System Principles

version 5.0
Typeset in \LaTeX{} from SGML source using the DOCBUILDER 3.0 Document System.
## Contents

1 System Principles ................................................................. 1
   1.1 System Principles ............................................................ 2
   Starting the System ............................................................. 2
   Restarting and Stopping the System ........................................... 3
   Command Line Arguments ..................................................... 4
   The Boot File ................................................................. 5
   Making a Boot File ........................................................... 6
   Starting the System with a Boot File ........................................ 6
   Code Loading Strategy ....................................................... 10
   Making an Embedded System ................................................. 11
   The Primitive Loader ....................................................... 11
   File Types ................................................................. 12

List of Tables ............................................................................ 13
Chapter 1

System Principles
This chapter describes the strategies and options which are available to start the Erlang runtime system of Erlang/OTP. This section includes the following topics:

- Starting the system
- Re-starting and stopping the system
- Command line arguments
- The boot file
- Code loading strategies
- Making a boot file
- Starting the system with a boot file
- Code loading strategy
- Making an embedded system
- The primitive loader.

**Starting the System**

An Erlang runtime system is started with the command:

```
erl [-boot B] [-config F] [-mode M] [-heart]
    [-loader L] [-id Id] [-nodes N1 N2 ... Nn]
    [-pa Dir1 Dir2 ... Dirn] [-pz Dir1 Dir2 ... Dirn]
    [-path Dir1 Dir2 ... Dirn]
    [-AppName Key Value]
    [Other args]
```

- **-boot B** tells the system to use the boot file named `B.boot` to boot the system. This boot file is responsible for the initial loading of the system. If `B` is not supplied it defaults to `start`.
  When Erlang starts, it searches for the boot file in the current working directory and then in `$ROOT/bin`, where `$ROOT` is the root of the Erlang distribution.
  If `-loader distributed` was specified, the script is fetched from one of the nodes N1, N2, ..., otherwise it is fetched by requesting it from the program given in the argument to the `-loader` parameter.
- **-config F** tells the system to use data in the system configuration file `F.config` to override the arguments contained in the application resource files for the set of applications used by this system.
- **-mode M** is the mode in which the system is started. M must be either `embedded` or `interactive`.
  If `-mode M` is omitted, it defaults to `interactive`. In embedded mode all, modules are loaded at system start.
- **-heart** This argument starts an external program which monitors the Erlang node. If the Erlang node hangs, or terminates abnormally, the `heart` program can restart the Erlang node.
-loader L defines the loader program L which fetches code when the system is started. L is either the atom distributed, or the name of an external program. If L is not supplied, it defaults to efile which is the normal Erlang filer.

-Id Id gives a unique identifier to each Erlang node. If omitted, Id defaults to the atom none. This flag is not required if the default loader efile is used. If the -name Name or -name Name parameters are given, Id must be the same as Name.

-nodes N1 N2 ... Nn must be supplied if -loader distributed is specified. N1, N2, ..., Nn are Erlang node names from which the system can fetch the modules to be loaded.

-pa Dir1 Dir2 ... Dirn defines a set of directories, Dir1, Dir2, .. Dirn which are added to the front of the standard search path which is defined in the start-up script.

-pz Dir1 Dir2 ... Dirn defines a set of directories, Dir1, Dir2, .. Dirn which are added to the end of the standard search path which is defined in the start-up script.

-path Dir1 Dir2 ... Dirn defines a set of directories, Dir1, Dir2, .. Dirn which replace the standard search path defined in the start-up script.

[-AppName Key Value] overrides the AppName application configuration parameter Key with Value.

[Other Args] are parsed in a standard manner and can be accessed by any application.

The following comments apply to the arguments listed above:

- The default loader is the program efile. Through erl_prim_loader, it provides a minimal file system interface between Erlang and the local file system in order to load code.

- When -loader L is specified, the primitive code loader must know how to retrieve a boot script with name B.boot.

- When -loader distributed -nodes N1 N2 ... Nn is specified, the boot servers with registered names boot_server are assumed to be running on all Erlang nodes N1, N2, ..., Nn. If they are not, the system waits for these boot servers to start. Requests are sent to these boot servers to obtain files with names {Id, Name} (Id is specified in the command line arguments). The boot servers must know how to map these names onto local file names. A simple boot server erl_boot_server is provided with the system.

- The boot file with extension .boot is created by evaluating the expression systools:script2boot("File"). This function converts the script file File.script to a boot file File.boot.

### Restarting and Stopping the System

The system is restarted and stopped with the following commands:

- init:restart(). This command restarts the system inside the running Erlang node. All applications are taken down smoothly, and all code is unloaded before the system is started again in the same way as it was started initially.

- init:reboot(). All applications are taken down smoothly, and all code is unloaded before the Erlang node terminates. The heart argument affects the reboot sequence as follows:
  1. If the -heart argument was supplied, the heart program tries to reboot according to the HEART_COMMAND environment variable.
  2. If this variable is not set, heart simply writes to std_out that it should have rebooted.
  3. If HEART_COMMAND is /usr/sbin/reboot, the whole machine is rebooted.
Chapter 1: System Principles

- init:stop(). All applications are taken down smoothly, and all code is unloaded. If the heart argument was supplied, the heart program is terminated before the Erlang node terminates.

**Command Line Arguments**

When the system has started, 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().

Erlang was started by giving a command of the form:

```
erl -flag1 arg1 arg2 -flag2 arg3 ...
```

When the `erl -flag1 ...` command has been issued, Erlang starts by spawning a new process and the system behaves as if the function `spawn(init, boot, [Args])` had been evaluated. `Args` is a list of all the command line arguments to `erl`. These are passed as strings. For example, the command `erl -id 123 -loader efile -script "abc" ...` causes the system to behave as if it had evaluated the following function:

```
spawn(init, boot, ["-id", "123", "-loader", "efile", "-script", "\"abc\"\"]).
```

The first thing `init` does is to call `init:parse_args(Args)` to “normalize” the input arguments. After normalization, the arguments can be accessed as follows:

- `init:get_argument(Flag) -> {ok, [[Arg]]} | error` tries to fetch the argument associated with `Flag`. The return value is either a list of argument lists, or the atom error. Flags can have multiple values. If the command line was `erl -p1 a b c -p2 a x -p1 ww zz`:
  - `init:get_argument(p1)` would return:
    `{ok, [["a", "b", "c"], ["ww", "zz"]]}`
  - `init:get_argument(p2)` would return:
    `{ok, [["a", "x"]]}`

This is why `get_argument` returns a list of lists, and not just a list.

- `init:get_arguments() -> [{Flag, [Arg]}]` returns all the command line arguments. For the command line given above, this would return:
  `{[p1,"a","b","c"],[p2,"a","x"],[p1,"ww","zz"]}`

Both `get_arguments/0` and `get_argument/1` preserve the argument order of the arguments supplied with the command line.

**Note:**

Applications should not normally be configured with command line flags, but should use the application environment instead. Refer to Configuring an Application in the Design Principles chapter for details.
The Boot File

The boot script is stored in a file with the extension .script. A typical boot script file may look as follows:

```
{script, {Name, Vsn},
 [  {progress, loading},
  {preLoaded, [Mod1, Mod2, ...]},
  {path, [Dir1,"$ROOT/Dir",...]},
  {primLoad, [Mod1, Mod2, ...]},
  ...
  {kernel_load_completed},
  {progress, loaded},
  {kernelProcess, Name, {Mod, Func, Args}},
  ...
  {apply, {Mod, Func, Args}},
  ...
  {progress, started}]).
```

The meanings of these terms are as follows:

- `{script, {Name, Vsn},...}` defines the script name and version.
- `{progress, Term}` sets the “progress” of the initialization program. The function `init:get_status()` returns the current value of the progress, which is `{InternalStatus,Progress}`.
- `{path, [Dir]}`. Dir is a string. This argument sets the load path of the system to [Dir]. The load path used to load modules is obtained from the initial load path, which is given in the script file, together with any path flags which were supplied in the command line arguments. The command line arguments modify the path as follows:
  - `-pa Dir1 Dir2 ...` Dirn adds the directories Dir1, Dir2, ..., Dirn to the front of the initial load path.
  - `-pz Dir1 Dir2 ...` Dirn adds the directories Dir1, Dir2, ..., Dirn to the end of the initial load path.
  - `-path Dir1 Dir2 ...` Dirn defines a set of directories Dir1, Dir2, ..., Dirn which replaces the search path given in the script file. Directory names in the path are interpreted as follows:
    - Directory names starting with `/` are assumed to be absolute path names.
    - Directory names not starting with `/` are assumed to be relative to the current working directory.
    - The special `$ROOT` variable can only be used in the script, not as a command line argument. The given directory is relative to the Erlang installation directory.
- `{primLoad, [Mod]}` loads the modules [Mod] from the directories specified in Path. The script interpreter fetches the appropriate module by calling the function `erl_prim_loader:get_file(Mod)`. A fatal error which terminates the system will occur if the module cannot be located.
- `{kernel_load_completed}` indicates that all modules which must be loaded before any processes are started are loaded. In interactive mode, all `{primLoad, [Mod]}` commands interpreted after this command are ignored, and these modules are loaded on demand. In embedded mode, `kernel_load_completed` is ignored, and all modules are loaded during system start.
Chapter 1: System Principles

- \{kernelProcess, Name, \{Mod, Func, Args\}\} starts a "kernel process". The kernel process Name is started by evaluating apply\(\text{Mod, Func, Args}\) which is expected to return \{ok, Pid\} or ignore. The init process monitors the behaviour of Pid and terminates the system if Pid dies. Kernel processes are key components of the runtime system. Users do not normally add new kernel processes.

- \{apply, \{Mod, Func, Args\}\}, The init process simply evaluates apply\(\text{Mod, Func, Args}\). The system terminates if this results in an error. The boot procedure hangs if this function never returns.

**Note:**

In the interactive system the code loader provides demand driven code loading, but in the embedded system the code loader loads all the code immediately. The same version of code is used in both cases. The code server calls init: get_argument\(\text{mode}\) to find out if it should run in demand mode, or non-demand driven mode.

**Making a Boot File**

If a boot script is written manually, the systools: script2boot\(\text{File}\) function can be used to generate the compiled (binary) form \text{File.boot} from the \text{File.script} file. However, it is recommended that the systools: make\{script\} function is used in order to create a boot script.

**Starting the System with a Boot File**

The command \text{erl -boot File} starts the system with a boot file called \text{File.boot}. An ASCII version of the boot file can be found in \text{File.script}.

The boot file is created by evaluating:

\[
\text{systools:script2boot\(\text{File}\)}
\]

Several standard boot files are available. For example, \text{start.script} starts the system as a plain Erlang runtime system with the application\_controller and the kernel applications.

\text{start.script}

The \text{start.script} is as follows:

\[
\{\text{script},\{\"OTP APN 181 01\","R1A\"\},\
  \{\text{preLoaded,\{init,erl_prim_loader\}},\
  \{\text{progress,preloaded}\},\
  \{\text{path,\[$\text{ROOT/lib/kernel-1.1/ebin}$,\
    "$\text{ROOT/lib/stdlib-1.1/ebin}$\]\}},\
  \{\text{primLoad,\{error_handler,ets,lib,\
    \}}\}
\]
\]
lists,
slave,
heart,
application_controller,
application_master,
application,
auth,
calendar,
code,
erlang,
erl_distribution,
erl_parse,
erl_scan,
io_lib,
io_lib_format,
io_lib_fread,
io_lib_pretty,
error_logger,
file,
gen,
gen_event,
gen_server,
global,
kernel,
net_kernel,
proc_lib,
rpc,
supervisor,
sys}}),
{kernal_load_completed},
{progress,kernel_load_completed},
{primLoad,[group,
user,
user_drv,
kernal_config,
net,
erl_boot_server,
net_adm]},
{primLoad,[math,
random,
ordsets,
shell_default,
timer,
gen_fsm,
pg,
unix,
dict,
pool,
string,
digraph,
io,
ep}]
Chapter 1: System Principles

log_mf_h,
queue,
erl_eval,
erl_id_trans,
shell,
erl_INTERNAL,
erl_lint,
error_logger_file_h,
error_logger_tty_h,
edlin,
erl_pp,
dets,
regexp,
supervisor_bridge]],
{progress,modules_loaded},
{kernelProcess,heart,{heart,start,[]}}],
{kernelProcess,error_logger,{error_logger,start_link,[]}}],
{kernelProcess,application_controller, application_controller, start,

[[{application, kernel, [{description,"ERTS CXC 138 10"}], {vsn,"1.1"}],
{modules, [{application,1},
erlang,1},
{group,1},
{rpc,1},
{application_controller,1},
{error_handler,1},
{heart,1},
{application_master,1},
{error_logger,1},
{init,1},
{user,1},
{auth,1},
{kernel,1},
{user_drv,1},
{code,1},
{kernel_config,1},
{net,1},
erl_boot_server,1},
erl_prim_loader,1},
{file,1},
{net_adm,1},
erl_distribution,1},
{global,1},
{net_kernel,1]}],
{registered, [{init, erl_prim_loader, heart,}]}]},
1.1: System Principles

error_logger,
application_controller,
kernel_sup,
kernel_config,
net_sup,
net_kernel,
auth,
code_server,
file_server,
boot_server,
global_name_server,
rex,
user},
{applications,[]},
{env,
 [{error_logger,tty},
  {os, {unix, 'solaris'}}],
 {maxT, infinity},
 {maxP, infinity},
 {mod, {kernel, []}}]],
{progress, {init_kernel_started}},
{apply, {application, load,
 [{application,
   stdlib,
   [{description,"ERTS CXC 138 10"},
    {ver,"1.1"}],
    {modules,
     [{c,1},
      {gen,1},
      {io_lib_format,1},
      {math,1},
      {random,1},
      {sys,1},
      {calendar,1},
      {gen_event,1},
      {io_lib_fread,1},
      {ordsets,1},
      {shell_default,1},
      {timer,1},
      {gen_fsm,1},
      {io_lib.pretty,1},
      {pg,1},
      {slave,1},
      {unix,1},
      {dict,1},
      {gen_server,1},
      {lib,1},
      {pool,1},
      {string,1},
      {digraph,1},
      {io,1},
      {lists,1},
      {proc_lib,1}],

System Principles
Code Loading Strategy

The code is always loaded relative to the current path and this path is obtained from the value given in the script file, possibly modified by the path manipulation flags in the command line.

This approach allows us to run the system in a number of different ways:

- Interactive mode. The system dynamically loads code on demand from the directories specified in the path command. This is the "normal" way to develop code.
- Embedded mode. The system loads all its code during system start-up. In special cases, all code can be located in a single directory. We would copy all files to a given directory and create a path to this directory only.
- Test mode. Test mode is typically used if we want to run some new test code together with a particular release of the embedded system. We want all the convenience of the interactive system.
1.1: System Principles

with code loading on demand, and the rigor of the embedded system. In test mode, we run the
system with command line arguments such as `-pa ".".

Making an Embedded System

When using the the interactive Erlang development environment, it often does not matter if things go
wrong at runtime. The main difference with an embedded system is that it is extremely important that
things do not go wrong at runtime.

Before building a release which is targeted for an embedded system, we must perform a large number of
compile-time checks on the code.

A boot script file can be created with the `systools:make_script` function. This function reads a .rel
release file and generates the boot script in accordance with the specified applications in the release file.
A boot script which is generated this way ensures that all code specified in the application resource files
are loaded and that all specified applications are started.

A complete release can be packaged with the `systools:make_tar` function. All application directories
and files are packaged according to the release file. The release file and the release upgrade script are
also included in the release package.

The Primitive Loader

Unlike the Erlang node, the primitive file loader “knows” how to fetch modules and scripts from its
environment.

The interface to the primitive loader is as follows:

- `erl_prim_loader:start(Id, L, Nodes) -> ok | error` starts the primitive loader with the
  arguments given in the command line.
- `erl_prim_loader:set_path([Dir]) -> ok` sets the path given in the boot file. The value of
  `[Dir]` comes from the command `{path, [Dir]} in the start-up script combined with the
  command line arguments.
- `erl_prim_loader:get_path() -> {ok, Path}` returns the Path used by the primitive loader.
- `erl_prim_loader:get_file(File) -> {ok, FullName, Bin} | error` loads a file from the
  current path. File is either an absolute file name or just the name of the file, for example
  `lists.beam`. FullName is the name of the file if the load succeeds. Bin is the contents of the file
  as a binary.

Note:

We assume the primitive loader to be running as long as the Erlang node is up and running. In the
interactive mode, the code server fetches all code through the loader and the
`application_controller` fetches configuration and application files this way.

If an other loader than the one distributed with the system is required, this loader must be implemented
by the user as an external port program. The `Loader` provided by the user must fulfill a protocol
defined for the `erl_prim_loader`, and it will be started by the `erl_prim_loader` using the
open_port([\{spawn,\text{\textit{Loader}}\},[\text{\textit{binary}}]]) function call. Refer to the Reference Manual for more information.

**File Types**

The following file types are defined in Erlang/OTP:

<table>
<thead>
<tr>
<th>Type</th>
<th>File name/Extension</th>
<th>Description</th>
<th>Manual page which describes the file syntax</th>
</tr>
</thead>
<tbody>
<tr>
<td>module</td>
<td>.erl</td>
<td>Erlang code</td>
<td>-</td>
</tr>
<tr>
<td>application</td>
<td>.app</td>
<td>Application resource file</td>
<td>app(4)</td>
</tr>
<tr>
<td>release</td>
<td>.rel</td>
<td>Release resource file</td>
<td>rel(4)</td>
</tr>
<tr>
<td>script</td>
<td>.script</td>
<td>Start script</td>
<td>script(4)</td>
</tr>
<tr>
<td>boot</td>
<td>.boot</td>
<td>Binary boot file</td>
<td>-</td>
</tr>
<tr>
<td>config</td>
<td>.config</td>
<td>Configuration file - used to override values in the \textit{.app} files</td>
<td>config(4)</td>
</tr>
<tr>
<td>application upgrade</td>
<td>.appup</td>
<td>Application upgrade</td>
<td>appup(4)</td>
</tr>
<tr>
<td>release upgrade script</td>
<td>.relup</td>
<td>Release upgrade script</td>
<td>relup(4)</td>
</tr>
</tbody>
</table>

Table 1.1: File Types
List of Tables

Chapter 1: System Principles
1.1 File Types .............................................. 12