OTP Design Principles

version 5.3

Typeset in LaTeX from SGML source using the DOCBUILDER 3.3.2 Document System.

Contents

Chapter 1

OTP Design Principles

1.1 Overview

The *OTP Design Principles* is a set of principles for how to structure Erlang code in terms of processes, modules and directories.

1.1.1 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.

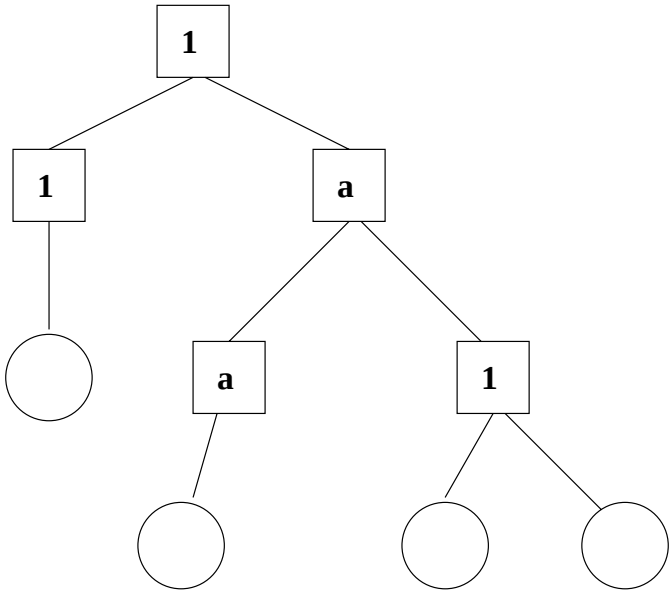

Figure 1.1: Supervision Tree

In the figure above, square boxes represents supervisors and circles represent workers.

1.1.2 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.

The module server corresponds, greatly simplified, to the Erlang/OTP behaviour gen_server.

The standard Erlang/OTP behaviours are:

gen_server [page 6] For implementing the server of a client-server relation.

gen_fsm [page 10] For implementing finite state machines.

gen_event [page 14] For implementing event handling functionality.

supervisor [page 17] For implementing a supervisor in a supervision tree.

The compiler understands the module attribute `-behaviour(Behaviour)` and will issue warnings about missing callback functions. Example:

```
-module(chs3).
-behaviour(gen_server).
...
```

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 will also make 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.

1.1.3 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 [page 29].

1.1.4 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 [page 42].

How to install a release in a target environment is described in the chapter about Target Systems in System Principles.

1.1.5 Release Handling

Release handling is upgrading and downgrading between different releases, in a (possibly) running system. How to do this is described in Release Handling [page 47].

1.2 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.

1.2.1 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.

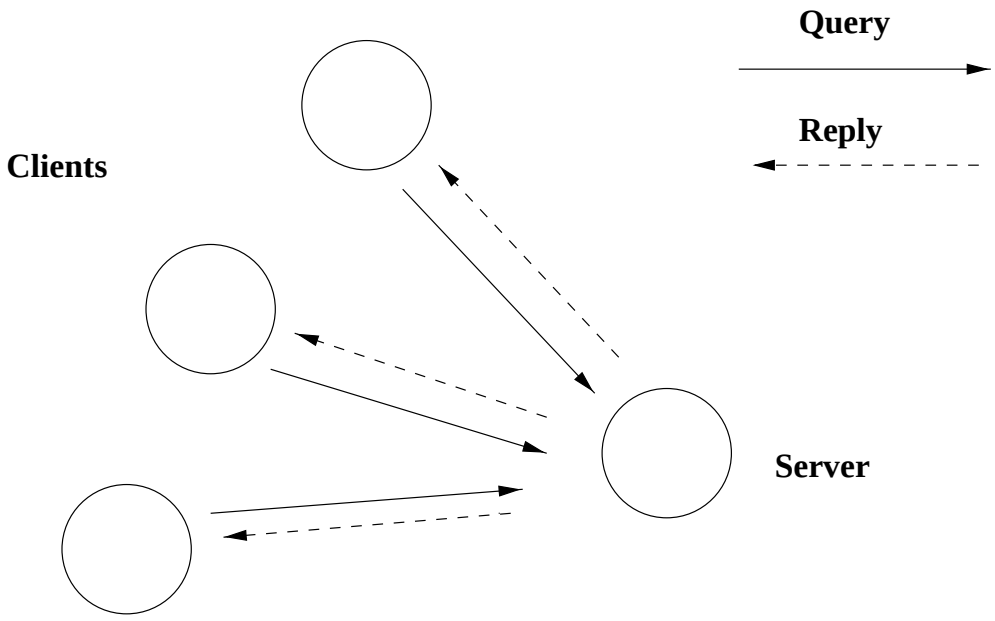

Figure 1.2: Client-Server Model

1.2.2 Example

An example of a simple server written in plain Erlang was given in Overview [page 2]. 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.

1.2.3 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.

1.2.4 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 remaining available channels Chs2.

Thus, the call ch3:alloc() will return the allocated channel Ch and the gen_server then waits for new requests, now with an updated list of available channels.

1.2.5 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 server.

```
handle_call({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.

1.2.6 Stopping

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 calling `exit(Pid, shutdown)`.

The gen_server process does not trap exit signals. If it is necessary to do some cleaning up before termination, it should be set to trap exit signals in the `init` function and another callback function `terminate(Reason, State)` should be implemented doing the cleaning up:

```
init(Args) ->
    ...,
    process_flag(trap_exit, true),
    ...,
    {ok, State}.

...

terminate(shutdown, State) ->
    ..code for cleaning up here..
    ok.
```

If the gen_server is not part of a supervision tree, a stop function may be useful. The gen_server will automatically call the callback function `terminate(Reason, State)` if another of the callback functions returns `{stop, Reason, State2}` instead of `{reply,...}` or `{noreply,...}`. 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.
```

1.2.7 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 on other messages could be for example timeouts or 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, State}.
```

1.3 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.

1.3.1 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'}
```

1.3.2 Example

A door with a code lock could be viewed as an FSM. Initially, the door is closed. When someone presses a button, this generates an event. If the button sequence pressed so far is the correct code, the door is opened and held open for 30 seconds (30000 milliseconds). If the button sequence is incomplete, the door remains closed and we wait for another button to be pressed. If the button sequence 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, closed/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, closed, {[], Code}}.

closed({button, Digit}, {SoFar, Code}) ->
```

```
    case [Digit|SoFar] of
        Code ->
            do_open(),
            {next_state, open, {[], Code}, 3000};
        Incomplete when length(Incomplete)<length(Code) ->
            {next_state, closed, {Incomplete, Code}};
        _Wrong ->
            {next_state, closed, {[], Code}};
    end.

open(timeout, State) ->
    do_close(),
    {next_state, closed, State}.
```

The code is explained in the next sections.

1.3.3 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`, `closed` 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)`. `init` is expected to return `{ok, StateName, StateData}`, where `StateName` is the name of the initial state of the gen_fsm. In this case `closed`, assuming the door is closed 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, closed, {[], 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.

1.3.4 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.

```
closed({button, Digit}, {SoFar, Code}) ->
    case [Digit|SoFar] of
        Code ->
            do_open(),
            {next_state, open, {[], Code}, 30000};
        Incomplete when length(Incomplete)<length(Code) ->
            {next_state, closed, {Incomplete, Code}};
        _Wrong ->
            {next_state, closed, {[], Code}};
    end.

open(timeout, State) ->
    do_close(),
    {next_state, closed, State}.
```

If the door is closed 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 opened and the gen_fsm goes to state `open`, or the door remains in state `closed`.

1.3.5 Timeouts

When a correct code has been givened, the door is opened and the following tuple is returned from `closed/2`:

```
{next_state, open, {[], Code}, 30000};
```

30000 is a timeout value in milliseconds. After 30000 milliseconds, 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 and then the door is closed:

```
open(timeout, State) ->
    do_close(),
    {next_state, closed, State}.
```

1.3.6 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}.
```

1.3.7 Stopping

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 calling `exit(Pid, shutdown)`.

The gen_fsm process does not trap exit signals. If it is necessary to do some cleaning up before termination, it should be set to trap exit signals in the `init` function and another callback function `terminate(Reason, State)` should be implemented doing the cleaning up:

```
init(Args) ->
    ...,
    process_flag(trap_exit, true),
    ...,
    {ok, StateName, StateData}.

...

terminate(shutdown, StateName, StateData) ->
    ..code for cleaning up here..
    ok.
```

If the gen_fsm is not part of a supervision tree, a stop function may be useful. The gen_fsm will automatically call the callback function `terminate(Reason, StateName, StateData)` if another of the callback functions returns `{stop, Reason, State2}` instead of `{next_state,...}`. Example:

```
-module(code_lock).
...
-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.
```

1.3.8 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 on other messages could be 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}.
```

1.4 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.

1.4.1 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 managar 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.

1.4.2 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.

1.4.3 Starting an Event Manager

Here is an example of how an event manager for handling errors, as described in the example above, is started from the shell:

```
1> gen_event:start_link({local, error_man}).
{ok,<0.30.0>}
```

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.

1.4.4 Adding an Event Handler

```
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 fo the event handler.

```
init(_Args) ->
    {ok, []}.
```

Here, init does not need any indata 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(_Args) ->
    {ok, Fd} = file:open(File, read),
    {ok, Fd}.
```

1.4.5 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}.
```

1.4.6 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).
```

1.4.7 Stopping

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 calling `exit(Pid, shutdown)`.

An event manager can also be stopped by calling:

```
> gen_event:stop(error_man).
ok
```

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.

1.5 Supervisor Behaviour

This section should be read in conjunction with `supervisor(3)`, where all details about the supervisor behaviour is given.

1.5.1 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 [page 20]. The child processes are started in the order specified by this list, and terminated in the reversed order.

1.5.2 Example

The callback module for a supervisor starting the server from the gen_server chapter [page 6] 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 [page 18].

1 and 60 defines the maximum restart frequency [page 19].

The tuple `{ch3, ...}` is a child specification [page 20].

1.5.3 Restart Strategy

one_for_one

If a child process terminates, only that process is restarted.

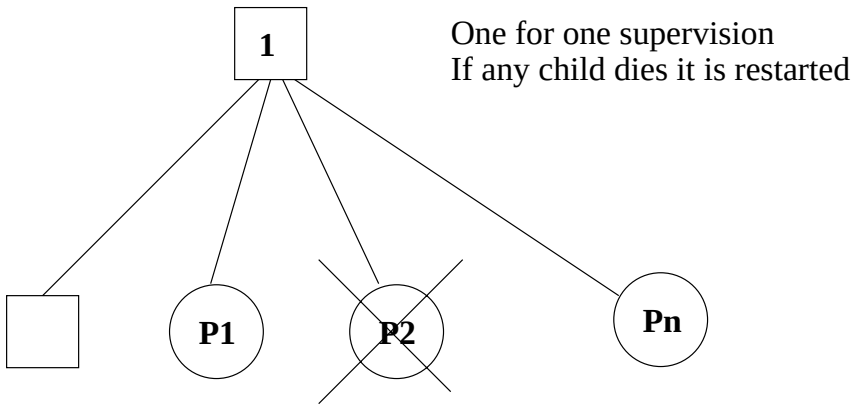

Figure 1.3: One_For_One Supervision

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.

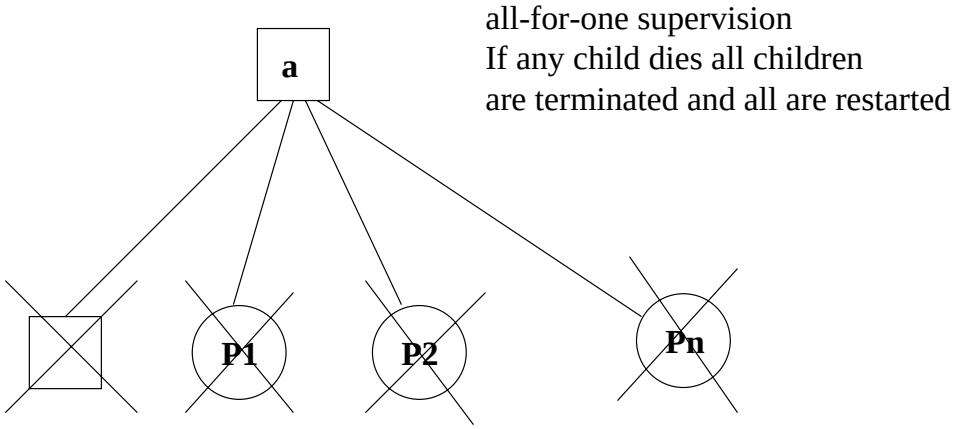

Figure 1.4: One_For_All Supervision

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.

1.5.4 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.

1.5.5 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 [page 47].

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 [page 15]:

```
{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]}
```

1.5.6 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.

1.5.7 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 [page 20].

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.

1.5.8 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 [page 20].

As with dynamically added child processes, the effects of deleting a static child process is lost if the supervisor itself restarts.

1.5.9 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)
```

1.5.10 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.

1.6 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 behaviour.

1.6.1 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 [page 10] 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]]}
```

1.6.2 Special Processes

This section describes how to write a process which comply to the OTP design principles, without making use of a 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 behaviours automatically understand these messages.

Example

The simple server from the Overview [page 2] 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 some information, for example about the ancestors and initial call, that is needed for a process in a supervision tree.

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()]).
```

start_link calls the function proc_lib:start_link. This functions spawns and links to a new process, executing ch4:init(Pid), where Pid is the pid of process calling ch4:start_link() (the parent process).

In init, all initialization including name registration is done. The 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, a term Deb is initialized in init 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*, the following function should be called:

```
sys:handle_debug(Deb, Func, Info, Event)
```

System events are:

- Incoming messages, represented as {in, Msg} or {in, Msg, From}.
- Outgoing messages, represented as {out, Msg, To}.
- The user may define own system events, represented by an arbitrary term.

Func is a tuple {Module, Name} (or a fun) and should be a user defined function used to format trace output. The function is called as Module:Name(Dev, Event, Info).

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 information.
- `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 behaviour 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.
```

1.7 Applications

This chapter should be read in conjunction with `app(4)` and `application(3)`.

1.7.1 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 [page 30], 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 [page 31]. 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 [page 42]), the code for each application is placed in a separate directory following a pre-defined directory structure [page 32].

1.7.2 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 [page 37]. `StartArgs` is defined by the key `mod` in the application resource 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 [page 33].

Example of an application callback module for packaging the supervision tree from the Supervisor [page 18] 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.

1.7.3 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 [page 42]), 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)`.

1.7.4 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. If source code is written in several different languages, a sub-directory with the name `e_src` can be created below the `src` directory to store the Erlang source code.

`ebin` Contains the Erlang object code, the `beam` files. The application resource 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.

1.7.5 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.

1.7.6 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.

1.7.7 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 similar to calling `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.

1.7.8 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"}]}
 ]}.
```

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 application resource file can be overridden by values in a *system configuration file*, see config(4). The command line argument -config Name tells the system to use data in the system configuration file Name.config.

The system configuration file should contain a list with one element for each application, a tuple {Application, [{Par1,Val1},...,{ParN,ValN}]}.

For example, a file test.config could be created with the following contents:

```
[{ch_app, [{file, "testlog"}]}].
```

Example:

```
% 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"}
```

The values in the application resource 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"}
```

1.7.9 Application Modes

An application is started in one of three modes. The mode is defined when starting the application using `application:start(Application, Mode)` and specifies what happens if the application terminates.

`application:start(Application)` is the same as calling `application:start(Application, temporary)`. The mode 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`.

1.8 Included Applications

1.8.1 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*.

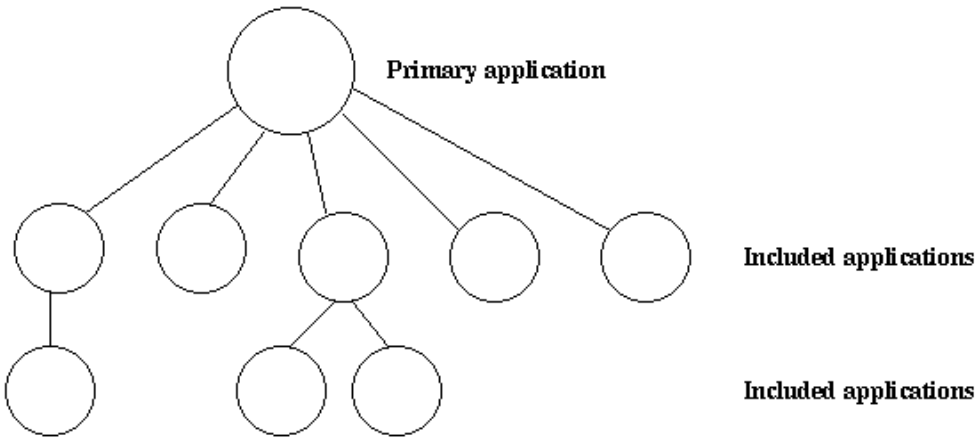

Figure 1.5: Primary Application and Included Applications.

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.

1.8.2 Specifying Included Applications

Which applications to include is defined by the `included_applications` key in the `.app` file.

```
{application, treeapp,
 [{description, "Tree application"},
  {vsn, "1"},
  {modules, [treeapp_cb, treeapp_sup, treeapp_server]},
  {registered, [treeapp_server]},
  {included_applications, [subtreeapp]},
  {applications, [kernel, stdlib, sasl]},
  {mod, {treeapp_cb,[]}},
  {env, [{file, "/usr/local/log"}]}
 ]}.
```

1.8.3 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.

```
{application, treeapp,
 [{description, "Tree application"},
  {vsn, "1"},
  {modules, [treeapp_cb, treeapp_sup, treeapp_server]},
  {registered, [treeapp_server]},
  {included_applications, [subtreeapp]},
  {start_phases, [{init,[]}, {go,[]}]},
  {applications, [kernel, stdlib, sasl]},
  {mod, {treeapp_cb,[]}},
  {env, [{file, "/usr/local/log"}]}
 ]}.

{application, subtreeapp,
 [{description, "Included application"},
  {vsn, "1"},
  {modules, [subtreeapp_sup, subtreeapp_server]},
  {registered, []},
  {start_phases, [{go,[]}]},
  {applications, [kernel, stdlib, sasl]},
  {mod, {subtreeapp_cb,[]}}
 ]}.
```

A start phase is defined as a tuple `{Phase,PhaseArgs}`, where `Phase` is an atom and `PhaseArgs` is a term.

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.
- 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 including application.

1.9 Distributed Applications

1.9.1 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 standard Erlang modules `global` or `pg` can be used for this purpose.

1.9.2 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.

1.9.3 Starting 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 [page 42]) 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.

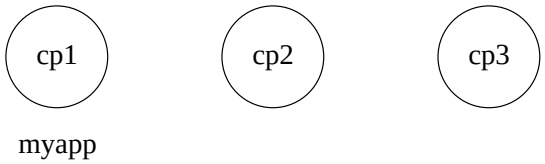

Figure 1.6: Application myapp - Situation 1

1.9.4 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 [page 35]), 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`.

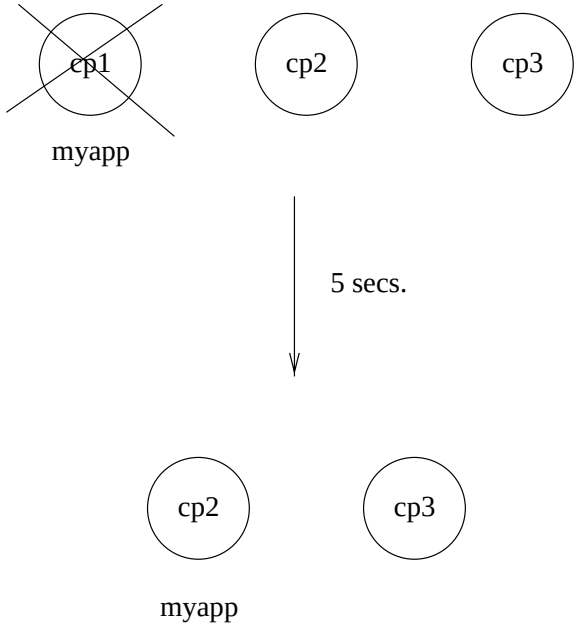

Figure 1.7: Application myapp - Situation 2

Suppose now that `cp2` goes down as well and does not restart within 5 seconds. `myapp` is now restarted on `cp3`.

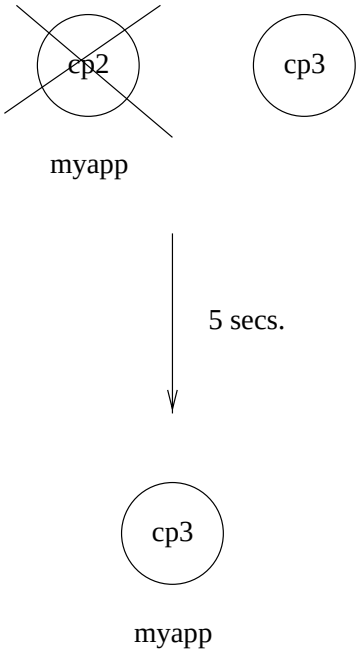

Figure 1.8: Application myapp - Situation 3

1.9.5 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.

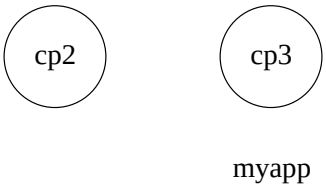

Figure 1.9: Application myapp - Situation 4

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.

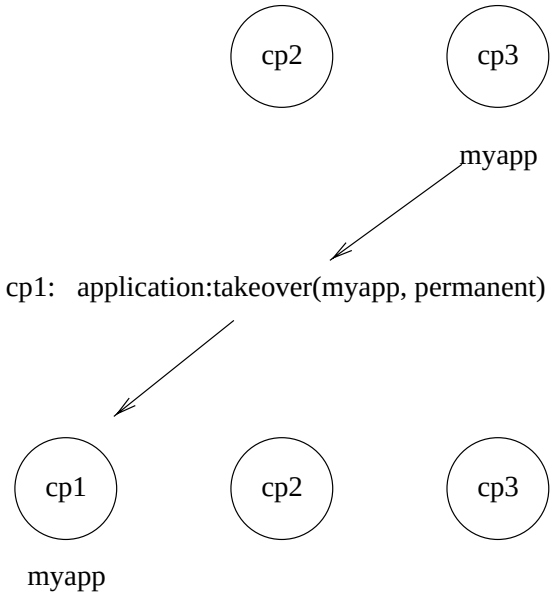

Figure 1.10: Application myapp - Situation 5

1.10 Releases

This chapter should be read in conjuction with rel(4), systools(3) and script(4).

1.10.1 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 [page 42] 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.

1.10.2 Release Resource File

To define a release, we create an *release resource file*, or in short .rel file:

```
{release, {Name,Vsn}, {erts, EVsn},
 [{Application1, AppVsn1},
   ...
  {ApplicationN, AppVsnN}]}.
```

The file must be named Rel.rel, where Rel is a unique name.

Name and Vsn (strings) are the name and version of the release.

Evsn (string) is the ERTS version the release is intended for.

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.

Example: We want to make a release of ch_app from the Applications [page 29] 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 release resource 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", "1"},
 {erts, "5.2.3.6"},
 [{kernel, "2.8.1.3"},
  {stdlib, "1.11.4.3"},
  {sasl, "1.9.4"},
  {ch_app, "1"}]
}.
```

1.10.3 Generating Boot Scripts

There are tools in the module systools available to build and check releases. The functions read the release resource file and the application resource files and performs syntax and dependency checks. The function systools:make_script/1,2 is used to generate a boot script.

```
1> systools:make_script("ch_rel-1", [local]).
ok
```

This creates a boot script ch_rel-1.script. "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.

A binary version ch_rel-1.boot of the script must be generated:

```
2> systools:script2boot("ch_rel-1").
ok
```

When starting Erlang/OTP using the boot script, all applications from the `.rel` file are automatically started:

```
% erl -boot ch_rel-1
Erlang (BEAM) emulator version 5.2.3.6 [hipe] [threads:0]

Eshell V5.2.3.6  (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
```

1.10.4 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:script2boot("ch_rel-1").
ok
3> 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 [page 32], the binary boot script renamed to `start.boot`, and the release resource file.

```
% tar tf ch_rel-1.tar
lib/kernel-2.8.1.3/ebin/kernel.app
lib/kernel-2.8.1.3/ebin/application.beam
...
lib/stdlib-1.11.4.3/ebin/stdlib.app
lib/stdlib-1.11.4.3/ebin/beam_lib.beam
```

```
...
lib/sasl-1.9.4/ebin/sasl.app
lib/sasl-1.9.4/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/1/start.boot
releases/ch_rel-1.rel
```

Options can be set to make the release package include source code and the ERTS binary as well.

The release package should be unpacked and installed using the release handler. See System Principles for how this is done when installing a target system and Release Handling [page 47] for how to do it in an existing system.

1.10.5 Directory Structure

Directory structure for the code installed by the release handler from a release package:

```
$ROOTDIR/lib/App1-AVsn1/ebin
                       /priv
            /App2-AVsn2/ebin
                        /priv
            ...
            /AppN-AVsnN/ebin
                        /priv
        /erts-EVsn/bin
        /releases/Vsn
        /bin
```

`lib` Application directories.

The boot script and relup files should be located in the `releases/Vsn` directory. `Vsn` is the release version found in the release resource file.

`erts-EVsn/bin` Erlang runtime system executables.

`releases/Vsn .rel` file.

`bin` Top level Erlang executable program `erl`.

Applications are not required to be located under the `$ROOTDIR/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 $ROOTDIR 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 $ROOTDIR directory contains the following:

```
$ROOTDIR/...
        /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 ROOTDIR per type of machine. For each type, some of the directories specified for the ROOTDIR directory should be included:

```
$ROOTDIR/...
        /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 `$ROOTDIR/clients/Type1`.

1.11 Release Handling

1.11.1 Release Handling Principles

A new release is assembled into a *release package*. Such a package is installed in a running system by giving commands to the *release handler*, which is an SASL process. A system has a unique *system version*, which is updated whenever a new release is installed. The system version is the version of the entire system, not just the OTP version.

If the system consists of several nodes, each node has its own system version. Release handling can be synchronized between nodes, or be done at one node at a time.

Changes may require a node to be brought down. If that is the case and the system consists of several nodes, the release upgrade can be done as follows;

1. move all applications from the node to be changed to other nodes,
2. take down the node,
3. do the change,
4. restart the node and move the applications back.

There are several different types of releases:

Operating system change. Can only be done by taking down the node. This kind of change is not supported by the release handler and therefore has to be performed manually. It is not possible to roll back automatically to a previous release, if there is an error.

Application code or data change. The release is installed without bringing down the running node. Some changes, for example change of C-programs, may be done by shutting down and restarting the affected processes.

Erlang emulator change. Can only be made by taking down the node. However, the release handler supports this type of change.

1.11.2 Administering Releases

This section describes how to build and install releases. Also refer to the SASL Reference Manual, `release_handler`, for more details.

The following steps are involved in administering releases:

1. A release package is built by using release building commands in the `systools` module. The package is assembled from application specification files, code files, data files, and a file, which describes how the release is installed in the system.
2. The release package is transferred to the target machine, e.g. by using `ftp`.
3. The release package is unpacked, which makes the system version in the release package available for installation by the `release_handler`, which interprets the *release upgrade script*, containing instructions for updating to the new version. If an installation fails in some way, the entire system is restarted from the old system version.
4. When the installation is complete, the system version must be made *permanent*. When permanent, the new version is used if the system restarts.

It is also possible to reinstall an old version, or reboot the system from an old version. There are functions to remove old releases from disk as well.

1.11.3 File Structure

The file structure used in an OTP system is described in Release Directories [page 45]. There are two ways of using this file structure together with the release handler.

The simplest way is to store all user-defined applications under `$OTP_ROOT/lib` in the same way as other OTP applications. The release handler takes care of everything, from unpacking a release to the removal of it. The release packages should be stored in the *releases directory* (default `$OTP_ROOT/releases`). This is where `release_handler:unpack_release/1` searches for the packages, and where the release handler stores its files. Each package is a compressed `tar` file. The files in the `tar` file are named relative to the `$OTP_ROOT` directory. For example, if a new version (say 1.3) of the application `snmp` is contained in the release package, the files in the `tar` file should be named `lib/snmp-1.3/*`.

The second way is to store all user-defined applications in some other place in the file system. In this case, some more work has to be done outside the release handler. Specifically, the release packages must be unpacked in some way and the release handler must be notified of where the new release is located. The following three functions are available in the module `release_handler` to handle this case:

- `set_unpacked/2`
- `set_removed/1`
- `install_file/2`.

1.11.4 Release Installation Files

The following files must be present when a release is installed. All file names are relative to the *releases* directory.

- `ReleaseFileName.rel`
- `Vsn/relup`
- `Vsn/start.boot`
- `Vsn/sys.config`

The location of the *releases* directory is specified with the configuration parameter `releases_dir` (default `$OTP_ROOT/releases`). In a target system, the default location is preferred, but during testing it may be more convenient to let the release handler write its files in a user specified directory, than in the `$OTP_ROOT` directory.

The files listed above are either present in the release package, or generated at the target machine and copied to their correct places using `release_handler:install_file/2`.

`Vsn` is the system version string.

ReleaseFileName.rel

The `ReleaseFileName.rel` file contains the name of the system, version of the release, the version of `erts` (the Erlang runtime system) and the applications, which are parts of the release. The file must contain the following Erlang term:

```
{release, {Name, Vsn}, {erts, EVsn},
 [{App, AVsn} | {App, AVsn, AType} | {App, AVsn, [App]} |
    {App, AVsn, AType, [App]}]}.
```

`Name`, `Vsn`, `EVsn` and `AVsn` are strings, `App` and `AType` are atoms. `ReleaseFileName` is a string given in the call to `release_handler:unpack_release(ReleaseFileName)`. `Name` is the name of the system (the same as found in the boot file). This file is further described in Release Structure [page 45].

relup

The `relup` file contains instructions on how to install the new version in the system. It must contain one Erlang term:

```
{Vsn, [{FromVsn, Descr, RuScript}], [{ToVsn, Descr, RuScript}]}.
```

`Vsn`, `FromVsn` and `ToVsn` are strings, `RuScript` is a release upgrade script. `Descr` is a user defined parameter, which is not processed by any release handling functions. It can be used to describe the release to an operator. Finally, it will be returned by `release_handler:install_release/1` and `release_handler:check_install_release/1`.

There is one tuple `{FromVsn, Descr, RuScript}` for each old system version which can be upgraded to the new version, and one tuple `{ToVsn, Descr, RuScript}` for each old version to which the new version can be downgraded.

start.boot

The `start.boot` file is the compiled `start.script` file. It is used to boot the Erlang machine.

sys.config

The `sys.config` is the system configuration file.

1.11.5 Release Handling Principles

The following sections describe the principles for updating parts of an OTP system.

Erlang Code

The code change feature in Erlang is made possible because Erlang allows two versions of a module to be present in the system: the *current* version and the *old* version. There is always a current version of a loaded module, but an old version of a module only exists if the module has been replaced in run-time by loading a new version. When a new version is loaded, the previously current version becomes the old version, and the new version becomes the current version. However, if there are both a current and old version of a module, a new version cannot be loaded, unless the old version is first explicitly purged.

A global function call is a call where a qualified module name is used, i.e. the call is of the form `M:F(A)` (or `apply(M, F, A)`). A global call causes `M:F` to be dynamically linked into the run-time code, which means that `M:F(A)` will be evaluated using the latest available version of the module, i.e. the current version.

A local function call is a call without a qualified module name, i.e. the call is of the form `F(A)`. The reference to `F` is resolved at compile time (irrespective of whether `F` is exported or not). By the very nature of `F(A)` being a local function call, `F` can only be called by a function that is defined in the very same module as that where `F` is defined. Hence a local function call is always evaluated in the same version of a module as that of the caller.

A *fun* is a function without a name. Like ordinary functions (i.e. functions which have names) its implementation is always bound to some module, and therefore funs are affected by code change as well. A reference to a fun is always indirect, as is the case for a global function call, where the reference is `M:F` (through an export table entry for the module), but the reference is not necessarily global. In fact, if a fun is called in the same module where it is defined, its reference will be resolved in the same way as a local function call is resolved. If a fun is called from a different module, its reference will be resolved as if the call was a global call, but with the additional requirement that the reference also match the particular implementation of the module where the fun was defined.

For each process there is a current function, i.e. the function that the process is currently evaluating. That function resides in some module. Hence a process has always a reference to at least one module. It may of course have references to other modules as well, because of nested, not yet finished calls.

Before a new version of a module can be loaded, the current version must be made old. If there is no old version, the new version is merely loaded, making the previously current version to the old version, and the new version becomes current. All processes that execute the version, which became old, will continue to do so, until they have no unfinished calls within the old version.

If there is an old version, it must first be purged to make room for the current version to become old. However, an old version should not be purged if there are processes that have references to it. Such processes must either be terminated, or the loading of the new version must be postponed until they have terminated by themselves or no longer have references to the old version. There are options for controlling this in release upgrade scripts.

To prevent processes from making calls to other processes during the release installation, they may be *suspended*. All processes implemented with the standard behaviors, or with `sys`, can be suspended. When suspended a process enters a special suspend loop instead of its usual main process loop. In the suspend loop, the process can only receive system messages and shut-down messages from its supervisor. The code change message is a special system message, and this message causes the process to change code to the new version, and possibly to transform its internal state. After the code change a process is *resumed*, i.e. it returns to its main loop.

We highlight here three different types of modules.

Functional module. A module, which does not contain a process loop, i.e. no process has constant references to this kind of module. `lists` is an example of a functional module.

Process module. A module, which contains a process loop, i.e. some process has constant reference to the module. `init` is an example of a process module.

Call-back module. A special case of a functional module which serves as a call-back module for a generic behavior such as `gen_server`. `file` is an example of a call-back module. A call to a call-back module is always a global call (i.e. it refers to the latest version of the module). This has some impacts upon how updates must be handled.

Modules of the above types are handled differently when changing code.

Functional Module If the API of a new version of a functional module is backward compatible, as may be the case of a bug fix or new functionality, we simply load the new version. After a short while, when no processes have references to the old version, the old module is purged.

A more complicated situation arises if the API of a functional module is changed so it is not longer backwards compatible. We must then make sure that no processes, directly or indirectly, try to call functions that have changed. We do this by writing new versions of all modules that use the API. Then, when performing the code change, all potential caller processes are suspended, new versions of the modules that uses the API are loaded, the new version of the functional module is loaded, and finally all suspended processes are resumed.

There are two alternatives available to manage this type of change:

1. Find all calls to the module, change them, and write dependencies in your release upgrade script. This may be manageable, if a function that has been incompatibly changed is called from only a few other functions.
2. Avoid this type of change. This is the only reasonable solution, if an incompatible function is called from many other modules. Instead a completely new function should be introduced, and the original function should be kept for backward compatibility. In the next release, when all other modules are changed as well, the original function can be deleted.

Process Module A process module should never contain global calls to itself (except for code that makes explicit code change). Therefore, a new version of a process module is merely loaded and all processes which are executing the module are told to change their code and, if required, to transform their internal state.

In practice, few modules are pure in the sense that they never contain global calls to themselves. If you use higher-order functions such as `lists:map/2` in a process module, there will be global calls to the module. Therefore, we cannot merely load the module because a process might, still running the old version of the module, make a call to the new version, which might be incompatible.

The only safe way to change code for a process module, is to have its implementation to understand system messages, and to change code by first suspending all processes that run the module, then order them to change code, and finally resume them.

Call-back Module As long as the type of the internal state of a call-back module has not changed, we can just simply load the new version of the module without suspending and resuming the processes involved in the code change. This case is similar to the case of a functional module.

If the type of the internal state has changed, we must first suspend the processes, tell them to change code and at the same time give them the possibility to transform their states, and finally resume them. This is similar to the case of a process module.

Dependencies Between Processes

It is possible that a group of processes, which communicate, must perform code changes while they are suspended. Some of the processes may otherwise use the old protocol while others use the new protocol. On the other hand, there may be time-out dependencies which restrict the number of processes that can perform a synchronized code change as one set. The more processes that are included in the set, the longer the processes are suspended.

There may also be problems with circular dependencies. The following scenario illustrates this situation.

- two modules `a` and `b` are dependent on each other,
- each module is executed by one process with the same name as the corresponding module,
- both are updated at the same time because the internal protocol between them has changed.

The following sequence of events may occur:

1. `a` is suspended.
2. the release handler tries to suspend `b`, but some microsecond before this happens, `b` tries to communicate with `a` which is now suspended
3. If `b` hangs in its call to `a`, the suspension of `b` fails and only `a` is updated.
4. If `b` notices that `a` does not answer and is able to deal with it, then `b` receives the suspend message and is suspended. Then both modules are updated and the processes are resumed.
5. When `a` resumes, there is a message waiting from `b`. This message may be of an old format which `a` does not recognize.

Situations of the type described, and many others, are highly application dependent. The author of the release upgrade script has to predict and avoid them. If the consequences are too difficult to manage, it may be better to entirely shut down and restart all affected processes. This reduces the problem of introducing new code and removes the need to do a synchronized change.

Finding Processes

For each application the `.appup` file specifies how the application is upgraded. The file contains specifications of which modules to change, and how to change them. The `relup` file is an assembly of all the `.appup` files.

For each application the release handler searches for all processes that have to perform a code change. It traverses the application supervision tree to find all child specifications of every supervisor in the tree. Each child specification lists all modules of the application that the child uses.

Hence it is by combining the list of modules to change with all children of supervisors that the release handler finds all processes that are subject to code change.

Port Programs

A port program runs as an external program in the operating system. The simplest way to do code change for a port program is to terminate it, and then start a new version of it.

If that is not adequate, code change may be performed by sending the port program a message telling it to return any data that must survive the termination. Then the program is terminated, and the new version is started and the survived data is to the new version of the port program.

Changing code for port programs is very application dependent. There is no special support for it in SASL.

Application Specification and Configuration Parameters

In each release, each application specification (i.e. the contents of the `.app` file of the application) is known to the release handler. Before any code change is performed for an application, the new environment variables are are made available for the application, i.e. those parameters specified by the `env` tag in the application specification. When the new version of an application is running it will be informed of any changed, new or removed environment variables (see application(Module) in the KERNEL Reference Manual). This means that old processes may read new variables before they are informed of the new release. We advise against the immediate removal of the old variables. Neither do we recommend that they be syntactically changed, although they may of course change their values. They can be safely removed in the next release, by which time it is known that no processes will read the old variables.

Mnesia Data or Schema Changes

Changing data or schemas in Mnesia is similar to changing code for functional modules. Many processes may read or write in the same table at the same time. If we change a table definition, we must make sure that all code which uses the table is changed at the same time.

One way of doing it is to let one process be responsible for one or several tables. This process creates the tables and changes the table definitions or table data. In this way a set of tables is connected with a module (process module or call-back module). When the process performs a code change, the tables are changed as well.

Upgrade vs. Downgrade

When a new release is installed, the system is *upgraded* to the new release. The release handler reads the `relup` file of the new release, and finds the upgrade script that corresponds to an upgrade from the current version to the new version of the system.

When an old release is reinstalled, the release handler reads the `relup` in the current release, and finds the *downgrade* script that corresponds to an downgrade from the current version to the old version of the system.

Usually a `relup` file for a new release contains one upgrade script and one downgrade script for each old version. If a soft downgrade is not wanted (an alternative is to reboot the system from the old release) the downgrade script is left out.

For each modified module in the new release, there are some instructions that specifies how to install that module in a system. When performing an *upgrade*, the following steps are typically involved:

1. Suspend the processes running the module.
2. Load the new code.
3. Tell the processes to switch to new code.
4. Tell the processes to change the internal state. This usually involves calling, *in the new module*, a `code_change` function that is responsible for state updates, e.g. transforming the state from the old format to the new.
5. Resume the processes.

The code change step is always performed when new code has been loaded and all processes are running the new code. The reason for this is that it is always the new version of the module that knows how to change the state from the old version.

When performing a *downgrade* the situation is different. The old module does not know how to transform the new state to the old version: the new format is unknown to the old code. Therefore, it is the responsibility of new code to revert the state back to the old version during downgrade. The following steps are involved:

1. Suspend the processes running the module.
2. Tell the processes to change the internal state. This usually involves calling, *in the current module*, a `code_change` function that is responsible for state reversals, i.e. transforming the state from the current format to the old.
3. Load the new code.
4. Tell the processes to switch code.
5. Resume the processes.

We note that for a process module, it is possible to load the code before a process change its internal state (since a process module never contains global calls to itself), thus making the steps needed for downgrade almost the same as for upgrade. The difference between the two cases is still in the order of switching code and changing state.

For a call-back module it is not actually necessary to tell the processes to switch code, since all calls to the call-back module are global calls. The difference between upgrade and downgrade is still in the order of loading code and performing state change.

The difference between how process modules and a call-back modules are handled in the downgrade case comes from the fact that a process module never contains global calls to itself. The code is thus *static* in the sense that a process executing a process module does not spontaneously switch to new loaded code. The opposite situation is a *dynamic* module, where a process executing the module spontaneously switches to the new code when it is loaded. A call-back module is always dynamic, and a process module static. A functional module is always dynamic.

1.11.6 Release Handling Instructions

This section describes the release upgrade and downgrade scripts. A script is a list of instructions which are interpreted by the release handler when an upgrade or downgrade is made.

There are two levels of instructions; the high-level instructions and the low-level instructions. High- and low-level instructions may be mixed in one script. However, the high-level instructions are translated to low-level instructions by the `systools:make_relup/3` command, because the release handler understands only low-level instructions.

Scripts have to be placed in the `.appup` file for each application. `systools:make_relup/3` assembles the scripts in all `.appup` files to form a `relup` file containing low-level instructions.

High-level Instructions

The high-level instructions are:

- {update, Module, Change, PrePurge, PostPurge, [Mod]} | {update, Module, Timeout, Change, PrePurge, PostPurge,[Mod]} | {update, Module, ModType, Timeout, Change, PrePurge, PostPurge,[Mod]}
  - Module = atom()
  - Timeout = default | infinity | int() > 0
  - ModType = static | dynamic
  - Change = soft | {advanced, Extra}
  - PrePurge = soft_purge | brutal_purge
  - PostPurge = soft_purge | brutal_purge
  - Mod = atom(). If the module is dependent on changes in other modules, these other modules are listed here.

  The instruction is used to update a process module or a call-back module. All processes that run the code of Module are suspended, and if the change is advanced they have to transform their states into the new states. Then the processes are resumed. If Module is dependent on other modules, the release handler will suspend processes in Module before suspending processes in the [Mod] modules. In case of circular dependencies, it will suspend processes in the order that update instructions appear in the script.
  soft means backwards compatible changes and advanced means internal data changes, or changes which are not backwards compatible. Extra is any term, which is used in the argument list of the code_change function in Module (call-back module); otherwise it becomes part of a code change message (process module).
  The optional parameter Timeout defines the time-out for the call to sys:suspend. It specifies how long to wait for a process to handle a suspend message and to get suspended. If no value is specified (or default is given), the default value defined in sys is used.
  The optional parameter ModType specifies if the code is static or dynamic, as defined in Upgrade vs. Downgrade [page 53] above. It needs to be specified only in the case of soft downgrades. Its value defaults to dynamic. Note; if this parameter is specified, Timeout is needed as well.
  PrePurge controls what action to take with processes that are executing an old version of this module. These are processes, which are left since an earlier release upgrade (or downgrade). Usually there are no such processes. If the value is soft_purge and such processes are found, the release will not be installed and the install_release/1 function returns {error, {old_processes, Module}}. If the value is brutal_purge, the processes which run old code are killed.
  PostPurge controls what action to take with processes that are executing old code when the new module has been installed. If the value is soft_purge, the release handler will purge the old code when no remaining processes execute the code. If the value is brutal_purge, the code is purged when the release is made permanent. All processes, which still are running old code are killed.
  The update instruction can also be used for functional modules. However, no processes will be suspended because no processes will have the functional module as its main module. Therefore, no processes perform code change.
- {load_module, Module, PrePurge, PostPurge, [Mod]}
  - Module = atom().
  - PrePurge = soft_purge | brutal_purge
  - PostPurge = soft_purge | brutal_purge

  - `Mod = atom()`. If the module is dependent on changes in other modules, these other modules are listed here.

  The instruction is used to update a functional module or a call-back module. It only loads the module. A call-back module which must perform a code change, or synchronize by being suspended, should use `update` instead.
  The object code is fetched in the beginning of the release upgrade, but the module is loaded when this instruction occurs.

- {`add_module, Mod`} The instruction adds a new module to the system. It loads the module.
- {`remove_application, Appl`} Removes an application. It calls `application:stop` and `application:unload` for the application.
- {`add_application, Appl`} Adds a new application. It calls `application:load` and `application:start` for the application.
- {`restart_application, Appl`} Restarts an existing application. The current version of the application is stopped and removed, and the new version of the application is loaded and started. The instruction is useful when the simplest way to change code for an application is to stop and restart the whole application.

Low-level instructions

The low-level instructions are:

- {`load_object_code, {Lib, LibVsn, [Module]}`} Reads each `Module` from the library `Lib-LibVsn` as a binary. It does not install the code, it just reads the files. The instruction should be placed first in the script in order to read all new code from file. This makes the suspend-load-resume cycle less time consuming. After this instruction has been executed, the code server is updated with the new version of `Lib`. Calls to `code:priv_dir(Lib)` which are made after this instruction return the new `priv` dir.
  `Lib` is typically the application name.
- `point_of_no_return` If a crash occurs after this instruction, the system cannot recover and is restarted from the old version. The instruction must only occur once in a script. It should be placed after all `load_object_code` operations and after user defined checks, which are performed with `apply`. The function `check_install_release/1` tries to evaluate all instructions before this command occurs in the script. Therefore, user defined checks must not have side effects, as they may be evaluated many times.
- {`load, {Module, PrePurge, PostPurge}`} Before this instruction occurs, the `Module` object code must have been loaded with with the `load_object_code` instruction. This instruction makes code out of the binary. `PrePurge = soft_purge | brutal_purge`, and `PostPurge = soft_purge | brutal_purge`.
- {`remove, {Module, PrePurge, PostPurge}`} Makes the current version of `Module` old. When it has been executed, there is no current version in the system. `PrePurge = soft_purge | brutal_purge`, and `PostPurge = soft_purge | brutal_purge`.
- {`purge, [Module]`} Kills all processes that run the old versions of the modules in `[Module]` and deletes all old versions.
- {`suspend, [Module | {Module, Timeout}]`} Tries to suspend all processes that execute `Module`. If a process does not respond, it is ignored. This may cause the process to die, either because it crashes when it spontaneously switches to new code, or as a result of a `purge` operation. If no `Timeout` is specified (or if `default` is given), the default time-out defined in the module `sys` is used.

- {code_change, [{Module, Extra}]} | {code_change, Mode, [{Module, Extra}]} This instruction sends a code_change system message using the function change_code in the module sys with the Extra argument to the *suspended* processes that run this code. Mode is either up or down. Default is up. In case of an upgrade, the message is sent to the suspended process, *after* the new code is loaded (the new version must contain functions to convert from the old internal state, to the the new internal state). In case of a downgrade, the message is sent to the suspended process, *before* the new code is loaded (the current version must contain functions to convert from the current internal state, to the the old internal state).
  Module uses the Extra argument internally in its code change function. Refer to the Reference Manual, module sys for further details.
  One of the arguments to the function sys:change_code is OldVsn. In the case of an upgrade it obtains its value from the attribute vsn in the old code, or undefined if no such attribute was defined. In the case of downgrade, it is the tuple {down, Vsn}, where Vsn is the version of the module as defined in the .app file, or undefined otherwise.
- {resume, [Module]} Resumes all previously suspended processes which execute in any of the modules in the list [Module].
- {stop, [Module]} Stops all processes which are in any of the modules in the list [Module]. The instruction is useful when the simplest way to change code for the [Module] is to stop and restart the processes which run the code. If a supervisor is stopped, all its children are stopped as well.
- {start, [Module]} Starts all previously stopped processes which are in any member of [Module]. The processes will regain their positions in the supervision tree.
- {sync_nodes, Id, [Node] | {M, F, A}} If {M, F, A} is specified, apply(M, F, A) is evaluated and must return a list of nodes. The instruction synchronizes the release installation with other nodes. Each node in the list of nodes must evaluate this command, with the same Id. The local node waits for all other nodes to evaluate the instruction before execution continues. In case a node goes down, it is considered to be an unrecoverable error, and the local node is restarted from the old release. There is no time-out for this instruction, which implies that it may hang forever if a user defined apply enters an infinite loop at some node. It is up to the user to ensure that the apply command eventually returns or makes the node to crash.
- {apply, {M, F, A}} Applies the function to the arguments. If the instruction appears before the point_of_no_return instruction, a failure of the application M:F(A) is caught, causing release_handler:install_release/1 to return {error, {'EXIT', Reason}}. If {error, Error} is thrown or returned by M:F, install_release/1 returns {error, Error}.
  If the instruction appears after the point_of_no_return instruction, and if the application M:F(A) fails, the system is restarted.
- restart_new_emulator Shuts down the current emulator and starts a new one. All processes are terminated gracefully. The new release must still be made permanent when the new emulator is up and running. Otherwise, the old emulator is started in case of a emulator restart. This instruction should be used when a new emulator is introduced, or if a complete reboot of the system should be done.

1.11.7 Release Handling Examples

This section includes several examples that show how different types of upgrades are handled. In call-back modules having the gen_server behavior, all call-back functions have been provided for reasons of clarity.

Update of Erlang Code

Several update examples are shown. Unless otherwise stated, it is assumed that all original modules are in the application `foo`, version `"1.1"`, and the updated version is `"1.2"`.

Simple Functional Module This example is about a pure functional module, i.e. a module the functions of which have no side effects. The original version of the module `lists2` has the following contents:

```
-module(lists2).
-vsn(1).

-export([assoc/2]).

assoc(Key, [{Key, Val} | _]) -> {ok, Val};
assoc(Key, [H | T]) -> assoc(Key, T);
assoc(Key, []) -> false.
```

The new version of the module adds a new function:

```
-module(lists2).
-vsn(2).

-export([assoc/2, multi_map/2]).

assoc(Key, [{Key, Val} | _]) -> {ok, Val};
assoc(Key, [H | T]) -> assoc(Key, T);
assoc(Key, []) -> false.

multi_map(Func, [[] | ListOfLists]) -> [];
multi_map(Func, ListOfLists) ->
    [apply(Func, lists:map({erlang, hd}, ListOfLists)) |
     multi_map(Func, lists:map({erlang, tl}, ListOfLists))].
```

The release upgrade instructions are:

```
[{load_module, lists2, soft_purge, soft_purge, []}]
```

Alternatively, the low-level instructions are:

```
[{load_object_code, {foo, "1.2", [lists2]}},
 point_of_no_return,
 {load, {lists2, soft_purge, soft_purge}}]
```

A More Complicated Functional Module Here we have a functional module bar that uses the module lists2 of the previous example. The original version is only dependent on the original version of lists2.

```
-module(bar).
-vsn(1).

-export([simple/1, complicated_sum/1]).

simple(X) ->
    case lists2:assoc(simple, X) of
        {ok, Val} -> Val;
        false -> false
    end.

complicated_sum([X, Y, Z]) -> cs(X, Y, Z).

cs([HX | TX], [HY | TY], [HZ | TZ]) ->
    NewRes = cs(TX, TY, TZ),
    [HX + HY + HZ | NewRes];
cs([], [], []) -> [].
```

The new version of bar uses the new functionality of lists2 in order to simplify the implementation of the useful function complicated_sum/1. It does not change its API in any way.

```
-module(bar).
-vsn(2).

-export([simple/1, complicated_sum/1]).

simple(X) ->
    case lists2:assoc(simple, X) of
        {ok, Val} -> Val;
        false -> false
    end.

complicated_sum(X) ->
    lists2:multi_map(fun(A,B,C) -> A+B+C end, X).
```

The release upgrade instructions, including instructions for lists2, are as follows:

```
[{load_module, lists2, soft_purge, soft_purge, []},
 {load_module, bar, soft_purge, soft_purge, [lists2]}]
```

> Note:
> We must state that bar is dependent on lists2 to make the release handler to load lists2 before it loads bar.

The low-level instructions are:

```
[{load_object_code, {foo, "1.2", [lists2, bar]}},
 point_of_no_return,
 {load, {lists2, soft_purge, soft_purge}}
 {load, {bar, soft_purge, soft_purge}}]
```

Advanced Functional Module Suppose now that we modify the return value of `lists2:assoc/2` from `{ok, Val}` to `{Key, Val}`. In order to do an upgrade, we would have to find all modules that call `lists2:assoc/2` directly or indirectly, and specify that these modules are dependent on `lists2`. In practice this might an unwieldy task, if if many other modules are using the `lists2` module, and the only reasonable way to perform an upgrade which restarts the whole system.

If we insist on doing a soft upgrade, the modification should be made backward compatible by introducing an new function (`assoc2/2`, say) that has the new return value, and not make any changes to the original function at all.

Advanced gen_server This example assumes that we have a `gen_server` process that must be updated because we have introduced a new function, and added a new data field in our internal state. The contents of the original module are as follows:

```
-module(gs1).
-vsn(1).
-behaviour(gen_server).

-export([get_data/0]).
-export([init/1, handle_call/3, handle_cast/2, handle_info/2,
         terminate/2, code_change/3]).

-record(state, {data}).

get_data() ->
    gen_server:call(gs1, get_data).

init([Data]) ->
    {ok, #state{data = Data}}.

handle_call(get_data, _From, State) ->
    {reply, {ok, State#state.data}, State}.

handle_cast(_Request, State) ->
    {noreply, State}.

handle_info(_Info, State) ->
    {noreply, State}.

terminate(_Reason, _State) ->
    ok.

code_change(_OldVsn, State, _Extra) ->
    {ok, State}.
```

The new module must translate the old state into the new state. Recall that a record is just syntactic sugar for a tuple:

```
-module(gs1).
-vsn(2).
-behaviour(gen_server).

-export([get_data/0, get_time/0]).
-export([init/1, handle_call/3, handle_cast/2, handle_info/2,
         terminate/2, code_change/3]).

-record(state, {data, time}).

get_data() ->
    gen_server:call(gs1, get_data).

get_time() ->
    gen_server:call(gs1, get_time).

init([Data]) ->
    {ok, #state{data = Data, time = erlang:time()}}.

handle_call(get_data, _From, State) ->
    {reply, {ok, State#state.data}, State};
handle_call(get_time, _From, State) ->
    {reply, {ok, State#state.time}, State}.

handle_cast(_Request, State) ->
    {noreply, State}.

handle_info(_Info, State) ->
    {noreply, State}.

terminate(_Reason, _State) ->
    ok.

code_change(1, {state, Data}, _Extra) ->
    {ok, #state{data = Data, time = erlang:time()}}.
```

The release upgrade instructions are as follows:

```
[{update, gs1, {advanced, []}, soft_purge, soft_purge, []}]
```

The alternative low-level instructions are:

```
[{load_object_code, {foo, "1.2", [gs1]}},
 point_of_no_return,
 {suspend, [gs1]},
 {load, {gs1, soft_purge, soft_purge}},
 {code_change, [{gs1, []}]},
 {resume, [gs1]}]
```

If we want to handle soft downgrade as well, the code would be as follows:

```
-module(gs1).
-vsn(2).
-behaviour(gen_server).

-export([get_data/0, get_time/0]).
-export([init/1, handle_call/3, handle_cast/2, handle_info/2,
         terminate/2, code_change/3]).

-record(state, {data, time}).

get_data() ->
    gen_server:call(gs1, get_data).
get_time() ->
    gen_server:call(gs1, get_time).

init([Data]) ->
    {ok, #state{data = Data, time = erlang:time()}}.

handle_call(get_data, _From, State) ->
    {reply, {ok, State#state.data}, State};
handle_call(get_time, _From, State) ->
    {reply, {ok, State#state.time}, State}.

handle_cast(_Request, State) ->
    {noreply, State}.

handle_info(_Info, State) ->
    {noreply, State}.

terminate(_Reason, _State) ->
    ok.

code_change(1, {state, Data}, _Extra) ->
    {ok, #state{data = Data, time = erlang:time()}};
code_change({down, 1}, #state{data = Data}, _Extra) ->
    {ok, {state, Data}}.
```

Note that we take care of translating the new state to the old format as well. The low-level instructions are:

```
[{load_object_code, {foo, "1.2", [gs1]}},
 point_of_no_return,
 {suspend, [gs1]},
 {code_change, [{gs1, []}]},
 {load, {gs1, soft_purge, soft_purge}},
 {resume, [gs1]}]
```

Advanced gen_server with Dependencies

This example assumes that we have `gen_server` process that uses the in gs1 as defined in the previous example.

The contents of the original module are as follows:

```
-module(gs2).
-vsn(1).
-behaviour(gen_server).

-export([is_operation_ok/1]).
-export([init/1, handle_call/3, handle_cast/2, handle_info/2,
         terminate/2, code_change/3]).

is_operation_ok(Op) ->
    gen_server:call(gs2, {is_operation_ok, Op}).

init([Data]) ->
    {ok, []}.

handle_call({is_operation_ok, Op}, _From, State) ->
    Data = gs1:get_data(),
    Reply = lists2:assoc(Op, Data),
    {reply, Reply, State}.

handle_cast(_Request, State) ->
    {noreply, State}.

handle_info(_Info, State) ->
    {noreply, State}.

terminate(_Reason, _State) ->
    ok.

code_change(_OldVsn, State, _Extra) ->
    {ok, State}.
```

The new version does not have to transform the internal state, hence the code_change/3 function is not really needed (it will not be called since the upgrade of gs2 is soft).

```
-module(gs2).
-vsn(2).
-behaviour(gen_server).

-export([is_operation_ok/1]).
-export([init/1, handle_call/3, handle_cast/2, handle_info/2,
         terminate/2, code_change/3]).

is_operation_ok(Op) ->
    gen_server:call(gs2, {is_operation_ok, Op}).

init([Data]) ->
    {ok, []}.

handle_call({is_operation_ok, Op}, _From, State) ->
    Data = gs1:get_data(),
    Time = gs1:get_time(),
    Reply = do_things(lists2:assoc(Op, Data), Time),
    {reply, Reply, State}.
```

```
handle_cast(_Request, State) ->
    {noreply, State}.

handle_info(_Info, State) ->
    {noreply, State}.

terminate(_Reason, _State) ->
    ok.

code_change(_OldVsn, State, _Extra) ->
    {ok, State}.

do_things({ok, Val}, Time) ->
    Val;
do_things(false, Time) ->
    {false, Time}.
```

The release upgrade instructions are:

```
[{update, gs1, {advanced, []}, soft_purge, soft_purge, []},
 {update, gs2, soft, soft_purge, soft_purge, [gs1]},
```

The corresponding low-level instructions are:

```
[{load_object_code, {foo, "1.2", [gs1, gs2]}},
 point_of_no_return,
 {suspend, [gs1, gs2]},
 {load, {gs1, soft_purge, soft_purge}},
 {load, {gs2, soft_purge, soft_purge}},
 {code_change, [{gs1, []}]},    % No gs2 here!
 {resume, [gs1, gs2]}]
```

Other Worker Processes

All other worker processes in a supervision tree, such as processes of the types `gen_event`, `gen_fsm`, and processes implemented by using `proc_lib` and `sys`, are handled in exactly the same way as processes of type `gen_server` are handled. Examples follow.

Simple gen_event

This example shows how an event handler may be updated. We do not make any assumptions about which event manager processes the handler is installed in, it is the responsibility of the release handler to find them. The contents of the original module is as follows:

```
-module(ge_h).
-vsn(1).
-behaviour(gen_event).

-export([get_events/1]).
-export([init/1, handle_event/2, handle_call/2, handle_info/2,
         terminate/2, code_change/3]).

get_events(Mgr) ->
    gen_event:call(Mgr, ge_h, get_events).
```

```
init(_) -> {ok, undefined}.

handle_event(Event, _LastEvent) ->
    {ok, Event}.

handle_call(get_events, LastEvent) ->
    {ok, [LastEvent], LastEvent}.

handle_info(Info, LastEvent) ->
    {ok, LastEvent}.

terminate(Arg, LastEvent) ->
    ok.

code_change(_OldVsn, LastEvent, _Extra) ->
    {ok, LastEvent}.
```

The new module decides to keep the two latest events in a list and must translate the old state into the new state.

```
-module(ge_h).
-vsn(2).
-behaviour(gen_event).

-export([get_events/1]).
-export([init/1, handle_event/2, handle_call/2, handle_info/2,
         terminate/2, code_change/3]).

get_events(Mgr) ->
    gen_event:call(Mgr, ge_h, get_events).

init(_) -> {ok, []}.

handle_event(Event, []) ->
    {ok, [Event]};
handle_event(Event, [Event1 | _]) ->
    {ok, [Event, Event1]}.

handle_call(get_events, Events) ->
    Events.

handle_info(Info, Events) ->
    {ok, Events}.

terminate(Arg, Events) ->
    ok.

code_change(1, undefined, _Extra) ->
    {ok, []};
code_change(1, LastEvent, _Extra) ->
    {ok, [LastEvent]}.
```

The release upgrade instructions are:

```
[{update, ge_h, {advanced, []}, soft_purge, soft_purge, []}]
```

The low-level instructions are:

```
[{load_object_code, {foo, "1.2", [ge_h]}},
 point_of_no_return,
 {suspend, [ge_h]},
 {load, {ge_h, soft_purge, soft_purge}},
 {code_change, [{ge_h, []}]},
 {resume, [ge_h]}]
```

> **Note:**
> These instructions are identical to those used for the gen_server.

Process Implemented with sys and proc_lib Processes implemented with sys and proc_lib are changed in the same way as processes that are implemented according to the gen_server behavior (which should not come as surprise, since gen_server et al. are implemented on top of sys and proc_lib). However, the code change function is defined differently. The original is as follows:

```
-module(sp).
-vsn(1).

-export([start/0, get_data/0]).
-export([init/1, system_continue/3, system_terminate/4]).

-record(state, {data}).

start() ->
    Pid = proc_lib:spawn_link(?MODULE, init, [self()]),
    {ok, Pid}.

get_data() ->
    sp_server ! {self(), get_data},
    receive
        {sp_server, Data} -> Data
    end.

init(Parent) ->
    register(sp_server, self()),
    process_flag(trap_exit, true),
    loop(#state{}, Parent).

loop(State, Parent) ->
    receive
        {system, From, Request} ->
            sys:handle_system_msg(Request, From, Parent, ?MODULE, [], State);
```

```
        {'EXIT', Parent, Reason} ->
            cleanup(State),
            exit(Reason);
        {From, get_data} ->
            From ! {sp_server, State#state.data},
            loop(State, Parent);
        _Any ->
            loop(State, Parent)
    end.

cleanup(State) -> ok.

%% Here are the sys call back functions
system_continue(Parent, _, State) ->
    loop(State, Parent).

system_terminate(Reason, Parent, _, State) ->
    cleanup(State),
    exit(Reason).
```

The new code, which takes care of up- and downgrade is as follows:

```
-module(sp).
-vsn(2).

-export([start/0, get_data/0, set_data/1]).
-export([init/1, system_continue/3, system_terminate/4,
        system_code_change/4]).

-record(state, {data, last_pid}).

start() ->
    Pid = proc_lib:spawn_link(?MODULE, init, [self()]),
    {ok, Pid}.

get_data() ->
    sp_server ! {self(), get_data},
    receive
        {sp_server, Data} -> Data
    end.

set_data(Data) ->
    sp_server ! {self(), set_data, Data}.

init(Parent) ->
    register(sp_server, self()),
    process_flag(trap_exit, true),
    loop(#state{last_pid = no_one}, Parent).

loop(State, Parent) ->
    receive
        {system, From, Request} ->
            sys:handle_system_msg(Request, From, Parent,
```

```
                                ?MODULE, [], State);
        {'EXIT', Parent, Reason} ->
            cleanup(State),
            exit(Reason);
        {From, get_data} ->
            From ! {sp_server, State#state.data},
            loop(State, Parent);
        {From, set_data, Data} ->
            loop(State#state{data = Data, last_pid = From}, Parent);
        _Any ->
            loop(State, Parent)
    end.

cleanup(State) -> ok.

%% Here are the sys call back functions
system_continue(Parent, _, State) ->
    loop(State, Parent).

system_terminate(Reason, Parent, _, State) ->
    cleanup(State),
    exit(Reason).

system_code_change({state, Data}, _Mod, 1, _Extra) ->
    {ok, #state{data = Data, last_pid = no_one}};
system_code_change(#state{data = Data}, _Mod, {down, 1}, _Extra) ->
    {ok, {state, Data}}.
```

The release upgrade instructions are:

```
[{update, sp, static, default, {advanced, []}, soft_purge, soft_purge, []}]
```

The low-level instructions are the same for upgrade and downgrade:

```
[{load_object_code, {foo, "1.2", [sp]}},
 point_of_no_return,
 {suspend, [sp]},
 {load, {sp, soft_purge, soft_purge}},
 {code_change, [{sp, []}]},
 {resume, [sp]}]
```

Supervisor This example assumes that a new version of an application adds a new process, and deletes one process from a supervisor. The original code is as follows:

```
-module(sup).
-vsn(1).
-behaviour(supervisor).
-export([init/1]).

init([]) ->
    SupFlags = {one_for_one, 4, 3600},
    Server = {my_server, {my_server, start_link, []},
```

```
            permanent, 2000, worker, [my_server]},
    GS1 = {gs1, {gs1, start_link, []}, permanent, 2000, worker, [gs1]},
    {ok, {SupFlags, [Server, GS1]}}.
```

The new code is as follows:

```
-module(sup).
-vsn(2).
-behaviour(supervisor).
-export([init/1]).

init([]) ->
    SupFlags = {one_for_one, 4, 3600},
    GS1 = {gs1, {gs1, start_link, []}, permanent, 2000, worker, [gs1]},
    GS2 = {gs2, {gs2, start_link, []}, permanent, 2000, worker, [gs2]},
    {ok, {SupFlags, [GS1, GS2]}}.
```

The release upgrade instructions are:

```
[{update, sup, {advanced, []}, soft_purge, soft_purge, []}
 {apply, {supervisor, terminate_child, [sup, my_server]}},
 {apply, {supervisor, delete_child, [sup, my_server]}},
 {apply, {supervisor, restart_child, [sup, gs2]}}]
```

The low-level instructions are:

```
[{load_object_code, {foo, "1.2", [sup]}},
 point_of_no_return,
 {suspend, [sup]},
 {load, {sup, soft_purge, soft_purge}},
 {code_change, [{sup, []}]},
 {resume, [sup]},
 {apply, {supervisor, terminate_child, [sup, my_server]}},
 {apply, {supervisor, delete_child, [sup, my_server]}},
 {apply, {supervisor, restart_child, [sup, gs2]}}]
```

High-level `update` instruction for a supervisor is mapped to a low-level advanced code change instruction. In the `code_change` function of the supervisor, the new child specification is installed, but no children are explicitly terminated or started. Therefore, children must be terminated, deleted and started by using the `apply` instruction.

Complex Dependencies As already mentioned, sometimes the simplest and safest way to introduce a new release is to terminate parts of the system, load the new code, and restart that part. However, individual processes cannot simply be killed, since their supervisors will restart them again. Instead supervisors must first be ordered to stop their children before now code can be loaded. Then supervisors are ordered to restart their children. All this is done by issuing the `stop` and `start` instructions.

The following example assumes that we have a supervisor `a` with two children `b` and `c`, where `b` is a worker and `c` is a supervisor for `d`. We want to restart all processes except for `a`. The upgrade instructions are as follows:

```
[{load_object_code, {foo, "1.2", [b,c,d]}},
 point_of_no_return,
 {stop, [b, c]},
 {load, {b, soft_purge, soft_purge}},
 {load, {c, soft_purge, soft_purge}},
 {load, {d, soft_purge, soft_purge}},
 {start, [b, c]}]
```

Note:
We do not need to explicitly stop `d`, this is done by the supervisor `c`.

A whole application cannot be stopped and started with the `stop` and `start` instructions. The instruction `restart_application` has to be used instead.

New Application The examples shown so far have dealt with changing an existing application. In order to introduce a completely new application we just have to have an `add_application` instruction, but we also have to make sure that the boot file of the new release contains enough in order to start it. The following example shows how to to introduce the application `new_appl`, which has just one module: `new_mod`.

The release upgrade instructions are:

```
[{add_application, new_appl}]
```

The corresponding low-level instructions are as follows (note that the application specification is used as argument to `application:start_application/1`):

```
[{load_object_code, {new_appl, "1.0", [new_mod]}},
 point_of_no_return,
 {load, {new_mod, soft_purge, soft_purge}},
 {apply, {application, start,
           [{application, new_appl,
             [{description, "NEW APPL"},
              {vsn, "1.0"},
              {modules, [new_mod]},
              {registered, []},
              {applications, [kernel, foo]},
              {env, []},
              {mod, {new_mod, start_link, []}}]},
            permanent]}}].
```

Remove an Application An application is removed in the same way as new applications are introduced. This example assumes that we want to remove the `new_appl` application:

```
[{remove_application, new_appl}]
```

The corresponding low_level instructions are:

```
[point_of_no_return,
 {apply, {application, stop, [new_appl]}},
 {remove, {new_mod, soft_purge, soft_purge}}].
```

Update of Port Programs

Each port program is controlled by a Erlang process called the *port controller*. A port program is updated by the port controller process. It is always done by terminating the old port program, and starting the new one.

Port Controller In this example we have a port controller process, where we must take care of the termination and restart of the port program ourselves. Also, we may prepare for the possibility of changing the Erlang code of the port controller only. The `gen_server` behavior is used to implement the port controller. The contents of the original module is as follows.

```
-module(portc).
-vsn(1).
-behaviour(gen_server).

-export([get_data/0]).
-export([init/1, handle_call/3, handle_info/2, code_change/3]).

-record(state, {port, data}).

get_data() -> gen_server:call(portc, get_data).

init([]) ->
    PortProg = code:priv_dir(foo) ++ "/bin/portc",
    Port = open_port({spawn, PortProg}, [binary, {packet, 2}]),
    {ok, #state{port = Port}}.

handle_call(get_data, _From, State) ->
    {reply, {ok, State#state.data}, State}.

handle_info({Port, Cmd}, State) ->
    NewState = do_cmd(Cmd, State),
    {noreply, NewState}.

code_change(_, State, change_port_only) ->
    State#state.port ! close,
    receive
        {Port, closed} -> true
    end,
    NPortProg = code:priv_dir(foo) ++ "/bin/portc",   % get new version
    NPort = open_port({spawn, NPortProg}, [binary, {packet, 2}]),
    {ok, State#state{port = NPort}}.
```

To change the port program without changing the Erlang code, we can use the following code:

```
[point_of_no_return,
 {suspend, [portc]},
 {code_change, [{portc, change_port_only}]},
 {resume, [portc]}]
```

Here we used low-level instructions only. In this example we also make use of the `Extra` argument of the `code_change/3` function.

Suppose now that we wish to change only the Erlang code. The new version of `portc` is as follows:

```
-module(portc).
-vsn(2).
-behaviour(gen_server).

-export([get_data/0]).
-export([init/1, handle_call/3, handle_info/2, code_change/3]).

-record(state, {port, data}).

get_data() -> gen_server:call(portc, get_data).

init([]) ->
    PortProg = code:priv_dir(foo) ++ "/bin/portc",
    Port = open_port({spawn, PortProg}, [binary, {packet, 2}]),
    {ok, #state{port = Port}}.

handle_call(get_data, _From, State) ->
    {reply, {ok, State#state.data}, State}.

handle_info({Port, Cmd}, State) ->
    NewState = do_cmd(Cmd, State),
    {noreply, NewState}.

code_change(_, State, change_port_only) ->
    State#state.port ! close,
    receive
        {Port, closed} -> true
    end,
    NPortProg = code:priv_dir(foo) ++ "/bin/portc",   % get new version
    NPort = open_port({spawn, NPortProg}, [binary, {packet, 2}]),
    {ok, State#state{port = NPort}};
code_change(1, State, change_erl_only) ->
    NState = transform_state(State),
    {ok, NState}.
```

The high-level instruction is:

```
[{update, portc, {advanced, change_erl_only}, soft_purge, soft_purge, []}]
```

The corresponding low-level instructions are:

```
[{load_object_code, {portc, 2, [portc]}},
 point_of_no_return,
 {suspend, [portc]},
 {load, {portc, soft_purge, soft_purge}},
 {code_change, [{portc, change_erl_only}]},
 {resume, [portc]}]
```

List of Figures