System Principles

version 5.5

Typeset in LaTeX from SGML source using the DocBuilder-0.9 Document System.

Contents

Chapter 1

System Principles

1.1 System Principles

1.1.1 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)`.

1.1.2 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.

1.1.3   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 [page 3].

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)`.

1.1.4 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)`.

1.1.5 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

1.2   Error Logging

1.2.1   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.

1.2.2   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>
```

1.3   Creating a First Target System

1.3.1   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.

1.3.2 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:

```
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 2. Now create the target system:

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

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

1. 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.

2. 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`.

3. 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.

4. Creates the temporary directory `tmp` and extracts the tar file `mysystem.tar.gz` into that directory.

5. Deletes the `erl` and `start` files from `tmp/erts-5.1/bin`. XXX Why.

6. Creates the directory `tmp/bin`.

7. Copies the previously creates file `plain.boot` to `tmp/bin/start.boot`.

8. Copies the files `epmd`, `run_erl`, and `to_erl` from the directory `tmp/erts-5.1/bin` to the directory `tmp/bin`.

9. Creates the file `tmp/releases/start_erl.data` with the contents "5.1 FIRST".

10. Recreates the file `mysystem.tar.gz` from the directories in the directory `tmp`, and removes `tmp`.

1.3.3   Installing a Target System

Step 3. Install the created target system in a suitable directory.

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

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

1. Extracts the tar file `mysystem.tar.gz` into the target directory `/usr/local/erl-target`.

2. In the target directory reads the file `releases/start_erl.data` in order to find the Erlang runtime system version ("5.1").

3. 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.

4. Finally the target `releases/RELEASES` file is created from data in the `releases/mysystem.rel` file.

1.3.4   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

```
/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:

```
/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.

1.3.5 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.

1.3.6 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.

1.3.7 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:rawopen(Src, {binary, read}),
    {ok, OutFd} = file:rawopen(Dest, {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).
```

System Principles

13

```
            case FileInfo#file_info.type of
                directory ->
                    {ok, DirFiles} = file:list_dir(FilePath),
```

List of Tables