary object will be reallocated to make place for the data to be appended.

7.4.3 Matching binaries

We will revisit the example shown earlier

DO (in R12B)

```
my_binary_to_list(<<H,T/binary>>) ->
    [H|my_binary_to_list(T)];
my_binary_to_list(<<>>) -> [].
```

too see what is happening under the hood.

The very first time `my_binary_to_list/1` is called, a *match context* will be created. The match context will point to the first byte of the binary. One byte will be matched out and the match context will be updated to point to the second byte in the binary.

In R11B, at this point a *sub binary* would be created. In R12B, the compiler sees that there is no point in creating a sub binary, because there will soon be a call to a function (in this case, to `my_binary_to_list/1` itself) that will immediately create a new match context and discard the sub binary.

Therefore, in R12B, `my_binary_to_list/1` will call itself with the match context instead of with a sub binary. The instruction that initializes the matching operation will basically do nothing when it sees that it was passed a match context instead of a binary.

When the end of the binary is reached and second clause matches, the match context will simply be discarded (removed in the next garbage collection, since there is no longer any reference to it).

To summarize, `my_binary_to_list/1` in R12B only needs to create *one* match context and no sub binaries. In R11B, if the binary contains *N* bytes, *N+1* match contexts and *N* sub binaries will be created.

In R11B, the fastest way to match binaries is:

DO NOT (in R12B)

```
my_complicated_binary_to_list(Bin) ->
    my_complicated_binary_to_list(Bin, 0).

my_complicated_binary_to_list(Bin, Skip) ->
    case Bin of
```

```
 <<_:Skip/binary,Byte,_/binary>> ->
     [Byte|my_complicated_binary_to_list(Bin, Skip+1)];
 <<_:Skip/binary>> ->
     []
    end.
```

This function cleverly avoids building sub binaries, but it cannot avoid building a match context in each recursion step. Therefore, in both R11B and R12B, `my_complicated_binary_to_list/1` builds *N+1* match contexts. (In a future release, the compiler might be able to generate code that reuses the match context, but don't hold your breath.)

Returning to `my_binary_to_list/1`, note that the match context was discarded when the entire binary had been traversed. What happens if the iteration stops before it has reached the end of the binary? Will the optimization still work?

```
after_zero(<<0,T/binary>>) ->
    T;
after_zero(<<_,T/binary>>) ->
    after_zero(T);
after_zero(<<>>) ->
    <<>>.
```

Yes, it will. The compiler will remove the building of the sub binary in the second clause

```
.
.
.
after_zero(<<_,T/binary>>) ->
    after_zero(T);
.
.
.
```

but will generate code that builds a sub binary in the first clause

```
after_zero(<<0,T/binary>>) ->
    T;
.
.
.
```

Therefore, `after_zero/1` will build one match context and one sub binary (assuming it is passed a binary that contains a zero byte).

Code like the following will also be optimized:

```
all_but_zeroes_to_list(Buffer, Acc, 0) ->
    {lists:reverse(Acc),Buffer};
all_but_zeroes_to_list(<<0,T/binary>>, Acc, Remaining) ->
    all_but_zeroes_to_list(T, Acc, Remaining-1);
all_but_zeroes_to_list(<<Byte,T/binary>>, Acc, Remaining) ->
    all_but_zeroes_to_list(T, [Byte|Acc], Remaining-1).
```

The compiler will remove building of sub binaries in the second and third clauses, and it will add an instruction to the first clause that will convert `Buffer` from a match context to a sub binary (or do nothing if `Buffer` already is a binary).

Before you begin to think that the compiler can optimize any binary patterns, here is a function that the compiler (currently, at least) is not able to optimize:

```
non_opt_eq([H|T1], <<H,T2/binary>>) ->
    non_opt_eq(T1, T2);
non_opt_eq([_|_], <<_,_/binary>>) ->
    false;
non_opt_eq([], <<>>) ->
    true.
```

It was briefly mentioned earlier that the compiler can only delay creation of sub binaries if it can be sure that the binary will not be shared. In this case, the compiler cannot be sure.

We will soon show how to rewrite `non_opt_eq/2` so that the delayed sub binary optimization can be applied, and more importantly, we will show how you can find out whether your code can be optimized.

The bin_opt_info option

Use the `bin_opt_info` option to have the compiler print a lot of information about binary optimizations. It can be given either to the compiler or `erlc`

```
erlc +bin_opt_info Mod.erl
```

or passed via an environment variable

```
export ERL_COMPILER_OPTIONS=bin_opt_info
```

Note that the `bin_opt_info` is not meant to be a permanent option added to your `Makefile`s, because it is not possible to eliminate all messages that it generates. Therefore, passing the option through the environment is in most cases the most practical approach.

The warnings will look like this:

```
./efficiency_guide.erl:60: Warning: NOT OPTIMIZED: sub binary is used or returned
./efficiency_guide.erl:62: Warning: OPTIMIZED: creation of sub binary delayed
```

To make it clearer exactly what code the warnings refer to, in the examples that follow, the warnings are inserted as comments after the clause they refer to:

```
after_zero(<<0,T/binary>>) ->
        %% NOT OPTIMIZED: sub binary is used or returned
    T;
after_zero(<<_,T/binary>>) ->
        %% OPTIMIZED: creation of sub binary delayed
    after_zero(T);
after_zero(<<>>) ->
```

```
    <<>>.
```

The warning for the first clause tells us that it is not possible to delay the creation of a sub binary, because it will be returned. The warning for the second clause tells us that a sub binary will not be created (yet).

It is time to revisit the earlier example of the code that could not be optimized and find out why:

```
non_opt_eq([H|T1], <<H,T2/binary>>) ->
        %% INFO: matching anything else but a plain variable to
 %%    the left of binary pattern will prevent delayed
 %%    sub binary optimization;
 %%    SUGGEST changing argument order
        %% NOT OPTIMIZED: called function non_opt_eq/2 does not
 %%    begin with a suitable binary matching instruction
    non_opt_eq(T1, T2);
non_opt_eq([_|_], <<_,_/binary>>) ->
    false;
non_opt_eq([], <<>>) ->
    true.
```

The compiler emitted two warnings. The `INFO` warning refers to the function `non_opt_eq/2` as a callee, indicating that any functions that call `non_opt_eq/2` will not be able to make delayed sub binary optimization. There is also a suggestion to change argument order. The second warning (that happens to refer to the same line) refers to the construction of the sub binary itself.

We will soon show another example that should make the distinction between `INFO` and `NOT OPTIMIZED` warnings somewhat clearer, but first we will heed the suggestion to change argument order:

```
opt_eq(<<H,T1/binary>>, [H|T2]) ->
        %% OPTIMIZED: creation of sub binary delayed
    opt_eq(T1, T2);
opt_eq(<<_,_/binary>>, [_|_]) ->
    false;
opt_eq(<<>>, []) ->
    true.
```

The compiler gives a warning for the following code fragment:

```
match_body([0|_], <<H,_/binary>>) ->
        %% INFO: matching anything else but a plain variable to
 %%    the left of binary pattern will prevent delayed
 %%    sub binary optimization;
 %%    SUGGEST changing argument order
    done;
.
.
.
```

The warning means that *if* there is a call to `match_body/2` (from another clause in `match_body/2` or another function), the delayed sub binary optimization will not be possible. There will be additional warnings for any place where a sub binary is matched out at the end of and passed as the second argument to `match_body/2`. For instance:

```
match_head(List, <<_:10,Data/binary>>) ->
```

```
       %% NOT OPTIMIZED: called function match_body/2 does not
%%     begin with a suitable binary matching instruction
   match_body(List, Data).
```

Unused variables

The compiler itself figures out if a variable is unused. The same code is generated for each of the following functions

```
count1(<<_,T/binary>>, Count) -> count1(T, Count+1);
count1(<<>>, Count) -> Count.

count2(<<H,T/binary>>, Count) -> count2(T, Count+1);
count2(<<>>, Count) -> Count.

count3(<<_H,T/binary>>, Count) -> count3(T, Count+1);
count3(<<>>, Count) -> Count.
```

In each iteration, the first 8 bits in the binary will be skipped, not matched out.

7.5 List handling

7.5.1 Creating a list

Lists can only be built starting from the end and attaching list elements at the beginning. If you use the `++` operator like this

```
List1 ++ List2
```

you will create a new list which is copy of the elements in `List1`, followed by `List2`. Looking at how `lists:append/1` or `++` would be implemented in plain Erlang, it can be seen clearly that the first list is copied:

```
append([H|T], Tail) ->
    [H|append(T, Tail)];
append([], Tail) ->
    Tail.
```

So the important thing when recursing and building a list is to make sure that you attach the new elements to the beginning of the list, so that you build *a* list, and not hundreds or thousands of copies of the growing result list.

Let us first look at how it should not be done:

DO NOT

```
bad_fib(N) ->
    bad_fib(N, 0, 1, []).

bad_fib(0, _Current, _Next, Fibs) ->
    Fibs;
bad_fib(N, Current, Next, Fibs) ->
    bad_fib(N - 1, Next, Current + Next, Fibs ++ [Current]).
```

Here we are not a building a list; in each iteration step we create a new list that is one element longer than the new previous list.

To avoid copying the result in each iteration, we must build the list in reverse order and reverse the list when we are done:

DO

```
tail_recursive_fib(N) ->
    tail_recursive_fib(N, 0, 1, []).

tail_recursive_fib(0, _Current, _Next, Fibs) ->
    lists:reverse(Fibs);
tail_recursive_fib(N, Current, Next, Fibs) ->
    tail_recursive_fib(N - 1, Next, Current + Next, [Current|Fibs]).
```

7.5.2 List comprehensions

Lists comprehensions still have a reputation for being slow. They used to be implemented using funs, which used to be slow.

In recent Erlang/OTP releases (including R12B), a list comprehension

```
[Expr(E) || E <- List]
```

is basically translated to a local function

```
'lc^0'([E|Tail], Expr) ->
    [Expr(E)|'lc^0'(Tail, Expr)];
'lc^0'([], _Expr) -> [].
```

In R12B, if the result of the list comprehension will *obviously* not be used, a list will not be constructed. For instance, in this code

```
[io:put_chars(E) || E <- List],
ok.
```

or in this code

```
.
.
.
case Var of
    ... ->
        [io:put_chars(E) || E <- List];
    ... ->
end,
some_function(...),
.
.
.
```

the value is neither assigned to a variable, nor passed to another function, nor returned, so there is no need to construct a list and the compiler will simplify the code for the list comprehension to

```
'lc^0'([E|Tail], Expr) ->
    Expr(E),
    'lc^0'(Tail, Expr);
'lc^0'([], _Expr) -> [].
```

7.5.3 Deep and flat lists

lists:flatten/1 builds an entirely new list. Therefore, it is expensive, and even *more* expensive than the `++` (which copies its left argument, but not its right argument).

In the following situations, you can easily avoid calling `lists:flatten/1`:

- When sending data to a port. Ports understand deep lists so there is no reason to flatten the list before sending it to the port.
- When calling BIFs that accept deep lists, such as *list_to_binary/1* or *iolist_to_binary/1*.
- When you know that your list is only one level deep, you can can use *lists:append/1*.

Port example

DO

```
    ...
    port_command(Port, DeepList)
    ...
```

DO NOT

```
    ...
    port_command(Port, lists:flatten(DeepList))
    ...
```

A common way to send a zero-terminated string to a port is the following:

DO NOT

```
    ...
    TerminatedStr = String ++ [0], % String="foo" => [$f, $o, $o, 0]
    port_command(Port, TerminatedStr)
    ...
```

Instead do like this:

DO

```
    ...
    TerminatedStr = [String, 0], % String="foo" => [[$f, $o, $o], 0]
    port_command(Port, TerminatedStr)
    ...
```

Append example

DO

```
> lists:append([[1], [2], [3]]).
[1,2,3]
>
```

DO NOT

```
> lists:flatten([[1], [2], [3]]).
[1,2,3]
>
```

7.5.4 Why you should not worry about recursive lists functions

In the performance myth chapter, the following myth was exposed: *Tail-recursive functions are MUCH faster than recursive functions.*

To summarize, in R12B there is usually not much difference between a body-recursive list function and tail-recursive function that reverses the list at the end. Therefore, concentrate on writing beautiful code and forget about the performance of your list functions. In the time-critical parts of your code (and only there), *measure* before rewriting your code.

Important note: This section talks about lists functions that *construct* lists. A tail-recursive function that does not construct a list runs in constant space, while the corresponding body-recursive function uses stack space proportional to the length of the list. For instance, a function that sums a list of integers, should *not* be written like this

DO NOT

```
recursive_sum([H|T]) -> H+recursive_sum(T);
recursive_sum([])    -> 0.
```

but like this

DO

```
sum(L) -> sum(L, 0).

sum([H|T], Sum) -> sum(T, Sum + H);
sum([], Sum)    -> Sum.
```

7.6 Functions

7.6.1 Pattern matching

Pattern matching in function head and in `case` and `receive` clauses are optimized by the compiler. With a few exceptions, there is nothing to gain by rearranging clauses.

One exception is pattern matching of binaries. The compiler will not rearrange clauses that match binaries. Placing the clause that matches against the empty binary *last* will usually be slightly faster than placing it *first*.

Here is a rather contrived example to show another exception:

DO NOT

```
atom_map1(one) -> 1;
atom_map1(two) -> 2;
atom_map1(three) -> 3;
atom_map1(Int) when is_integer(Int) -> Int;
atom_map1(four) -> 4;
atom_map1(five) -> 5;
atom_map1(six) -> 6.
```

The problem is the clause with the variable `Int`. Since a variable can match anything, including the atoms `four`, `five`, and `six` that the following clauses also will match, the compiler must generate sub-optimal code that will execute as follows:

First the input value is compared to `one`, `two`, and `three` (using a single instruction that does a binary search; thus, quite efficient even if there are many values) to select which one of the first three clauses to execute (if any).

If none of the first three clauses matched, the fourth clause will match since a variable always matches. If the guard test `is_integer(Int)` succeeds, the fourth clause will be executed.

If the guard test failed, the input value is compared to `four`, `five`, and `six`, and the appropriate clause is selected. (There will be a `function_clause` exception if none of the values matched.)

Rewriting to either

DO

```
atom_map2(one) -> 1;
atom_map2(two) -> 2;
atom_map2(three) -> 3;
atom_map2(four) -> 4;
atom_map2(five) -> 5;
atom_map2(six) -> 6;
atom_map2(Int) when is_integer(Int) -> Int.
```

or

DO

```
atom_map3(Int) when is_integer(Int) -> Int;
atom_map3(one) -> 1;
atom_map3(two) -> 2;
atom_map3(three) -> 3;
atom_map3(four) -> 4;
atom_map3(five) -> 5;
atom_map3(six) -> 6.
```

will give slightly more efficient matching code.

Here is a less contrived example:

DO NOT

```
map_pairs1(_Map, [], Ys) ->
```

```
    Ys;
map_pairs1(_Map, Xs, [] ) ->
    Xs;
map_pairs1(Map, [X|Xs], [Y|Ys]) ->
    [Map(X, Y)|map_pairs1(Map, Xs, Ys)].
```

The first argument is *not* a problem. It is variable, but it is a variable in all clauses. The problem is the variable in the second argument, `Xs`, in the middle clause. Because the variable can match anything, the compiler is not allowed to rearrange the clauses, but must generate code that matches them in the order written.

If the function is rewritten like this

DO

```
map_pairs2(_Map, [], Ys) ->
    Ys;
map_pairs2(_Map, [_|_]=Xs, [] ) ->
    Xs;
map_pairs2(Map, [X|Xs], [Y|Ys]) ->
    [Map(X, Y)|map_pairs2(Map, Xs, Ys)].
```

the compiler is free rearrange the clauses. It will generate code similar to this

DO NOT (already done by the compiler)

```
explicit_map_pairs(Map, Xs0, Ys0) ->
    case Xs0 of
 [X|Xs] ->
     case Ys0 of
  [Y|Ys] ->
      [Map(X, Y)|explicit_map_pairs(Map, Xs, Ys)];
  [] ->
      Xs0
     end;
 [] ->
     Ys0
   end.
```

which should be slightly faster for presumably the most common case that the input lists are not empty or very short. (Another advantage is that Dialyzer is able to deduce a better type for the variable `Xs`.)

7.6.2 Function Calls

Here is an intentionally rough guide to the relative costs of different kinds of calls. It is based on benchmark figures run on Solaris/Sparc:

- Calls to local or external functions (`foo()`, `m:foo()`) are the fastest kind of calls.
- Calling or applying a fun (`Fun()`, `apply(Fun, [])`) is about *three times* as expensive as calling a local function.
- Applying an exported function (`Mod:Name()`, `apply(Mod, Name, [])`) is about twice as expensive as calling a fun, or about *six times* as expensive as calling a local function.

Notes and implementation details

Calling and applying a fun does not involve any hash-table lookup. A fun contains an (indirect) pointer to the function that implements the fun.

> Warning:
>
> *Tuples are not fun(s).* A "tuple fun", `{Module,Function}`, is not a fun. The cost for calling a "tuple fun" is similar to that of `apply/3` or worse. Using "tuple funs" is *strongly discouraged*, as they may not be supported in a future release.

`apply/3` must look up the code for the function to execute in a hash table. Therefore, it will always be slower than a direct call or a fun call.

It no longer matters (from a performance point of view) whether you write

```
Module:Function(Arg1, Arg2)
```

or

```
apply(Module, Function, [Arg1,Arg2])
```

(The compiler internally rewrites the latter code into the former.)

The following code

```
apply(Module, Function, Arguments)
```

is slightly slower because the shape of the list of arguments is not known at compile time.

7.6.3 Memory usage in recursion

When writing recursive functions it is preferable to make them tail-recursive so that they can execute in constant memory space.

DO

```
list_length(List) ->
    list_length(List, 0).

list_length([], AccLen) ->
    AccLen; % Base case

list_length([_|Tail], AccLen) ->
    list_length(Tail, AccLen + 1). % Tail-recursive
```

DO NOT

```
list_length([]) ->
    0. % Base case
list_length([_ | Tail]) ->
    list_length(Tail) + 1. % Not tail-recursive
```

7.7 Tables and databases

7.7.1 Ets, Dets and Mnesia

Every example using Ets has a corresponding example in Mnesia. In general all Ets examples also apply to Dets tables.

Select/Match operations

Select/Match operations on Ets and Mnesia tables can become very expensive operations. They usually need to scan the complete table. You should try to structure your data so that you minimize the need for select/match operations. However, if you really need a select/match operation, it will still be more efficient than using `tab2list`. Examples of this and also of ways to avoid select/match will be provided in some of the following sections. The functions `ets:select/2` and `mnesia:select/3` should be preferred over `ets:match/2`,`ets:match_object/2`, and `mnesia:match_object/3`.

> **Note:**
>
> There are exceptions when the complete table is not scanned, for instance if part of the key is bound when searching an `ordered_set` table, or if it is a Mnesia table and there is a secondary index on the field that is selected/matched. If the key is fully bound there will, of course, be no point in doing a select/match, unless you have a bag table and you are only interested in a sub-set of the elements with the specific key.

When creating a record to be used in a select/match operation you want most of the fields to have the value '_'. The easiest and fastest way to do that is as follows:

```
#person{age = 42, _ = '_'}.
```

Deleting an element

The delete operation is considered successful if the element was not present in the table. Hence all attempts to check that the element is present in the Ets/Mnesia table before deletion are unnecessary. Here follows an example for Ets tables.

DO

```
...
ets:delete(Tab, Key),
...
```

DO NOT

```
...
case ets:lookup(Tab, Key) of
    [] ->
        ok;
    [_|_] ->
        ets:delete(Tab, Key)
end,
...
```

Data fetching

Do not fetch data that you already have! Consider that you have a module that handles the abstract data type Person. You export the interface function `print_person/1` that uses the internal functions `print_name/1`, `print_age/1`, `print_occupation/1`.

> **Note:**
>
> If the functions `print_name/1` and so on, had been interface functions the matter comes in to a whole new light, as you do not want the user of the interface to know about the internal data representation.

DO

```
%%% Interface function
print_person(PersonId) ->
    %% Look up the person in the named table person,
    case ets:lookup(person, PersonId) of
        [Person] ->
            print_name(Person),
            print_age(Person),
            print_occupation(Person);
        [] ->
            io:format("No person with ID = ~p~n", [PersonID])
    end.

%%% Internal functions
print_name(Person) ->
    io:format("No person ~p~n", [Person#person.name]).

print_age(Person) ->
    io:format("No person ~p~n", [Person#person.age]).

print_occupation(Person) ->
    io:format("No person ~p~n", [Person#person.occupation]).
```

DO NOT

```
%%% Interface function
print_person(PersonId) ->
    %% Look up the person in the named table person,
    case ets:lookup(person, PersonId) of
        [Person] ->
            print_name(PersonID),
            print_age(PersonID),
            print_occupation(PersonID);
        [] ->
            io:format("No person with ID = ~p~n", [PersonID])
    end.

%%% Internal functionss
print_name(PersonID) ->
    [Person] = ets:lookup(person, PersonId),
    io:format("No person ~p~n", [Person#person.name]).

print_age(PersonID) ->
    [Person] = ets:lookup(person, PersonId),
```

```
    io:format("No person ~p~n", [Person#person.age]).

print_occupation(PersonID) ->
    [Person] = ets:lookup(person, PersonId),
    io:format("No person ~p~n", [Person#person.occupation]).
```

Non-persistent data storage

For non-persistent database storage, prefer Ets tables over Mnesia local_content tables. Even the Mnesia `dirty_write` operations carry a fixed overhead compared to Ets writes. Mnesia must check if the table is replicated or has indices, this involves at least one Ets lookup for each `dirty_write`. Thus, Ets writes will always be faster than Mnesia writes.

tab2list

Assume we have an Ets-table, which uses `idno` as key, and contains:

```
[#person{idno = 1, name = "Adam",  age = 31, occupation = "mailman"},
 #person{idno = 2, name = "Bryan", age = 31, occupation = "cashier"},
 #person{idno = 3, name = "Bryan", age = 35, occupation = "banker"},
 #person{idno = 4, name = "Carl",  age = 25, occupation = "mailman"}]
```

If we *must* return all data stored in the Ets-table we can use `ets:tab2list/1`. However, usually we are only interested in a subset of the information in which case `ets:tab2list/1` is expensive. If we only want to extract one field from each record, e.g., the age of every person, we should use:

DO

```
...
ets:select(Tab,[{ #person{idno='_',
                          name='_',
                          age='$1',
                          occupation = '_'},
                [],
                ['$1']}]),
...
```

DO NOT

```
...
TabList = ets:tab2list(Tab),
lists:map(fun(X) -> X#person.age end, TabList),
...
```

If we are only interested in the age of all persons named Bryan, we should:

DO

```
...
ets:select(Tab,[{ #person{idno='_',
                          name="Bryan",
                          age='$1',
                          occupation = '_'},
                [],
```

```
                ['$1']}]),
...
```

DO NOT

```
...
TabList = ets:tab2list(Tab),
lists:foldl(fun(X, Acc) -> case X#person.name of
                               "Bryan" ->
                                   [X#person.age|Acc];
                                _ ->
                                    Acc
                           end
            end, [], TabList),
...
```

REALLY DO NOT

```
...
TabList = ets:tab2list(Tab),
BryanList = lists:filter(fun(X) -> X#person.name == "Bryan" end,
                         TabList),
lists:map(fun(X) -> X#person.age end, BryanList),
...
```

If we need all information stored in the Ets table about persons named Bryan we should:

DO

```
...
ets:select(Tab, [{#person{idno='_',
                          name="Bryan",
                          age='_',
                          occupation = '_'}, [], ['$_']}]),
...
```

DO NOT

```
...
TabList = ets:tab2list(Tab),
lists:filter(fun(X) -> X#person.name == "Bryan" end, TabList),
...
```

Ordered_set tables

If the data in the table should be accessed so that the order of the keys in the table is significant, the table type `ordered_set` could be used instead of the more usual `set` table type. An `ordered_set` is always traversed in Erlang term order with regard to the key field so that return values from functions such as `select`, `match_object`, and `foldl` are ordered by the key values. Traversing an `ordered_set` with the `first` and `next` operations also returns the keys ordered.

> Note:
>
> An `ordered_set` only guarantees that objects are processed in *key* order. Results from functions as `ets:select/2` appear in the *key* order even if the key is not included in the result.

7.7.2 Ets specific

Utilizing the keys of the Ets table

An Ets table is a single key table (either a hash table or a tree ordered by the key) and should be used as one. In other words, use the key to look up things whenever possible. A lookup by a known key in a set Ets table is constant and for a ordered_set Ets table it is O(logN). A key lookup is always preferable to a call where the whole table has to be scanned. In the examples above, the field `idno` is the key of the table and all lookups where only the name is known will result in a complete scan of the (possibly large) table for a matching result.

A simple solution would be to use the `name` field as the key instead of the `idno` field, but that would cause problems if the names were not unique. A more general solution would be create a second table with name as key and idno as data, i.e. to index (invert) the table with regards to the `name` field. The second table would of course have to be kept consistent with the master table. Mnesia could do this for you, but a home brew index table could be very efficient compared to the overhead involved in using Mnesia.

An index table for the table in the previous examples would have to be a bag (as keys would appear more than once) and could have the following contents:

```
[#index_entry{name="Adam", idno=1},
 #index_entry{name="Bryan", idno=2},
 #index_entry{name="Bryan", idno=3},
 #index_entry{name="Carl", idno=4}]
```

Given this index table a lookup of the `age` fields for all persons named "Bryan" could be done like this:

```
...
MatchingIDs = ets:lookup(IndexTable,"Bryan"),
lists:map(fun(#index_entry{idno = ID}) ->
                [#person{age = Age}] = ets:lookup(PersonTable, ID),
                Age
          end,
          MatchingIDs),
...
```

Note that the code above never uses `ets:match/2` but instead utilizes the `ets:lookup/2` call. The `lists:map/2` call is only used to traverse the `idno`s matching the name "Bryan" in the table; therefore the number of lookups in the master table is minimized.

Keeping an index table introduces some overhead when inserting records in the table, therefore the number of operations gained from the table has to be weighted against the number of operations inserting objects in the table. However, note that the gain when the key can be used to lookup elements is significant.

7.7.3 Mnesia specific

Secondary index

If you frequently do a lookup on a field that is not the key of the table, you will lose performance using "mnesia:select/match_object" as this function will traverse the whole table. You may create a secondary index instead and use "mnesia:index_read" to get faster access, however this will require more memory. Example:

```
-record(person, {idno, name, age, occupation}).
        ...
{atomic, ok} =
mnesia:create_table(person, [{index,[#person.age]},
                             {attributes,
                                   record_info(fields, person)}]),
{atomic, ok} = mnesia:add_table_index(person, age),
...

PersonsAge42 =
     mnesia:dirty_index_read(person, 42, #person.age),
...
```

Transactions

Transactions is a way to guarantee that the distributed Mnesia database remains consistent, even when many different processes update it in parallel. However if you have real time requirements it is recommended to use dirty operations instead of transactions. When using the dirty operations you lose the consistency guarantee, this is usually solved by only letting one process update the table. Other processes have to send update requests to that process.

```
...
% Using transaction

Fun = fun() ->
          [mnesia:read({Table, Key}),
           mnesia:read({Table2, Key2})]
      end,

{atomic, [Result1, Result2]}  = mnesia:transaction(Fun),
...

% Same thing using dirty operations
...

Result1 = mnesia:dirty_read({Table, Key}),
Result2 = mnesia:dirty_read({Table2, Key2}),
...
```

7.8 Processes

7.8.1 Creation of an Erlang process

An Erlang process is lightweight compared to operating systems threads and processes.

A newly spawned Erlang process uses 309 words of memory in the non-SMP emulator without HiPE support. (SMP support and HiPE support will both add to this size.) The size can be found out like this:

```
Erlang (BEAM) emulator version 5.6 [async-threads:0] [kernel-poll:false]

Eshell V5.6  (abort with ^G)
1> Fun = fun() -> receive after infinity -> ok end end.
#Fun<...>
2> {_,Bytes} = process_info(spawn(Fun), memory).
{memory,1232}
3> Bytes div erlang:system_info(wordsize).
309
```

The size includes 233 words for the heap area (which includes the stack). The garbage collector will increase the heap as needed.

The main (outer) loop for a process *must* be tail-recursive. If not, the stack will grow until the process terminates.

DO NOT

```
loop() ->
  receive
     {sys, Msg} ->
         handle_sys_msg(Msg),
         loop();
     {From, Msg} ->
          Reply = handle_msg(Msg),
          From ! Reply,
          loop()
  end,
  io:format("Message is processed~n", []).
```

The call to `io:format/2` will never be executed, but a return address will still be pushed to the stack each time `loop/0` is called recursively. The correct tail-recursive version of the function looks like this:

DO

```
   loop() ->
      receive
         {sys, Msg} ->
            handle_sys_msg(Msg),
            loop();
         {From, Msg} ->
            Reply = handle_msg(Msg),
            From ! Reply,
            loop()
    end.
```

Initial heap size

The default initial heap size of 233 words is quite conservative in order to support Erlang systems with hundreds of thousands or even millions of processes. The garbage collector will grow and shrink the heap as needed.

In a system that use comparatively few processes, performance *might* be improved by increasing the minimum heap size using either the `+h` option for *erl* or on a process-per-process basis using the `min_heap_size` option for *spawn_opt/4*.

The gain is twofold: Firstly, although the garbage collector will grow the heap, it will it grow it step by step, which will be more costly than directly establishing a larger heap when the process is spawned. Secondly, the garbage collector

may also shrink the heap if it is much larger than the amount of data stored on it; setting the minimum heap size will prevent that.

> Warning:
>
> The emulator will probably use more memory, and because garbage collections occur less frequently, huge binaries could be kept much longer.

In systems with many processes, computation tasks that run for a short time could be spawned off into a new process with a higher minimum heap size. When the process is done, it will send the result of the computation to another process and terminate. If the minimum heap size is calculated properly, the process may not have to do any garbage collections at all. *This optimization should not be attempted without proper measurements.*

7.8.2 Process messages

All data in messages between Erlang processes is copied, with the exception of *refc binaries* on the same Erlang node.

When a message is sent to a process on another Erlang node, it will first be encoded to the Erlang External Format before being sent via an TCP/IP socket. The receiving Erlang node decodes the message and distributes it to the right process.

The constant pool

Constant Erlang terms (also called *literals*) are now kept in constant pools; each loaded module has its own pool. The following function

DO (in R12B and later)

```
days_in_month(M) ->
    element(M, {31,28,31,30,31,30,31,31,30,31,30,31}).
```

will no longer build the tuple every time it is called (only to have it discarded the next time the garbage collector was run), but the tuple will be located in the module's constant pool.

But if a constant is sent to another process (or stored in an ETS table), it will be *copied*. The reason is that the run-time system must be able to keep track of all references to constants in order to properly unload code containing constants. (When the code is unloaded, the constants will be copied to the heap of the processes that refer to them.) The copying of constants might be eliminated in a future release.

Loss of sharing

Shared sub-terms are *not* preserved when a term is sent to another process, passed as the initial process arguments in the `spawn` call, or stored in an ETS table. That is an optimization. Most applications do not send message with shared sub-terms.

Here is an example of how a shared sub-term can be created:

```
kilo_byte() ->
    kilo_byte(10, [42]).

kilo_byte(0, Acc) ->
    Acc;
kilo_byte(N, Acc) ->
```

```
    kilo_byte(N-1, [Acc|Acc]).
```

`kilo_byte/1` creates a deep list. If we call `list_to_binary/1`, we can convert the deep list to a binary of 1024 bytes:

```
1> byte_size(list_to_binary(efficiency_guide:kilo_byte())).
1024
```

Using the `erts_debug:size/1` BIF we can see that the deep list only requires 22 words of heap space:

```
2> erts_debug:size(efficiency_guide:kilo_byte()).
22
```

Using the `erts_debug:flat_size/1` BIF, we can calculate the size of the deep list if sharing is ignored. It will be the size of the list when it has been sent to another process or stored in an ETS table:

```
3> erts_debug:flat_size(efficiency_guide:kilo_byte()).
4094
```

We can verify that sharing will be lost if we insert the data into an ETS table:

```
4> T = ets:new(tab, []).
17
5> ets:insert(T, {key,efficiency_guide:kilo_byte()}).
true
6> erts_debug:size(element(2, hd(ets:lookup(T, key)))).
4094
7> erts_debug:flat_size(element(2, hd(ets:lookup(T, key)))).
4094
```

When the data has passed through an ETS table, `erts_debug:size/1` and `erts_debug:flat_size/1` return the same value. Sharing has been lost.

In a future release of Erlang/OTP, we might implement a way to (optionally) preserve sharing. We have no plans to make preserving of sharing the default behaviour, since that would penalize the vast majority of Erlang applications.

7.8.3 The SMP emulator

The SMP emulator (introduced in R11B) will take advantage of multi-core or multi-CPU computer by running several Erlang schedulers threads (typically, the same as the number of cores). Each scheduler thread schedules Erlang processes in the same way as the Erlang scheduler in the non-SMP emulator.

To gain performance by using the SMP emulator, your application *must have more than one runnable Erlang process* most of the time. Otherwise, the Erlang emulator can still only run one Erlang process at the time, but you must still pay the overhead for locking. Although we try to reduce the locking overhead as much as possible, it will never become exactly zero.

Benchmarks that may seem to be concurrent are often sequential. The estone benchmark, for instance, is entirely sequential. So is also the most common implementation of the "ring benchmark"; usually one process is active, while the others wait in a `receive` statement.

The *percept* application can be used to profile your application to see how much potential (or lack thereof) it has for concurrency.

7.9 Drivers

This chapter provides a (very) brief overview on how to write efficient drivers. It is assumed that you already have a good understanding of drivers.

7.9.1 Drivers and concurrency

The run-time system will always take a lock before running any code in a driver.

By default, that lock will be at the driver level, meaning that if several ports has been opened to the same driver, only code for one port at the same time can be running.

A driver can be configured to instead have one lock for each port.

If a driver is used in a functional way (i.e. it holds no state, but only does some heavy calculation and returns a result), several ports with registered names can be opened beforehand and the port to be used can be chosen based on the scheduler ID like this:

```
-define(PORT_NAMES(),
 {some_driver_01, some_driver_02, some_driver_03, some_driver_04,
  some_driver_05, some_driver_06, some_driver_07, some_driver_08,
  some_driver_09, some_driver_10, some_driver_11, some_driver_12,
  some_driver_13, some_driver_14, some_driver_15, some_driver_16}).

client_port() ->
    element(erlang:system_info(scheduler_id) rem tuple_size(?PORT_NAMES()) + 1,
     ?PORT_NAMES()).
```

As long as there are no more than 16 schedulers, there will never be any lock contention on the port lock for the driver.

7.9.2 Avoiding copying of binaries when calling a driver

There are basically two ways to avoid copying a binary that is sent to a driver.

If the `Data` argument for *port_control/3* is a binary, the driver will be passed a pointer to the contents of the binary and the binary will not be copied. If the `Data` argument is an iolist (list of binaries and lists), all binaries in the iolist will be copied.

Therefore, if you want to send both a pre-existing binary and some additional data to a driver without copying the binary, you must call `port_control/3` twice; once with the binary and once with the additional data. However, that will only work if there is only one process communicating with the port (because otherwise another process could call the driver in-between the calls).

Another way to avoid copying binaries is to implement an `outputv` callback (instead of an `output` callback) in the driver. If a driver has an `outputv` callback, refc binaries passed in an iolist in the `Data` argument for *port_command/2* will be passed as references to the driver.

7.9.3 Returning small binaries from a driver

The run-time system can represent binaries up to 64 bytes as heap binaries. They will always be copied when sent in a messages, but they will require less memory if they are not sent to another process and garbage collection is cheaper.

If you know that the binaries you return are always small, you should use driver API calls that do not require a pre-allocated binary, for instance *driver_output()* or *driver_output_term()* using the `ERL_DRV_BUF2BINARY` format, to allow the run-time to construct a heap binary.

7.9.4 Returning big binaries without copying from a driver

To avoid copying data when a big binary is sent or returned from the driver to an Erlang process, the driver must first allocate the binary and then send it to an Erlang process in some way.

Use *driver_alloc_binary()* to allocate a binary.

There are several ways to send a binary created with `driver_alloc_binary()`.

- From the `control` callback, a binary can be returned provided that *set_port_control()* has been called with the flag value `PORT_CONTROL_FLAG_BINARY`.
- A single binary can be sent with *driver_output_binary()*.
- Using *driver_output_term()* or *driver_send_term()*, a binary can be included in an Erlang term.

7.10 Advanced

7.10.1 Memory

A good start when programming efficiently is to have knowledge about how much memory different data types and operations require. It is implementation-dependent how much memory the Erlang data types and other items consume, but here are some figures for erts-5.2 system (OTP release R9B). (There have been no significant changes in R13.)

The unit of measurement is memory words. There exists both a 32-bit and a 64-bit implementation, and a word is therefore, 4 bytes or 8 bytes, respectively.

Data type	Memory size
Integer (-16#7FFFFF < i <16#7FFFFF)	1 word
Integer (big numbers)	3..N words
Atom	1 word. Note: an atom refers into an atom table which also consumes memory. The atom text is stored once for each unique atom in this table. The atom table is *not* garbage-collected.
Float	On 32-bit architectures: 4 words On 64-bit architectures: 3 words
Binary	3..6 + data (can be shared)
List	1 word per element + the size of each element
String (is the same as a list of integers)	2 words per character
Tuple	2 words + the size of each element
Pid	1 word for a process identifier from the current local node, and 5 words for a process identifier from another

	node. Note: a process identifier refers into a process table and a node table which also consumes memory.
Port	1 word for a port identifier from the current local node, and 5 words for a port identifier from another node. Note: a port identifier refers into a port table and a node table which also consumes memory.
Reference	On 32-bit architectures: 5 words for a reference from the current local node, and 7 words for a reference from another node. On 64-bit architectures: 4 words for a reference from the current local node, and 6 words for a reference from another node. Note: a reference refers into a node table which also consumes memory.
Fun	9..13 words + size of environment. Note: a fun refers into a fun table which also consumes memory.
Ets table	Initially 768 words + the size of each element (6 words + size of Erlang data). The table will grow when necessary.
Erlang process	327 words when spawned including a heap of 233 words.

Table 10.1: Memory size of different data types

7.10.2 System limits

The Erlang language specification puts no limits on number of processes, length of atoms etc., but for performance and memory saving reasons, there will always be limits in a practical implementation of the Erlang language and execution environment.

Processes

The maximum number of simultaneously alive Erlang processes is by default 32768. This limit can be raised up to at most 268435456 processes at startup (see documentation of the system flag *+P* in the *erl(1)* documentation). The maximum limit of 268435456 processes will at least on a 32-bit architecture be impossible to reach due to memory shortage.

Distributed nodes

Known nodes

A remote node Y has to be known to node X if there exist any pids, ports, references, or funs (Erlang data types) from Y on X, or if X and Y are connected. The maximum number of remote nodes simultaneously/ever known to a node is limited by the *maximum number of atoms* available for node names. All data concerning remote nodes, except for the node name atom, are garbage-collected.

Connected nodes

The maximum number of simultaneously connected nodes is limited by either the maximum number of simultaneously known remote nodes, *the maximum number of (Erlang) ports* available, or *the maximum number of sockets* available.

Characters in an atom
255
Atoms
The maximum number of atoms is 1048576.
Ets-tables
The default is 1400, can be changed with the environment variable `ERL_MAX_ETS_TABLES`.
Elements in a tuple
The maximum number of elements in a tuple is 67108863 (26 bit unsigned integer). Other factors such as the available memory can of course make it hard to create a tuple of that size.
Size of binary
In the 32-bit implementation of Erlang, 536870911 bytes is the largest binary that can be constructed or matched using the bit syntax. (In the 64-bit implementation, the maximum size is 2305843009213693951 bytes.) If the limit is exceeded, bit syntax construction will fail with a `system_limit` exception, while any attempt to match a binary that is too large will fail. This limit is enforced starting with the R11B-4 release; in earlier releases, operations on too large binaries would in general either fail or give incorrect results. In future releases of Erlang/OTP, other operations that create binaries (such as `list_to_binary/1`) will probably also enforce the same limit.
Total amount of data allocated by an Erlang node
The Erlang runtime system can use the complete 32 (or 64) bit address space, but the operating system often limits a single process to use less than that.
length of a node name
An Erlang node name has the form host@shortname or host@longname. The node name is used as an atom within the system so the maximum size of 255 holds for the node name too.
Open ports

The maximum number of simultaneously open Erlang ports is by default 1024. This limit can be raised up to at most 268435456 at startup (see environment variable *ERL_MAX_PORTS* in *erlang(3)*) The maximum limit of 268435456 open ports will at least on a 32-bit architecture be impossible to reach due to memory shortage.

Open files, and sockets
The maximum number of simultaneously open files and sockets depend on *the maximum number of Erlang ports* available, and operating system specific settings and limits.
Number of arguments to a function or fun
256

7.11 Profiling

7.11.1 Do not guess about performance - profile

Even experienced software developers often guess wrong about where the performance bottlenecks are in their programs.

Therefore, profile your program to see where the performance bottlenecks are and concentrate on optimizing them.

Erlang/OTP contains several tools to help finding bottlenecks.

`fprof` and `eprof` provide the most detailed information about where the time is spent, but they significantly slow downs the programs they profile.

If the program is too big to be profiled by `fprof` or `eprof`, `cover` and `cprof` could be used to locate parts of the code that should be more thoroughly profiled using `fprof` or `eprof`.

`cover` provides execution counts per line per process, with less overhead than `fprof/eprof`. Execution counts can with some caution be used to locate potential performance bottlenecks. The most lightweight tool is `cprof`, but it only provides execution counts on a function basis (for all processes, not per process).

7.11.2 Big systems

If you have a big system it might be interesting to run profiling on a simulated and limited scenario to start with. But bottlenecks have a tendency to only appear or cause problems when there are many things going on at the same time, and when there are many nodes involved. Therefore it is desirable to also run profiling in a system test plant on a real target system.

When your system is big you do not want to run the profiling tools on the whole system. You want to concentrate on processes and modules that you know are central and stand for a big part of the execution.

7.11.3 What to look for

When analyzing the result file from the profiling activity you should look for functions that are called many times and have a long "own" execution time (time excluded calls to other functions). Functions that just are called very many times can also be interesting, as even small things can add up to quite a bit if they are repeated often. Then you need to ask yourself what can I do to reduce this time. Appropriate types of questions to ask yourself are:

- Can I reduce the number of times the function is called?
- Are there tests that can be run less often if I change the order of tests?
- Are there redundant tests that can be removed?
- Is there some expression calculated giving the same result each time?
- Is there other ways of doing this that are equivalent and more efficient?
- Can I use another internal data representation to make things more efficient?

These questions are not always trivial to answer. You might need to do some benchmarks to back up your theory, to avoid making things slower if your theory is wrong. See *benchmarking*.

7.11.4 Tools

fprof

`fprof` measures the execution time for each function, both own time i.e how much time a function has used for its own execution, and accumulated time i.e. including called functions. The values are displayed per process. You also get to know how many times each function has been called. `fprof` is based on trace to file in order to minimize runtime performance impact. Using fprof is just a matter of calling a few library functions, see fprof manual page under the application tools.

`fprof` was introduced in version R8 of Erlang/OTP. Its predecessor `eprof` that is based on the Erlang trace BIFs, is still available, see eprof manual page under the application tools. Eprof shows how much time has been used by each process, and in which function calls this time has been spent. Time is shown as percentage of total time, not as absolute time.

cover

`cover`'s primary use is coverage analysis to verify test cases, making sure all relevant code is covered. `cover` counts how many times each executable line of code is executed when a program is run. This is done on a per module basis. Of course this information can be used to determine what code is run very frequently and could therefore be subject for optimization. Using cover is just a matter of calling a few library functions, see cover manual page under the application tools.

cprof

`cprof` is something in between `fprof` and `cover` regarding features. It counts how many times each function is called when the program is run, on a per module basis. `cprof` has a low performance degradation (versus `fprof` and `eprof`) and does not need to recompile any modules to profile (versus `cover`).

Tool summarization

Tool	Results	Size of result	Effects on program execution time	Records number of calls	Records Execution time	Records called by	Records garbage collection
`fprof`	per process to screen/ file	large	significant slowdown	yes	total and own	yes	yes
`eprof`	per process/ function to screen/file	medium	significant slowdown	yes	only total	no	no
`cover`	per module to screen/ file	small	moderate slowdown	yes, per line	no	no	no
`cprof`	per module to caller	small	small slowdown	yes	no	no	no

Table 11.1:

7.11.5 Benchmarking

The main purpose of benchmarking is to find out which implementation of a given algorithm or function is the fastest. Benchmarking is far from an exact science. Today's operating systems generally run background tasks that are difficult to turn off. Caches and multiple CPU cores doesn't make it any easier. It would be best to run Unix-computers in single-user mode when benchmarking, but that is inconvenient to say the least for casual testing.

Benchmarks can measure wall-clock time or CPU time.

timer:tc/3 measures wall-clock time. The advantage with wall-clock time is that I/O, swapping, and other activities in the operating-system kernel are included in the measurements. The disadvantage is that the the measurements will vary wildly. Usually it is best to run the benchmark several times and note the shortest time - that time should be the minimum time that is possible to achieve under the best of circumstances.

statistics/1 with the argument `runtime` measures CPU time spent in the Erlang virtual machine. The advantage is that the results are more consistent from run to run. The disadvantage is that the time spent in the operating system kernel (such as swapping and I/O) are not included. Therefore, measuring CPU time is misleading if any I/O (file or sockets) are involved.

It is probably a good idea to do both wall-clock measurements and CPU time measurements.

Some additional advice:

- The granularity of both types measurement could be quite high so you should make sure that each individual measurement lasts for at least several seconds.
- To make the test fair, each new test run should run in its own, newly created Erlang process. Otherwise, if all tests runs in the same process, the later tests would start out with larger heap sizes and therefore probably does less garbage collections. You could also consider restarting the Erlang emulator between each test.
- Do not assume that the fastest implementation of a given algorithm on computer architecture X also is the fast on computer architecture Y.

8 User's Guide

8.1 Introduction

8.1.1 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.

8.1.2 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.

8.2 Overview

8.2.1 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*, 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.

8.2.2 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* 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.

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*, and distributed Erlang, see *above*.

Examples:C node example.

Jinterface

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

8.2.3 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`.

8.2.4 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.

8.2.5 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.

8.3 Problem Example

8.3.1 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.

8.4 Ports

This is an example of how to solve the *example problem* by using a port.

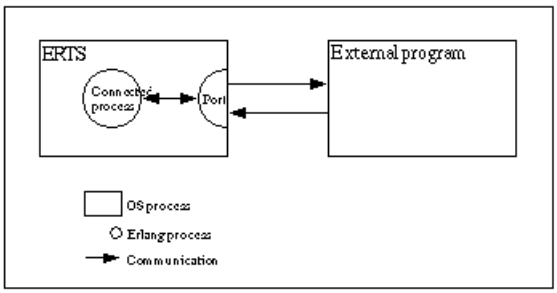

Figure 4.1: Port Communication.

8.4.1 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
\011{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.
```

8.4.2 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.

8.4.3 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
```

8.5 Erl_Interface

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

8.5.1 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.

8.5.2 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);
  }
}
```

8.5.3 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
```

8.6 Port drivers

This is an example of how to solve the *example problem* by using a linked in port driver.

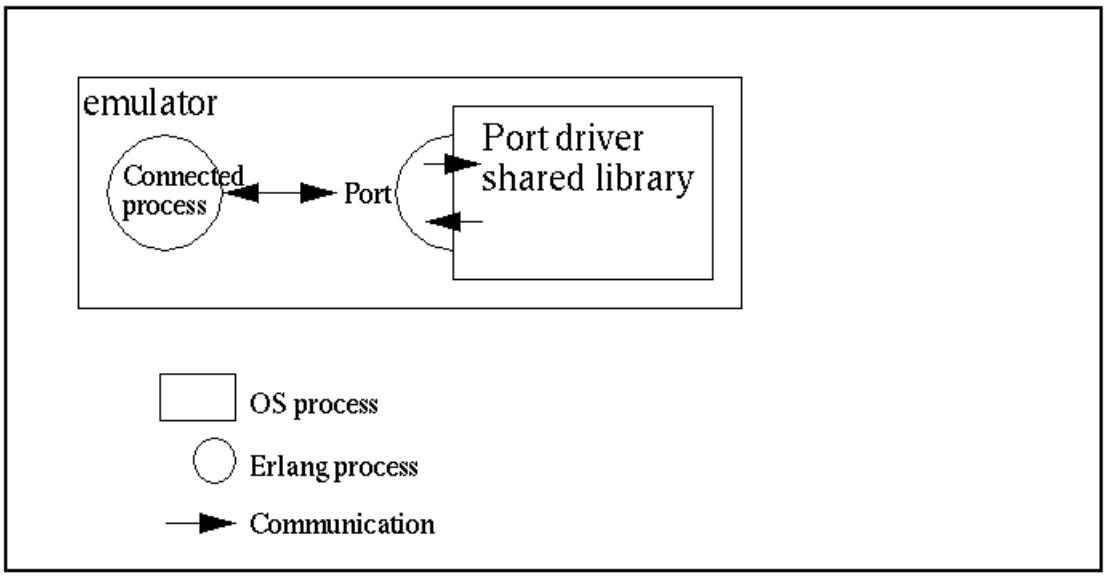

Figure 6.1: Port Driver Communication.

8.6.1 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.

8.6.2 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;
\011{error, already_loaded} -> ok;
\011_ -> 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
\011            {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.
```

8.6.3 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;
}
```

8.6.4 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
```

8.7 C Nodes

This is an example of how to solve the *example problem* 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.

8.7.1 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.
```

8.7.2 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\n\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\n\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\n\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);
      }
    }
  }
}
```

8.7.3 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
```

9 User's Guide

9.1 Overview

The *OTP Design Principles* is a set of principles for how to structure Erlang code in terms of processes, modules and directories.

9.1.1 Supervision Trees

A basic concept in Erlang/OTP is the *supervision tree*. This is a process structuring model based on the idea of *workers* and *supervisors*.

- Workers are processes which perform computations, that is, they do the actual work.
- Supervisors are processes which monitor the behaviour of workers. A supervisor can restart a worker if something goes wrong.
- The supervision tree is a hierarchical arrangement of code into supervisors and workers, making it possible to design and program fault-tolerant software.

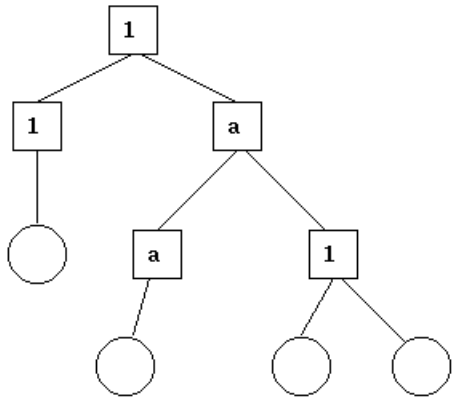

Figure 1.1: Supervision Tree

In the figure above, square boxes represents supervisors and circles represent workers.

9.1.2 Behaviours

In a supervision tree, many of the processes have similar structures, they follow similar patterns. For example, the supervisors are very similar in structure. The only difference between them is which child processes they supervise. Also, many of the workers are servers in a server-client relation, finite state machines, or event handlers such as error loggers.

Behaviours are formalizations of these common patterns. The idea is to divide the code for a process in a generic part (a behaviour module) and a specific part (a *callback module*).

The behaviour module is part of Erlang/OTP. To implement a process such as a supervisor, the user only has to implement the callback module which should export a pre-defined set of functions, the *callback functions*.

An example to illustrate how code can be divided into a generic and a specific part: Consider the following code (written in plain Erlang) for a simple server, which keeps track of a number of "channels". Other processes can allocate and free the channels by calling the functions `alloc/0` and `free/1`, respectively.

```
-module(ch1).
-export([start/0]).
-export([alloc/0, free/1]).
-export([init/0]).

start() ->
    spawn(ch1, init, []).

alloc() ->
    ch1 ! {self(), alloc},
    receive
        {ch1, Res} ->
            Res
    end.

free(Ch) ->
    ch1 ! {free, Ch},
    ok.

init() ->
    register(ch1, self()),
    Chs = channels(),
    loop(Chs).

loop(Chs) ->
    receive
        {From, alloc} ->
            {Ch, Chs2} = alloc(Chs),
            From ! {ch1, Ch},
            loop(Chs2);
        {free, Ch} ->
            Chs2 = free(Ch, Chs),
            loop(Chs2)
    end.
```

The code for the server can be rewritten into a generic part `server.erl`:

```
-module(server).
-export([start/1]).
-export([call/2, cast/2]).
-export([init/1]).

start(Mod) ->
    spawn(server, init, [Mod]).

call(Name, Req) ->
    Name ! {call, self(), Req},
    receive
        {Name, Res} ->
            Res
    end.

cast(Name, Req) ->
    Name ! {cast, Req},
    ok.
```

```
init(Mod) ->
    register(Mod, self()),
    State = Mod:init(),
    loop(Mod, State).

loop(Mod, State) ->
    receive
        {call, From, Req} ->
            {Res, State2} = Mod:handle_call(Req, State),
            From ! {Mod, Res},
            loop(Mod, State2);
        {cast, Req} ->
            State2 = Mod:handle_cast(Req, State),
            loop(Mod, State2)
    end.
```

and a callback module `ch2.erl`:

```
-module(ch2).
-export([start/0]).
-export([alloc/0, free/1]).
-export([init/0, handle_call/2, handle_cast/2]).

start() ->
    server:start(ch2).

alloc() ->
    server:call(ch2, alloc).

free(Ch) ->
    server:cast(ch2, {free, Ch}).

init() ->
    channels().

handle_call(alloc, Chs) ->
    alloc(Chs). % => {Ch,Chs2}

handle_cast({free, Ch}, Chs) ->
    free(Ch, Chs). % => Chs2
```

Note the following:

- The code in `server` can be re-used to build many different servers.
- The name of the server, in this example the atom `ch2`, is hidden from the users of the client functions. This means the name can be changed without affecting them.
- The protcol (messages sent to and received from the server) is hidden as well. This is good programming practice and allows us to change the protocol without making changes to code using the interface functions.
- We can extend the functionality of `server`, without having to change `ch2` or any other callback module.

(In `ch1.erl` and `ch2.erl` above, the implementation of `channels/0`, `alloc/1` and `free/2` has been intentionally left out, as it is not relevant to the example. For completeness, one way to write these functions are given below. Note that this is an example only, a realistic implementation must be able to handle situations like running out of channels to allocate etc.)

```
channels() ->
   {_Allocated = [], _Free = lists:seq(1,100)}.
```

```
alloc({Allocated, [H|T] = _Free}) ->
   {H, {[H|Allocated], T}}.

free(Ch, {Alloc, Free} = Channels) ->
   case lists:member(Ch, Alloc) of
      true ->
         {lists:delete(Ch, Alloc), [Ch|Free]};
      false ->
         Channels
   end.
```

Code written without making use of behaviours may be more efficient, but the increased efficiency will be at the expense of generality. The ability to manage all applications in the system in a consistent manner is very important.

Using behaviours also makes it easier to read and understand code written by other programmers. Ad hoc programming structures, while possibly more efficient, are always more difficult to understand.

The module `server` corresponds, greatly simplified, to the Erlang/OTP behaviour `gen_server`.

The standard Erlang/OTP behaviours are:

gen_server
: For implementing the server of a client-server relation.

gen_fsm
: For implementing finite state machines.

gen_event
: For implementing event handling functionality.

supervisor
: For implementing a supervisor in a supervision tree.

The compiler understands the module attribute `-behaviour(Behaviour)` and issues warnings about missing callback functions. Example:

```
-module(chs3).
-behaviour(gen_server).
...

3> c(chs3).
./chs3.erl:10: Warning: undefined call-back function handle_call/3
{ok,chs3}
```

9.1.3 Applications

Erlang/OTP comes with a number of components, each implementing some specific functionality. Components are with Erlang/OTP terminology called *applications*. Examples of Erlang/OTP applications are Mnesia, which has everything needed for programming database services, and Debugger which is used to debug Erlang programs. The minimal system based on Erlang/OTP consists of the applications Kernel and STDLIB.

The application concept applies both to program structure (processes) and directory structure (modules).

The simplest kind of application does not have any processes, but consists of a collection of functional modules. Such an application is called a *library application*. An example of a library application is STDLIB.

An application with processes is easiest implemented as a supervision tree using the standard behaviours.

How to program applications is described in *Applications*.

9.1.4 Releases

A *release* is a complete system made out from a subset of the Erlang/OTP applications and a set of user-specific applications.

How to program releases is described in *Releases*.

How to install a release in a target environment is described in the chapter about Target Systems in System Principles.

9.1.5 Release Handling

Release handling is upgrading and downgrading between different versions of a release, in a (possibly) running system. How to do this is described in *Release Handling*.

9.2 Gen_Server Behaviour

This chapter should be read in conjunction with *gen_server(3)*, where all interface functions and callback functions are described in detail.

9.2.1 Client-Server Principles

The client-server model is characterized by a central server and an arbitrary number of clients. The client-server model is generally used for resource management operations, where several different clients want to share a common resource. The server is responsible for managing this resource.

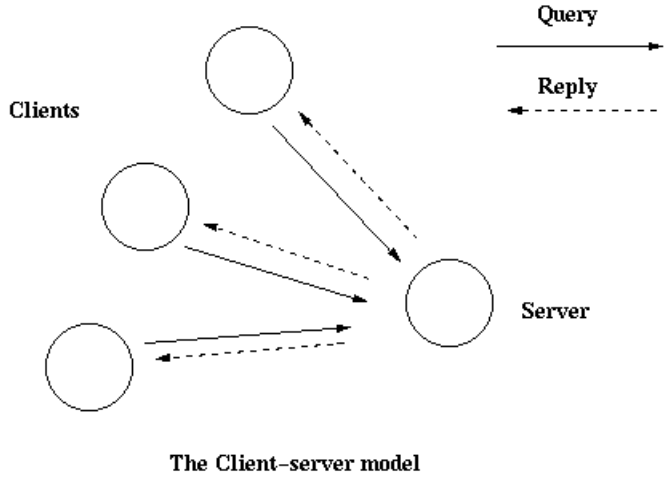

Figure 2.1: Client-Server Model

9.2.2 Example

An example of a simple server written in plain Erlang was given in *Overview*. The server can be re-implemented using `gen_server`, resulting in this callback module:

```
-module(ch3).
-behaviour(gen_server).

-export([start_link/0]).
-export([alloc/0, free/1]).
-export([init/1, handle_call/3, handle_cast/2]).
```

```
start_link() ->
    gen_server:start_link({local, ch3}, ch3, [], []).

alloc() ->
    gen_server:call(ch3, alloc).

free(Ch) ->
    gen_server:cast(ch3, {free, Ch}).

init(_Args) ->
    {ok, channels()}.

handle_call(alloc, _From, Chs) ->
    {Ch, Chs2} = alloc(Chs),
    {reply, Ch, Chs2}.

handle_cast({free, Ch}, Chs) ->
    Chs2 = free(Ch, Chs),
    {noreply, Chs2}.
```

The code is explained in the next sections.

9.2.3 Starting a Gen_Server

In the example in the previous section, the gen_server is started by calling `ch3:start_link()`:

```
start_link() ->
    gen_server:start_link({local, ch3}, ch3, [], []) => {ok, Pid}
```

`start_link` calls the function `gen_server:start_link/4`. This function spawns and links to a new process, a gen_server.

- The first argument `{local, ch3}` specifies the name. In this case, the gen_server will be locally registered as `ch3`.

  If the name is omitted, the gen_server is not registered. Instead its pid must be used. The name could also be given as `{global, Name}`, in which case the gen_server is registered using `global:register_name/2`.
- The second argument, `ch3`, is the name of the callback module, that is the module where the callback functions are located.

  In this case, the interface functions (`start_link`, `alloc` and `free`) are located in the same module as the callback functions (`init`, `handle_call` and `handle_cast`). This is normally good programming practice, to have the code corresponding to one process contained in one module.
- The third argument, [], is a term which is passed as-is to the callback function `init`. Here, `init` does not need any indata and ignores the argument.
- The fourth argument, [], is a list of options. See `gen_server(3)` for available options.

If name registration succeeds, the new gen_server process calls the callback function `ch3:init([])`. `init` is expected to return `{ok, State}`, where `State` is the internal state of the gen_server. In this case, the state is the available channels.

```
init(_Args) ->
    {ok, channels()}.
```

Note that `gen_server:start_link` is synchronous. It does not return until the gen_server has been initialized and is ready to receive requests.

`gen_server:start_link` must be used if the gen_server is part of a supervision tree, i.e. is started by a supervisor. There is another function `gen_server:start` to start a stand-alone gen_server, i.e. a gen_server which is not part of a supervision tree.

9.2.4 Synchronous Requests - Call

The synchronous request `alloc()` is implemented using `gen_server:call/2`:

```
alloc() ->
    gen_server:call(ch3, alloc).
```

`ch3` is the name of the gen_server and must agree with the name used to start it. `alloc` is the actual request.

The request is made into a message and sent to the gen_server. When the request is received, the gen_server calls `handle_call(Request, From, State)` which is expected to return a tuple `{reply, Reply, State1}`. `Reply` is the reply which should be sent back to the client, and `State1` is a new value for the state of the gen_server.

```
handle_call(alloc, _From, Chs) ->
    {Ch, Chs2} = alloc(Chs),
    {reply, Ch, Chs2}.
```

In this case, the reply is the allocated channel `Ch` and the new state is the set of remaining available channels `Chs2`.

Thus, the call `ch3:alloc()` returns the allocated channel `Ch` and the gen_server then waits for new requests, now with an updated list of available channels.

9.2.5 Asynchronous Requests - Cast

The asynchronous request `free(Ch)` is implemented using `gen_server:cast/2`:

```
free(Ch) ->
    gen_server:cast(ch3, {free, Ch}).
```

`ch3` is the name of the gen_server. `{free, Ch}` is the actual request.

The request is made into a message and sent to the gen_server. `cast`, and thus `free`, then returns `ok`.

When the request is received, the gen_server calls `handle_cast(Request, State)` which is expected to return a tuple `{noreply, State1}`. `State1` is a new value for the state of the gen_server.

```
handle_cast({free, Ch}, Chs) ->
    Chs2 = free(Ch, Chs),
    {noreply, Chs2}.
```

In this case, the new state is the updated list of available channels `Chs2`. The gen_server is now ready for new requests.

9.2.6 Stopping

In a Supervision Tree

If the gen_server is part of a supervision tree, no stop function is needed. The gen_server will automatically be terminated by its supervisor. Exactly how this is done is defined by a *shutdown strategy* set in the supervisor.

If it is necessary to clean up before termination, the shutdown strategy must be a timeout value and the gen_server must be set to trap exit signals in the `init` function. When ordered to shutdown, the gen_server will then call the callback function `terminate(shutdown, State)`:

```
init(Args) ->
    ...,
    process_flag(trap_exit, true),
    ...,
    {ok, State}.

...

terminate(shutdown, State) ->
    ..code for cleaning up here..
    ok.
```

Stand-Alone Gen_Servers

If the gen_server is not part of a supervision tree, a stop function may be useful, for example:

```
...
export([stop/0]).
...

stop() ->
    gen_server:cast(ch3, stop).
...

handle_cast(stop, State) ->
    {stop, normal, State};
handle_cast({free, Ch}, State) ->
    ....

...

terminate(normal, State) ->
    ok.
```

The callback function handling the `stop` request returns a tuple `{stop, normal, State1}`, where `normal` specifies that it is a normal termination and `State1` is a new value for the state of the gen_server. This will cause the gen_server to call `terminate(normal,State1)` and then terminate gracefully.

9.2.7 Handling Other Messages

If the gen_server should be able to receive other messages than requests, the callback function `handle_info(Info, State)` must be implemented to handle them. Examples of other messages are exit messages, if the gen_server is linked to other processes (than the supervisor) and trapping exit signals.

```
handle_info({'EXIT', Pid, Reason}, State) ->
```

```
    ..code to handle exits here..
    {noreply, State1}.
```

9.3 Gen_Fsm Behaviour

This chapter should be read in conjunction with `gen_fsm(3)`, where all interface functions and callback functions are described in detail.

9.3.1 Finite State Machines

A finite state machine, FSM, can be described as a set of relations of the form:

```
State(S) x Event(E) -> Actions(A), State(S')
```

These relations are interpreted as meaning:

If we are in state `S` and the event `E` occurs, we should perform the actions `A` and make a transition to the state `S'`.

For an FSM implemented using the `gen_fsm` behaviour, the state transition rules are written as a number of Erlang functions which conform to the following convention:

```
StateName(Event, StateData) ->
    .. code for actions here ...
    {next_state, StateName', StateData'}
```

9.3.2 Example

A door with a code lock could be viewed as an FSM. Initially, the door is locked. Anytime someone presses a button, this generates an event. Depending on what buttons have been pressed before, the sequence so far may be correct, incomplete or wrong.

If it is correct, the door is unlocked for 30 seconds (30000 ms). If it is incomplete, we wait for another button to be pressed. If it is is wrong, we start all over, waiting for a new button sequence.

Implementing the code lock FSM using `gen_fsm` results in this callback module:

```
-module(code_lock).
-behaviour(gen_fsm).

-export([start_link/1]).
-export([button/1]).
-export([init/1, locked/2, open/2]).

start_link(Code) ->
    gen_fsm:start_link({local, code_lock}, code_lock, Code, []).

button(Digit) ->
    gen_fsm:send_event(code_lock, {button, Digit}).

init(Code) ->
    {ok, locked, {[], Code}}.

locked({button, Digit}, {SoFar, Code}) ->
    case [Digit|SoFar] of
        Code ->
```

```
            do_unlock(),
            {next_state, open, {[], Code}, 3000};
        Incomplete when length(Incomplete)<length(Code) ->
            {next_state, locked, {Incomplete, Code}};
        _Wrong ->
            {next_state, locked, {[], Code}}
    end.

open(timeout, State) ->
    do_lock(),
    {next_state, locked, State}.
```

The code is explained in the next sections.

9.3.3 Starting a Gen_Fsm

In the example in the previous section, the gen_fsm is started by calling `code_lock:start_link(Code)`:

```
start_link(Code) ->
    gen_fsm:start_link({local, code_lock}, code_lock, Code, []).
```

`start_link` calls the function `gen_fsm:start_link/4`. This function spawns and links to a new process, a gen_fsm.

- The first argument `{local, code_lock}` specifies the name. In this case, the gen_fsm will be locally registered as `code_lock`.

  If the name is omitted, the gen_fsm is not registered. Instead its pid must be used. The name could also be given as `{global, Name}`, in which case the gen_fsm is registered using `global:register_name/2`.
- The second argument, `code_lock`, is the name of the callback module, that is the module where the callback functions are located.

  In this case, the interface functions (`start_link` and `button`) are located in the same module as the callback functions (`init`, `locked` and `open`). This is normally good programming practice, to have the code corresponding to one process contained in one module.
- The third argument, `Code`, is a term which is passed as-is to the callback function `init`. Here, `init` gets the correct code for the lock as indata.
- The fourth argument, [], is a list of options. See `gen_fsm(3)` for available options.

If name registration succeeds, the new gen_fsm process calls the callback function `code_lock:init(Code)`. This function is expected to return `{ok, StateName, StateData}`, where `StateName` is the name of the initial state of the gen_fsm. In this case `locked`, assuming the door is locked to begin with. `StateData` is the internal state of the gen_fsm. (For gen_fsms, the internal state is often referred to 'state data' to distinguish it from the state as in states of a state machine.) In this case, the state data is the button sequence so far (empty to begin with) and the correct code of the lock.

```
init(Code) ->
    {ok, locked, {[], Code}}.
```

Note that `gen_fsm:start_link` is synchronous. It does not return until the gen_fsm has been initialized and is ready to receive notifications.

`gen_fsm:start_link` must be used if the gen_fsm is part of a supervision tree, i.e. is started by a supervisor. There is another function `gen_fsm:start` to start a stand-alone gen_fsm, i.e. a gen_fsm which is not part of a supervision tree.

9.3.4 Notifying About Events

The function notifying the code lock about a button event is implemented using `gen_fsm:send_event/2`:

```
button(Digit) ->
    gen_fsm:send_event(code_lock, {button, Digit}).
```

`code_lock` is the name of the gen_fsm and must agree with the name used to start it. `{button, Digit}` is the actual event.

The event is made into a message and sent to the gen_fsm. When the event is received, the gen_fsm calls `StateName(Event, StateData)` which is expected to return a tuple `{next_state, StateName1, StateData1}`. `StateName` is the name of the current state and `StateName1` is the name of the next state to go to. `StateData1` is a new value for the state data of the gen_fsm.

```
locked({button, Digit}, {SoFar, Code}) ->
    case [Digit|SoFar] of
        Code ->
            do_unlock(),
            {next_state, open, {[], Code}, 30000};
        Incomplete when length(Incomplete)<length(Code) ->
            {next_state, locked, {Incomplete, Code}};
        _Wrong ->
            {next_state, locked, {[], Code}};
    end.

open(timeout, State) ->
    do_lock(),
    {next_state, locked, State}.
```

If the door is locked and a button is pressed, the complete button sequence so far is compared with the correct code for the lock and, depending on the result, the door is either unlocked and the gen_fsm goes to state `open`, or the door remains in state `locked`.

9.3.5 Timeouts

When a correct code has been givened, the door is unlocked and the following tuple is returned from `locked/2`:

```
{next_state, open, {[], Code}, 30000};
```

30000 is a timeout value in milliseconds. After 30000 ms, i.e. 30 seconds, a timeout occurs. Then `StateName(timeout, StateData)` is called. In this case, the timeout occurs when the door has been in state `open` for 30 seconds. After that the door is locked again:

```
open(timeout, State) ->
    do_lock(),
    {next_state, locked, State}.
```

9.3.6 All State Events

Sometimes an event can arrive at any state of the gen_fsm. Instead of sending the message with `gen_fsm:send_event/2` and writing one clause handling the event for each state function, the message can be sent with `gen_fsm:send_all_state_event/2` and handled with `Module:handle_event/3`:

```
-module(code_lock).
...
-export([stop/0]).
...

stop() ->
    gen_fsm:send_all_state_event(code_lock, stop).

...

handle_event(stop, _StateName, StateData) ->
    {stop, normal, StateData}.
```

9.3.7 Stopping

In a Supervision Tree

If the gen_fsm is part of a supervision tree, no stop function is needed. The gen_fsm will automatically be terminated by its supervisor. Exactly how this is done is defined by a *shutdown strategy* set in the supervisor.

If it is necessary to clean up before termination, the shutdown strategy must be a timeout value and the gen_fsm must be set to trap exit signals in the `init` function. When ordered to shutdown, the gen_fsm will then call the callback function `terminate(shutdown, StateName, StateData)`:

```
init(Args) ->
    ...,
    process_flag(trap_exit, true),
    ...,
    {ok, StateName, StateData}.

...

terminate(shutdown, StateName, StateData) ->
    ..code for cleaning up here..
    ok.
```

Stand-Alone Gen_Fsms

If the gen_fsm is not part of a supervision tree, a stop function may be useful, for example:

```
...
-export([stop/0]).
...

stop() ->
    gen_fsm:send_all_state_event(code_lock, stop).
...

handle_event(stop, _StateName, StateData) ->
    {stop, normal, StateData}.
```

```
...

terminate(normal, _StateName, _StateData) ->
    ok.
```

The callback function handling the `stop` event returns a tuple `{stop,normal,StateData1}`, where `normal` specifies that it is a normal termination and `StateData1` is a new value for the state data of the gen_fsm. This will cause the gen_fsm to call `terminate(normal,StateName,StateData1)` and then terminate gracefully:

9.3.8 Handling Other Messages

If the gen_fsm should be able to receive other messages than events, the callback function `handle_info(Info, StateName, StateData)` must be implemented to handle them. Examples of other messages are exit messages, if the gen_fsm is linked to other processes (than the supervisor) and trapping exit signals.

```
handle_info({'EXIT', Pid, Reason}, StateName, StateData) ->
    ..code to handle exits here..
    {next_state, StateName1, StateData1}.
```

9.4 Gen_Event Behaviour

This chapter should be read in conjunction with `gen_event(3)`, where all interface functions and callback functions are described in detail.

9.4.1 Event Handling Principles

In OTP, an *event manager* is a named object to which events can be sent. An *event* could be, for example, an error, an alarm or some information that should be logged.

In the event manager, zero, one or several *event handlers* are installed. When the event manager is notified about an event, the event will be processed by all the installed event handlers. For example, an event manager for handling errors can by default have a handler installed which writes error messages to the terminal. If the error messages during a certain period should be saved to a file as well, the user adds another event handler which does this. When logging to file is no longer necessary, this event handler is deleted.

An event manager is implemented as a process and each event handler is implemented as a callback module.

The event manager essentially maintains a list of `{Module, State}` pairs, where each `Module` is an event handler, and `State` the internal state of that event handler.

9.4.2 Example

The callback module for the event handler writing error messages to the terminal could look like:

```
-module(terminal_logger).
-behaviour(gen_event).

-export([init/1, handle_event/2, terminate/2]).

init(_Args) ->
    {ok, []}.

handle_event(ErrorMsg, State) ->
    io:format("***Error*** ~p~n", [ErrorMsg]),
```

```
    {ok, State}.

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

The callback module for the event handler writing error messages to a file could look like:

```
-module(file_logger).
-behaviour(gen_event).

-export([init/1, handle_event/2, terminate/2]).

init(File) ->
    {ok, Fd} = file:open(File, read),
    {ok, Fd}.

handle_event(ErrorMsg, Fd) ->
    io:format(Fd, "***Error*** ~p~n", [ErrorMsg]),
    {ok, Fd}.

terminate(_Args, Fd) ->
    file:close(Fd).
```

The code is explained in the next sections.

9.4.3 Starting an Event Manager

To start an event manager for handling errors, as described in the example above, call the following function:

```
gen_event:start_link({local, error_man})
```

This function spawns and links to a new process, an event manager.

The argument, `{local, error_man}` specifies the name. In this case, the event manager will be locally registered as `error_man`.

If the name is omitted, the event manager is not registered. Instead its pid must be used. The name could also be given as `{global, Name}`, in which case the event manager is registered using `global:register_name/2`.

`gen_event:start_link` must be used if the event manager is part of a supervision tree, i.e. is started by a supervisor. There is another function `gen_event:start` to start a stand-alone event manager, i.e. an event manager which is not part of a supervision tree.

9.4.4 Adding an Event Handler

Here is an example using the shell on how to start an event manager and add an event handler to it:

```
1> gen_event:start({local, error_man}).
{ok,<0.31.0>}
2> gen_event:add_handler(error_man, terminal_logger, []).
ok
```

This function sends a message to the event manager registered as `error_man`, telling it to add the event handler `terminal_logger`. The event manager will call the callback function `terminal_logger:init([])`, where

the argument [] is the third argument to `add_handler`. `init` is expected to return `{ok, State}`, where `State` is the internal state of the event handler.

```
init(_Args) ->
    {ok, []}.
```

Here, `init` does not need any input data and ignores its argument. Also, for `terminal_logger` the internal state is not used. For `file_logger`, the internal state is used to save the open file descriptor.

```
init(File) ->
    {ok, Fd} = file:open(File, read),
    {ok, Fd}.
```

9.4.5 Notifying About Events

```
3> gen_event:notify(error_man, no_reply).
***Error*** no_reply
ok
```

`error_man` is the name of the event manager and `no_reply` is the event.

The event is made into a message and sent to the event manager. When the event is received, the event manager calls `handle_event(Event, State)` for each installed event handler, in the same order as they were added. The function is expected to return a tuple `{ok, State1}`, where `State1` is a new value for the state of the event handler.

In `terminal_logger`:

```
handle_event(ErrorMsg, State) ->
    io:format("***Error*** ~p~n", [ErrorMsg]),
    {ok, State}.
```

In `file_logger`:

```
handle_event(ErrorMsg, Fd) ->
    io:format(Fd, "***Error*** ~p~n", [ErrorMsg]),
    {ok, Fd}.
```

9.4.6 Deleting an Event Handler

```
4> gen_event:delete_handler(error_man, terminal_logger, []).
ok
```

This function sends a message to the event manager registered as `error_man`, telling it to delete the event handler `terminal_logger`. The event manager will call the callback function `terminal_logger:terminate([], State)`, where the argument [] is the third argument to `delete_handler`. `terminate` should be the opposite of `init` and do any necessary cleaning up. Its return value is ignored.

For `terminal_logger`, no cleaning up is necessary:

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

For `file_logger`, the file descriptor opened in `init` needs to be closed:

```
terminate(_Args, Fd) ->
    file:close(Fd).
```

9.4.7 Stopping

When an event manager is stopped, it will give each of the installed event handlers the chance to clean up by calling `terminate/2`, the same way as when deleting a handler.

In a Supervision Tree

If the event manager is part of a supervision tree, no stop function is needed. The event manager will automatically be terminated by its supervisor. Exactly how this is done is defined by a *shutdown strategy* set in the supervisor.

Stand-Alone Event Managers

An event manager can also be stopped by calling:

```
> gen_event:stop(error_man).
ok
```

9.5 Supervisor Behaviour

This section should be read in conjunction with `supervisor(3)`, where all details about the supervisor behaviour is given.

9.5.1 Supervision Principles

A supervisor is responsible for starting, stopping and monitoring its child processes. The basic idea of a supervisor is that it should keep its child processes alive by restarting them when necessary.

Which child processes to start and monitor is specified by a list of *child specifications*. The child processes are started in the order specified by this list, and terminated in the reversed order.

9.5.2 Example

The callback module for a supervisor starting the server from the *gen_server chapter* could look like this:

```
-module(ch_sup).
-behaviour(supervisor).

-export([start_link/0]).
-export([init/1]).

start_link() ->
    supervisor:start_link(ch_sup, []).
```

```
init(_Args) ->
    {ok, {{one_for_one, 1, 60},
          [{ch3, {ch3, start_link, []},
            permanent, brutal_kill, worker, [ch3]}]}}.
```

`one_for_one` is the *restart strategy*.

1 and 60 defines the *maximum restart frequency*.

The tuple `{ch3, ...}` is a *child specification*.

9.5.3 Restart Strategy

one_for_one

If a child process terminates, only that process is restarted.

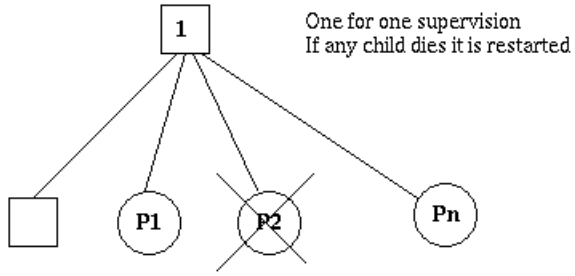

Figure 5.1: One_For_One Supervision

one_for_all

If a child process terminates, all other child processes are terminated and then all child processes, including the terminated one, are restarted.

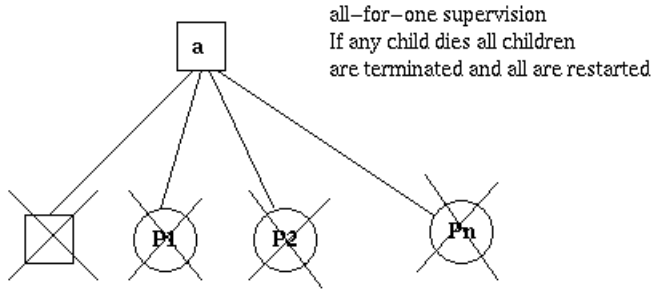

Figure 5.2: One_For_All Supervision

rest_for_one

If a child process terminates, the 'rest' of the child processes -- i.e. the child processes after the terminated process in start order -- are terminated. Then the terminated child process and the rest of the child processes are restarted.

9.5.4 Maximum Restart Frequency

The supervisors have a built-in mechanism to limit the number of restarts which can occur in a given time interval. This is determined by the values of the two parameters `MaxR` and `MaxT` in the start specification returned by the callback function `init`:

```
init(...) ->
    {ok, {{RestartStrategy, MaxR, MaxT},
          [ChildSpec, ...]}}.
```

If more than `MaxR` number of restarts occur in the last `MaxT` seconds, then the supervisor terminates all the child processes and then itself.

When the supervisor terminates, then the next higher level supervisor takes some action. It either restarts the terminated supervisor, or terminates itself.

The intention of the restart mechanism is to prevent a situation where a process repeatedly dies for the same reason, only to be restarted again.

9.5.5 Child Specification

This is the type definition for a child specification:

```
{Id, StartFunc, Restart, Shutdown, Type, Modules}
    Id = term()
    StartFunc = {M, F, A}
        M = F = atom()
        A = [term()]
    Restart = permanent | transient | temporary
    Shutdown = brutal_kill | integer() &gt;=0 | infinity
    Type = worker | supervisor
    Modules = [Module] | dynamic
        Module = atom()
```

- `Id` is a name that is used to identify the child specification internally by the supervisor.
- `StartFunc` defines the function call used to start the child process. It is a module-function-arguments tuple used as `apply(M, F, A)`.

  It should be (or result in) a call to `supervisor:start_link`, `gen_server:start_link`, `gen_fsm:start_link` or `gen_event:start_link`. (Or a function compliant with these functions, see `supervisor(3)` for details.
- `Restart` defines when a terminated child process should be restarted.
  - A `permanent` child process is always restarted.
  - A `temporary` child process is never restarted.
  - A `transient` child process is restarted only if it terminates abnormally, i.e. with another exit reason than `normal`.
- `Shutdown` defines how a child process should be terminated.
  - `brutal_kill` means the child process is unconditionally terminated using `exit(Child, kill)`.
  - An integer timeout value means that the supervisor tells the child process to terminate by calling `exit(Child, shutdown)` and then waits for an exit signal back. If no exit signal is received within the specified time, the child process is unconditionally terminated using `exit(Child, kill)`.

  - If the child process is another supervisor, it should be set to `infinity` to give the subtree enough time to shutdown.
- `Type` specifies if the child process is a supervisor or a worker.
- `Modules` should be a list with one element `[Module]`, where `Module` is the name of the callback module, if the child process is a supervisor, gen_server or gen_fsm. If the child process is a gen_event, `Modules` should be `dynamic`.

  This information is used by the release handler during upgrades and downgrades, see *Release Handling*.

Example: The child specification to start the server `ch3` in the example above looks like:

```
{ch3,
 {ch3, start_link, []},
 permanent, brutal_kill, worker, [ch3]}
```

Example: A child specification to start the event manager from the chapter about *gen_event*:

```
{error_man,
 {gen_event, start_link, [{local, error_man}]},
 permanent, 5000, worker, dynamic}
```

Both the server and event manager are registered processes which can be expected to be accessible at all times, thus they are specified to be `permanent`.

`ch3` does not need to do any cleaning up before termination, thus no shutdown time is needed but `brutal_kill` should be sufficient. `error_man` may need some time for the event handlers to clean up, thus `Shutdown` is set to 5000 ms.

Example: A child specification to start another supervisor:

```
{sup,
 {sup, start_link, []},
 transient, infinity, supervisor, [sup]}
```

9.5.6 Starting a Supervisor

In the example above, the supervisor is started by calling `ch_sup:start_link()`:

```
start_link() ->
    supervisor:start_link(ch_sup, []).
```

`ch_sup:start_link` calls the function `supervisor:start_link/2`. This function spawns and links to a new process, a supervisor.

- The first argument, `ch_sup`, is the name of the callback module, that is the module where the `init` callback function is located.
- The second argument, [], is a term which is passed as-is to the callback function `init`. Here, `init` does not need any indata and ignores the argument.

In this case, the supervisor is not registered. Instead its pid must be used. A name can be specified by calling `supervisor:start_link({local, Name}, Module, Args)` or `supervisor:start_link({global, Name}, Module, Args)`.

The new supervisor process calls the callback function `ch_sup:init([])`. `init` is expected to return `{ok, StartSpec}`:

```
init(_Args) ->
    {ok, {{one_for_one, 1, 60},
          [{ch3, {ch3, start_link, []},
            permanent, brutal_kill, worker, [ch3]}]}}.
```

The supervisor then starts all its child processes according to the child specifications in the start specification. In this case there is one child process, `ch3`.

Note that `supervisor:start_link` is synchronous. It does not return until all child processes have been started.

9.5.7 Adding a Child Process

In addition to the static supervision tree, we can also add dynamic child processes to an existing supervisor with the following call:

```
supervisor:start_child(Sup, ChildSpec)
```

`Sup` is the pid, or name, of the supervisor. `ChildSpec` is a *child specification*.

Child processes added using `start_child/2` behave in the same manner as the other child processes, with the following important exception: If a supervisor dies and is re-created, then all child processes which were dynamically added to the supervisor will be lost.

9.5.8 Stopping a Child Process

Any child process, static or dynamic, can be stopped in accordance with the shutdown specification:

```
supervisor:terminate_child(Sup, Id)
```

The child specification for a stopped child process is deleted with the following call:

```
supervisor:delete_child(Sup, Id)
```

`Sup` is the pid, or name, of the supervisor. `Id` is the id specified in the *child specification*.

As with dynamically added child processes, the effects of deleting a static child process is lost if the supervisor itself restarts.

9.5.9 Simple-One-For-One Supervisors

A supervisor with restart strategy `simple_one_for_one` is a simplified one_for_one supervisor, where all child processes are dynamically added instances of the same process.

Example of a callback module for a simple_one_for_one supervisor:

```
-module(simple_sup).
-behaviour(supervisor).

-export([start_link/0]).
-export([init/1]).

start_link() ->
    supervisor:start_link(simple_sup, []).

init(_Args) ->
    {ok, {{simple_one_for_one, 0, 1},
          [{call, {call, start_link, []},
            temporary, brutal_kill, worker, [call]}]}}.
```

When started, the supervisor will not start any child processes. Instead, all child processes are added dynamically by calling:

```
supervisor:start_child(Sup, List)
```

`Sup` is the pid, or name, of the supervisor. `List` is an arbitrary list of terms which will be added to the list of arguments specified in the child specification. If the start function is specified as `{M, F, A}`, then the child process is started by calling `apply(M, F, A++List)`.

For example, adding a child to `simple_sup` above:

```
supervisor:start_child(Pid, [id1])
```

results in the child process being started by calling `apply(call, start_link, []++[id1])`, or actually:

```
call:start_link(id1)
```

9.5.10 Stopping

Since the supervisor is part of a supervision tree, it will automatically be terminated by its supervisor. When asked to shutdown, it will terminate all child processes in reversed start order according to the respective shutdown specifications, and then terminate itself.

9.6 Sys and Proc_Lib

The module `sys` contains functions for simple debugging of processes implemented using behaviours.

There are also functions that, together with functions in the module `proc_lib`, can be used to implement a *special process*, a process which comply to the OTP design principles without making use of a standard behaviour. They can also be used to implement user defined (non-standard) behaviours.

Both `sys` and `proc_lib` belong to the STDLIB application.

9.6.1 Simple Debugging

The module `sys` contains some functions for simple debugging of processes implemented using behaviours. We use the `code_lock` example from the *gen_event* chapter to illustrate this:

```
% erl
Erlang (BEAM) emulator version 5.2.3.6 [hipe] [threads:0]

Eshell V5.2.3.6  (abort with ^G)
1> code_lock:start_link([1,2,3,4]).
{ok,<0.32.0>}
2> sys:statistics(code_lock, true).
ok
3> sys:trace(code_lock, true).
ok
4> code_lock:button(4).
*DBG* code_lock got event {button,4} in state closed
ok
*DBG* code_lock switched to state closed
5> code_lock:button(3).
*DBG* code_lock got event {button,3} in state closed
ok
*DBG* code_lock switched to state closed
6> code_lock:button(2).
*DBG* code_lock got event {button,2} in state closed
ok
*DBG* code_lock switched to state closed
7> code_lock:button(1).
*DBG* code_lock got event {button,1} in state closed
ok
OPEN DOOR
*DBG* code_lock switched to state open
*DBG* code_lock got event timeout in state open
CLOSE DOOR
*DBG* code_lock switched to state closed
8> sys:statistics(code_lock, get).
{ok,[{start_time,{{2003,6,12},{14,11,40}}},
     {current_time,{{2003,6,12},{14,12,14}}},
     {reductions,333},
     {messages_in,5},
     {messages_out,0}]}
9> sys:statistics(code_lock, false).
ok
10> sys:trace(code_lock, false).
ok
11> sys:get_status(code_lock).
{status,<0.32.0>,
        {module,gen_fsm},
        [[{'$ancestors',[<0.30.0>]},
          {'$initial_call',{gen,init_it,
                                [gen_fsm,<0.30.0>,<0.30.0>,
                                 {local,code_lock},
                                 code_lock,
                                 [1,2,3,4],
                                 []]}}],
         running,<0.30.0>,[],
         [code_lock,closed,{[],[1,2,3,4]},code_lock,infinity]]}
```

9.6.2 Special Processes

This section describes how to write a process which comply to the OTP design principles, without making use of a standard behaviour. Such a process should:

- be started in a way that makes the process fit into a supervision tree,
- support the `sys`*debug facilities*, and
- take care of *system messages*.

System messages are messages with special meaning, used in the supervision tree. Typical system messages are requests for trace output, and requests to suspend or resume process execution (used during release handling). Processes implemented using standard behaviours automatically understand these messages.

Example

The simple server from the *Overview* chapter, implemented using `sys` and `proc_lib` so it fits into a supervision tree:

```
-module(ch4).
-export([start_link/0]).
-export([alloc/0, free/1]).
-export([init/1]).
-export([system_continue/3, system_terminate/4,
         write_debug/3]).

start_link() ->
    proc_lib:start_link(ch4, init, [self()]).

alloc() ->
    ch4 ! {self(), alloc},
    receive
        {ch4, Res} ->
            Res
    end.

free(Ch) ->
    ch4 ! {free, Ch},
    ok.

init(Parent) ->
    register(ch4, self()),
    Chs = channels(),
    Deb = sys:debug_options([]),
    proc_lib:init_ack(Parent, {ok, self()}),
    loop(Chs, Parent, Deb).

loop(Chs, Parent, Deb) ->
    receive
        {From, alloc} ->
            Deb2 = sys:handle_debug(Deb, {ch4, write_debug},
                                    ch4, {in, alloc, From}),
            {Ch, Chs2} = alloc(Chs),
            From ! {ch4, Ch},
            Deb3 = sys:handle_debug(Deb2, {ch4, write_debug},
                                    ch4, {out, {ch4, Ch}, From}),
            loop(Chs2, Parent, Deb3);
        {free, Ch} ->
            Deb2 = sys:handle_debug(Deb, {ch4, write_debug},
                                    ch4, {in, {free, Ch}}),
            Chs2 = free(Ch, Chs),
            loop(Chs2, Parent, Deb2);

        {system, From, Request} ->
            sys:handle_system_msg(Request, From, Parent,
                                  ch4, Deb, Chs)
    end.

system_continue(Parent, Deb, Chs) ->
    loop(Chs, Parent, Deb).

system_terminate(Reason, Parent, Deb, Chs) ->
    exit(Reason).
```

```
write_debug(Dev, Event, Name) ->
    io:format(Dev, "~p event = ~p~n", [Name, Event]).
```

Example on how the simple debugging functions in `sys` can be used for `ch4` as well:

```
% erl
Erlang (BEAM) emulator version 5.2.3.6 [hipe] [threads:0]

Eshell V5.2.3.6  (abort with ^G)
1> ch4:start_link().
{ok,<0.30.0>}
2> sys:statistics(ch4, true).
ok
3> sys:trace(ch4, true).
ok
4> ch4:alloc().
ch4 event = {in,alloc,<0.25.0>}
ch4 event = {out,{ch4,ch1},<0.25.0>}
ch1
5> ch4:free(ch1).
ch4 event = {in,{free,ch1}}
ok
6> sys:statistics(ch4, get).
{ok,[{start_time,{{2003,6,13},{9,47,5}}},
     {current_time,{{2003,6,13},{9,47,56}}},
     {reductions,109},
     {messages_in,2},
     {messages_out,1}]}
7> sys:statistics(ch4, false).
ok
8> sys:trace(ch4, false).
ok
9> sys:get_status(ch4).
{status,<0.30.0>,
        {module,ch4},
        [[{'$ancestors',[<0.25.0>]},{'$initial_call',{ch4,init,[<0.25.0>]}}],
         running,<0.25.0>,[],
         [ch1,ch2,ch3]]}
```

Starting the Process

A function in the `proc_lib` module should be used to start the process. There are several possible functions, for example `spawn_link/3,4` for asynchronous start and `start_link/3,4,5` for synchronous start.

A process started using one of these functions will store information that is needed for a process in a supervision tree, for example about the ancestors and initial call.

Also, if the process terminates with another reason than `normal` or `shutdown`, a crash report (see SASL User's Guide) is generated.

In the example, synchronous start is used. The process is started by calling `ch4:start_link()`:

```
start_link() ->
    proc_lib:start_link(ch4, init, [self()]).
```

`ch4:start_link` calls the function `proc_lib:start_link`. This function takes a module name, a function name and an argument list as arguments and spawns and links to a new process. The new process starts by executing

the given function, in this case `ch4:init(Pid)`, where `Pid` is the pid (`self()`) of the first process, that is the parent process.

In `init`, all initialization including name registration is done. The new process must also acknowledge that it has been started to the parent:

```
init(Parent) ->
    ...
    proc_lib:init_ack(Parent, {ok, self()}),
    loop(...).
```

`proc_lib:start_link` is synchronous and does not return until `proc_lib:init_ack` has been called.

Debugging

To support the debug facilites in `sys`, we need a *debug structure*, a term `Deb` which is initialized using `sys:debug_options/1`:

```
init(Parent) ->
    ...
    Deb = sys:debug_options([]),
    ...
    loop(Chs, Parent, Deb).
```

`sys:debug_options/1` takes a list of options as argument. Here the list is empty, which means no debugging is enabled initially. See `sys(3)` for information about possible options.

Then for each *system event* that we want to be logged or traced, the following function should be called.

```
sys:handle_debug(Deb, Func, Info, Event) => Deb1
```

- `Deb` is the debug structure.
- `Func` is a tuple `{Module, Name}` (or a fun) and should specify a (user defined) function used to format trace output. For each system event, the format function is called as `Module:Name(Dev, Event, Info)`, where:
  - `Dev` is the IO device to which the output should be printed. See `io(3)`.
  - `Event` and `Info` are passed as-is from `handle_debug`.
- `Info` is used to pass additional information to `Func`, it can be any term and is passed as-is.
- `Event` is the system event. It is up to the user to define what a system event is and how it should be represented, but typically at least incoming and outgoing messages are considered system events and represented by the tuples `{in,Msg[,From]}` and `{out,Msg,To}`, respectively.

`handle_debug` returns an updated debug structure `Deb1`.

In the example, `handle_debug` is called for each incoming and outgoing message. The format function `Func` is the function `ch4:write_debug/3` which prints the message using `io:format/3`.

```
loop(Chs, Parent, Deb) ->
    receive
        {From, alloc} ->
            Deb2 = sys:handle_debug(Deb, {ch4, write_debug},
                                    ch4, {in, alloc, From}),
            {Ch, Chs2} = alloc(Chs),
```

```
            From ! {ch4, Ch},
            Deb3 = sys:handle_debug(Deb2, {ch4, write_debug},
                                    ch4, {out, {ch4, Ch}, From}),
            loop(Chs2, Parent, Deb3);
        {free, Ch} ->
            Deb2 = sys:handle_debug(Deb, {ch4, write_debug},
                                    ch4, {in, {free, Ch}}),
            Chs2 = free(Ch, Chs),
            loop(Chs2, Parent, Deb2);
        ...
    end.

write_debug(Dev, Event, Name) ->
    io:format(Dev, "~p event = ~p~n", [Name, Event]).
```

Handling System Messages

System messages are received as:

```
{system, From, Request}
```

The content and meaning of these messages do not need to be interpreted by the process. Instead the following function should be called:

```
sys:handle_system_msg(Request, From, Parent, Module, Deb, State)
```

This function does not return. It will handle the system message and then call:

```
Module:system_continue(Parent, Deb, State)
```

if process execution should continue, or:

```
Module:system_terminate(Reason, Parent, Deb, State)
```

if the process should terminate. Note that a process in a supervision tree is expected to terminate with the same reason as its parent.

- `Request` and `From` should be passed as-is from the system message to the call to `handle_system_msg`.
- `Parent` is the pid of the parent.
- `Module` is the name of the module.
- `Deb` is the debug structure.
- `State` is a term describing the internal state and is passed to `system_continue`/`system_terminate`.

In the example:

```
loop(Chs, Parent, Deb) ->
    receive
        ...

        {system, From, Request} ->
```

```
            sys:handle_system_msg(Request, From, Parent,
                                  ch4, Deb, Chs)
    end.

system_continue(Parent, Deb, Chs) ->
    loop(Chs, Parent, Deb).

system_terminate(Reason, Parent, Deb, Chs) ->
    exit(Reason).
```

If the special process is set to trap exits, note that if the parent process terminates, the expected behavior is to terminate with the same reason:

```
init(...) ->
    ...,
    process_flag(trap_exit, true),
    ...,
    loop(...).

loop(...) ->
    receive
        ...

        {'EXIT', Parent, Reason} ->
            ..maybe some cleaning up here..
            exit(Reason);
        ...
    end.
```

9.6.3 User-Defined Behaviours

To implement a user-defined behaviour, write code similar to code for a special process but calling functions in a callback module for handling specific tasks.

If it is desired that the compiler should warn for missing callback functions, as it does for the OTP behaviours, implement and export the function:

```
behaviour_info(callbacks) ->
    [{Name1,Arity1},...,{NameN,ArityN}].
```

where each `{Name,Arity}` specifies the name and arity of a callback function.

When the compiler encounters the module attribute `-behaviour(Behaviour).` in a module `Mod`, it will call `Behaviour:behaviour_info(callbacks)` and compare the result with the set of functions actually exported from `Mod`, and issue a warning if any callback function is missing.

Example:

```
%% User-defined behaviour module
-module(simple_server).
-export([start_link/2,...]).
-export([behaviour_info/1]).

behaviour_info(callbacks) ->
    [{init,1},
     {handle_req,1},
```

```
     {terminate,0}].

start_link(Name, Module) ->
    proc_lib:start_link(?MODULE, init, [self(), Name, Module]).

init(Parent, Name, Module) ->
    register(Name, self()),
    ...,
    Dbg = sys:debug_options([]),
    proc_lib:init_ack(Parent, {ok, self()}),
    loop(Parent, Module, Deb, ...).

...
```

In a callback module:

```
-module(db).
-behaviour(simple_server).

-export([init/0, handle_req/1, terminate/0]).

...
```

9.7 Applications

This chapter should be read in conjunction with `app(4)` and `application(3)`.

9.7.1 Application Concept

When we have written code implementing some specific functionality, we might want to make the code into an *application*, that is a component that can be started and stopped as a unit, and which can be re-used in other systems as well.

To do this, we create an *application callback module*, where we describe how the application should be started and stopped.

Then, an *application specification* is needed, which is put in an *application resource file*. Among other things, we specify which modules the application consists of and the name of the callback module.

If we use `systools`, the Erlang/OTP tools for packaging code (see *Releases*), the code for each application is placed in a separate directory following a pre-defined *directory structure*.

9.7.2 Application Callback Module

How to start and stop the code for the application, i.e. the supervision tree, is described by two callback functions:

```
start(StartType, StartArgs) -> {ok, Pid} | {ok, Pid, State}
stop(State)
```

`start` is called when starting the application and should create the supervision tree by starting the top supervisor. It is expected to return the pid of the top supervisor and an optional term `State`, which defaults to []. This term is passed as-is to `stop`.

`StartType` is usually the atom `normal`. It has other values only in the case of a takeover or failover, see *Distributed Applications*. `StartArgs` is defined by the key `mod` in the *application resource file* file.

`stop/1` is called *after* the application has been stopped and should do any necessary cleaning up. Note that the actual stopping of the application, that is the shutdown of the supervision tree, is handled automatically as described in *Starting and Stopping Applications*.

Example of an application callback module for packaging the supervision tree from the *Supervisor* chapter:

```
-module(ch_app).
-behaviour(application).

-export([start/2, stop/1]).

start(_Type, _Args) ->
    ch_sup:start_link().

stop(_State) ->
    ok.
```

A library application, which can not be started or stopped, does not need any application callback module.

9.7.3 Application Resource File

To define an application, we create an *application specification* which is put in an *application resource file*, or in short `.app` file:

```
{application, Application, [Opt1,...,OptN]}.
```

`Application`, an atom, is the name of the application. The file must be named `Application.app`.

Each `Opt` is a tuple `{Key, Value}` which define a certain property of the application. All keys are optional. Default values are used for any omitted keys.

The contents of a minimal `.app` file for a library application `libapp` looks like this:

```
{application, libapp, []}.
```

The contents of a minimal `.app` file `ch_app.app` for a supervision tree application like `ch_app` looks like this:

```
{application, ch_app,
 [{mod, {ch_app,[]}}]}.
```

The key `mod` defines the callback module and start argument of the application, in this case `ch_app` and [], respectively. This means that

```
ch_app:start(normal, [])
```

will be called when the application should be started and

```
ch_app:stop([])
```

will be called when the application has been stopped.

When using `systools`, the Erlang/OTP tools for packaging code (see *Releases*), the keys `description`, `vsn`, `modules`, `registered` and `applications` should also be specified:

```
{application, ch_app,
 [{description, "Channel allocator"},
  {vsn, "1"},
  {modules, [ch_app, ch_sup, ch3]},
  {registered, [ch3]},
  {applications, [kernel, stdlib, sasl]},
  {mod, {ch_app,[]}}
 ]}.
```

`description`
: A short description, a string. Defaults to "".

`vsn`
: Version number, a string. Defaults to "".

`modules`
: All modules *introduced* by this application. `systools` uses this list when generating boot scripts and tar files. A module must be defined in one and only one application. Defaults to [].

`registered`
: All names of registered processes in the application. `systools` uses this list to detect name clashes between applications. Defaults to [].

`applications`
: All applications which must be started before this application is started. `systools` uses this list to generate correct boot scripts. Defaults to [], but note that all applications have dependencies to at least `kernel` and `stdlib`.

The syntax and contents of of the application resource file are described in detail in `app(4)`.

9.7.4 Directory Structure

When packaging code using `systools`, the code for each application is placed in a separate directory `lib/Application-Vsn`, where `Vsn` is the version number.

This may be useful to know, even if `systools` is not used, since Erlang/OTP itself is packaged according to the OTP principles and thus comes with this directory structure. The code server (see `code(3)`) will automatically use code from the directory with the highest version number, if there are more than one version of an application present.

The application directory structure can of course be used in the development environment as well. The version number may then be omitted from the name.

The application directory have the following sub-directories:

- `src`
- `ebin`
- `priv`
- `include`

`src`
: Contains the Erlang source code.

`ebin`
: Contains the Erlang object code, the `beam` files. The `.app` file is also placed here.

`priv`
: Used for application specific files. For example, C executables are placed here. The function `code:priv_dir/1` should be used to access this directory.

`include`
: Used for include files.

9.7.5 Application Controller

When an Erlang runtime system is started, a number of processes are started as part of the Kernel application. One of these processes is the *application controller* process, registered as `application_controller`.

All operations on applications are coordinated by the application controller. It is interfaced through the functions in the module `application`, see `application(3)`. In particular, applications can be loaded, unloaded, started and stopped.

9.7.6 Loading and Unloading Applications

Before an application can be started, it must be *loaded*. The application controller reads and stores the information from the `.app` file.

```
1> application:load(ch_app).
ok
2> application:loaded_applications().
[{kernel,"ERTS  CXC 138 10","2.8.1.3"},
 {stdlib,"ERTS  CXC 138 10","1.11.4.3"},
 {ch_app,"Channel allocator","1"}]
```

An application that has been stopped, or has never been started, can be unloaded. The information about the application is erased from the internal database of the application controller.

```
3> application:unload(ch_app).
ok
4> application:loaded_applications().
[{kernel,"ERTS  CXC 138 10","2.8.1.3"},
 {stdlib,"ERTS  CXC 138 10","1.11.4.3"}]
```

> Note:
>
> Loading/unloading an application does not load/unload the code used by the application. Code loading is done the usual way.

9.7.7 Starting and Stopping Applications

An application is started by calling:

```
5> application:start(ch_app).
ok
6> application:which_applications().
[{kernel,"ERTS  CXC 138 10","2.8.1.3"},
 {stdlib,"ERTS  CXC 138 10","1.11.4.3"},
```

```
 {ch_app,"Channel allocator","1"}]
```

If the application is not already loaded, the application controller will first load it using `application:load/1`. It will check the value of the `applications` key, to ensure that all applications that should be started before this application are running.

The application controller then creates an *application master* for the application. The application master is the group leader of all the processes in the application. The application master starts the application by calling the application callback function `start/2` in the module, and with the start argument, defined by the `mod` key in the `.app` file.

An application is stopped, but not unloaded, by calling:

```
7> application:stop(ch_app).
ok
```

The application master stops the application by telling the top supervisor to shutdown. The top supervisor tells all its child processes to shutdown etc. and the entire tree is terminated in reversed start order. The application master then calls the application callback function `stop/1` in the module defined by the `mod` key.

9.7.8 Configuring an Application

An application can be configured using *configuration parameters*. These are a list of `{Par, Val}` tuples specified by a key `env` in the `.app` file.

```
{application, ch_app,
 [{description, "Channel allocator"},
  {vsn, "1"},
  {modules, [ch_app, ch_sup, ch3]},
  {registered, [ch3]},
  {applications, [kernel, stdlib, sasl]},
  {mod, {ch_app,[]}},
  {env, [{file, "/usr/local/log"}]}
 ]}.
```

`Par` should be an atom, `Val` is any term. The application can retrieve the value of a configuration parameter by calling `application:get_env(App, Par)` or a number of similar functions, see `application(3)`.

Example:

```
% erl
Erlang (BEAM) emulator version 5.2.3.6 [hipe] [threads:0]

Eshell V5.2.3.6  (abort with ^G)
1> application:start(ch_app).
ok
2> application:get_env(ch_app, file).
{ok,"/usr/local/log"}
```

The values in the `.app` file can be overridden by values in a *system configuration file*. This is a file which contains configuration parameters for relevant applications:

```
[{Application1, [{Par11,Val11},...]},
 ...,
```

```
{ApplicationN, [{ParN1,ValN1},...]}].
```

The system configuration should be called `Name.config` and Erlang should be started with the command line argument `-config Name`. See `config(4)` for more information.

Example: A file `test.config` is created with the following contents:

```
[{ch_app, [{file, "testlog"}]}].
```

The value of `file` will override the value of `file` as defined in the `.app` file:

```
% erl -config test
Erlang (BEAM) emulator version 5.2.3.6 [hipe] [threads:0]

Eshell V5.2.3.6  (abort with ^G)
1> application:start(ch_app).
ok
2> application:get_env(ch_app, file).
{ok,"testlog"}
```

If *release handling* is used, exactly one system configuration file should be used and that file should be called `sys.config`

The values in the `.app` file, as well as the values in a system configuration file, can be overridden directly from the command line:

```
% erl -ApplName Par1 Val1 ... ParN ValN
```

Example:

```
% erl -ch_app file '"testlog"'
Erlang (BEAM) emulator version 5.2.3.6 [hipe] [threads:0]

Eshell V5.2.3.6  (abort with ^G)
1> application:start(ch_app).
ok
2> application:get_env(ch_app, file).
{ok,"testlog"}
```

9.7.9 Application Start Types

A *start type* is defined when starting the application:

```
application:start(Application, Type)
```

`application:start(Application)` is the same as calling `application:start(Application, temporary)`. The type can also be `permanent` or `transient`:

- If a permanent application terminates, all other applications and the runtime system are also terminated.

- If a transient application terminates with reason `normal`, this is reported but no other applications are terminated. If a transient application terminates abnormally, that is with any other reason than `normal`, all other applications and the runtime system are also terminated.
- If a temporary application terminates, this is reported but no other applications are terminated.

It is always possible to stop an application explicitly by calling `application:stop/1`. Regardless of the mode, no other applications will be affected.

Note that transient mode is of little practical use, since when a supervision tree terminates, the reason is set to `shutdown`, not `normal`.

9.8 Included Applications

9.8.1 Definition

An application can *include* other applications. An *included application* has its own application directory and `.app` file, but it is started as part of the supervisor tree of another application.

An application can only be included by one other application.

An included application can include other applications.

An application which is not included by any other application is called a *primary application*.

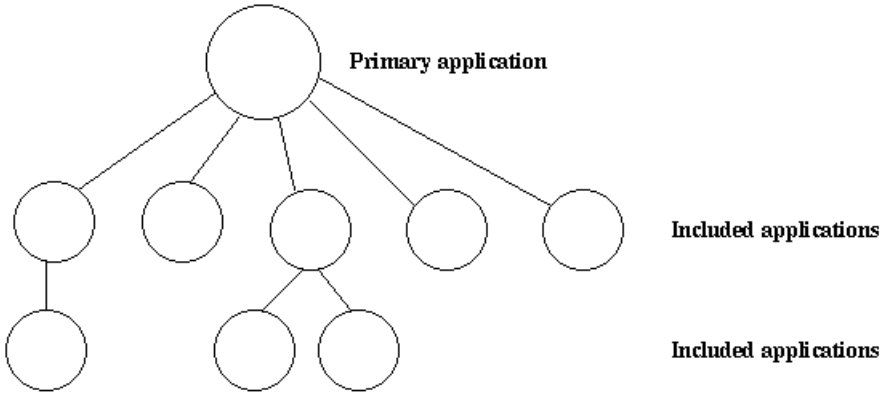

Figure 8.1: Primary Application and Included Applications.

The application controller will automatically load any included applications when loading a primary application, but not start them. Instead, the top supervisor of the included application must be started by a supervisor in the including application.

This means that when running, an included application is in fact part of the primary application and a process in an included application will consider itself belonging to the primary application.

9.8.2 Specifying Included Applications

Which applications to include is defined by the `included_applications` key in the `.app` file.

```
{application, prim_app,
 [{description, "Tree application"},
  {vsn, "1"},
  {modules, [prim_app_cb, prim_app_sup, prim_app_server]},
```

```
 {registered, [prim_app_server]},
 {included_applications, [incl_app]},
 {applications, [kernel, stdlib, sasl]},
 {mod, {prim_app_cb,[]}},
 {env, [{file, "/usr/local/log"}]}
]}.
```

9.8.3 Synchronizing Processes During Startup

The supervisor tree of an included application is started as part of the supervisor tree of the including application. If there is a need for synchronization between processes in the including and included applications, this can be achieved by using *start phases*.

Start phases are defined by the `start_phases` key in the `.app` file as a list of tuples `{Phase,PhaseArgs}`, where `Phase` is an atom and `PhaseArgs` is a term. Also, the value of the `mod` key of the including application must be set to `{application_starter,[Module,StartArgs]}`, where `Module` as usual is the application callback module and `StartArgs` a term provided as argument to the callback function `Module:start/2`.

```
{application, prim_app,
 [{description, "Tree application"},
  {vsn, "1"},
  {modules, [prim_app_cb, prim_app_sup, prim_app_server]},
  {registered, [prim_app_server]},
  {included_applications, [incl_app]},
  {start_phases, [{init,[]}, {go,[]}]},
  {applications, [kernel, stdlib, sasl]},
  {mod, {application_starter,[prim_app_cb,[]]}},
  {env, [{file, "/usr/local/log"}]}
 ]}.

{application, incl_app,
 [{description, "Included application"},
  {vsn, "1"},
  {modules, [incl_app_cb, incl_app_sup, incl_app_server]},
  {registered, []},
  {start_phases, [{go,[]}]},
  {applications, [kernel, stdlib, sasl]},
  {mod, {incl_app_cb,[]}}
 ]}.
```

When starting a primary application with included applications, the primary application is started the normal way: The application controller creates an application master for the application, and the application master calls `Module:start(normal, StartArgs)` to start the top supervisor.

Then, for the primary application and each included application in top-down, left-to-right order, the application master calls `Module:start_phase(Phase, Type, PhaseArgs)` for each phase defined for for the primary application, in that order. Note that if a phase is not defined for an included application, the function is not called for this phase and application.

The following requirements apply to the `.app` file for an included application:

- The `{mod, {Module,StartArgs}}` option must be included. This option is used to find the callback module `Module` of the application. `StartArgs` is ignored, as `Module:start/2` is called only for the primary application.
- If the included application itself contains included applications, instead the option `{mod, {application_starter, [Module,StartArgs]}}` must be included.

- The `{start_phases, [{Phase,PhaseArgs}]}` option must be included, and the set of specified phases must be a subset of the set of phases specified for the primary application.

When starting `prim_app` as defined above, the application controller will call the following callback functions, before `application:start(prim_app)` returns a value:

```
application:start(prim_app)
 => prim_app_cb:start(normal, [])
 => prim_app_cb:start_phase(init, normal, [])
 => prim_app_cb:start_phase(go, normal, [])
 => incl_app_cb:start_phase(go, normal, [])
ok
```

9.9 Distributed Applications

9.9.1 Definition

In a distributed system with several Erlang nodes, there may be a need to control applications in a distributed manner. If the node, where a certain application is running, goes down, the application should be restarted at another node.

Such an application is called a *distributed application*. Note that it is the control of the application which is distributed, all applications can of course be distributed in the sense that they, for example, use services on other nodes.

Because a distributed application may move between nodes, some addressing mechanism is required to ensure that it can be addressed by other applications, regardless on which node it currently executes. This issue is not addressed here, but the Kernel module `global` or STDLIB module `pg` can be used for this purpose.

9.9.2 Specifying Distributed Applications

Distributed applications are controlled by both the application controller and a distributed application controller process, `dist_ac`. Both these processes are part of the `kernel` application. Therefore, distributed applications are specified by configuring the `kernel` application, using the following configuration parameter (see also `kernel(6)`):

`distributed = [{Application, [Timeout,] NodeDesc}]`

Specifies where the application `Application = atom()` may execute. `NodeDesc = [Node | {Node,...,Node}]` is a list of node names in priority order. The order between nodes in a tuple is undefined.

`Timeout = integer()` specifies how many milliseconds to wait before restarting the application at another node. Defaults to 0.

For distribution of application control to work properly, the nodes where a distributed application may run must contact each other and negotiate where to start the application. This is done using the following `kernel` configuration parameters:

`sync_nodes_mandatory = [Node]`
Specifies which other nodes must be started (within the timeout specified by `sync_nodes_timeout`.

`sync_nodes_optional = [Node]`
Specifies which other nodes can be started (within the timeout specified by `sync_nodes_timeout`.

`sync_nodes_timeout = integer() | infinity`
Specifies how many milliseconds to wait for the other nodes to start.

When started, the node will wait for all nodes specified by `sync_nodes_mandatory` and `sync_nodes_optional` to come up. When all nodes have come up, or when all mandatory nodes have come up and the time specified by `sync_nodes_timeout` has elapsed, all applications will be started. If not all mandatory nodes have come up, the node will terminate.

Example: An application `myapp` should run at the node `cp1@cave`. If this node goes down, `myapp` should be restarted at `cp2@cave` or `cp3@cave`. A system configuration file `cp1.config` for `cp1@cave` could look like:

```
[{kernel,
  [{distributed, [{myapp, 5000, [cp1@cave, {cp2@cave, cp3@cave}]}]},
   {sync_nodes_mandatory, [cp2@cave, cp3@cave]},
   {sync_nodes_timeout, 5000}
  ]
 }
].
```

The system configuration files for `cp2@cave` and `cp3@cave` are identical, except for the list of mandatory nodes which should be `[cp1@cave, cp3@cave]` for `cp2@cave` and `[cp1@cave, cp2@cave]` for `cp3@cave`.

> Note:
>
> All involved nodes must have the same value for `distributed` and `sync_nodes_timeout`, or the behaviour of the system is undefined.

9.9.3 Starting and Stopping Distributed Applications

When all involved (mandatory) nodes have been started, the distributed application can be started by calling `application:start(Application)` at *all of these nodes.*

It is of course also possible to use a boot script (see *Releases*) which automatically starts the application.

The application will be started at the first node, specified by the `distributed` configuration parameter, which is up and running. The application is started as usual. That is, an application master is created and calls the application callback function:

```
Module:start(normal, StartArgs)
```

Example: Continuing the example from the previous section, the three nodes are started, specifying the system configuration file:

```
> erl -sname cp1 -config cp1
> erl -sname cp2 -config cp2
> erl -sname cp3 -config cp3
```

When all nodes are up and running, `myapp` can be started. This is achieved by calling `application:start(myapp)` at all three nodes. It is then started at `cp1`, as shown in the figure below.

Figure 9.1: Application myapp - Situation 1

Similarly, the application must be stopped by calling `application:stop(Application)` at all involved nodes.

9.9.4 Failover

If the node where the application is running goes down, the application is restarted (after the specified timeout) at the first node, specified by the `distributed` configuration parameter, which is up and running. This is called a *failover*.

The application is started the normal way at the new node, that is, by the application master calling:

```
Module:start(normal, StartArgs)
```

Exception: If the application has the `start_phases` key defined (see *Included Applications*), then the application is instead started by calling:

```
Module:start({failover, Node}, StartArgs)
```

where `Node` is the terminated node.

Example: If `cp1` goes down, the system checks which one of the other nodes, `cp2` or `cp3`, has the least number of running applications, but waits for 5 seconds for `cp1` to restart. If `cp1` does not restart and `cp2` runs fewer applications than `cp3`, then `myapp` is restarted on `cp2`.

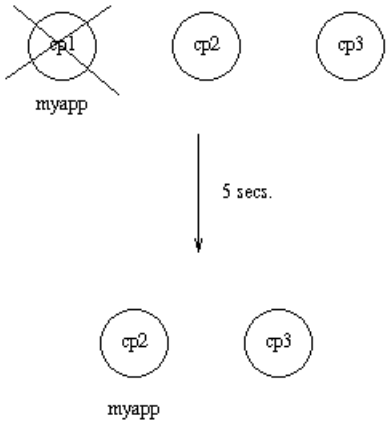

Figure 9.2: Application myapp - Situation 2

Suppose now that `cp2` goes down as well and does not restart within 5 seconds. `myapp` is now restarted on `cp3`.

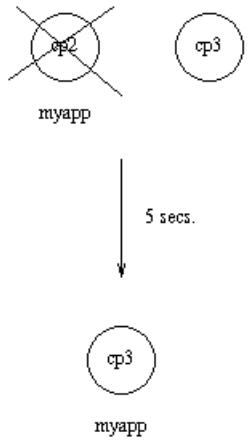

Figure 9.3: Application myapp - Situation 3

9.9.5 Takeover

If a node is started, which has higher priority according to `distributed`, than the node where a distributed application is currently running, the application will be restarted at the new node and stopped at the old node. This is called a *takeover*.

The application is started by the application master calling:

```
Module:start({takeover, Node}, StartArgs)
```

where `Node` is the old node.

Example: If `myapp` is running at `cp3`, and if `cp2` now restarts, it will not restart `myapp`, because the order between nodes `cp2` and `cp3` is undefined.

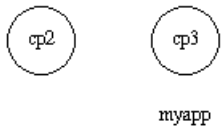

Figure 9.4: Application myapp - Situation 4

However, if `cp1` restarts as well, the function `application:takeover/2` moves `myapp` to `cp1`, because `cp1` has a higher priority than `cp3` for this application. In this case, `Module:start({takeover, cp3@cave}, StartArgs)` is executed at `cp1` to start the application.

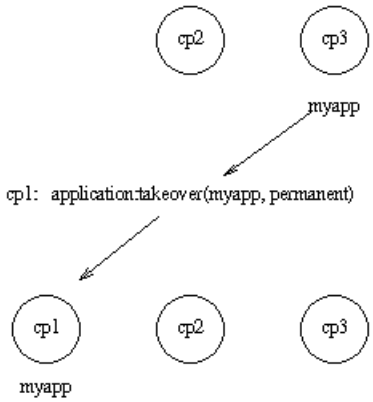

Figure 9.5: Application myapp - Situation 5

9.10 Releases

This chapter should be read in conjuction with `rel(4)`, `systools(3)` and `script(4)`.

9.10.1 Release Concept

When we have written one or more applications, we might want to create a complete system consisting of these applications and a subset of the Erlang/OTP applications. This is called a *release*.

To do this, we create a *release resource file* which defines which applications are included in the release.

The release resource file is used to generate *boot scripts* and *release packages*. A system which is transfered to and installed at another site is called a *target system*. How to use a release package to create a target system is described in System Principles.

9.10.2 Release Resource File

To define a release, we create a *release resource file*, or in short `.rel` file, where we specify the name and version of the release, which ERTS version it is based on, and which applications it consists of:

```
{release, {Name,Vsn}, {erts, EVsn},
 [{Application1, AppVsn1},
   ...
  {ApplicationN, AppVsnN}]}.
```

The file must be named `Rel.rel`, where `Rel` is a unique name.

`Name`, `Vsn` and `Evsn` are strings.

Each `Application` (atom) and `AppVsn` (string) is the name and version of an application included in the release. Note the the minimal release based on Erlang/OTP consists of the `kernel` and `stdlib` applications, so these applications must be included in the list.

Example: We want to make a release of `ch_app` from the *Applications* chapter. It has the following `.app` file:

```
{application, ch_app,
 [{description, "Channel allocator"},
  {vsn, "1"},
```

```
  {modules, [ch_app, ch_sup, ch3]},
  {registered, [ch3]},
  {applications, [kernel, stdlib, sasl]},
  {mod, {ch_app,[]}}
 ]}.
```

The `.rel` file must also contain `kernel`, `stdlib` and `sasl`, since these applications are required by `ch_app`. We call the file `ch_rel-1.rel`:

```
{release,
 {"ch_rel", "A"},
 {erts, "5.3"},
 [{kernel, "2.9"},
  {stdlib, "1.12"},
  {sasl, "1.10"},
  {ch_app, "1"}]
}.
```

9.10.3 Generating Boot Scripts

There are tools in the SASL module `systools` available to build and check releases. The functions read the `.rel` and `.app` files and performs syntax and dependency checks. The function `systools:make_script/1,2` is used to generate a boot script (see System Principles).

```
1> systools:make_script("ch_rel-1", [local]).
ok
```

This creates a boot script, both the readable version `ch_rel-1.script` and the binary version used by the runtime system, `ch_rel-1.boot`. `"ch_rel-1"` is the name of the `.rel` file, minus the extension. `local` is an option that means that the directories where the applications are found are used in the boot script, instead of `$ROOT/lib`. (`$ROOT` is the root directory of the installed release.) This is a useful way to test a generated boot script locally.

When starting Erlang/OTP using the boot script, all applications from the `.rel` file are automatically loaded and started:

```
% erl -boot ch_rel-1
Erlang (BEAM) emulator version 5.3

Eshell V5.3  (abort with ^G)
1>
=PROGRESS REPORT==== 13-Jun-2003::12:01:15 ===
          supervisor: {local,sasl_safe_sup}
             started: [{pid,<0.33.0>},
                       {name,alarm_handler},
                       {mfa,{alarm_handler,start_link,[]}},
                       {restart_type,permanent},
                       {shutdown,2000},
                       {child_type,worker}]

...

=PROGRESS REPORT==== 13-Jun-2003::12:01:15 ===
         application: sasl
          started_at: nonode@nohost
```

```
...
=PROGRESS REPORT==== 13-Jun-2003::12:01:15 ===
         application: ch_app
          started_at: nonode@nohost
```

9.10.4 Creating a Release Package

There is a function `systools:make_tar/1,2` which takes a `.rel` file as input and creates a zipped tar-file with the code for the specified applications, a *release package*.

```
1> systools:make_script("ch_rel-1").
ok
2> systools:make_tar("ch_rel-1").
ok
```

The release package by default contains the `.app` files and object code for all applications, structured according to the *application directory structure*, the binary boot script renamed to `start.boot`, and the `.rel` file.

```
% tar tf ch_rel-1.tar
lib/kernel-2.9/ebin/kernel.app
lib/kernel-2.9/ebin/application.beam
...
lib/stdlib-1.12/ebin/stdlib.app
lib/stdlib-1.12/ebin/beam_lib.beam
...
lib/sasl-1.10/ebin/sasl.app
lib/sasl-1.10/ebin/sasl.beam
...
lib/ch_app-1/ebin/ch_app.app
lib/ch_app-1/ebin/ch_app.beam
lib/ch_app-1/ebin/ch_sup.beam
lib/ch_app-1/ebin/ch3.beam
releases/A/start.boot
releases/ch_rel-1.rel
```

Note that a new boot script was generated, without the `local` option set, before the release package was made. In the release package, all application directories are placed under `lib`. Also, we do not know where the release package will be installed, so we do not want any hardcoded absolute paths in the boot script here.

If a `relup` file and/or a system configuration file called `sys.config` is found, these files are included in the release package as well. See *Release Handling*.

Options can be set to make the release package include source code and the ERTS binary as well.

Refer to System Principles for how to install the first target system, using a release package, and to *Release Handling* for how to install a new release package in an existing system.

9.10.5 Directory Structure

Directory structure for the code installed by the release handler from a release package:

```
$ROOT/lib/App1-AVsn1/ebin
                     /priv
         /App2-AVsn2/ebin
                     /priv
```

```
        ...
        /AppN-AVsnN/ebin
                   /priv
    /erts-EVsn/bin
    /releases/Vsn
    /bin
```

`lib`
: Application directories.

`erts-EVsn/bin`
: Erlang runtime system executables.

`releases/Vsn`
: `.rel` file and boot script `start.boot`.
If present in the release package,
`relup` and/or `sys.config`.

`bin`
: Top level Erlang runtime system executables.

Applications are not required to be located under the `$ROOT/lib` directory. Accordingly, several installation directories may exist which contain different parts of a system. For example, the previous example could be extended as follows:

```
$SECOND_ROOT/.../SApp1-SAVsn1/ebin
                              /priv
                /SApp2-SAVsn2/ebin
                              /priv
                ...
                /SAppN-SAVsnN/ebin
                              /priv

$THIRD_ROOT/TApp1-TAVsn1/ebin
                         /priv
           /TApp2-TAVsn2/ebin
                         /priv
           ...
           /TAppN-TAVsnN/ebin
                         /priv
```

The `$SECOND_ROOT` and `$THIRD_ROOT` are introduced as `variables` in the call to the `systools:make_script/2` function.

Disk-Less and/or Read-Only Clients

If a complete system consists of some disk-less and/or read-only client nodes, a `clients` directory should be added to the `$ROOT` directory. By a read-only node we mean a node with a read-only file system.

The `clients` directory should have one sub-directory per supported client node. The name of each client directory should be the name of the corresponding client node. As a minimum, each client directory should contain the `bin` and `releases` sub-directories. These directories are used to store information about installed releases and to appoint the current release to the client. Accordingly, the `$ROOT` directory contains the following:

```
$ROOT/...
    /clients/ClientName1/bin
                         /releases/Vsn
             /ClientName2/bin
                         /releases/Vsn
```

```
           ...
           /ClientNameN/bin
                       /releases/Vsn
```

This structure should be used if all clients are running the same type of Erlang machine. If there are clients running different types of Erlang machines, or on different operating systems, the `clients` directory could be divided into one sub-directory per type of Erlang machine. Alternatively, you can set up one `$ROOT` per type of machine. For each type, some of the directories specified for the `$ROOT` directory should be included:

```
$ROOT/...
    /clients/Type1/lib
                  /erts-EVsn
                  /bin
                  /ClientName1/bin
                              /releases/Vsn
                  /ClientName2/bin
                              /releases/Vsn
                  ...
                  /ClientNameN/bin
                              /releases/Vsn
           ...
           /TypeN/lib
                 /erts-EVsn
                 /bin
                 ...
```

With this structure, the root directory for clients of `Type1` is `$ROOT/clients/Type1`.

9.11 Release Handling

9.11.1 Release Handling Principles

An important feature of the Erlang programming language is the ability to change module code in run-time, *code replacement*, as described in *Erlang Reference Manual*.

Based on this feature, the OTP application SASL provides a framework for upgrading and downgrading between different versions of an entire release in run-time. This is what we call *release handling*.

The framework consists of off-line support (`systools`) for generating scripts and building release packages, and on-line support (`release_handler`) for unpacking and installing release packages.

Note that the minimal system based on Erlang/OTP, enabling release handling, thus consists of Kernel, STDLIB and SASL.

- A release is created as described in the previous chapter *Releases*. The release is transferred to and installed at target environment. Refer to *System Principles* for information of how to install the first target system.
- Modifications, for example error corrections, are made to the code in the development environment.
- At some point it is time to make a new version of release. The relevant `.app` files are updated and a new `.rel` file is written.
- For each modified application, an *application upgrade file*, `.appup`, is created. In this file, it is described how to upgrade and/or downgrade between the old and new version of the application.
- Based on the `.appup` files, a *release upgrade file* called `relup`, is created. This file describes how to upgrade and/or downgrade between the old and new version of the entire release.
- A new release package is made and transferred to the target system.
- The new release package is unpacked using the release handler.

- The new version of the release is installed, also using the release handler. This is done by evaluating the instructions in `relup`. Modules may be added, deleted or re-loaded, applications may be started, stopped or re-started etc. In some cases, it is even necessary to restart the entire emulator.

  If the installation fails, the system may be rebooted. The old release version is then automatically used.
- If the installation succeeds, the new version is made the default version, which should now be used in case of a system reboot.

The next chapter, *Appup Cookbook*, contains examples of `.appup` files for typical cases of upgrades/downgrades that are normally easy to handle in run-time. However, there are a many aspects that can make release handling complicated. To name a few examples:

- Complicated or circular dependencies can make it difficult or even impossible to decide in which order things must be done without risking run-time errors during an upgrade or downgrade. Dependencies may be:
  - between nodes,
  - between processes, and
  - between modules.
- During release handling, non-affected processes continue normal execution. This may lead to timeouts or other problems. For example, new processes created in the time window between suspending processes using a certain module and loading a new version of this module, may execute old code.

It is therefore recommended that code is changed in as small steps as possible, and always kept backwards compatible.

9.11.2 Requirements

For release handling to work properly, the runtime system needs to have knowledge about which release it is currently running. It must also be able to change (in run-time) which boot script and system configuration file should be used if the system is rebooted, for example by `heart` after a failure. Therefore, Erlang must be started as an embedded system, see *Embedded System* for information on how to do this.

For system reboots to work properly, it is also required that the system is started with heart beat monitoring, see `erl(1)` and `heart(3)`.

Other requirements:

- The boot script included in a release package must be generated from the same `.rel` file as the release package itself.

  Information about applications are fetched from the script when an upgrade or downgrade is performed.
- The system must be configured using one and only one system configuration file, called `sys.config`.

  If found, this file is automatically included when a release package is created.
- All versions of a release, except the first one, must contain a `relup` file.

  If found, this file is automatically included when a release package is created.

9.11.3 Distributed Systems

If the system consists of several Erlang nodes, each node may use its own version of the release. The release handler is a locally registered process and must be called at each node where an upgrade or downgrade is required. There is a release handling instruction that can be used to synchronize the release handler processes at a number of nodes: `sync_nodes`. See `appup(4)`.

9.11.4 Release Handling Instructions

OTP supports a set of *release handling instructions* that is used when creating `.appup` files. The release handler understands a subset of these, the *low-level* instructions. To make it easier for the user, there are also a number of *high-level* instructions, which are translated to low-level instructions by `systools:make_relup`.

Here, some of the most frequently used instructions are described. The complete list of instructions is found in `appup(4)`.

First, some definitions:

Residence module

The module where a process has its tail-recursive loop function(s). If the tail-recursive loop functions are implemented in several modules, all those modules are residence modules for the process.

Functional module

A module which is not a residence module for any process.

Note that for a process implemented using an OTP behaviour, the behaviour module is the residence module for that process. The callback module is a functional module.

load_module

If a simple extension has been made to a functional module, it is sufficient to simply load the new version of the module into the system, and remove the old version. This is called *simple code replacement* and for this the following instruction is used:

```
{load_module, Module}
```

update

If a more complex change has been made, for example a change to the format of the internal state of a gen_server, simple code replacement is not sufficient. Instead it is necessary to suspend the processes using the module (to avoid that they try to handle any requests before the code replacement is completed), ask them to transform the internal state format and switch to the new version of the module, remove the old version and last, resume the processes. This is called *synchronized code replacement* and for this the following instructions are used:

```
{update, Module, {advanced, Extra}}
{update, Module, supervisor}
```

`update` with argument `{advanced,Extra}` is used when changing the internal state of a behaviour as described above. It will cause behaviour processes to call the callback function `code_change`, passing the term `Extra` and some other information as arguments. See the man pages for the respective behaviours and *Appup Cookbook*.

`update` with argument `supervisor` is used when changing the start specification of a supervisor. See *Appup Cookbook*.

The release handler finds the processes *using* a module to update by traversing the supervision tree of each running application and checking all the child specifications:

```
{Id, StartFunc, Restart, Shutdown, Type, Modules}
```

A process is using a module if the name is listed in `Modules` in the child specification for the process.

If `Modules=dynamic`, which is the case for event managers, the event manager process informs the release handler about the list of currently installed event handlers (gen_fsm) and it is checked if the module name is in this list instead.

The release handler suspends, asks for code change, and resumes processes by calling the functions `sys:suspend/1,2`, `sys:change_code/4,5` and `sys:resume/1,2` respectively.

add_module and delete_module

If a new module is introduced, the following instruction is used:

```
{add_module, Module}
```

The instruction loads the module and is absolutely necessary when running Erlang in embedded mode. It is not strictly required when running Erlang in interactive (default) mode, since the code server automatically searches for and loads unloaded modules.

The opposite of `add_module` is `delete_module` which unloads a module:

```
{delete_module, Module}
```

Note that any process, in any application, with `Module` as residence module, is killed when the instruction is evaluated. The user should therefore ensure that all such processes are terminated before deleting the module, to avoid a possible situation with failing supervisor restarts.

Application Instructions

Instruction for adding an application:

```
{add_application, Application}
```

Adding an application means that the modules defined by the `modules` key in the `.app` file are loaded using a number of `add_module` instructions, then the application is started.

Instruction for removing an application:

```
{remove_application, Application}
```

Removing an application means that the application is stopped, the modules are unloaded using a number of `delete_module` instructions and then the application specification is unloaded from the application controller.

Instruction for removing an application:

```
{restart_application, Application}
```

Restarting an application means that the application is stopped and then started again similar to using the instructions `remove_application` and `add_application` in sequence.

apply (low-level)

To call an arbitrary function from the release handler, the following instruction is used:

```
{apply, {M, F, A}}
```

The release handler will evalute `apply(M, F, A)`.

restart_new_emulator (low-level)

This instruction is used when changing to a new emulator version, or if a system reboot is needed for some other reason. Requires that the system is started with heart beat monitoring, see `erl(1)` and `heart(3)`.

When the release handler encounters the instruction, it shuts down the current emulator by calling `init:reboot()`, see `init(3)`. All processes are terminated gracefully and the system can then be rebooted by the heart program, using the new release version. This new version must still be made permanent when the new emulator is up and running. Otherwise, the old version is used in case of a new system reboot.

On UNIX, the release handler tells the heart program which command to use to reboot the system. Note that the environment variable `HEART_COMMAND`, normally used by the heart program, in this case is ignored. The command instead defaults to `$ROOT/bin/start`. Another command can be set by using the SASL configuration parameter `start_prg`, see `sasl(6)`.

9.11.5 Application Upgrade File

To define how to upgrade/downgrade between the current version and previous versions of an application, we create an *application upgrade file*, or in short `.appup` file. The file should be called `Application.appup`, where `Application` is the name of the application:

```
{Vsn,
 [{UpFromVsn1, InstructionsU1},
  ...,
  {UpFromVsnK, InstructionsUK}],
 [{DownToVsn1, InstructionsD1},
  ...,
  {DownToVsnK, InstructionsDK}]}.
```

`Vsn`, a string, is the current version of the application, as defined in the `.app` file. Each `UpFromVsn` is a previous version of the application to upgrade from, and each `DownToVsn` is a previous version of the application to downgrade to. Each `Instructions` is a list of release handling instructions.

The syntax and contents of the `appup` file are described in detail in `appup(4)`.

In the chapter *Appup Cookbook*, examples of `.appup` files for typical upgrade/downgrade cases are given.

Example: Consider the release `ch_rel-1` from the *Releases* chapter. Assume we want to add a function `available/0` to the server `ch3` which returns the number of available channels:

(Hint: When trying out the example, make the changes in a copy of the original directory, so that the first versions are still available.)

```
-module(ch3).
-behaviour(gen_server).

-export([start_link/0]).
-export([alloc/0, free/1]).
-export([available/0]).
-export([init/1, handle_call/3, handle_cast/2]).

start_link() ->
    gen_server:start_link({local, ch3}, ch3, [], []).
```

```
alloc() ->
    gen_server:call(ch3, alloc).

free(Ch) ->
    gen_server:cast(ch3, {free, Ch}).

available() ->
    gen_server:call(ch3, available).

init(_Args) ->
    {ok, channels()}.

handle_call(alloc, _From, Chs) ->
    {Ch, Chs2} = alloc(Chs),
    {reply, Ch, Chs2};
handle_call(available, _From, Chs) ->
    N = available(Chs),
    {reply, N, Chs}.

handle_cast({free, Ch}, Chs) ->
    Chs2 = free(Ch, Chs),
    {noreply, Chs2}.
```

A new version of the `ch_app.app` file must now be created, where the version is updated:

```
{application, ch_app,
 [{description, "Channel allocator"},
  {vsn, "2"},
  {modules, [ch_app, ch_sup, ch3]},
  {registered, [ch3]},
  {applications, [kernel, stdlib, sasl]},
  {mod, {ch_app,[]}}
 ]}.
```

To upgrade `ch_app` from `"1"` to `"2"` (and to downgrade from `"2"` to `"1"`), we simply need to load the new (old) version of the `ch3` callback module. We create the application upgrade file `ch_app.appup` in the `ebin` directory:

```
{"2",
 [{"1", [{load_module, ch3}]}],
 [{"1", [{load_module, ch3}]}]
}.
```

9.11.6 Release Upgrade File

To define how to upgrade/downgrade between the new version and previous versions of a release, we create a *release upgrade file*, or in short `relup` file.

This file does not need to be created manually, it can be generated by `systools:make_relup/3,4`. The relevant versions of the `.rel` file, `.app` files and `.appup` files are used as input. It is deducted which applications should be added and deleted, and which applications that need to be upgraded and/or downgraded. The instructions for this is fetched from the `.appup` files and transformed into a single list of low-level instructions in the right order.

If the `relup` file is relatively simple, it can be created manually. Remember that it should only contain low-level instructions.

The syntax and contents of the release upgrade file are described in detail in `relup(4)`.

Example, continued from the previous section. We have a new version "2" of `ch_app` and an `.appup` file. We also need a new version of the `.rel` file. This time the file is called `ch_rel-2.rel` and the release version string is changed changed from "A" to "B":

```
{release,
 {"ch_rel", "B"},
 {erts, "5.3"},
 [{kernel, "2.9"},
  {stdlib, "1.12"},
  {sasl, "1.10"},
  {ch_app, "2"}]
}.
```

Now the `relup` file can be generated:

```
1> systools:make_relup("ch_rel-2", ["ch_rel-1"], ["ch_rel-1"]).
ok
```

This will generate a `relup` file with instructions for how to upgrade from version "A" ("ch_rel-1") to version "B" ("ch_rel-2") and how to downgrade from version "B" to version "A".

Note that both the old and new versions of the `.app` and `.rel` files must be in the code path, as well as the `.appup` and (new) `.beam` files. It is possible to extend the code path by using the option `path`:

```
1> systools:make_relup("ch_rel-2", ["ch_rel-1"], ["ch_rel-1"],
[{path,["../ch_rel-1",
"../ch_rel-1/lib/ch_app-1/ebin"]}]).
ok
```

9.11.7 Installing a Release

When we have made a new version of a release, a release package can be created with this new version and transferred to the target environment.

To install the new version of the release in run-time, the *release handler* is used. This is a process belonging to the SASL application, that handles unpacking, installation, and removal of release packages. It is interfaced through the module `release_handler`, which is described in detail in `release_handler(3)`.

Assuming there is a target system up and running with installation root directory `$ROOT`, the release package with the new version of the release should be copied to `$ROOT/releases`.

The first action is to *unpack* the release package, the files are then extracted from the package:

```
release_handler:unpack_release(ReleaseName) => {ok, Vsn}
```

`ReleaseName` is the name of the release package except the `.tar.gz` extension. `Vsn` is the version of the unpacked release, as defined in its `.rel` file.

A directory `$ROOT/lib/releases/Vsn` will be created, where the `.rel` file, the boot script `start.boot`, the system configuration file `sys.config` and `relup` are placed. For applications with new version numbers, the application directories will be placed under `$ROOT/lib`. Unchanged applications are not affected.

An unpacked release can be *installed*. The release handler then evaluates the instructions in `relup`, step by step:

```
release_handler:install_release(Vsn) => {ok, FromVsn, []}
```

If an error occurs during the installation, the system is rebooted using the old version of the release. If installation succeeds, the system is afterwards using the new version of the release, but should anything happen and the system is rebooted, it would start using the previous version again. To be made the default version, the newly installed release must be made *permanent*, which means the previous version becomes *old*:

```
release_handler:make_permanent(Vsn) => ok
```

The system keeps information about which versions are old and permanent in the files `$ROOT/releases/RELEASES` and `$ROOT/releases/start_erl.data`.

To downgrade from `Vsn` to `FromVsn`, `install_release` must be called again:

```
release_handler:install_release(FromVsn) => {ok, Vsn, []}
```

An installed, but not permanent, release can be *removed*. Information about the release is then deleted from `$ROOT/releases/RELEASES` and the release specific code, that is the new application directories and the `$ROOT/releases/Vsn` directory, are removed.

```
release_handler:remove_release(Vsn) => ok
```

Example, continued from the previous sections:

1) Create a target system as described in *System Principles* of the first version `"A"` of `ch_rel` from the *Releases* chapter. This time `sys.config` must be included in the release package. If no configuration is needed, the file should contain the empty list:

```
[].
```

2) Start the system as a simple target system. Note that in reality, it should be started as an embedded system. However, using `erl` with the correct boot script and `.config` file is enough for illustration purposes:

```
% cd $ROOT
% bin/erl -boot $ROOT/releases/A/start -config $ROOT/releases/A/sys
...
```

`$ROOT` is the installation directory of the target system.

3) In another Erlang shell, generate start scripts and create a release package for the new version `"B"`. Remember to include (a possible updated) `sys.config` and the `relup` file, see *Release Upgrade File* above.

```
1> systools:make_script("ch_rel-2").
ok
2> systools:make_tar("ch_rel-2").
ok
```

The new release package now contains version "2" of `ch_app` and the `relup` file as well:

```
% tar tf ch_rel-2.tar
lib/kernel-2.9/ebin/kernel.app
lib/kernel-2.9/ebin/application.beam
...
lib/stdlib-1.12/ebin/stdlib.app
lib/stdlib-1.12/ebin/beam_lib.beam
...
lib/sasl-1.10/ebin/sasl.app
lib/sasl-1.10/ebin/sasl.beam
...
lib/ch_app-2/ebin/ch_app.app
lib/ch_app-2/ebin/ch_app.beam
lib/ch_app-2/ebin/ch_sup.beam
lib/ch_app-2/ebin/ch3.beam
releases/B/start.boot
releases/B/relup
releases/B/sys.config
releases/ch_rel-2.rel
```

4) Copy the release package `ch_rel-2.tar.gz` to the `$ROOT/releases` directory.

5) In the running target system, unpack the release package:

```
1> release_handler:unpack_release("ch_rel-2").
{ok,"B"}
```

The new application version `ch_app-2` is installed under `$ROOT/lib` next to `ch_app-1`. The `kernel`, `stdlib` and `sasl` directories are not affected, as they have not changed.

Under `$ROOT/releases`, a new directory `B` is created, containing `ch_rel-2.rel`, `start.boot`, `sys.config` and `relup`.

6) Check if the function `ch3:available/0` is available:

```
2> ch3:available().
** exception error: undefined function ch3:available/0
```

7) Install the new release. The instructions in `$ROOT/releases/B/relup` are executed one by one, resulting in the new version of `ch3` being loaded. The function `ch3:available/0` is now available:

```
3> release_handler:install_release("B").
{ok,"A",[]}
4> ch3:available().
3
5> code:which(ch3).
".../lib/ch_app-2/ebin/ch3.beam"
6> code:which(ch_sup).
".../lib/ch_app-1/ebin/ch_sup.beam"
```

Note that processes in `ch_app` for which code have not been updated, for example the supervisor, are still evaluating code from `ch_app-1`.

8) If the target system is now rebooted, it will use version "A" again. The "B" version must be made permanent, in order to be used when the system is rebooted.

```
7> release_handler:make_permanent("B").
ok
```

9.11.8 Updating Application Specifications

When a new version of a release is installed, the application specifications are automatically updated for all loaded applications.

> **Note:**
>
> The information about the new application specifications are fetched from the boot script included in the release package. It is therefore important that the boot script is generated from the same `.rel` file as is used to build the release package itself.

Specifically, the application configuration parameters are automatically updated according to (in increasing priority order):

- The data in the boot script, fetched from the new application resource file `App.app`
- The new `sys.config`
- Command line arguments `-App Par Val`

This means that parameter values set in the other system configuration files, as well as values set using `application:set_env/3`, are disregarded.

When an installed release is made permanent, the system process `init` is set to point out the new `sys.config`.

After the installation, the application controller will compare the old and new configuration parameters for all running applications and call the callback function:

```
Module:config_change(Changed, New, Removed)
```

`Module` is the application callback module as defined by the `mod` key in the `.app` file. `Changed` and `New` are lists of `{Par,Val}` for all changed and added configuration parameters, respectively. `Removed` is a list of all parameters `Par` that have been removed.

The function is optional and may be omitted when implementing an application callback module.

9.12 Appup Cookbook

This chapter contains examples of `.appup` files for typical cases of upgrades/downgrades done in run-time.

9.12.1 Changing a Functional Module

When a change has been made to a functional module, for example if a new function has been added or a bug has been corrected, simple code replacement is sufficient.

Example:

```
{"2",
 [{"1", [{load_module, m}]}],
 [{"1", [{load_module, m}]}]
}.
```

9.12.2 Changing a Residence Module

In a system implemented according to the OTP Design Principles, all processes, except system processes and special processes, reside in one of the behaviours `supervisor`, `gen_server`, `gen_fsm` or `gen_event`. These belong to the STDLIB application and upgrading/downgrading normally requires an emulator restart.

OTP thus provides no support for changing residence modules except in the case of *special processes*.

9.12.3 Changing a Callback Module

A callback module is a functional module, and for code extensions simple code replacement is sufficient.

Example: When adding a function to `ch3` as described in the example in *Release Handling*, `ch_app.appup` looks as follows:

```
{"2",
 [{"1", [{load_module, ch3}]}],
 [{"1", [{load_module, ch3}]}]
}.
```

OTP also supports changing the internal state of behaviour processes, see *Changing Internal State* below.

9.12.4 Changing Internal State

In this case, simple code replacement is not sufficient. The process must explicitly transform its state using the callback function `code_change` before switching to the new version of the callback module. Thus synchronized code replacement is used.

Example: Consider the gen_server `ch3` from the chapter about the *gen_server behaviour*. The internal state is a term `Chs` representing the available channels. Assume we want add a counter `N` which keeps track of the number of `alloc` requests so far. This means we need to change the format to `{Chs,N}`.

The `.appup` file could look as follows:

```
{"2",
 [{"1", [{update, ch3, {advanced, []}}]}],
 [{"1", [{update, ch3, {advanced, []}}]}]
}.
```

The third element of the `update` instruction is a tuple `{advanced,Extra}` which says that the affected processes should do a state transformation before loading the new version of the module. This is done by the processes calling the callback function `code_change` (see `gen_server(3)`). The term `Extra`, in this case [], is passed as-is to the function:

```
-module(ch3).
...
-export([code_change/3]).
...
```

```
code_change({down, _Vsn}, {Chs, N}, _Extra) ->
    {ok, Chs};
code_change(_Vsn, Chs, _Extra) ->
    {ok, {Chs, 0}}.
```

The first argument is `{down,Vsn}` in case of a downgrade, or `Vsn` in case of an upgrade. The term `Vsn` is fetched from the 'original' version of the module, i.e. the version we are upgrading from, or downgrading to.

The version is defined by the module attribute `vsn`, if any. There is no such attribute in `ch3`, so in this case the version is the checksum (a huge integer) of the BEAM file, an uninteresting value which is ignored.

(The other callback functions of `ch3` need to be modified as well and perhaps a new interface function added, this is not shown here).

9.12.5 Module Dependencies

Assume we extend a module by adding a new interface function, as in the example in *Release Handling*, where a function `available/0` is added to `ch3`.

If we also add a call to this function, say in the module `m1`, a run-time error could occur during release upgrade if the new version of `m1` is loaded first and calls `ch3:available/0` before the new version of `ch3` is loaded.

Thus, `ch3` must be loaded before `m1` is, in the upgrade case, and vice versa in the downgrade case. We say that `m1` *is dependent on* `ch3`. In a release handling instruction, this is expressed by the element `DepMods`:

```
{load_module, Module, DepMods}
{update, Module, {advanced, Extra}, DepMods}
```

`DepMods` is a list of modules, on which `Module` is dependent.

Example: The module `m1` in the application `myapp` is dependent on `ch3` when upgrading from "1" to "2", or downgrading from "2" to "1":

```
myapp.appup:

{"2",
 [{"1", [{load_module, m1, [ch3]}]}],
 [{"1", [{load_module, m1, [ch3]}]}]
}.

ch_app.appup:

{"2",
 [{"1", [{load_module, ch3}]}],
 [{"1", [{load_module, ch3}]}]
}.
```

If `m1` and `ch3` had belonged to the same application, the `.appup` file could have looked like this:

```
{"2",
 [{"1",
   [{load_module, ch3},
    {load_module, m1, [ch3]}]}],
 [{"1",
   [{load_module, ch3},
    {load_module, m1, [ch3]}]}]
```

```
}.
```

Note that it is `m1` that is dependent on `ch3` also when downgrading. `systools` knows the difference between up- and downgrading and will generate a correct `relup`, where `ch3` is loaded before `m1` when upgrading but `m1` is loaded before `ch3` when downgrading.

9.12.6 Changing Code For a Special Process

In this case, simple code replacement is not sufficient. When a new version of a residence module for a special process is loaded, the process must make a fully qualified call to its loop function to switch to the new code. Thus synchronized code replacement must be used.

> **Note:**
>
> The name(s) of the user-defined residence module(s) must be listed in the `Modules` part of the child specification for the special process, in order for the release handler to find the process.

Example. Consider the example `ch4` from the chapter about *sys and proc_lib*. When started by a supervisor, the child specification could look like this:

```
{ch4, {ch4, start_link, []},
 permanent, brutal_kill, worker, [ch4]}
```

If `ch4` is part of the application `sp_app` and a new version of the module should be loaded when upgrading from version "1" to "2" of this application, `sp_app.appup` could look like this:

```
{"2",
 [{"1", [{update, ch4, {advanced, []}}]}],
 [{"1", [{update, ch4, {advanced, []}}]}]
}.
```

The `update` instruction must contain the tuple `{advanced,Extra}`. The instruction will make the special process call the callback function `system_code_change/4`, a function the user must implement. The term `Extra`, in this case [], is passed as-is to `system_code_change/4`:

```
-module(ch4).
...
-export([system_code_change/4]).
...

system_code_change(Chs, _Module, _OldVsn, _Extra) ->
    {ok, Chs}.
```

The first argument is the internal state `State` passed from the function `sys:handle_system_msg(Request, From, Parent, Module, Deb, State)`, called by the special process when a system message is received. In `ch4`, the internal state is the set of available channels `Chs`.

The second argument is the name of the module (`ch4`).

The third argument is `Vsn` or `{down,Vsn}` as described for *gen_server:code_change/3*.

In this case, all arguments but the first are ignored and the function simply returns the internal state again. This is enough if the code only has been extended. If we had wanted to change the internal state (similar to the example in *Changing Internal State*), it would have been done in this function and `{ok,Chs2}` returned.

9.12.7 Changing a Supervisor

The supervisor behaviour supports changing the internal state, i.e. changing restart strategy and maximum restart frequency properties, as well as changing existing child specifications.

Adding and deleting child processes are also possible, but not handled automatically. Instructions must be given by in the `.appup` file.

Changing Properties

Since the supervisor should change its internal state, synchronized code replacement is required. However, a special `update` instruction must be used.

The new version of the callback module must be loaded first both in the case of upgrade and downgrade. Then the new return value of `init/1` can be checked and the internal state be changed accordingly.

The following `upgrade` instruction is used for supervisors:

```
{update, Module, supervisor}
```

Example: Assume we want to change the restart strategy of `ch_sup` from the *Supervisor Behaviour* chapter from one_for_one to one_for_all. We change the callback function `init/1` in `ch_sup.erl`:

```
-module(ch_sup).
...

init(_Args) ->
    {ok, {{one_for_all, 1, 60}, ...}}.
```

The file `ch_app.appup`:

```
{"2",
 [{"1", [{update, ch_sup, supervisor}]}],
 [{"1", [{update, ch_sup, supervisor}]}]
}.
```

Changing Child Specifications

The instruction, and thus the `.appup` file, when changing an existing child specification, is the same as when changing properties as described above:

```
{"2",
 [{"1", [{update, ch_sup, supervisor}]}],
 [{"1", [{update, ch_sup, supervisor}]}]
}.
```

The changes do not affect existing child processes. For example, changing the start function only specifies how the child process should be restarted, if needed later on.

Note that the id of the child specification cannot be changed.

Note also that changing the `Modules` field of the child specification may affect the release handling process itself, as this field is used to identify which processes are affected when doing a synchronized code replacement.

Adding And Deleting Child Processes

As stated above, changing child specifications does not affect existing child processes. New child specifications are automatically added, but not deleted. Also, child processes are not automatically started or terminated. Instead, this must be done explicitly using `apply` instructions.

Example: Assume we want to add a new child process `m1` to `ch_sup` when upgrading `ch_app` from "1" to "2". This means `m1` should be deleted when downgrading from "2" to "1":

```
{"2",
 [{"1",
   [{update, ch_sup, supervisor},
    {apply, {supervisor, restart_child, [ch_sup, m1]}}
   ]}],
 [{"1",
   [{apply, {supervisor, terminate_child, [ch_sup, m1]}},
    {apply, {supervisor, delete_child, [ch_sup, m1]}},
    {update, ch_sup, supervisor}
   ]}]
}.
```

Note that the order of the instructions is important.

Note also that the supervisor must be registered as `ch_sup` for the script to work. If the supervisor is not registered, it cannot be accessed directly from the script. Instead a help function that finds the pid of the supervisor and calls `supervisor:restart_child` etc. must be written, and it is this function that should be called from the script using the `apply` instruction.

If the module `m1` is introduced in version "2" of `ch_app`, it must also be loaded when upgrading and deleted when downgrading:

```
{"2",
 [{"1",
   [{add_module, m1},
    {update, ch_sup, supervisor},
    {apply, {supervisor, restart_child, [ch_sup, m1]}}
   ]}],
 [{"1",
   [{apply, {supervisor, terminate_child, [ch_sup, m1]}},
    {apply, {supervisor, delete_child, [ch_sup, m1]}},
    {update, ch_sup, supervisor},
    {delete_module, m1}
   ]}]
}.
```

Note again that the order of the instructions is important. When upgrading, `m1` must be loaded and the supervisor's child specification changed, before the new child process can be started. When downgrading, the child process must be terminated before child specification is changed and the module is deleted.

9.12.8 Adding or Deleting a Module

Example: A new functional module `m` is added to `ch_app`:

```
{"2",
 [{"1", [{add_module, m}]}],
 [{"1", [{delete_module, m}]}]
```

9.12.9 Starting or Terminating a Process

In a system structured according to the OTP design principles, any process would be a child process belonging to a supervisor, see *Adding and Deleting Child Processes* above.

9.12.10 Adding or Removing an Application

When adding or removing an application, no `.appup` file is needed. When generating `relup`, the `.rel` files are compared and `add_application` and `remove_application` instructions are added automatically.

9.12.11 Restarting an Application

Restarting an application is useful when a change is too complicated to be made without restarting the processes, for example if the supervisor hierarchy has been restructured.

Example: When adding a new child `m1` to `ch_sup`, as in the *example above*, an alternative to updating the supervisor is to restart the entire application:

```
{"2",
 [{"1", [{restart_application, ch_app}]}],
 [{"1", [{restart_application, ch_app}]}]
}.
```

9.12.12 Changing an Application Specification

When installing a release, the application specifications are automatically updated before evaluating the `relup` script. Hence, no instructions are needed in the `.appup` file:

```
{"2",
 [{"1", []}],
 [{"1", []}]
}.
```

9.12.13 Changing Application Configuration

Changing an application configuration by updating the `env` key in the `.app` file is an instance of changing an application specification, *see above*.

Alternatively, application configuration parameters can be added or updated in `sys.config`.

9.12.14 Changing Included Applications

The release handling instructions for adding, removing and restarting applications apply to primary applications only. There are no corresponding instructions for included applications. However, since an included application is really a supervision tree with a topmost supervisor, started as a child process to a supervisor in the including application, a `relup` file can be manually created.

Example: Assume we have a release containing an application `prim_app` which have a supervisor `prim_sup` in its supervision tree.

In a new version of the release, our example application `ch_app` should be included in `prim_app`. That is, its topmost supervisor `ch_sup` should be started as a child process to `prim_sup`.

1) Edit the code for `prim_sup`:

```
init(...) ->
    {ok, {...supervisor flags...,
          [...,
           {ch_sup, {ch_sup,start_link,[]},
            permanent,infinity,supervisor,[ch_sup]},
           ...]}}.
```

2) Edit the `.app` file for `prim_app`:

```
{application, prim_app,
 [...,
  {vsn, "2"},
  ...,
  {included_applications, [ch_app]},
  ...
 ]}.
```

3) Create a new `.rel` file, including `ch_app`:

```
{release,
 ...,
 [...,
  {prim_app, "2"},
  {ch_app, "1"}]}.
```

Application Restart

4a) One way to start the included application is to restart the entire `prim_app` application. Normally, we would then use the `restart_application` instruction in the `.appup` file for `prim_app`.

However, if we did this and then generated a `relup` file, not only would it contain instructions for restarting (i.e. removing and adding) `prim_app`, it would also contain instructions for starting `ch_app` (and stopping it, in the case of downgrade). This is due to the fact that `ch_app` is included in the new `.rel` file, but not in the old one.

Instead, a correct `relup` file can be created manually, either from scratch or by editing the generated version. The instructions for starting/stopping `ch_app` are replaced by instructions for loading/unloading the application:

```
{"B",
 [{"A",
   [],
   [{load_object_code,{ch_app,"1",[ch_sup,ch3]}},
    {load_object_code,{prim_app,"2",[prim_app,prim_sup]}},
    point_of_no_return,
    {apply,{application,stop,[prim_app]}},
    {remove,{prim_app,brutal_purge,brutal_purge}},
    {remove,{prim_sup,brutal_purge,brutal_purge}},
    {purge,[prim_app,prim_sup]},
    {load,{prim_app,brutal_purge,brutal_purge}},
    {load,{prim_sup,brutal_purge,brutal_purge}},
    {load,{ch_sup,brutal_purge,brutal_purge}},
```

```
    {load,{ch3,brutal_purge,brutal_purge}},
    {apply,{application,load,[ch_app]}},
    {apply,{application,start,[prim_app,permanent]}}]}],
 [{"A",
   [],
   [{load_object_code,{prim_app,"1",[prim_app,prim_sup]}},
    point_of_no_return,
    {apply,{application,stop,[prim_app]}},
    {apply,{application,unload,[ch_app]}},
    {remove,{ch_sup,brutal_purge,brutal_purge}},
    {remove,{ch3,brutal_purge,brutal_purge}},
    {purge,[ch_sup,ch3]},
    {remove,{prim_app,brutal_purge,brutal_purge}},
    {remove,{prim_sup,brutal_purge,brutal_purge}},
    {purge,[prim_app,prim_sup]},
    {load,{prim_app,brutal_purge,brutal_purge}},
    {load,{prim_sup,brutal_purge,brutal_purge}},
    {apply,{application,start,[prim_app,permanent]}}]}]
}.
```

Supervisor Change

4b) Another way to start the included application (or stop it in the case of downgrade) is by combining instructions for adding and removing child processes to/from `prim_sup` with instructions for loading/unloading all `ch_app` code and its application specification.

Again, the `relup` file is created manually. Either from scratch or by editing a generated version. Load all code for `ch_app` first, and also load the application specification, before `prim_sup` is updated. When downgrading, `prim_sup` should be updated first, before the code for `ch_app` and its application specification are unloaded.

```
{"B",
 [{"A",
   [],
   [{load_object_code,{ch_app,"1",[ch_sup,ch3]}},
    {load_object_code,{prim_app,"2",[prim_sup]}},
    point_of_no_return,
    {load,{ch_sup,brutal_purge,brutal_purge}},
    {load,{ch3,brutal_purge,brutal_purge}},
    {apply,{application,load,[ch_app]}},
    {suspend,[prim_sup]},
    {load,{prim_sup,brutal_purge,brutal_purge}},
    {code_change,up,[{prim_sup,[]}]},
    {resume,[prim_sup]},
    {apply,{supervisor,restart_child,[prim_sup,ch_sup]}}]}],
 [{"A",
   [],
   [{load_object_code,{prim_app,"1",[prim_sup]}},
    point_of_no_return,
    {apply,{supervisor,terminate_child,[prim_sup,ch_sup]}},
    {apply,{supervisor,delete_child,[prim_sup,ch_sup]}},
    {suspend,[prim_sup]},
    {load,{prim_sup,brutal_purge,brutal_purge}},
    {code_change,down,[{prim_sup,[]}]},
    {resume,[prim_sup]},
    {remove,{ch_sup,brutal_purge,brutal_purge}},
    {remove,{ch3,brutal_purge,brutal_purge}},
    {purge,[ch_sup,ch3]},
    {apply,{application,unload,[ch_app]}}]}]
}.
```

9.12.15 Changing Non-Erlang Code

Changing code for a program written in another programming language than Erlang, for example a port program, is very application dependent and OTP provides no special support for it.

Example, changing code for a port program: Assume that the Erlang process controlling the port is a gen_server `portc` and that the port is opened in the callback function `init/1`:

```
init(...) ->
    ...,
    PortPrg = filename:join(code:priv_dir(App), "portc"),
    Port = open_port({spawn,PortPrg}, [...]),
    ...,
    {ok, #state{port=Port, ...}}.
```

If the port program should be updated, we can extend the code for the gen_server with a `code_change` function which closes the old port and opens a new port. (If necessary, the gen_server may first request data that needs to be saved from the port program and pass this data to the new port):

```
code_change(_OldVsn, State, port) ->
    State#state.port ! close,
    receive
        {Port,close} ->
            true
    end,
    PortPrg = filename:join(code:priv_dir(App), "portc"),
    Port = open_port({spawn,PortPrg}, [...]),
    {ok, #state{port=Port, ...}}.
```

Update the application version number in the `.app` file and write an `.appup` file:

```
["2",
 [{"1", [{update, portc, {advanced,port}}]}],
 [{"1", [{update, portc, {advanced,port}}]}]
].
```

Make sure the `priv` directory where the C program is located is included in the new release package:

```
1> systools:make_tar("my_release", [{dirs,[priv]}]).
...
```

9.12.16 Emulator Restart

If the emulator can or should be restarted, the very simple `.relup` file can be created manually:

```
{"B",
 [{"A",
   [],
   [restart_new_emulator]}],
 [{"A",
   [],
```

```
   [restart_new_emulator]}]
}.
```

This way, the release handler framework with automatic packing and unpacking of release packages, automatic path updates etc. can be used without having to specify `.appup` files.

If some transformation of persistent data, for example database contents, needs to be done before installing the new release version, instructions for this can be added to the `.relup` file as well.

10 User's Guide

10.1 Introduction

The operation and maintenance support in OTP consists of a generic model for management subsystems in OTP, and some components to be used in these subsystems. This document describes the model.

The main idea in the model is that it is management protocol independent. Thus, it is not tied to any specific management protocol. An API is defined which can be used to write adaptations for specific management protocols.

Each OAM component in OTP is implemented as one sub application, which can be included in a management application for the system. Note that such a complete management application is not in the scope of this generic functionality. Examples illustrating how such an application can be built are included however.

10.1.1 Terminology

The protocol independent architectural model on the network level is the well-known Client-Server model for management operations. This model is based on the client-server principle, where the manager (client) sends A request is sent from a manager to an agent when it accesses management information.to the agent (server), the agent sends A reply is sent from the agent as a response to a request from a manager.back to the manager. There are two main differences to the normal client-server model. First, there are usually a few managers that communicate with many agents; and second, the agent may spontaneously send A notification is sent spontaneously from an agent to a manager, e.g. an alarm.to the manager. The picture below illustrates the idea.

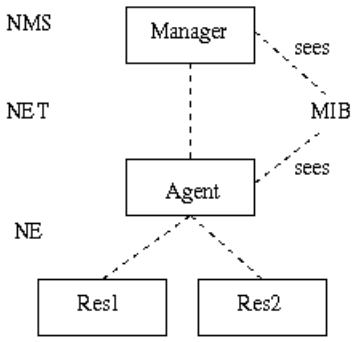

Figure 1.1: Terminology

The manager is often referred to as the , to emphasize that it usually is realized as a program that presents data to an operator.

The agent is an entity that executes within a . In OTP, the network element may be a distributed system, meaning that the distributed system is managed as one entity. Of course, the agent may be configured to be able to run on one of several nodes, making it a distributed OTP application.

The management information is defined in an . It is a formal definition of which information the agent makes available to the manager. The manager accesses the MIB through a management protocol, such as SNMP, CMIP, HTTP or CORBA. Each of these protocols have their own MIB definition language. In SNMP, it is a subset of ASN.1, in CMIP it is GDMO, in HTTP it is implicit, and using CORBA, it is IDL. Usually, the entities defined in the MIB are called , although these objects do not have to be objects in the OO way,for example, a simple scalar variable defined in an

MIB is called a Managed Object. The Managed Objects are logical objects, not necessarily with a one-to-one mapping to the resources.

10.1.2 Model

In this section, the generic protocol independent model for use within an OTP based network element is presented. This model is used by all operation and maintenance components, and may be used by the applications. The advantage of the model is that it clearly separates the resources from the management protocol. The resources do not need to be aware of which management protocol is used to manage the system. This makes it possible to manage the same resources with different protocols.

The different entities involved in this model are the which terminates the management protocol, and the which is to be managed, i.e. the actual application entities. The resources should in general have no knowledge of the management protocol used, and the agent should have no knowledge of the managed resources. This implies that some sort of translation mechanism must be used, to translate the management operations to operations on the resources. This translation mechanism is usually called *instrumentation*, and the function that implements it is called . The instrumentation functions are written for each combination of management protocol and resource to be managed. For example, if an application is to be managed by SNMP and HTTP, two sets of instrumentation functions are defined; one that maps SNMP requests to the resources, and one that e.g. generates an HTML page for some resources.

When a manager makes a request to the agent, we have the following picture:

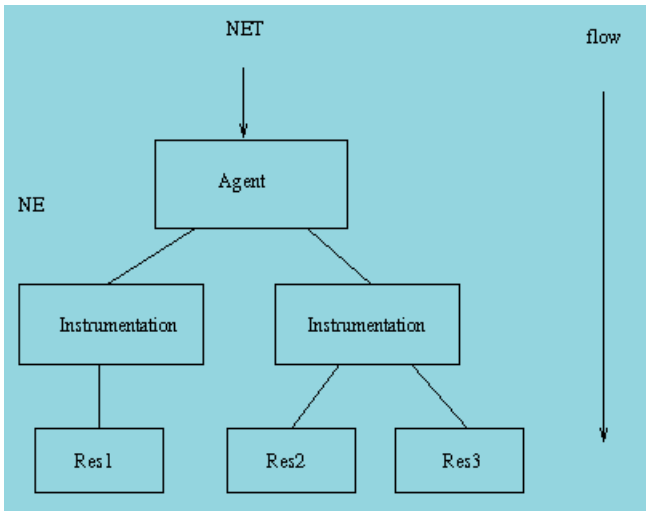

Figure 1.2: Request to an agent by a manager

Note that the mapping between instrumentation function and resource is not necessarily 1-1. It is also possible to write one instrumentation function for each resource, and use that function from different protocols.

The agent receives a request and maps this request to calls to one or several instrumentation functions. The instrumentation functions perform operations on the resources to implement the semantics associated with the managed object.

For example, a system that is managed with SNMP and HTTP may be structured in the following way:

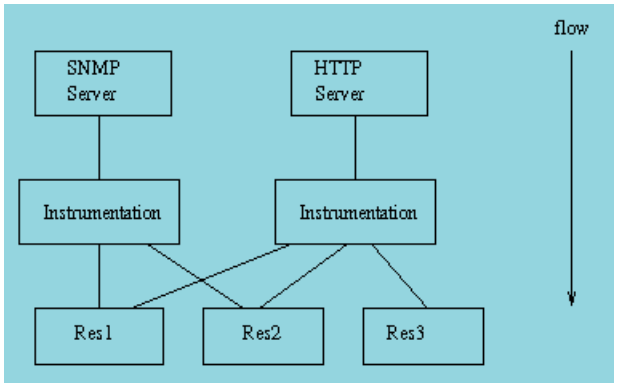

Figure 1.3: Structure of a system managed with SNMP and HTTP

The resources may send notifications to the manager as well. Examples of notifications are events and alarms. There is a need for the resource to generate protocol independent notifications. The following picture illustrates how this is achieved:

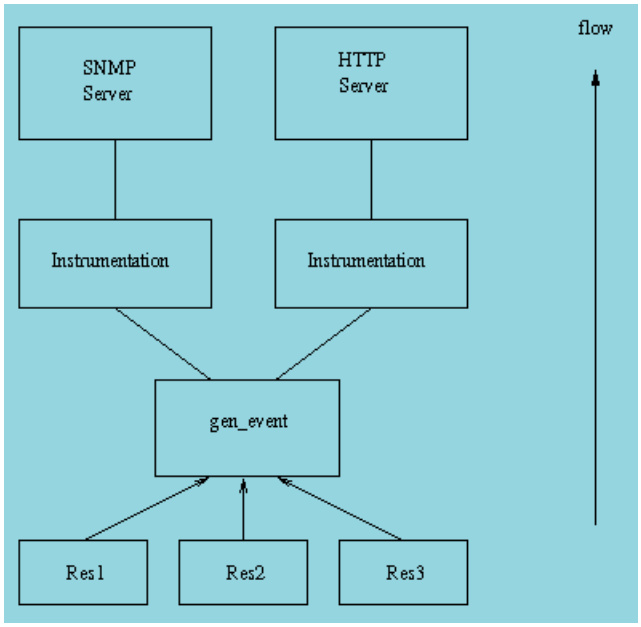

Figure 1.4: Notification handling

The main idea is that the resource sends the notfications as Erlang terms to a dedicated `gen_event` process. Into this process, handlers for the different management protocols are installed. When an event is received by this process, it is forwarded to each installed handler. The handlers are responsible for translating the event into a notification to be sent over the management protocol. For example, a handler for SNMP would translate each event into an SNMP trap.

10.1.3 SNMP based OAM

For all OAM components, SNMP adaptations are provided. Other adaptations may be defined in the future.

The OAM components, and some other OTP applications, define SNMP MIBs. All these MIBs are written in SNMPv2 SMI syntax, as defined in RFC1902. For convenience we also deliver the SNMPv1 SMI equivalent. All MIBs are designed to be v1/v2 compatible, i.e. the v2 MIBs do not use any construct not available in v1.

MIB structure

The top-level OTP MIB is called `OTP-REG`, and it is included in the `sasl` application. All other OTP mibs import some objects from this MIB.

Each MIB is contained in one application. The MIB text files are stored under `mibs/<MIB>.mib` in the application directory. The generated `.hrl` files with constant declarations are stored under `include/<MIB>.hrl`, and the compiled MIBs are stored under `priv/mibs/<MIB>.bin`. For example, the `OTP-MIB` is included in the `sasl` application:

```
sasl-1.3/mibs/OTP-MIB.mib
         include/OTP-MIB.hrl
         priv/mibs/OTP-MIB.bin
```

An application that needs to IMPORT this mib into another MIB, should use the `il` option to the snmp mib compiler:

```
snmp:c("MY-MIB", [{il, ["sasl/priv/mibs"]}]).
```

If the application needs to include the generated `.hrl` file, it should use the `-include_lib` directive to the Erlang compiler.

```
-module(my_mib).

-include_lib("sasl/include/OTP-MIB.hrl").
```

The following MIBs are defined in the OTP system:

OTP-REG (sasl)

This MIB contains the top-level OTP registration objects, used by all other MIBs.

OTP-TC (sasl)

This MIB contains the general Textual Conventions, which can be used by any other MIB.

OTP-MIB (sasl)

This MIB contains objects for instrumentation of the Erlang nodes, the Erlang machines and the applications in the system.

OTP-OS-MON-MIB (os_mon)

This MIB contains objects for instrumentation of disk, memory and cpu usage of the nodes in the system.

OTP-SNMPEA-MIB (snmp)

This MIB contains objects for instrumentation and control of the extensible snmp agent itself. Note that the agent also implements the standard SNMPv2-MIB (or v1 part of MIB-II, if SNMPv1 is used).

OTP-EVA-MIB (eva)

This MIB contains objects for instrumentation and control of the events and alarms in the system.

OTP-LOG-MIB (eva)

This MIB contains objects for instrumentation and control of the logs and FTP transfer of logs.

OTP-EVA-LOG-MIB (eva)

This MIB contains objects for instrumentation and control of the events and alarm logs in the system.

OTP-SNMPEA-LOG-MIB (eva)

This MIB contains objects for instrumentation and control of the snmp audit trail log in the system.

The different applications use different strategies for loading the MIBs into the agent. Some MIB implementations are code-only, while others need a server. One way, used by the code-only mib implementations, is for the user to call a function such as `otp_mib:init(Agent)` to load the MIB, and `otp_mib:stop(Agent)` to unload the MIB. See the application manual page for each application for a description of how to load each MIB.