OTP Design Principles

version 5.5

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.

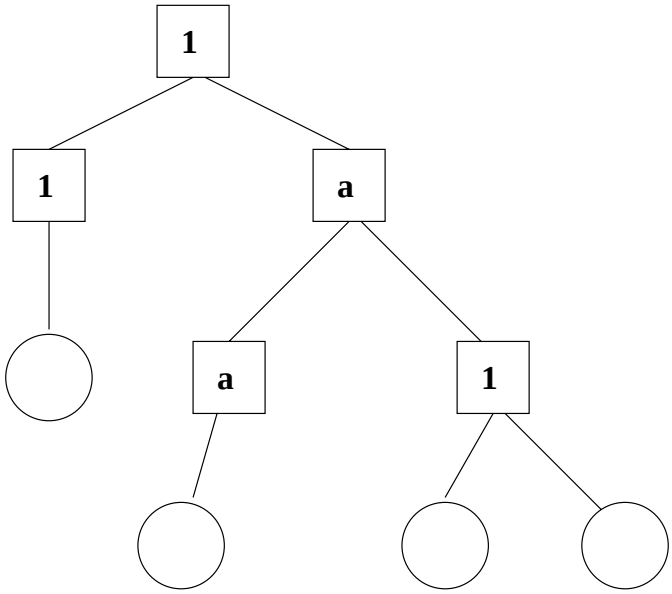

Figure 1.1: Supervision Tree

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.

Code written without making use of behaviours may be more efficient, but the increased efficiency will be at the expense of generality. The ability to manage all applications in the system in a consistent manner is very important.

Using behaviours also makes it easier to read and understand code written by other programmers. Ad hoc programming structures, while possibly more efficient, are always more difficult to understand.

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

The standard Erlang/OTP behaviours are:

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

gen_fsm [page 10] For implementing finite state machines.

gen_event [page 15] For implementing event handling functionality.

supervisor [page 18] For implementing a supervisor in a supervision tree.

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

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

3> c(chs3).
./chs3.erl:10: Warning: undefined call-back function handle_call/3
{ok,chs3}
```

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 31].

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 45].

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 versions of a release, in a (possibly) running system. How to do this is described in Release Handling [page 49].

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.

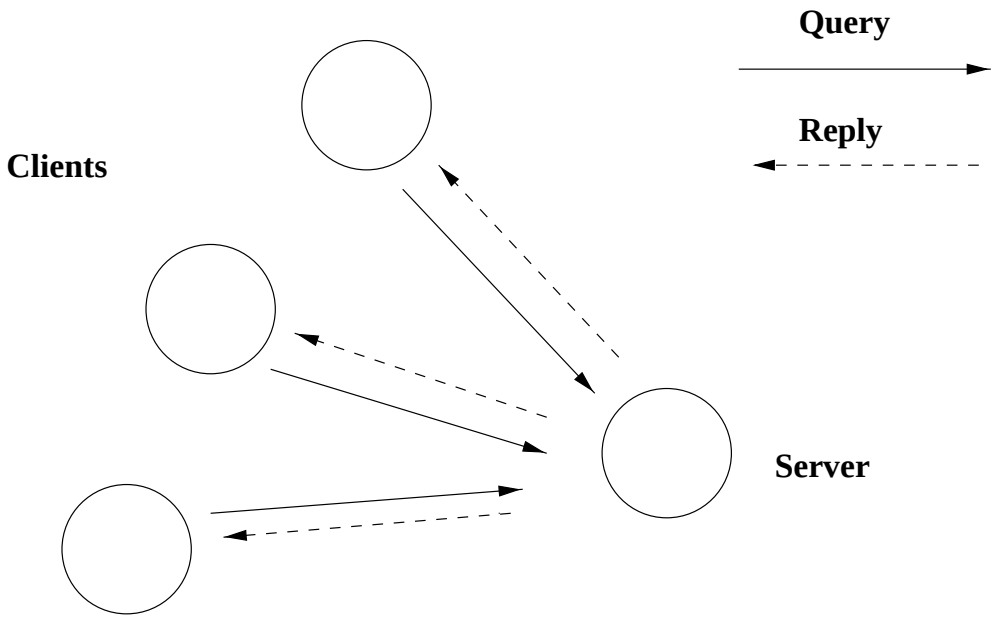

Figure 1.2: Client-Server Model

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.

gen_server:start_link must be used if the gen_server is part of a supervision tree, i.e. is started by a supervisor. There is another function gen_server:start to start a stand-alone gen_server, i.e. a gen_server which is not part of a supervision tree.

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 set of remaining available channels Chs2.

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

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 gen_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

In a Supervision Tree

If the gen_server is part of a supervision tree, no stop function is needed. The gen_server will automatically be terminated by its supervisor. Exactly how this is done is defined by a shutdown strategy [page 21] set in the supervisor.

If it is necessary to clean up before termination, the shutdown strategy must be a timeout value and the gen_server must be set to trap exit signals in the `init` function. When ordered to shutdown, the gen_server will then call the callback function `terminate(shutdown, State)`:

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

...

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

Stand-Alone Gen_Servers

If the gen_server is not part of a supervision tree, a stop function may be useful, for example:

```
...
export([stop/0]).
...

stop() ->
    gen_server:cast(ch3, stop).
...

handle_cast(stop, State) ->
    {stop, normal, State};
handle_cast({free, Ch}, State) ->
    ....

...

terminate(normal, State) ->
    ok.
```

The callback function handling the `stop` request returns a tuple `{stop, normal, State1}`, where `normal` specifies that it is a normal termination and `State1` is a new value for the state of the gen_server. This will cause the gen_server to call `terminate(normal,State1)` and then terminate gracefully.

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 of other messages are exit messages, if the gen_server is linked to other processes (than the supervisor) and trapping exit signals.

```
handle_info({'EXIT', Pid, Reason}, State) ->
    ..code to handle exits here..
    {noreply, State1}.
```

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 locked. Anytime someone presses a button, this generates an event. Depending on what buttons have been pressed before, the sequence so far may be correct, incomplete or wrong.

If it is correct, the door is unlocked for 30 seconds (30000 ms). If it is incomplete, we wait for another button to be pressed. If it is is wrong, we start all over, waiting for a new button sequence.

Implementing the code lock FSM using `gen_fsm` results in this callback module:

```
-module(code_lock).
-behaviour(gen_fsm).

-export([start_link/1]).
-export([button/1]).
-export([init/1, locked/2, open/2]).

start_link(Code) ->
    gen_fsm:start_link({local, code_lock}, code_lock, Code, []).

button(Digit) ->
    gen_fsm:send_event(code_lock, {button, Digit}).

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

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

open(timeout, State) ->
    do_lock(),
    {next_state, locked, State}.
```

The code is explained in the next sections.

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`, `locked` and `open`). This is normally good programming practice, to have the code corresponding to one process contained in one module.
- The third argument, `Code`, is a term which is passed as-is to the callback function `init`. Here, `init` gets the correct code for the lock as indata.
- The fourth argument, `[]`, is a list of options. See `gen_fsm(3)` for available options.

If name registration succeeds, the new gen_fsm process calls the callback function `code_lock:init(Code)`. This function is expected to return `{ok, StateName, StateData}`, where `StateName` is the name of the initial state of the gen_fsm. In this case `locked`, assuming the door is locked to begin with. `StateData` is the internal state of the gen_fsm. (For gen_fsms, the internal state is often referred to 'state data' to distinguish it from the state as in states of a state machine.) In this case, the state data is the button sequence so far (empty to begin with) and the correct code of the lock.

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

Note that `gen_fsm:start_link` is synchronous. It does not return until the gen_fsm has been initialized and is ready to receive notifications.

`gen_fsm:start_link` must be used if the gen_fsm is part of a supervision tree, i.e. is started by a supervisor. There is another function `gen_fsm:start` to start a stand-alone gen_fsm, i.e. a gen_fsm which is not part of a supervision tree.

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.

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

```
open(timeout, State) ->
    do_lock(),
    {next_state, locked, State}.
```

If the door is locked and a button is pressed, the complete button sequence so far is compared with the correct code for the lock and, depending on the result, the door is either unlocked and the gen_fsm goes to state open, or the door remains in state locked.

1.3.5 Timeouts

When a correct code has been givened, the door is unlocked and the following tuple is returned from locked/2:

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

30000 is a timeout value in milliseconds. After 30000 ms, i.e. 30 seconds, a timeout occurs. Then StateName(timeout, StateData) is called. In this case, the timeout occurs when the door has been in state open for 30 seconds. After that the door is locked again:

```
open(timeout, State) ->
    do_lock(),
    {next_state, locked, State}.
```

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

In a Supervision Tree

If the gen_fsm is part of a supervision tree, no stop function is needed. The gen_fsm will automatically be terminated by its supervisor. Exactly how this is done is defined by a shutdown strategy [page 21] set in the supervisor.

If it is necessary to clean up before termination, the shutdown strategy must be a timeout value and the gen_fsm must be set to trap exit signals in the `init` function. When ordered to shutdown, the gen_fsm will then call the callback function `terminate(shutdown, StateName, StateData)`:

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

...

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

Stand-Alone Gen_Fsms

If the gen_fsm is not part of a supervision tree, a stop function may be useful, for example:

```
...
-export([stop/0]).
...

stop() ->
    gen_fsm:send_all_state_event(code_lock, stop).
...

handle_event(stop, _StateName, StateData) ->
    {stop, normal, StateData}.

...

terminate(normal, _StateName, _StateData) ->
    ok.
```

The callback function handling the `stop` event returns a tuple {`stop`,`normal`,`StateData1`}, where `normal` specifies that it is a normal termination and `StateData1` is a new value for the state data of the gen_fsm. This will cause the gen_fsm to call `terminate(normal,StateName,StateData1)` and then terminate gracefully:

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 of other messages are exit messages, if the gen_fsm is linked to other processes (than the supervisor) and trapping exit signals.

```
handle_info({'EXIT', Pid, Reason}, StateName, StateData) ->
    ..code to handle exits here..
    {next_state, StateName1, StateData1}.
```

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 manager is notified about an event, the event will be processed by all the installed event handlers. For example, an event manager for handling errors can by default have a handler installed which writes error messages to the terminal. If the error messages during a certain period should be saved to a file as well, the user adds another event handler which does this. When logging to file is no longer necessary, this event handler is deleted.

An event manager is implemented as a process and each event handler is implemented as a callback module.

The event manager essentially maintains a list of {`Module, State`} pairs, where each `Module` is an event handler, and `State` the internal state of that event handler.

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

To start an event manager for handling errors, as described in the example above, call the following function:

```
gen_event:start_link({local, error_man})
```

This function spawns and links to a new process, an event manager.

The argument, {local, error_man} specifies the name. In this case, the event manager will be locally registered as error_man.

If the name is omitted, the event manager is not registered. Instead its pid must be used. The name could also be given as {global, Name}, in which case the event manager is registered using global:register_name/2.

gen_event:start_link must be used if the event manager is part of a supervision tree, i.e. is started by a supervisor. There is another function gen_event:start to start a stand-alone event manager, i.e. an event manager which is not part of a supervision tree.

1.4.4 Adding an Event Handler

Here is an example using the shell on how to start an event manager and add an event handler to it:

```
1> gen_event:start({local, error_man}).
{ok,<0.31.0>}
2> gen_event:add_handler(error_man, terminal_logger, []).
ok
```

This function sends a message to the event manager registered as error_man, telling it to add the event handler terminal_logger. The event manager will call the callback function terminal_logger:init([]), where the argument [] is the third argument to add_handler. init is expected to return {ok, State}, where State is the internal state 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

When an event manager is stopped, it will give each of the installed event handlers the chance to clean up by calling terminate/2, the same way as when deleting a handler.

In a Supervision Tree

If the event manager is part of a supervision tree, no stop function is needed. The event manager will automatically be terminated by its supervisor. Exactly how this is done is defined by a shutdown strategy [page 21] set in the supervisor.

Stand-Alone Event Managers

An event manager can also be stopped by calling:

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

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 21]. 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 19].

1 and 60 defines the maximum restart frequency [page 20].

The tuple {ch3, ...} is a child specification [page 21].

1.5.3 Restart Strategy

one_for_one

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

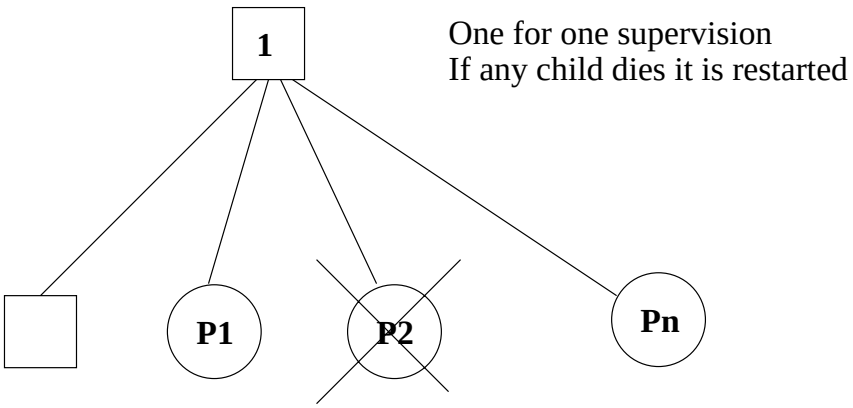

Figure 1.3: One_For_One Supervision

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.

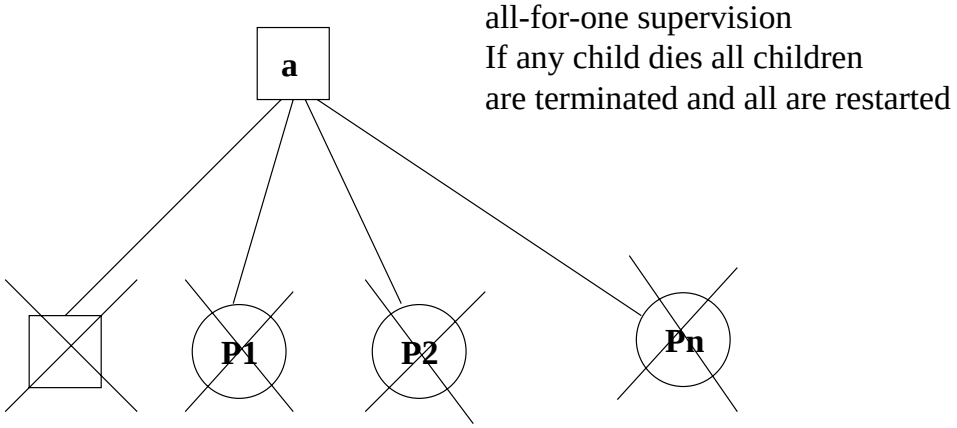

Figure 1.4: One_For_All Supervision

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 49].

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 16]:

```
{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 21].

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 21].

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 standard behaviour. They can also be used to implement user defined (non-standard) behaviours.

Both sys and proc_lib belong to the STDLIB application.

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 standard behaviour. Such a process should:

- be started in a way that makes the process fit into a supervision tree,
- support the sys debug facilities [page 28], and
- take care of system messages [page 29].

System messages are messages with special meaning, used in the supervision tree. Typical system messages are requests for trace output, and requests to suspend or resume process execution (used during release handling). Processes implemented using standard behaviours automatically understand these messages.

Example

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

Also, if the process terminates with another reason than normal or shutdown, a crash report (see SASL User's Guide) is generated.

In the example, synchronous start is used. The process is started by calling ch4:start_link():

```
start_link() ->
    proc_lib:start_link(ch4, init, [self()]).
```

ch4:start_link calls the function proc_lib:start_link. This function takes a module name, a function name and an argument list as arguments and spawns and links to a new process. The new process starts by executing the given function, in this case ch4:init(Pid), where Pid is the pid (self()) of the first process, that is the parent process.

In init, all initialization including name registration is done. The new process must also acknowledge that it has been started to the parent:

```
init(Parent) ->
    ...
    proc_lib:init_ack(Parent, {ok, self()}),
    loop(...).
```

proc_lib:start_link is synchronous and does not return until proc_lib:init_ack has been called.

Debugging

To support the debug facilites in sys, we need a *debug structure*, a term Deb which is initialized using sys:debug_options/1:

```
init(Parent) ->
    ...
    Deb = sys:debug_options([]),
    ...
    loop(Chs, Parent, Deb).
```

sys:debug_options/1 takes a list of options as argument. Here the list is empty, which means no debugging is enabled initially. See sys(3) for information about possible options.

Then for each *system event* that we want to be logged or traced, the following function should be called.

```
sys:handle_debug(Deb, Func, Info, Event) => Deb1
```

- Deb is the debug structure.
- Func is a tuple {Module, Name} (or a fun) and should specify a (user defined) function used to format trace output. For each system event, the format function is called as Module:Name(Dev, Event, Info), where:

  - Dev is the IO device to which the output should be printed. See io(3).
  - Event and Info are passed as-is from handle_debug.

- Info is used to pass additional information to Func, it can be any term and is passed as-is.
- Event is the system event. It is up to the user to define what a system event is and how it should be represented, but typically at least incoming and outgoing messages are considered system events and represented by the tuples {in,Msg[,From]} and {out,Msg,To}, respectively.

handle_debug returns an updated debug structure Deb1.

In the example, handle_debug is called for each incoming and outgoing message. The format function Func is the function ch4:write_debug/3 which prints the message using io:format/3.

```
loop(Chs, Parent, Deb) ->
    receive
        {From, alloc} ->
            Deb2 = sys:handle_debug(Deb, {ch4, write_debug},
                                    ch4, {in, alloc, From}),
            {Ch, Chs2} = alloc(Chs),
            From ! {ch4, Ch},
            Deb3 = sys:handle_debug(Deb2, {ch4, write_debug},
                                    ch4, {out, {ch4, Ch}, From}),
            loop(Chs2, Parent, Deb3);
        {free, Ch} ->
            Deb2 = sys:handle_debug(Deb, {ch4, write_debug},
                                    ch4, {in, {free, Ch}}),
            Chs2 = free(Ch, Chs),
            loop(Chs2, Parent, Deb2);
        ...
    end.

write_debug(Dev, Event, Name) ->
    io:format(Dev, "~p event = ~p~n", [Name, Event]).
```

Handling System Messages

System messages are received as:

```
{system, From, Request}
```

The content and meaning of these messages do not need to be interpreted by the process. Instead the following function should be called:

```
sys:handle_system_msg(Request, From, Parent, Module, Deb, State)
```

This function does not return. It will handle the system message and then call:

```
Module:system_continue(Parent, Deb, State)
```

if process execution should continue, or:

```
Module:system_terminate(Reason, Parent, Deb, State)
```

if the process should terminate. Note that a process in a supervision tree is expected to terminate with the same reason as its parent.

- `Request` and `From` should be passed as-is from the system message to the call to `handle_system_msg`.
- `Parent` is the pid of the parent.
- `Module` is the name of the module.
- `Deb` is the debug structure.
- `State` is a term describing the internal state and is passed to `system_continue`/`system_terminate`.

In the example:

```
loop(Chs, Parent, Deb) ->
    receive
        ...

        {system, From, Request} ->
            sys:handle_system_msg(Request, From, Parent,
                                  ch4, Deb, Chs)
    end.

system_continue(Parent, Deb, Chs) ->
    loop(Chs, Parent, Deb).

system_terminate(Reason, Parent, Deb, Chs) ->
    exit(Reason).
```

If the special process is set to trap exits, note that if the parent process terminates, the expected behavior is to terminate with the same reason:

```
init(...) ->
    ...,
    process_flag(trap_exit, true),
    ...,
    loop(...).

loop(...) ->
    receive
        ...

        {'EXIT', Parent, Reason} ->
            ..maybe some cleaning up here..
            exit(Reason);
        ...
    end.
```

1.6.3 User-Defined Behaviours

To implement a user-defined behaviour, write code similar to code for a special process but calling functions in a callback module for handling specific tasks.

If it is desired that the compiler should warn for missing callback functions, as it does for the OTP behaviours, implement and export the function:

```
behaviour_info(callbacks) ->
    [{Name1,Arity1},...,{NameN,ArityN}].
```

where each `{Name,Arity}` specifies the name and arity of a callback function.

When the compiler encounters the module attribute `-behaviour(Behaviour).` in a module `Mod`, it will call `Behaviour:behaviour_info(callbacks)` and compare the result with the set of functions actually exported from `Mod`, and issue a warning if any callback function is missing.

Example:

```
%% User-defined behaviour module
-module(simple_server).
-export([start_link/2,...]).
-export([behaviour_info/1]).

behaviour_info(callbacks) ->
    [{init,1},
     {handle_req,1},
     {terminate,0}].

start_link(Name, Module) ->
    proc_lib:start_link(?MODULE, init, [self(), Name, Module]).

init(Parent, Name, Module) ->
    register(Name, self()),
    ...,
    Dbg = sys:debug_options([]),
    proc_lib:init_ack(Parent, {ok, self()}),
    loop(Parent, Module, Deb, ...).

...
```

In a callback module:

```
-module(db).
-behaviour(simple_server).

-export([init/0, handle_req/1, terminate/0]).

...
```

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 32], 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 33]. 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 45]), the code for each application is placed in a separate directory following a pre-defined directory structure [page 34].

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 40]. `StartArgs` is defined by the key `mod` in the application resource file [page 33] 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 35].

Example of an application callback module for packaging the supervision tree from the Supervisor [page 19] 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 45]), 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.

`ebin` Contains the Erlang object code, the `beam` files. The `.app` file is also placed here.

`priv` Used for application specific files. For example, C executables are placed here. The function `code:priv_dir/1` should be used to access this directory.

`include` Used for include files.

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 using `application:load/1`. It will check the value of the `applications` key, to ensure that all applications that should be started before this application are running.

The application controller then creates an *application master* for the application. The application master is the group leader of all the processes in the application. The application master starts the application by calling the application callback function `start/2` in the module, and with the start argument, defined by the `mod` key in the `.app` file.

An application is stopped, but not unloaded, by calling:

```
7> application:stop(ch_app).
ok
```

The application master stops the application by telling the top supervisor to shutdown. The top supervisor tells all its child processes to shutdown etc. and the entire tree is terminated in reversed start order. The application master then calls the application callback function `stop/1` in the module defined by the `mod` key.

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"}]}
 ]}.
```

Par should be an atom, Val is any term. The application can retrieve the value of a configuration parameter by calling application:get_env(App, Par) or a number of similar functions, see application(3).

Example:

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

Eshell V5.2.3.6  (abort with ^G)
1> application:start(ch_app).
ok
2> application:get_env(ch_app, file).
{ok,"/usr/local/log"}
```

The values in the .app file can be overridden by values in a *system configuration file*. This is a file which contains configuration parameters for relevant applications:

```
[{Application1, [{Par11,Val11},...]},
 ...,
 {ApplicationN, [{ParN1,ValN1},...]}].
```

The system configuration should be called Name.config and Erlang should be started with the command line argument -config Name. See config(4) for more information.

Example: A file test.config is created with the following contents:

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

The value of file will override the value of file as defined in the .app file:

```
% erl -config test
Erlang (BEAM) emulator version 5.2.3.6 [hipe] [threads:0]

Eshell V5.2.3.6  (abort with ^G)
1> application:start(ch_app).
ok
2> application:get_env(ch_app, file).
{ok,"testlog"}
```

If release handling [page 59] is used, exactly one system configuration file should be used and that file should be called `sys.config`

The values in the `.app` file, as well as the values in a system configuration file, can be overridden directly from the command line:

```
% erl -ApplName Par1 Val1 ... ParN ValN
```

Example:

```
% erl -ch_app file '"testlog"'
Erlang (BEAM) emulator version 5.2.3.6 [hipe] [threads:0]

Eshell V5.2.3.6  (abort with ^G)
1> application:start(ch_app).
ok
2> application:get_env(ch_app, file).
{ok,"testlog"}
```

1.7.9 Application Start Types

A *start type* is defined when starting the application:

```
application:start(Application, Type)
```

`application:start(Application)` is the same as calling `application:start(Application, temporary)`. The type can also be `permanent` or `transient`:

- If a permanent application terminates, all other applications and the runtime system are also terminated.
- If a transient application terminates with reason `normal`, this is reported but no other applications are terminated. If a transient application terminates abnormally, that is with any other reason than `normal`, all other applications and the runtime system are also terminated.
- If a temporary application terminates, this is reported but no other applications are terminated.

It is always possible to stop an application explicitly by calling `application:stop/1`. Regardless of the mode, no other applications will be affected.

Note that transient mode is of little practical use, since when a supervision tree terminates, the reason is set to `shutdown`, not `normal`.

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*.

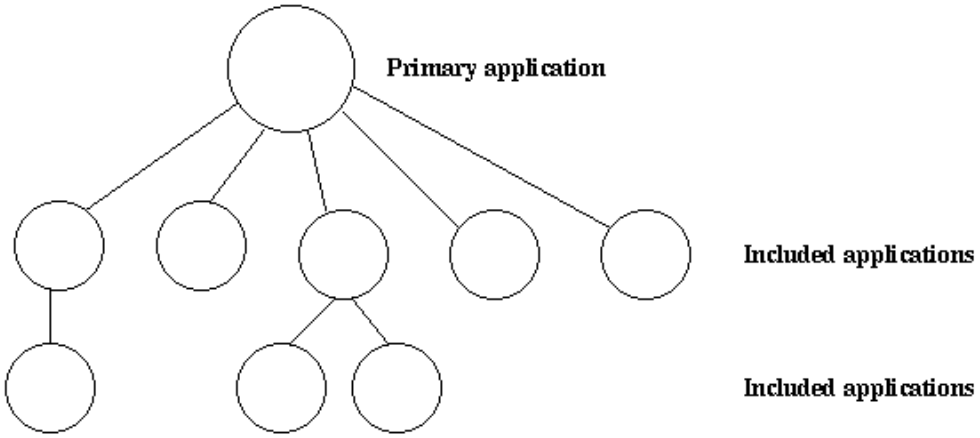

Figure 1.5: Primary Application and Included Applications.

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, prim_app,
 [{description, "Tree application"},
  {vsn, "1"},
  {modules, [prim_app_cb, prim_app_sup, prim_app_server]},
  {registered, [prim_app_server]},
  {included_applications, [incl_app]},
  {applications, [kernel, stdlib, sasl]},
  {mod, {prim_app_cb,[]}},
  {env, [{file, "/usr/local/log"}]}
 ]}.
```

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 as a list of tuples `{Phase,PhaseArgs}`, where `Phase` is an atom and `PhaseArgs` is a term. Also, the value of the `mod` key of the including application must be set to `{application_starter,[Module,StartArgs]}`, where `Module` as usual is the application callback module and `StartArgs` a term provided as argument to the callback function `Module:start/2`.

```
{application, prim_app,
 [{description, "Tree application"},
  {vsn, "1"},
  {modules, [prim_app_cb, prim_app_sup, prim_app_server]},
  {registered, [prim_app_server]},
  {included_applications, [incl_app]},
  {start_phases, [{init,[]}, {go,[]}]},
  {applications, [kernel, stdlib, sasl]},
  {mod, {application_starter,[prim_app_cb,[]]}},
  {env, [{file, "/usr/local/log"}]}
 ]}.

{application, incl_app,
 [{description, "Included application"},
  {vsn, "1"},
  {modules, [incl_app_cb, incl_app_sup, incl_app_server]},
  {registered, []},
  {start_phases, [{go,[]}]},
  {applications, [kernel, stdlib, sasl]},
  {mod, {incl_app_cb,[]}}
 ]}.
```

When starting a primary application with included applications, the primary application is started the normal way: The application controller creates an application master for the application, and the application master calls `Module:start(normal, StartArgs)` to start the top supervisor.

Then, for the primary application and each included application in top-down, left-to-right order, the application master calls `Module:start_phase(Phase, Type, PhaseArgs)` for each phase defined for for the primary application, in that order. Note that if a phase is not defined for an included application, the function is not called for this phase and application.

The following requirements apply to the `.app` file for an included application:

- The `{mod, {Module,StartArgs}}` option must be included. This option is used to find the callback module `Module` of the application. `StartArgs` is ignored, as `Module:start/2` is called only for the primary application.
- If the included application itself contains included applications, instead the option `{mod, {application_starter, [Module,StartArgs]}}` must be included.
- The `{start_phases, [{Phase,PhaseArgs}]}` option must be included, and the set of specified phases must be a subset of the set of phases specified for the primary application.

When starting `prim_app` as defined above, the application controller will call the following callback functions, before `application:start(prim_app)` returns a value:

```
application:start(prim_app)
 => prim_app_cb:start(normal, [])
 => prim_app_cb:start_phase(init, normal, [])
 => prim_app_cb:start_phase(go, normal, [])
 => incl_app_cb:start_phase(go, normal, [])
ok
```

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 Kernel module `global` or STDLIB module `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 and Stopping Distributed Applications

When all involved (mandatory) nodes have been started, the distributed application can be started by calling `application:start(Application)` at *all of these nodes*.

It is of course also possible to use a boot script (see Releases [page 45]) 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.

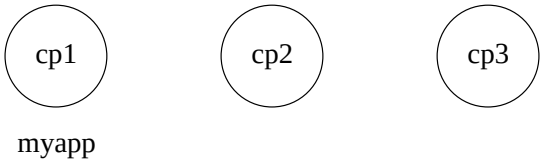

Figure 1.6: Application myapp - Situation 1

Similarly, the application must be stopped by calling application:stop(Application) at all involved nodes.

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 38]), 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.

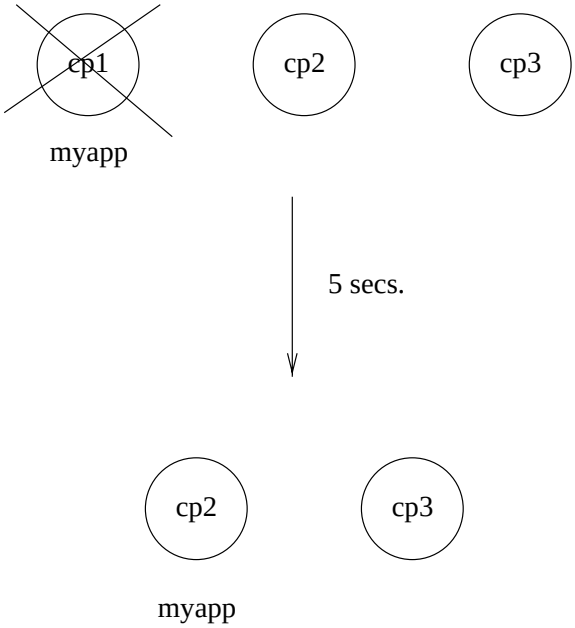

Figure 1.7: Application myapp - Situation 2

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

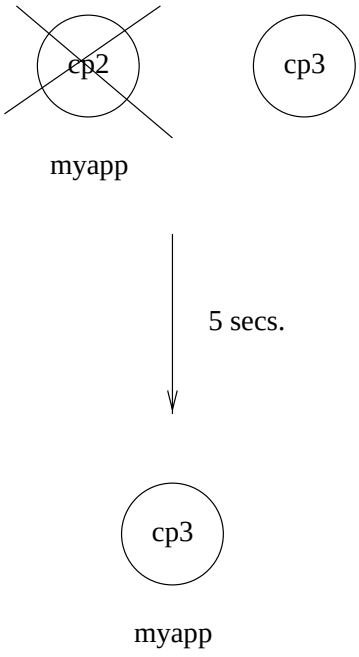

Figure 1.8: Application myapp - Situation 3

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.

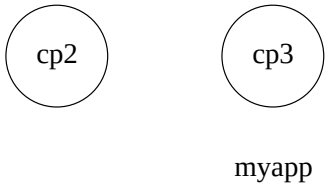

Figure 1.9: Application myapp - Situation 4

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.

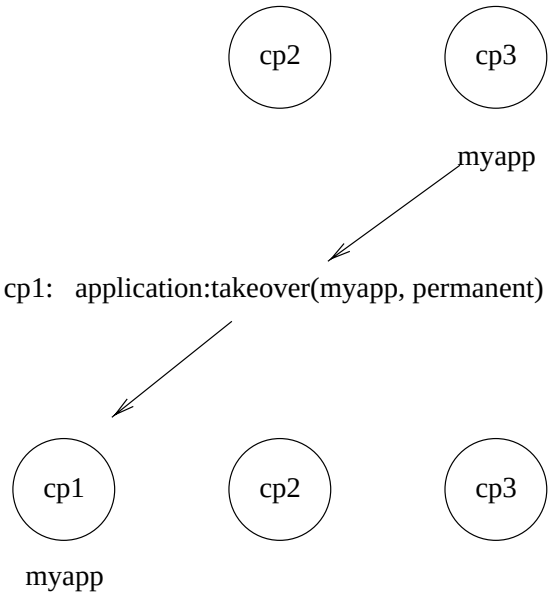

Figure 1.10: Application myapp - Situation 5

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 45] which defines which applications are included in the release.

The release resource file is used to generate boot scripts [page 46] and release packages [page 47]. 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 a *release resource file*, or in short `.rel` file, where we specify the name and version of the release, which ERTS version it is based on, and which applications it consists of:

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

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

`Name`, `Vsn` and `Evsn` are strings.

Each `Application` (atom) and `AppVsn` (string) is the name and version of an application included in the release. Note the the minimal release based on Erlang/OTP consists of the `kernel` and `stdlib` applications, so these applications must be included in the list.

Example: We want to make a release of `ch_app` from the Applications [page 32] chapter. It has the following `.app` file:

```
{application, ch_app,
 [{description, "Channel allocator"},
  {vsn, "1"},
  {modules, [ch_app, ch_sup, ch3]},
  {registered, [ch3]},
  {applications, [kernel, stdlib, sasl]},
  {mod, {ch_app,[]}}
 ]}.
```

The `.rel` file must also contain `kernel`, `stdlib` and `sasl`, since these applications are required by `ch_app`. We call the file `ch_rel-1.rel`:

```
{release,
 {"ch_rel", "A"},
 {erts, "5.3"},
 [{kernel, "2.9"},
  {stdlib, "1.12"},
  {sasl, "1.10"},
  {ch_app, "1"}]
}.
```

1.10.3 Generating Boot Scripts

There are tools in the SASL module `systools` available to build and check releases. The functions read the `.rel` and `.app` files and performs syntax and dependency checks. The function `systools:make_script/1,2` is used to generate a boot script (see System Principles).

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

This creates a boot script, both the readable version `ch_rel-1.script` and the binary version used by the runtime system, `ch_rel-1.boot`. `"ch_rel-1"` is the name of the `.rel` file, minus the extension. `local` is an option that means that the directories where the applications are found are used in the boot script, instead of `$ROOT/lib`. (`$ROOT` is the root directory of the installed release.) This is a useful way to test a generated boot script locally.

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

```
% erl -boot ch_rel-1
Erlang (BEAM) emulator version 5.3

Eshell V5.3  (abort with ^G)
1>
=PROGRESS REPORT==== 13-Jun-2003::12:01:15 ===
          supervisor: {local,sasl_safe_sup}
             started: [{pid,<0.33.0>},
                       {name,alarm_handler},
                       {mfa,{alarm_handler,start_link,[]}},
                       {restart_type,permanent},
                       {shutdown,2000},
                       {child_type,worker}]

...

=PROGRESS REPORT==== 13-Jun-2003::12:01:15 ===
         application: sasl
          started_at: nonode@nohost

...
=PROGRESS REPORT==== 13-Jun-2003::12:01:15 ===
         application: ch_app
          started_at: nonode@nohost
```

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: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 34], the binary boot script renamed to `start.boot`, and the `.rel` file.

```
% tar tf ch_rel-1.tar
lib/kernel-2.9/ebin/kernel.app
lib/kernel-2.9/ebin/application.beam
...
lib/stdlib-1.12/ebin/stdlib.app
lib/stdlib-1.12/ebin/beam_lib.beam
...
lib/sasl-1.10/ebin/sasl.app
lib/sasl-1.10/ebin/sasl.beam
...
lib/ch_app-1/ebin/ch_app.app
lib/ch_app-1/ebin/ch_app.beam
lib/ch_app-1/ebin/ch_sup.beam
lib/ch_app-1/ebin/ch3.beam
releases/A/start.boot
releases/ch_rel-1.rel
```

Note that a new boot script was generated, without the `local` option set, before the release package was made. In the release package, all application directories are placed under `lib`. Also, we do not know where the release package will be installed, so we do not want any hardcoded absolute paths in the boot script here.

If a `relup` file and/or a system configuration file called `sys.config` is found, these files are included in the release package as well. See Release Handling [page 51].

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

Refer to System Principles for how to install the first target system, using a release package, and to Release Handling [page 49] for how to install a new release package in an existing system.

1.10.5 Directory Structure

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

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

`lib` Application directories.

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

`releases/Vsn` `.rel` file and boot script `start.boot`.
If present in the release package,
`relup` and/or `sys.config`.

`bin` Top level Erlang runtime system executables.

Applications are not required to be located under the `$ROOT/lib` directory. Accordingly, several installation directories may exist which contain different parts of a system. For example, the previous example could be extended as follows:

```
$SECOND_ROOT/.../SApp1-SAVsn1/ebin
                              /priv
                /SApp2-SAVsn2/ebin
                              /priv
                ...
                /SAppN-SAVsnN/ebin
                              /priv

$THIRD_ROOT/TApp1-TAVsn1/ebin
                         /priv
           /TApp2-TAVsn2/ebin
                         /priv
           ...
           /TAppN-TAVsnN/ebin
                         /priv
```

The `$SECOND_ROOT` and `$THIRD_ROOT` are introduced as `variables` in the call to the `systools:make_script/2` function.

Disk-Less and/or Read-Only Clients

If a complete system consists of some disk-less and/or read-only client nodes, a `clients` directory should be added to the `$ROOT` directory. By a read-only node we mean a node with a read-only file system.

The `clients` directory should have one sub-directory per supported client node. The name of each client directory should be the name of the corresponding client node. As a minimum, each client directory should contain the `bin` and `releases` sub-directories. These directories are used to store information about installed releases and to appoint the current release to the client. Accordingly, the `$ROOT` directory contains the following:

```
$ROOT/...
    /clients/ClientName1/bin
                         /releases/Vsn
            /ClientName2/bin
                         /releases/Vsn
            ...
            /ClientNameN/bin
                         /releases/Vsn
```

This structure should be used if all clients are running the same type of Erlang machine. If there are clients running different types of Erlang machines, or on different operating systems, the `clients` directory could be divided into one sub-directory per type of Erlang machine. Alternatively, you can set up one `$ROOT` per type of machine. For each type, some of the directories specified for the `$ROOT` directory should be included:

```
$ROOT/...
    /clients/Type1/lib
                  /erts-EVsn
                  /bin
                  /ClientName1/bin
                               /releases/Vsn
                  /ClientName2/bin
                               /releases/Vsn
                  ...
                  /ClientNameN/bin
                               /releases/Vsn
            ...
            /TypeN/lib
                  /erts-EVsn
                  /bin
                  ...
```

With this structure, the root directory for clients of `Type1` is `$ROOT/clients/Type1`.

1.11 Release Handling

1.11.1 Release Handling Principles

An important feature of the Erlang programming language is the ability to change module code in run-time, *code replacement*, as described in *Erlang Reference Manual*.

Based on this feature, the OTP application SASL provides a framework for upgrading and downgrading between different versions of an entire release in run-time. This is what we call *release handling*.

The framework consists of off-line support (`systools`) for generating scripts and building release packages, and on-line support (`release_handler`) for unpacking and installing release packages.

Note that the minimal system based on Erlang/OTP, enabling release handling, thus consists of Kernel, STDLIB and SASL.

1. A release is created as described in the previous chapter Releases [page 45]. The release is transferred to and installed at target environment. Refer to *System Principles* for information of how to install the first target system.
2. Modifications, for example error corrections, are made to the code in the development environment.
3. At some point it is time to make a new version of release. The relevant `.app` files are updated and a new `.rel` file is written.
4. For each modified application, an application upgrade file [page 54], `.appup`, is created. In this file, it is described how to upgrade and/or downgrade between the old and new version of the application.
5. Based on the `.appup` files, a release upgrade file [page 55] called `relup`, is created. This file describes how to upgrade and/or downgrade between the old and new version of the entire release.
6. A new release package is made and transferred to the target system.
7. The new release package is unpacked using the release handler.
8. The new version of the release is installed, also using the release handler. This is done by evaluating the instructions in `relup`. Modules may be added, deleted or re-loaded, applications may be started, stopped or re-started etc. In some cases, it is even necessary to restart the entire emulator.
   If the installation fails, the system may be rebooted. The old release version is then automatically used.
9. If the installation succeeds, the new version is made the default version, which should now be used in case of a system reboot.

The next chapter, Appup Cookbook [page 60], contains examples of `.appup` files for typical cases of upgrades/downgrades that are normally easy to handle in run-time. However, there are a many aspects that can make release handling complicated. To name a few examples:

- Complicated or circular dependencies can make it difficult or even impossible to decide in which order things must be done without risking run-time errors during an upgrade or downgrade. Dependencies may be:
  - between nodes,
  - between processes, and
  - between modules.
- During release handling, non-affected processes continue normal execution. This may lead to timeouts or other problems. For example, new processes created in the time window between suspending processes using a certain module and loading a new version of this module, may execute old code.

It is therefore recommended that code is changed in as small steps as possible, and always kept backwards compatible.

1.11.2 Requirements

For release handling to work properly, the runtime system needs to have knowledge about which release it is currently running. It must also be able to change (in run-time) which boot script and system configuration file should be used if the system is rebooted, for example by `heart` after a failure. Therefore, Erlang must be started as an embedded system, see *Embedded System* for information on how to do this.

For system reboots to work properly, it is also required that the system is started with heart beat monitoring, see `erl(1)` and `heart(3)`.

Other requirements:

- The boot script included in a release package must be generated from the same `.rel` file as the release package itself.
  Information about applications are fetched from the script when an upgrade or downgrade is performed.
- The system must be configured using one and only one system configuration file, called `sys.config`.
  If found, this file is automatically included when a release package is created.
- All versions of a release, except the first one, must contain a `relup` file.
  If found, this file is automatically included when a release package is created.

1.11.3 Distributed Systems

If the system consists of several Erlang nodes, each node may use its own version of the release. The release handler is a locally registered process and must be called at each node where an upgrade or downgrade is required. There is a release handling instruction that can be used to synchronize the release handler processes at a number of nodes: `sync_nodes`. See `appup(4)`.

1.11.4 Release Handling Instructions

OTP supports a set of *release handling instructions* that is used when creating `.appup` files. The release handler understands a subset of these, the *low-level* instructions. To make it easier for the user, there are also a number of *high-level* instructions, which are translated to low-level instructions by `systools:make_relup`.

Here, some of the most frequently used instructions are described. The complete list of instructions is found in `appup(4)`.

First, some definitions:

Residence module The module where a process has its tail-recursive loop function(s). If the tail-recursive loop functions are implemented in several modules, all those modules are residence modules for the process.

Functional module A module which is not a residence module for any process.

Note that for a process implemented using an OTP behaviour, the behaviour module is the residence module for that process. The callback module is a functional module.

load_module

If a simple extension has been made to a functional module, it is sufficient to simply load the new version of the module into the system, and remove the old version. This is called *simple code replacement* and for this the following instruction is used:

```
{load_module, Module}
```

update

If a more complex change has been made, for example a change to the format of the internal state of a gen_server, simple code replacement is not sufficient. Instead it is necessary to suspend the processes using the module (to avoid that they try to handle any requests before the code replacement is completed), ask them to transform the internal state format and switch to the new version of the module, remove the old version and last, resume the processes. This is called *synchronized code replacement* and for this the following instructions are used:

```
{update, Module, {advanced, Extra}}
{update, Module, supervisor}
```

`update` with argument `{advanced,Extra}` is used when changing the internal state of a behaviour as described above. It will cause behaviour processes to call the callback function `code_change`, passing the term `Extra` and some other information as arguments. See the man pages for the respective behaviours and Appup Cookbook [page 61].

`update` with argument `supervisor` is used when changing the start specification of a supervisor. See Appup Cookbook [page 63].

The release handler finds the processes *using* a module to update by traversing the supervision tree of each running application and checking all the child specifications:

```
{Id, StartFunc, Restart, Shutdown, Type, Modules}
```

A process is using a module if the name is listed in `Modules` in the child specification for the process.

If `Modules=dynamic`, which is the case for event managers, the event manager process informs the release handler about the list of currently installed event handlers (gen_fsm) and it is checked if the module name is in this list instead.

The release handler suspends, asks for code change, and resumes processes by calling the functions `sys:suspend/1,2`, `sys:change_code/4,5` and `sys:resume/1,2` respectively.

add_module and delete_module

If a new module is introduced, the following instruction is used:

```
{add_module, Module}
```

The instruction loads the module and is absolutely necessary when running Erlang in embedded mode. It is not strictly required when running Erlang in interactive (default) mode, since the code server automatically searches for and loads unloaded modules.

The opposite of `add_module` is `delete_module` which unloads a module:

```
{delete_module, Module}
```

Note that any process, in any application, with `Module` as residence module, is killed when the instruction is evaluated. The user should therefore ensure that all such processes are terminated before deleting the module, to avoid a possible situation with failing supervisor restarts.

Application Instructions

Instruction for adding an application:

```
{add_application, Application}
```

Adding an application means that the modules defined by the `modules` key in the `.app` file are loaded using a number of `add_module` instructions, then the application is started.

Instruction for removing an application:

```
{remove_application, Application}
```

Removing an application means that the application is stopped, the modules are unloaded using a number of `delete_module` instructions and then the application specification is unloaded from the application controller.

Instruction for removing an application:

```
{restart_application, Application}
```

Restarting an application means that the application is stopped and then started again similar to using the instructions `remove_application` and `add_application` in sequence.

apply (low-level)

To call an arbitrary function from the release handler, the following instruction is used:

```
{apply, {M, F, A}}
```

The release handler will evalute `apply(M, F, A)`.

restart_new_emulator (low-level)

This instruction is used when changing to a new emulator version, or if a system reboot is needed for some other reason. Requires that the system is started with heart beat monitoring, see `erl(1)` and `heart(3)`.

When the release handler encounters the instruction, it shuts down the current emulator by calling `init:reboot()`, see `init(3)`. All processes are terminated gracefully and the system can then be rebooted by the heart program, using the new release version. This new version must still be made permanent when the new emulator is up and running. Otherwise, the old version is used in case of a new system reboot.

On UNIX, the release handler tells the heart program which command to use to reboot the system. Note that the environment variable `HEART_COMMAND`, normally used by the heart program, in this case is ignored. The command instead defaults to `$ROOT/bin/start`. Another command can be set by using the SASL configuration parameter `start_prg`, see `sasl(6)`.

1.11.5 Application Upgrade File

To define how to upgrade/downgrade between the current version and previous versions of an application, we create an *application upgrade file*, or in short `.appup` file. The file should be called `Application.appup`, where `Application` is the name of the application:

```
{Vsn,
 [{UpFromVsn1, InstructionsU1},
  ...,
  {UpFromVsnK, InstructionsUK}],
 [{DownToVsn1, InstructionsD1},
  ...,
  {DownToVsnK, InstructionsDK}]}.
```

`Vsn`, a string, is the current version of the application, as defined in the `.app` file. Each `UpFromVsn` is a previous version of the application to upgrade from, and each `DownToVsn` is a previous version of the application to downgrade to. Each `Instructions` is a list of release handling instructions.

The syntax and contents of the `appup` file are described in detail in `appup(4)`.

In the chapter Appup Cookbook [page 60], examples of `.appup` files for typical upgrade/downgrade cases are given.

Example: Consider the release `ch_rel-1` from the Releases [page 45] chapter. Assume we want to add a function `available/0` to the server `ch3` which returns the number of available channels:

(Hint: When trying out the example, make the changes in a copy of the original directory, so that the first versions are still available.)

```
-module(ch3).
-behaviour(gen_server).

-export([start_link/0]).
-export([alloc/0, free/1]).
-export([available/0]).
-export([init/1, handle_call/3, handle_cast/2]).

start_link() ->
    gen_server:start_link({local, ch3}, ch3, [], []).
```

```
alloc() ->
    gen_server:call(ch3, alloc).

free(Ch) ->
    gen_server:cast(ch3, {free, Ch}).

available() ->
    gen_server:call(ch3, available).

init(_Args) ->
    {ok, channels()}.

handle_call(alloc, _From, Chs) ->
    {Ch, Chs2} = alloc(Chs),
    {reply, Ch, Chs2};
handle_call(available, _From, Chs) ->
    N = available(Chs),
    {reply, N, Chs}.

handle_cast({free, Ch}, Chs) ->
    Chs2 = free(Ch, Chs),
    {noreply, Chs2}.
```

A new version of the `ch_app.app` file must now be created, where the version is updated:

```
{application, ch_app,
 [{description, "Channel allocator"},
  {vsn, "2"},
  {modules, [ch_app, ch_sup, ch3]},
  {registered, [ch3]},
  {applications, [kernel, stdlib, sasl]},
  {mod, {ch_app,[]}}
 ]}.
```

To upgrade `ch_app` from `"1"` to `"2"` (and to downgrade from `"2"` to `"1"`), we simply need to load the new (old) version of the `ch3` callback module. We create the application upgrade file `ch_app.appup` in the `ebin` directory:

```
{"2",
 [{"1", [{load_module, ch3}]}],
 [{"1", [{load_module, ch3}]}]
}.
```

1.11.6 Release Upgrade File

To define how to upgrade/downgrade between the new version and previous versions of a release, we create a *release upgrade file*, or in short `relup` file.

This file does not need to be created manually, it can be generated by `systools:make_relup/3,4`. The relevant versions of the `.rel` file, `.app` files and `.appup` files are used as input. It is deducted which applications should be added and deleted, and which applications that need to be upgraded and/or

downgraded. The instructions for this is fetched from the .appup files and transformed into a single list of low-level instructions in the right order.

If the relup file is relatively simple, it can be created manually. Remember that it should only contain low-level instructions.

The syntax and contents of the release upgrade file are described in detail in relup(4).

Example, continued from the previous section. We have a new version "2" of ch_app and an .appup file. We also need a new version of the .rel file. This time the file is called ch_rel-2.rel and the release version string is changed changed from "A" to "B":

```
{release,
 {"ch_rel", "B"},
 {erts, "5.3"},
 [{kernel, "2.9"},
  {stdlib, "1.12"},
  {sasl, "1.10"},
  {ch_app, "2"}]
}.
```

Now the relup file can be generated:

```
1> systools:make_relup("ch_rel-2", ["ch_rel-1"], ["ch_rel-1"]).
ok
```

This will generate a relup file with instructions for how to upgrade from version "A" ("ch_rel-1") to version "B" ("ch_rel-2") and how to downgrade from version "B" to version "A".

Note that both the old and new versions of the .app and .rel files must be in the code path, as well as the .appup and (new) .beam files. It is possible to extend the code path by using the option path:

```
1> systools:make_relup("ch_rel-2", ["ch_rel-1"], ["ch_rel-1"],
   [{path,["../ch_rel-1",
           "../ch_rel-1/lib/ch_app-1/ebin"]}]).
ok
```

1.11.7 Installing a Release

When we have made a new version of a release, a release package can be created with this new version and transferred to the target environment.

To install the new version of the release in run-time, the *release handler* is used. This is a process belonging to the SASL application, that handles unpacking, installation, and removal of release packages. It is interfaced through the module release_handler, which is described in detail in release_handler(3).

Assuming there is a target system up and running with installation root directory $ROOT, the release package with the new version of the release should be copied to $ROOT/releases.

The first action is to *unpack* the release package, the files are then extracted from the package:

```
release_handler:unpack_release(ReleaseName) => {ok, Vsn}
```

ReleaseName is the name of the release package except the .tar.gz extension. Vsn is the version of the unpacked release, as defined in its .rel file.

A directory $ROOT/lib/releases/Vsn will be created, where the .rel file, the boot script start.boot, the system configuration file sys.config and relup are placed. For applications with new version numbers, the application directories will be placed under $ROOT/lib. Unchanged applications are not affected.

An unpacked release can be *installed*. The release handler then evaluates the instructions in relup, step by step:

```
release_handler:install_release(Vsn) => {ok, FromVsn, []}
```

If an error occurs during the installation, the system is rebooted using the old version of the release. If installation succeeds, the system is afterwards using the new version of the release, but should anything happen and the system is rebooted, it would start using the previous version again. To be made the default version, the newly installed release must be made *permanent*, which means the previous version becomes *old*:

```
release_handler:make_permanent(Vsn) => ok
```

The system keeps information about which versions are old and permanent in the files $ROOT/releases/RELEASES and $ROOT/releases/start_erl.data.

To downgrade from Vsn to FromVsn, install_release must be called again:

```
release_handler:install_release(FromVsn) => {ok, Vsn, []}
```

An installed, but not permanent, release can be *removed*. Information about the release is then deleted from $ROOT/releases/RELEASES and the release specific code, that is the new application directories and the $ROOT/releases/Vsn directory, are removed.

```
release_handler:remove_release(Vsn) => ok
```

Example, continued from the previous sections:

1) Create a target system as described in *System Principles* of the first version "A" of ch_rel from the Releases [page 45] chapter. This time sys.config must be included in the release package. If no configuration is needed, the file should contain the empty list:

```
[].
```

2) Start the system as a simple target system. Note that in reality, it should be started as an embedded system. However, using erl with the correct boot script and .config file is enough for illustration purposes:

```
% cd $ROOT
% bin/erl -boot $ROOT/releases/A/start -config $ROOT/releases/A/sys
...
```

$ROOT is the installation directory of the target system.

3) In another Erlang shell, generate start scripts and create a release package for the new version "B". Remember to include (a possible updated) sys.config and the relup file, see Release Upgrade File [page 55] above.

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

The new release package now contains version “2” of ch_app and the relup file as well:

```
% tar tf ch_rel-2.tar
lib/kernel-2.9/ebin/kernel.app
lib/kernel-2.9/ebin/application.beam
...
lib/stdlib-1.12/ebin/stdlib.app
lib/stdlib-1.12/ebin/beam_lib.beam
...
lib/sasl-1.10/ebin/sasl.app
lib/sasl-1.10/ebin/sasl.beam
...
lib/ch_app-2/ebin/ch_app.app
lib/ch_app-2/ebin/ch_app.beam
lib/ch_app-2/ebin/ch_sup.beam
lib/ch_app-2/ebin/ch3.beam
releases/B/start.boot
releases/B/relup
releases/B/sys.config
releases/ch_rel-2.rel
```

4) Copy the release package ch_rel-2.tar.gz to the $ROOT/releases directory.

5) In the running target system, unpack the release package:

```
1> release_handler:unpack_release("ch_rel-2").
{ok,"B"}
```

The new application version ch_app-2 is installed under $ROOT/lib next to ch_app-1. The kernel, stdlib and sasl directories are not affected, as they have not changed.

Under $ROOT/releases, a new directory B is created, containing ch_rel-2.rel, start.boot, sys.config and relup.

6) Check if the function ch3:available/0 is available:

```
2> ch3:available().
** exited: {undef,[{ch3,available,[]},
                   {erl_eval,do_apply,5},
                   {shell,eval_loop,2}]} **
```

7) Install the new release. The instructions in $ROOT/releases/B/relup are executed one by one, resulting in the new version of ch3 being loaded. The function ch3:available/0 is now available:

```
3> release_handler:install_release("B").
{ok,"A",[]}
4> ch3:available().
3
5> code:which(ch3).
".../lib/ch_app-2/ebin/ch3.beam"
6> code:which(ch_sup).
".../lib/ch_app-1/ebin/ch_sup.beam"
```

Note that processes in `ch_app` for which code have not been updated, for example the supervisor, are still evaluating code from `ch_app-1`.

8) If the target system is now rebooted, it will use version "A" again. The "B" version must be made permanent, in order to be used when the system is rebooted.

```
7> release_handler:make_permanent("B").
ok
```

1.11.8 Updating Application Specifications

When a new version of a release is installed, the application specifications are automatically updated for all loaded applications.

> **Note:**
> The information about the new application specifications are fetched from the boot script included in the release package. It is therefore important that the boot script is generated from the same `.rel` file as is used to build the release package itself.

Specifically, the application configuration parameters are automatically updated according to (in increasing priority order):

1. The data in the boot script, fetched from the new application resource file `App.app`
2. The new `sys.config`
3. Command line arguments `-App Par Val`

This means that parameter values set in the other system configuration files, as well as values set using `application:set_env/3`, are disregarded.

When an installed release is made permanent, the system process `init` is set to point out the new `sys.config`.

After the installation, the application controller will compare the old and new configuration parameters for all running applications and call the callback function:

```
Module:config_change(Changed, New, Removed)
```

`Module` is the application callback module as defined by the `mod` key in the `.app` file. `Changed` and `New` are lists of `{Par,Val}` for all changed and added configuration parameters, respectively. `Removed` is a list of all parameters `Par` that have been removed.

The function is optional and may be omitted when implementing an application callback module.

1.12 Appup Cookbook

This chapter contains examples of `.appup` files for typical cases of upgrades/downgrades done in run-time.

1.12.1 Changing a Functional Module

When a change has been made to a functional module, for example if a new function has been added or a bug has been corrected, simple code replacement is sufficient.

Example:

```
{"2",
 [{"1", [{load_module, m}]}],
 [{"1", [{load_module, m}]}]
}.
```

1.12.2 Changing a Residence Module

In a system implemented according to the OTP Design Principles, all processes, except system processes and special processes, reside in one of the behaviours `supervisor`, `gen_server`, `gen_fsm` or `gen_event`. These belong to the STDLIB application and upgrading/downgrading normally requires an emulator restart.

OTP thus provides no support for changing residence modules except in the case of special processes [page 62].

1.12.3 Changing a Callback Module

A callback module is a functional module, and for code extensions simple code replacement is sufficient.

Example: When adding a function to `ch3` as described in the example in Release Handling [page 54], `ch_app.appup` looks as follows:

```
{"2",
 [{"1", [{load_module, ch3}]}],
 [{"1", [{load_module, ch3}]}]
}.
```

OTP also supports changing the internal state of behaviour processes, see Changing Internal State [page 61] below.

1.12.4 Changing Internal State

In this case, simple code replacement is not sufficient. The process must explicitly transform its state using the callback function `code_change` before switching to the new version of the callback module. Thus synchronized code replacement is used.

Example: Consider the gen_server `ch3` from the chapter about the gen_server behaviour [page 6]. The internal state is a term `Chs` representing the available channels. Assume we want add a counter `N` which keeps track of the number of `alloc` requests so far. This means we need to change the format to `{Chs,N}`.

The `.appup` file could look as follows:

```
{"2",
 [{"1", [{update, ch3, {advanced, []}}]}],
 [{"1", [{update, ch3, {advanced, []}}]}]
}.
```

The third element of the `update` instruction is a tuple `{advanced,Extra}` which says that the affected processes should do a state transformation before loading the new version of the module. This is done by the processes calling the callback function `code_change` (see `gen_server(3)`). The term `Extra`, in this case `[]`, is passed as-is to the function:

```
-module(ch3).
...
-export([code_change/3]).
...
code_change({down, _Vsn}, {Chs, N}, _Extra) ->
    {ok, Chs};
code_change(_Vsn, Chs, _Extra) ->
    {ok, {Chs, 0}}.
```

The first argument is `{down,Vsn}` in case of a downgrade, or `Vsn` in case of an upgrade. The term `Vsn` is fetched from the 'original' version of the module, i.e. the version we are upgrading from, or downgrading to.

The version is defined by the module attribute `vsn`, if any. There is no such attribute in `ch3`, so in this case the version is the checksum (a huge integer) of the BEAM file, an uninteresting value which is ignored.

(The other callback functions of `ch3` need to be modified as well and perhaps a new interface function added, this is not shown here).

1.12.5 Module Dependencies

Assume we extend a module by adding a new interface function, as in the example in Release Handling [page 54], where a function `available/0` is added to `ch3`.

If we also add a call to this function, say in the module `m1`, a run-time error could occur during release upgrade if the new version of `m1` is loaded first and calls `ch3:available/0` before the new version of `ch3` is loaded.

Thus, `ch3` must be loaded before `m1` is, in the upgrade case, and vice versa in the downgrade case. We say that `m1` *is dependent on* `ch3`. In a release handling instruction, this is expressed by the element `DepMods`:

```
{load_module, Module, DepMods}
{update, Module, {advanced, Extra}, DepMods}
```

DepMods is a list of modules, on which Module is dependent.

Example: The module m1 in the application myapp is dependent on ch3 when upgrading from “1” to “2”, or downgrading from “2” to “1”:

myapp.appup:

```
{"2",
 [{"1", [{load_module, m1, [ch3]}]}],
 [{"1", [{load_module, m1, [ch3]}]}]
}.
```

ch_app.appup:

```
{"2",
 [{"1", [{load_module, ch3}]}],
 [{"1", [{load_module, ch3}]}]
}.
```

If m1 and ch3 had belonged to the same application, the .appup file could have looked like this:

```
{"2",
 [{"1",
   [{load_module, ch3},
    {load_module, m1, [ch3]}]}],
 [{"1",
   [{load_module, ch3},
    {load_module, m1, [ch3]}]}]
}.
```

Note that it is m1 that is dependent on ch3 also when downgrading. systools knows the difference between up- and downgrading and will generate a correct relup, where ch3 is loaded before m1 when upgrading but m1 is loaded before ch3 when downgrading.

1.12.6 Changing Code For a Special Process

In this case, simple code replacement is not sufficient. When a new version of a residence module for a special process is loaded, the process must make a fully qualified call to its loop function to switch to the new code. Thus synchronized code replacement must be used.

> **Note:**
> The name(s) of the user-defined residence module(s) must be listed in the Modules part of the child specification for the special process, in order for the release handler to find the process.

Example. Consider the example ch4 from the chapter about sys and proc_lib [page 26]. When started by a supervisor, the child specification could look like this:

```
{ch4, {ch4, start_link, []},
 permanent, brutal_kill, worker, [ch4]}
```

If ch4 is part of the application sp_app and a new version of the module should be loaded when upgrading from version “1” to “2” of this application, sp_app.appup could look like this:

```
{"2",
 [{"1", [{update, ch4, {advanced, []}}]}],
 [{"1", [{update, ch4, {advanced, []}}]}]
}.
```

The update instruction must contain the tuple {advanced,Extra}. The instruction will make the special process call the callback function system_code_change/4, a function the user must implement. The term Extra, in this case [], is passed as-is to system_code_change/4:

```
-module(ch4).
...
-export([system_code_change/4]).
...

system_code_change(Chs, _Module, _OldVsn, _Extra) ->
    {ok, Chs}.
```

The first argument is the internal state State passed from the function sys:handle_system_msg(Request, From, Parent, Module, Deb, State), called by the special process when a system message is received. In ch4, the internal state is the set of available channels Chs.

The second argument is the name of the module (ch4).

The third argument is Vsn or {down,Vsn} as described for gen_server:code_change/3 [page 61].

In this case, all arguments but the first are ignored and the function simply returns the internal state again. This is enough if the code only has been extended. If we had wanted to change the internal state (similar to the example in Changing Internal State [page 61]), it would have been done in this function and {ok,Chs2} returned.

1.12.7 Changing a Supervisor

The supervisor behaviour supports changing the internal state, i.e. changing restart strategy and maximum restart frequency properties, as well as changing existing child specifications.

Adding and deleting child processes are also possible, but not handled automatically. Instructions must be given by in the .appup file.

Changing Properties

Since the supervisor should change its internal state, synchronized code replacement is required. However, a special `update` instruction must be used.

The new version of the callback module must be loaded first both in the case of upgrade and downgrade. Then the new return value of `init/1` can be checked and the internal state be changed accordingly.

The following `upgrade` instruction is used for supervisors:

```
{update, Module, supervisor}
```

Example: Assume we want to change the restart strategy of `ch_sup` from the Supervisor Behaviour [page 19] chapter from one_for_one to one_for_all. We change the callback function `init/1` in `ch_sup.erl`:

```
-module(ch_sup).
...

init(_Args) ->
    {ok, {{one_for_all, 1, 60}, ...}}.
```

The file `ch_app.appup`:

```
{"2",
 [{"1", [{update, ch_sup, supervisor}]}],
 [{"1", [{update, ch_sup, supervisor}]}]
}.
```

Changing Child Specifications

The instruction, and thus the `.appup` file, when changing an existing child specification, is the same as when changing properties as described above:

```
{"2",
 [{"1", [{update, ch_sup, supervisor}]}],
 [{"1", [{update, ch_sup, supervisor}]}]
}.
```

The changes do not affect existing child processes. For example, changing the start function only specifies how the child process should be restarted, if needed later on.

Note that the id of the child specification cannot be changed.

Note also that changing the `Modules` field of the child specification may affect the release handling process itself, as this field is used to identify which processes are affected when doing a synchronized code replacement.

Adding And Deleting Child Processes

As stated above, changing child specifications does not affect existing child processes. New child specifications are automatically added, but not deleted. Also, child processes are not automatically started or terminated. Instead, this must be done explicitly using `apply` instructions.

Example: Assume we want to add a new child process `m1` to `ch_sup` when upgrading `ch_app` from "1" to "2". This means `m1` should be deleted when downgrading from "2" to "1":

```
{"2",
 [{"1",
   [{update, ch_sup, supervisor},
    {apply, {supervisor, restart_child, [ch_sup, m1]}}
   ]}],
 [{"1",
   [{apply, {supervisor, terminate_child, [ch_sup, m1]}},
    {apply, {supervisor, delete_child, [ch_sup, m1]}},
    {update, ch_sup, supervisor}
   ]}]
}.
```

Note that the order of the instructions is important.

Note also that the supervisor must be registered as `ch_sup` for the script to work. If the supervisor is not registered, it cannot be accessed directly from the script. Instead a help function that finds the pid of the supervisor and calls `supervisor:restart_child` etc. must be written, and it is this function that should be called from the script using the `apply` instruction.

If the module `m1` is introduced in version "2" of `ch_app`, it must also be loaded when upgrading and deleted when downgrading:

```
{"2",
 [{"1",
   [{add_module, m1},
    {update, ch_sup, supervisor},
    {apply, {supervisor, restart_child, [ch_sup, m1]}}
   ]}],
 [{"1",
   [{apply, {supervisor, terminate_child, [ch_sup, m1]}},
    {apply, {supervisor, delete_child, [ch_sup, m1]}},
    {update, ch_sup, supervisor},
    {delete_module, m1}
   ]}]
}.
```

Note again that the order of the instructions is important. When upgrading, `m1` must be loaded and the supervisor's child specification changed, before the new child process can be started. When downgrading, the child process must be terminated before child specification is changed and the module is deleted.

1.12.8 Adding or Deleting a Module

Example: A new functional module `m` is added to `ch_app`:

```
{"2",
 [{"1", [{add_module, m}]}],
 [{"1", [{delete_module, m}]}]
```

1.12.9 Starting or Terminating a Process

In a system structured according to the OTP design principles, any process would be a child process belonging to a supervisor, see Adding and Deleting Child Processes [page 64] above.

1.12.10 Adding or Removing an Application

When adding or removing an application, no `.appup` file is needed. When generating `relup`, the `.rel` files are compared and `add_application` and `remove_application` instructions are added automatically.

1.12.11 Restarting an Application

Restarting an application is useful when a change is too complicated to be made without restarting the processes, for example if the supervisor hierarchy has been restructured.

Example: When adding a new child `m1` to `ch_sup`, as in the example above [page 64], an alternative to updating the supervisor is to restart the entire application:

```
{"2",
 [{"1", [{restart_application, ch_app}]}],
 [{"1", [{restart_application, ch_app}]}]
}.
```

1.12.12 Changing an Application Specification

When installing a release, the application specifications are automatically updated before evaluating the `relup` script. Hence, no instructions are needed in the `.appup` file:

```
{"2",
 [{"1", []}],
 [{"1", []}]
}.
```

1.12.13 Changing Application Configuration

Changing an application configuration by updating the `env` key in the `.app` file is an instance of changing an application specification, see above [page 66].

Alternatively, application configuration parameters can be added or updated in `sys.config`.

1.12.14 Changing Included Applications

The release handling instructions for adding, removing and restarting applications apply to primary applications only. There are no corresponding instructions for included applications. However, since an included application is really a supervision tree with a topmost supervisor, started as a child process to a supervisor in the including application, a `relup` file can be manually created.

Example: Assume we have a release containing an application `prim_app` which have a supervisor `prim_sup` in its supervision tree.

In a new version of the release, our example application `ch_app` should be included in `prim_app`. That is, its topmost supervisor `ch_sup` should be started as a child process to `prim_sup`.

1) Edit the code for `prim_sup`:

```
init(...) ->
    {ok, {...supervisor flags...,
          [...,
           {ch_sup, {ch_sup,start_link,[]},
            permanent,infinity,supervisor,[ch_sup]},
           ...]}}.
```

2) Edit the `.app` file for `prim_app`:

```
{application, prim_app,
 [...,
  {vsn, "2"},
  ...,
  {included_applications, [ch_app]},
  ...
 ]}.
```

3) Create a new `.rel` file, including `ch_app`:

```
{release,
 ...,
 [...,
  {prim_app, "2"},
  {ch_app, "1"}]}.
```

Application Restart

4a) One way to start the included application is to restart the entire `prim_app` application. Normally, we would then use the `restart_application` instruction in the `.appup` file for `prim_app`.

However, if we did this and then generated a `relup` file, not only would it contain instructions for restarting (i.e. removing and adding) `prim_app`, it would also contain instructions for starting `ch_app` (and stopping it, in the case of downgrade). This is due to the fact that `ch_app` is included in the new `.rel` file, but not in the old one.

Instead, a correct `relup` file can be created manually, either from scratch or by editing the generated version. The instructions for starting/stopping `ch_app` are replaced by instructions for loading/unloading the application:

```
{"B",
 [{"A",
   [],
   [{load_object_code,{ch_app,"1",[ch_sup,ch3]}},
    {load_object_code,{prim_app,"2",[prim_app,prim_sup]}},
    point_of_no_return,
    {apply,{application,stop,[prim_app]}},
    {remove,{prim_app,brutal_purge,brutal_purge}},
    {remove,{prim_sup,brutal_purge,brutal_purge}},
    {purge,[prim_app,prim_sup]},
    {load,{prim_app,brutal_purge,brutal_purge}},
    {load,{prim_sup,brutal_purge,brutal_purge}},
    {load,{ch_sup,brutal_purge,brutal_purge}},
    {load,{ch3,brutal_purge,brutal_purge}},
    {apply,{application,load,[ch_app]}},
    {apply,{application,start,[prim_app,permanent]}}]},
 [{"A",
   [],
   [{load_object_code,{prim_app,"1",[prim_app,prim_sup]}},
    point_of_no_return,
    {apply,{application,stop,[prim_app]}},
    {apply,{application,unload,[ch_app]}},
    {remove,{ch_sup,brutal_purge,brutal_purge}},
    {remove,{ch3,brutal_purge,brutal_purge}},
    {purge,[ch_sup,ch3]},
    {remove,{prim_app,brutal_purge,brutal_purge}},
    {remove,{prim_sup,brutal_purge,brutal_purge}},
    {purge,[prim_app,prim_sup]},
    {load,{prim_app,brutal_purge,brutal_purge}},
    {load,{prim_sup,brutal_purge,brutal_purge}},
    {apply,{application,start,[prim_app,permanent]}}]}]
}.
```

Supervisor Change

4b) Another way to start the included application (or stop it in the case of downgrade) is by combining instructions for adding and removing child processes to/from `prim_sup` with instructions for loading/unloading all `ch_app` code and its application specification.

Again, the `relup` file is created manually. Either from scratch or by editing a generated version. Load all code for `ch_app` first, and also load the application specification, before `prim_sup` is updated. When downgrading, `prim_sup` should be updated first, before the code for `ch_app` and its application specification are unloaded.

```
{"B",
 [{"A",
   [],
   [{load_object_code,{ch_app,"1",[ch_sup,ch3]}},
    {load_object_code,{prim_app,"2",[prim_sup]}},
    point_of_no_return,
    {load,{ch_sup,brutal_purge,brutal_purge}},
    {load,{ch3,brutal_purge,brutal_purge}},
    {apply,{application,load,[ch_app]}},
```

```
    {suspend,[prim_sup]},
    {load,{prim_sup,brutal_purge,brutal_purge}},
    {code_change,up,[{prim_sup,[]}]},
    {resume,[prim_sup]},
    {apply,{supervisor,restart_child,[prim_sup,ch_sup]}}]},
 [{"A",
   [],
   [{load_object_code,{prim_app,"1",[prim_sup]}},
    point_of_no_return,
    {apply,{supervisor,terminate_child,[prim_sup,ch_sup]}},
    {apply,{supervisor,delete_child,[prim_sup,ch_sup]}},
    {suspend,[prim_sup]},
    {load,{prim_sup,brutal_purge,brutal_purge}},
    {code_change,down,[{prim_sup,[]}]},
    {resume,[prim_sup]},
    {remove,{ch_sup,brutal_purge,brutal_purge}},
    {remove,{ch3,brutal_purge,brutal_purge}},
    {purge,[ch_sup,ch3]},
    {apply,{application,unload,[ch_app]}}]}]
}.
```

1.12.15 Changing Non-Erlang Code

Changing code for a program written in another programming language than Erlang, for example a port program, is very application dependent and OTP provides no special support for it.

Example, changing code for a port program: Assume that the Erlang process controlling the port is a gen_server `portc` and that the port is opened in the callback function `init/1`:

```
init(...) ->
    ...,
    PortPrg = filename:join(code:priv_dir(App), "portc"),
    Port = open_port({spawn,PortPrg}, [...]),
    ...,
    {ok, #state{port=Port, ...}}.
```

If the port program should be updated, we can extend the code for the gen_server with a `code_change` function which closes the old port and opens a new port. (If necessary, the gen_server may first request data that needs to be saved from the port program and pass this data to the new port):

```
code_change(_OldVsn, State, port) ->
    State#state.port ! close,
    receive
        {Port,close} ->
            true
    end,
    PortPrg = filename:join(code:priv_dir(App), "portc"),
    Port = open_port({spawn,PortPrg}, [...]),
    {ok, #state{port=Port, ...}}.
```

Update the application version number in the `.app` file and write an `.appup` file:

```
["2",
 [{"1", [{update, portc, {advanced,port}}]}],
 [{"1", [{update, portc, {advanced,port}}]}]
].
```

Make sure the `priv` directory where the C program is located is included in the new release package:

```
1> systools:make_tar("my_release", [{dirs,[priv]}]).
...
```

1.12.16 Emulator Restart

If the emulator can or should be restarted, the very simple `.relup` file can be created manually:

```
{"B",
 [{"A",
   [],
   [restart_new_emulator]}],
 [{"A",
   [],
   [restart_new_emulator]}]
}.
```

This way, the release handler framework with automatic packing and unpacking of release packages, automatic path updates etc. can be used without having to specify `.appup` files.

If some transformation of persistent data, for example database contents, needs to be done before installing the new release version, instructions for this can be added to the `.relup` file as well.

List of Figures