

Erlang/OTP System Documentation

Copyright © 1997-2023 Ericsson AB. All Rights Reserved.
Erlang/OTP System Documentation 13.1.4
January 17, 2023

Copyright © 1997-2023 Ericsson AB. All Rights Reserved.
Licensed under the Apache License, Version 2.0 (the "License"); you may not use this file except in compliance with the License. You may obtain a copy of the License at http://www.apache.org/licenses/LICENSE-2.0 Unless required by applicable law or agreed to in writing, software distributed under the License is distributed on an "AS IS" BASIS, WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied. See the License for the specific language governing permissions and limitations under the License. Ericsson AB. All Rights Reserved..

January 17, 2023

1 General Information

1.1 Deprecations

1.1.1 Introduction

This document lists all deprecated functionality in Erlang/OTP. For more information regarding the strategy regarding deprecations see the documentation of Support, Compatibility, Deprecations, and Removal.

1.1.2 OTP 25

Functions Deprecated in OTP 25

- `crypto:crypto_dyn_iv_init/3` (see the documentation for details)
- `crypto:crypto_dyn_iv_update/3` (see the documentation for details)
- `ct_slave:_/_` (use ?CT_PEER(), or the 'peer' module instead)
- `erts_alloc_config:_/_` (this module will be removed in OTP 26.0. See the documentation for details)
- `httpd_util:decode_hex/1` (use uri_string:unquote function instead)
- `httpd_util:encode_hex/1` (use uri_string:quote function instead)
- `slave:_/_` (use the 'peer' module instead)

1.1.3 OTP 24

Erlang Distribution Without Large Node Container Support

Communication over the Erlang distribution without support for large node container data types (version 4) is as of OTP 24 deprecated and is scheduled for removal in OTP 26. That is, as of OTP 26, support for large node container data types will become mandatory.

Old Link Protocol

The old link protocol used when communicating over the Erlang distribution is as of OTP 24 deprecated and support for it is scheduled for removal in OTP 26. As of OTP 26, the new link protocol will become mandatory. That is, Erlang nodes will then refuse to connect to nodes not implementing the new link protocol. If you implement the Erlang distribution yourself, you are, however, encouraged to implement the new link protocol as soon as possible since the old protocol can cause links to enter an inconsistent state.

?NO_APP macro

The ?NO_APP macro in the edoc include file `edoc_doclet.hrl` has been deprecated.

Functions Deprecated in OTP 24

- `code:is_module_native/1` (HiPE has been removed)
- `disk_log:accessible_logs/0` (use disk_log:all/0 instead)
- `disk_log:lclose/1` (use disk_log:close/1 instead)
- `disk_log:lclose/2` (use disk_log:close/1 instead)
- `erlang:phash/2` (use erlang:phash2/2 instead)
- `ftp:start_service/1` (use ftp:open/2 instead)

- `ftp:stop_service/1` (use ftp:close/1 instead)
- `httpd_util:flatlength/1` (use erlang:iolist_size/1 instead)
- `httpd_util:hexlist_to_integer/1` (use erlang:list_to_integer/2 with base 16 instead)
- `httpd_util:integer_to_hexlist/1` (use erlang:integer_to_list/2 with base 16 instead)
- `httpd_util:strip/1` (use string:trim/1 instead)
- `httpd_util:suffix/1` (use filename:extension/1 and string:trim/2 instead)
- `zlib:adler32/2` (use erlang:adler32/1 instead)
- `zlib:adler32/3` (use erlang:adler32/2 instead)
- `zlib:adler32_combine/4` (use erlang:adler_combine/3 instead)
- `zlib:crc32/1` (use erlang:crc32/1 on the uncompressed data instead)
- `zlib:crc32/2` (use erlang:crc32/1 instead)
- `zlib:crc32/3` (use erlang:crc32/2 instead)
- `zlib:crc32_combine/4` (use erlang:crc32_combine/3 instead)
- `zlib:getBufSize/1` (this function will be removed in a future release)
- `zlib:inflateChunk/1` (use safeInflate/2 instead)
- `zlib:inflateChunk/2` (use safeInflate/2 instead)
- `zlib:setBufSize/2` (this function will be removed in a future release)

1.1.4 OTP 23

Crypto Old API

The Old API is deprecated as of OTP 23 and has been removed in OTP 24.

For replacement functions see the New API.

http_uri

Since OTP 21 the recommended module to handle URIs is uri_string. The module http_uri does not provide a implementation that satisfies the RFC.

ssh

The public key algorithm `'ssh-rsa` is regarded as insecure due to its usage of SHA1, and is therefore deprecated. It will not be available by default from OTP-24.

The public key algorithm `'ssh-dss` is regarded as insecure due to its usage of SHA1 and its short key length, and is therefore deprecated. It is not available by default from OTP-23.

Distributed Disk Logs

As of OTP 23, the distributed `disk_log` feature has been deprecated and it has also been removed in OTP 24.

erl_interface registry

As of OTP 23, the `registry` functionality part of `erl_interface` has been deprecated and it has also been removed in OTP 24.

Functions Deprecated in OTP 23

- `http_uri:decode/1` (use uri_string:unquote function instead)
- `http_uri:encode/1` (use uri_string:quote function instead)
- `httpd:parse_query/1` (use uri_string:dissect_query/1 instead)

1.1.5 OTP 22

VxWorks Support

Some parts of OTP has had limited VxWorks support, such as for example `erl_interface`. This support is as of OTP 22 formally deprecated and has also been removed in OTP 23.

Legacy parts of erl_interface

The old legacy `erl_interface` library (functions with prefix `erl_`) is deprecated as of OTP 22. These parts of `erl_interface` has been informally deprecated for a very long time. You typically want to replace the usage of the `erl_interface` library with the use of the `ei` library which also is part of the `erl_interface` application. The old legacy `erl_interface` library has also been removed in OTP 23.

System Events

The format of "System Events" as defined in the man page for sys has been clarified and cleaned up. Due to this, code that relied on the internal badly documented previous (before this change) format of OTP's "System Events", needs to be changed.

In the wake of this the function sys:get_debug/3 that returns data with undocumented and internal format (and therefore is practically useless) has been deprecated, and a new function sys:get_log/1 has been added, that hopefully does what the deprecated function was intended for.

Functions Deprecated in OTP 22

- `net:broadcast/3` (use rpc:eval_everywhere/3 instead)
- `net:call/4` (use rpc:call/4 instead)
- `net:cast/4` (use rpc:cast/4 instead)
- `net:ping/1` (use net_adm:ping/1 instead)
- `net:sleep/1` (use 'receive after T -> ok end' instead)
- `sys:get_debug/3` (incorrectly documented and only for internal use. Can often be replaced with sys:get_log/1)

1.1.6 OTP 20

Functions Deprecated in OTP 20

- `crypto:rand_uniform/2` (use rand:uniform/1 instead)
- `gen_fsm:_/_` (use the 'gen_statem' module instead)

1.1.7 OTP 19

SSL/TLS

For security reasons SSL-3.0 is no longer supported by default, but can be configured.

Functions Deprecated in OTP 19

- `code:rehash/0` (the code path cache feature has been removed)
- `queue:lait/1` (use queue:liat/1 instead)
- `random:_/_` (use the 'rand' module instead)

1.1.8 OTP 18

erlang:now/0

New time functionality and a new time API was introduced. For more information see the Time and Time Correction chapter in the ERTS User's guide and specifically the Dos and Donts section on how to replace usage of `erlang:now/0`.

httpd_conf module

API functions in the module `httpd_conf` was deprecated in favor of standard modules such as `lists`, `string`, `filelib`, and `erlang`.

Functions Deprecated in OTP 18

- `erlang:now/0` (see the "Time and Time Correction in Erlang" chapter of the ERTS User's Guide for more information)

1.1.9 OTP 16

Functions Deprecated in OTP 16

- `wxCalendarCtrl:enableYearChange/1` (not available in wxWidgets-2.9 and later)
- `wxCalendarCtrl:enableYearChange/2` (not available in wxWidgets-2.9 and later)

1.1.10 OTP 12

inets - httpd Apache config files

A new config file format was introduced.

Functions Deprecated in OTP 12

- `auth:cookie/0` (use erlang:get_cookie/0 instead)
- `auth:cookie/1` (use erlang:set_cookie/2 instead)
- `auth:is_auth/1` (use net_adm:ping/1 instead)
- `auth:node_cookie/_` (use erlang:set_cookie/2 and net_adm:ping/1 instead)
- `calendar:local_time_to_universal_time/1` (use calendar:local_time_to_universal_time_dst/1 instead)

1.2 Removed Functionality

1.2.1 Introduction

This document lists all removed functionality in Erlang/OTP. For more information regarding the strategy regarding removals see the documentation of Support, Compatibility, Deprecations, and Removal.

1.2.2 OTP 25

Functions Removed in OTP 25

- `filename:safe_relative_path/1` (use filelib:safe_relative_path/2 instead)
- `http_uri:parse/1` (use uri_string functions instead)
- `http_uri:parse/2` (use uri_string functions instead)
- `http_uri:scheme_defaults/0` (use uri_string functions instead)

- `public_key:ssh_decode/2` (use ssh_file:decode/2 instead)
- `public_key:ssh_encode/2` (use ssh_file:encode/2 instead)
- `public_key:ssh_hostkey_fingerprint/1` (use ssh:hostkey_fingerprint/1 instead)
- `public_key:ssh_hostkey_fingerprint/2` (use ssh:hostkey_fingerprint/2 instead)

1.2.3 OTP 24

erl_interface registry

The `registry` functionality part of `erl_interface` was as of OTP 23 deprecated and was removed in OTP 24.

Compilation of Latin-1 Encoded Erlang Files

The Erlang compiler now refuses to compile source files encoded in Latin-1 without a `%% coding: latin-1` comment at the beginning of the file.

igor and erl_tidy modules in syntax_tools

The `igor` and `erl_tidy` modules have been removed from OTP and is now maintained by their original author Richard Carlsson. They can be found at **github.com/richcarl/igor** and **github.com/richcarl/erl_tidy**, respectively.

Distributed Disk Logs

The distributed `disk_log` feature was as of OTP 23 deprecated and was removed in OTP 24.

Old Crypto API

The Old API was removed in OTP 24. The support was formally deprecated as of OTP 23.

For replacement functions see the New API.

Megaco version 3 encoding config

The pre-release version 3 encoding configs; `prev3a`, `prev3b` and `prev3c` was removed in OTP 24. Use the full version instead.

The (encoding) config option for the full version, `{version3, 3}`, will still be supported, even though its no longer necessary to specify it this way.

Functions Removed in OTP 24

- `crypto:block_decrypt/3` (use crypto:crypto_one_time/4 or crypto:crypto_init/3 + crypto:crypto_update/2 + crypto:crypto_final/1 instead)
- `crypto:block_decrypt/4` (use crypto:crypto_one_time/5, crypto:crypto_one_time_aead/6,7 or crypto:crypto_(dyn_iv)?_init + crypto:crypto_(dyn_iv)?_update + crypto:crypto_final instead)
- `crypto:block_encrypt/3` (use crypto:crypto_one_time/4 or crypto:crypto_init/3 + crypto:crypto_update/2 + crypto:crypto_final/1 instead)
- `crypto:block_encrypt/4` (use crypto:crypto_one_time/5, crypto:crypto_one_time_aead/6,7 or crypto:crypto_(dyn_iv)?_init + crypto:crypto_(dyn_iv)?_update + crypto:crypto_final instead)
- `crypto:cmac/3` (use crypto:mac/4 instead)
- `crypto:cmac/4` (use crypto:macN/5 instead)
- `crypto:hmac/3` (use crypto:mac/4 instead)
- `crypto:hmac/4` (use crypto:macN/5 instead)
- `crypto:hmac_final/1` (use crypto:mac_final/1 instead)
- `crypto:hmac_final_n/2` (use crypto:mac_finalN/2 instead)
- `crypto:hmac_init/2` (use crypto:mac_init/3 instead)

- `crypto:hmac_update/2` (use crypto:mac_update/2 instead)
- `crypto:next_iv/_` (see the 'New and Old API' chapter of the CRYPTO User's guide)
- `crypto:poly1305/2` (use crypto:mac/3 instead)
- `crypto:stream_decrypt/2` (use crypto:crypto_update/2 instead)
- `crypto:stream_encrypt/2` (use crypto:crypto_update/2 instead)
- `crypto:stream_init/_` (use crypto:crypto_init/3 + crypto:crypto_update/2 + crypto:crypto_final/1 or crypto:crypto_one_time/4 instead)
- `filename:find_src/_` (use filelib:find_source/1,3 instead)
- `pg2:_/_` (this module was removed in OTP 24. Use 'pg' instead)
- `ssl:cipher_suites/0` (use cipher_suites/2,3 instead)
- `ssl:cipher_suites/1` (use cipher_suites/2,3 instead)
- `ssl:ssl_accept/_` (use ssl_handshake/1,2,3 instead)

1.2.4 OTP 23

VxWorks Support

Some parts of OTP has had limited VxWorks support, such as `erl_interface`. This support was removed in OTP 23. This limited support was formally deprecated as of OTP 22.

Legacy parts of erl_interface

The old legacy `erl_interface` library (functions with prefix `erl_`) was removed in OTP 23. These parts of `erl_interface` has been informally deprecated for a very long time, and was formally deprecated in OTP 22. You typically want to replace the usage of the `erl_interface` library with the use of the `ei` library which also is part of the `erl_interface` application.

httpd_conf module

API functions in the module called `httpd_conf` was deprecated in favor of standard modules such as `lists`, `string`, `filelib`, and `erlang`. Formally deprecated as of OTP 18.

inets - httpd Apache config files

Support for the Apache-compatible config files was removed in OTP 23. A new config file format was introduced in OTP 12.

SSL/TLS

For security reasons SSL-3.0 is no longer supported at all.

Functions Removed in OTP 23

- `erlang:get_stacktrace/0` (use the new try/catch syntax for retrieving the stack backtrace)
- `httpd_conf:check_enum/2` (use lists:member/2 instead)
- `httpd_conf:clean/1` (use string:strip/1 instead or possibly the re module)
- `httpd_conf:custom_clean/3` (use string:strip/1 instead or possibly the re module)
- `httpd_conf:is_directory/1` (use filelib:is_dir/1 instead)
- `httpd_conf:is_file/1` (use filelib:is_file/1 instead)
- `httpd_conf:make_integer/1` (use erlang:list_to_integer/1 instead)

1.2.5 OTP 22

Functions Removed in OTP 22

- `os_mon_mib:_/_` (this module was removed in OTP 22.0)

1.2.6 OTP 20

Functions Removed in OTP 20

- `asn1ct:decode/_` (use Mod:decode/2 instead)
- `asn1ct:encode/_` (use Mod:encode/2 instead)
- `erlang:hash/2` (use erlang:phash2/2 instead)
- `ssl:connection_info/1` (use ssl:connection_information/[1,2] instead)
- `ssl:negotiated_next_protocol/1` (use ssl:negotiated_protocol/1 instead)

1.2.7 OTP 19

Functions Removed in OTP 19

- `core_lib:get_anno/1` (use cerl:get_ann/1 instead)
- `core_lib:is_literal/1` (use cerl:is_literal/1 instead)
- `core_lib:is_literal_list/1` (use cerl:is_literal_list/1 instead)
- `core_lib:literal_value/1` (use cerl:concrete/1 instead)
- `core_lib:set_anno/2` (use cerl:set_ann/2 instead)
- `erl_lint:modify_line/2` (use erl_parse:map_anno/2 instead)
- `erl_parse:get_attribute/2` (erl_anno:{column,line,location,text}/1 instead)
- `erl_parse:get_attributes/1` (erl_anno:{column,line,location,text}/1 instead)
- `erl_parse:set_line/2` (use erl_anno:set_line/2)
- `erl_scan:attributes_info/_` (use erl_anno:{column,line,location,text}/1 instead)
- `erl_scan:set_attribute/3` (use erl_anno:set_line/2 instead)
- `erl_scan:token_info/_` (use erl_scan:{category,column,line,location,symbol,text}/1 instead)
- `rpc:safe_multi_server_call/2` (use rpc:multi_server_call/2 instead)
- `rpc:safe_multi_server_call/3` (use rpc:multi_server_call/3 instead)

1.3 Scheduled for Removal

1.3.1 Introduction

This document list all functionality in Erlang/OTP that currently are scheduled for removal. For more information regarding the strategy regarding removal of functionality see the documentation of Support, Compatibility, Deprecations, and Removal.

1.3.2 OTP 27

Functions Scheduled for Removal in OTP 27

- `crypto:crypto_dyn_iv_init/3` (see the documentation for details)
- `crypto:crypto_dyn_iv_update/3` (see the documentation for details)
- `ct_slave:_/_` (use ?CT_PEER(), or the 'peer' module instead)

- `slave:_/_` (use the 'peer' module instead)
- `zlib:adler32/2` (use erlang:adler32/1 instead)
- `zlib:adler32/3` (use erlang:adler32/2 instead)
- `zlib:adler32_combine/4` (use erlang:adler_combine/3 instead)
- `zlib:crc32/1` (use erlang:crc32/1 on the uncompressed data instead)
- `zlib:crc32/2` (use erlang:crc32/1 instead)
- `zlib:crc32/3` (use erlang:crc32/2 instead)
- `zlib:crc32_combine/4` (use erlang:crc32_combine/3 instead)
- `zlib:getBufSize/1` (this function will be removed in a future release)
- `zlib:inflateChunk/1` (use safeInflate/2 instead)
- `zlib:inflateChunk/2` (use safeInflate/2 instead)
- `zlib:setBufSize/2` (this function will be removed in a future release)

1.3.3 OTP 26

Erlang Distribution Without Large Node Container Support

Communication over the Erlang distribution without support for large node container data types (version 4) is as of OTP 24 deprecated and support for it is scheduled for removal in OTP 26. That is, as of OTP 26, support for large node container data types will become mandatory.

Old Link Protocol

The old link protocol used when communicating over the Erlang distribution is as of OTP 24 deprecated and support for it is scheduled for removal in OTP 26. As of OTP 26 the new link protocol will become mandatory. That is, Erlang nodes will then refuse to connect to nodes not implementing the new link protocol. If you implement the Erlang distribution yourself, you are, however, encouraged to implement the new link protocol as soon as possible since the old protocol can cause links to enter an inconsistent state.

Functions Scheduled for Removal in OTP 26

- `code:is_module_native/1` (HiPE has been removed)
- `code:rehash/0` (the code path cache feature has been removed)
- `disk_log:accessible_logs/0` (use disk_log:all/0 instead)
- `disk_log:lclose/1` (use disk_log:close/1 instead)
- `disk_log:lclose/2` (use disk_log:close/1 instead)
- `erts_alloc_config:_/_` (this module will be removed in OTP 26.0. See the documentation for details)
- `ftp:start_service/1` (use ftp:open/2 instead)
- `ftp:stop_service/1` (use ftp:close/1 instead)
- `http_uri:decode/1` (use uri_string:unquote function instead)
- `http_uri:encode/1` (use uri_string:quote function instead)
- `httpd_util:decode_hex/1` (use uri_string:unquote function instead)
- `httpd_util:encode_hex/1` (use uri_string:quote function instead)
- `httpd_util:flatlength/1` (use erlang:iolist_size/1 instead)
- `httpd_util:hexlist_to_integer/1` (use erlang:list_to_integer/2 with base 16 instead)
- `httpd_util:integer_to_hexlist/1` (use erlang:integer_to_list/2 with base 16 instead)
- `httpd_util:strip/1` (use string:trim/1 instead)
- `httpd_util:suffix/1` (use filename:extension/1 and string:trim/2 instead)

1.4 Upcoming Potential Incompatibilities

1.4.1 Introduction

This document lists planned upcoming potential incompatibilities in Erlang/OTP.

1.4.2 OTP 25

Distribution flags will become mandatory

In OTP 25, more distribution flags will become mandatory. That is, Erlang nodes will refuse to connect to nodes not implementing all of the mandatory distribution flags. If you implement the Erlang distribution protocol yourself, you will need to implement support for all mandatory distribution flags in order to communicate with Erlang nodes running OTP 25.

The following distribution flags will become mandatory in OTP 25:

`DFLAG_BIT_BINARIES`
: Support for bitstrings.

`DFLAG_EXPORT_PTR_TAG`
: Support for external funs (`fun Module:Name/Arity`).

`DFLAG_MAP_TAGS`
: Support for maps.

`DFLAG_NEW_FLOATS`
: Support for the new encoding of floats.

`DFLAG_FUN_TAGS`
: Support for funs, but only in the new format (`NEW_FUN_EXT`) because `DFLAG_NEW_FUN_TAGS` is also mandatory.

1.4.3 OTP 26

The distribution flag DFLAG_V4_NC will become mandatory

As of OTP 26, the distribution flag DFLAG_V4_NC will become mandatory. If you implement the Erlang distribution protocol yourself, you will need to implement support for `DFLAG_V4_NC` in order to communicate with Erlang nodes running OTP 26.

The new link protocol will become mandatory

As of OTP 26, the new link protocol will become mandatory. That is, Erlang nodes will then refuse to connect to nodes not implementing the new link protocol. If you implement the Erlang distribution yourself, you are, however, encouraged to implement the new link protocol as soon as possible since the old protocol can cause links to enter an inconsistent state.

Atoms will be encoded as UTF-8 by default

As of OTP 26, the functions `erlang:term_to_binary/1,2` and `erlang:term_to_iovec/1,2` will encode all atoms as UTF-8 by default. The current default behavior is to encode atoms as Latin-1 if possible.

If you implement your own decoding of the Erlang external format you must either:

- Make sure your implementation supports the UTF-8 encodings `ATOM_UTF8_EXT` and `SMALL_ATOM_UTF8_EXT`.
- Call `erlang:term_to_binary/2` or `erlang:term_to_iovec/2` with option `{minor_version,1}` to force Latin-1 encoding. This is a more short-term solution as Latin-1 encoding may be phased out and removed in later OTP releases.

The default timewarp mode will change to multi-time warp mode

The default Time Warp Mode will be changed from no time warp mode to multi-time warp mode. See Time and Time Correction in Erlang for details on how this will effect your system.

2 Installation Guide

This section describes how to install Erlang/OTP on UNIX and Windows.

2.1 Installing the Binary Release

2.1.1 Windows

The system is delivered as a Windows Installer executable. Get it from http://www.erlang.org/download.html

Installing

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

Verifying

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

  Expect a command-line window to pop up with an output looking something like this:

  ```
  Erlang/OTP 17 [erts-6.0] [64-bit] [smp:2:2]

  Eshell V6.0  (abort with ^G)
  1>
  ```

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

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

  This closes the Erlang/OTP shell.

2.2 Building and Installing Erlang/OTP

2.2.1 Introduction

This document describes how to build and install Erlang/OTP-25. Erlang/OTP should be possible to build from source on any Unix/Linux system, including OS X. You are advised to read the whole document before attempting to build and install Erlang/OTP.

The source code can be downloaded from the official site of Erlang/OTP or GitHub.

- **http://www.erlang.org**
- **https://github.com/erlang/otp**

2.2.2 Required Utilities

These are the tools you need in order to unpack and build Erlang/OTP.

Unpacking

- GNU unzip, or a modern uncompress.
- A TAR program that understands the GNU TAR format for long filenames.

Building

- GNU `make`
- Compiler -- GNU C Compiler, `gcc` or the C compiler frontend for LLVM, `clang`.
- Perl 5
- `ncurses`, `termcap`, or `termlib` -- The development headers and libraries are needed, often known as `ncurses-devel`. Use `--without-termcap` to build without any of these libraries. Note that in this case only the old shell (without any line editing) can be used.
- `sed` -- Stream Editor for basic text transformation.

Building in Git

Build the same way as when building the unpacked tar file.

Building on OS X

- Xcode -- Download and install via the Mac App Store. Read about Building on a Mac before proceeding.

Installing

- An `install` program that can take multiple file names.

2.2.3 Optional Utilities

Some applications are automatically skipped if the dependencies aren't met. Here is a list of utilities needed for those applications. You will also find the utilities needed for building the documentation.

Building

- OpenSSL -- The opensource toolkit for Secure Socket Layer and Transport Layer Security. Required for building the application `crypto`. Further, `ssl` and `ssh` require a working crypto application and will also be skipped if OpenSSL is missing. The `public_key` application is available without `crypto`, but the functionality will be very limited.

  The development package of OpenSSL including the header files are needed as well as the binary command program `openssl`. At least version 0.9.8 of OpenSSL is required. Read more and download from **http://www.openssl.org**.
- Oracle Java SE JDK -- The Java Development Kit (Standard Edition). Required for building the application `jinterface`. At least version 1.6.0 of the JDK is required.

  Download from **http://www.oracle.com/technetwork/java/javase/downloads**. We have also tested with IBM's JDK 1.6.0.
- `flex` -- Headers and libraries are needed to build the flex scanner for the `megaco` application on Unix/Linux.
- wxWidgets -- Toolkit for GUI applications. Required for building the `wx` application. At least version 3.0 of wxWidgets is required.

  Download from **http://sourceforge.net/projects/wxwindows/files/3.0.0/** or get it from GitHub: **https://github.com/wxWidgets/wxWidgets**

  Further instructions on wxWidgets, read Building with wxErlang.

Building Documentation

- `xsltproc` -- A command line XSLT processor.

  A tool for applying XSLT stylesheets to XML documents. Download xsltproc from **http://xmlsoft.org/XSLT/xsltproc2.html**.
- `fop` -- Apache FOP print formatter (requires Java). Can be downloaded from **http://xmlgraphics.apache.org/fop**.

2.2.4 How to Build and Install Erlang/OTP

The following instructions are for building **the released source tar ball**.

The variable `$ERL_TOP` will be mentioned a lot of times. It refers to the top directory in the source tree. More information about `$ERL_TOP` can be found in the make and $ERL_TOP section below.

Unpacking

Start by unpacking the Erlang/OTP distribution file with your GNU compatible TAR program.

```
$ tar -zxf otp_src_25.2.1.tar.gz    # Assuming bash/sh
```

Now change directory into the base directory and set the `$ERL_TOP` variable.

```
$ cd otp_src_25.2.1
$ export ERL_TOP=`pwd`    # Assuming bash/sh
```

Configuring

Run the following commands to configure the build:

```
$ ./configure [ options ]
```

By default, Erlang/OTP release will be installed in `/usr/local/{bin,lib/erlang}`. If you for instance don't have the permission to install in the standard location, you can install Erlang/OTP somewhere else. For example, to install in `/opt/erlang/25.2.1/{bin,lib/erlang}`, use the `--prefix=/opt/erlang/25.2.1` option.

On some platforms Perl may behave strangely if certain locales are set. If you get errors when building, try setting the LANG variable:

```
$ export LANG=C   # Assuming bash/sh
```

Building

Build the Erlang/OTP release.

```
$ make
```

Testing

Before installation you should test whether your build is working properly by running our smoke test. The smoke test is a subset of the complete Erlang/OTP test suites. First you will need to build and release the test suites.

```
$ make release_tests
```

This creates an additional folder in `$ERL_TOP/release` called `tests`. Now, it's time to start the smoke test.

```
$ cd release/tests/test_server
$ $ERL_TOP/bin/erl -s ts install -s ts smoke_test batch -s init stop
```

To verify that everything is ok you should open `$ERL_TOP/release/tests/test_server/index.html` in your web browser and make sure that there are zero failed test cases.

Note:

On builds without `crypto`, `ssl` and `ssh` there is a failed test case for undefined functions. Verify that the failed test case log only shows calls to skipped applications.

Installing

You are now ready to install the Erlang/OTP release! The following command will install the release on your system.

```
$ make install
```

Running

You should now have a working release of Erlang/OTP! Jump to System Principles for instructions on running Erlang/OTP.

How to Build the Documentation

Make sure you're in the top directory in the source tree.

```
$ cd $ERL_TOP
```

If you have just built Erlang/OTP in the current source tree, you have already ran `configure` and do not need to do this again; otherwise, run `configure`.

```
$ ./configure [Configure Args]
```

When building the documentation you need a full Erlang/OTP-25.2.1 system in the `$PATH`.

```
$ export PATH=$ERL_TOP/bin:$PATH     # Assuming bash/sh
```

For the FOP print formatter, two steps must be taken:

- Adding the location of your installation of `fop` in `$FOP_HOME`.

  ```
  $ export FOP_HOME=/path/to/fop/dir # Assuming bash/sh
  ```

- Adding the `fop` script (in `$FOP_HOME`) to your `$PATH`, either by adding `$FOP_HOME` to `$PATH`, or by copying the `fop` script to a directory already in your `$PATH`.

Build the documentation.

```
$ make docs
```

It is possible to limit which types of documentation is build by passing the `DOC_TARGETS` environment variable to `make docs`. The currently available types are: `html`, `pdf`, `man` and `chunks`. Example:

```
$ make docs DOC_TARGETS=chunks
```

Build Issues

We have sometimes experienced problems with Oracle's `java` running out of memory when running `fop`. Increasing the amount of memory available as follows has in our case solved the problem.

```
$ export FOP_OPTS="-Xmx<Installed amount of RAM in MB>m"
```

More information can be found at

- **http://xmlgraphics.apache.org/fop/0.95/running.html#memory**.

How to Install the Documentation

The documentation can be installed either using the `install-docs` target, or using the `release_docs` target.

- If you have installed Erlang/OTP using the `install` target, install the documentation using the `install-docs` target. Install locations determined by `configure` will be used. `$DESTDIR` can be used the same way as when doing `make install`.

```
$ make install-docs
```

- If you have installed Erlang/OTP using the `release` target, install the documentation using the `release_docs` target. You typically want to use the same `RELEASE_ROOT` as when invoking `make release`.

```
$ make release_docs RELEASE_ROOT=<release dir>
```

It is possible to limit which types of documentation is released using the same `DOC_TARGETS` environment variable as when building documentation.

Accessing the Documentation

After installation you can access the documentation by

- Reading man pages. Make sure that `erl` is referring to the installed version. For example `/usr/local/bin/erl`. Try viewing at the man page for Mnesia

```
$ erl -man mnesia
```

- Browsing the html pages by loading the page `/usr/local/lib/erlang/doc/erlang/index.html` or `<BaseDir>/lib/erlang/doc/erlang/index.html` if the prefix option has been used.
- Read the embedded documentation by using the built-in shell functions `h/1,2,3` or `ht/1,2,3`.

How to Install the Pre-formatted Documentation

Pre-formatted **html documentation** and **man pages** can be downloaded from

- **http://www.erlang.org/download.html**.

Extract the html archive in the installation directory.

```
$ cd <ReleaseDir>
$ tar -zxf otp_html_25.2.1.tar.gz
```

For `erl -man <page>` to work the Unix manual pages have to be installed in the same way, i.e.

```
$ cd <ReleaseDir>
$ tar -zxf otp_man_25.2.1.tar.gz
```

Where `<ReleaseDir>` is

- `<PrefixDir>/lib/erlang` if you have installed Erlang/OTP using `make install`.
- `$DESTDIR<PrefixDir>/lib/erlang` if you have installed Erlang/OTP using `make install DESTDIR=<TmpInstallDir>`.
- `RELEASE_ROOT` if you have installed using `make release RELEASE_ROOT=<ReleaseDir>`.

2.2.5 Advanced configuration and build of Erlang/OTP

If you want to tailor your Erlang/OTP build and installation, please read on for detailed information about the individual steps.

make and $ERL_TOP

All the makefiles in the entire directory tree use the environment variable `ERL_TOP` to find the absolute path of the installation. The `configure` script will figure this out and set it in the top level Makefile (which, when building, it will pass on). However, when developing it is sometimes convenient to be able to run make in a subdirectory. To do this you must set the `ERL_TOP` variable before you run make.

For example, assume your GNU make program is called `make` and you want to rebuild the application `STDLIB`, then you could do:

```
$ cd lib/stdlib; env ERL_TOP=<Dir> make
```

where `<Dir>` would be what you find `ERL_TOP` is set to in the top level Makefile.

otp_build vs configure/make

Building Erlang/OTP can be done either by using the `$ERL_TOP/otp_build` script, or by invoking `$ERL_TOP/configure` and `make` directly. Building using `otp_build` is easier since it involves fewer steps, but the `otp_build` build procedure is not as flexible as the `configure`/`make` build procedure. The binary releases for Windows that we deliver are built using `otp_build`.

Configuring

The configure script is created by the GNU autoconf utility, which checks for system specific features and then creates a number of makefiles.

The configure script allows you to customize a number of parameters; type `./configure --help` or `./configure --help=recursive` for details. `./configure --help=recursive` will give help for all `configure` scripts in all applications.

One of the things you can specify is where Erlang/OTP should be installed. By default Erlang/OTP will be installed in `/usr/local/{bin,lib/erlang}`. To keep the same structure but install in a different place, `<Dir>` say, use the `--prefix` argument like this: `./configure --prefix=<Dir>`.

Some of the available `configure` options are:

- `--prefix=PATH` - Specify installation prefix.
- `--disable-parallel-configure` - Disable parallel execution of `configure` scripts (parallel execution is enabled by default)
- `--{enable,disable}-jit` - Force enabling or disabling of the JIT.
- `--{enable,disable}-kernel-poll` - Kernel poll support (enabled by default if possible)
- `--enable-m64-build` - Build 64-bit binaries using the `-m64` flag to `(g)cc`
- `--enable-m32-build` - Build 32-bit binaries using the `-m32` flag to `(g)cc`
- `--{enable,disable}-pie` - Build position independent executable binaries.
- `--with-assumed-cache-line-size=SIZE` - Set assumed cache-line size in bytes. Default is 64. Valid values are powers of two between and including 16 and 8192. The runtime system use this value in order to try to avoid false sharing. A too large value wastes memory. A to small value will increase the amount of false sharing.
- `--{with,without}-termcap` - termcap (without implies that only the old Erlang shell can be used)
- `--with-javac=JAVAC` - Specify Java compiler to use
- `--{with,without}-javac` - Java compiler (without implies that the `jinterface` application won't be built)
- `--{enable,disable}-builtin-zlib` - Use the built-in source for zlib.
- `--{enable,disable}-dynamic-ssl-lib` - Enable or disable dynamic OpenSSL libraries when linking the crypto NIF. By default dynamic linking is done unless it does not work or is if it is a Windows system.
- `--{with,without}-ssl` - OpenSSL (without implies that the `crypto`, `ssh`, and `ssl` won't be built)
- `--with-ssl=PATH` - Specify base location of OpenSSL include and lib directories.
- `--with-ssl-incl=PATH` - Specify base location of OpenSSL `include` directory (if different than base location specified by --with-ssl=PATH).
- `--with-ssl-zlib=PATH` - Path to static zlib library to link the crypto NIF with. This zlib library is most often not necessary but might be needed in order to link the NIF in some cases.

- `--with-ssl-lib-subdir=RELATIVE_PATH` - Specify extra OpenSSL lib sub-directory to search in (relative to base directory).
- `--with-ssl-rpath=yes|no|PATHS` - Runtime library path for OpenSSL. Default is `yes`, which equates to a number of standard locations. If `no`, then no runtime library paths will be used. Anything else should be a comma or colon separated list of paths.
- `--with-libatomic_ops=PATH` - Use the `libatomic_ops` library for atomic memory accesses. If `configure` should inform you about no native atomic implementation available, you typically want to try using the `libatomic_ops` library. It can be downloaded from **https://github.com/ivmai/libatomic_ops/**.
- `--disable-smp-require-native-atomics` - By default `configure` will fail if an SMP runtime system is about to be built, and no implementation for native atomic memory accesses can be found. If this happens, you are encouraged to find a native atomic implementation that can be used, e.g., using `libatomic_ops`, but by passing `--disable-smp-require-native-atomics` you can build using a fallback implementation based on mutexes or spinlocks. Performance of the SMP runtime system will however suffer immensely without an implementation for native atomic memory accesses.
- `--enable-static-{nifs,drivers}` - To allow usage of nifs and drivers on OSs that do not support dynamic linking of libraries it is possible to statically link nifs and drivers with the main Erlang VM binary. This is done by passing a comma separated list to the archives that you want to statically link. e.g. `--enable-static-nifs=/home/$USER/my_nif.a`. The paths have to be absolute. For drivers, the driver name has to be the same as the filename. You also have to define `STATIC_ERLANG_NIF_LIBNAME` (see `erl_nif` documentation) or `STATIC_ERLANG_DRIVER` when compiling the .o files for the nif/driver. If your nif/driver depends on some other dynamic library, you now have to link that to the Erlang VM binary. This is easily achieved by passing `LIBS=-llibname` to configure.
- `--without-$app` - By default all applications in Erlang/OTP will be included in a release. If this is not wanted it is possible to specify that Erlang/OTP should be compiled without one or more applications, i.e. `--without-wx`. There is no automatic dependency handling between applications. If you disable an application that another application depends on, you also have to disable the dependent application.
- `--enable-gettimeofday-as-os-system-time` - Force usage of `gettimeofday()` for OS system time.
- `--enable-prefer-elapsed-monotonic-time-during-suspend` - Prefer an OS monotonic time source with elapsed time during suspend.
- `--disable-prefer-elapsed-monotonic-time-during-suspend` - Do not prefer an OS monotonic time source with elapsed time during suspend.
- `--with-clock-resolution=high|low` - Try to find clock sources for OS system time, and OS monotonic time with higher or lower resolution than chosen by default. Note that both alternatives may have a negative impact on the performance and scalability compared to the default clock sources chosen.
- `--disable-saved-compile-time` - Disable saving of compile date and time in the emulator binary.
- `--enable-ei-dynamic-lib` - Make erl_interface build a shared library in addition to the archive normally built.

If you or your system has special requirements please read the `Makefile` for additional configuration information.

Important Variables Inspected by configure

Compiler and Linker

- `CC` - C compiler.
- `CFLAGS` - C compiler flags. Defaults to "-g -O2". If you set it, these will be removed.
- `STATIC_CFLAGS` - Static C compiler flags.
- `CFLAG_RUNTIME_LIBRARY_PATH` - This flag should set runtime library search path for the shared libraries. Note that this actually is a linker flag, but it needs to be passed via the compiler.
- `CPP` - C pre-processor.

- `CPPFLAGS` - C pre-processor flags.
- `CXX` - C++ compiler.
- `CXXFLAGS` - C++ compiler flags.
- `LD` - Linker.
- `LDFLAGS` - Linker flags.
- `LIBS` - Libraries.

Dynamic Erlang Driver Linking

> **Note:**
>
> Either set all or none of the `DED_LD*` variables (with the exception of `DED_LDFLAGS_CONFTEST`).

- `DED_LD` - Linker for Dynamically loaded Erlang Drivers.
- `DED_LDFLAGS` - Linker flags to use with `DED_LD`.
- `DED_LDFLAGS_CONFTEST` - Linker flags to use with `DED_LD` in configure link tests if `DED_LDFLAGS` cannot be used in such tests. If not set, `DED_LDFLAGS` will be used in configure tests.
- `DED_LD_FLAG_RUNTIME_LIBRARY_PATH` - This flag should set runtime library search path for shared libraries when linking with `DED_LD`.

Large File Support

> **Note:**
>
> Either set all or none of the `LFS_*` variables.

- `LFS_CFLAGS` - Large file support C compiler flags.
- `LFS_LDFLAGS` - Large file support linker flags.
- `LFS_LIBS` - Large file support libraries.

Other Tools

- `RANLIB` - `ranlib` archive index tool.
- `AR` - `ar` archiving tool.
- `GETCONF` - `getconf` system configuration inspection tool. `getconf` is currently used for finding out large file support flags to use, and on Linux systems for finding out if we have an NPTL thread library or not.

Updating configure Scripts

Generated `configure` scripts are nowadays included in the git repository.

If you modify any `configure.in` files or the `erts/aclocal.m4` file, you need to regenerate `configure` scripts before the changes will take effect. First ensure that you have GNU `autoconf` of version 2.69 in your path. Then execute `./otp_build update_configure [--no-commit]` in the `$ERL_TOP` directory. The `otp_build` script will verify that `autoconf` is of correct version and will refuse to update the `configure` scripts if it is of any other version.

Atomic Memory Operations and the VM

The VM with SMP support makes quite a heavy use of atomic memory operations. An implementation providing native atomic memory operations is therefore very important when building Erlang/OTP. By default the VM will refuse to build if native atomic memory operations are not available.

Erlang/OTP itself provides implementations of native atomic memory operations that can be used when compiling with a `gcc` compatible compiler for 32/64-bit x86, 32/64-bit SPARC V9, 32-bit PowerPC, or 32-bit Tile. When compiling with a `gcc` compatible compiler for other architectures, the VM may be able to make use of native atomic

operations using the `__atomic_*` builtins (may be available when using a `gcc` of at least version 4.7) and/or using the `__sync_*` builtins (may be available when using a `gcc` of at least version 4.1). If only the `gcc`'s `__sync_*` builtins are available, the performance will suffer. Such a configuration should only be used as a last resort. When compiling on Windows using a MicroSoft Visual C++ compiler native atomic memory operations are provided by Windows APIs.

Native atomic implementation in the order preferred:

- The implementation provided by Erlang/OTP.
- The API provided by Windows.
- The implementation based on the `gcc` `__atomic_*` builtins.
- If none of the above are available for your architecture/compiler, you are recommended to build and install **libatomic_ops** before building Erlang/OTP. The `libatomic_ops` library provides native atomic memory operations for a variety of architectures and compilers. When building Erlang/OTP you need to inform the build system of where the `libatomic_ops` library is installed using the `--with-libatomic_ops=PATH` `configure` switch.
- As a last resort, the implementation solely based on the `gcc` `__sync_*` builtins. This will however cause lots of expensive and unnecessary memory barrier instructions to be issued. That is, performance will suffer. The `configure` script will warn at the end of its execution if it cannot find any other alternative than this.

Building

Building Erlang/OTP on a relatively fast computer takes approximately 5 minutes. To speed it up, you can utilize parallel make with the `-j<num_jobs>` option.

```
$ export MAKEFLAGS=-j8    # Assuming bash/sh
$ make
```

If you've upgraded the source with a patch you may need to clean up from previous builds before the new build. Make sure to read the Pre-built Source Release section below before doing a `make clean`.

Other useful information can be found at our GitHub wiki:

- **https://github.com/erlang/otp/wiki**

Within Git

Build the same way as when building the unpacked tar file.

OS X (Darwin)

Make sure that the command `hostname` returns a valid fully qualified host name (this is configured in `/etc/hostconfig`). Otherwise you might experience problems when running distributed systems.

If you develop linked-in drivers (shared library) you need to link using `gcc` and the flags `-bundle -flat_namespace -undefined suppress`. You also include `-fno-common` in `CFLAGS` when compiling. Use `.so` as the library suffix.

If you have Xcode 4.3, or later, you will also need to download "Command Line Tools" via the Downloads preference pane in Xcode.

Building with wxErlang

If you want to build the `wx` application, you will need to get wxWidgets-3.0 (`wxWidgets-3.0.3.tar.bz2` from **https://github.com/wxWidgets/wxWidgets/releases/download/v3.0.3/wxWidgets-3.0.3.tar.bz2**) or get it from github with bug fixes:

```
$ git clone --branch WX_3_0_BRANCH git@github.com:wxWidgets/wxWidgets.git
```

The wxWidgets-3.1 version should also work if 2.8 compatibility is enabled, add `--enable-compat28` to configure commands below.

Configure and build wxWidgets (shared library on linux):

```
$ ./configure --prefix=/usr/local
$ make && sudo make install
$ export PATH=/usr/local/bin:$PATH
```

Configure and build wxWidgets (static library on linux):

```
$ export CFLAGS=-fPIC
$ export CXXFLAGS=-fPIC
$ ./configure --prefix=/usr/local --disable-shared
$ make && sudo make install
$ export PATH=/usr/local/bin:$PATH
```

Configure and build wxWidgets (on Mavericks - 10.9):

```
$ ./configure --with-cocoa --prefix=/usr/local
or without support for old versions and with static libs
$ ./configure --with-cocoa --prefix=/usr/local --with-macosx-version-min=10.9 --disable-shared
$ make
$ sudo make install
$ export PATH=/usr/local/bin:$PATH
```

Check that you got the correct wx-config

```
$ which wx-config && wx-config --version-full
```

Build Erlang/OTP

```
$ export PATH=/usr/local/bin:$PATH
$ cd $ERL_TOP
$ ./configure
$ make
$ sudo make install
```

Pre-built Source Release

The source release is delivered with a lot of platform independent build results already pre-built. If you want to remove these pre-built files, invoke `./otp_build remove_prebuilt_files` from the `$ERL_TOP` directory. After you have done this, you can build exactly the same way as before, but the build process will take a much longer time.

Warning:

Doing `make clean` in an arbitrary directory of the source tree, may remove files needed for bootstrapping the build.

Doing `./otp_build save_bootstrap` from the `$ERL_TOP` directory before doing `make clean` will ensure that it will be possible to build after doing `make clean`. `./otp_build save_bootstrap` will be invoked automatically when `make` is invoked from `$ERL_TOP` with either the `clean` target, or the default target. It is also automatically invoked if `./otp_build remove_prebuilt_files` is invoked.

If you need to verify the bootstrap beam files match the provided source files, use `./otp_build update_primary` to create a new commit that contains differences, if any exist.

How to Build a Debug Enabled Erlang RunTime System

After completing all the normal building steps described above a debug enabled runtime system can be built. To do this you have to change directory to `$ERL_TOP/erts/emulator` and execute:

```
$ (cd $ERL_TOP/erts/emulator && make debug)
```

This will produce a beam.smp.debug executable. The file are installed along side with the normal (opt) version `beam.smp`.

To start the debug enabled runtime system execute:

```
$ $ERL_TOP/bin/cerl -debug
```

The debug enabled runtime system features lock violation checking, assert checking and various sanity checks to help a developer ensure correctness. Some of these features can be enabled on a normal beam using appropriate configure options.

There are other types of runtime systems that can be built as well using the similar steps just described.

```
$ (cd $ERL_TOP/erts/emulator && make $TYPE)
```

where `$TYPE` is `opt`, `gcov`, `gprof`, `debug`, `valgrind`, `asan` or `lcnt`. These different beam types are useful for debugging and profiling purposes.

Installing

- Staged install using **DESTDIR**. You can perform the install phase in a temporary directory and later move the installation into its correct location by use of the `DESTDIR` variable:

  ```
  $ make DESTDIR=<tmp install dir> install
  ```

  The installation will be created in a location prefixed by `$DESTDIR`. It can, however, not be run from there. It needs to be moved into the correct location before it can be run. If `DESTDIR` have not been set but `INSTALL_PREFIX` has been set, `DESTDIR` will be set to `INSTALL_PREFIX`. Note that `INSTALL_PREFIX` in pre R13B04 was buggy and behaved as `EXTRA_PREFIX` (see below). There are lots of areas of use for an installation procedure using `DESTDIR`, e.g. when creating a package, cross compiling, etc. Here is an example where the installation should be located under `/opt/local`:

  ```
  $ ./configure --prefix=/opt/local
  $ make
  $ make DESTDIR=/tmp/erlang-build install
  $ cd /tmp/erlang-build/opt/local
  $     # gnu-tar is used in this example
  $ tar -zcf /home/me/my-erlang-build.tgz *
  $ su -
  Password: *****
  $ cd /opt/local
  $ tar -zxf /home/me/my-erlang-build.tgz
  ```

- Install using the `release` target. Instead of doing `make install` you can create the installation in whatever directory you like using the `release` target and run the `Install` script yourself. `RELEASE_ROOT` is used for specifying the directory where the installation should be created. This is what by default ends up under `/usr/local/lib/erlang` if you do the install using `make install`. All installation paths provided in the `configure` phase are ignored, as well as `DESTDIR`, and `INSTALL_PREFIX`. If you want links from a specific `bin` directory to the installation you have to set those up yourself. An example where Erlang/OTP should be located at `/home/me/OTP`:

```
$ ./configure
$ make
$ make RELEASE_ROOT=/home/me/OTP release
$ cd /home/me/OTP
$ ./Install -minimal /home/me/OTP
$ mkdir -p /home/me/bin
$ cd /home/me/bin
$ ln -s /home/me/OTP/bin/erl erl
$ ln -s /home/me/OTP/bin/erlc erlc
$ ln -s /home/me/OTP/bin/escript escript
...
```

The `Install` script should currently be invoked as follows in the directory where it resides (the top directory):

```
$ ./Install [-cross] [-minimal|-sasl] <ERL_ROOT>
```

where:

- `-minimal` Creates an installation that starts up a minimal amount of applications, i.e., only `kernel` and `stdlib` are started. The minimal system is normally enough, and is what `make install` uses.
- `-sasl` Creates an installation that also starts up the `sasl` application.
- `-cross` For cross compilation. Informs the install script that it is run on the build machine.
- `<ERL_ROOT>` - The absolute path to the Erlang installation to use at run time. This is often the same as the current working directory, but does not have to be. It can follow any other path through the file system to the same directory.

If neither `-minimal`, nor `-sasl` is passed as argument you will be prompted.

- Test install using `EXTRA_PREFIX`. The content of the `EXTRA_PREFIX` variable will prefix all installation paths when doing `make install`. Note that `EXTRA_PREFIX` is similar to `DESTDIR`, but it does **not** have the same effect as `DESTDIR`. The installation can and have to be run from the location specified by `EXTRA_PREFIX`. That is, it can be useful if you want to try the system out, running test suites, etc, before doing the real install without `EXTRA_PREFIX`.

Symbolic Links in --bindir

When doing `make install` and the default installation prefix is used, relative symbolic links will be created from `/usr/local/bin` to all public Erlang/OTP executables in `/usr/local/lib/erlang/bin`. The installation phase will try to create relative symbolic links as long as `--bindir` and the Erlang bin directory, located under `--libdir`, both have `--exec-prefix` as prefix. Where `--exec-prefix` defaults to `--prefix`. `--prefix`, `--exec-prefix`, `--bindir`, and `--libdir` are all arguments that can be passed to `configure`. One can force relative, or absolute links by passing `BINDIR_SYMLINKS=relative|absolute` as arguments to `make` during the install phase. Note that such a request might cause a failure if the request cannot be satisfied.

2.2.6 Erlang/OTP test architectures

Erlang/OTP are currently tested on the following hardware and operating systems. This is not an exhaustive list, but we try to keep it as up to date as possible.

Architecture

- x86, x86-64
- Aarch32, Aarch64
- powerpc, powerpc64le

Operating system

- Fedora 31
- FreeBSD

- macOS 10.4 - 11.2
- MontaVista 4
- NetBSD
- OpenBSD
- SLES 10, 11, 12
- SunOS 5.11
- Ubuntu 10.04 - 20.04
- Windows 10, Windows Server 2019

2.3 Cross Compiling Erlang/OTP

Table of Contents

- Introduction
  - otp_build Versus configure/make
  - Cross Configuration
  - What can be Cross Compiled?
  - Compatibility
  - Patches
- Build and Install Procedure
  - Building With configure/make Directly
    - Building a Bootstrap System
    - Cross Building the System
    - Installing
      - Installing Using Paths Determined by configure
      - Installing Manually
  - Building With the otp_build Script
- Building and Installing the Documentation
- Testing the cross compiled system
- Currently Used Configuration Variables
  - Variables for otp_build Only
  - Cross Compiler and Other Tools
  - Cross System Root Locations
  - Optional Feature, and Bug Tests

2.3.1 Introduction

This document describes how to cross compile Erlang/OTP-25. You are advised to read the whole document before attempting to cross compile Erlang/OTP. However, before reading this document, you should read the $ERL_TOP/HOWTO/INSTALL.md document which describes building and installing Erlang/OTP in general. `$ERL_TOP` is the top directory in the source tree.

otp_build Versus configure/make

Building Erlang/OTP can be done either by using the `$ERL_TOP/otp_build` script, or by invoking `$ERL_TOP/configure` and `make` directly. Building using `otp_build` is easier since it involves fewer steps, but the `otp_build` build procedure is not as flexible as the `configure`/`make` build procedure. Note that `otp_build configure` will produce a default configuration that differs from what `configure` will produce by default. For

example, currently `--disable-dynamic-ssl-lib` is added to the `configure` command line arguments unless `--enable-dynamic-ssl-lib` has been explicitly passed. The binary releases that we deliver are built using `otp_build`. The defaults used by `otp_build configure` may change at any time without prior notice.

Cross Configuration

The `$ERL_TOP/xcomp/erl-xcomp.conf.template` file contains all available cross configuration variables and can be used as a template when creating a cross compilation configuration. All cross configuration variables are also listed at the end of this document. For examples of working cross configurations see the `$ERL_TOP/xcomp/erl-xcomp-TileraMDE2.0-tilepro.conf` file and the `$ERL_TOP/xcomp/erl-xcomp-x86_64-saf-linux-gnu.conf` file. If the default behavior of a variable is satisfactory, the variable does not need to be set. However, the `configure` script will issue a warning when a default value is used. When a variable has been set, no warning will be issued.

A cross configuration file can be passed to `otp_build configure` using the `--xcomp-conf` command line argument. Note that `configure` does not accept this command line argument. When using the `configure` script directly, pass the configuration variables as arguments to `configure` using a `<VARIABLE>=<VALUE>` syntax. Variables can also be passed as environment variables to `configure`. However, if you pass the configuration in the environment, make sure to unset all of these environment variables before invoking `make`; otherwise, the environment variables might set make variables in some applications, or parts of some applications, and you may end up with an erroneously configured build.

What can be Cross Compiled?

All Erlang/OTP applications except the `wx` application can be cross compiled. The build of the `wx` driver will currently be automatically disabled when cross compiling.

Compatibility

The build system, including cross compilation configuration variables used, may be subject to non backward compatible changes without prior notice. Current cross build system has been tested when cross compiling some Linux/GNU systems, but has only been partly tested for more esoteric platforms.

Patches

Please submit any patches for cross compiling in a way consistent with this system. All input is welcome as we have a very limited set of cross compiling environments to test with. If a new configuration variable is needed, add it to `$ERL_TOP/xcomp/erl-xcomp.conf.template`, and use it in `configure.in`. Other files that might need to be updated are:

- `$ERL_TOP/xcomp/erl-xcomp-vars.sh`
- `$ERL_TOP/erl-build-tool-vars.sh`
- `$ERL_TOP/erts/aclocal.m4`
- `$ERL_TOP/xcomp/README.md`
- `$ERL_TOP/xcomp/erl-xcomp-*.conf`

Note that this might be an incomplete list of files that need to be updated.

General information on how to submit patches can be found at: **http://wiki.github.com/erlang/otp/submitting-patches**

2.3.2 Build and Install Procedure

If you are building in Git, you want to read the Building in Git section of $ERL_TOP/HOWTO/INSTALL.md before proceeding.

We will first go through the `configure`/`make` build procedure which people probably are most familiar with.

Building With configure/make Directly

(1)

Change directory into the top directory of the Erlang/OTP source tree.

```
$ cd $ERL_TOP
```

In order to compile Erlang code, a small Erlang bootstrap system has to be built, or an Erlang/OTP system of the same release as the one being built has to be provided in the `$PATH`. The Erlang/OTP for the target system will be built using this Erlang system, together with the cross compilation tools provided.

If you want to build using a compatible Erlang/OTP system in the `$PATH`, jump to (3).

Building a Bootstrap System

(2)

```
$ ./configure --enable-bootstrap-only
$ make
```

The `--enable-bootstrap-only` argument to `configure` isn't strictly necessary, but will speed things up. It will only run `configure` in applications necessary for the bootstrap, and will disable a lot of things not needed by the bootstrap system. If you run `configure` without `--enable-boostrap-only` you also have to run make as `make bootstrap`; otherwise, the whole system will be built.

Cross Building the System

(3)

```
$ ./configure --host=<HOST> --build=<BUILD> [Other Config Args]
$ make
```

`<HOST>` is the host/target system that you build for. It does not have to be a full `CPU-VENDOR-OS` triplet, but can be. The full canonicalized `CPU-VENDOR-OS` triplet will be created by executing `$ERL_TOP/make/autoconf/config.sub <HOST>`. If `config.sub` fails, you need to be more specific.

`<BUILD>` should equal the `CPU-VENDOR-OS` triplet of the system that you build on. If you execute `$ERL_TOP/make/autoconf/config.guess`, it will in most cases print the triplet you want to use for this.

The use of `<HOST>` and `<BUILD>` values that differ will trigger cross compilation. Note that if `<HOST>` and `<BUILD>` differ, the canonicalized values of `<HOST>` and `<BUILD>` must also differ. If they do not, the configuration will fail.

Pass the cross compilation variables as command line arguments to `configure` using a `<VARIABLE>=<VALUE>` syntax.

Note:

You can **not** pass a configuration file using the `--xcomp-conf` argument when you invoke `configure` directly. The `--xcomp-conf` argument can only be passed to `otp_build configure`.

`make` will verify that the Erlang/OTP system used when building is of the same release as the system being built, and will fail if this is not the case. It is possible, however not recommended, to force the cross compilation even though the wrong Erlang/OTP system is used. This by invoking `make` like this: `make ERL_XCOMP_FORCE_DIFFERENT_OTP=yes`.

Warning:

Invoking `make ERL_XCOMP_FORCE_DIFFERENT_OTP=yes` might fail, silently produce suboptimal code, or silently produce erroneous code.

Installing

You can either install using the installation paths determined by `configure` (4), or install manually using (5).

Installing Using Paths Determined by configure

(4)

```
$ make install DESTDIR=<TEMPORARY_PREFIX>
```

`make install` will install at a location specified when doing `configure`. `configure` arguments specifying where the installation should reside are for example: `--prefix`, `--exec-prefix`, `--libdir`, `--bindir`, etc. By default it will install under `/usr/local`. You typically do not want to install your cross build under `/usr/local` on your build machine. Using **DESTDIR** will cause the installation paths to be prefixed by `$DESTDIR`. This makes it possible to install and package the installation on the build machine without having to place the installation in the same directory on the build machine as it should be executed from on the target machine.

When `make install` has finished, change directory into `$DESTDIR`, package the system, move it to the target machine, and unpack it. Note that the installation will only be working on the target machine at the location determined by `configure`.

Installing Manually

(5)

```
$ make release RELEASE_ROOT=<RELEASE_DIR>
```

`make release` will copy what you have built for the target machine to `<RELEASE_DIR>`. The `Install` script will not be run. The content of `<RELEASE_DIR>` is what by default ends up in `/usr/local/lib/erlang`.

The `Install` script used when installing Erlang/OTP requires common Unix tools such as `sed` to be present in your `$PATH`. If your target system does not have such tools, you need to run the `Install` script on your build machine before packaging Erlang/OTP. The `Install` script should currently be invoked as follows in the directory where it resides (the top directory):

```
$ ./Install [-cross] [-minimal|-sasl] <ERL_ROOT>
```

where:

- `-minimal` Creates an installation that starts up a minimal amount of applications, i.e., only `kernel` and `stdlib` are started. The minimal system is normally enough, and is what `make install` uses.
- `-sasl` Creates an installation that also starts up the `sasl` application.
- `-cross` For cross compilation. Informs the install script that it is run on the build machine.
- `<ERL_ROOT>` - The absolute path to the Erlang installation to use at run time. This is often the same as the current working directory, but does not have to be. It can follow any other path through the file system to the same directory.

If neither `-minimal`, nor `-sasl` is passed as argument you will be prompted.

You can now either do:

(6)

- Decide where the installation should be located on the target machine, run the `Install` script on the build machine, and package the installed installation. The installation just need to be unpacked at the right location on the target machine:

```
$ cd <RELEASE_DIR>
$ ./Install -cross [-minimal|-sasl] <ABSOLUTE_INSTALL_DIR_ON_TARGET>
```

or:

(7)

- Package the installation in `<RELEASE_DIR>`, place it wherever you want on your target machine, and run the `Install` script on your target machine:

```
$ cd <ABSOLUTE_INSTALL_DIR_ON_TARGET>
$ ./Install [-minimal|-sasl] <ABSOLUTE_INSTALL_DIR_ON_TARGET>
```

Building With the otp_build Script

(8)

```
$ cd $ERL_TOP
```

(9)

```
$ ./otp_build configure --xcomp-conf=<FILE> [Other Config Args]
```

alternatively:

```
$ ./otp_build configure --host=<HOST> --build=<BUILD> [Other Config Args]
```

If you have your cross compilation configuration in a file, pass it using the `--xcomp-conf=<FILE>` command line argument. If not, pass `--host=<HOST>`, `--build=<BUILD>`, and the configuration variables using a `<VARIABLE>=<VALUE>` syntax on the command line (same as in (3)). Note that `<HOST>` and `<BUILD>` have to be passed one way or the other; either by using `erl_xcomp_host=<HOST>` and `erl_xcomp_build=<BUILD>` in the configuration file, or by using the `--host=<HOST>`, and `--build=<BUILD>` command line arguments.

`otp_build configure` will configure both for the bootstrap system on the build machine and the cross host system.

(10)

```
$ ./otp_build boot -a
```

`otp_build boot -a` will first build a bootstrap system for the build machine and then do the cross build of the system.

(11)

```
$ ./otp_build release -a <RELEASE_DIR>
```

`otp_build release -a` will do the same as (5), and you will after this have to do a manual install either by doing (6), or (7).

2.3.3 Building and Installing the Documentation

After the system has been cross built you can build and install the documentation the same way as after a native build of the system. See the How to Build the Documentation section in the $ERL_TOP/HOWTO/INSTALL.md document for information on how to build the documentation.

2.3.4 Testing the cross compiled system

Some of the tests that come with erlang use native code to test. This means that when cross compiling erlang you also have to cross compile test suites in order to run tests on the target host. To do this you first have to release the tests as usual.

```
$ make release_tests
```

or

```
$ ./otp_build tests
```

The tests will be released into `$ERL_TOP/release/tests`. After releasing the tests you have to install the tests on the build machine. You supply the same xcomp file as to `./otp_build` in (9).

```
$ cd $ERL_TOP/release/tests/test_server/
$ $ERL_TOP/bootstrap/bin/erl -eval 'ts:install([{xcomp,"<FILE>"}])' -s ts compile_testcases -s init stop
```

You should get a lot of printouts as the testcases are compiled. Once done you should copy the entire `$ERL_TOP/release/tests` folder to the cross host system.

Then go to the cross host system and setup the erlang installed in (4) or (5) to be in your `$PATH`. Then go to what previously was `$ERL_TOP/release/tests/test_server` and issue the following command.

```
$ erl -s ts install -s ts run all_tests -s init stop
```

The configure should be skipped and all tests should hopefully pass. For more details about how to use ts run `erl -s ts help -s init stop`

2.3.5 Currently Used Configuration Variables

Note that you cannot define arbitrary variables in a cross compilation configuration file. Only the ones listed below will be guaranteed to be visible throughout the whole execution of all `configure` scripts. Other variables needs to be defined as arguments to `configure` or exported in the environment.

Variables for otp_build Only

Variables in this section are only used, when configuring Erlang/OTP for cross compilation using `$ERL_TOP/otp_build configure`.

Note:

These variables currently have **no** effect if you configure using the `configure` script directly.

- `erl_xcomp_build` - The build system used. This value will be passed as `--build=$erl_xcomp_build` argument to the `configure` script. It does not have to be a full `CPU-VENDOR-OS` triplet, but can be. The full `CPU-VENDOR-OS` triplet will be created by `$ERL_TOP/make/autoconf/config.sub $erl_xcomp_build`. If set to `guess`, the build system will be guessed using `$ERL_TOP/make/autoconf/config.guess`.
- `erl_xcomp_host` - Cross host/target system to build for. This value will be passed as `--host=$erl_xcomp_host` argument to the `configure` script. It does not have to be a full `CPU-VENDOR-OS` triplet, but can be. The full `CPU-VENDOR-OS` triplet will be created by `$ERL_TOP/make/autoconf/config.sub $erl_xcomp_host`.
- `erl_xcomp_configure_flags` - Extra configure flags to pass to the `configure` script.

Cross Compiler and Other Tools

If the cross compilation tools are prefixed by `<HOST>-` you probably do not need to set these variables (where `<HOST>` is what has been passed as `--host=<HOST>` argument to `configure`). Compiler and other tools can otherwise be identified via variables passed as arguments on the command line to `configure`, in then xcomp file, or as environment variables. For more information see the Important Variables Inspected by configure section of the $ERL_TOP/HOWTO/INSTALL.md document.

Cross System Root Locations

- `erl_xcomp_sysroot` - The absolute path to the system root of the cross compilation environment. Currently, the `crypto`, `odbc`, `ssh` and `ssl` applications need the system root. These applications will be skipped if the system root has not been set. The system root might be needed for other things too. If this is the case and the system root has not been set, `configure` will fail and request you to set it.
- `erl_xcomp_isysroot` - The absolute path to the system root for includes of the cross compilation environment. If not set, this value defaults to `$erl_xcomp_sysroot`, i.e., only set this value if the include system root path is not the same as the system root path.

Optional Feature, and Bug Tests

These tests cannot (always) be done automatically when cross compiling. You usually do not need to set these variables.

Warning:

Setting these variables wrong may cause hard to detect runtime errors. If you need to change these values, **really** make sure that the values are correct.

Note:

Some of these values will override results of tests performed by `configure`, and some will not be used until `configure` is sure that it cannot figure the result out.

The `configure` script will issue a warning when a default value is used. When a variable has been set, no warning will be issued.

- `erl_xcomp_after_morecore_hook` - `yes|no`. Defaults to `no`. If `yes`, the target system must have a working `__after_morecore_hook` that can be used for tracking used `malloc()` implementations core memory usage. This is currently only used by unsupported features.
- `erl_xcomp_bigendian` - `yes|no`. No default. If `yes`, the target system must be big endian. If `no`, little endian. This can often be automatically detected, but not always. If not automatically detected, `configure` will fail unless this variable is set. Since no default value is used, `configure` will try to figure this out automatically.
- `erl_xcomp_double_middle` - `yes|no`. Defaults to `no`. If `yes`, the target system must have doubles in "middle-endian" format. If `no`, it has "regular" endianness.
- `erl_xcomp_clock_gettime_cpu_time` - `yes|no`. Defaults to `no`. If `yes`, the target system must have a working `clock_gettime()` implementation that can be used for retrieving process CPU time.
- `erl_xcomp_getaddrinfo` - `yes|no`. Defaults to `no`. If `yes`, the target system must have a working `getaddrinfo()` implementation that can handle both IPv4 and IPv6.
- `erl_xcomp_gethrvtime_procfs_ioctl` - `yes|no`. Defaults to `no`. If `yes`, the target system must have a working `gethrvtime()` implementation and is used with procfs `ioctl()`.
- `erl_xcomp_dlsym_brk_wrappers` - `yes|no`. Defaults to `no`. If `yes`, the target system must have a working `dlsym(RTLD_NEXT, <S>)` implementation that can be used on `brk` and `sbrk` symbols used by the `malloc()` implementation in use, and by this track the `malloc()` implementations core memory usage. This is currently only used by unsupported features.

- `erl_xcomp_kqueue` - `yes|no`. Defaults to `no`. If `yes`, the target system must have a working `kqueue()` implementation that returns a file descriptor which can be used by `poll()` and/or `select()`. If `no` and the target system has not got `epoll()` or `/dev/poll`, the kernel-poll feature will be disabled.
- `erl_xcomp_linux_clock_gettime_correction` - `yes|no`. Defaults to `yes` on Linux; otherwise, `no`. If `yes`, `clock_gettime(CLOCK_MONOTONIC, _)` on the target system must work. This variable is recommended to be set to `no` on Linux systems with kernel versions less than 2.6.
- `erl_xcomp_linux_nptl` - `yes|no`. Defaults to `yes` on Linux; otherwise, `no`. If `yes`, the target system must have NPTL (Native POSIX Thread Library). Older Linux systems have LinuxThreads instead of NPTL (Linux kernel versions typically less than 2.6).
- `erl_xcomp_linux_usable_sigaltstack` - `yes|no`. Defaults to `yes` on Linux; otherwise, `no`. If `yes`, `sigaltstack()` must be usable on the target system. `sigaltstack()` on Linux kernel versions less than 2.4 are broken.
- `erl_xcomp_linux_usable_sigusrx` - `yes|no`. Defaults to `yes`. If `yes`, the `SIGUSR1` and `SIGUSR2` signals must be usable by the ERTS. Old LinuxThreads thread libraries (Linux kernel versions typically less than 2.2) used these signals and made them unusable by the ERTS.
- `erl_xcomp_poll` - `yes|no`. Defaults to `no` on Darwin/MacOSX; otherwise, `yes`. If `yes`, the target system must have a working `poll()` implementation that also can handle devices. If `no`, `select()` will be used instead of `poll()`.
- `erl_xcomp_putenv_copy` - `yes|no`. Defaults to `no`. If `yes`, the target system must have a `putenv()` implementation that stores a copy of the key/value pair.
- `erl_xcomp_reliable_fpe` - `yes|no`. Defaults to `no`. If `yes`, the target system must have reliable floating point exceptions.
- `erl_xcomp_posix_memalign` - `yes|no`. Defaults to `yes` if `posix_memalign` system call exists; otherwise `no`. If `yes`, the target system must have a `posix_memalign` implementation that accepts larger than page size alignment.
- `erl_xcomp_code_model_small` - `yes|no`. Default to `no`. If `yes`, the target system must place the beam.smp executable in the lower 2 GB of memory. That is it should not use position independent executable.

2.4 How to Build Erlang/OTP on Windows

Table of Contents

- Introduction
- Short Version
- Tools you Need and Their Environment
- The Shell Environment
- Building and Installing
- Development
- Frequently Asked Questions

2.4.1 Introduction

This section describes how to build the Erlang emulator and the OTP libraries on Windows. Note that the Windows binary releases are still a preferred alternative if one does not have Microsoft’s development tools and/or don’t want to install WSL.

The instructions apply to Windows 10 (v.1809 and later) supporting the WSL.1 (Windows Subsystem for Linux v.1) and using Ubuntu 18.04 release.

The procedure described uses WSL as a build environment. You run the bash shell in WSL and use the gnu configure/make etc to do the build. The emulator C-source code is, however, mostly compiled with Microsoft Visual C++™,

producing a native Windows binary. This is the same procedure as we use to build the pre-built binaries. Why we use VC++ and not gcc is explained further in the FAQ section.

These instructions apply for both 32-bit and 64-bit Windows. Note that even if you build a 64-bit version of Erlang, most of the directories and files involved are still named win32. Some occurrences of the name win64 are however present. The installation file for a 64-bit Windows version of Erlang, for example, is `otp_win64_25.exe`.

If you feel comfortable with the environment and build system, and have all the necessary tools, you have a great opportunity to make the Erlang/OTP distribution for Windows better. Please submit any suggestions or patches to our **git project** to let them find their way into the next version of Erlang. If making changes to the build system (like makefiles etc) please bear in mind that the same makefiles are used on Unix, so that your changes don't break other platforms. That of course goes for C-code too; system specific code resides in the `$ERL_TOP/erts/emulator/sys/win32` and `$ERL_TOP/erts/etc/win32` directories mostly. The `$ERL_TOP/erts/emulator/beam` directory is for common code.

2.4.2 Short Version

In the following sections, we've described as much as we could about the installation of the tools needed. Once the tools are installed, building is quite easy. We have also tried to make these instructions understandable for people with limited Unix experience. WSL is a whole new environment to some Windows users, why careful explanation of environment variables etc seemed to be in place.

This is the short story though, for the experienced and impatient:

- Get and install complete WSL environment
  - Install Visual Studio 2019
  - Get and install windows JDK-8
  - Get and install windows NSIS 3.05 or later (3.05 tried and working)
  - Get, build and install OpenSSL v1.1.1d or later (up to 1.1.1d tried & working) with static libs.
  - Get, build and install wxWidgets-3.1.3 or later (up to 3.1.3 tried & working) with static libs.
  - Get the Erlang source distribution (from **http://www.erlang.org/download.html**) and unpack with `tar` to the windows disk for example to: /mnt/c/src/
  - Install mingw-gcc, and make: `sudo apt install g++-mingw-w64 gcc-mingw-w64 make`
  - `$ cd UNPACK_DIR`
  - Modify PATH and other environment variables so that all these tools are runnable from a bash shell. Still standing in `$ERL_TOP`, issue the following commands (for 32-bit Windows, remove the x64 from the first row and change `otp_win64_25` to `otp_win32_25` on the last row):

```
$ eval `./otp_build env_win32 x64`
$ ./otp_build configure
$ ./otp_build boot -a
$ ./otp_build release -a
$ ./otp_build installer_win32
$ release/win32/otp_win64_25 /S
```

  Voila! `Start->Programs->Erlang OTP 25->Erlang` starts the Erlang Windows shell.

2.4.3 Tools you Need and Their Environment

You need some tools to be able to build Erlang/OTP on Windows. Most notably you'll need WSL (with ubuntu), Visual Studio and Microsofts Windows SDK, but you might also want a Java compiler, the NSIS install system, OpenSSL and wxWidgets. Well, here's some information about the different tools:

- WSL: Install WSL and Ubuntu in Windows 10 **https://docs.microsoft.com/en-us/windows/wsl/install-win10**

We have used and tested with WSL-1, WSL-2 was not available and may not be preferred when building Erlang/OTP since access to the windows disk is (currently) slower WSL-2.

- Visual Studio 2019 Download and run the installer from: **http://visualstudio.microsoft.com/downloads** Install C++ and SDK packages to the default installation directory.
- Java JDK 8 or later (optional) If you don't care about Java, you can skip this step. The result will be that jinterface is not built.

  Our Java code (jinterface, ic) is tested on windows with JDK 8. Get it for Windows and install it, the JRE is not enough.

  URL: **http://www.oracle.com/java/technologies/javase-downloads.html**

  Add javac to your path environment, in my case this means:

  ```
  `PATH="/mnt/c/Program\ Files/Java/jdk1.8.0_241/bin:$PATH`
  ```

  No `CLASSPATH` or anything is needed. Type `javac.exe` in the bash prompt and you should get a list of available Java options.
- Nullsoft NSIS installer system (optional) You need this to build the self installing package.

  Download and run the installer from: URL: **http://nsis.sourceforge.net/download**

  Add 'makensis.exe' to your path environment:

  ```
  `PATH="/mnt/c/Program\ Files/NSIS/Bin:$PATH`
  ```

  Type `which makensis.exe` in the bash prompt and you should get the path to the program.
- OpenSSL (optional) You need this to build crypto, ssh and ssl libs.

  We recommend v1.1.1d or later. There are prebuilt available binaries, which you can just download and install, available here: URL: **http://wiki.openssl.org/index.php/Binaries**

  Install into `C:/OpenSSL-Win64` (or `C:/OpenSSL-Win32`)
- wxWidgets (optional) You need this to build wx and use gui's in debugger and observer.

  We recommend v3.1.4 or later. Unpack into `c:/opt/local64/pgm/wxWidgets-3.1.4`

  If the `wxUSE_POSTSCRIPT` isn't enabled in `c:/opt/local64/pgm/wxWidgets-3.1.4/include/wx/msw/setup.h`, enable it.

  We recommend to enable for wxWebView wxUSE**WEBVIEW**EDGE.

  - Download the nuget package 'Microsoft.Web.WebView2' (Version 0.9.488 or newer)
  - Extract the package (it's a zip archive) to wxWidgets/3rdparty/webview2 (you should have 3rdparty/webview2/build/native/include/WebView2.h file after unpacking it)
  - Enable wxUSE**WEBVIEW**EDGE in `c:/opt/local64/pgm/wxWidgets-3.1.4/include/wx/msw/setup.h`

  Build with:

  ```
  C:\...\> cd c:\opt\local64\pgm\wxWidgets-3.1.4\build\msw
  C:\...\> nmake TARGET_CPU=amd64 BUILD=release SHARED=0 DIR_SUFFIX_CPU= -f makefile.vc
  ```

  Remove the `TARGET_CPU=amd64` for 32bit build.
- Get the Erlang source distribution (from **http://www.erlang.org/download.html**). The same as for Unix platforms. Preferably use tar to unpack the source tar.gz (`tar zxf otp_src_25.tar.gz`) to somewhere on the windows disk, `/mnt/c/path/to/otp_src`

  NOTE: It is important that source on the windows disk.

Set the environment ERL_TOP to point to the root directory of the source distribution. Let's say I stood in /mnt/c/src and unpacked otp_src_25.tar.gz, I then add the following to .profile:

```
ERL_TOP=/mnt/c/src/otp_src_25
export ERL_TOP
```

2.4.4 The Shell Environment

The path variable should now contain the windows paths to javac.exe and makensis.exe.

Setup the environment with:

```
$ export PATH
$ cd /mnt/c/path/to/otp_src/
$ eval `./otp_build env_win32 x64`
```

This should setup the additional environment variables.

This should do the final touch to the environment and building should be easy after this. You could run ./otp_build env_win32 without eval just to see what it does, and to see that the environment it sets seems OK. The path is cleaned of spaces if possible (using DOS style short names instead), the variables OVERRIDE_TARGET, CC, CXX, AR and RANLIB are set to their respective wrappers and the directories $ERL_TOP/erts/etc/win32/wsl_tools/vc and $ERL_TOP/erts/etc/win32/wsl_tools are added first in the PATH.

Now you can check which erlc you have by writing type erlc in your shell. It should reside in $ERL_TOP/erts/etc/win32/wsl_tools.

And running cl.exe should print the Microsoft compiler usage message.

The needed compiler environment variables are setup inside otp_build via erts/etc/win32/wsl_tools/SetupWSLcross.bat. It contains some hardcoded paths, if your installation path is different it can be added to that file.

2.4.5 Building and Installing

Building is easiest using the otp_build script:

```
$ ./otp_build configure <optional configure options>
$ ./otp_build boot -a
$ ./otp_build release -a <installation directory>
$ ./otp_build installer_win32 <installation directory> # optional
```

Now you will have a file called otp_win32_25.exe or otp_win64_25.exe in the <installation directory>, i.e. $ERL_TOP/release/win32.

Lets get into more detail:

- $ ./otp_build configure - This runs the newly generated configure scripts with options making configure behave nicely. The target machine type is plainly win32, so a lot of the configure-scripts recognize this awkward target name and behave accordingly. The CC variable also makes the compiler be cc.sh, which wraps MSVC++, so all configure tests regarding the C compiler gets to run the right compiler. A lot of the tests are not needed on Windows, but we thought it best to run the whole configure anyway.
- $ ./otp_build boot -a - This uses the bootstrap directory (shipped with the source, $ERL_TOP/bootstrap) to build a complete OTP system. When this is done you can run erl from within the source tree; just type $ERL_TOP/bin/erl and you should have the prompt.
- $ ./otp_build release -a - Builds a commercial release tree from the source tree. The default is to put it in $ERL_TOP/release/win32. You can give any directory as parameter, but it doesn't really matter if you're going to build a self extracting installer too.

- `$ ./otp_build installer_win32` - Creates the self extracting installer executable. The executable `otp_win32_25.exe` or `otp_win64_25.exe` will be placed in the top directory of the release created in the previous step. If no release directory is specified, the release is expected to have been built to `$ERL_TOP/release/win32`, which also will be the place where the installer executable will be placed. If you specified some other directory for the release (i.e. `./otp_build release -a /tmp/erl_release`), you're expected to give the same parameter here, (i.e. `./otp_build installer_win32 /tmp/erl_release`). You need to have a full NSIS installation and `makensis.exe` in your path for this to work. Once you have created the installer, you can run it to install Erlang/OTP in the regular way, just run the executable and follow the steps in the installation wizard. To get all default settings in the installation without any questions asked, you run the executable with the parameter `/S` (capital S) like in:

  ```
  $ cd $ERL_TOP
  $ release/win32/otp_win32_25 /S
  ...
  ```

  or

  ```
  $ cd $ERL_TOP
  $ release/win32/otp_win64_25 /S
  ...
  ```

  and after a while Erlang/OTP-25 will have been installed in `C:\Program Files\erl13.1.4\`, with shortcuts in the menu etc.

2.4.6 Development

Once the system is built, you might want to change it. Having a test release in some nice directory might be useful, but you can also run Erlang from within the source tree. The target `local_setup`, makes the program `$ERL_TOP/bin/erl.exe` usable and it also uses all the OTP libraries in the source tree.

If you hack the emulator, you can build the emulator executable by standing in `$ERL_TOP/erts/emulator` and do a simple

```
$ make opt
```

Note that you need to have run `(cd $ERL_TOP && eval `./otp_build env_win32`)` in the particular shell before building anything on Windows. After doing a make opt you can test your result by running `$ERL_TOP/bin/erl`. If you want to copy the result to a release directory (say `/tmp/erl_release`), you do this (still in `$ERL_TOP/erts/emulator`)

```
$ make TESTROOT=/tmp/erl_release release
```

That will copy the emulator executables.

To make a debug build of the emulator, you need to recompile both `beam.dll` (the actual runtime system) and `erlexec.dll`. Do like this

```
$ cd $ERL_TOP
$ rm bin/win32/erlexec.dll
$ cd erts/emulator
$ make debug
$ cd ../etc
$ make debug
```

and sometimes

```
$ cd $ERL_TOP
$ make local_setup
```

So now when you run `$ERL_TOP/erl.exe`, you should have a debug compiled emulator, which you will see if you do a:

```
1> erlang:system_info(system_version).
```

in the erlang shell. If the returned string contains `[debug]`, you got a debug compiled emulator.

To hack the erlang libraries, you simply do a `make opt` in the specific "applications" directory, like:

```
$ cd $ERL_TOP/lib/stdlib
$ make opt
```

or even in the source directory...

```
$ cd $ERL_TOP/lib/stdlib/src
$ make opt
```

Note that you're expected to have a fresh Erlang in your path when doing this, preferably the plain 25 you have built in the previous steps. You could also add `$ERL_TOP/bootstrap/bin` to your `PATH` before rebuilding specific libraries. That would give you a good enough Erlang system to compile any OTP erlang code. Setting up the path correctly is a little bit tricky. You still need to have `$ERL_TOP/erts/etc/win32/wsl_tools/vc` and `$ERL_TOP/erts/etc/win32/wsl_tools` **before** the actual emulator in the path. A typical setting of the path for using the bootstrap compiler would be:

```
$ export PATH=$ERL_TOP/erts/etc/win32/wsl_tools/vc\
:$ERL_TOP/erts/etc/win32/wsl_tools:$ERL_TOP/bootstrap/bin:$PATH
```

That should make it possible to rebuild any library without hassle...

If you want to copy a library (an application) newly built, to a release area, you do like with the emulator:

```
$ cd $ERL_TOP/lib/stdlib
$ make TESTROOT=/tmp/erlang_release release
```

Remember that:

- Windows specific C-code goes in the `$ERL_TOP/erts/emulator/sys/win32`, `$ERL_TOP/erts/emulator/drivers/win32` or `$ERL_TOP/erts/etc/win32`.
- Windows specific erlang code should be used conditionally and the host OS tested in **runtime**, the exactly same beam files should be distributed for every platform! So write code like:

  ```
  case os:type() of
      {win32,_} ->
          do_windows_specific();
      Other ->
          do_fallback_or_exit()
  end,
  ```

That's basically all you need to get going.

2.4.7 Frequently Asked Questions

- Q: So, now I can build Erlang using GCC on Windows?

  A: No, unfortunately not. You'll need Microsoft's Visual C++ still. A Bourne-shell script (cc.sh) wraps the Visual C++ compiler and runs it from within the WSL environment. All other tools needed to build Erlang are free-ware/open source, but not the C compiler.
- Q: Why haven't you got rid of VC++ then, you ******?

A: Well, partly because it's a good compiler - really! Actually it's been possible in late R11-releases to build using mingw instead of visual C++ (you might see the remnants of that in some scripts and directories). Unfortunately the development of the SMP version for Windows broke the mingw build and we chose to focus on the VC++ build as the performance has been much better in the VC++ versions. The mingw build will possibly be back, but as long as VC++ gives better performance, the commercial build will be a VC++ one.

- Q: Hah, I saw you, you used GCC even though you said you didn't!

  A: OK, I admit, one of the files is compiled using MinGW's GCC and the resulting object code is then converted to MS VC++ compatible coff using a small C hack. It's because that particular file, `beam_emu.c` benefits immensely from being able to use the GCC labels-as-values extension, which boosts emulator performance by up to 50%. That does unfortunately not (yet) mean that all of OTP could be compiled using GCC. That particular source code does not do anything system specific and actually is adopted to the fact that GCC is used to compile it on Windows.

- Q: So now there's a MS VC++ project file somewhere and I can build OTP using the nifty VC++ GUI?

  A: No, never. The hassle of keeping the project files up to date and do all the steps that constitute an OTP build from within the VC++ GUI is simply not worth it, maybe even impossible. A VC++ project file for Erlang/OTP will never happen.

- Q: So how does it all work then?

  A: WSL/Ubuntu is the environment, it's almost like you had a virtual Unix machine inside Windows. Configure, given certain parameters, then creates makefiles that are used by the environment's gnu-make to built the system. Most of the actual compilers etc are not, however, WSL tools, so we've written a couple of wrappers (Bourne-shell scripts), which reside in `$ERL_TOP/etc/win32/wsl_tools`. They all do conversion of parameters and switches common in the Unix environment to fit the native Windows tools. Most notable is of course the paths, which in WSL are Unix-like paths with "forward slashes" (/) and no drive letters. The WSL specific command `wslpath` is used for most of the path conversions in a WSL environment. Luckily most compilers accept forward slashes instead of backslashes as path separators, but one still have to get the drive letters etc right, though. The wrapper scripts are not general in the sense that, for example, cc.sh would understand and translate every possible gcc option and pass correct options to cl.exe. The principle is that the scripts are powerful enough to allow building of Erlang/OTP, no more, no less. They might need extensions to cope with changes during the development of Erlang, and that's one of the reasons we made them into shell-scripts and not Perl-scripts. We believe they are easier to understand and change that way.

  In `$ERL_TOP`, there is a script called `otp_build`. That script handles the hassle of giving all the right parameters to `configure`/`make` and also helps you set up the correct environment variables to work with the Erlang source under WSL.

- Q: Can I build something that looks exactly as the commercial release?

  A: Yes, we use the exact same build procedure.

- Q: Which version of WSL and other tools do you use then?

  A: We use WSL 1 with Ubuntu 18.04. The GCC we used for 25 was version 7.3-win32. We used Visual studio 2019, Sun's JDK 1.8.0_241, NSIS 3.05, Win32 OpenSSL 1.1.1d and wxWidgets-3.1.3.

2.5 Patching OTP Applications

2.5.1 Introduction

This document describes the process of patching an existing OTP installation with one or more Erlang/OTP applications of newer versions than already installed. The tool `otp_patch_apply` is available for this specific purpose. It resides in the top directory of the Erlang/OTP source tree.

The `otp_patch_apply` tool utilizes the runtime_dependencies tag in the application resource file. This information is used to determine if the patch can be installed in the given Erlang/OTP installation directory.

Read more about the version handling introduced in Erlang/OTP release 17, which also describes how to determine if an installation includes one or more patched applications.

If you want to apply patches of multiple OTP applications that resides in different OTP versions, you have to apply these patches in multiple steps. It is only possible to apply multiple OTP applications from the same OTP version at once.

2.5.2 Prerequisites

It's assumed that the reader is familiar with building and installing Erlang/OTP. To be able to patch an application, the following must exist:

- An Erlang/OTP installation.
- An Erlang/OTP source tree containing the updated applications that you want to patch into the existing Erlang/OTP installation.

2.5.3 Using otp_patch_apply

Warning:

Patching applications is a one-way process. Create a backup of your OTP installation directory before proceeding.

First of all, build the OTP source tree at `$ERL_TOP` containing the updated applications.

Note:

Before applying a patch you need to do a **full** build of OTP in the source directory.

Configure and build all applications in OTP:

```
$ configure
$ make
```

or

```
$ ./otp_build configure
$ ./otp_build boot -a
```

If you have installed documentation in the OTP installation, also build the documentation:

```
$ make docs
```

After the successful build it's time to patch. The source tree directory, the directory of the installation and the applications to patch are given as arguments to `otp_patch_apply`. The dependencies of each application are validated against the applications in the installation and the other applications given as arguments. If a dependency error is detected, the script will be aborted.

The `otp_patch_apply` syntax:

```
$ otp_patch_apply -s <Dir> -i <Dir> [-l <Dir>] [-c] [-f] [-h] \
      [-n] [-v] <App1> [... <AppN>]

-s <Dir>  -- OTP source directory that contains build results.
-i <Dir>  -- OTP installation directory to patch.
-l <Dir>  -- Alternative OTP source library directory path(s)
             containing build results of OTP applications.
             Multiple paths should be colon separated.
-c        -- Cleanup (remove) old versions of applications
             patched in the installation.
-f        -- Force patch of application(s) even though
             dependencies are not fulfilled (should only be
             considered in a test environment).
-h        -- Print help then exit.
-n        -- Do not install documentation.
-v        -- Print version then exit.
<AppX>    -- Application to patch.

Environment Variable:
  ERL_LIBS  -- Alternative OTP source library directory path(s)
               containing build results of OTP applications.
               Multiple paths should be colon separated.
```

Note:

The complete build environment is required while running `otp_patch_apply`.

Note:

All source directories identified by `-s` and `-l` should contain build results of OTP applications.

For example, if the user wants to install patched versions of `mnesia` and `ssl` built in `/home/me/git/otp` into the OTP installation located in `/opt/erlang/my_otp` type

```
$ otp_patch_apply -s /home/me/git/otp -i /opt/erlang/my_otp \
  mnesia ssl
```

Note:

If the list of applications contains core applications, i.e `erts`, `kernel`, `stdlib` or `sasl`, the `Install` script in the patched Erlang/OTP installation must be rerun.

The patched applications are appended to the list of installed applications. Take a look at `<InstallDir>/releases/OTP-REL/installed_application_versions`.

2.5.4 Sanity check

The application dependencies can be checked using the Erlang shell. Application dependencies are verified among installed applications by `otp_patch_apply`, but these are not necessarily those actually loaded. By calling `system_information:sanity_check()` one can validate dependencies among applications actually loaded.

```
1> system_information:sanity_check().
    ok
```

Please take a look at the reference of sanity_check() for more information.

3 System Principles

3.1 System Principles

3.1.1 Starting the System

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

```
% erl
Erlang/OTP 17 [erts-6.0] [hipe] [smp:8:8]

Eshell V6.0  (abort with ^G)
1>
```

`erl` understands a number of command-line arguments, see the erl(1) manual page in ERTS. Some of them are also described in this chapter.

Application programs can access the values of the command-line arguments by calling the function `init:get_argument(Key)` or `init:get_arguments()`. See the init(3) manual page in ERTS.

3.1.2 Restarting and Stopping the System

The runtime system is halted by calling `halt/0,1`. For details, see the erlang(3) manual page in ERTS.

The module `init` contains functions for restarting, rebooting, and stopping the runtime system:

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

For details, see the init(3) manual page in ERTS.

The runtime system terminates if the Erlang shell is terminated.

3.1.3 Boot Scripts

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

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

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

```
% erl -boot start_all
```

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

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

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

For a detailed description of the syntax and contents of the boot script, see the `script(4)` manual page in SASL.

Default Boot Scripts

Erlang/OTP comes with these boot scripts:

- `start_clean.boot` - Loads the code for and starts the applications Kernel and STDLIB.
- `start_sasl.boot` - Loads the code for and starts the applications Kernel, STDLIB, and SASL).
- `no_dot_erlang.boot` - Loads the code for and starts the applications Kernel and STDLIB. Skips loading the file `.erlang`. Useful for scripts and other tools that are to behave the same irrespective of user preferences.

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

User-Defined Boot Scripts

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

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

For more information about `.rel` files, see OTP Design Principles and the rel(4) manual page in SASL.

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

3.1.4 Code Loading Strategy

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

```
% erl -mode embedded
```

Default mode is `interactive` and extra `-mode` flags are ignored.

The mode properties are as follows:

- In embedded mode, all code is loaded during system startup according to the boot script. (Code can also be loaded later by explicitly ordering the code server to do so.)
- In interactive mode, the code is dynamically loaded when first referenced. When a call to a function in a module is made, and the module is not loaded, the code server searches the code path and loads the module into the system.

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

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

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

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

3.1.5 File Types

The following file types are defined in Erlang/OTP:

File Type	File Name/Extension	Documented in
Module	`.erl`	Erlang Reference Manual
Include file	`.hrl`	Erlang Reference Manual
Release resource file	`.rel`	rel(4) manual page in SASL
Application resource file	`.app`	app(4) manual page in Kernel
Boot script	`.script`	script(4) manual page in SASL
Binary boot script	`.boot`	-
Configuration file	`.config`	config(4) manual page in Kernel
Application upgrade file	`.appup`	appup(4) manual page in SASL
Release upgrade file	`relup`	relup(4) manual page in SASL

Table 1.1: File Types

3.2 Error Logging

3.2.1 Error Information From the Runtime System

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

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

The error information is handled by Logger, which is part of the Kernel application.

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

For information about Logger and its user interface, see the logger(3) manual page and the Logging section in the Kernel User's Guide. The system can be configured so that log events are written to file or to tty, or both. In addition, user-defined applications can send and format log events using Logger.

3.2.2 Log events from OTP behaviours

The standard behaviours (`supervisor`, `gen_server`, and so on) send progress and error information to Logger. Progress reports are by default not logged, but can be enabled by setting the primary log level to `info`, for example by using the Kernel configuration parameter `logger_level`. Supervisor reports, crash reports and other error and information reports are by default logged through the log handler which is set up when the Kernel application is started.

Prior to Erlang/OTP 21.0, supervisor, crash, and progress reports were only logged when the SASL application was running. This behaviour can, for backwards compatibility, be enabled by setting the Kernel configuration parameter `logger_sasl_compatible` to `true`. For more information, see SASL Error Logging in the SASL User's Guide.

```
% erl -kernel logger_level info
Erlang/OTP 21 [erts-10.0] [source-13c50db] [64-bit] [smp:4:4] [ds:4:4:10] [async-threads:1] [hipe]

=PROGRESS REPORT==== 8-Jun-2018::16:54:19.916404 ===
    application: kernel
    started_at: nonode@nohost
=PROGRESS REPORT==== 8-Jun-2018::16:54:19.922908 ===
    application: stdlib
    started_at: nonode@nohost
=PROGRESS REPORT==== 8-Jun-2018::16:54:19.925755 ===
    supervisor: {local,kernel_safe_sup}
    started: [{pid,<0.74.0>},
              {id,disk_log_sup},
              {mfargs,{disk_log_sup,start_link,[]}},
              {restart_type,permanent},
              {shutdown,1000},
              {child_type,supervisor}]
=PROGRESS REPORT==== 8-Jun-2018::16:54:19.926056 ===
    supervisor: {local,kernel_safe_sup}
    started: [{pid,<0.75.0>},
              {id,disk_log_server},
              {mfargs,{disk_log_server,start_link,[]}},
              {restart_type,permanent},
              {shutdown,2000},
              {child_type,worker}]
Eshell V10.0  (abort with ^G)
1>
```

3.3 Creating and Upgrading a Target System

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

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

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

The following sections deal with target systems with different requirements of functionality:

- A **basic target system** that can be started by calling the ordinary `erl` script.

- A **simple target system** where also code replacement in runtime can be performed.
- An **embedded target system** where there is also support for logging output from the system to file for later inspection, and where the system can be started automatically at boot time.

Here is only considered the case when Erlang/OTP is running on a UNIX system.

The `sasl` application includes the example Erlang module `target_system.erl`, which contains functions for creating and installing a target system. This module is used in the following examples. The source code of the module is listed in Listing of target_system.erl

3.3.1 Creating a Target System

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

Step 1. Create a `.rel` file (see the rel(4) manual page in SASL), which specifies the ERTS version and lists all applications that are to be included in the new basic target system. An example is the following `mysystem.rel` file:

```
%% mysystem.rel
{release,
 {"MYSYSTEM", "FIRST"},
 {erts, "5.10.4"},
 [{kernel, "2.16.4"},
  {stdlib, "1.19.4"},
  {sasl, "2.3.4"},
  {pea, "1.0"}]}.
```

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

Step 2. Start Erlang/OTP from the directory where the `mysystem.rel` file resides:

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

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

Step 3. Create the target system:

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

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

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

  ```
  erts-5.10.4/bin/
  releases/FIRST/start.boot
  releases/FIRST/mysystem.rel
  releases/mysystem.rel
  lib/kernel-2.16.4/
  lib/stdlib-1.19.4/
  lib/sasl-2.3.4/
  lib/pea-1.0/
  ```

  The file `releases/FIRST/start.boot` is a copy of our `mysystem.boot`

The release resource file `mysystem.rel` is duplicated in the tar file. Originally, this file was only stored in the `releases` directory to make it possible for the `release_handler` to extract this file separately. After unpacking the tar file, `release_handler` would automatically copy the file to `releases/FIRST`. However, sometimes the tar file is unpacked without involving the `release_handler` (for example, when unpacking the first target system). The file is therefore now instead duplicated in the tar file so no manual copying is needed.

- Creates the temporary directory `tmp` and extracts the tar file `mysystem.tar.gz` into that directory.
- Deletes the files `erl` and `start` from `tmp/erts-5.10.4/bin`. These files are created again from source when installing the release.
- Creates the directory `tmp/bin`.
- Copies the previously created file `plain.boot` to `tmp/bin/start.boot`.
- Copies the files `epmd`, `run_erl`, and `to_erl` from the directory `tmp/erts-5.10.4/bin` to the directory `tmp/bin`.
- Creates the directory `tmp/log`, which is used if the system is started as embedded with the `bin/start` script.
- Creates the file `tmp/releases/start_erl.data` with the contents "5.10.4 FIRST". This file is to be passed as data file to the `start_erl` script.
- Recreates the file `mysystem.tar.gz` from the directories in the directory `tmp` and removes `tmp`.

3.3.2 Installing a Target System

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

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

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

- Extracts the tar file `mysystem.tar.gz` into the target directory `/usr/local/erl-target`.
- In the target directory reads the file `releases/start_erl.data` to find the Erlang runtime system version ("5.10.4").
- Substitutes `%FINAL_ROOTDIR%` and `%EMU%` for `/usr/local/erl-target` and `beam`, respectively, in the files `erl.src`, `start.src`, and `start_erl.src` of the target `erts-5.10.4/bin` directory, and puts the resulting files `erl`, `start`, and `run_erl` in the target `bin` directory.
- Finally the target `releases/RELEASES` file is created from data in the file `releases/mysystem.rel`.

3.3.3 Starting a Target System

Now we have a target system that can be started in various ways. We start it as a **basic target system** by invoking:

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

Here only the Kernel and STDLIB applications are started, that is, the system is started as an ordinary development system. Only two files are needed for all this to work:

- `bin/erl` (obtained from `erts-5.10.4/bin/erl.src`)
- `bin/start.boot` (a copy of `plain.boot`)

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

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

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

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

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

The shell script `start`, which is generated from erts-5.10.4/bin/start.src during installation, is only an example. Edit it to suite your needs. Typically it is executed when the UNIX system boots.

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

`start_erl` requires:

- The root directory (`"/usr/local/erl-target"`)
- The releases directory (`"/usr/local/erl-target/releases"`
- The location of the file `start_erl.data`

It performs the following:

- Reads the runtime system version (`"5.10.4"`) and release version (`"FIRST"`) from the file `start_erl.data`.
- Starts the runtime system of the version found.
- Provides the flag `-boot` specifying the boot file of the release version found (`"releases/FIRST/start.boot"`).

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

The `start_erl` shell script is normally not to be altered by the user.

3.3.4 System Configuration Parameters

As was mentioned in the previous section, `start_erl` requires a `sys.config` in the release version directory (`"releases/FIRST/sys.config"`). If there is no such file, the system start fails. Such a file must therefore also be added.

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

However, it can also be convenient to replace variables in within a `sys.config` on the target after unpacking but before running the release. If you have a `sys.config.src` it will be included and is not required to be a valid Erlang term file like `sys.config`. Before running the release you must have a valid `sys.config` in the same directory, so using `sys.config.src` requires having some tool to populate what is needed and write `sys.config` to disk before booting the release.

3.3.5 Differences From the Install Script

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

3.3.6 Creating the Next Version

In this example the Pea application has been changed, and so are the applications ERTS, Kernel, STDLIB and SASL.

Step 1. Create the file `.rel`:

```
%% mysystem2.rel
{release,
 {"MYSYSTEM", "SECOND"},
 {erts, "6.0"},
 [{kernel, "3.0"},
  {stdlib, "2.0"},
  {sasl, "2.4"},
  {pea, "2.0"}]}.
```

Step 2. Create the application upgrade file (see the appup(4) manual page in SASL) for Pea, for example:

```
%% pea.appup
{"2.0",
 [{"1.0",[{load_module,pea_lib}]}],
 [{"1.0",[{load_module,pea_lib}]}]}.
```

Step 3. From the directory where the file `mysystem2.rel` resides, start the Erlang/OTP system, giving the path to the new version of Pea:

```
os> erl -pa /home/user/target_system/myapps/pea-2.0/ebin
```

Step 4. Create the release upgrade file (see the relup(4) manual page in SASL):

```
1> systools:make_relup("mysystem2",["mysystem"],["mysystem"],
    [{path,["/home/user/target_system/myapps/pea-1.0/ebin",
    "/my/old/erlang/lib/*/ebin"]}]).
```

Here `"mysystem"` is the base release and `"mysystem2"` is the release to upgrade to.

The `path` option is used for pointing out the old version of all applications. (The new versions are already in the code path - assuming of course that the Erlang node on which this is executed is running the correct version of Erlang/OTP.)

Step 5. Create the new release:

```
2> target_system:create("mysystem2").
```

Given that the file `relup` generated in Step 4 is now located in the current directory, it is automatically included in the release package.

3.3.7 Upgrading the Target System

This part is done on the target node, and for this example we want the node to be running as an embedded system with the `-heart` option, allowing automatic restart of the node. For more information, see Starting a Target System.

We add `-heart` to `bin/start`:

```
#!/bin/sh
ROOTDIR=/usr/local/erl-target/

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

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

$ROOTDIR/bin/run_erl -daemon /tmp/ $ROOTDIR/log "exec $ROOTDIR/bin/start_erl $ROOTDIR\
$RELDIR $START_ERL_DATA -heart"
```

We use the simplest possible `sys.config`, which we store in `releases/FIRST`:

```
%% sys.config
[].
```

Finally, to prepare the upgrade, we must put the new release package in the `releases` directory of the first target system:

```
os> cp mysystem2.tar.gz /usr/local/erl-target/releases
```

Assuming that the node has been started as follows:

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

It can be accessed as follows:

```
os> /usr/local/erl-target/bin/to_erl /tmp/erlang.pipe.1
```

Logs can be found in `/usr/local/erl-target/log`. This directory is specified as an argument to `run_erl` in the start script listed above.

Step 1. Unpack the release:

```
1> {ok,Vsn} = release_handler:unpack_release("mysystem2").
```

Step 2. Install the release:

```
2> release_handler:install_release(Vsn).
{continue_after_restart,"FIRST",[]}
heart: Tue Apr  1 12:15:10 2014: Erlang has closed.
heart: Tue Apr  1 12:15:11 2014: Executed "/usr/local/erl-target/bin/start /usr/local/erl-target/releases/ne
[End]
```

The above return value and output after the call to `release_handler:install_release/1` means that the `release_handler` has restarted the node by using `heart`. This is always done when the upgrade involves a change of the applications ERTS, Kernel, STDLIB, or SASL. For more information, see Upgrade when Erlang/OTP has Changed.

The node is accessible through a new pipe:

```
os> /usr/local/erl-target/bin/to_erl /tmp/erlang.pipe.2
```

Check which releases there are in the system:

```
1> release_handler:which_releases().
[{"MYSYSTEM","SECOND",
  ["kernel-3.0","stdlib-2.0","sasl-2.4","pea-2.0"],
  current},
 {"MYSYSTEM","FIRST",
  ["kernel-2.16.4","stdlib-1.19.4","sasl-2.3.4","pea-1.0"],
  permanent}]
```

Our new release, "SECOND", is now the current release, but we can also see that our "FIRST" release is still permanent. This means that if the node would be restarted now, it would come up running the "FIRST" release again.

Step 3. Make the new release permanent:

```
2> release_handler:make_permanent("SECOND").
```

Check the releases again:

```
3> release_handler:which_releases().
[{"MYSYSTEM","SECOND",
  ["kernel-3.0","stdlib-2.0","sasl-2.4","pea-2.0"],
  permanent},
 {"MYSYSTEM","FIRST",
  ["kernel-2.16.4","stdlib-1.19.4","sasl-2.3.4","pea-1.0"],
  old}]
```

We see that the new release version is `permanent`, so it would be safe to restart the node.

3.3.8 Listing of target_system.erl

This module can also be found in the `examples` directory of the SASL application.

```
-module(target_system).
-export([create/1, create/2, install/2]).

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

%% create(RelFileName)
%%
create(RelFileName) ->
    create(RelFileName,[]).

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

    io:fwrite("Making \"~ts.script\" and \"~ts.boot\" files ...~n",
       [PlainRelFileName,PlainRelFileName]),
    make_script(PlainRelFileName,SystoolsOpts),

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

    TarFileName = RelFileName ++ ".tar.gz",
    io:fwrite("Creating tar file ~ts ...~n", [TarFileName]),
    make_tar(RelFileName,SystoolsOpts),

    TmpDir = filename:join(Dir,"tmp"),
    io:fwrite("Creating directory ~tp ...~n",[TmpDir]),
    file:make_dir(TmpDir),

    io:fwrite("Extracting ~ts into directory ~ts ...~n", [TarFileName,TmpDir]),
    extract_tar(TarFileName, TmpDir),

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

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

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

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

    %% This is needed if 'start' script created from 'start.src' shall
    %% be used as it points out this directory as log dir for 'run_erl'
    TmpLogDir = filename:join([TmpDir, "log"]),
    io:fwrite("Creating temporary directory ~ts ...~n", [TmpLogDir]),
    ok = file:make_dir(TmpLogDir),

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

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


install(RelFileName, RootDir) ->
    TarFile = RelFileName ++ ".tar.gz",
    io:fwrite("Extracting ~ts ...~n", [TarFile]),
    extract_tar(TarFile, RootDir),
    StartErlDataFile = filename:join([RootDir, "releases", "start_erl.data"]),
    {ok, StartErlData} = read_txt_file(StartErlDataFile),
    [ErlVsn, _RelVsn| _] = string:tokens(StartErlData, " \n"),
    ErtsBinDir = filename:join([RootDir, "erts-" ++ ErlVsn, "bin"]),
    BinDir = filename:join([RootDir, "bin"]),
    io:fwrite("Substituting in erl.src, start.src and start_erl.src to "
              "form erl, start and start_erl ...\n"),
    subst_src_scripts(["erl", "start", "start_erl"], ErtsBinDir, BinDir,
                      [{"FINAL_ROOTDIR", RootDir}, {"EMU", "beam"}],
                      [preserve]),
    %%! Workaround for pre OTP 17.0: start.src and start_erl.src did
    %%! not have correct permissions, so the above 'preserve' option did not help
    ok = file:change_mode(filename:join(BinDir,"start"),8#0755),
    ok = file:change_mode(filename:join(BinDir,"start_erl"),8#0755),
```

```
    io:fwrite("Creating the RELEASES file ...\n"),
    create_RELEASES(RootDir, filename:join([RootDir, "releases",
         filename:basename(RelFileName)])).

%% LOCALS

%% make_script(RelFileName,Opts)
%%
make_script(RelFileName,Opts) ->
    systools:make_script(RelFileName, [no_module_tests,
           {outdir,filename:dirname(RelFileName)}
           |Opts]).

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

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

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

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

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

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

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

subst([$%, C| Rest], Vars, Result) when $A =< C, C =< $Z ->
    subst_var([C| Rest], Vars, Result, []);
subst([$%, C| Rest], Vars, Result) when $a =< C, C =< $z ->
```

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

subst_var([$%| Rest], Vars, Result, VarAcc) ->
    Key = lists:reverse(VarAcc),
    case lists:keysearch(Key, 1, Vars) of
        {value, {Key, Value}} ->
            subst(Rest, Vars, lists:reverse(Value, Result));
        false ->
            subst(Rest, Vars, [$%| VarAcc ++ [$%| Result]])
    end;
subst_var([C| Rest], Vars, Result, VarAcc) ->
    subst_var(Rest, Vars, Result, [C| VarAcc]);
subst_var([], Vars, Result, VarAcc) ->
    subst([], Vars, [VarAcc ++ [$%| Result]]).

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

copy_file(Src, Dest, Opts) ->
    {ok,_} = file:copy(Src, Dest),
    case lists:member(preserve, Opts) of
        true ->
            {ok, FileInfo} = file:read_file_info(Src),
            file:write_file_info(Dest, FileInfo);
        false ->
            ok
    end.

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

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

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

remove_all_files(Dir, Files) ->
    lists:foreach(fun(File) ->
                          FilePath = filename:join([Dir, File]),
                          case filelib:is_dir(FilePath) of
                              true ->
                                  {ok, DirFiles} = file:list_dir(FilePath),
                                  remove_all_files(FilePath, DirFiles),
                                  file:del_dir(FilePath);
                              _ ->
                                  file:delete(FilePath)
                          end
                  end, Files).
```

3.4 Upgrade when Erlang/OTP has Changed

3.4.1 Introduction

As of Erlang/OTP 17, most applications deliver a valid application upgrade file (`appup`). In earlier releases, a majority of the applications in Erlang/OTP did not support upgrade. Many of the applications use the `restart_application` instruction. These are applications for which it is not crucial to support real soft upgrade, for example, tools and library applications. The `restart_application` instruction ensures that all modules in the application are reloaded and thereby running the new code.

3.4.2 Upgrade of Core Applications

The core applications ERTS, Kernel, STDLIB, and SASL never allow real soft upgrade, but require the Erlang emulator to be restarted. This is indicated to the `release_handler` by the upgrade instruction `restart_new_emulator`. This instruction is always the very first instruction executed, and it restarts the emulator with the new versions of the above mentioned core applications and the old versions of all other applications. When the node is back up, all other upgrade instructions are executed, making sure each application is finally running its new version.

It might seem strange to do a two-step upgrade instead of just restarting the emulator with the new version of all applications. The reason for this design decision is to allow `code_change` functions to have side effects, for example, changing data on disk. It also guarantees that the upgrade mechanism for non-core applications does not differ depending on whether or not core applications are changed at the same time.

If, however, the more brutal variant is preferred, the the release upgrade file can be handwritten using only the single upgrade instruction `restart_emulator`. This instruction, in contrast to `restart_new_emulator`, causes the emulator to restart with the new versions of **all** applications.

Note: If other instructions are included before `restart_emulator` in the handwritten `relup` file, they are executed in the old emulator. This is a big risk since there is no guarantee that new beam code can be loaded into the old emulator. Adding instructions after `restart_emulator` has no effect as the `release_handler` will not execute them.

For information about the release upgrade file, see the relup(4) manual page in SASL. For more information about upgrade instructions, see the appup(4) manual page in SASL.

3.4.3 Applications that Still do Not Allow Code Upgrade

A few applications, such as HiPE, do not support upgrade. This is indicated by an application upgrade file containing only `{Vsn,[],[]}`. Any attempt at creating a release upgrade file with such input fails. The only way to force an upgrade involving applications like this is to handwrite the file `relup`, preferably as described above with only the `restart_emulator` instruction.

3.5 Versions

3.5.1 OTP Version

As of OTP release 17, the OTP release number corresponds to the major part of the OTP version. The OTP version as a concept was introduced in OTP 17. The version scheme used is described in detail in Version Scheme.

OTP of a specific version is a set of applications of specific versions. The application versions identified by an OTP version corresponds to application versions that have been tested together by the Erlang/OTP team at Ericsson AB. An OTP system can, however, be put together with applications from different OTP versions. Such a combination of application versions has not been tested by the Erlang/OTP team. It is therefore **always preferred to use OTP applications from one single OTP version**.

Release candidates have an `-rc<N>` suffix. The suffix `-rc0` is used during development up to the first release candidate.

Retrieving Current OTP Version

In an OTP source code tree, the OTP version can be read from the text file `<OTP source root>/OTP_VERSION`. The absolute path to the file can be constructed by calling `filename:join([code:root_dir(), "OTP_VERSION"])`.

In an installed OTP development system, the OTP version can be read from the text file `<OTP installation root>/releases/<OTP release number>/OTP_VERSION`. The absolute path to the file can by constructed by calling `filename:join([code:root_dir(), "releases", erlang:system_info(otp_release), "OTP_VERSION"])`.

If the version read from the `OTP_VERSION` file in a development system has a `**` suffix, the system has been patched using the `otp_patch_apply` tool. In this case, the system consists of application versions from multiple OTP versions. The version preceding the `**` suffix corresponds to the OTP version of the base system that has been patched. Notice that if a development system is updated by other means than `otp_patch_apply`, the file `OTP_VERSION` can identify an incorrect OTP version.

No `OTP_VERSION` file is placed in a target system created by OTP tools. This since one easily can create a target system where it is hard to even determine the base OTP version. You can, however, place such a file there if you know the OTP version.

OTP Versions Table

The text file `<OTP source root>/otp_versions.table`, which is part of the source code, contains information about all OTP versions from OTP 17.0 up to the current OTP version. Each line contains information about application versions that are part of a specific OTP version, and has the following format:

```
<OtpVersion> : <ChangedAppVersions> # <UnchangedAppVersions> :
```

`<OtpVersion>` has the format `OTP-<VSN>`, that is, the same as the git tag used to identify the source.

`<ChangedAppVersions>` and `<UnchangedAppVersions>` are space-separated lists of application versions and has the format `<application>-<vsn>`.

- `<ChangedAppVersions>` corresponds to changed applications with new version numbers in this OTP version.
- `<UnchangedAppVersions>` corresponds to unchanged application versions in this OTP version.

Both of them can be empty, but not at the same time. If `<ChangedAppVersions>` is empty, no changes have been made that change the build result of any application. This could, for example, be a pure bug fix of the build system. The order of lines is undefined. All white-space characters in this file are either space (character 32) or line-break (character 10).

By using ordinary UNIX tools like `sed` and `grep` one can easily find answers to various questions like:

- Which OTP versions are `kernel-3.0` part of?

  ```
  $ grep ' kernel-3\.0 ' otp_versions.table
  ```

- In which OTP version was `kernel-3.0` introduced?

  ```
  $ sed 's/#.*//;/ kernel-3\.0 /!d' otp_versions.table
  ```

The above commands give a bit more information than the exact answers, but adequate information when manually searching for answers to these questions.

Warning:

The format of the `otp_versions.table` might be subject to changes during the OTP 17 release.

3.5.2 Application Version

As of OTP 17.0 application versions use the same version scheme as the OTP version. Application versions part of a release candidate will however not have an `-rc<N>` suffix as the OTP version. Also note that a major increment in an application version does not necessarily imply a major increment of the OTP version. This depends on whether the major change in the application is considered as a major change for OTP as a whole or not.

3.5.3 Version Scheme

Note:

The version scheme was changed as of OTP 17.0. This implies that application versions used prior to OTP 17.0 do not adhere to this version scheme. A list of application versions used in OTP 17.0 is included at the end of this section

In the normal case, a version is constructed as `<Major>.<Minor>.<Patch>`, where `<Major>` is the most significant part.

However, more dot-separated parts than this can exist. The dot-separated parts consist of non-negative integers. If all parts less significant than `<Minor>` equals `0`, they are omitted. The three normal parts `<Major>.<Minor>.<Patch>` are changed as follows:

- `<Major>` - Increases when major changes, including incompatibilities, are made.
- `<Minor>` - Increases when new functionality is added.
- `<Patch>` - Increases when pure bug fixes are made.

When a part in the version number increases, all less significant parts are set to `0`.

An application version or an OTP version identifies source code versions. That is, it implies nothing about how the application or OTP has been built.

Order of Versions

Version numbers in general are only partially ordered. However, normal version numbers (with three parts) as of OTP 17.0 have a total or linear order. This applies both to normal OTP versions and normal application versions.

When comparing two version numbers that have an order, one compare each part as ordinary integers from the most significant part to less significant parts. The order is defined by the first parts of the same significance that differ. An OTP version with a larger version includes all changes that are part of a smaller OTP version. The same goes for application versions.

In general, versions can have more than three parts. The versions are then only partially ordered. Such versions are only used when branching off from another branch. When an extra part (out of the normal three parts) is added to a version number, a new branch of versions is made. The new branch has a linear order against the base version. However, versions on different branches have no order, and therefore one can only conclude that they all include what is included in their closest common ancestor. When branching multiple times from the same base version, `0` parts are added between the base version and the least significant `1` part until a unique version is found. Versions that have an order can be compared as described in the previous paragraph.

An example of branched versions: The version `6.0.2.1` is a branched version from the base version `6.0.2`. Versions on the form `6.0.2.<X>` can be compared with normal versions smaller than or equal to `6.0.2`, and other versions on the form `6.0.2.<X>`. The version `6.0.2.1` will include all changes in `6.0.2`. However, `6.0.3` will most

likely **not** include all changes in `6.0.2.1` (note that these versions have no order). A second branched version from the base version `6.0.2` will be version `6.0.2.0.1`, and a third branched version will be `6.0.2.0.0.1`.

3.5.4 Releases and Patches

When a new OTP release is released it will have an OTP version on the form `<Major>.0` where the major OTP version number equals the release number. The major version number is increased one step since the last major version. All other OTP versions with the same major OTP version number are patches on that OTP release.

Patches are either released as maintenance patch packages or emergency patch packages. The only difference is that maintenance patch packages are planned and usually contain more changes than emergency patch packages. Emergency patch packages are released to solve one or more specific issues when such are discovered.

The release of a maintenance patch package usually imply an increase of the OTP `<Minor>` version while the release of an emergency patch package usually imply an increase of the OTP `<Patch>` version. This is however not necessarily always the case since changes of OTP versions are based on the actual changes in the code and not based on whether the patch was planned or not. For more information see the Version Scheme section above.

3.5.5 OTP Versions Tree

All released OTP versions can be found in the **OTP Versions Tree** which is automatically updated whenever we release a new OTP version. Note that every version number as such explicitly define its position in the version tree. Nothing more than the version numbers are needed in order to construct the tree. The root of the tree is OTP version 17.0 which is when we introduced the new version scheme. The green versions are normal versions released on the main track. Old OTP releases will be maintained for a while on `maint` branches that have branched off from the main track. Old `maint` branches always branch off from the main track when the next OTP release is introduced into the main track. Versions on these old `maint` branches are marked blue. Besides the green and blue versions, there are also gray versions. These are versions on branches introduced in order to fix a specific problem for a specific customer on a specific base version. Branches with gray versions will typically become dead ends very quickly if not immediately.

3.5.6 OTP 17.0 Application Versions

The following list details the application versions that were part of OTP 17.0. If the normal part of an application version number compares as smaller than the corresponding application version in the list, the version number does not adhere to the version scheme introduced in OTP 17.0 and is to be considered as not having an order against versions used as of OTP 17.0.

- `asn1-3.0`
- `common_test-1.8`
- `compiler-5.0`
- `cosEvent-2.1.15`
- `cosEventDomain-1.1.14`
- `cosFileTransfer-1.1.16`
- `cosNotification-1.1.21`
- `cosProperty-1.1.17`
- `cosTime-1.1.14`
- `cosTransactions-1.2.14`
- `crypto-3.3`
- `debugger-4.0`
- `dialyzer-2.7`
- `diameter-1.6`
- `edoc-0.7.13`

- eldap-1.0.3
- erl_docgen-0.3.5
- erl_interface-3.7.16
- erts-6.0
- et-1.5
- eunit-2.2.7
- gs-1.5.16
- hipe-3.10.3
- ic-4.3.5
- inets-5.10
- jinterface-1.5.9
- kernel-3.0
- megaco-3.17.1
- mnesia-4.12
- observer-2.0
- odbc-2.10.20
- orber-3.6.27
- os_mon-2.2.15
- ose-1.0
- otp_mibs-1.0.9
- parsetools-2.0.11
- percept-0.8.9
- public_key-0.22
- reltool-0.6.5
- runtime_tools-1.8.14
- sasl-2.4
- snmp-4.25.1
- ssh-3.0.1
- ssl-5.3.4
- stdlib-2.0
- syntax_tools-1.6.14
- test_server-3.7
- tools-2.6.14
- typer-0.9.6
- webtool-0.8.10
- wx-1.2
- xmerl-1.3.7

3.6 Support, Compatibility, Deprecations, and Removal

3.6.1 Introduction

This document describes strategy regarding supported Releases, compatibility, deprecations and removal of functionality. This document was introduced in OTP 21. Actions taken regarding these issues before OTP 21 did not adhere this document.

3.6.2 Supported Releases

In general, bugs are only fixed on the latest release, and new features are introduced in the upcoming release that is under development. However, when we, due to internal reasons, fix bugs on older releases, these will be available and announced as well.

Due to the above, pull requests are only accepted on the `maint` and the `master` branches in our **git repository**. The `maint` branch contains changes planned for the next maintenance patch package on the latest OTP release and the `master` branch contain changes planned for the upcoming OTP release.

3.6.3 Compatibility

We always strive to remain as compatible as possible even in the cases where we give no compatibility guarantees.

Different parts of the system will be handled differently regarding compatibility. The following items describe how different parts of the system are handled.

Erlang Distribution

Erlang nodes can communicate across at least two preceding and two subsequent releases.

Compiled BEAM Code, NIF Libraries and Drivers

Compiled code can be loaded on at least two subsequent releases.

Loading on previous releases is **not** supported.

APIs

Compatible between releases.

Compiler Warnings

New warnings may be issued between releases.

Command Line Arguments

Incompatible changes may occur between releases.

OTP Build Procedures

Incompatible changes may occur between releases.

Under certain circumstances incompatible changes might be introduced even in parts of the system that should be compatible between releases. Things that might trigger incompatible changes like this are:

Security Issues

It might be necessary to introduce incompatible changes in order to solve a security issue. This kind of incompatibility might occur in a patch.

Bug Fixes

We will not be bug-compatible. A bug fix might introduce incompatible changes. This kind of incompatibility might occur in a patch.

Severe Previous Design Issues

> Some parts of OTP were designed a very long time ago and did not necessarily take today's computing environments into account. In some cases the consequences of those design decisions are too severe. This may be performance wise, scalability wise, etc. If we deem the consequences too severe, we might introduce incompatible changes. This kind of incompatibility will not be introduced in a patch, but instead in the next release.

Peripheral, trace, and debug functionality is at greater risk of being changed in an incompatible way than functionality in the language itself and core libraries used during operation.

3.6.4 Deprecation

Functionality is deprecated when new functionality is introduced that is preferred to be used instead of the old functionality that is being deprecated. The deprecation does **not** imply removal of the functionality unless an upcoming removal is explicitly stated in the deprecation.

Deprecated functionality will be documented as deprecated, and compiler warnings will be issued, when appropriate, as early as possible. That is, the new preferred functionality will appear at the same time as the deprecation is issued. A new deprecation will at least be announced in a release note and the documentation.

3.6.5 Removal

Legacy solutions may eventually need to be removed. In such cases, they will be phased out on a long enough time period to give users the time to adapt. Before removal of functionality it will be deprecated at least during one release with an explicit announcement about the upcoming removal. A new deprecation will at least be announced in a release note and the documentation.

Peripheral, trace, and debug functionality is at greater risk of removal than functionality in the language itself and core libraries used during operation.

4 Embedded Systems User's Guide

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

Note:

This is a supplementary section. You also need to read Section 1 Installation Guide.

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

4.1 Embedded Solaris

This section describes the operating system-specific parts of OTP that relate to Solaris.

4.1.1 Memory Use

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

4.1.2 Disk Space Use

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

4.1.3 Installing an Embedded System

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

- Creating user and installation directory
- Installing an embedded system
- Configuring automatic start at boot
- Making a hardware watchdog available
- Changing permission for reboot
- Setting TERM environment variable
- Adding patches
- Installing module os_sup in application os_mon

Several of the procedures in this section require expert knowledge of the Solaris operating system. For most of them super user privilege is needed.

Creating User and Installation Directory

It is recommended that the embedded environment is run by an ordinary user, that is, a user who does not have super user privileges.

In this section, it is assumed that the username is `otpuser` and that the home directory of that user is:

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

It is also assumed that in the home directory of `otpuser`, there is a directory named `otp`, the full path of which is:

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

This directory is the **installation directory** of the embedded environment.

Installing an Embedded System

The procedure for installing an embedded system is the same as for an ordinary system (see Installation Guide), except for the following:

- The (compressed) tape archive file is to be extracted in the installation directory defined above.
- It is not needed to link the start script to a standard directory like `/usr/local/bin`.

Configuring Automatic Start at Boot

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

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

```
S75otp.system
```

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

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

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

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

# Start/stop processes required for Erlang

case "$1" in
'start')
        # Start the Erlang emulator
        #
        su - otpuser -c "/export/home/otpuser/otp/bin/start" &
        ;;
'stop')
        killproc beam
        ;;
*)
        echo "Usage: $0 { start | stop }"
        ;;
esac
```

File `/export/home/otpuser/otp/bin/start` referred to in the above script is precisely the `start` script described in **Starting Erlang**. The script variable `OTP_ROOT` in that `start` script corresponds to the following example path used in this section:

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

The `start` script is to be edited accordingly.

Use of the `killproc` procedure in the above script can be combined with a call to `erl_call`, for example:

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

To take Erlang down gracefully, see the `erl_call(1)` manual page in `erl_interface` for details on the use of `erl_call`. However, that requires that Erlang runs as a distributed node, which is not always the case.

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

Making Hardware Watchdog Available

For Solaris running on VME boards from Force Computers, the onboard hardware watchdog can be activated, provided a VME bus driver is added to the operating system (see also Installation Problems).

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

Changing Permissions for Reboot

If the `HEART_COMMAND` environment variable is to be set in the `start` script in **Starting Erlang**, and if the value is to be set to the path of the Solaris `reboot` command, that is:

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

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

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

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

Setting TERM Environment Variable

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

```
TERM=sun
```

This is to be added to the `start` script.

Adding Patches

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

Installing Module os_sup in Application os_mon

The following four installation procedures require super user privilege:

Installation

- **Make a copy of the Solaris standard configuration file for `syslogd`:**
  - Make a copy of the Solaris standard configuration file for `syslogd`. This file is usually named `syslog.conf` and found in directory `/etc`.
  - The filename of the copy must be `syslog.conf.ORIG`. The directory location is optional; usually it is `/etc`. A simple way to do this is to issue the following command:

    ```
    cp /etc/syslog.conf /etc/syslog.conf.ORIG
    ```
- **Make an Erlang-specific configuration file for `syslogd`:**
  - Make an edited copy of the backup copy previously made.
  - The filename must be `syslog.conf.OTP`. The path must be the same as the backup copy.
  - The format of the configuration file is found in the `syslog.conf(5)` manual page, by issuing the command `man syslog.conf`.
  - Usually a line is added that is to state:
    - Which types of information that is to be supervised by Erlang
    - The name of the file (actually a named pipe) that is to receive the information
  - If, for example, only information originating from the UNIX kernel is to be supervised, the line is to begin with `kern.LEVEL`. For the possible values of `LEVEL`, see `syslog.conf(5)`.

  - After at least one tab-character, the line added is to contain the full name of the named pipe where `syslogd` writes its information. The path must be the same as for the files `syslog.conf.ORIG` and `syslog.conf.OTP`. The filename must be `syslog.otp`.
  - If the directory for the files `syslog.conf.ORIG` and `syslog.conf.OTP` is `/etc`, the line in `syslog.conf.OTP` is as follows:

    ```
    kern.LEVEL               /etc/syslog.otp
    ```

- **Check the file privileges of the configuration files:**
  - The configuration files is to have `rw-r--r--` file privileges and be owned by root.
  - A simple way to do this is to issue these commands:

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

  - Notice that if the files `syslog.conf.ORIG` and `syslog.conf.OTP` are not in directory `/etc`, the file path in the second and third command must be modified.
- **Modify file privileges and ownership of the `mod_syslog` utility:**
  - The file privileges and ownership of the `mod_syslog` utility must be modified.
  - The full name of the binary executable file is derived from the position of application `os_mon` in the file system by adding `/priv/bin/mod_syslog`. The generic full name of the binary executable file is thus:

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

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

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

Testing the Application Configuration File

The following procedure does not require root privilege:

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

  The generic full name of the file is thus:

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

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

Parameter	Function	Standard value

`start_os_sup`	Specifies if `os_sup` is to be started or not.	`true` for the first instance on the hardware; `false` for the other instances
`os_sup_own`	The directory for (1) back-up copy and (2) Erlang-specific configuration file for `syslogd`	`"/etc"`
`os_sup_syslogconf`	The full name for the Solaris standard configuration file for `syslogd`	`"/etc/syslog.conf"`
`error_tag`	The tag for the messages that are sent to the error logger in the Erlang runtime system	`std_error`

Table 1.1: Configuration Parameters

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

Example: Contents of an application configuration file:

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

Related Documents

See the `os_mon(3)` application, the `application(3)` manual page in Kernel, and the `erl(1)` manual page in ERTS.

Installation Problems

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

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

Warning:

It is recommended to add these lines to avoid a clash. The clash can make it impossible to boot the system.

4.1.4 Starting Erlang

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

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

Basically, the procedure is as follows:

- The `start` program is called when the machine is started.
- It calls `run_erl`, which sets up things so the operator can attach to the system.
- It calls `start_erl`, which calls the correct version of `erlexec` (which is located in `<ERL_INSTALL_DIR>/erts-EVsn/bin`) with the correct `boot` and `config` files.

4.1.5 Programs

start

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

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

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

The `<RELDIR>` directory is where new release packets are installed, and where the release handler keeps information about releases. For more information, see the `release_handler(3)` manual page in SASL.

The following script illustrates the default behaviour of the program:

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

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

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

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

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

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

ROOTDIR=/usr/local/otp

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

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

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

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

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

run_erl

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

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

Here:

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

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

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

The most recent log file is the rightmost in each row. That is, the most recent file is the one with the highest number, or if there are already four files, the one before the skip.

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

to_erl

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

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

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

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

start_erl

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

The following script illustrates the behaviour of the program:

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

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

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

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

If a diskless and/or read-only client node with the SASL configuration parameter `static_emulator` set to `true` is about to start, the `-boot` and `-config` flags must be changed.

As such a client cannot read a new `start_erl.data` file (the file cannot be changed dynamically). The boot and config files are always fetched from the same place (but with new contents if a new release has been installed).

The `release_handler` copies these files to the `bin` directory in the client directory at the master nodes whenever a new release is made permanent.

Assuming the same `CLIENTDIR` as above, the last line is to look like:

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

4.2 Windows NT

This section describes the operating system-specific parts of OTP that relate to Windows NT.

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

4.2.1 Memory Use

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

4.2.2 Disk Space Use

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

4.2.3 Installing an Embedded System

Normal Windows NT installation is performed. No additional application programs are needed, such as Internet Explorer or web server. Networking with TCP/IP is required.

Service Pack 4 or later must be installed.

Hardware Watchdog

For Windows NT running on standard PCs with ISA and/or PCI bus, an extension card with a hardware watchdog can be installed.

For more information, see the `heart(3)` manual page in Kernel.

4.2.4 Starting Erlang

On an embedded system, the `erlsrv` module is to be used to install the Erlang process as a Windows system service. This service can start after Windows NT has booted.

For more information, see the `erlsrv` manual page in ERTS.

5 Getting Started With Erlang

5.1 Introduction

This section is a quick start tutorial to get you started with Erlang. Everything in this section is true, but only part of the truth. For example, only the simplest form of the syntax is shown, not all esoteric forms. Also, parts that are greatly simplified are indicated with *manual*. This means that a lot more information on the subject is to be found in the Erlang book or in Erlang Reference Manual.

5.1.1 Prerequisites

The reader of this section is assumed to be familiar with the following:

- Computers in general
- Basics on how computers are programmed

5.1.2 Omitted Topics

The following topics are not treated in this section:

- References.
- Local error handling (catch/throw).
- Single direction links (monitor).
- Handling of binary data (binaries / bit syntax).
- List comprehensions.
- How to communicate with the outside world and software written in other languages (ports); this is described in Interoperability Tutorial.
- Erlang libraries (for example, file handling).
- OTP and (in consequence) the Mnesia database.
- Hash tables for Erlang terms (ETS).
- Changing code in running systems.

5.2 Sequential Programming

5.2.1 The Erlang Shell

Most operating systems have a command interpreter or shell, UNIX and Linux have many, Windows has the command prompt. Erlang has its own shell where bits of Erlang code can be written directly, and be evaluated to see what happens (see the shell(3) manual page in STDLIB).

Start the Erlang shell (in Linux or UNIX) by starting a shell or command interpreter in your operating system and typing `erl`. You will see something like this.

```
% erl
Erlang R15B (erts-5.9.1) [source] [smp:8:8] [rq:8] [async-threads:0] [hipe] [kernel-poll:false]

Eshell V5.9.1  (abort with ^G)
1>
```

Type "2 + 5." in the shell and then press Enter (carriage return). Notice that you tell the shell you are done entering code by finishing with a full stop "." and a carriage return.

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

As shown, the Erlang shell numbers the lines that can be entered, (as 1> 2>) and that it correctly says that 2 + 5 is 7. If you make writing mistakes in the shell, you can delete with the backspace key, as in most shells. There are many more editing commands in the shell (see tty - A command line interface in ERTS User's Guide).

(Notice that many line numbers given by the shell in the following examples are out of sequence. This is because this tutorial was written and code-tested in separate sessions).

Here is a bit more complex calculation:

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

Notice the use of brackets, the multiplication operator "*", and the division operator "/", as in normal arithmetic (see Expressions).

Press Control-C to shut down the Erlang system and the Erlang shell.

The following output is shown:

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

Type "a" to leave the Erlang system.

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

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

5.2.2 Modules and Functions

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

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

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

It is not hard to guess that this program doubles the value of numbers. The first two lines of the code are described later. Let us compile the program. This can be done in an Erlang shell as follows, where `c` means compile:

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

The `{ok,tut}` means that the compilation is OK. If it says "error" it means that there is some mistake in the text that you entered. Additional error messages gives an idea to what is wrong so you can modify the text and then try to compile the program again.

Now run the program:

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

As expected, double of 10 is 20.

Now let us get back to the first two lines of the code. Erlang programs are written in files. Each file contains an Erlang **module**. The first line of code in the module is the module name (see Modules):

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

Thus, the module is called **tut**. Notice the full stop "." at the end of the line. The files which are used to store the module must have the same name as the module but with the extension ".erl". In this case the file name is `tut.erl`. When using a function in another module, the syntax `module_name:function_name(arguments)` is used. So the following means call function `double` in module `tut` with argument "10".

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

The second line says that the module `tut` contains a function called `double`, which takes one argument (`X` in our example):

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

The second line also says that this function can be called from outside the module `tut`. More about this later. Again, notice the "." at the end of the line.

Now for a more complicated example, the factorial of a number. For example, the factorial of 4 is 4 * 3 * 2 * 1, which equals 24.

Enter the following code in a file named `tut1.erl`:

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

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

So this is a module, called `tut1` that contains a function called `fac>`, which takes one argument, `N`.

The first part says that the factorial of 1 is 1.:

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

Notice that this part ends with a semicolon ";" that indicates that there is more of the function `fac>` to come.

The second part says that the factorial of N is N multiplied by the factorial of N - 1:

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

Notice that this part ends with a "." saying that there are no more parts of this function.

Compile the file:

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

And now calculate the factorial of 4.

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

Here the function `fac>` in module `tut1` is called with argument `4`.

A function can have many arguments. Let us expand the module `tut1` with the function to multiply two numbers:

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

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

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

Notice that it is also required to expand the `-export` line with the information that there is another function `mult` with two arguments.

Compile:

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

Try out the new function `mult`:

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

In this example the numbers are integers and the arguments in the functions in the code `N`, `X`, and `Y` are called variables. Variables must start with a capital letter (see Variables). Examples of variables are `Number`, `ShoeSize`, and `Age`.

5.2.3 Atoms

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

Enter the next program in a file named `tut2.erl`). It can be useful for converting from inches to centimeters and conversely:

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

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

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

Compile:

```
9> c(tut2).
{ok,tut2}
```

Test:

```
10> tut2:convert(3, inch).
1.1811023622047243
11> tut2:convert(7, centimeter).
17.78
```

Notice the introduction of decimals (floating point numbers) without any explanation. Hopefully you can cope with that.

Let us see what happens if something other than `centimeter` or `inch` is entered in the `convert` function:

```
12> tut2:convert(3, miles).
** exception error: no function clause matching tut2:convert(3,miles) (tut2.erl, line 4)
```

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

```
13> v(12).
{'EXIT',{function_clause,[{tut2,convert,
                                [3,miles],
                                [{file,"tut2.erl"},{line,4}]},
                           {erl_eval,do_apply,6,
                                     [{file,"erl_eval.erl"},{line,677}]},
                           {shell,exprs,7,[{file,"shell.erl"},{line,687}]},
                           {shell,eval_exprs,7,[{file,"shell.erl"},{line,642}]},
                           {shell,eval_loop,3,
                                  [{file,"shell.erl"},{line,627}]}]}}
```

5.2.4 Tuples

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

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

Does this mean that 3 is in inches? Or does it mean that 3 is in centimeters and is to be converted to inches? Erlang has a way to group things together to make things more understandable. These are called **tuples** and are surrounded by curly brackets, "{" and "}".

So, `{inch,3}` denotes 3 inches and `{centimeter,5}` denotes 5 centimeters. Now let us write a new program that converts centimeters to inches and conversely. Enter the following code in a file called `tut3.erl`):

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

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

Compile and test:

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

Notice on line 16 that 5 inches is converted to centimeters and back again and reassuringly get back to the original value. That is, the argument to a function can be the result of another function. Consider how line 16 (above) works. The argument given to the function `{inch,5}` is first matched against the first head clause of `convert_length`, that is, `convert_length({centimeter,X})`. It can be seen that `{centimeter,X}` does not match `{inch,5}` (the head is the bit before the "->"). This having failed, let us try the head of the next clause that is, `convert_length({inch,Y})`. This matches, and `Y` gets the value 5.

Tuples can have more than two parts, in fact as many parts as you want, and contain any valid Erlang **term**. For example, to represent the temperature of various cities of the world:

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

Tuples have a fixed number of items in them. Each item in a tuple is called an **element**. In the tuple `{moscow, {c,-10}}`, element 1 is `moscow` and element 2 is `{c,-10}`. Here `c` represents Celsius and `f` Fahrenheit.

5.2.5 Lists

Whereas tuples group things together, it is also needed to represent lists of things. Lists in Erlang are surrounded by square brackets, "[" and "]". For example, a list of the temperatures of various cities in the world can be:

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

Notice that this list was so long that it did not fit on one line. This does not matter, Erlang allows line breaks at all "sensible places" but not, for example, in the middle of atoms, integers, and others.

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

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

To separate the first elements of the list from the rest of the list, `|` is used. `First` has got value 1 and `TheRest` has got the value [2,3,4,5].

Another example:

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

Here you see the use of | to get the first two elements from the list. If you try to get more elements from the list than there are elements in the list, an error is returned. Notice also the special case of the list with no elements, []:

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

In the previous examples, new variable names are used, instead of reusing the old ones: `First`, `TheRest`, `E1`, `E2`, `R`, `A`, `B`, and `C`. The reason for this is that a variable can only be given a value once in its context (scope). More about this later.

The following example shows how to find the length of a list. Enter the following code in a file named `tut4.erl`):

```
-module(tut4).

-export([list_length/1]).

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

Compile and test:

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

Explanation:

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

The length of an empty list is obviously 0.

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

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

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

In general, tuples are used where "records" or "structs" are used in other languages. Also, lists are used when representing things with varying sizes, that is, where linked lists are used in other languages.

Erlang does not have a string data type. Instead, strings can be represented by lists of Unicode characters. This implies for example that the list `[97,98,99]` is equivalent to "abc". The Erlang shell is "clever" and guesses what list you mean and outputs it in what it thinks is the most appropriate form, for example:

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

5.2.6 Maps

Maps are a set of key to value associations. These associations are encapsulated with "#{" and "}". To create an association from "key" to value 42:

```
> #{ "key" => 42 }.
#{"key" => 42}
```

Let us jump straight into the deep end with an example using some interesting features.

The following example shows how to calculate alpha blending using maps to reference color and alpha channels. Enter the code in a file named `color.erl`):

```
-module(color).

-export([new/4, blend/2]).

-define(is_channel(V), (is_float(V) andalso V >= 0.0 andalso V =< 1.0)).

new(R,G,B,A) when ?is_channel(R), ?is_channel(G),
                  ?is_channel(B), ?is_channel(A) ->
    #{red => R, green => G, blue => B, alpha => A}.

blend(Src,Dst) ->
    blend(Src,Dst,alpha(Src,Dst)).

blend(Src,Dst,Alpha) when Alpha > 0.0 ->
    Dst#{
        red   := red(Src,Dst) / Alpha,
        green := green(Src,Dst) / Alpha,
        blue  := blue(Src,Dst) / Alpha,
        alpha := Alpha
    };
blend(_,Dst,_) ->
    Dst#{
        red   := 0.0,
        green := 0.0,
        blue  := 0.0,
        alpha := 0.0
    }.

alpha(#{alpha := SA}, #{alpha := DA}) ->
    SA + DA*(1.0 - SA).

red(#{red := SV, alpha := SA}, #{red := DV, alpha := DA}) ->
    SV*SA + DV*DA*(1.0 - SA).
green(#{green := SV, alpha := SA}, #{green := DV, alpha := DA}) ->
    SV*SA + DV*DA*(1.0 - SA).
blue(#{blue := SV, alpha := SA}, #{blue := DV, alpha := DA}) ->
    SV*SA + DV*DA*(1.0 - SA).
```

Compile and test:

```
> c(color).
{ok,color}
> C1 = color:new(0.3,0.4,0.5,1.0).
#{alpha => 1.0,blue => 0.5,green => 0.4,red => 0.3}
> C2 = color:new(1.0,0.8,0.1,0.3).
#{alpha => 0.3,blue => 0.1,green => 0.8,red => 1.0}
> color:blend(C1,C2).
#{alpha => 1.0,blue => 0.5,green => 0.4,red => 0.3}
> color:blend(C2,C1).
#{alpha => 1.0,blue => 0.38,green => 0.52,red => 0.51}
```

This example warrants some explanation:

```
-define(is_channel(V), (is_float(V) andalso V >= 0.0 andalso V =< 1.0)).
```

First a macro `is_channel` is defined to help with the guard tests. This is only here for convenience and to reduce syntax cluttering. For more information about macros, see The Preprocessor.

```
new(R,G,B,A) when ?is_channel(R), ?is_channel(G),
                  ?is_channel(B), ?is_channel(A) ->
    #{red => R, green => G, blue => B, alpha => A}.
```

The function `new/4` creates a new map term and lets the keys `red`, `green`, `blue`, and `alpha` be associated with an initial value. In this case, only float values between and including 0.0 and 1.0 are allowed, as ensured by the `?is_channel/1` macro for each argument. Only the `=>` operator is allowed when creating a new map.

By calling `blend/2` on any color term created by `new/4`, the resulting color can be calculated as determined by the two map terms.

The first thing `blend/2` does is to calculate the resulting alpha channel:

```
alpha(#{alpha := SA}, #{alpha := DA}) ->
    SA + DA*(1.0 - SA).
```

The value associated with key `alpha` is fetched for both arguments using the `:=` operator. The other keys in the map are ignored, only the key `alpha` is required and checked for.

This is also the case for functions `red/2`, `blue/2`, and `green/2`.

```
red(#{red := SV, alpha := SA}, #{red := DV, alpha := DA}) ->
    SV*SA + DV*DA*(1.0 - SA).
```

The difference here is that a check is made for two keys in each map argument. The other keys are ignored.

Finally, let us return the resulting color in `blend/3`:

```
blend(Src,Dst,Alpha) when Alpha > 0.0 ->
    Dst#{
        red   := red(Src,Dst) / Alpha,
        green := green(Src,Dst) / Alpha,
        blue  := blue(Src,Dst) / Alpha,
        alpha := Alpha
    };
```

The `Dst` map is updated with new channel values. The syntax for updating an existing key with a new value is with the `:=` operator.

5.2.7 Standard Modules and Manual Pages

Erlang has many standard modules to help you do things. For example, the module `io` contains many functions that help in doing formatted input/output. To look up information about standard modules, the command `erl -man` can

be used at the operating shell or command prompt (the same place as you started `erl`). Try the operating system shell command:

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

MODULE
     io - Standard I/O Server Interface Functions

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

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

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

5.2.8 Writing Output to a Terminal

It is nice to be able to do formatted output in examples, so the next example shows a simple way to use the `io:format` function. Like all other exported functions, you can test the `io:format` function in the shell:

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

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

5.2.9 A Larger Example

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

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

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

%% Only this function is exported
format_temps([])->                      % No output for an empty list
    ok;
format_temps([City | Rest]) ->
    print_temp(convert_to_celsius(City)),
    format_temps(Rest).

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

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

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

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

Notice also that when testing the program from the shell, the input is spread over two lines as the line was too long.

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

Here is a function call as `convert_to_celsius({moscow,{c,-10}})` as the argument to the function `print_temp`. When function calls are **nested** like this, they execute (evaluate) from the inside out. That is, first `convert_to_celsius({moscow,{c,-10}})` is evaluated, which gives the value `{moscow,{c,-10}}` as the temperature is already in Celsius. Then `print_temp({moscow,{c,-10}})` is evaluated. The function `convert_to_celsius` works in a similar way to the `convert_length` function in the previous example.

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

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

5.2.10 Matching, Guards, and Scope of Variables

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

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

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

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

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

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

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

In `list_max/2`, you walk down the list and use `Head` instead of `Result_so_far` when `Head` > `Result_so_far`. `when` is a special word used before the -> in the function to say that you only use this part of the function if the test that follows is true. A test of this type is called **guard**. If the guard is false (that is, the guard fails), the next part of the function is tried. In this case, if `Head` is not greater than `Result_so_far`, then it must be smaller or equal to it. This means that a guard on the next part of the function is not needed.

Some useful operators in guards are:

- < less than
- > greater than
- == equal
- >= greater or equal
- =< less or equal
- /= not equal

(see Guard Sequences).

To change the above program to one that works out the minimum value of the element in a list, you only need to write < instead of >. (But it would be wise to change the name of the function to `list_min`.)

Earlier it was mentioned that a variable can only be given a value once in its scope. In the above you see that `Result_so_far` is given several values. This is OK since every time you call `list_max/2` you create a new scope and one can regard `Result_so_far` as a different variable in each scope.

Another way of creating and giving a variable a value is by using the match operator = . So if you write `M = 5`, a variable called `M` is created with the value 5. If, in the same scope, you then write `M = 6`, an error is returned. Try this out in the shell:

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

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

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

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

If you try to do the same again with another city, an error is returned:

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

Variables can also be used to improve the readability of programs. For example, in function `list_max/2` above, you can write:

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

This is possibly a little clearer.

5.2.11 More About Lists

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

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

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

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

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

```
-module(tut8).

-export([reverse/1]).

reverse(List) ->
    reverse(List, []).

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

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

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

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

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

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

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

The module `lists` contains many functions for manipulating lists, for example, for reversing them. So before writing a list-manipulating function it is a good idea to check if one not already is written for you (see the lists(3) manual page in STDLIB).

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

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

format_temps(List_of_cities) ->
    convert_list_to_c(List_of_cities).

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

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

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

Test the function:

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

Explanation:

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

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

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

or:

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

This is done until the end of the list is reached, that is, the list is empty:

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

Now when the list is converted, a function to print it is added:

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

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

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

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

convert_list_to_c([]) ->
    [].

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

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

Now a function has to be added to find the cities with the maximum and minimum temperatures. The following program is not the most efficient way of doing this as you walk through the list of cities four times. But it is better to first strive for clarity and correctness and to make programs efficient only if needed.

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

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

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

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

convert_list_to_c([]) ->
    [].

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

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

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

find_max_and_min([], Max_City, Min_City) ->
    {Max_City, Min_City}.

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

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

5.2.12 If and Case

The function `find_max_and_min` works out the maximum and minimum temperature. A new construct, `if`, is introduced here. If works as follows:

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

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

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

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

test_if(A, B) ->
    if
        A == 5 ->
            io:format("A == 5~n", []),
            a_equals_5;
        B == 6 ->
            io:format("B == 6~n", []),
            b_equals_6;
        A == 2, B == 3 ->                       %That is A equals 2 and B equals 3
            io:format("A == 2, B == 3~n", []),
            a_equals_2_b_equals_3;
        A == 1 ; B == 7 ->                      %That is A equals 1 or B equals 7
            io:format("A == 1 ; B == 7~n", []),
            a_equals_1_or_b_equals_7
    end.
```

Testing this program gives:

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

Notice that `tut9:test_if(33,33)` does not cause any condition to succeed. This leads to the run time error `if_clause`, here nicely formatted by the shell. See Guard Sequences for details of the many guard tests available.

`case` is another construct in Erlang. Recall that the `convert_length` function was written as:

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

The same program can also be written as:

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

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

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

Both `case` and `if` have **return values**, that is, in the above example `case` returned either `{inch,X/2.54}` or `{centimeter,Y*2.54}`. The behaviour of `case` can also be modified by using guards. The following example clarifies this. It tells us the length of a month, given the year. The year must be known, since February has 29 days in a leap year.

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

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

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

5.2.13 Built-In Functions (BIFs)

BIFs are functions that for some reason are built-in to the Erlang virtual machine. BIFs often implement functionality that is impossible or is too inefficient to implement in Erlang. Some BIFs can be called using the function name only but they are by default belonging to the `erlang` module. For example, the call to the BIF `trunc` below is equivalent to a call to `erlang:trunc`.

As shown, first it is checked if a year is leap. If a year is divisible by 400, it is a leap year. To determine this, first divide the year by 400 and use the BIF `trunc` (more about this later) to cut off any decimals. Then multiply by 400 again and see if the same value is returned again. For example, year 2004:

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

2000 is not the same as 2004, so 2004 is not divisible by 400. Year 2000:

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

That is, a leap year. The next two `trunc`-tests evaluate if the year is divisible by 100 or 4 in the same way. The first `if` returns `leap` or `not_leap`, which lands up in the variable `Leap`. This variable is used in the guard for `feb` in the following `case` that tells us how long the month is.

This example showed the use of `trunc`. It is easier to use the Erlang operator `rem` that gives the remainder after division, for example:

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

So instead of writing:

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

it can be written:

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

There are many other BIFs such as `trunc`. Only a few BIFs can be used in guards, and you cannot use functions you have defined yourself in guards. (see Guard Sequences) (For advanced readers: This is to ensure that guards do not have side effects.) Let us play with a few of these functions in the shell:

```
75> trunc(5.6).
5
76> round(5.6).
6
77> length([a,b,c,d]).
4
78> float(5).
5.0
79> is_atom(hello).
true
80> is_atom("hello").
false
81> is_tuple({paris, {c, 30}}).
true
82> is_tuple([paris, {c, 30}]).
false
```

All of these can be used in guards. Now for some BIFs that cannot be used in guards:

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

These three BIFs do conversions that would be difficult (or impossible) to do in Erlang.

5.2.14 Higher-Order Functions (Funs)

Erlang, like most modern functional programming languages, has higher-order functions. Here is an example using the shell:

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

Here is defined a function that doubles the value of a number and assigned this function to a variable. Thus `Xf(5)` returns value 10. Two useful functions when working with lists are `foreach` and `map`, which are defined as follows:

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

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

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

```
88> Add_3 = fun(X) -> X + 3 end.
#Fun<erl_eval.5.123085357>
89> lists:map(Add_3, [1,2,3]).
[4,5,6]
```

Let us (again) print the temperatures in a list of cities:

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

Let us now define a fun that can be used to go through a list of cities and temperatures and transform them all to Celsius.

```
-module(tut13).

-export([convert_list_to_c/1]).

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

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

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

The `convert_to_c` function is the same as before, but here it is used as a fun:

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

When a function defined elsewhere is used as a fun, it can be referred to as `Function/Arity` (remember that `Arity` = number of arguments). So in the `map`-call `lists:map(fun convert_to_c/1, List)` is written. As shown, `convert_list_to_c` becomes much shorter and easier to understand.

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

```
-module(tut13).

-export([convert_list_to_c/1]).

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

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

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

In `sort` the fun is used:

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

Here the concept of an **anonymous variable** "_" is introduced. This is simply shorthand for a variable that gets a value, but the value is ignored. This can be used anywhere suitable, not just in funs. `Temp1 < Temp2` returns `true` if `Temp1` is less than `Temp2`.

5.3 Concurrent Programming

5.3.1 Processes

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

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

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

```
-module(tut14).

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

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

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

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

As shown, the function `say_something` writes its first argument the number of times specified by second argument. The function `start` starts two Erlang processes, one that writes "hello" three times and one that writes "goodbye" three times. Both processes use the function `say_something`. Notice that a function used in this way by `spawn`, to start a process, must be exported from the module (that is, in the `-export` at the start of the module).

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

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

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

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

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

5.3.2 Message Passing

In the following example two processes are created and they send messages to each other a number of times.

```
-module(tut15).

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

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

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

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

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

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

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

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

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

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

This process executes:

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

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

The process "pong" now does:

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

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

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

Notice there is no ";" before the `end`.

Messages between Erlang processes are simply valid Erlang terms. That is, they can be lists, tuples, integers, atoms, pids, and so on.

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

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

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

Now back to the ping pong example.

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

```
{ping, Ping_PID}
```

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

```
Ping_PID ! pong
```

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

```
Pid ! Message
```

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

After sending the message `pong` to the process "ping", "pong" calls the `pong` function again, which causes it to get back to the `receive` again and wait for another message.

Now let us look at the process "ping". Recall that it was started by executing:

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

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

The second clause sends a message to "pong":

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

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

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

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

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

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

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

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

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

5.3.3 Registered Process Names

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

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

Let us now rewrite the ping pong example using this and give the name `pong` to the "pong" process:

```
-module(tut16).

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

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

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

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

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

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

Here the `start/0` function,

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

both spawns the "pong" process and gives it the name `pong`. In the "ping" process, messages can be sent to `pong` by:

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

`ping/2` now becomes `ping/1` as the argument `Pong_PID` is not needed.

5.3.4 Distributed Programming

Let us rewrite the ping pong program with "ping" and "pong" on different computers. First a few things are needed to set up to get this to work. The distributed Erlang implementation provides a very basic authentication mechanism to prevent unintentional access to an Erlang system on another computer. Erlang systems which talk to each other must have the same **magic cookie**. The easiest way to achieve this is by having a file called `.erlang.cookie` in your home directory on all machines on which you are going to run Erlang systems communicating with each other:

- On Windows systems the home directory is the directory pointed out by the environment variable $HOME - you may need to set this.
- On Linux or UNIX you can safely ignore this and simply create a file called `.erlang.cookie` in the directory you get to after executing the command `cd` without any argument.

The `.erlang.cookie` file is to contain a line with the same atom. For example, on Linux or UNIX, in the OS shell:

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

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

When you start an Erlang system that is going to talk to other Erlang systems, you must give it a name, for example:

```
$ erl -sname my_name
```

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

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

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

```
-module(tut17).

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

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

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

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

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

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

Let us assume there are two computers called gollum and kosken. First a node is started on kosken, called ping, and then a node on gollum, called pong.

On kosken (on a Linux/UNIX system):

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

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

On gollum:

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

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

Now the "pong" process on gollum is started:

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

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

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

As shown, the ping pong program has run. On the "pong" side:

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

Looking at the `tut17` code, you see that the `pong` function itself is unchanged, the following lines work in the same way irrespective of on which node the "ping" process is executes:

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

Thus, Erlang pids contain information about where the process executes. So if you know the pid of a process, the "!" operator can be used to send it a message disregarding if the process is on the same node or on a different node.

A difference is how messages are sent to a registered process on another node:

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

A tuple `{registered_name,node_name}` is used instead of just the `registered_name`.

In the previous example, "ping" and "pong" were started from the shells of two separate Erlang nodes. `spawn` can also be used to start processes in other nodes.

The next example is the ping pong program, yet again, but this time "ping" is started in another node:

```
-module(tut18).

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

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

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

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

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

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

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

Notice that all the output is received on gollum. This is because the I/O system finds out where the process is spawned from and sends all output there.

5.3.5 A Larger Example

Now for a larger example with a simple "messenger". The messenger is a program that allows users to log in on different nodes and send simple messages to each other.

Before starting, notice the following:

- This example only shows the message passing logic - no attempt has been made to provide a nice graphical user interface, although this can also be done in Erlang.
- This sort of problem can be solved easier by use of the facilities in OTP, which also provide methods for updating code on the fly and so on (see OTP Design Principles).
- The first program contains some inadequacies regarding handling of nodes which disappear. These are corrected in a later version of the program.

The messenger is set up by allowing "clients" to connect to a central server and say who and where they are. That is, a user does not need to know the name of the Erlang node where another user is located to send a message.

File `messenger.erl`:

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

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

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

%%% This is the server process for the "messenger"
%%% the user list has the format [{ClientPid1, Name1},{ClientPid22, Name2},...]
server(User_List) ->
    receive
        {From, logon, Name} ->
            New_User_List = server_logon(From, Name, User_List),
            server(New_User_List);
```

```
        {From, logoff} ->
            New_User_List = server_logoff(From, User_List),
            server(New_User_List);
        {From, message_to, To, Message} ->
            server_transfer(From, To, Message, User_List),
            io:format("list is now: ~p~n", [User_List]),
            server(User_List)
    end.

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


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

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


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


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

logoff() ->
    mess_client ! logoff.

message(ToName, Message) ->
```

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


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

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

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

To use this program, you need to:

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

In the following example using this program, nodes are started on four different computers. If you do not have that many machines available on your network, you can start several nodes on the same machine.

Four Erlang nodes are started up: messenger@super, c1@bilbo, c2@kosken, c3@gollum.

First the server at messenger@super is started up:

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

Now Peter logs on at c1@bilbo:

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

James logs on at c2@kosken:

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

And Fred logs on at c3@gollum:

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

Now Peter sends Fred a message:

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

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

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

James now tries to send a message to Fred:

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

But this fails as Fred has already logged off.

First let us look at some of the new concepts that have been introduced.

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

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

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

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

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

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

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

There are many very useful functions in the `lists` module.

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

A process terminates when there is nothing more for it to do, that is, the last function it calls simply returns and does not call another function. Another way for a process to terminate is for it to call `exit/1`. The argument to `exit/1` has a special meaning, which is discussed later. In this example, `exit(normal)` is done, which has the same effect as a process running out of functions to call.

The BIF `whereis(RegisteredName)` checks if a registered process of name `RegisteredName` exists. If it exists, the pid of that process is returned. If it does not exist, the atom `undefined` is returned.

You should by now be able to understand most of the code in the messenger-module. Let us study one case in detail: a message is sent from one user to another.

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

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

After testing that the client process exists:

```
whereis(mess_client)
```

And a message is sent to `mess_client`:

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

The client sends the message to the server by:

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

And waits for a reply from the server.

The server receives this message and calls:

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

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

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

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

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

This is received by the client, which in turn does `exit(normal)` and terminates. If `keysearch` returns `{value, {From,Name}}` it is certain that the user is logged on and that his name (peter) is in variable `Name`.

Let us now call:

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

Notice that as this is `server_transfer/5`, it is not the same as the previous function `server_transfer/4`. Another `keysearch` is done on `User_List` to find the pid of the client corresponding to fred:

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

This time argument 2 is used, which is the second element in the tuple. If this returns the atom `false`, fred is not logged on and the following message is sent:

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

This is received by the client.

If `keysearch` returns:

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

The following message is sent to fred's client:

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

The following message is sent to peter's client:

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

Fred's client receives the message and prints it:

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

Peter's client receives the message in the `await_result` function.

5.4 Robustness

Several things are wrong with the messenger example in A Larger Example. For example, if a node where a user is logged on goes down without doing a logoff, the user remains in the server's `User_List`, but the client disappears. This makes it impossible for the user to log on again as the server thinks the user already is logged on.

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

5.4.1 Time-outs

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

```
-module(tut19).

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

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

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

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

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

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

After this is compiled and the file `tut19.beam` is copied to the necessary directories, the following is seen on (pong@kosken):

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

And the following is seen on (ping@gollum):

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

The time-out is set in:

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

The time-out (`after 5000`) is started when `receive` is entered. The time-out is canceled if `{ping,Ping_PID}` is received. If `{ping,Ping_PID}` is not received, the actions following the time-out are done after 5000 milliseconds. `after` must be last in the `receive`, that is, preceded by all other message reception specifications in the `receive`. It is also possible to call a function that returned an integer for the time-out:

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

In general, there are better ways than using time-outs to supervise parts of a distributed Erlang system. Time-outs are usually appropriate to supervise external events, for example, if you have expected a message from some external system within a specified time. For example, a time-out can be used to log a user out of the messenger system if they have not accessed it for, say, ten minutes.

5.4.2 Error Handling

Before going into details of the supervision and error handling in an Erlang system, let us see how Erlang processes terminate, or in Erlang terminology, **exit**.

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

A process which encounters a runtime error (for example, divide by zero, bad match, trying to call a function that does not exist and so on) exits with an error, that is, has an **abnormal** exit. A process which executes exit(Reason) where `Reason` is any Erlang term except the atom `normal`, also has an abnormal exit.

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

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

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

The default behaviour in the two other cases (that is, abnormal exit) above is to:

- Bypass all messages to the receiving process.
- Kill the receiving process.
- Propagate the same error signal to the links of the killed process.

In this way you can connect all processes in a transaction together using links. If one of the processes exits abnormally, all the processes in the transaction are killed. As it is often wanted to create a process and link to it at the same time, there is a special BIF, spawn_link that does the same as `spawn`, but also creates a link to the spawned process.

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

```
-module(tut20).

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

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

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

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

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

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

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

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

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

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

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

```
-module(tut21).

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

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

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

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

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

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

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

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

5.4.3 The Larger Example with Robustness Added

Let us return to the messenger program and add changes to make it more robust:

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

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

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

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

server(User_List) ->
    receive
        {From, logon, Name} ->
```

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

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

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

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


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

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

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

logoff() ->
    mess_client ! logoff.
```

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

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

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

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

The following changes are added:

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

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

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

Also, a time-out of five seconds has been introduced in the `await_result` function. That is, if the server does not reply within five seconds (5000 ms), the client terminates. This is only needed in the logon sequence before the client and the server are linked.

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

5.5 Records and Macros

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

5.5.1 The Larger Example Divided into Several Files

To illustrate this, the messenger example from the previous section is divided into the following five files:

- `mess_config.hrl`

  Header file for configuration data
- `mess_interface.hrl`

  Interface definitions between the client and the messenger
- `user_interface.erl`

  Functions for the user interface
- `mess_client.erl`

  Functions for the client side of the messenger
- `mess_server.erl`

  Functions for the server side of the messenger

While doing this, the message passing interface between the shell, the client, and the server is cleaned up and is defined using **records**. Also, **macros** are introduced:

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

logoff() ->
    mess_client ! logoff.

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

```
            From !  #server_reply{message=sent}
    end.

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

5.5.2 Header Files

As shown above, some files have extension `.hrl`. These are header files that are included in the `.erl` files by:

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

for example:

```
-include("mess_interface.hrl").
```

In the case above the file is fetched from the same directory as all the other files in the messenger example. (*manual*).

.hrl files can contain any valid Erlang code but are most often used for record and macro definitions.

5.5.3 Records

A record is defined as:

```
-record(name_of_record,{field_name1, field_name2, field_name3, ......}).
```

For example:

```
-record(message_to,{to_name, message}).
```

This is equivalent to:

```
{message_to, To_Name, Message}
```

Creating a record is best illustrated by an example:

```
#message_to{message="hello", to_name=fred)
```

This creates:

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

Notice that you do not have to worry about the order you assign values to the various parts of the records when you create it. The advantage of using records is that by placing their definitions in header files you can conveniently define interfaces that are easy to change. For example, if you want to add a new field to the record, you only have to change the code where the new field is used and not at every place the record is referred to. If you leave out a field when creating a record, it gets the value of the atom `undefined`. (*manual*)

Pattern matching with records is very similar to creating records. For example, inside a `case` or `receive`:

```
#message_to{to_name=ToName, message=Message} ->
```

This is the same as:

```
{message_to, ToName, Message}
```

5.5.4 Macros

Another thing that has been added to the messenger is a macro. The file `mess_config.hrl` contains the definition:

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

This file is included in `mess_server.erl`:

```
-include("mess_config.hrl").
```

Every occurrence of `?server_node` in `mess_server.erl` is now replaced by `messenger@super`.

A macro is also used when spawning the server process:

```
spawn(?MODULE, server, [])
```

This is a standard macro (that is, defined by the system, not by the user). `?MODULE` is always replaced by the name of the current module (that is, the `-module` definition near the start of the file). There are more advanced ways of using macros with, for example, parameters (*manual*).

The three Erlang (`.erl`) files in the messenger example are individually compiled into object code file (`.beam`). The Erlang system loads and links these files into the system when they are referred to during execution of the code. In this case, they are simply put in our current working directory (that is, the place you have done "cd" to). There are ways of putting the `.beam` files in other directories.

In the messenger example, no assumptions have been made about what the message being sent is. It can be any valid Erlang term.

6 Erlang Reference Manual

6.1 Introduction

This section is the Erlang reference manual. It describes the Erlang programming language.

6.1.1 Purpose

The focus of the Erlang reference manual is on the language itself, not the implementation of it. The language constructs are described in text and with examples rather than formally specified. This is to make the manual more readable. The Erlang reference manual is not intended as a tutorial.

Information about implementation of Erlang can, for example, be found, in the following:

- System Principles

  Starting and stopping, boot scripts, code loading, logging, creating target systems
- Efficiency Guide

  Memory consumption, system limits
- ERTS User's Guide

  Crash dumps, drivers

6.1.2 Prerequisites

It is assumed that the reader has done some programming and is familiar with concepts such as data types and programming language syntax.

6.1.3 Document Conventions

In this section, the following terminology is used:

- A **sequence** is one or more items. For example, a clause body consists of a sequence of expressions. This means that there must be at least one expression.
- A **list** is any number of items. For example, an argument list can consist of zero, one, or more arguments.

If a feature has been added in R13A or later, this is mentioned in the text.

6.1.4 Complete List of BIFs

For a complete list of BIFs, their arguments and return values, see erlang(3) manual page in ERTS.

6.1.5 Reserved Words

The following are reserved words in Erlang:

```
after and andalso band begin bnot bor bsl bsr bxor case catch cond div end fun
if let not of or orelse receive rem try when xor
```

Note: `cond` and `let`, while reserved, are currently not used by the language.

6.2 Character Set and Source File Encoding

6.2.1 Character Set

The syntax of Erlang tokens allow the use of the full ISO-8859-1 (Latin-1) character set. This is noticeable in the following ways:

- All the Latin-1 printable characters can be used and are shown without the escape backslash convention.
- Atoms and variables can use all Latin-1 letters.

Octal	**Decimal**		**Class**
200 - 237	128 - 159		Control characters
240 - 277	160 - 191	- ¿	Punctuation characters
300 - 326	192 - 214	À - Ö	Uppercase letters
327	215	×	Punctuation character
330 - 336	216 - 222	Ø - Þ	Uppercase letters
337 - 366	223 - 246	ß - ö	Lowercase letters
367	247	÷	Punctuation character
370 - 377	248 - 255	ø - ÿ	Lowercase letters

Table 2.1: Character Classes

In Erlang/OTP R16B the syntax of Erlang tokens was extended to handle Unicode. The support was limited to string literals and comments. More about the usage of Unicode in Erlang source files can be found in STDLIB's User's Guide.

From Erlang/OTP 20, atoms and function names are also allowed to contain Unicode characters outside the ISO-Latin-1 range. Module names, application names, and node names are still restricted to the ISO-Latin-1 range.

6.2.2 Source File Encoding

The Erlang source file `encoding` is selected by a comment in one of the first two lines of the source file. The first string that matches the regular expression `coding\s*[:=]\s*([-a-zA-Z0-9])+` selects the encoding. If the matching string is an invalid encoding, it is ignored. The valid encodings are `Latin-1` and `UTF-8`, where the case of the characters can be chosen freely.

The following example selects UTF-8 as default encoding:

```
%% coding: utf-8
```

Two more examples, both selecting Latin-1 as default encoding:

```
%% For this file we have chosen encoding = Latin-1
```

```
%% -*- coding: latin-1 -*-
```

The default encoding for Erlang source files is changed from Latin-1 to UTF-8 since Erlang/OTP 17.0.

6.3 Data Types

Erlang provides a number of data types, which are listed in this section.

Note that Erlang has no user defined types, only composite types (data structures) made of Erlang terms. This means that any function testing for a composite type, typically named `is_type/1`, might return `true` for a term that coincides with the chosen representation. The corresponding functions for built in types do not suffer from this.

6.3.1 Terms

A piece of data of any data type is called a **term**.

6.3.2 Number

There are two types of numeric literals, **integers** and **floats**. Besides the conventional notation, there are two Erlang-specific notations:

- **`$char`**
  ASCII value or unicode code-point of the character **`char`**.
- **`base#value`**
  Integer with the base **`base`**, that must be an integer in the range 2..36.

Leading zeroes are ignored. Single underscore _ can be inserted between digits as a visual separator.

Examples:

```
1> 42.
42
2> -1_234_567_890.
-1234567890
3> $A.
65
4> $\n.
10
5> 2#101.
5
6> 16#1f.
31
7> 16#4865_316F_774F_6C64.
5216630098191412324
8> 2.3.
2.3
9> 2.3e3.
2.3e3
10> 2.3e-3.
0.0023
11> 1_234.333_333
1234.333333
```

Representation of Floating Point Numbers

When working with floats you may not see what you expect when printing or doing arithmetic operations. This is because floats are represented by a fixed number of bits in a base-2 system while printed floats are represented with a base-10 system. Erlang uses 64-bit floats. Here are examples of this phenomenon:

```
> 0.1+0.2.
0.30000000000000004
```

The real numbers `0.1` and `0.2` cannot be represented exactly as floats.

```
> {36028797018963968.0, 36028797018963968 == 36028797018963968.0,
  36028797018963970.0, 36028797018963970 == 36028797018963970.0}.
{3.602879701896397e16, true,
 3.602879701896397e16, false}.
```

The value `36028797018963968` can be represented exactly as a float value but Erlang's pretty printer rounds `36028797018963968.0` to `3.602879701896397e16` (=`36028797018963970.0`) as all values in the range `[36028797018963966.0, 36028797018963972.0]` are represented by `36028797018963968.0`.

For more information about floats and issues with them see:

- **What Every Programmer Should Know About Floating-Point Arithmetic**,
- **0.30000000000000004.com/**, and
- **Floating Point Arithmetic: Issues and Limitations**.

If you need to work with decimal fractions, for instance if you need to represent money, then you should use a library that handles that or work in cents instead of euros so that you do not need decimal fractions.

6.3.3 Atom

An atom is a literal, a constant with name. An atom is to be enclosed in single quotes (') if it does not begin with a lower-case letter or if it contains other characters than alphanumeric characters, underscore (_), or @.

Examples:

```
hello
phone_number
'Monday'
'phone number'
```

6.3.4 Bit Strings and Binaries

A bit string is used to store an area of untyped memory.

Bit strings are expressed using the bit syntax.

Bit strings that consist of a number of bits that are evenly divisible by eight, are called **binaries**

Examples:

```
1> <<10,20>>.
<<10,20>>
2> <<"ABC">>.
<<"ABC">>
1> <<1:1,0:1>>.
<<2:2>>
```

For more examples, see Programming Examples.

6.3.5 Reference

A term that is unique among connected nodes. A reference can be created by calling the `make_ref/0` BIF. The `is_reference/1` BIF can be used to test if a term is a reference.

6.3.6 Fun

A fun is a functional object. Funs make it possible to create an anonymous function and pass the function itself -- not its name -- as argument to other functions.

Example:

```
1> Fun1 = fun (X) -> X+1 end.
#Fun<erl_eval.6.39074546>
2> Fun1(2).
3
```

Read more about funs in Fun Expressions. For more examples, see Programming Examples.

6.3.7 Port Identifier

A port identifier identifies an Erlang port.

`open_port/2`, which is used to create ports, returns a value of this data type.

Read more about ports in Ports and Port Drivers.

6.3.8 PID

PID is an abbreviation for process identifier. Each process has a PID which identifies the process. PIDs are unique among processes that are alive on connected nodes. However, a PID of a terminated process may be reused as a PID for a new process after a while.

The BIF `self/0` returns the PID of the calling process. When creating a new process, the parent process will be able to get the PID of the child process either via the return value, as is the case when calling the `spawn/3` BIF, or via a message, which is the case when calling the `spawn_request/5` BIF. A PID is typically used when when sending a process a signal. The `is_pid/1` BIF can be used to test whether a term is a PID.

Example:

```
-module(m).
-export([loop/0]).

loop() ->
    receive
        who_are_you ->
            io:format("I am ~p~n", [self()]),
            loop()
    end.

1> P = spawn(m, loop, []).
<0.58.0>
2> P ! who_are_you.
I am <0.58.0>
who_are_you
```

Read more about processes in Processes.

6.3.9 Tuple

A tuple is a compound data type with a fixed number of terms:

```
{Term1,...,TermN}
```

Each term `Term` in the tuple is called an **element**. The number of elements is said to be the **size** of the tuple.

There exists a number of BIFs to manipulate tuples.

Examples:

```
1> P = {adam,24,{july,29}}.
{adam,24,{july,29}}
2> element(1,P).
adam
3> element(3,P).
{july,29}
4> P2 = setelement(2,P,25).
{adam,25,{july,29}}
5> tuple_size(P).
3
6> tuple_size({}).
0
```

6.3.10 Map

A map is a compound data type with a variable number of key-value associations:

```
#{Key1=>Value1,...,KeyN=>ValueN}
```

Each key-value association in the map is called an **association pair**. The key and value parts of the pair are called **elements**. The number of association pairs is said to be the **size** of the map.

There exists a number of BIFs to manipulate maps.

Examples:

```
1> M1 = #{name=>adam,age=>24,date=>{july,29}}.
#{age => 24,date => {july,29},name => adam}
2> maps:get(name,M1).
adam
3> maps:get(date,M1).
{july,29}
4> M2 = maps:update(age,25,M1).
#{age => 25,date => {july,29},name => adam}
5> map_size(M).
3
6> map_size(#{}).
0
```

A collection of maps processing functions can be found in `maps` manual page in STDLIB.

Read more about maps in Map Expressions.

Note:

Maps are considered to be experimental during Erlang/OTP R17.

6.3.11 List

A list is a compound data type with a variable number of terms.

```
[Term1,...,TermN]
```

Each term `Term` in the list is called an **element**. The number of elements is said to be the **length** of the list.

Formally, a list is either the empty list `[]` or consists of a **head** (first element) and a **tail** (remainder of the list). The **tail** is also a list. The latter can be expressed as `[H|T]`. The notation `[Term1,...,TermN]` above is equivalent with the list `[Term1|[...|[TermN|[]]]]`.

Example:

`[]` is a list, thus
`[c|[]]` is a list, thus
`[b|[c|[]]]` is a list, thus
`[a|[b|[c|[]]]]` is a list, or in short `[a,b,c]`

A list where the tail is a list is sometimes called a **proper list**. It is allowed to have a list where the tail is not a list, for example, `[a|b]`. However, this type of list is of little practical use.

Examples:

```
1> L1 = [a,2,{c,4}].
[a,2,{c,4}]
2> [H|T] = L1.
[a,2,{c,4}]
3> H.
a
4> T.
[2,{c,4}]
5> L2 = [d|T].
[d,2,{c,4}]
6> length(L1).
3
7> length([]).
0
```

A collection of list processing functions can be found in the lists manual page in STDLIB.

6.3.12 String

Strings are enclosed in double quotes ("), but is not a data type in Erlang. Instead, a string `"hello"` is shorthand for the list `[$h,$e,$l,$l,$o]`, that is, `[104,101,108,108,111]`.

Two adjacent string literals are concatenated into one. This is done in the compilation, thus, does not incur any runtime overhead.

Example:

```
"string" "42"
```

is equivalent to

```
"string42"
```

6.3.13 Record

A record is a data structure for storing a fixed number of elements. It has named fields and is similar to a struct in C. However, a record is not a true data type. Instead, record expressions are translated to tuple expressions during compilation. Therefore, record expressions are not understood by the shell unless special actions are taken. For details, see the shell(3) manual page in STDLIB).

Examples:

```
-module(person).
-export([new/2]).

-record(person, {name, age}).

new(Name, Age) ->
    #person{name=Name, age=Age}.

1> person:new(ernie, 44).
{person,ernie,44}
```

Read more about records in Records. More examples can be found in Programming Examples.

6.3.14 Boolean

There is no Boolean data type in Erlang. Instead the atoms `true` and `false` are used to denote Boolean values.

Examples:

```
1> 2 =< 3.
true
2> true or false.
true
```

6.3.15 Escape Sequences

Within strings and quoted atoms, the following escape sequences are recognized:

Sequence	**Description**
\b	Backspace
\d	Delete
\e	Escape
\f	Form feed
\n	Newline
\r	Carriage return
\s	Space
\t	Tab
\v	Vertical tab

\XYZ, \YZ, \Z	Character with octal representation XYZ, YZ or Z
\xXY	Character with hexadecimal representation XY
\x{X...}	Character with hexadecimal representation; X... is one or more hexadecimal characters
\^a...\^z \^A...\^Z	Control A to control Z
\'	Single quote
\"	Double quote
\\	Backslash

Table 3.1: Recognized Escape Sequences

6.3.16 Type Conversions

There are a number of BIFs for type conversions.

Examples:

```
1> atom_to_list(hello).
"hello"
2> list_to_atom("hello").
hello
3> binary_to_list(<<"hello">>).
"hello"
4> binary_to_list(<<104,101,108,108,111>>).
"hello"
5> list_to_binary("hello").
<<104,101,108,108,111>>
6> float_to_list(7.0).
"7.00000000000000000000e+00"
7> list_to_float("7.000e+00").
7.0
8> integer_to_list(77).
"77"
9> list_to_integer("77").
77
10> tuple_to_list({a,b,c}).
[a,b,c]
11> list_to_tuple([a,b,c]).
{a,b,c}
12> term_to_binary({a,b,c}).
<<131,104,3,100,0,1,97,100,0,1,98,100,0,1,99>>
13> binary_to_term(<<131,104,3,100,0,1,97,100,0,1,98,100,0,1,99>>).
{a,b,c}
14> binary_to_integer(<<"77">>).
77
15> integer_to_binary(77).
<<"77">>
16> float_to_binary(7.0).
<<"7.00000000000000000000e+00">>
17> binary_to_float(<<"7.000e+00">>).
7.0
```

6.4 Pattern Matching

6.4.1 Pattern Matching

Variables are bound to values through the **pattern matching** mechanism. Pattern matching occurs when evaluating a function call, `case`- `receive`- `try`- expressions and match operator (=) expressions.

In a pattern matching, a left-hand side pattern is matched against a right-hand side term. If the matching succeeds, any unbound variables in the pattern become bound. If the matching fails, a run-time error occurs.

Examples:

```
1> X.
** 1:1: variable 'X' is unbound **
2> X = 2.
2
3> X + 1.
3
4> {X, Y} = {1, 2}.
** exception error: no match of right hand side value {1,2}
5> {X, Y} = {2, 3}.
{2,3}
6> Y.
3
```

6.5 Modules

6.5.1 Module Syntax

Erlang code is divided into **modules**. A module consists of a sequence of attributes and function declarations, each terminated by period (.).

Example:

```
-module(m).          % module attribute
-export([fact/1]).   % module attribute

fact(N) when N>0 ->  % beginning of function declaration
    N * fact(N-1);   %  |
fact(0) ->           %  |
    1.               % end of function declaration
```

For a description of function declarations, see Function Declaration Syntax.

6.5.2 Module Attributes

A **module attribute** defines a certain property of a module.

A module attribute consists of a tag and a value:

```
-Tag(Value).
```

`Tag` must be an atom, while `Value` must be a literal term. As a convenience in user-defined attributes, if the literal term `Value` has the syntax `Name/Arity` (where `Name` is an atom and `Arity` a positive integer), the term `Name/Arity` is translated to `{Name,Arity}`.

Any module attribute can be specified. The attributes are stored in the compiled code and can be retrieved by calling `Module:module_info(attributes)`, or by using the module beam_lib(3) in STDLIB.

Several module attributes have predefined meanings. Some of them have arity two, but user-defined module attributes must have arity one.

Pre-Defined Module Attributes

Pre-defined module attributes is to be placed before any function declaration.

`-module(Module).`

Module declaration, defining the name of the module. The name `Module`, an atom, is to be same as the file name minus the extension `.erl`. Otherwise code loading does not work as intended.

This attribute is to be specified first and is the only mandatory attribute.

`-export(Functions).`

Exported functions. Specifies which of the functions, defined within the module, that are visible from outside the module.

`Functions` is a list `[Name1/Arity1, ..., NameN/ArityN]`, where each `NameI` is an atom and `ArityI` an integer.

`-import(Module,Functions).`

Imported functions. Can be called the same way as local functions, that is, without any module prefix.

`Module`, an atom, specifies which module to import functions from. `Functions` is a list similar as for `export`.

`-compile(Options).`

Compiler options. `Options` is a single option or a list of options. This attribute is added to the option list when compiling the module. See the compile(3) manual page in Compiler.

`-vsn(Vsn).`

Module version. `Vsn` is any literal term and can be retrieved using `beam_lib:version/1`, see the beam_lib(3) manual page in STDLIB.

If this attribute is not specified, the version defaults to the MD5 checksum of the module.

`-on_load(Function).`

This attribute names a function that is to be run automatically when a module is loaded. For more information, see Running a Function When a Module is Loaded.

`-nifs(Functions).`

Specifies which of the functions, defined within the module, that may be loaded as NIFs with `erlang:load_nif/2`.

`Functions` is a list `[Name1/Arity1, ..., NameN/ArityN]`, where each `NameI` is an atom and `ArityI` an integer.

> **Note:**
>
> The `-nifs()` attribute was introduced in OTP 25.0. For older Erlang source code without it, any functions in the module may be loaded as NIFs. However, it is recommended to declare the NIFs with the `-nifs` attribute. This allows the compiler to make better decisions regarding optimizations for example.
>
> There is no need to add `-nifs([])` in modules that do not load NIFs. The lack of any call to `erlang:load_nif/2`, from within the module, is enough for the compiler to draw the same conclusion.

Behaviour Module Attribute

It is possible to specify that the module is the callback module for a **behaviour**:

```
-behaviour(Behaviour).
```

The atom `Behaviour` gives the name of the behaviour, which can be a user-defined behaviour or one of the following OTP standard behaviours:

- `gen_server`
- `gen_statem`
- `gen_event`
- `supervisor`

The spelling `behavior` is also accepted.

The callback functions of the module can be specified either directly by the exported function `behaviour_info/1`:

```
behaviour_info(callbacks) -> Callbacks.
```

or by a `-callback` attribute for each callback function:

```
-callback Name(Arguments) -> Result.
```

Here, `Arguments` is a list of zero or more arguments. The `-callback` attribute is to be preferred since the extra type information can be used by tools to produce documentation or find discrepancies.

Read more about behaviours and callback modules in OTP Design Principles.

Record Definitions

The same syntax as for module attributes is used for record definitions:

```
-record(Record,Fields).
```

Record definitions are allowed anywhere in a module, also among the function declarations. Read more in Records.

Preprocessor

The same syntax as for module attributes is used by the preprocessor, which supports file inclusion, macros, and conditional compilation:

```
-include("SomeFile.hrl").
-define(Macro,Replacement).
```

Read more in Preprocessor.

Setting File and Line

The same syntax as for module attributes is used for changing the pre-defined macros `?FILE` and `?LINE`:

```
-file(File, Line).
```

This attribute is used by tools, such as Yecc, to inform the compiler that the source program is generated by another tool. It also indicates the correspondence of source files to lines of the original user-written file, from which the source program is produced.

Types and function specifications

A similar syntax as for module attributes is used for specifying types and function specifications:

```
-type my_type() :: atom() | integer().
-spec my_function(integer()) -> integer().
```

Read more in Types and Function specifications.

The description is based on **EEP8 - Types and function specifications**, which is not to be further updated.

6.5.3 The feature directive

While not a module attribute, but rather a directive (since it might affect syntax), there is the `-feature(..)` directive used for enabling and disabling features.

The syntax is similar to that of an attribute, but has two arguments:

```
-feature(FeatureName, enable | disable).
```

Note that the feature directive can only appear in a prefix of the module.

6.5.4 Comments

Comments can be placed anywhere in a module except within strings and quoted atoms. A comment begins with the character "%", continues up to, but does not include the next end-of-line, and has no effect. Notice that the terminating end-of-line has the effect of white space.

6.5.5 module_info/0 and module_info/1 functions

The compiler automatically inserts the two special, exported functions into each module:

- `Module:module_info/0`
- `Module:module_info/1`

These functions can be called to retrieve information about the module.

module_info/0

The `module_info/0` function in each module, returns a list of `{Key,Value}` tuples with information about the module. Currently, the list contain tuples with the following `Key`s: `module`, `attributes`, `compile`, `exports`, `md5` and `native`. The order and number of tuples may change without prior notice.

module_info/1

The call `module_info(Key)`, where `Key` is an atom, returns a single piece of information about the module.

The following values are allowed for `Key`:

`module`

Returns an atom representing the module name.

`attributes`

Returns a list of `{AttributeName,ValueList}` tuples, where `AttributeName` is the name of an attribute, and `ValueList` is a list of values. Notice that a given attribute can occur more than once in the list with different values if the attribute occurs more than once in the module.

The list of attributes becomes empty if the module is stripped with the beam_lib(3) module (in STDLIB).

`compile`

Returns a list of tuples with information about how the module was compiled. This list is empty if the module has been stripped with the beam_lib(3) module (in STDLIB).

`md5`

Returns a binary representing the MD5 checksum of the module. If the module has native code loaded, this will be the MD5 of the native code, not the BEAM bytecode.

`exports`

Returns a list of `{Name,Arity}` tuples with all exported functions in the module.

`functions`

Returns a list of `{Name,Arity}` tuples with all functions in the module.

`nifs`

Returns a list of `{Name,Arity}` tuples with all NIF functions in the module.

`native`

Return `true` if the module has native compiled code. Return `false` otherwise. In a system compiled without HiPE support, the result is always `false`

6.6 Functions

6.6.1 Function Declaration Syntax

A **function declaration** is a sequence of function clauses separated by semicolons, and terminated by period (.).

A **function clause** consists of a clause head and a clause body, separated by `->`.

A clause **head** consists of the function name, an argument list, and an optional guard sequence beginning with the keyword `when`:

```
Name(Pattern11,...,Pattern1N) [when GuardSeq1] ->
    Body1;
...;
Name(PatternK1,...,PatternKN) [when GuardSeqK] ->
    BodyK.
```

The function name is an atom. Each argument is a pattern.

The number of arguments `N` is the **arity** of the function. A function is uniquely defined by the module name, function name, and arity. That is, two functions with the same name and in the same module, but with different arities are two different functions.

A function named `f` in the module `m` and with arity `N` is often denoted as `m:f/N`.

A clause **body** consists of a sequence of expressions separated by comma (,):

```
Expr1,
...,
ExprN
```

Valid Erlang expressions and guard sequences are described in Expressions.

Example:

```
fact(N) when N>0 ->  % first clause head
    N * fact(N-1);   % first clause body

fact(0) ->           % second clause head
    1.               % second clause body
```

6.6.2 Function Evaluation

When a function `m:f/N` is called, first the code for the function is located. If the function cannot be found, an `undef` runtime error occurs. Notice that the function must be exported to be visible outside the module it is defined in.

If the function is found, the function clauses are scanned sequentially until a clause is found that fulfills both of the following two conditions:

- The patterns in the clause head can be successfully matched against the given arguments.
- The guard sequence, if any, is true.

If such a clause cannot be found, a `function_clause` runtime error occurs.

If such a clause is found, the corresponding clause body is evaluated. That is, the expressions in the body are evaluated sequentially and the value of the last expression is returned.

Consider the function `fact`:

```
-module(m).
-export([fact/1]).

fact(N) when N>0 ->
    N * fact(N-1);
fact(0) ->
    1.
```

Assume that you want to calculate the factorial for 1:

```
1> m:fact(1).
```

Evaluation starts at the first clause. The pattern `N` is matched against argument 1. The matching succeeds and the guard (`N>0`) is true, thus `N` is bound to 1, and the corresponding body is evaluated:

```
N * fact(N-1) => (N is bound to 1)
1 * fact(0)
```

Now, `fact(0)` is called, and the function clauses are scanned sequentially again. First, the pattern `N` is matched against 0. The matching succeeds, but the guard (`N>0`) is false. Second, the pattern 0 is matched against 0. The matching succeeds and the body is evaluated:

```
1 * fact(0) =>
1 * 1 =>
1
```

Evaluation has succeed and `m:fact(1)` returns 1.

If `m:fact/1` is called with a negative number as argument, no clause head matches. A `function_clause` runtime error occurs.

6.6.3 Tail recursion

If the last expression of a function body is a function call, a **tail recursive** call is done. This is to ensure that no system resources, for example, call stack, are consumed. This means that an infinite loop can be done if it uses tail-recursive calls.

Example:

```
loop(N) ->
    io:format("~w~n", [N]),
    loop(N+1).
```

The earlier factorial example can act as a counter-example. It is not tail-recursive, since a multiplication is done on the result of the recursive call to `fact(N-1)`.

6.6.4 Built-In Functions (BIFs)

BIFs are implemented in C code in the runtime system. BIFs do things that are difficult or impossible to implement in Erlang. Most of the BIFs belong to the module `erlang` but there are also BIFs belonging to a few other modules, for example `lists` and `ets`.

The most commonly used BIFs belonging to `erlang(3)` are **auto-imported**. They do not need to be prefixed with the module name. Which BIFs that are auto-imported is specified in the erlang(3) module in ERTS. For example, standard-type conversion BIFs like `atom_to_list` and BIFs allowed in guards can be called without specifying the module name.

Examples:

```
1> tuple_size({a,b,c}).
3
2> atom_to_list('Erlang').
"Erlang"
```

Notice that it is normally the set of auto-imported BIFs that are referred to when talking about 'BIFs'.

6.7 Types and Function Specifications

6.7.1 The Erlang Type Language

Erlang is a dynamically typed language. Still, it comes with a notation for declaring sets of Erlang terms to form a particular type. This effectively forms specific subtypes of the set of all Erlang terms.

Subsequently, these types can be used to specify types of record fields and also the argument and return types of functions.

Type information can be used for the following:

- To document function interfaces
- To provide more information for bug detection tools, such as Dialyzer
- To be exploited by documentation tools, such as EDoc, for generating program documentation of various forms

It is expected that the type language described in this section supersedes and replaces the purely comment-based `@type` and `@spec` declarations used by EDoc.

6.7.2 Types and their Syntax

Types describe sets of Erlang terms. Types consist of, and are built from, a set of predefined types, for example, `integer()`, `atom()`, and `pid()`. Predefined types represent a typically infinite set of Erlang terms that belong to this type. For example, the type `atom()` denotes the set of all Erlang atoms.

For integers and atoms, it is allowed for singleton types; for example, the integers `-1` and `42`, or the atoms `'foo'` and `'bar'`. All other types are built using unions of either predefined types or singleton types. In a type union between a type and one of its subtypes, the subtype is absorbed by the supertype. Thus, the union is then treated as if the subtype was not a constituent of the union. For example, the type union:

```
atom() | 'bar' | integer() | 42
```

describes the same set of terms as the type union:

```
atom() | integer()
```

Because of subtype relations that exist between types, types form a lattice where the top-most element, `any()`, denotes the set of all Erlang terms and the bottom-most element, `none()`, denotes the empty set of terms.

The set of predefined types and the syntax for types follows:

```
Type :: any()                 %% The top type, the set of all Erlang terms
      | none()                %% The bottom type, contains no terms
      | pid()
      | port()
      | reference()
      | []                    %% nil
      | Atom
      | Bitstring
      | float()
      | Fun
      | Integer
      | List
      | Map
      | Tuple
      | Union
      | UserDefined           %% described in Type Declarations of User-Defined Types

Atom :: atom()
      | Erlang_Atom           %% 'foo', 'bar', ...

Bitstring :: <<>>
           | <<_:M>>          %% M is an Integer_Value that evaluates to a positive integer
           | <<_:_*N>>        %% N is an Integer_Value that evaluates to a positive integer
           | <<_:M, _:_*N>>

Fun :: fun()                  %% any function
     | fun((...) -> Type)     %% any arity, returning Type
     | fun(() -> Type)
     | fun((TList) -> Type)

Integer :: integer()
         | Integer_Value
         | Integer_Value..Integer_Value      %% specifies an integer range

Integer_Value :: Erlang_Integer              %% ..., -1, 0, 1, ... 42 ...
               | Erlang_Character            %% $a, $b ...
               | Integer_Value BinaryOp Integer_Value
               | UnaryOp Integer_Value

BinaryOp :: '*' | 'div' | 'rem' | 'band' | '+' | '-' | 'bor' | 'bxor' | 'bsl' | 'bsr'

UnaryOp :: '+' | '-' | 'bnot'

List :: list(Type)                           %% Proper list ([]-terminated)
      | maybe_improper_list(Type1, Type2)    %% Type1=contents, Type2=termination
      | nonempty_improper_list(Type1, Type2) %% Type1 and Type2 as above
      | nonempty_list(Type)                  %% Proper non-empty list

Map :: #{}                                   %% denotes the empty map
     | #{AssociationList}

Tuple :: tuple()                             %% denotes a tuple of any size
       | {}
       | {TList}

AssociationList :: Association
                 | Association, AssociationList

Association :: Type := Type                  %% denotes a mandatory association
             | Type => Type                  %% denotes an optional association

TList :: Type
       | Type, TList
```

```
Union :: Type1 | Type2
```

Integer values are either integer or character literals or expressions consisting of possibly nested unary or binary operations that evaluate to an integer. Such expressions can also be used in bit strings and ranges.

The general form of bit strings is `<<_:M, _:_*N>>`, where `M` and `N` must evaluate to positive integers. It denotes a bit string that is `M + (k*N)` bits long (that is, a bit string that starts with `M` bits and continues with `k` segments of `N` bits each, where `k` is also a positive integer). The notations `<<_:_*N>>`, `<<_:M>>`, and `<<>>` are convenient shorthands for the cases that `M` or `N`, or both, are zero.

Because lists are commonly used, they have shorthand type notations. The types `list(T)` and `nonempty_list(T)` have the shorthands `[T]` and `[T,...]`, respectively. The only difference between the two shorthands is that `[T]` can be an empty list but `[T,...]` cannot.

Notice that the shorthand for `list()`, that is, the list of elements of unknown type, is `[_]` (or `[any()]`), not `[]`. The notation `[]` specifies the singleton type for the empty list.

The general form of map types is `#{AssociationList}`. The key types in `AssociationList` are allowed to overlap, and if they do, the leftmost association takes precedence. A map association has a key in `AssociationList` if it belongs to this type. `AssociationList` can contain both mandatory `(:=)` and optional `(=>)` association types. If an association type is mandatory, an association with that type needs to be present. In the case of an optional association type it is not required for the key type to be present.

The notation `#{}` specifies the singleton type for the empty map. Note that this notation is not a shorthand for the `map()` type.

For convenience, the following types are also built-in. They can be thought as predefined aliases for the type unions also shown in the table.

Built-in type	**Defined as**
`term()`	`any()`
`binary()`	`<<_:_*8>>`
`nonempty_binary()`	`<<_:8, _:_*8>>`
`bitstring()`	`<<_:_*1>>`
`nonempty_bitstring()`	`<<_:1, _:_*1>>`
`boolean()`	`'false' | 'true'`
`byte()`	`0..255`
`char()`	`0..16#10ffff`
`nil()`	`[]`
`number()`	`integer() | float()`
`list()`	`[any()]`
`maybe_improper_list()`	`maybe_improper_list(any(), any())`
`nonempty_list()`	`nonempty_list(any())`

`string()`	`[char()]`
`nonempty_string()`	`[char(),...]`
`iodata()`	`iolist() | binary()`
`iolist()`	`maybe_improper_list(byte() | binary() | iolist(), binary() | [])`
`map()`	`#{any() => any()}`
`function()`	`fun()`
`module()`	`atom()`
`mfa()`	`{module(),atom(),arity()}`
`arity()`	`0..255`
`identifier()`	`pid() | port() | reference()`
`node()`	`atom()`
`timeout()`	`'infinity' | non_neg_integer()`
`no_return()`	`none()`

Table 7.1: Built-in types, predefined aliases

In addition, the following three built-in types exist and can be thought as defined below, though strictly their "type definition" is not valid syntax according to the type language defined above.

Built-in type	**Can be thought defined by the syntax**
`non_neg_integer()`	`0..`
`pos_integer()`	`1..`
`neg_integer()`	`..-1`

Table 7.2: Additional built-in types

Users are not allowed to define types with the same names as the predefined or built-in ones. This is checked by the compiler and its violation results in a compilation error.

Note:

The following built-in list types also exist, but they are expected to be rarely used. Hence, they have long names:

```
nonempty_maybe_improper_list() :: nonempty_maybe_improper_list(any(), any())
nonempty_improper_list(Type1, Type2)
nonempty_maybe_improper_list(Type1, Type2)
```

where the last two types define the set of Erlang terms one would expect.

Also for convenience, record notation is allowed to be used. Records are shorthands for the corresponding tuples:

```
Record :: #Erlang_Atom{}
        | #Erlang_Atom{Fields}
```

Records are extended to possibly contain type information. This is described in Type Information in Record Declarations.

6.7.3 Type Declarations of User-Defined Types

As seen, the basic syntax of a type is an atom followed by closed parentheses. New types are declared using `-type` and `-opaque` attributes as in the following:

```
-type my_struct_type() :: Type.
-opaque my_opaq_type() :: Type.
```

The type name is the atom `my_struct_type`, followed by parentheses. `Type` is a type as defined in the previous section. A current restriction is that `Type` can contain only predefined types, or user-defined types which are either of the following:

- Module-local type, that is, with a definition that is present in the code of the module
- Remote type, that is, type defined in, and exported by, other modules; more about this soon.

For module-local types, the restriction that their definition exists in the module is enforced by the compiler and results in a compilation error. (A similar restriction currently exists for records.)

Type declarations can also be parameterized by including type variables between the parentheses. The syntax of type variables is the same as Erlang variables, that is, starts with an upper-case letter. Naturally, these variables can - and is to - appear on the RHS of the definition. A concrete example follows:

```
-type orddict(Key, Val) :: [{Key, Val}].
```

A module can export some types to declare that other modules are allowed to refer to them as **remote types**. This declaration has the following form:

```
-export_type([T1/A1, ..., Tk/Ak]).
```

Here the Ti's are atoms (the name of the type) and the Ai's are their arguments

Example:

```
-export_type([my_struct_type/0, orddict/2]).
```

Assuming that these types are exported from module `'mod'`, you can refer to them from other modules using remote type expressions like the following:

```
mod:my_struct_type()
mod:orddict(atom(), term())
```

It is not allowed to refer to types that are not declared as exported.

Types declared as `opaque` represent sets of terms whose structure is not supposed to be visible from outside of their defining module. That is, only the module defining them is allowed to depend on their term structure. Consequently, such types do not make much sense as module local - module local types are not accessible by other modules anyway - and is always to be exported.

Read more on Opaques

6.7.4 Type Information in Record Declarations

The types of record fields can be specified in the declaration of the record. The syntax for this is as follows:

```
-record(rec, {field1 :: Type1, field2, field3 :: Type3}).
```

For fields without type annotations, their type defaults to any(). That is, the previous example is a shorthand for the following:

```
-record(rec, {field1 :: Type1, field2 :: any(), field3 :: Type3}).
```

In the presence of initial values for fields, the type must be declared after the initialization, as follows:

```
-record(rec, {field1 = [] :: Type1, field2, field3 = 42 :: Type3}).
```

The initial values for fields are to be compatible with (that is, a member of) the corresponding types. This is checked by the compiler and results in a compilation error if a violation is detected.

Note:

Before Erlang/OTP 19, for fields without initial values, the singleton type `'undefined'` was added to all declared types. In other words, the following two record declarations had identical effects:

```
-record(rec, {f1 = 42 :: integer(),
              f2      :: float(),
              f3      :: 'a' | 'b'}).

-record(rec, {f1 = 42 :: integer(),
              f2      :: 'undefined' | float(),
              f3      :: 'undefined' | 'a' | 'b'}).
```

This is no longer the case. If you require `'undefined'` in your record field type, you must explicitly add it to the typespec, as in the 2nd example.

Any record, containing type information or not, once defined, can be used as a type using the following syntax:

```
#rec{}
```

In addition, the record fields can be further specified when using a record type by adding type information about the field as follows:

```
#rec{some_field :: Type}
```

Any unspecified fields are assumed to have the type in the original record declaration.

Note:

When records are used to create patterns for ETS and Mnesia match functions, Dialyzer may need some help not to emit bad warnings. For example:

```
   -type height() :: pos_integer().
   -record(person, {name :: string(), height :: height()}).

   lookup(Name, Tab) ->
ets:match_object(Tab, #person{name = Name, _ = '_'}).
```

Dialyzer will emit a warning since `'_'` is not in the type of record field `height`.

The recommended way of dealing with this is to declare the smallest record field types to accommodate all your needs, and then create refinements as needed. The modified example:

```
   -record(person, {name :: string(), height :: height() | '_'}).

   -type person() :: #person{height :: height()}.
```

In specifications and type declarations the type `person()` is to be preferred before `#person{}`.

6.7.5 Specifications for Functions

A specification (or contract) for a function is given using the `-spec` attribute. The general format is as follows:

```
-spec Function(ArgType1, ..., ArgTypeN) -> ReturnType.
```

An implementation of the function with the same name `Function` must exist in the current module, and the arity of the function must match the number of arguments, else a compilation error occurs.

The following longer format with module name is also valid as long as `Module` is the name of the current module. This can be useful for documentation purposes.

```
-spec Module:Function(ArgType1, ..., ArgTypeN) -> ReturnType.
```

Also, for documentation purposes, argument names can be given:

```
-spec Function(ArgName1 :: Type1, ..., ArgNameN :: TypeN) -> RT.
```

A function specification can be overloaded. That is, it can have several types, separated by a semicolon (`;`):

```
-spec foo(T1, T2) -> T3
       ; (T4, T5) -> T6.
```

A current restriction, which currently results in a warning by Dialyzer, is that the domains of the argument types cannot overlap. For example, the following specification results in a warning:

```
-spec foo(pos_integer()) -> pos_integer()
       ; (integer()) -> integer().
```

Type variables can be used in specifications to specify relations for the input and output arguments of a function. For example, the following specification defines the type of a polymorphic identity function:

```
-spec id(X) -> X.
```

Notice that the above specification does not restrict the input and output type in any way. These types can be constrained by guard-like subtype constraints and provide bounded quantification:

```
-spec id(X) -> X when X :: tuple().
```

Currently, the `::` constraint (read as «is a subtype of») is the only guard constraint that can be used in the `when` part of a `-spec` attribute.

Note:

The above function specification uses multiple occurrences of the same type variable. That provides more type information than the following function specification, where the type variables are missing:

```
-spec id(tuple()) -> tuple().
```

The latter specification says that the function takes some tuple and returns some tuple. The specification with the `X` type variable specifies that the function takes a tuple and returns **the same** tuple.

However, it is up to the tools that process the specifications to choose whether to take this extra information into account or not.

The scope of a `::` constraint is the `(...) -> RetType` specification after which it appears. To avoid confusion, it is suggested that different variables are used in different constituents of an overloaded contract, as shown in the following example:

```
-spec foo({X, integer()}) -> X when X :: atom()
      ; ([Y]) -> Y when Y :: number().
```

Some functions in Erlang are not meant to return; either because they define servers or because they are used to throw exceptions, as in the following function:

```
my_error(Err) -> erlang:throw({error, Err}).
```

For such functions, it is recommended to use the special `no_return()` type for their "return", through a contract of the following form:

```
-spec my_error(term()) -> no_return().
```

Note:

Erlang uses the shorthand version _ as an anonymous type variable equivalent to `term()` or `any()`. For example, the following function

```
-spec Function(string(), _) -> string().
```

is equivalent to:

```
-spec Function(string(), any()) -> string().
```

6.8 Opaques

6.8.1 Opaque Type Aliases

The main use case for opacity in Erlang is to hide the implementation of a data type, enabling evolving the API while minimizing the risk of breaking consumers. The runtime does not check opacity. Dialyzer provides some opacity-checking, but the rest is up to convention.

This document explains what Erlang opacity is (and the trade-offs involved) via the example of OTP's `sets:set()` data type. This type **was** defined in `sets` module like this:

```
-opaque set(Element) :: #set{segs :: segs(Element)}.
```

OTP 24 changed the definition to the following, in **this commit**

```
-opaque set(Element) :: #set{segs :: segs(Element)} | #{Element => ?VALUE}.
```

And this change was safer and more backwards-compatible than if the type had been defined with `-type` instead of `-opaque` . Here's why: when a module defines an `-opaque` , the contract is that only the defining module should rely on the definition of the type: no other modules should rely on the definition.

This means that code that pattern-matched on `set` as a record/tuple technically broke the contract, and opted in to being potentially broken when the definition of `set()` changed. Before OTP 24, this code printed `ok` . In OTP 24 it may error:

```
case sets:new() of
    Set when is_tuple(Set) ->
        io:format("ok")
end.
```

When working with an opaque defined in another module, here are some recommendations:

- Don't examine the underlying type using pattern-matching, guards, or functions that reveal the type, such as `tuple_size/1` .
- Instead, use functions provided by the module for working with the type. For example, `sets` module provides `sets:new/0` , `sets:add/2` , `sets:is_element/2` , etc.
- `sets:set(a)` is a subtype of `sets:set(a | b)` and not the other way around. Generally, you can rely on the property that `the_opaque(T)` is a subtype of `the_opaque(U)` when T is a subtype of U.

When defining your own opaques, here are some recommendations:

- Since consumers are expected to not rely on the definition of the opaque type, you must provide functions for constructing and querying/deconstructing intances of your opaque type. For example, sets can be constructed with `sets:new/0`, `sets:from_list/1`, `sets:add/2`, queried with `sets:is_element/2`, and deconstructed with`sets:to_list/1`.
- Don't define an opaque with a type variable in parameter position. This breaks the normal and expected behavior that (for example) `my_type(a)` is a subtype of `my_type(a | b)`
- Add specs to exported functions that use the opaque type

Note that opaques can be harder to work with for consumers, since the consumer is expected not to pattern-match and must instead use functions that the author of the opaque type provides to use instances of the type.

Also, opacity in Erlang is skin-deep: the runtime does not enforce opacity-checking. So now that sets are implemented in terms of maps, an `is_map` check on a set **will** pass. The opacity rules are only enforced by convention and by additional tooling such as Dialyzer. And this enforcement is not total: For example, determined consumer of `sets` can still do things that reveal the structure of the set, such as by printing, serializing, or using a set as `term()` and then inspecting via functions like `is_map` or `maps:get/2` . And Dialyzer must make some **approximations** . Opacity checking has limitations, but is still a vital tool in scalable Erlang development.

6.9 Expressions

In this section, all valid Erlang expressions are listed. When writing Erlang programs, it is also allowed to use macro- and record expressions. However, these expressions are expanded during compilation and are in that sense not true Erlang expressions. Macro- and record expressions are covered in separate sections:

- Preprocessor
- Records

6.9.1 Expression Evaluation

All subexpressions are evaluated before an expression itself is evaluated, unless explicitly stated otherwise. For example, consider the expression:

```
Expr1 + Expr2
```

`Expr1` and `Expr2`, which are also expressions, are evaluated first - in any order - before the addition is performed.

Many of the operators can only be applied to arguments of a certain type. For example, arithmetic operators can only be applied to numbers. An argument of the wrong type causes a `badarg` runtime error.

6.9.2 Terms

The simplest form of expression is a term, that is an integer, float, atom, string, list, map, or tuple. The return value is the term itself.

6.9.3 Variables

A variable is an expression. If a variable is bound to a value, the return value is this value. Unbound variables are only allowed in patterns.

Variables start with an uppercase letter or underscore (_). Variables can contain alphanumeric characters, underscore and @.

Examples:

```
X
Name1
PhoneNumber
Phone_number

_
_Height
```

Variables are bound to values using pattern matching. Erlang uses **single assignment**, that is, a variable can only be bound once.

The **anonymous variable** is denoted by underscore (_) and can be used when a variable is required but its value can be ignored.

Example:

```
[H|_] = [1,2,3]
```

Variables starting with underscore (_), for example, `_Height`, are normal variables, not anonymous. However, they are ignored by the compiler in the sense that they do not generate warnings.

Example:

The following code:

```
member(_, []) ->
    [].
```

can be rewritten to be more readable:

```
member(Elem, []) ->
    [].
```

This causes a warning for an unused variable, `Elem`, if the code is compiled with the flag `warn_unused_vars` set. Instead, the code can be rewritten to:

```
member(_Elem, []) ->
    [].
```

Notice that since variables starting with an underscore are not anonymous, this matches:

```
{_,_} = {1,2}
```

But this fails:

```
{_N,_N} = {1,2}
```

The scope for a variable is its function clause. Variables bound in a branch of an `if`, `case`, or `receive` expression must be bound in all branches to have a value outside the expression. Otherwise they are regarded as 'unsafe' outside the expression.

For the `try` expression variable scoping is limited so that variables bound in the expression are always 'unsafe' outside the expression.

6.9.4 Patterns

A pattern has the same structure as a term but can contain unbound variables.

Example:

```
Name1
[H|T]
{error,Reason}
```

Patterns are allowed in clause heads, `case` and `receive` expressions, and match expressions.

Match Operator = in Patterns

If `Pattern1` and `Pattern2` are valid patterns, the following is also a valid pattern:

```
Pattern1 = Pattern2
```

When matched against a term, both `Pattern1` and `Pattern2` are matched against the term. The idea behind this feature is to avoid reconstruction of terms.

Example:

```
f({connect,From,To,Number,Options}, To) ->
    Signal = {connect,From,To,Number,Options},
    ...;
f(Signal, To) ->
    ignore.
```

can instead be written as

```
f({connect,_,To,_,_} = Signal, To) ->
    ...;
f(Signal, To) ->
    ignore.
```

String Prefix in Patterns

When matching strings, the following is a valid pattern:

```
f("prefix" ++ Str) -> ...
```

This is syntactic sugar for the equivalent, but harder to read:

```
f([$p,$r,$e,$f,$i,$x | Str]) -> ...
```

Expressions in Patterns

An arithmetic expression can be used within a pattern if it meets both of the following two conditions:

- It uses only numeric or bitwise operators.
- Its value can be evaluated to a constant when complied.

Example:

```
case {Value, Result} of
    {?THRESHOLD+1, ok} -> ...
```

6.9.5 Match

The following matches `Expr1`, a pattern, against `Expr2`:

```
Expr1 = Expr2
```

If the matching succeeds, any unbound variable in the pattern becomes bound and the value of `Expr2` is returned.

If the matching fails, a `badmatch` run-time error occurs.

Examples:

```
1> {A, B} = {answer, 42}.
{answer,42}
2> A.
answer
3> {C, D} = [1, 2].
** exception error: no match of right-hand side value [1,2]
```

6.9.6 Function Calls

```
ExprF(Expr1,...,ExprN)
ExprM:ExprF(Expr1,...,ExprN)
```

In the first form of function calls, `ExprM:ExprF(Expr1,...,ExprN)`, each of `ExprM` and `ExprF` must be an atom or an expression that evaluates to an atom. The function is said to be called by using the **fully qualified function name**. This is often referred to as a **remote** or **external function call**.

Example:

```
lists:keysearch(Name, 1, List)
```

In the second form of function calls, `ExprF(Expr1,...,ExprN)`, `ExprF` must be an atom or evaluate to a fun.

If `ExprF` is an atom, the function is said to be called by using the **implicitly qualified function name**. If the function `ExprF` is locally defined, it is called. Alternatively, if `ExprF` is explicitly imported from the `M` module, `M:ExprF(Expr1,...,ExprN)` is called. If `ExprF` is neither declared locally nor explicitly imported, `ExprF` must be the name of an automatically imported BIF.

Examples:

```
handle(Msg, State)
spawn(m, init, [])
```

Examples where `ExprF` is a fun:

```
1> Fun1 = fun(X) -> X+1 end,
Fun1(3).
4
2> fun lists:append/2([1,2], [3,4]).
[1,2,3,4]
3>
```

Notice that when calling a local function, there is a difference between using the implicitly or fully qualified function name. The latter always refers to the latest version of the module. See Compilation and Code Loading and Function Evaluation.

Local Function Names Clashing With Auto-Imported BIFs

If a local function has the same name as an auto-imported BIF, the semantics is that implicitly qualified function calls are directed to the locally defined function, not to the BIF. To avoid confusion, there is a compiler directive available, `-compile({no_auto_import,[F/A]})`, that makes a BIF not being auto-imported. In certain situations, such a compile-directive is mandatory.

Warning:

Before OTP R14A (ERTS version 5.8), an implicitly qualified function call to a function having the same name as an auto-imported BIF always resulted in the BIF being called. In newer versions of the compiler, the local function is called instead. This is to avoid that future additions to the set of auto-imported BIFs do not silently change the behavior of old code.

However, to avoid that old (pre R14) code changed its behavior when compiled with OTP version R14A or later, the following restriction applies: If you override the name of a BIF that was auto-imported in OTP versions prior to R14A (ERTS version 5.8) and have an implicitly qualified call to that function in your code, you either need to explicitly remove the auto-import using a compiler directive, or replace the call with a fully qualified function call. Otherwise you get a compilation error. See the following example:

```
-export([length/1,f/1]).

-compile({no_auto_import,[length/1]}). % erlang:length/1 no longer autoimported

length([]) ->
    0;
length([H|T]) ->
    1 + length(T). %% Calls the local function length/1

f(X) when erlang:length(X) > 3 -> %% Calls erlang:length/1,
                                  %% which is allowed in guards
    long.
```

The same logic applies to explicitly imported functions from other modules, as to locally defined functions. It is not allowed to both import a function from another module and have the function declared in the module at the same time:

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

-compile({no_auto_import,[length/1]}). % erlang:length/1 no longer autoimported

-import(mod,[length/1]).

f(X) when erlang:length(X) > 33 -> %% Calls erlang:length/1,
                                   %% which is allowed in guards

    erlang:length(X);              %% Explicit call to erlang:length in body

f(X) ->
    length(X).                     %% mod:length/1 is called
```

For auto-imported BIFs added in Erlang/OTP R14A and thereafter, overriding the name with a local function or explicit import is always allowed. However, if the `-compile({no_auto_import,[F/A])` directive is not used, the compiler issues a warning whenever the function is called in the module using the implicitly qualified function name.

6.9.7 If

```

if
    GuardSeq1 ->
        Body1;
    ...;
    GuardSeqN ->
        BodyN
end
```

The branches of an `if`-expression are scanned sequentially until a guard sequence `GuardSeq` that evaluates to true is found. Then the corresponding `Body` (sequence of expressions separated by ',') is evaluated.

The return value of `Body` is the return value of the `if` expression.

If no guard sequence is evaluated as true, an `if_clause` run-time error occurs. If necessary, the guard expression `true` can be used in the last branch, as that guard sequence is always true.

Example:

```
is_greater_than(X, Y) ->
    if
        X>Y ->
            true;
        true -> % works as an 'else' branch
            false
    end
```

6.9.8 Case

```
case Expr of
    Pattern1 [when GuardSeq1] ->
        Body1;
    ...;
    PatternN [when GuardSeqN] ->
        BodyN
end
```

The expression `Expr` is evaluated and the patterns `Pattern` are sequentially matched against the result. If a match succeeds and the optional guard sequence `GuardSeq` is true, the corresponding `Body` is evaluated.

The return value of `Body` is the return value of the `case` expression.

If there is no matching pattern with a true guard sequence, a `case_clause` run-time error occurs.

Example:

```
is_valid_signal(Signal) ->
    case Signal of
        {signal, _What, _From, _To} ->
            true;
        {signal, _What, _To} ->
            true;
        _Else ->
            false
    end.
```

6.9.9 Maybe

Note:

`maybe` is an experimental new feature introduced in OTP 25. By default, it is disabled. To enable `maybe`, use compiler option `{feature,maybe_expr,enable}`. The feature must also be enabled in the runtime using the `-enable-feature` option to `erl`.

```
maybe
    Expr1,
    ...,
    ExprN
end
```

The expressions in a `maybe` block are evaluated sequentially. If all expressions are evaluated successfully, the return value of the `maybe` block is `ExprN`. However, execution can be short-circuited by a conditional match expression:

```
Expr1 ?= Expr2
```

`?=` is called the conditional match operator. It is only allowed to be used at the top-level of a `maybe` block. It matches the pattern `Expr1` against `Expr2`. If the matching succeeds, any unbound variable in the pattern becomes bound. If the expression is the last expression in the `maybe` block, it also returns the value of `Expr2`. If the matching is unsuccessful, the rest of the expressions in the `maybe` block are skipped and the return value of the `maybe` block is `Expr2`.

None of the variables bound in a `maybe` block must be used in the code that follows the block.

Here is an example:

```
maybe
    {ok, A} ?= a(),
    true = A >= 0,
    {ok, B} ?= b(),
    A + B
end
```

Let us first assume that `a()` returns `{ok,42}` and `b()` returns `{ok,58}`. With those return values, all of the match operators will succeed, and the return value of the `maybe` block is `A + B`, which is equal to `42 + 58 = 100`.

Now let us assume that `a()` returns `error`. The conditional match operator in `{ok, A} ?= a()` fails to match, and the return value of the `maybe` block is the value of the expression that failed to match, namely `error`. Similarly, if `b()` returns `wrong`, the return value of the `maybe` block is `wrong`.

Finally, let us assume that `a()` returns `-1`. Because `true = A >= 0` uses the match operator `=`, a `{badmatch,false}` run-time error occurs when the expression fails to match the pattern.

The example can be written in a less succient way using nested case expressions:

```
case a() of
    {ok, A} ->
        true = A >= 0,
        case b() of
            {ok, B} ->
                A + B;
            Other1 ->
                Other1
        end;
    Other2 ->
        Other2
end
```

The `maybe` block can be augmented with `else` clauses:

```
maybe
    Expr1,
    ...,
    ExprN
else
    Pattern1 [when GuardSeq1] ->
        Body1;
    ...;
    PatternN [when GuardSeqN] ->
        BodyN
end
```

If a conditional match operator fails, the failed expression is matched against the patterns in all clauses between the `else` and `end` keywords. If a match succeeds and the optional guard sequence `GuardSeq` is true, the corresponding `Body` is evaluated. The value returned from the body is the return value of the `maybe` block.

If there is no matching pattern with a true guard sequence, an `else_clause` run-time error occurs.

None of the variables bound in a `maybe` block must be used in the `else` clauses. None of the variables bound in the `else` clauses must be used in the code that follows the `maybe` block.

Here is the previous example augmented with a `else` clauses:

```
maybe
    {ok, A} ?= a(),
    true = A >= 0,
    {ok, B} ?= b(),
    A + B
else
    error -> error;
    wrong -> error
end
```

The `else` clauses translate the failing value from the conditional match operators to the value `error`. If the failing value is not one of the recognized values, a `else_clause` run-time error occurs.

6.9.10 Send

```
Expr1 ! Expr2
```

Sends the value of `Expr2` as a message to the process specified by `Expr1`. The value of `Expr2` is also the return value of the expression.

`Expr1` must evaluate to a pid, an alias (reference), a port, a registered name (atom), or a tuple `{Name,Node}`. `Name` is an atom and `Node` is a node name, also an atom.

- If `Expr1` evaluates to a name, but this name is not registered, a `badarg` run-time error occurs.
- Sending a message to a reference never fails, even if the reference is no longer (or never was) an alias.
- Sending a message to a pid never fails, even if the pid identifies a non-existing process.
- Distributed message sending, that is, if `Expr1` evaluates to a tuple `{Name,Node}` (or a pid located at another node), also never fails.

6.9.11 Receive

```
receive
    Pattern1 [when GuardSeq1] ->
        Body1;
    ...;
    PatternN [when GuardSeqN] ->
        BodyN
end
```

Fetches a received message present in the message queue of the process. The first message in the message queue is matched sequentially against the patterns from top to bottom. If no match was found, the matching sequence is repeated for the second message in the queue, and so on. Messages are queued in the order they were received. If a match succeeds, that is, if the `Pattern` matches and the optional guard sequence `GuardSeq` is true, then the message is removed from the message queue and the corresponding `Body` is evaluated. All other messages in the message queue remain unchanged.

The return value of `Body` is the return value of the `receive` expression.

`receive` never fails. The execution is suspended, possibly indefinitely, until a message arrives that matches one of the patterns and with a true guard sequence.

Example:

```
wait_for_onhook() ->
    receive
        onhook ->
            disconnect(),
            idle();
        {connect, B} ->
            B ! {busy, self()},
            wait_for_onhook()
    end.
```

The `receive` expression can be augmented with a timeout:

```
receive
    Pattern1 [when GuardSeq1] ->
        Body1;
    ...;
    PatternN [when GuardSeqN] ->
        BodyN
after
    ExprT ->
        BodyT
end
```

`receive..after` works exactly as `receive`, except that if no matching message has arrived within `ExprT` milliseconds, then `BodyT` is evaluated instead. The return value of `BodyT` then becomes the return value of the `receive..after` expression. `ExprT` is to evaluate to an integer, or the atom `infinity`. The allowed integer range is from 0 to 4294967295, that is, the longest possible timeout is almost 50 days. With a zero value the timeout occurs immediately if there is no matching message in the message queue.

The atom `infinity` will make the process wait indefinitely for a matching message. This is the same as not using a timeout. It can be useful for timeout values that are calculated at runtime.

Example:

```
wait_for_onhook() ->
    receive
        onhook ->
            disconnect(),
            idle();
        {connect, B} ->
            B ! {busy, self()},
            wait_for_onhook()
    after
        60000 ->
            disconnect(),
            error()
    end.
```

It is legal to use a `receive..after` expression with no branches:

```
receive
after
    ExprT ->
        BodyT
end
```

This construction does not consume any messages, only suspends execution in the process for `ExprT` milliseconds. This can be used to implement simple timers.

Example:

```
timer() ->
    spawn(m, timer, [self()]).

timer(Pid) ->
    receive
    after
        5000 ->
            Pid ! timeout
    end.
```

6.9.12 Term Comparisons

```
Expr1 op Expr2
```

op	Description
==	Equal to
/=	Not equal to
=<	Less than or equal to
<	Less than
>=	Greater than or equal to
>	Greater than
=:=	Exactly equal to
=/=	Exactly not equal to

Table 9.1: Term Comparison Operators.

The arguments can be of different data types. The following order is defined:

```
number < atom < reference < fun < port < pid < tuple < map < nil < list < bit string
```

`nil` in the previous expression represents the empty list (`[]`), which is regarded as a separate type from `list/0`. That is why `nil < list`.

Lists are compared element by element. Tuples are ordered by size, two tuples with the same size are compared element by element.

Bit strings are compared bit by bit. If one bit string is a prefix of the other, the shorter bit string is considered smaller.

Maps are ordered by size, two maps with the same size are compared by keys in ascending term order and then by values in key order. In maps key order integers types are considered less than floats types.

Atoms are compared using their string value, codepoint by codepoint.

When comparing an integer to a float, the term with the lesser precision is converted into the type of the other term, unless the operator is one of `=:=` or `=/=`. A float is more precise than an integer until all significant figures of the float are to the left of the decimal point. This happens when the float is larger/smaller than +/-9007199254740992.0. The conversion strategy is changed depending on the size of the float because otherwise comparison of large floats and integers would lose their transitivity.

Term comparison operators return the Boolean value of the expression, `true` or `false`.

Examples:

```
1> 1==1.0.
true
2> 1=:=1.0.
false
3> 1 > a.
false
4> #{c => 3} > #{a => 1, b => 2}.
false
5> #{a => 1, b => 2} == #{a => 1.0, b => 2.0}.
true
6> <<2:2>> < <<128>>.
true
7> <<3:2>> < <<128>>.
false
```

6.9.13 Arithmetic Expressions

```
op Expr
Expr1 op Expr2
```

Operator	Description	Argument Type
+	Unary +	Number
-	Unary -	Number
+		number
-		Number
*		Number
/	Floating point division	Number
bnot	Unary bitwise NOT	Integer
div	Integer division	Integer
rem	Integer remainder of X/Y	Integer
band	Bitwise AND	Integer
bor	Bitwise OR	Integer
bxor	Arithmetic bitwise XOR	Integer

bsl	Arithmetic bitshift left	Integer
bsr	Bitshift right	Integer

Table 9.2: Arithmetic Operators.

Examples:

```
1> +1.
1
2> -1.
-1
3> 1+1.
2
4> 4/2.
2.0
5> 5 div 2.
2
6> 5 rem 2.
1
7> 2#10 band 2#01.
0
8> 2#10 bor 2#01.
3
9> a + 10.
** exception error: an error occurred when evaluating an arithmetic expression
     in operator  +/2
        called as a + 10
10> 1 bsl (1 bsl 64).
** exception error: a system limit has been reached
     in operator  bsl/2
        called as 1 bsl 18446744073709551616
```

6.9.14 Boolean Expressions

```
op Expr
Expr1 op Expr2
```

Operator	**Description**
not	Unary logical NOT
and	Logical AND
or	Logical OR
xor	Logical XOR

Table 9.3: Logical Operators.

Examples:

```
1> not true.
false
2> true and false.
false
3> true xor false.
true
4> true or garbage.
** exception error: bad argument
     in operator  or/2
        called as true or garbage
```

6.9.15 Short-Circuit Expressions

```
Expr1 orelse Expr2
Expr1 andalso Expr2
```

`Expr2` is evaluated only if necessary. That is, `Expr2` is evaluated only if:

- `Expr1` evaluates to `false` in an `orelse` expression.

or

- `Expr1` evaluates to `true` in an `andalso` expression.

Returns either the value of `Expr1` (that is, `true` or `false`) or the value of `Expr2` (if `Expr2` is evaluated).

Example 1:

```
case A >= -1.0 andalso math:sqrt(A+1) > B of
```

This works even if `A` is less than `-1.0`, since in that case, `math:sqrt/1` is never evaluated.

Example 2:

```
OnlyOne = is_atom(L) orelse
         (is_list(L) andalso length(L) == 1),
```

From Erlang/OTP R13A, `Expr2` is no longer required to evaluate to a Boolean value. As a consequence, `andalso` and `orelse` are now tail-recursive. For instance, the following function is tail-recursive in Erlang/OTP R13A and later:

```
all(Pred, [Hd|Tail]) ->
    Pred(Hd) andalso all(Pred, Tail);
all(_, []) ->
    true.
```

6.9.16 List Operations

```
Expr1 ++ Expr2
Expr1 -- Expr2
```

The list concatenation operator `++` appends its second argument to its first and returns the resulting list.

The list subtraction operator `--` produces a list that is a copy of the first argument. The procedure is as follows: for each element in the second argument, the first occurrence of this element (if any) is removed.

Example:

```
1> [1,2,3]++[4,5].
[1,2,3,4,5]
2> [1,2,3,2,1,2]--[2,1,2].
[3,1,2]
```

6.9.17 Map Expressions

Creating Maps

Constructing a new map is done by letting an expression `K` be associated with another expression `V`:

```
#{ K => V }
```

New maps can include multiple associations at construction by listing every association:

```
#{ K1 => V1, .., Kn => Vn }
```

An empty map is constructed by not associating any terms with each other:

```
#{}
```

All keys and values in the map are terms. Any expression is first evaluated and then the resulting terms are used as **key** and **value** respectively.

Keys and values are separated by the `=>` arrow and associations are separated by a comma `,`.

Examples:

```
M0 = #{},                 % empty map
M1 = #{a => <<"hello">>}, % single association with literals
M2 = #{1 => 2, b => b},   % multiple associations with literals
M3 = #{k => {A,B}},       % single association with variables
M4 = #{{"w", 1} => f()}.  % compound key associated with an evaluated expression
```

Here, `A` and `B` are any expressions and `M0` through `M4` are the resulting map terms.

If two matching keys are declared, the latter key takes precedence.

Example:

```
1> #{1 => a, 1 => b}.
#{1 => b }
2> #{1.0 => a, 1 => b}.
#{1 => b, 1.0 => a}
```

The order in which the expressions constructing the keys (and their associated values) are evaluated is not defined. The syntactic order of the key-value pairs in the construction is of no relevance, except in the recently mentioned case of two matching keys.

Updating Maps

Updating a map has a similar syntax as constructing it.

An expression defining the map to be updated, is put in front of the expression defining the keys to be updated and their respective values:

```
M#{ K => V }
```

Here `M` is a term of type map and `K` and `V` are any expression.

If key `K` does not match any existing key in the map, a new association is created from key `K` to value `V`.

If key K matches an existing key in map M, its associated value is replaced by the new value V. In both cases, the evaluated map expression returns a new map.

If M is not of type map, an exception of type badmap is thrown.

To only update an existing value, the following syntax is used:

```
M#{ K := V }
```

Here M is a term of type map, V is an expression and K is an expression that evaluates to an existing key in M.

If key K does not match any existing keys in map M, an exception of type badarg is triggered at runtime. If a matching key K is present in map M, its associated value is replaced by the new value V, and the evaluated map expression returns a new map.

If M is not of type map, an exception of type badmap is thrown.

Examples:

```
M0 = #{},
M1 = M0#{a => 0},
M2 = M1#{a => 1, b => 2},
M3 = M2#{"function" => fun() -> f() end},
M4 = M3#{a := 2, b := 3}.  % 'a' and 'b' was added in `M1` and `M2`.
```

Here M0 is any map. It follows that M1 .. M4 are maps as well.

More examples:

```

1> M = #{1 => a}.
#{1 => a }
2> M#{1.0 => b}.
#{1 => a, 1.0 => b}.
3> M#{1 := b}.
#{1 => b}
4> M#{1.0 := b}.
** exception error: bad argument
```

As in construction, the order in which the key and value expressions are evaluated is not defined. The syntactic order of the key-value pairs in the update is of no relevance, except in the case where two keys match. In that case, the latter value is used.

Maps in Patterns

Matching of key-value associations from maps is done as follows:

```
#{ K := V } = M
```

Here M is any map. The key K must be a guard expression, with all variables already bound. V can be any pattern with either bound or unbound variables.

If the variable V is unbound, it becomes bound to the value associated with the key K, which must exist in the map M. If the variable V is bound, it must match the value associated with K in M.

Note:

Before OTP 23, the expression defining the key K was restricted to be either a single variable or a literal.

Example:

```
1> M = #{"tuple" => {1,2}}.
#{"tuple" => {1,2}}
2> #{"tuple" := {1,B}} = M.
#{"tuple" => {1,2}}
3> B.
2.
```

This binds variable `B` to integer `2`.

Similarly, multiple values from the map can be matched:

```
#{ K1 := V1, .., Kn := Vn } = M
```

Here keys `K1 .. Kn` are any expressions with literals or bound variables. If all key expressions evaluate successfully and all keys exist in map `M`, all variables in `V1 .. Vn` is matched to the associated values of their respective keys.

If the matching conditions are not met, the match fails, either with:

- A `badmatch` exception.

  This is if it is used in the context of the match operator as in the example.
- Or resulting in the next clause being tested in function heads and case expressions.

Matching in maps only allows for `:=` as delimiters of associations.

The order in which keys are declared in matching has no relevance.

Duplicate keys are allowed in matching and match each pattern associated to the keys:

```
#{ K := V1, K := V2 } = M
```

Matching an expression against an empty map literal, matches its type but no variables are bound:

```
#{} = Expr
```

This expression matches if the expression `Expr` is of type map, otherwise it fails with an exception `badmatch`.

Here the key to be retrieved is constructed from an expression:

```
#{{tag,length(List)} := V} = Map
```

`List` must be an already bound variable.

Matching Syntax

Matching of literals as keys are allowed in function heads:

```
%% only start if not_started
handle_call(start, From, #{ state := not_started } = S) ->
...
    {reply, ok, S#{ state := start }};

%% only change if started
handle_call(change, From, #{ state := start } = S) ->
...
    {reply, ok, S#{ state := changed }};
```

Maps in Guards

Maps are allowed in guards as long as all subexpressions are valid guard expressions.

The following guard BIFs handle maps:

- is_map/1 in the `erlang` module

- is_map_key/2 in the `erlang` module
- map_get/2 in the `erlang` module
- map_size/1 in the `erlang` module

6.9.18 Bit Syntax Expressions

```
<<>>
<<E1,...,En>>
```

Each element `Ei` specifies a **segment** of the bit string. Each element `Ei` is a value, followed by an optional **size expression** and an optional **type specifier list**.

```
Ei = Value |
     Value:Size |
     Value/TypeSpecifierList |
     Value:Size/TypeSpecifierList
```

Used in a bit string construction, `Value` is an expression that is to evaluate to an integer, float, or bit string. If the expression is not a single literal or variable, it is to be enclosed in parentheses.

Used in a bit string matching, `Value` must be a variable, or an integer, float, or string.

Notice that, for example, using a string literal as in `<<"abc">>` is syntactic sugar for `<<$a,$b,$c>>`.

Used in a bit string construction, `Size` is an expression that is to evaluate to an integer.

Used in a bit string matching, `Size` must be a guard expression that evaluates to an integer. All variables in the guard expression must be already bound.

Note:

Before OTP 23, `Size` was restricted to be an integer or a variable bound to an integer.

The value of `Size` specifies the size of the segment in units (see below). The default value depends on the type (see below):

- For `integer` it is 8.
- For `float` it is 64.
- For `binary` and `bitstring` it is the whole binary or bit string.

In matching, this default value is only valid for the last element. All other bit string or binary elements in the matching must have a size specification.

For the `utf8`, `utf16`, and `utf32` types, `Size` must not be given. The size of the segment is implicitly determined by the type and value itself.

`TypeSpecifierList` is a list of type specifiers, in any order, separated by hyphens (-). Default values are used for any omitted type specifiers.

`Type`= `integer` | `float` | `binary` | `bytes` | `bitstring` | `bits` | `utf8` | `utf16` | `utf32`
: The default is `integer`. `bytes` is a shorthand for `binary` and `bits` is a shorthand for `bitstring`. See below for more information about the `utf` types.

`Signedness`= `signed` | `unsigned`
: Only matters for matching and when the type is `integer`. The default is `unsigned`.

`Endianness` = `big` | `little` | `native`

Native-endian means that the endianness is resolved at load time to be either big-endian or little-endian, depending on what is native for the CPU that the Erlang machine is run on. Endianness only matters when the Type is either `integer`, `utf16`, `utf32`, or `float`. The default is `big`.

`Unit` = `unit:IntegerLiteral`

The allowed range is 1..256. Defaults to 1 for `integer`, `float`, and `bitstring`, and to 8 for `binary`. No unit specifier must be given for the types `utf8`, `utf16`, and `utf32`.

The value of `Size` multiplied with the unit gives the number of bits. A segment of type `binary` must have a size that is evenly divisible by 8. For a segment of type `float` the size must be either 64, 32, or 16.

Note:

When constructing binaries, if the size `N` of an integer segment is too small to contain the given integer, the most significant bits of the integer are silently discarded and only the `N` least significant bits are put into the binary.

The types `utf8`, `utf16`, and `utf32` specifies encoding/decoding of the **Unicode Transformation Format**s UTF-8, UTF-16, and UTF-32, respectively.

When constructing a segment of a `utf` type, `Value` must be an integer in the range 0..16#D7FF or 16#E000....16#10FFFF. Construction fails with a `badarg` exception if `Value` is outside the allowed ranges. The size of the resulting binary segment depends on the type or `Value`, or both:

- For `utf8`, `Value` is encoded in 1-4 bytes.
- For `utf16`, `Value` is encoded in 2 or 4 bytes.
- For `utf32`, `Value` is always be encoded in 4 bytes.

When constructing, a literal string can be given followed by one of the UTF types, for example: `<<"abc"/utf8>>` which is syntactic sugar for `<<$a/utf8,$b/utf8,$c/utf8>>`.

A successful match of a segment of a `utf` type, results in an integer in the range 0..16#D7FF or 16#E000..16#10FFFF. The match fails if the returned value falls outside those ranges.

A segment of type `utf8` matches 1-4 bytes in the binary, if the binary at the match position contains a valid UTF-8 sequence. (See RFC-3629 or the Unicode standard.)

A segment of type `utf16` can match 2 or 4 bytes in the binary. The match fails if the binary at the match position does not contain a legal UTF-16 encoding of a Unicode code point. (See RFC-2781 or the Unicode standard.)

A segment of type `utf32` can match 4 bytes in the binary in the same way as an `integer` segment matches 32 bits. The match fails if the resulting integer is outside the legal ranges mentioned above.

Examples:

```
1> Bin1 = <<1,17,42>>.
<<1,17,42>>
2> Bin2 = <<"abc">>.
<<97,98,99>>
3> Bin3 = <<1,17,42:16>>.
<<1,17,0,42>>
4> <<A,B,C:16>> = <<1,17,42:16>>.
<<1,17,0,42>>
5> C.
42
6> <<D:16,E,F>> = <<1,17,42:16>>.
<<1,17,0,42>>
7> D.
273
8> F.
42
9> <<G,H/binary>> = <<1,17,42:16>>.
<<1,17,0,42>>
10> H.
<<17,0,42>>
11> <<G,J/bitstring>> = <<1,17,42:12>>.
<<1,17,2,10:4>>
12> J.
<<17,2,10:4>>
13> <<1024/utf8>>.
<<208,128>>
```

Notice that bit string patterns cannot be nested.

Notice also that "`B=<<1>>`" is interpreted as "`B =< <1>>`" which is a syntax error. The correct way is to write a space after '=': "`B = <<1>>`".

More examples are provided in Programming Examples.

6.9.19 Fun Expressions

```
fun
    [Name](Pattern11,...,Pattern1N) [when GuardSeq1] ->
              Body1;
    ...;
    [Name](PatternK1,...,PatternKN) [when GuardSeqK] ->
              BodyK
end
```

A fun expression begins with the keyword `fun` and ends with the keyword `end`. Between them is to be a function declaration, similar to a regular function declaration, except that the function name is optional and is to be a variable, if any.

Variables in a fun head shadow the function name and both shadow variables in the function clause surrounding the fun expression. Variables bound in a fun body are local to the fun body.

The return value of the expression is the resulting fun.

Examples:

```
1> Fun1 = fun (X) -> X+1 end.
#Fun<erl_eval.6.39074546>
2> Fun1(2).
3
3> Fun2 = fun (X) when X>=5 -> gt; (X) -> lt end.
#Fun<erl_eval.6.39074546>
4> Fun2(7).
gt
5> Fun3 = fun Fact(1) -> 1; Fact(X) when X > 1 -> X * Fact(X - 1) end.
#Fun<erl_eval.6.39074546>
6> Fun3(4).
24
```

The following fun expressions are also allowed:

```
fun Name/Arity
fun Module:Name/Arity
```

In `Name/Arity`, `Name` is an atom and `Arity` is an integer. `Name/Arity` must specify an existing local function. The expression is syntactic sugar for:

```
fun (Arg1,...,ArgN) -> Name(Arg1,...,ArgN) end
```

In `Module:Name/Arity`, `Module`, and `Name` are atoms and `Arity` is an integer. Starting from Erlang/OTP R15, `Module`, `Name`, and `Arity` can also be variables. A fun defined in this way refers to the function `Name` with arity `Arity` in the **latest** version of module `Module`. A fun defined in this way is not dependent on the code for the module in which it is defined.

More examples are provided in Programming Examples.

6.9.20 Catch and Throw

```
catch Expr
```

Returns the value of `Expr` unless an exception occurs during the evaluation. In that case, the exception is caught.

For exceptions of class `error`, that is, run-time errors, `{'EXIT',{Reason,Stack}}` is returned.

For exceptions of class `exit`, that is, the code called `exit(Term)`, `{'EXIT',Term}` is returned.

For exceptions of class `throw`, that is the code called `throw(Term)`, `Term` is returned.

`Reason` depends on the type of error that occurred, and `Stack` is the stack of recent function calls, see Exit Reasons.

Examples:

```
1> catch 1+2.
3
2> catch 1+a.
{'EXIT',{badarith,[...]}}
```

The BIF `throw(Any)` can be used for non-local return from a function. It must be evaluated within a `catch`, which returns the value `Any`.

Example:

```
5> catch throw(hello).
hello
```

If `throw/1` is not evaluated within a catch, a `nocatch` run-time error occurs.

6.9.21 Try

```
try Exprs
catch
    Class1:ExceptionPattern1[:Stacktrace] [when ExceptionGuardSeq1] ->
        ExceptionBody1;
    ClassN:ExceptionPatternN[:Stacktrace] [when ExceptionGuardSeqN] ->
        ExceptionBodyN
end
```

This is an enhancement of catch. It gives the possibility to:

- Distinguish between different exception classes.
- Choose to handle only the desired ones.
- Passing the others on to an enclosing `try` or `catch`, or to default error handling.

Notice that although the keyword `catch` is used in the `try` expression, there is not a `catch` expression within the `try` expression.

It returns the value of `Exprs` (a sequence of expressions `Expr1, ..., ExprN`) unless an exception occurs during the evaluation. In that case the exception is caught and the patterns `ExceptionPattern` with the right exception class `Class` are sequentially matched against the caught exception. If a match succeeds and the optional guard sequence `ExceptionGuardSeq` is true, the corresponding `ExceptionBody` is evaluated to become the return value.

`Stacktrace`, if specified, must be the name of a variable (not a pattern). The stack trace is bound to the variable when the corresponding `ExceptionPattern` matches.

If an exception occurs during evaluation of `Exprs` but there is no matching `ExceptionPattern` of the right `Class` with a true guard sequence, the exception is passed on as if `Exprs` had not been enclosed in a `try` expression.

If an exception occurs during evaluation of `ExceptionBody`, it is not caught.

It is allowed to omit `Class` and `Stacktrace`. An omitted `Class` is shorthand for `throw`:

```
try Exprs
catch
    ExceptionPattern1 [when ExceptionGuardSeq1] ->
        ExceptionBody1;
    ExceptionPatternN [when ExceptionGuardSeqN] ->
        ExceptionBodyN
end
```

The `try` expression can have an `of` section:

```
try Exprs of
    Pattern1 [when GuardSeq1] ->
        Body1;
    ...;
    PatternN [when GuardSeqN] ->
        BodyN
catch
    Class1:ExceptionPattern1[:Stacktrace] [when ExceptionGuardSeq1] ->
        ExceptionBody1;
    ...;
    ClassN:ExceptionPatternN[:Stacktrace] [when ExceptionGuardSeqN] ->
        ExceptionBodyN
end
```

If the evaluation of `Exprs` succeeds without an exception, the patterns `Pattern` are sequentially matched against the result in the same way as for a case expression, except that if the matching fails, a `try_clause` run-time error occurs instead of a `case_clause`.

Only exceptions occurring during the evaluation of `Exprs` can be caught by the `catch` section. Exceptions occurring in a `Body` or due to a failed match are not caught.

The `try` expression can also be augmented with an `after` section, intended to be used for cleanup with side effects:

```
try Exprs of
    Pattern1 [when GuardSeq1] ->
        Body1;
    ...;
    PatternN [when GuardSeqN] ->
        BodyN
catch
    Class1:ExceptionPattern1[:Stacktrace] [when ExceptionGuardSeq1] ->
        ExceptionBody1;
    ...;
    ClassN:ExceptionPatternN[:Stacktrace] [when ExceptionGuardSeqN] ->
        ExceptionBodyN
after
    AfterBody
end
```

`AfterBody` is evaluated after either `Body` or `ExceptionBody`, no matter which one. The evaluated value of `AfterBody` is lost; the return value of the `try` expression is the same with an `after` section as without.

Even if an exception occurs during evaluation of `Body` or `ExceptionBody`, `AfterBody` is evaluated. In this case the exception is passed on after `AfterBody` has been evaluated, so the exception from the `try` expression is the same with an `after` section as without.

If an exception occurs during evaluation of `AfterBody` itself, it is not caught. So if `AfterBody` is evaluated after an exception in `Exprs`, `Body`, or `ExceptionBody`, that exception is lost and masked by the exception in `AfterBody`.

The `of`, `catch`, and `after` sections are all optional, as long as there is at least a `catch` or an `after` section. So the following are valid `try` expressions:

```
try Exprs of
    Pattern when GuardSeq ->
        Body
after
    AfterBody
end

try Exprs
catch
    ExpressionPattern ->
        ExpressionBody
after
    AfterBody
end

try Exprs after AfterBody end
```

Next is an example of using `after`. This closes the file, even in the event of exceptions in `file:read/2` or in `binary_to_term/1`. The exceptions are the same as without the `try...after...end` expression:

```
termize_file(Name) ->
    {ok,F} = file:open(Name, [read,binary]),
    try
        {ok,Bin} = file:read(F, 1024*1024),
        binary_to_term(Bin)
    after
        file:close(F)
    end.
```

Next is an example of using `try` to emulate `catch Expr`:

```
try Expr
catch
    throw:Term -> Term;
    exit:Reason -> {'EXIT',Reason}
    error:Reason:Stk -> {'EXIT',{Reason,Stk}}
end
```

6.9.22 Parenthesized Expressions

```
(Expr)
```

Parenthesized expressions are useful to override operator precedences, for example, in arithmetic expressions:

```
1> 1 + 2 * 3.
7
2> (1 + 2) * 3.
9
```

6.9.23 Block Expressions

```
begin
   Expr1,
   ...,
   ExprN
end
```

Block expressions provide a way to group a sequence of expressions, similar to a clause body. The return value is the value of the last expression `ExprN`.

6.9.24 List Comprehensions

List comprehensions is a feature of many modern functional programming languages. Subject to certain rules, they provide a succinct notation for generating elements in a list.

List comprehensions are analogous to set comprehensions in Zermelo-Frankel set theory and are called ZF expressions in Miranda. They are analogous to the `setof` and `findall` predicates in Prolog.

List comprehensions are written with the following syntax:

```
[Expr || Qualifier1,...,QualifierN]
```

Here, `Expr` is an arbitrary expression, and each `Qualifier` is either a generator or a filter.

- A **generator** is written as:
  `Pattern <- ListExpr`.

`ListExpr` must be an expression, which evaluates to a list of terms.

- A **bit string generator** is written as:
  `BitstringPattern <= BitStringExpr.`
  `BitStringExpr` must be an expression, which evaluates to a bitstring.
- A **filter** is an expression, which evaluates to `true` or `false`, or a guard expression. If the filter is not a guard expression and evaluates to a non-Boolean value `Val`, an exception `{bad_filter, Val}` is triggered at runtime.

The variables in the generator patterns shadow previously bound variables, including variables bound in a previous generator pattern.

A list comprehension returns a list, where the elements are the result of evaluating `Expr` for each combination of generator list elements and bit string generator elements, for which all filters are true.

Example:

```
1> [X*2 || X <- [1,2,3]].
[2,4,6]
```

When there are no generators or bit string generators, a list comprehension returns either a list with one element (the result of evaluating `Expr`) if all filters are true or an empty list otherwise.

Example:

```
1> [2 || is_integer(2)].
[2]
2> [x || is_integer(x)].
[]
```

More examples are provided in Programming Examples.

6.9.25 Bit String Comprehensions

Bit string comprehensions are analogous to List Comprehensions. They are used to generate bit strings efficiently and succinctly.

Bit string comprehensions are written with the following syntax:

```
<< BitStringExpr || Qualifier1,...,QualifierN >>
```

`BitStringExpr` is an expression that evaluates to a bit string. If `BitStringExpr` is a function call, it must be enclosed in parentheses. Each `Qualifier` is either a generator, a bit string generator or a filter.

- A **generator** is written as:
  `Pattern <- ListExpr.`
  `ListExpr` must be an expression that evaluates to a list of terms.
- A **bit string generator** is written as:
  `BitstringPattern <= BitStringExpr.`
  `BitStringExpr` must be an expression that evaluates to a bitstring.
- A **filter** is an expression, which evaluates to `true` or `false`, or a guard expression. If the filter is not a guard expression and evaluates to a non-Boolean value `Val`, an exception `{bad_filter, Val}` is triggered at runtime.

The variables in the generator patterns shadow previously bound variables, including variables bound in a previous generator pattern.

A bit string comprehension returns a bit string, which is created by concatenating the results of evaluating `BitString` for each combination of bit string generator elements, for which all filters are true.

Example:

```
1> << << (X*2) >> ||
<<X>> <= << 1,2,3 >> >>.
<<2,4,6>>
```

More examples are provided in Programming Examples.

6.9.26 Guard Sequences

A **guard sequence** is a sequence of guards, separated by semicolon (;). The guard sequence is true if at least one of the guards is true. (The remaining guards, if any, are not evaluated.)

`Guard1;...;GuardK`

A **guard** is a sequence of guard expressions, separated by comma (,). The guard is true if all guard expressions evaluate to `true`.

`GuardExpr1,...,GuardExprN`

6.9.27 Guard Expressions

The set of valid **guard expressions** is a subset of the set of valid Erlang expressions. The reason for restricting the set of valid expressions is that evaluation of a guard expression must be guaranteed to be free of side effects. Valid guard expressions are the following:

- Variables
- Constants (atoms, integer, floats, lists, tuples, records, binaries, and maps)
- Expressions that construct atoms, integer, floats, lists, tuples, records, binaries, and maps
- Expressions that update a map
- The record expressions `Expr#Name.Field` and `#Name.Field`
- Calls to the BIFs specified in tables **Type Test BIFs** and **Other BIFs Allowed in Guard Expressions**
- Term comparisons
- Arithmetic expressions
- Boolean expressions
- Short-circuit expressions (`andalso`/`orelse`)

`is_atom/1`
`is_binary/1`
`is_bitstring/1`
`is_boolean/1`
`is_float/1`
`is_function/1`
`is_function/2`

is_integer/1
is_list/1
is_map/1
is_number/1
is_pid/1
is_port/1
is_record/2
is_record/3
is_reference/1
is_tuple/1

Table 9.4: Type Test BIFs

Notice that most type test BIFs have older equivalents, without the `is_` prefix. These old BIFs are retained for backwards compatibility only and are not to be used in new code. They are also only allowed at top level. For example, they are not allowed in Boolean expressions in guards.

abs(Number)
bit_size(Bitstring)
byte_size(Bitstring)
element(N, Tuple)
float(Term)
hd(List)
is_map_key(Key, Map)
length(List)
map_get(Key, Map)
map_size(Map)
node()
node(Pid|Ref|Port)
round(Number)

`self()`
`size(Tuple|Bitstring)`
`tl(List)`
`trunc(Number)`
`tuple_size(Tuple)`

Table 9.5: Other BIFs Allowed in Guard Expressions

If an arithmetic expression, a Boolean expression, a short-circuit expression, or a call to a guard BIF fails (because of invalid arguments), the entire guard fails. If the guard was part of a guard sequence, the next guard in the sequence (that is, the guard following the next semicolon) is evaluated.

6.9.28 Operator Precedence

Operator precedence in falling priority:

:	
#	
Unary + - bnot not	
/ * div rem band and	Left associative
+ - bor bxor bsl bsr or xor	Left associative
++ --	Right associative
== /= =< < >= > =:= =/=	
andalso	
orelse	
= !	Right associative
?=	
catch	

Table 9.6: Operator Precedence

When evaluating an expression, the operator with the highest priority is evaluated first. Operators with the same priority are evaluated according to their associativity.

Example:

The left associative arithmetic operators are evaluated left to right:

```
6 + 5 * 4 - 3 / 2 evaluates to
6 + 20 - 1.5 evaluates to
26 - 1.5 evaluates to
24.5
```

6.10 Preprocessor

6.10.1 File Inclusion

A file can be included as follows:

```
-include(File).
-include_lib(File).
```

`File`, a string, is to point out a file. The contents of this file are included as is, at the position of the directive.

Include files are typically used for record and macro definitions that are shared by several modules. It is recommended to use the file name extension `.hrl` for include files.

`File` can start with a path component $VAR, for some string VAR. If that is the case, the value of the environment variable VAR as returned by `os:getenv(VAR)` is substituted for $VAR. If `os:getenv(VAR)` returns `false`, $VAR is left as is.

If the filename `File` is absolute (possibly after variable substitution), the include file with that name is included. Otherwise, the specified file is searched for in the following directories, and in this order:

- The current working directory
- The directory where the module is being compiled
- The directories given by the `include` option

For details, see the erlc(1) manual page in ERTS and compile(3) manual page in Compiler.

Examples:

```
-include("my_records.hrl").
-include("incdir/my_records.hrl").
-include("/home/user/proj/my_records.hrl").
-include("$PROJ_ROOT/my_records.hrl").
```

`include_lib` is similar to `include`, but is not to point out an absolute file. Instead, the first path component (possibly after variable substitution) is assumed to be the name of an application.

Example:

```
-include_lib("kernel/include/file.hrl").
```

The code server uses `code:lib_dir(kernel)` to find the directory of the current (latest) version of Kernel, and then the subdirectory `include` is searched for the file `file.hrl`.

6.10.2 Defining and Using Macros

A macro is defined as follows:

```
-define(Const, Replacement).
-define(Func(Var1,...,VarN), Replacement).
```

A macro definition can be placed anywhere among the attributes and function declarations of a module, but the definition must come before any usage of the macro.

If a macro is used in several modules, it is recommended that the macro definition is placed in an include file.

A macro is used as follows:

```
?Const
?Func(Arg1,...,ArgN)
```

Macros are expanded during compilation. A simple macro `?Const` is replaced with `Replacement`.

Example:

```
-define(TIMEOUT, 200).
...
call(Request) ->
    server:call(refserver, Request, ?TIMEOUT).
```

This is expanded to:

```
call(Request) ->
    server:call(refserver, Request, 200).
```

A macro `?Func(Arg1,...,ArgN)` is replaced with `Replacement`, where all occurrences of a variable `Var` from the macro definition are replaced with the corresponding argument `Arg`.

Example:

```
-define(MACRO1(X, Y), {a, X, b, Y}).
...
bar(X) ->
    ?MACRO1(a, b),
    ?MACRO1(X, 123)
```

This is expanded to:

```
bar(X) ->
    {a,a,b,b},
    {a,X,b,123}.
```

It is good programming practice, but not mandatory, to ensure that a macro definition is a valid Erlang syntactic form.

To view the result of macro expansion, a module can be compiled with the `'P'` option. `compile:file(File, ['P'])`. This produces a listing of the parsed code after preprocessing and parse transforms, in the file `File.P`.

6.10.3 Predefined Macros

The following macros are predefined:

`?MODULE`
: The name of the current module.

`?MODULE_STRING`.
: The name of the current module, as a string.

`?FILE`.
: The file name of the current module.

`?LINE`.
: The current line number.

`?MACHINE`.
: The machine name, `'BEAM'`.

`?FUNCTION_NAME`
: The name of the current function.

`?FUNCTION_ARITY`
: The arity (number of arguments) for the current function.

`?OTP_RELEASE`
: The OTP release that the currently executing ERTS application is part of, as an integer. For details, see `erlang:system_info(otp_release)`. This macro was introduced in OTP release 21.

`?FEATURE_AVAILABLE(Feature)`
: Expands to `true` if the feature `Feature` is available. The feature might or might not be enabled. This macro was introduced with OTP release 25.

`?FEATURE_ENABLED(Feature)`
: Expands to `true` if the feature `Feature` is enabled. This macro was introduced with OTP release 25.

6.10.4 Macros Overloading

It is possible to overload macros, except for predefined macros. An overloaded macro has more than one definition, each with a different number of arguments.

The feature was added in Erlang 5.7.5/OTP R13B04.

A macro `?Func(Arg1,...,ArgN)` with a (possibly empty) list of arguments results in an error message if there is at least one definition of `Func` with arguments, but none with N arguments.

Assuming these definitions:

```
-define(F0(), c).
-define(F1(A), A).
-define(C, m:f).
```

the following does not work:

```
f0() ->
    ?F0. % No, an empty list of arguments expected.

f1(A) ->
    ?F1(A, A). % No, exactly one argument expected.
```

On the other hand,

```
f() ->
    ?C().
```

is expanded to

```
f() ->
    m:f().
```

6.10.5 Flow Control in Macros

The following macro directives are supplied:

`-undef(Macro).`
: Causes the macro to behave as if it had never been defined.

`-ifdef(Macro).`
: Evaluate the following lines only if `Macro` is defined.

`-ifndef(Macro).`
: Evaluate the following lines only if `Macro` is not defined.

`-else.`
: Only allowed after an `ifdef` or `ifndef` directive. If that condition is false, the lines following `else` are evaluated instead.

`-endif.`
: Specifies the end of an `ifdef`, an `ifndef` directive, or the end of an `if` or `elif` directive.

`-if(Condition).`
: Evaluates the following lines only if `Condition` evaluates to true.

`-elif(Condition).`
: Only allowed after an `if` or another `elif` directive. If the preceding `if` or `elif` directives do not evaluate to true, and the `Condition` evaluates to true, the lines following the `elif` are evaluated instead.

Note:

The macro directives cannot be used inside functions.

Example:

```
-module(m).
...

-ifdef(debug).
-define(LOG(X), io:format("{~p,~p}: ~p~n", [?MODULE,?LINE,X])).
-else.
-define(LOG(X), true).
-endif.

...
```

When trace output is desired, `debug` is to be defined when the module `m` is compiled:

```

% erlc -Ddebug m.erl

or

1> c(m, {d, debug}).
{ok,m}
```

`?LOG(Arg)` is then expanded to a call to `io:format/2` and provide the user with some simple trace output.

Example:

```
-module(m)
...
-ifdef(OTP_RELEASE).
  %% OTP 21 or higher
  -if(?OTP_RELEASE >= 22).
    %% Code that will work in OTP 22 or higher
  -elif(?OTP_RELEASE >= 21).
    %% Code that will work in OTP 21 or higher
  -endif.
-else.
  %% OTP 20 or lower.
-endif.
...
```

The code uses the `OTP_RELEASE` macro to conditionally select code depending on release.

6.10.6 The -feature() directive

The directive `-feature(FeatureName, enable | disable)` can be used to enable or disable the feature `FeatureName`. This is the preferred way of enabling (disabling) features, although it is possible to do it with options to the compiler as well.

Note that the `-feature(..)` directive may only appear before any syntax is used. In practice this means it should appear before any `-export(..)` or record definitions.

6.10.7 -error() and -warning() directives

The directive `-error(Term)` causes a compilation error.

Example:

```
-module(t).
-export([version/0]).

-ifdef(VERSION).
version() -> ?VERSION.
-else.
-error("Macro VERSION must be defined.").
version() -> "".
-endif.
```

The error message will look like this:

```

% erlc t.erl
t.erl:7: -error("Macro VERSION must be defined.").
```

The directive `-warning(Term)` causes a compilation warning.

Example:

```
-module(t).
-export([version/0]).

-ifndef(VERSION).
-warning("Macro VERSION not defined -- using default version.").
-define(VERSION, "0").
-endif.
version() -> ?VERSION.
```

The warning message will look like this:

```

% erlc t.erl
t.erl:5: Warning: -warning("Macro VERSION not defined -- using default version.").
```

The `-error()` and `-warning()` directives were added in OTP 19.

6.10.8 Stringifying Macro Arguments

The construction `??Arg`, where `Arg` is a macro argument, is expanded to a string containing the tokens of the argument. This is similar to the `#arg` stringifying construction in C.

Example:

```
-define(TESTCALL(Call), io:format("Call ~s: ~w~n", [??Call, Call])).

?TESTCALL(myfunction(1,2)),
?TESTCALL(you:function(2,1)).
```

results in

```
io:format("Call ~s: ~w~n",["myfunction ( 1 , 2 )",myfunction(1,2)]),
io:format("Call ~s: ~w~n",["you : function ( 2 , 1 )",you:function(2,1)]).
```

That is, a trace output, with both the function called and the resulting value.

6.11 Records

A record is a data structure for storing a fixed number of elements. It has named fields and is similar to a struct in C. Record expressions are translated to tuple expressions during compilation. Therefore, record expressions are not understood by the shell unless special actions are taken. For details, see the shell(3) manual page in STDLIB.

More examples are provided in Programming Examples.

6.11.1 Defining Records

A record definition consists of the name of the record, followed by the field names of the record. Record and field names must be atoms. Each field can be given an optional default value. If no default value is supplied, `undefined` is used.

```
-record(Name, {Field1 [= Value1],
               ...
               FieldN [= ValueN]}).
```

A record definition can be placed anywhere among the attributes and function declarations of a module, but the definition must come before any usage of the record.

If a record is used in several modules, it is recommended that the record definition is placed in an include file.

6.11.2 Creating Records

The following expression creates a new `Name` record where the value of each field `FieldI` is the value of evaluating the corresponding expression `ExprI`:

```
#Name{Field1=Expr1,...,FieldK=ExprK}
```

The fields can be in any order, not necessarily the same order as in the record definition, and fields can be omitted. Omitted fields get their respective default value instead.

If several fields are to be assigned the same value, the following construction can be used:

```
#Name{Field1=Expr1,...,FieldK=ExprK, _=ExprL}
```

Omitted fields then get the value of evaluating `ExprL` instead of their default values. This feature is primarily intended to be used to create patterns for ETS and Mnesia match functions.

Example:

```
-record(person, {name, phone, address}).

...

lookup(Name, Tab) ->
    ets:match_object(Tab, #person{name=Name, _='_'}).
```

6.11.3 Accessing Record Fields

```
Expr#Name.Field
```

Returns the value of the specified field. `Expr` is to evaluate to a `Name` record.

The following expression returns the position of the specified field in the tuple representation of the record:

```
#Name.Field
```

Example:

```
-record(person, {name, phone, address}).

...

lookup(Name, List) ->
    lists:keysearch(Name, #person.name, List).
```

6.11.4 Updating Records

```
Expr#Name{Field1=Expr1,...,FieldK=ExprK}
```

`Expr` is to evaluate to a `Name` record. A copy of this record is returned, with the value of each specified field `FieldI` changed to the value of evaluating the corresponding expression `ExprI`. All other fields retain their old values.

6.11.5 Records in Guards

Since record expressions are expanded to tuple expressions, creating records and accessing record fields are allowed in guards. However all subexpressions, for example, for field initiations, must be valid guard expressions as well.

Examples:

```
handle(Msg, State) when Msg==#msg{to=void, no=3} ->
    ...

handle(Msg, State) when State#state.running==true ->
    ...
```

There is also a type test BIF `is_record(Term, RecordTag)`.

Example:

```
is_person(P) when is_record(P, person) ->
    true;
is_person(_P) ->
    false.
```

6.11.6 Records in Patterns

A pattern that matches a certain record is created in the same way as a record is created:

```
#Name{Field1=Expr1,...,FieldK=ExprK}
```

In this case, one or more of `Expr1`...`ExprK` can be unbound variables.

6.11.7 Nested Records

Beginning with Erlang/OTP R14, parentheses when accessing or updating nested records can be omitted. Assume the following record definitions:

```
-record(nrec0, {name = "nested0"}).
-record(nrec1, {name = "nested1", nrec0=#nrec0{}}).
-record(nrec2, {name = "nested2", nrec1=#nrec1{}}).

N2 = #nrec2{},
```

Before R14, parentheses were needed as follows:

```
"nested0" = ((N2#nrec2.nrec1)#nrec1.nrec0)#nrec0.name,
N0n = ((N2#nrec2.nrec1)#nrec1.nrec0)#nrec0{name = "nested0a"},
```

Since R14, the following can also be written:

```
"nested0" = N2#nrec2.nrec1#nrec1.nrec0#nrec0.name,
N0n = N2#nrec2.nrec1#nrec1.nrec0#nrec0{name = "nested0a"},
```

6.11.8 Internal Representation of Records

Record expressions are translated to tuple expressions during compilation. A record defined as:

```
-record(Name, {Field1,...,FieldN}).
```

is internally represented by the tuple:

```
{Name,Value1,...,ValueN}
```

Here each `ValueI` is the default value for `FieldI`.

To each module using records, a pseudo function is added during compilation to obtain information about records:

```
record_info(fields, Record) -> [Field]
record_info(size, Record) -> Size
```

`Size` is the size of the tuple representation, that is, one more than the number of fields.

In addition, `#Record.Name` returns the index in the tuple representation of `Name` of the record `Record`.

`Name` must be an atom.

6.12 Errors and Error Handling

6.12.1 Terminology

Errors can roughly be divided into four different types:

Compile-time errors
: When the compiler fails to compile the program, for example a syntax error.

Logical errors
: When a program does not behave as intended, but does not crash. An example is that nothing happens when a button in a graphical user interface is clicked.

Run-time errors
: When a crash occurs. An example is when an operator is applied to arguments of the wrong type. The Erlang programming language has built-in features for handling of run-time errors. A run-time error can also be emulated by calling `error(Reason)`. Run-time errors are exceptions of class `error`.

Generated errors
: When the code itself calls `exit/1` or `throw/1`. Generated errors are exceptions of class `exit` or `throw`.

When an exception occurs in Erlang, execution of the process that evaluated the erroneous expression is stopped. This is referred to as a **failure**, that execution or evaluation **fails**, or that the process **fails**, **terminates**, or **exits**. Notice that a process can terminate/exit for other reasons than a failure.

A process that terminates emits an **exit signal** with an **exit reason** that describes why the process terminated. Normally, some information about any erroneous termination is printed to the terminal. See Process Termination in the Processes chapter for more details on termination.

6.12.2 Exceptions

Exceptions are run-time errors or generated errors and are of three different classes, with different origins. The try expression can distinguish between the different classes, whereas the catch expression cannot. `try` and `catch` are described in Expressions.

Class	**Origin**
`error`	Run-time error, for example, `1+a`, or the process called `error/1,2`
`exit`	The process called `exit/1`
`throw`	The process called `throw/1`

Table 12.1: Exception Classes.

All of the above exceptions can also be generated by calling `erlang:raise/3`.

An exception consists of its class, an exit reason (see Exit Reason), and a stack trace (which aids in finding the code location of the exception).

The stack trace can be bound to a variable from within a `try` expression for any exception class, or as part of the exit reason when a run-time error is caught by a `catch`. Example:

```
> {'EXIT',{test,Stacktrace}} = (catch error(test)), Stacktrace.
[{shell,apply_fun,3,[]},
 {erl_eval,do_apply,6,[]},
 ...]
> try throw(test) catch Class:Reason:Stacktrace -> Stacktrace end.
[{shell,apply_fun,3,[]},
 {erl_eval,do_apply,6,[]},
 ...]
```

The call-stack back trace (stacktrace)

The stack back-trace (**stacktrace**) is a list contains `{Module, Function, Arity, ExtraInfo}` and/or `{Fun, Arity, ExtraInfo}` tuples. The field `Arity` in the tuple can be the argument list of that function call instead of an arity integer, depending on the exception.

`ExtraInfo` is a (possibly empty) list of two-element tuples in any order that provides additional information about the exception. The first element is an atom describing the type of information in the second element. The following items can occur:

`error_info`
: The second element of the tuple is a map providing additional information about what caused the exception. This information can be created by calling `error/3` and is used by `erl_error:format_exception/4`.

`file`
: The second element of the tuple is a string (list of characters) representing the filename of the source file of the function.

`line`
: The second element of the tuple is the line number (an integer > 0) in the source file where the exception occurred or the function was called.

Warning:

Developers should rely on stacktrace entries only for debugging purposes.

The VM performs tail call optimization, which does not add new entries to the stacktrace, and also limits stacktraces to a certain depth. Furthermore, compiler options, optimizations and future changes may add or remove stacktrace entries, causing any code that expects the stacktrace to be in a certain order or contain specific items to fail.

The only exception to this rule is the class `error` with the reason `undef` which is guaranteed to include the `Module`, `Function` and `Arity` of the attempted function as the first stacktrace entry.

6.12.3 Handling of Run-time Errors in Erlang

Error Handling Within Processes

It is possible to prevent run-time errors and other exceptions from causing the process to terminate by using `catch` or `try`, see Expressions about catch and try.

Error Handling Between Processes

Processes can monitor other processes and detect process terminations, see Processes.

6.12.4 Exit Reasons

When a run-time error occurs, that is an exception of class `error`. The exit reason is a tuple `{Reason,Stack}`, where `Reason` is a term indicating the type of error:

Reason	Type of Error
`badarg`	Bad argument. The argument is of wrong data type, or is otherwise badly formed.
`badarith`	Bad argument in an arithmetic expression.
`{badmatch,V}`	Evaluation of a match expression failed. The value `V` did not match.
`function_clause`	No matching function clause is found when evaluating a function call.
`{case_clause,V}`	No matching branch is found when evaluating a `case` expression. The value `V` did not match.
`if_clause`	No true branch is found when evaluating an `if` expression.
`{try_clause,V}`	No matching branch is found when evaluating the of-section of a `try` expression. The value `V` did not match.
`undef`	The function cannot be found when evaluating a function call.
`{badfun,F}`	Something is wrong with a fun `F`.
`{badarity,F}`	A fun is applied to the wrong number of arguments. `F` describes the fun and the arguments.
`timeout_value`	The timeout value in a `receive..after` expression is evaluated to something else than an integer or `infinity`.
`noproc`	Trying to link or monitor to a non-existing process or port.
`noconnection`	A link or monitor to a remote process was broken because a connection between the nodes could not be established or was severed.
`{nocatch,V}`	Trying to evaluate a `throw` outside a `catch`. `V` is the thrown term.
`system_limit`	A system limit has been reached. See Efficiency Guide for information about system limits.

Table 12.2: Exit Reasons

`Stack` is the stack of function calls being evaluated when the error occurred, given as a list of tuples `{Module,Name,Arity,ExtraInfo}` with the most recent function call first. The most recent function call tuple can in some cases be `{Module,Name,[Arg],ExtraInfo}`.

6.13 Features

Introduced in OTP 25, Erlang has the concept of selectable features. A feature can change, add or remove behaviour of the language and/or runtime system. Examples can include

- Adding new syntactical constructs to the language
- Change the semantics of an existing construct
- Change the behaviour of some runtime aspect

A feature will start out with a status of experimental part of OTP, making it possible to try out for users and give feedback. The possibility to try out features is enabled by options to the compiler, directives in a module and options to the runtime system. Even when a feature is not experimental it will still be possible to enable or disable it. This makes it possible to adapt a code base at a suitable pace instead of being forced when changing to a new release.

The status of a feature will eventually end up as being either a permanent part of OTP or rejected, being removed and no longer selectable.

6.13.1 Life cycle of features

A feature is in one of four possible states:

Experimental
: The initial state, is meant for trying out and collecting feedback. The feature can be enabled but is disabled by default.

Approved
: The feature has been finalised and is now part of OTP. By default it is enabled, but can be disabled.

Permanent
: The feature is now a permanent part of OTP. It can no longer be disabled.

Rejected
: The feature never reached the approved state and will not be part of OTP. It cannot be enabled.

After leaving the experimental state, a feature can enter any of the other three states, and if the next state is approved, the feature will eventually end up in the permanent state. A feature can change state only in connection with a release.

A feature may be in the approved state for several releases.

State	Default	Configurable	Available
Experimental	disabled	yes	yes
Approved	enabled	yes	yes
Permanent	enabled	no	yes
Rejected	disabled	no	no

Table 13.1: Feature States

- Being configurable means the possibility to enable or disable the feature by means of compiler options and directives in the file being compiled.
- Being available can be seen using the `FEATURE_AVAILABLE` macro.

6.13.2 Enabling and Disabling Features

To use a feature that is in the experimental state, it has to be enabled during compilation. This can be done in a number of different ways:

Options to `erlc`
: Options `-enable-feature` and `-disable-feature` can be used to enable or disable individual features.

Compiler options
: The compiler option `{feature, <feature>, enable|disable}` can be used either as a `+<term>` option to `erlc` or in the options argument to functions in the `compile` module.

The feature directive
: Inside a prefix of a module, one can use a `-feature(<feature>, enable|disable)` directive. This is the preferred method of enabling and disabling features.

Note that to load a module compiled with features enabled, the corresponding features must be enabled in the runtime. This is done using options `-enable-feature` and `-disable-feature` to `erl`. This is to allow the possibility to prevent the use of experimental features in, e.g., production. This will catch experimental features used in both own and third party components. An active choice to use experimental features must be done.

6.13.3 Preprocessor Additions

To allow for conditional compilation during transitioning of a code base and/or trying out experimental features feature `predefined macros` `?FEATURE_AVAILABLE(Feature)` and `?FEATURE_ENABLED(Feature)` are available.

6.13.4 Information about Existing Features

The module `erl_features` `erl_features` exports a number of functions that can be used to obtain information about current features as well as the features used when compiling a module.

One can also use the `erlc` options `-list-features` and `-describe-feature <feature>` to get information about existing features.

Additionally, there is the compiler option `warn_keywords` that can be used to find atoms in the code base that might collide with keywords in features not yet enabled.

6.13.5 Existing Features

The following configurable features exist:

`maybe_expr` (experimental)
: Implementation of the `maybe` expression proposed in **EEP 49**.

6.14 Processes

6.14.1 Processes

Erlang is designed for massive concurrency. Erlang processes are lightweight (grow and shrink dynamically) with small memory footprint, fast to create and terminate, and the scheduling overhead is low.

6.14.2 Process Creation

A process is created by calling `spawn()`:

```
spawn(Module, Name, Args) -> pid()
  Module = Name = atom()
  Args = [Arg1,...,ArgN]
    ArgI = term()
```

`spawn()` creates a new process and returns the pid.

The new process starts executing in `Module:Name(Arg1,...,ArgN)` where the arguments are the elements of the (possible empty) `Args` argument list.

There exist a number of different `spawn` BIFs:

- `spawn/1,2,3,4`
- `spawn_link/1,2,3,4`
- `spawn_monitor/1,2,3,4`
- `spawn_opt/2,3,4,5`
- `spawn_request/1,2,3,4,5`

6.14.3 Registered Processes

Besides addressing a process by using its pid, there are also BIFs for registering a process under a name. The name must be an atom and is automatically unregistered if the process terminates:

BIF	**Description**
`register(Name, Pid)`	Associates the name `Name`, an atom, with the process `Pid`.
`registered()`	Returns a list of names that have been registered using `register/2`.
`whereis(Name)`	Returns the pid registered under `Name`, or `undefined` if the name is not registered.

Table 14.1: Name Registration BIFs

6.14.4 Process Aliases

When sending a message to a process, the receiving process can be identified by a PID, a registered name, or a *process alias* which is a term of the type reference. The typical use case that process aliases were designed for is a request/reply scenario. Using a process alias when sending the reply makes it possible for the receiver of the reply to prevent the reply from reaching its message queue if the operation times out or if the connection between the processes is lost.

A process alias can be used as identifier of the receiver when sending a message using the send operator `!` or send BIFs such as `erlang:send/2`. As long as the process alias is active, messages will be delivered the same way as if the process identifier of the process that created the alias had been used. When the alias has been deactivated, messages sent using the alias will be dropped before entering the message queue of the receiver. Note that messages that at deactivation time already have entered the message queue will **not** be removed.

A process alias is created either by calling one of the `alias/0,1` BIFs or by creating an alias and a monitor simultaneously. If the alias is created together with a monitor, the same reference will be used both as monitor reference and alias. Creating a monitor and an alias at the same time is done by passing the `{alias, _}` option to the `monitor/3` BIF. The `{alias, _}` option can also be passed when creating a monitor via `spawn_opt()`, or `spawn_request()`.

A process alias can be deactivated by the process that created it by calling the `unalias/1` BIF. It is also possible to automatically deactivate an alias on certain events. See the documentation of the `alias/1` BIF, and the `{alias, _}` option of the `monitor/3` BIF for more information about automatic deactivation of aliases.

It is **not** possible to:

- create an alias identifying another process than the caller.
- deactivate an alias unless it identifies the caller.
- look up an alias.
- look up the process identified by an alias.
- check if an alias is active or not.
- check if a reference is an alias.

These are all intentional design decisions relating to performance, scalability, and distribution transparency.

6.14.5 Process Termination

When a process terminates, it always terminates with an **exit reason**. The reason can be any term.

A process is said to terminate **normally**, if the exit reason is the atom `normal`. A process with no more code to execute terminates normally.

A process terminates with an exit reason `{Reason,Stack}` when a run-time error occurs. See Exit Reasons.

A process can terminate itself by calling one of the following BIFs:

- `exit(Reason)`
- `erlang:error(Reason)`
- `erlang:error(Reason, Args)`

The process then terminates with reason `Reason` for `exit/1` or `{Reason,Stack}` for the others.

A process can also be terminated if it receives an exit signal with another exit reason than `normal`, see Error Handling.

6.14.6 Signals

All communication between Erlang processes and Erlang ports is done by sending and receiving asynchronous signals. The most common signals are Erlang message signals. A message signal can be sent using the send operator `!`. A received message can be fetched from the message queue by the receiving process using the `receive` expression.

Synchronous communication can be broken down into multiple asynchronous signals. An example of such a synchronous communication is a call to the `erlang:process_info/2` BIF when the first argument does not equal the process identifier of the calling process. The caller sends an asynchronous signal requesting information, and then blocks waiting for the reply signal containing the requested information. When the request signal reaches its destination, the destination process replies with the requested information.

Sending Signals

There are many signals that processes and ports use to communicate. The list below contains the most important signals. In all the cases of request/reply signal pairs, the request signal is sent by the process calling the specific BIF, and the reply signal is sent back to it when the requested operation has been performed.

`message`
: Sent when using the send operator `!`, or when calling one of the `erlang:send/2,3` or `erlang:send_nosuspend/2,3` BIFs.

`link`
: Sent when calling the link/1 BIF.

`unlink`
: Sent when calling the unlink/1 BIF.

`exit`
: Sent either when explicitly sending an `exit` signal by calling the exit/2 BIF, or when a linked process terminates. If the signal is sent due to a link, the signal is sent after all *directly visible Erlang resources* used by the process have been released.

`monitor`
: Sent when calling one of the monitor/2,3 BIFs.

`demonitor`
: Sent when calling one of the demonitor/1,2 BIFs, or when a process monitoring another process terminates.

`down`
: Sent by a monitored process or port that terminates. The signal is sent after all *directly visible Erlang resources* used by the process or the port have been released.

`change`
: Sent by the clock service on the local runtime system, when the time offset changes, to processes which have monitored the `time_offset`.

`group_leader`
: Sent when calling the group_leader/2 BIF.

`spawn_request`/`spawn_reply`, `open_port_request`/`open_port_reply`
: Sent due to a call to one of the `spawn/1,2,3,4`, `spawn_link/1,2,3,4`, `spawn_monitor/1,2,3,4`, `spawn_opt/2,3,4,5`, `spawn_request/1,2,3,4,5`, or `erlang:open_port/2` BIFs. The request signal is sent to the spawn service which responds with the reply signal.

`alive_request`/`alive_reply`
: Sent due to a call to the is_process_alive/1 BIF.

`garbage_collect_request`/`garbage_collect_reply`, `check_process_code_request`/`check_process_code_reply`, `process_info_request`/`process_info_reply`
: Sent due to a call to one of the garbage_collect/1,2, erlang:check_process_code/2,3, or process_info/1,2 BIFs. Note that if the request is directed towards the caller itself and it is a synchronous request, no signaling will be performed and the caller will instead synchronously perform the request before returning from the BIF.

`port_command`, `port_connect`, `port_close`
: Sent by a process to a port on the local node using the send operator `!`, or by calling one of the `send()` BIFs. The signal is sent by passing a term on the format `{Owner, {command, Data}}`, `{Owner, {connect, Pid}}`, or `{Owner, close}` as message.

`port_command_request`/`port_command_reply`, `port_connect_request`/`port_connect_reply`, `port_close_request`/`port_close_reply`, `port_control_request`/`port_control_reply`, `port_call_request`/`port_call_reply`, `port_info_request`/`port_info_reply`
: Sent due to a call to one of the `erlang:port_command/2,3`, `erlang:port_connect/2`, `erlang:port_close/1`, `erlang:port_control/3`, `erlang:port_call/3`, `erlang:port_info/1,2` BIFs. The request signal is sent to a port on the local node which responds with the reply signal.

`register_name_request`/`register_name_reply`, `unregister_name_request`/`unregister_name_reply`, `whereis_name_request`/`whereis_name_reply`
: Sent due to a call to one of the `register/2`, `unregister/1`, or `whereis/1` BIFs. The request signal is sent to the name service, which responds with the reply signal.

`timer_start_request`/`timer_start_reply`, `timer_cancel_request`/`timer_cancel_reply`
: Sent due to a call to one of the `erlang:send_after/3,4`, `erlang:start_timer/3,4`, or `erlang:cancel_timer/1,2` BIFs. The request signal is sent to the timer service which responds with the reply signal.

The clock service, the name service, the timer service, and the spawn service mentioned previously are services provided by the runtime system. Each of these services consists of multiple independently executing entities. Such a service can be viewed as a group of processes, and could actually be implemented like that. Since each service consists of multiple independently executing entities, the order between multiple signals sent from one service to one process is **not** preserved. Note that this does **not** violate the signal ordering guarantee of the language.

The realization of the signals described above may change both at runtime and due to changes in implementation. You may be able to detect such changes using `receive` tracing or by inspecting message queues. However, these are internal implementation details of the runtime system that you should **not** rely on. As an example, many of the reply signals above are ordinary message signals. When the operation is synchronous, the reply signals do not have to be message signals. The current implementation takes advantage of this and, depending on the state of the system, use alternative ways of delivering the reply signals. The implementation of these reply signals may also, at any time, be changed to not use message signals where it previously did.

Receiving Signals

Signals are received asynchronously and automatically. There is nothing a process must do to handle the reception of signals, or can do to prevent it. In particular, signal reception is **not** tied to the execution of a `receive` expression, but can happen anywhere in the execution flow of a process.

When a signal is received by a process, some kind of action is taken. The specific action taken depends on the signal type, contents of the signal, and the state of the receiving process. Actions taken for the most common signals:

`message`
: If the message signal was sent using a process alias that is no longer active, the message signal will be dropped; otherwise, if the alias is still active or the message signal was sent by other means, the message is added to the end of the message queue. When the message has been added to the message queue, the receiving process can fetch the message from the message queue using the `receive` expression.

`link`, `unlink`
: Very simplified it can be viewed as updating process local information about the link. A detailed description of the link protocol can be found in the *Distribution Protocol* chapter of the *ERTS User's Guide*.

`exit`
: Set the receiver in an exiting state, drop the signal, or convert the signal into a message and add it to the end of the message queue. If the receiver is set in an exiting state, no more Erlang code will be executed and the process is scheduled for termination. The section *Receiving Exit Signals* below gives more details on the action taken when an `exit` signal is received.

`monitor`, `demonitor`
: Update process local information about the monitor.

`down`, `change`
: Convert into a message if the corresponding monitor is still active; otherwise, drop the signal. If the signal is converted into a message, it is also added to the end of the message queue.

`group_leader`
: Change the group leader of the process.

`spawn_reply`
: Convert into a message, or drop the signal depending on the reply and how the `spawn_request` signal was configured. If the signal is converted into a message it is also added to the end of the message queue. For more information see the `spawn_request()` BIF.

`alive_request`
: Schedule execution of the *is alive* test. If the process is in an exiting state, the *is alive* test will not be executed until after all *directly visible Erlang resources* used by the process have been released. The `alive_reply` will be sent after the *is alive* test has executed.

`process_info_request`, `garbage_collect_request`, `check_process_code_request`
: Schedule execution of the requested operation. The reply signal will be sent when the operation has been executed.

Note that some actions taken when a signal is received involves **scheduling** further actions which will result in a reply signal when these scheduled actions have completed. This implies that the reply signals may be sent in a different order than the order of the incoming signals that triggered these operations. This does, however, **not** violate the signal ordering guarantee of the language.

The order of messages in the message queue of a process reflects the order in which the signals corresponding to the messages has been received since all signals that add messages to the message queue add them at the end of the message queue. Messages corresponding to signals from the same sender are also ordered in the same order as the signals were sent due to the signal ordering guarantee of the language.

Directly Visible Erlang Resources

As described above, `exit` signals due to links, `down` signals, and reply signals from an exiting process due to `alive_request`s are not sent until all *directly visible Erlang resources* held by the terminating process have been released. With *directly visible Erlang resources* we here mean all resources made available by the language excluding resources held by heap data, dirty native code execution and the process identifier of the terminating process. Examples of *directly visible Erlang resources* are registered name and ETS tables.

The Excluded Resources

The process identifier of the process cannot be released for reuse until everything regarding the process has been released.

A process executing dirty native code in a NIF when it receives an exit signal will be set into an exiting state even if it is still executing dirty native code. *Directly visible Erlang resources* will be released, but the runtime system cannot force the native code to stop executing. The runtime system tries to prevent the execution of the dirty native code from effecting other processes by, for example, disabling functionality such as `enif_send()` when used from a terminated process, but if the NIF is not well behaved it can still effect other processes. A well behaved dirty NIF should test if the process it is executing in has exited, and if so stop executing.

In the general case, the heap of a process cannot be removed before all signals that it needs to send have been sent. Resources held by heap data are the memory blocks containing the heap, but also include things referred to from the heap such as off heap binaries, and resources held via NIF resource objects on the heap.

Delivery of Signals

The amount of time that passes between the time a signal is sent and the arrival of the signal at the destination is unspecified but positive. If the receiver has terminated, the signal does not arrive, but it can trigger another signal. For example, a `link` signal sent to a non-existing process triggers an `exit` signal, which is sent back to where the `link` signal originated from. When communicating over the distribution, signals can be lost if the distribution channel goes down.

The only signal ordering guarantee given is the following: if an entity sends multiple signals to the same destination entity, the order is preserved; that is, if `A` sends a signal `S1` to `B`, and later sends signal `S2` to `B`, `S1` is guaranteed not to arrive after `S2`. Note that `S1` may, or may not have been lost.

Irregularities

Synchronous Error Checking

Some functionality that send signals have synchronous error checking when sending locally on a node and fail if the receiver is not present at the time when the signal is sent:

- The send operator `!`, `erlang:send/2,3`, BIFs and `erlang:send_nosuspend/2,3` BIFs when the receiver is identified by a name that is expected to be registered locally.
- `erlang:link/1`
- `erlang:group_leader/2`

Unexpected Behaviours of Exit Signals

When a process sends an exit signal with exit reason `normal` to itself by calling `erlang:exit(self(), normal)` it will be terminated when the `exit` signal is received. In all other cases when an exit signal with exit reason `normal` is received, it is dropped.

When an `exit` signal with exit reason `kill` is received, the action taken is different depending on whether the signal was sent due to a linked process terminating, or the signal was explicitly sent using the `exit/2` BIF. When sent using the `exit/2` BIF, the signal cannot be trapped, while it can be trapped if the signal was sent due to a link.

Blocking Signaling Over Distribution

When sending a signal over a distribution channel, the sending process may be suspended even though the signal is supposed to be sent asynchronously. This is due to the built in flow control over the channel that has been present more or less for ever. When the size of the output buffer for the channel reach the *distribution buffer busy limit*, processes sending on the channel will be suspended until the size of the buffer shrinks below the limit. The size of the limit can be inspected by calling `erlang:system_info(dist_buf_busy_limit)`. Since this functionality has been present for so long, it is not possible to remove it, but it is possible to increase the limit to a point where it more or less never is reached using the `erl` command line argument `+zdbbl`. Note that if you do raise the limit like this, you need to take care of flow control yourself to ensure that you do not get into a situation with excessive memory usage.

The irregularities mentioned above cannot be fixed as they have been part of Erlang too long and it would break a lot of existing code.

6.14.7 Links

Two processes can be **linked** to each other. Also a process and a port that reside on the same node can be linked to each other. A link between two processes can be created if one of them calls the `link/1` BIF with the process identifier of the other process as argument. Links can also be created using one the following spawn BIFs `spawn_link()`, `spawn_opt()`, or `spawn_request()`. The spawn operation and the link operation will be performed atomically, in these cases.

If one of the participants of a link terminates, it will send an exit signal to the other participant. The exit signal will contain the exit reason of the terminated participant.

A link can be removed by calling the `unlink/1` BIF.

Links are bidirectional and there can only be one link between two processes. Repeated calls to `link()` have no effect. Either one of the involved processes may create or remove a link.

Links are used to monitor the behavior of other processes, see Error Handling.

6.14.8 Error Handling

Erlang has a built-in feature for error handling between processes. Terminating processes emit exit signals to all linked processes, which can terminate as well or handle the exit in some way. This feature can be used to build hierarchical program structures where some processes are supervising other processes, for example, restarting them if they terminate abnormally.

See OTP Design Principles for more information about OTP supervision trees, which use this feature.

Sending Exit Signals

When a process or port terminates it will send exit signals to all processes and ports that it is linked to. The exit signal will contain the following information:

Sender identifier

The process or port identifier of the process or port that terminated.

Receiver identifier

The process or port identifier of the process or port which the exit signal is sent to.

The `link` flag

This flag will be set indicating that the exit signal was sent due to a link.

Exit reason

The exit reason of the process or port that terminated or the atom:

- `noproc` in case no process or port was found when setting up a link in a preceding call to the `link(PidOrPort)` BIF. The process or port identified as sender of the exit signal will equal the `PidOrPort` argument passed to `link/1`.
- `noconnection` in case the linked processes resides on different nodes and the connection between the nodes was lost or could not be established. The process or port identified as sender of the exit signal might in this case still be alive.

Exit signals can also be sent explicitly by calling the `exit(PidOrPort, Reason)` BIF. The exit signal is sent to the process or port identified by the `PidOrPort` argument. The exit signal sent will contain the following information:

Sender identifier

The process identifier of the process that called `exit/2`.

Receiver identifier

The process or port identifier of the process or port which the exit signal is sent to.

The `link` flag

This flag will not be set, indicating that this exit signal was not sent due to a link.

Exit reason

The term passed as `Reason` in the call to `exit/2`. If `Reason` is the atom `kill`, the receiver cannot trap the exit signal and will unconditionally terminate when it receives the signal.

Receiving Exit Signals

What happens when a process receives an exit signal depends on:

- The trap exit state of the receiver at the time when the exit signal is received.
- The exit reason of the exit signal.
- The sender of the exit signal.
- The state of the `link` flag of the exit signal. If the `link` flag is set, the exit signal was sent due to a link; otherwise, the exit signal was sent by a call to the `exit/2` BIF.
- If the `link` flag is set, what happens also depends on whether the link is still active or not when the exit signal is received.

Based on the above states, the following will happen when an exit signal is received by a process:

- The exit signal is silently dropped if:
  - the `link` flag of the exit signal is set and the corresponding link has been deactivated.
  - the exit reason of the exit signal is the atom `normal`, the receiver is not trapping exits, and the receiver and sender are not the same process.
- The receiving process is terminated if:
  - the `link` flag of the exit signal is not set, and the exit reason of the exit signal is the atom `kill`. The receiving process will terminate with the atom `killed` as exit reason.

  - the receiver is not trapping exits, and the exit reason is something other than the atom `normal`. Also, if the `link` flag of the exit signal is set, the link also needs to be active otherwise the exit signal will be dropped. The exit reason of the receiving process will equal the exit reason of the exit signal. Note that if the `link` flag is set, an exit reason of `kill` will **not** be converted to `killed`.
  - the exit reason of the exit signal is the atom `normal` and the sender of the exit signal is the same process as the receiver. The `link` flag cannot be set in this case. The exit reason of the receiving process will be the atom `normal`.
- The exit signal is converted to a message signal and added to the end of the message queue of the receiver, if the receiver is trapping exits, the `link` flag of the exit signal is:
  - not set, and the exit reason of the signal is not the atom `kill`.
  - set, and the corresponding link is active. Note that an exit reason of `kill` will **not** terminate the process in this case and it will not be converted to `killed`.

  The converted message will be on the form `{'EXIT', SenderID, Reason}` where `Reason` equals the exit reason of the exit signal and `SenderID` is the identifier of the process or port that sent the exit signal.

6.14.9 Monitors

An alternative to links are **monitors**. A process `Pid1` can create a monitor for `Pid2` by calling the BIF `erlang:monitor(process, Pid2)`. The function returns a reference `Ref`.

If `Pid2` terminates with exit reason `Reason`, a 'DOWN' message is sent to `Pid1`:

```
{'DOWN', Ref, process, Pid2, Reason}
```

If `Pid2` does not exist, the 'DOWN' message is sent immediately with `Reason` set to `noproc`.

Monitors are unidirectional. Repeated calls to `erlang:monitor(process, Pid)` creates several independent monitors, and each one sends a 'DOWN' message when `Pid` terminates.

A monitor can be removed by calling `erlang:demonitor(Ref)`.

Monitors can be created for processes with registered names, also at other nodes.

6.14.10 Process Dictionary

Each process has its own process dictionary, accessed by calling the following BIFs:

```
put(Key, Value)
get(Key)
get()
get_keys(Value)
erase(Key)
erase()
```

6.15 Distributed Erlang

6.15.1 Distributed Erlang System

A **distributed Erlang system** consists of a number of Erlang runtime systems communicating with each other. Each such runtime system is called a **node**. Message passing between processes at different nodes, as well as links and monitors, are transparent when pids are used. Registered names, however, are local to each node. This means that the node must be specified as well when sending messages, and so on, using registered names.

The distribution mechanism is implemented using TCP/IP sockets. How to implement an alternative carrier is described in the ERTS User's Guide.

Warning:

Starting a distributed node without also specifying `-proto_dist inet_tls` will expose the node to attacks that may give the attacker complete access to the node and in extension the cluster. When using un-secure distributed nodes, make sure that the network is configured to keep potential attackers out. See the Using SSL for Erlang Distribution User's Guide for details on how to setup a secure distributed node.

6.15.2 Nodes

A **node** is an executing Erlang runtime system that has been given a name, using the command-line flag `-name` (long names) or `-sname` (short names).

The format of the node name is an atom `name@host`. `name` is the name given by the user. `host` is the full host name if long names are used, or the first part of the host name if short names are used. Function `node()` returns the name of the node.

Example:

```
% erl -name dilbert
(dilbert@uab.ericsson.se)1> node().
'dilbert@uab.ericsson.se'

% erl -sname dilbert
(dilbert@uab)1> node().
dilbert@uab
```

The node name can also be given in runtime by calling `net_kernel:start/1`.

Example:

```
% erl
1> node().
nonode@nohost
2> net_kernel:start([dilbert,shortnames]).
{ok,<0.102.0>}
(dilbert@uab)3> node().
dilbert@uab
```

Note:

A node with a long node name cannot communicate with a node with a short node name.

6.15.3 Node Connections

The nodes in a distributed Erlang system are loosely connected. The first time the name of another node is used, for example, if `spawn(Node,M,F,A)` or `net_adm:ping(Node)` is called, a connection attempt to that node is made.

Connections are by default transitive. If a node A connects to node B, and node B has a connection to node C, then node A also tries to connect to node C. This feature can be turned off by using the command-line flag `-connect_all false`, see the erl(1) manual page in ERTS.

If a node goes down, all connections to that node are removed. Calling `erlang:disconnect_node(Node)` forces disconnection of a node.

The list of (visible) nodes currently connected to is returned by `nodes()`.

6.15.4 epmd

The Erlang Port Mapper Daemon **epmd** is automatically started at every host where an Erlang node is started. It is responsible for mapping the symbolic node names to machine addresses. See the epmd(1) manual page in ERTS.

6.15.5 Hidden Nodes

In a distributed Erlang system, it is sometimes useful to connect to a node without also connecting to all other nodes. An example is some kind of O&M functionality used to inspect the status of a system, without disturbing it. For this purpose, a **hidden node** can be used.

A hidden node is a node started with the command-line flag `-hidden`. Connections between hidden nodes and other nodes are not transitive, they must be set up explicitly. Also, hidden nodes does not show up in the list of nodes returned by `nodes()`. Instead, `nodes(hidden)` or `nodes(connected)` must be used. This means, for example, that the hidden node is not added to the set of nodes that `global` is keeping track of.

6.15.6 Dynamic Node Name

If the node name is set to **`undefined`** the node will be started in a special mode to be the temporary client of another node. The node will then request a dynamic node name from the first node it connects to. In addition these distribution settings will be set:

```
-dist_listen false -hidden -kernel dist_auto_connect never
```

As `-dist_auto_connect` is set to `never`, `net_kernel:connect_node/1` must be called in order to setup connections. If the first established connection is closed (which gave the node its dynamic name), then any other connections will also be closed and the node will lose its dynamic node name. A new call to `net_kernel:connect_node/1` can be made to get a new dynamic node name. The node name may change if the distribution is dropped and then set up again.

> **Note:**
>
> The **dynamic node name** feature is supported from OTP 23. Both the temporary client node and the first connected peer node (supplying the dynamic node name) must be at least OTP 23 for it to work.

6.15.7 C Nodes

A **C node** is a C program written to act as a hidden node in a distributed Erlang system. The library **Erl_Interface** contains functions for this purpose. For more information about C nodes, see the Erl_Interface application and Interoperability Tutorial..

6.15.8 Security

Note:

"Security" here does **not** mean cryptographically secure, but rather security against accidental misuse, such as preventing a node from connecting to a cluster with which it is not intended to communicate.

Furthermore, the communication between nodes is per default in clear text. If you need strong security, please see Using TLS for Erlang Distribution in the SSL application's User's Guide.

Also, the default random cookie mentioned in the following text is not very unpredictable. A better one can be generated using primitives in the `crypto` module, though this still does not make the initial handshake cryptographically secure. And inter-node communication is still in clear text.

Authentication determines which nodes are allowed to communicate with each other. In a network of different Erlang nodes, it is built into the system at the lowest possible level. All nodes use a **magic cookie**, which is an Erlang atom, when connecting another node.

During the connection setup, after node names have been exchanged, the magic cookies the nodes present to each other are compared. If they do not match, the connection is rejected. The cookies themselves are never transferred, instead they are compared using hashed challenges, although not in a cryptographically secure manner.

At start-up, a node has a random atom assigned as its default magic cookie and the cookie of other nodes is assumed to be `nocookie`. The first action of the Erlang network authentication server (`auth`) is then to search for a file named `.erlang.cookie` in the user's home directory and then in `filename:basedir(user_config, "erlang")`. If none of the files exist, a `.erlang.cooke` file is created in the user's home directory. The UNIX permissions mode of the file is set to octal 400 (read-only by user) and its content is a random string. An atom `Cookie` is created from the contents of the file and the cookie of the local node is set to this using `erlang:set_cookie(Cookie)`. This sets the default cookie that the local node will use for all other nodes.

Thus, groups of users with identical cookie files get Erlang nodes that can communicate freely since they use the same magic cookie. Users who want to run nodes where the cookie files are on different file systems must make certain that their cookie files are identical.

For a node `Node1` using magic cookie `Cookie` to be able to connect to, and to accept a connection from, another node `Node2` that uses a different cookie `DiffCookie`, the function `erlang:set_cookie(Node2, DiffCookie)` must first be called at `Node1`. Distributed systems with multiple home directories (differing cookie files) can be handled in this way.

Note:

With this setup `Node1` and `Node2` agree on which cookie to use: `Node1` uses its explicitly configured `DiffCookie` for `Node2`, and `Node2` uses its default cookie `DiffCookie`.

You can also use a `DiffCookie` that neither `Node1` nor `Node2` has as its default cookie, if you also call `erlang:set_cookie(Node1, DiffCookie)` in `Node2` before establishing connection

Because node names are exchanged during connection setup before cookies are selected, connection setup works regardless of which node that initiates it.

Note that to configure `Node1` to use `Node2`'s default cookie when communicating with `Node2`, **and vice versa** results in a broken configuration (if the cookies are different) because then both nodes use the other node's (differing) cookie.

The default when a connection is established between two nodes, is to immediately connect all other visible nodes as well. This way, there is always a fully connected network. If there are nodes with different cookies, this method can

be inappropriate (since it may not be feasible to configure different cookies for all possible nodes) and the command-line flag `-connect_all false` must be set, see the erl(1) manual page in ERTS.

The magic cookie of the local node can be retrieved by calling `erlang:get_cookie()`.

6.15.9 Distribution BIFs

Some useful BIFs for distributed programming (for more information, see the erlang(3) manual page in ERTS:

BIF	**Description**
`erlang:disconnect_node(Node)`	Forces the disconnection of a node.
`erlang:get_cookie()`	Returns the magic cookie of the current node.
`erlang:get_cookie(Node)`	Returns the magic cookie for node `Node`.
`is_alive()`	Returns `true` if the runtime system is a node and can connect to other nodes, `false` otherwise.
`monitor_node(Node, true|false)`	Monitors the status of `Node`. A message `{nodedown, Node}` is received if the connection to it is lost.
`node()`	Returns the name of the current node. Allowed in guards.
`node(Arg)`	Returns the node where `Arg`, a pid, reference, or port, is located.
`nodes()`	Returns a list of all visible nodes this node is connected to.
`nodes(Arg)`	Depending on `Arg`, this function can return a list not only of visible nodes, but also hidden nodes and previously known nodes, and so on.
`erlang:set_cookie(Cookie)`	Sets the magic cookie, `Cookie` to use when connecting all nodes that have no explicit cookie set with `erlang:set_cookie/2`.
`erlang:set_cookie(Node, Cookie)`	Sets the magic cookie used when connecting `Node`. If `Node` is the current node, `Cookie` is used when connecting all nodes that have no explicit cookie set with this function.
`spawn[_link|_opt](Node, Fun)`	Creates a process at a remote node.
`spawn[_link|opt](Node, Module, FunctionName, Args)`	Creates a process at a remote node.

Table 15.1: Distribution BIFs

6.15.10 Distribution Command-Line Flags

Examples of command-line flags used for distributed programming (for more information, see the erl(1) manual page in ERTS:

Command-Line Flag	Description
`-connect_all false`	Only explicit connection set-ups are used.
`-hidden`	Makes a node into a hidden node.
`-name Name`	Makes a runtime system into a node, using long node names.
`-setcookie Cookie`	Same as calling `erlang:set_cookie(Cookie)`.
`-setcookie Node Cookie`	Same as calling `erlang:set_cookie(Node, Cookie)`.
`-sname Name`	Makes a runtime system into a node, using short node names.

Table 15.2: Distribution Command-Line Flags

6.15.11 Distribution Modules

Examples of modules useful for distributed programming:

In the Kernel application:

Module	Description
`global`	A global name registration facility.
`global_group`	Grouping nodes to global name registration groups.
`net_adm`	Various Erlang net administration routines.
`net_kernel`	Erlang networking kernel.

Table 15.3: Kernel Modules Useful For Distribution.

In the STDLIB application:

Module	Description
`slave`	Start and control of slave nodes.

Table 15.4: STDLIB Modules Useful For Distribution.

6.16 Compilation and Code Loading

How code is compiled and loaded is not a language issue, but is system-dependent. This section describes compilation and code loading in Erlang/OTP with references to relevant parts of the documentation.

6.16.1 Compilation

Erlang programs must be **compiled** to object code. The compiler can generate a new file that contains the object code. The current abstract machine, which runs the object code, is called BEAM, therefore the object files get the suffix `.beam`. The compiler can also generate a binary which can be loaded directly.

The compiler is located in the module `compile` (see the compile(3) manual page in Compiler).

```
compile:file(Module)
compile:file(Module, Options)
```

The Erlang shell understands the command `c(Module)` which both compiles and loads `Module`.

There is also a module `make`, which provides a set of functions similar to the UNIX type Make functions, see the make(3) manual page in Tools.

The compiler can also be accessed from the OS prompt, see the erl(1) manual page in ERTS.

```
% erl -compile Module1...ModuleN
% erl -make
```

The `erlc` program provides an even better way to compile modules from the shell, see the erlc(1) manual page in ERTS. It understands a number of flags that can be used to define macros, add search paths for include files, and more.

```
% erlc <flags> File1.erl...FileN.erl
```

6.16.2 Code Loading

The object code must be **loaded** into the Erlang runtime system. This is handled by the **code server**, see the code(3) manual page in Kernel.

The code server loads code according to a code loading strategy, which is either **interactive** (default) or **embedded**. In interactive mode, code is searched for in a **code path** and loaded when first referenced. In embedded mode, code is loaded at start-up according to a **boot script**. This is described in System Principles .

6.16.3 Code Replacement

Erlang supports change of code in a running system. Code replacement is done on module level.

The code of a module can exist in two variants in a system: **current** and **old**. When a module is loaded into the system for the first time, the code becomes 'current'. If then a new instance of the module is loaded, the code of the previous instance becomes 'old' and the new instance becomes 'current'.

Both old and current code is valid, and can be evaluated concurrently. Fully qualified function calls always refer to current code. Old code can still be evaluated because of processes lingering in the old code.

If a third instance of the module is loaded, the code server removes (purges) the old code and any processes lingering in it is terminated. Then the third instance becomes 'current' and the previously current code becomes 'old'.

To change from old code to current code, a process must make a fully qualified function call.

Example:

```
-module(m).
-export([loop/0]).

loop() ->
    receive
        code_switch ->
            m:loop();
        Msg ->
            ...
            loop()
    end.
```

To make the process change code, send the message `code_switch` to it. The process then makes a fully qualified call to `m:loop()` and changes to current code. Notice that `m:loop/0` must be exported.

For code replacement of funs to work, use the syntax `fun Module:FunctionName/Arity`.

6.16.4 Running a Function When a Module is Loaded

The `-on_load()` directive names a function that is to be run automatically when a module is loaded.

Its syntax is as follows:

```
-on_load(Name/0).
```

It is not necessary to export the function. It is called in a freshly spawned process (which terminates as soon as the function returns).

The function must return `ok` if the module is to become the new current code for the module and become callable.

Returning any other value or generating an exception causes the new code to be unloaded. If the return value is not an atom, a warning error report is sent to the error logger.

If there already is current code for the module, that code will remain current and can be called until the `on_load` function has returned. If the `on_load` function fails, the current code (if any) will remain current. If there is no current code for a module, any process that makes an external call to the module before the `on_load` function has finished will be suspended until the `on_load` function have finished.

Note:

Before OTP 19, if the `on_load` function failed, any previously current code would become old, essentially leaving the system without any working and reachable instance of the module. That problem has been eliminated in OTP 19.

In embedded mode, first all modules are loaded. Then all `on_load` functions are called. The system is terminated unless all of the `on_load` functions return `ok`.

Example:

```
-module(m).
-on_load(load_my_nifs/0).

load_my_nifs() ->
    NifPath = ...,    %Set up the path to the NIF library.
    Info = ...,       %Initialize the Info term
    erlang:load_nif(NifPath, Info).
```

If the call to `erlang:load_nif/2` fails, the module is unloaded and a warning report is sent to the error loader.

6.17 Ports and Port Drivers

Examples of how to use ports and port drivers are provided in Interoperability Tutorial. For information about the BIFs mentioned, see the erlang(3) manual page in ERTS.

6.17.1 Ports

Ports provide the basic mechanism for communication with the external world, from Erlang's point of view. They provide a byte-oriented interface to an external program. When a port has been created, Erlang can communicate with it by sending and receiving lists of bytes, including binaries.

The Erlang process creating a port is said to be the **port owner**, or the **connected process** of the port. All communication to and from the port must go through the port owner. If the port owner terminates, so does the port (and the external program, if it is written correctly).

The external program resides in another OS process. By default, it reads from standard input (file descriptor 0) and writes to standard output (file descriptor 1). The external program is to terminate when the port is closed.

6.17.2 Port Drivers

It is possible to write a driver in C according to certain principles and dynamically link it to the Erlang runtime system. The linked-in driver looks like a port from the Erlang programmer's point of view and is called a **port driver**.

Warning:

An erroneous port driver causes the entire Erlang runtime system to leak memory, hang or crash.

For information about port drivers, see the erl_driver(4) manual page in ERTS, driver_entry(1) manual page in ERTS, and erl_ddll(3) manual page in Kernel.

6.17.3 Port BIFs

To create a port:

`open_port(PortName, PortSettings`	Returns a port identifier `Port` as the result of opening a new Erlang port. Messages can be sent to, and received from, a port identifier, just like a pid. Port identifiers can also be linked to using `link/1`, or registered under a name using `register/2`.

Table 17.1: Port Creation BIF

`PortName` is usually a tuple `{spawn,Command}`, where the string `Command` is the name of the external program. The external program runs outside the Erlang workspace, unless a port driver with the name `Command` is found. If `Command` is found, that driver is started.

`PortSettings` is a list of settings (options) for the port. The list typically contains at least a tuple `{packet,N}`, which specifies that data sent between the port and the external program are preceded by an N-byte length indicator. Valid values for N are 1, 2, or 4. If binaries are to be used instead of lists of bytes, the option `binary` must be included.

The port owner `Pid` can communicate with the port `Port` by sending and receiving messages. (In fact, any process can send the messages to the port, but the port owner must be identified in the message).

As of Erlang/OTP R16, messages sent to ports are delivered truly asynchronously. The underlying implementation previously delivered messages to ports synchronously. Message passing has however always been documented as an

asynchronous operation. Hence, this is not to be an issue for an Erlang program communicating with ports, unless false assumptions about ports have been made.

In the following tables of examples, `Data` must be an I/O list. An I/O list is a binary or a (possibly deep) list of binaries or integers in the range 0..255:

Message	**Description**
`{Pid,{command,Data}}`	Sends `Data` to the port.
`{Pid,close}`	Closes the port. Unless the port is already closed, the port replies with `{Port,closed}` when all buffers have been flushed and the port really closes.
`{Pid,{connect,NewPid}}`	Sets the port owner of `Port` to `NewPid`. Unless the port is already closed, the port replies with`{Port,connected}` to the old port owner. Note that the old port owner is still linked to the port, but the new port owner is not.

Table 17.2: Messages Sent To a Port

Message	**Description**
`{Port,{data,Data}}`	`Data` is received from the external program.
`{Port,closed}`	Reply to `Port ! {Pid,close}`.
`{Port,connected}`	Reply to `Port ! {Pid,{connect,NewPid}}`.
`{'EXIT',Port,Reason}`	If the port has terminated for some reason.

Table 17.3: Messages Received From a Port

Instead of sending and receiving messages, there are also a number of BIFs that can be used:

Port BIF	**Description**
`port_command(Port,Data)`	Sends `Data` to the port.
`port_close(Port)`	Closes the port.
`port_connect(Port,NewPid)`	Sets the port owner of `Port`to `NewPid`. The old port owner `Pid` stays linked to the port and must call `unlink(Port)` if this is not desired.
`erlang:port_info(Port,Item)`	Returns information as specified by `Item`.

`erlang:ports()`	Returns a list of all ports on the current node.

Table 17.4: Port BIFs

Some additional BIFs that apply to port drivers: `port_control/3` and `erlang:port_call/3`.

7 Programming Examples

This section contains examples on using records, funs, list comprehensions, and the bit syntax.

7.1 Records

7.1.1 Records and Tuples

The main advantage of using records rather than tuples is that fields in a record are accessed by name, whereas fields in a tuple are accessed by position. To illustrate these differences, suppose that you want to represent a person with the tuple `{Name, Address, Phone}`.

To write functions that manipulate this data, remember the following:

- The `Name` field is the first element of the tuple.
- The `Address` field is the second element.
- The `Phone` field is the third element.

For example, to extract data from a variable `P` that contains such a tuple, you can write the following code and then use pattern matching to extract the relevant fields:

```
Name = element(1, P),
Address = element(2, P),
...
```

Such code is difficult to read and understand, and errors occur if the numbering of the elements in the tuple is wrong. If the data representation of the fields is changed, by re-ordering, adding, or removing fields, all references to the person tuple must be checked and possibly modified.

Records allow references to the fields by name, instead of by position. In the following example, a record instead of a tuple is used to store the data:

```
-record(person, {name, phone, address}).
```

This enables references to the fields of the record by name. For example, if `P` is a variable whose value is a `person` record, the following code access the name and address fields of the records:

```
Name = P#person.name,
Address = P#person.address,
...
```

Internally, records are represented using tagged tuples:

```
{person, Name, Phone, Address}
```

7.1.2 Defining a Record

This following definition of a `person` is used in several examples in this section. Three fields are included, `name`, `phone`, and `address`. The default values for `name` and `phone` is "" and [], respectively. The default value for `address` is the atom `undefined`, since no default value is supplied for this field:

```
-record(person, {name = "", phone = [], address}).
```

The record must be defined in the shell to enable use of the record syntax in the examples:

```
> rd(person, {name = "", phone = [], address}).
person
```

This is because record definitions are only available at compile time, not at runtime. For details on records in the shell, see the shell(3) manual page in STDLIB.

7.1.3 Creating a Record

A new `person` record is created as follows:

```
> #person{phone=[0,8,2,3,4,3,1,2], name="Robert"}.
#person{name = "Robert",phone = [0,8,2,3,4,3,1,2],address = undefined}
```

As the `address` field was omitted, its default value is used.

From Erlang 5.1/OTP R8B, a value to all fields in a record can be set with the special field _. _ means "all fields not explicitly specified".

Example:

```
> #person{name = "Jakob", _ = '_'}.
#person{name = "Jakob",phone = '_',address = '_'}
```

It is primarily intended to be used in `ets:match/2` and `mnesia:match_object/3`, to set record fields to the atom `'_'`. (This is a wildcard in `ets:match/2`.)

7.1.4 Accessing a Record Field

The following example shows how to access a record field:

```
> P = #person{name = "Joe", phone = [0,8,2,3,4,3,1,2]}.
#person{name = "Joe",phone = [0,8,2,3,4,3,1,2],address = undefined}
> P#person.name.
"Joe"
```

7.1.5 Updating a Record

The following example shows how to update a record:

```
> P1 = #person{name="Joe", phone=[1,2,3], address="A street"}.
#person{name = "Joe",phone = [1,2,3],address = "A street"}
> P2 = P1#person{name="Robert"}.
#person{name = "Robert",phone = [1,2,3],address = "A street"}
```

7.1.6 Type Testing

The following example shows that the guard succeeds if `P` is record of type `person`:

```
foo(P) when is_record(P, person) -> a_person;
foo(_) -> not_a_person.
```

7.1.7 Pattern Matching

Matching can be used in combination with records, as shown in the following example:

```
> P3 = #person{name="Joe", phone=[0,0,7], address="A street"}.
#person{name = "Joe",phone = [0,0,7],address = "A street"}
> #person{name = Name} = P3, Name.
"Joe"
```

The following function takes a list of `person` records and searches for the phone number of a person with a particular name:

```
find_phone([#person{name=Name, phone=Phone} | _], Name) ->
    {found,  Phone};
find_phone([_| T], Name) ->
    find_phone(T, Name);
find_phone([], Name) ->
    not_found.
```

The fields referred to in the pattern can be given in any order.

7.1.8 Nested Records

The value of a field in a record can be an instance of a record. Retrieval of nested data can be done stepwise, or in a single step, as shown in the following example:

```
-record(name, {first = "Robert", last = "Ericsson"}).
-record(person, {name = #name{}, phone}).

demo() ->
  P = #person{name= #name{first="Robert",last="Virding"}, phone=123},
  First = (P#person.name)#name.first.
```

Here, `demo()` evaluates to `"Robert"`.

7.1.9 A Longer Example

Comments are embedded in the following example:

```
%% File: person.hrl

%%--------------------------------------------------------------
%% Data Type: person
%% where:
%%    name:  A string (default is undefined).
%%    age:   An integer (default is undefined).
%%    phone: A list of integers (default is []).
%%    dict:  A dictionary containing various information
%%           about the person.
%%           A {Key, Value} list (default is the empty list).
%%--------------------------------------------------------------
-record(person, {name, age, phone = [], dict = []}).
```

```
-module(person).
-include("person.hrl").
-compile(export_all). % For test purposes only.

%% This creates an instance of a person.
%%   Note: The phone number is not supplied so the
%%         default value [] will be used.

make_hacker_without_phone(Name, Age) ->
   #person{name = Name, age = Age,
           dict = [{computer_knowledge, excellent},
                   {drinks, coke}]}.

%% This demonstrates matching in arguments

print(#person{name = Name, age = Age,
              phone = Phone, dict = Dict}) ->
  io:format("Name: ~s, Age: ~w, Phone: ~w ~n"
            "Dictionary: ~w.~n", [Name, Age, Phone, Dict]).

%% Demonstrates type testing, selector, updating.

birthday(P) when is_record(P, person) ->
   P#person{age = P#person.age + 1}.

register_two_hackers() ->
   Hacker1 = make_hacker_without_phone("Joe", 29),
   OldHacker = birthday(Hacker1),
   % The central_register_server should have
   % an interface function for this.
   central_register_server ! {register_person, Hacker1},
   central_register_server ! {register_person,
             OldHacker#person{name = "Robert",
                              phone = [0,8,3,2,4,5,3,1]}}.
```

7.2 Funs

7.2.1 map

The following function, `double`, doubles every element in a list:

```
double([H|T]) -> [2*H|double(T)];
double([])    -> [].
```

Hence, the argument entered as input is doubled as follows:

```
> double([1,2,3,4]).
[2,4,6,8]
```

The following function, `add_one`, adds one to every element in a list:

```
add_one([H|T]) -> [H+1|add_one(T)];
add_one([])    -> [].
```

The functions `double` and `add_one` have a similar structure. This can be used by writing a function `map` that expresses this similarity:

```
map(F, [H|T]) -> [F(H)|map(F, T)];
map(F, [])    -> [].
```

The functions `double` and `add_one` can now be expressed in terms of `map` as follows:

```
double(L)  -> map(fun(X) -> 2*X end, L).
add_one(L) -> map(fun(X) -> 1 + X end, L).
```

`map(F, List)` is a function that takes a function `F` and a list `L` as arguments and returns a new list, obtained by applying `F` to each of the elements in `L`.

The process of abstracting out the common features of a number of different programs is called **procedural abstraction**. Procedural abstraction can be used to write several different functions that have a similar structure, but differ in some minor detail. This is done as follows:

- **Step 1.** Write one function that represents the common features of these functions.
- **Step 2.** Parameterize the difference in terms of functions that are passed as arguments to the common function.

7.2.2 foreach

This section illustrates procedural abstraction. Initially, the following two examples are written as conventional functions.

This function prints all elements of a list onto a stream:

```
print_list(Stream, [H|T]) ->
    io:format(Stream, "~p~n", [H]),
    print_list(Stream, T);
print_list(Stream, []) ->
    true.
```

This function broadcasts a message to a list of processes:

```
broadcast(Msg, [Pid|Pids]) ->
    Pid ! Msg,
    broadcast(Msg, Pids);
broadcast(_, []) ->
    true.
```

These two functions have a similar structure. They both iterate over a list and do something to each element in the list. The "something" is passed on as an extra argument to the function that does this.

The function `foreach` expresses this similarity:

```
foreach(F, [H|T]) ->
    F(H),
    foreach(F, T);
foreach(F, []) ->
    ok.
```

Using the function `foreach`, the function `print_list` becomes:

```
foreach(fun(H) -> io:format(S, "~p~n",[H]) end, L)
```

Using the function `foreach`, the function `broadcast` becomes:

```
foreach(fun(Pid) -> Pid ! M end, L)
```

`foreach` is evaluated for its side-effect and not its value. `foreach(Fun ,L)` calls `Fun(X)` for each element `X` in `L` and the processing occurs in the order that the elements were defined in `L`. `map` does not define the order in which its elements are processed.

7.2.3 Syntax of Funs

Funs are written with the following syntax (see Fun Expressions for full description):

```
F = fun (Arg1, Arg2, ... ArgN) ->
        ...
    end
```

This creates an anonymous function of `N` arguments and binds it to the variable `F`.

Another function, `FunctionName`, written in the same module, can be passed as an argument, using the following syntax:

```
F = fun FunctionName/Arity
```

With this form of function reference, the function that is referred to does not need to be exported from the module.

It is also possible to refer to a function defined in a different module, with the following syntax:

```
F = fun Module:FunctionName/Arity
```

In this case, the function must be exported from the module in question.

The following program illustrates the different ways of creating funs:

```
-module(fun_test).
-export([t1/0, t2/0]).
-import(lists, [map/2]).

t1() -> map(fun(X) -> 2 * X end, [1,2,3,4,5]).

t2() -> map(fun double/1, [1,2,3,4,5]).

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

The fun `F` can be evaluated with the following syntax:

```
F(Arg1, Arg2, ..., Argn)
```

To check whether a term is a fun, use the test `is_function/1` in a guard.

Example:

```
f(F, Args) when is_function(F) ->
   apply(F, Args);
f(N, _) when is_integer(N) ->
   N.
```

Funs are a distinct type. The BIFs `erlang:fun_info/1,2` can be used to retrieve information about a fun, and the BIF `erlang:fun_to_list/1` returns a textual representation of a fun. The `check_process_code/2` BIF returns `true` if the process contains funs that depend on the old version of a module.

7.2.4 Variable Bindings Within a Fun

The scope rules for variables that occur in funs are as follows:

- All variables that occur in the head of a fun are assumed to be "fresh" variables.
- Variables that are defined before the fun, and that occur in function calls or guard tests within the fun, have the values they had outside the fun.
- Variables cannot be exported from a fun.

The following examples illustrate these rules:

```
print_list(File, List) ->
    {ok, Stream} = file:open(File, write),
    foreach(fun(X) -> io:format(Stream,"~p~n",[X]) end, List),
    file:close(Stream).
```

Here, the variable `X`, defined in the head of the fun, is a new variable. The variable `Stream`, which is used within the fun, gets its value from the `file:open` line.

As any variable that occurs in the head of a fun is considered a new variable, it is equally valid to write as follows:

```
print_list(File, List) ->
    {ok, Stream} = file:open(File, write),
    foreach(fun(File) ->
                io:format(Stream,"~p~n",[File])
            end, List),
    file:close(Stream).
```

Here, `File` is used as the new variable instead of `X`. This is not so wise because code in the fun body cannot refer to the variable `File`, which is defined outside of the fun. Compiling this example gives the following diagnostic:

```
./FileName.erl:Line: Warning: variable 'File'
      shadowed in 'fun'
```

This indicates that the variable `File`, which is defined inside the fun, collides with the variable `File`, which is defined outside the fun.

The rules for importing variables into a fun has the consequence that certain pattern matching operations must be moved into guard expressions and cannot be written in the head of the fun. For example, you might write the following code if you intend the first clause of `F` to be evaluated when the value of its argument is `Y`:

```
f(...) ->
    Y = ...
    map(fun(X) when X == Y ->
              ;
           (_) ->
              ...
        end, ...)
    ...
```

instead of writing the following code:

```
f(...) ->
    Y = ...
    map(fun(Y) ->
              ;
           (_) ->
              ...
        end, ...)
    ...
```

7.2.5 Funs and Module Lists

The following examples show a dialogue with the Erlang shell. All the higher order functions discussed are exported from the module `lists`.

map

`map` takes a function of one argument and a list of terms:

```
map(F, [H|T]) -> [F(H)|map(F, T)];
map(F, [])    -> [].
```

It returns the list obtained by applying the function to every argument in the list.

When a new fun is defined in the shell, the value of the fun is printed as `Fun#<erl_eval>`:

```
> Double = fun(X) -> 2 * X end.
#Fun<erl_eval.6.72228031>
> lists:map(Double, [1,2,3,4,5]).
[2,4,6,8,10]
```

any

`any` takes a predicate `P` of one argument and a list of terms:

```
any(Pred, [H|T]) ->
    case Pred(H) of
        true  ->  true;
        false ->  any(Pred, T)
    end;
any(Pred, []) ->
    false.
```

A predicate is a function that returns `true` or `false`. `any` is `true` if there is a term `X` in the list such that `P(X)` is `true`.

A predicate `Big(X)` is defined, which is `true` if its argument is greater that 10:

```
> Big =  fun(X) -> if X > 10 -> true; true -> false end end.
#Fun<erl_eval.6.72228031>
> lists:any(Big, [1,2,3,4]).
false
> lists:any(Big, [1,2,3,12,5]).
true
```

all

`all` has the same arguments as `any`:

```
all(Pred, [H|T]) ->
    case Pred(H) of
        true  ->  all(Pred, T);
        false ->  false
    end;
all(Pred, []) ->
    true.
```

It is `true` if the predicate applied to all elements in the list is `true`.

```
> lists:all(Big, [1,2,3,4,12,6]).
false
> lists:all(Big, [12,13,14,15]).
true
```

foreach

`foreach` takes a function of one argument and a list of terms:

```
foreach(F, [H|T]) ->
    F(H),
    foreach(F, T);
foreach(F, []) ->
    ok.
```

The function is applied to each argument in the list. `foreach` returns `ok`. It is only used for its side-effect:

```
> lists:foreach(fun(X) -> io:format("~w~n",[X]) end, [1,2,3,4]).
1
2
3
4
ok
```

foldl

`foldl` takes a function of two arguments, an accumulator and a list:

```
foldl(F, Accu, [Hd|Tail]) ->
    foldl(F, F(Hd, Accu), Tail);
foldl(F, Accu, []) -> Accu.
```

The function is called with two arguments. The first argument is the successive elements in the list. The second argument is the accumulator. The function must return a new accumulator, which is used the next time the function is called.

If you have a list of lists `L = ["I","like","Erlang"]`, then you can sum the lengths of all the strings in `L` as follows:

```
> L = ["I","like","Erlang"].
["I","like","Erlang"]
10> lists:foldl(fun(X, Sum) -> length(X) + Sum end, 0, L).
11
```

`foldl` works like a `while` loop in an imperative language:

```
L =  ["I","like","Erlang"],
Sum = 0,
while( L != []){
    Sum += length(head(L)),
    L = tail(L)
end
```

mapfoldl

`mapfoldl` simultaneously maps and folds over a list:

```
mapfoldl(F, Accu0, [Hd|Tail]) ->
    {R,Accu1} = F(Hd, Accu0),
    {Rs,Accu2} = mapfoldl(F, Accu1, Tail),
    {[R|Rs], Accu2};
mapfoldl(F, Accu, []) -> {[], Accu}.
```

The following example shows how to change all letters in `L` to upper case and then count them.

First the change to upper case:

```
> Upcase =  fun(X) when $a =< X,  X =< $z -> X + $A - $a;
(X) -> X
end.
#Fun<erl_eval.6.72228031>
> Upcase_word =
fun(X) ->
lists:map(Upcase, X)
end.
#Fun<erl_eval.6.72228031>
> Upcase_word("Erlang").
"ERLANG"
> lists:map(Upcase_word, L).
["I","LIKE","ERLANG"]
```

Now, the fold and the map can be done at the same time:

```
> lists:mapfoldl(fun(Word, Sum) ->
{Upcase_word(Word), Sum + length(Word)}
end, 0, L).
{["I","LIKE","ERLANG"],11}
```

filter

`filter` takes a predicate of one argument and a list and returns all elements in the list that satisfy the predicate:

```
filter(F, [H|T]) ->
    case F(H) of
        true  -> [H|filter(F, T)];
        false -> filter(F, T)
    end;
filter(F, []) -> [].
```

```
> lists:filter(Big, [500,12,2,45,6,7]).
[500,12,45]
```

Combining maps and filters enables writing of very succinct code. For example, to define a set difference function `diff(L1, L2)` to be the difference between the lists `L1` and `L2`, the code can be written as follows:

```
diff(L1, L2) ->
    filter(fun(X) -> not member(X, L2) end, L1).
```

This gives the list of all elements in L1 that are not contained in L2.

The AND intersection of the list `L1` and `L2` is also easily defined:

```
intersection(L1,L2) -> filter(fun(X) -> member(X,L1) end, L2).
```

takewhile

`takewhile(P, L)` takes elements `X` from a list `L` as long as the predicate `P(X)` is true:

```
takewhile(Pred, [H|T]) ->
    case Pred(H) of
        true  -> [H|takewhile(Pred, T)];
        false -> []
    end;
takewhile(Pred, []) ->
    [].
```

```
> lists:takewhile(Big, [200,500,45,5,3,45,6]).
[200,500,45]
```

dropwhile

`dropwhile` is the complement of `takewhile`:

```
dropwhile(Pred, [H|T]) ->
    case Pred(H) of
        true  -> dropwhile(Pred, T);
        false -> [H|T]
    end;
dropwhile(Pred, []) ->
    [].
```

```
> lists:dropwhile(Big, [200,500,45,5,3,45,6]).
[5,3,45,6]
```

splitwith

`splitwith(P, L)` splits the list `L` into the two sublists `{L1, L2}`, where `L = takewhile(P, L)` and `L2 = dropwhile(P, L)`:

```
splitwith(Pred, L) ->
    splitwith(Pred, L, []).

splitwith(Pred, [H|T], L) ->
    case Pred(H) of
        true  -> splitwith(Pred, T, [H|L]);
        false -> {reverse(L), [H|T]}
    end;
splitwith(Pred, [], L) ->
    {reverse(L), []}.
```

```
> lists:splitwith(Big, [200,500,45,5,3,45,6]).
{[200,500,45],[5,3,45,6]}
```

7.2.6 Funs Returning Funs

So far, only functions that take funs as arguments have been described. More powerful functions, that themselves return funs, can also be written. The following examples illustrate these type of functions.

Simple Higher Order Functions

`Adder(X)` is a function that given `X`, returns a new function `G` such that `G(K)` returns `K + X`:

```
> Adder = fun(X) -> fun(Y) -> X + Y end end.
#Fun<erl_eval.6.72228031>
> Add6 = Adder(6).
#Fun<erl_eval.6.72228031>
> Add6(10).
16
```

Infinite Lists

The idea is to write something like:

```
-module(lazy).
-export([ints_from/1]).
ints_from(N) ->
    fun() ->
            [N|ints_from(N+1)]
    end.
```

Then proceed as follows:

```
> XX = lazy:ints_from(1).
#Fun<lazy.0.29874839>
> XX().
[1|#Fun<lazy.0.29874839>]
> hd(XX()).
1
> Y = tl(XX()).
#Fun<lazy.0.29874839>
> hd(Y()).
2
```

And so on. This is an example of "lazy embedding".

Parsing

The following examples show parsers of the following type:

```
Parser(Toks) -> {ok, Tree, Toks1} | fail
```

`Toks` is the list of tokens to be parsed. A successful parse returns `{ok, Tree, Toks1}`.

- `Tree` is a parse tree.
- `Toks1` is a tail of `Tree` that contains symbols encountered after the structure that was correctly parsed.

An unsuccessful parse returns `fail`.

The following example illustrates a simple, functional parser that parses the grammar:

```
(a | b) & (c | d)
```

The following code defines a function `pconst(X)` in the module `funparse`, which returns a fun that parses a list of tokens:

```
pconst(X) ->
    fun (T) ->
       case T of
           [X|T1] -> {ok, {const, X}, T1};
           _      -> fail
       end
    end.
```

This function can be used as follows:

```
> P1 = funparse:pconst(a).
#Fun<funparse.0.22674075>
> P1([a,b,c]).
{ok,{const,a},[b,c]}
> P1([x,y,z]).
fail
```

Next, the two higher order functions `pand` and `por` are defined. They combine primitive parsers to produce more complex parsers.

First `pand`:

```
pand(P1, P2) ->
    fun (T) ->
        case P1(T) of
            {ok, R1, T1} ->
                case P2(T1) of
                    {ok, R2, T2} ->
                        {ok, {'and', R1, R2}};
                    fail ->
                        fail
                end;
            fail ->
                fail
        end
    end.
```

Given a parser `P1` for grammar `G1`, and a parser `P2` for grammar `G2`, `pand(P1, P2)` returns a parser for the grammar, which consists of sequences of tokens that satisfy `G1`, followed by sequences of tokens that satisfy `G2`.

`por(P1, P2)` returns a parser for the language described by the grammar `G1` or `G2`:

```
por(P1, P2) ->
    fun (T) ->
        case P1(T) of
            {ok, R, T1} ->
                {ok, {'or',1,R}, T1};
            fail ->
                case P2(T) of
                    {ok, R1, T1} ->
                        {ok, {'or',2,R1}, T1};
                    fail ->
                        fail
                end
        end
    end.
```

The original problem was to parse the grammar `(a | b) & (c | d)`. The following code addresses this problem:

```
grammar() ->
    pand(
         por(pconst(a), pconst(b)),
         por(pconst(c), pconst(d))).
```

The following code adds a parser interface to the grammar:

```
parse(List) ->
    (grammar())(List).
```

The parser can be tested as follows:

```
> funparse:parse([a,c]).
{ok,{'and',{'or',1,{const,a}},{'or',1,{const,c}}}}
> funparse:parse([a,d]).
{ok,{'and',{'or',1,{const,a}},{'or',2,{const,d}}}}
> funparse:parse([b,c]).
{ok,{'and',{'or',2,{const,b}},{'or',1,{const,c}}}}
> funparse:parse([b,d]).
{ok,{'and',{'or',2,{const,b}},{'or',2,{const,d}}}}
> funparse:parse([a,b]).
fail
```

7.3 List Comprehensions

7.3.1 Simple Examples

This section starts with a simple example, showing a generator and a filter:

```
> [X || X <- [1,2,a,3,4,b,5,6], X > 3].
[a,4,b,5,6]
```

This is read as follows: The list of X such that X is taken from the list `[1,2,a,...]` and X is greater than 3.

The notation `X <- [1,2,a,...]` is a generator and the expression `X > 3` is a filter.

An additional filter, `is_integer(X)`, can be added to restrict the result to integers:

```
> [X || X <- [1,2,a,3,4,b,5,6], is_integer(X), X > 3].
[4,5,6]
```

Generators can be combined. For example, the Cartesian product of two lists can be written as follows:

```
> [{X, Y} || X <- [1,2,3], Y <- [a,b]].
[{1,a},{1,b},{2,a},{2,b},{3,a},{3,b}]
```

7.3.2 Quick Sort

The well-known quick sort routine can be written as follows:

```
sort([Pivot|T]) ->
    sort([ X || X <- T, X < Pivot]) ++
    [Pivot] ++
    sort([ X || X <- T, X >= Pivot]);
sort([]) -> [].
```

The expression `[X || X <- T, X < Pivot]` is the list of all elements in `T` that are less than `Pivot`.

`[X || X <- T, X >= Pivot]` is the list of all elements in `T` that are greater than or equal to `Pivot`.

A list sorted as follows:

- The first element in the list is isolated and the list is split into two sublists.
- The first sublist contains all elements that are smaller than the first element in the list.
- The second sublist contains all elements that are greater than, or equal to, the first element in the list.
- Then the sublists are sorted and the results are combined.

7.3.3 Permutations

The following example generates all permutations of the elements in a list:

```
perms([]) -> [[]];
perms(L)  -> [[H|T] || H <- L, T <- perms(L--[H])].
```

This takes `H` from `L` in all possible ways. The result is the set of all lists `[H|T]`, where `T` is the set of all possible permutations of `L`, with `H` removed:

```
> perms([b,u,g]).
[[b,u,g],[b,g,u],[u,b,g],[u,g,b],[g,b,u],[g,u,b]]
```

7.3.4 Pythagorean Triplets

Pythagorean triplets are sets of integers `{A,B,C}` such that `A**2 + B**2 = C**2`.

The function `pyth(N)` generates a list of all integers `{A,B,C}` such that `A**2 + B**2 = C**2` and where the sum of the sides is equal to, or less than, `N`:

```
pyth(N) ->
    [ {A,B,C} ||
        A <- lists:seq(1,N),
        B <- lists:seq(1,N),
        C <- lists:seq(1,N),
        A+B+C =< N,
        A*A+B*B == C*C
    ].
```

```
> pyth(3).
[].
> pyth(11).
[].
> pyth(12).
[{3,4,5},{4,3,5}]
> pyth(50).
[{3,4,5},
 {4,3,5},
 {5,12,13},
 {6,8,10},
 {8,6,10},
 {8,15,17},
 {9,12,15},
 {12,5,13},
 {12,9,15},
 {12,16,20},
 {15,8,17},
 {16,12,20}]
```

The following code reduces the search space and is more efficient:

```
pyth1(N) ->
   [{A,B,C} ||
       A <- lists:seq(1,N-2),
       B <- lists:seq(A+1,N-1),
       C <- lists:seq(B+1,N),
       A+B+C =< N,
       A*A+B*B == C*C ].
```

7.3.5 Simplifications With List Comprehensions

As an example, list comprehensions can be used to simplify some of the functions in `lists.erl`:

```
append(L)   ->  [X || L1 <- L, X <- L1].
map(Fun, L) -> [Fun(X) || X <- L].
filter(Pred, L) -> [X || X <- L, Pred(X)].
```

7.3.6 Variable Bindings in List Comprehensions

The scope rules for variables that occur in list comprehensions are as follows:

- All variables that occur in a generator pattern are assumed to be "fresh" variables.
- Any variables that are defined before the list comprehension, and that are used in filters, have the values they had before the list comprehension.
- Variables cannot be exported from a list comprehension.

As an example of these rules, suppose you want to write the function `select`, which selects certain elements from a list of tuples. Suppose you write `select(X, L) -> [Y || {X, Y} <- L].` with the intention of extracting all tuples from `L`, where the first item is `X`.

Compiling this gives the following diagnostic:

```
./FileName.erl:Line: Warning: variable 'X' shadowed in generate
```

This diagnostic warns that the variable `X` in the pattern is not the same as the variable `X` that occurs in the function head.

Evaluating `select` gives the following result:

```
> select(b,[{a,1},{b,2},{c,3},{b,7}]).
[1,2,3,7]
```

This is not the wanted result. To achieve the desired effect, `select` must be written as follows:

```
select(X, L) ->  [Y || {X1, Y} <- L, X == X1].
```

The generator now contains unbound variables and the test has been moved into the filter.

This now works as expected:

```
> select(b,[{a,1},{b,2},{c,3},{b,7}]).
[2,7]
```

Also note that a variable in a generator pattern will shadow a variable with the same name bound in a previous generator pattern. For example:

```
> [{X,Y} || X <- [1,2,3], X=Y <- [a,b,c]].
[{a,a},{b,b},{c,c},{a,a},{b,b},{c,c},{a,a},{b,b},{c,c}]
```

A consequence of the rules for importing variables into a list comprehensions is that certain pattern matching operations must be moved into the filters and cannot be written directly in the generators.

To illustrate this, do **not** write as follows:

```
f(...) ->
    Y = ...
    [ Expression || PatternInvolving Y  <- Expr, ...]
    ...
```

Instead, write as follows:

```
f(...) ->
    Y = ...
    [ Expression || PatternInvolving Y1  <- Expr, Y == Y1, ...]
    ...
```

7.4 Bit Syntax

7.4.1 Introduction

The complete specification for the bit syntax appears in the Reference Manual.

In Erlang, a Bin is used for constructing binaries and matching binary patterns. A Bin is written with the following syntax:

```
      <<E1, E2, ... En>>
```

A Bin is a low-level sequence of bits or bytes. The purpose of a Bin is to enable construction of binaries:

```
Bin = <<E1, E2, ... En>>
```

All elements must be bound. Or match a binary:

```
<<E1, E2, ... En>> = Bin
```

Here, `Bin` is bound and the elements are bound or unbound, as in any match.

A Bin does not need to consist of a whole number of bytes.

A **bitstring** is a sequence of zero or more bits, where the number of bits does not need to be divisible by 8. If the number of bits is divisible by 8, the bitstring is also a binary.

Each element specifies a certain **segment** of the bitstring. A segment is a set of contiguous bits of the binary (not necessarily on a byte boundary). The first element specifies the initial segment, the second element specifies the following segment, and so on.

The following examples illustrate how binaries are constructed, or matched, and how elements and tails are specified.

Examples

Example 1: A binary can be constructed from a set of constants or a string literal:

```
Bin11 = <<1, 17, 42>>,
Bin12 = <<"abc">>
```

This gives two binaries of size 3, with the following evaluations:

- `binary_to_list(Bin11)` evaluates to `[1, 17, 42]`.
- `binary_to_list(Bin12)` evaluates to `[97, 98, 99]`.

Example 2: Similarly, a binary can be constructed from a set of bound variables:

```
A = 1, B = 17, C = 42,
Bin2 = <<A, B, C:16>>
```

This gives a binary of size 4. Here, a **size expression** is used for the variable `C` to specify a 16-bits segment of `Bin2`.

`binary_to_list(Bin2)` evaluates to `[1, 17, 00, 42]`.

Example 3: A Bin can also be used for matching. `D`, `E`, and `F` are unbound variables, and `Bin2` is bound, as in Example 2:

```
<<D:16, E, F/binary>> = Bin2
```

This gives `D = 273`, `E = 00`, and F binds to a binary of size 1: `binary_to_list(F) = [42]`.

Example 4: The following is a more elaborate example of matching. Here, `Dgram` is bound to the consecutive bytes of an IP datagram of IP protocol version 4. The ambition is to extract the header and the data of the datagram:

```
-define(IP_VERSION, 4).
-define(IP_MIN_HDR_LEN, 5).

DgramSize = byte_size(Dgram),
case Dgram of
    <<?IP_VERSION:4, HLen:4, SrvcType:8, TotLen:16,
      ID:16, Flgs:3, FragOff:13,
      TTL:8, Proto:8, HdrChkSum:16,
      SrcIP:32,
      DestIP:32, RestDgram/binary>> when HLen>=5, 4*HLen=<DgramSize ->
        OptsLen = 4*(HLen - ?IP_MIN_HDR_LEN),
        <<Opts:OptsLen/binary,Data/binary>> = RestDgram,
    ...
end.
```

Here, the segment corresponding to the `Opts` variable has a **type modifier**, specifying that `Opts` is to bind to a binary. All other variables have the default type equal to unsigned integer.

An IP datagram header is of variable length. This length is measured in the number of 32-bit words and is given in the segment corresponding to `HLen`. The minimum value of `HLen` is 5. It is the segment corresponding to `Opts` that is variable, so if `HLen` is equal to 5, `Opts` becomes an empty binary.

The tail variables `RestDgram` and `Data` bind to binaries, as all tail variables do. Both can bind to empty binaries.

The match of `Dgram` fails if one of the following occurs:

- The first 4-bits segment of `Dgram` is not equal to 4.
- `HLen` is less than 5.
- The size of `Dgram` is less than `4*HLen`.

7.4.2 Lexical Note

Notice that "`B=<<1>>`" will be interpreted as "`B =< <1>>`", which is a syntax error. The correct way to write the expression is: `B = <<1>>`.

7.4.3 Segments

Each segment has the following general syntax:

`Value:Size/TypeSpecifierList`

The `Size` or the `TypeSpecifier`, or both, can be omitted. Thus, the following variants are allowed:

- `Value`
- `Value:Size`
- `Value/TypeSpecifierList`

Default values are used when specifications are missing. The default values are described in Defaults.

The `Value` part is any expression, when used in binary construction. Used in binary matching, the `Value` part must be a literal or a variable. For more information about the `Value` part, see Constructing Binaries and Bitstrings and Matching Binaries.

The `Size` part of the segment multiplied by the unit in `TypeSpecifierList` (described later) gives the number of bits for the segment. In construction, `Size` is any expression that evaluates to an integer. In matching, `Size` must be a constant expression or a variable.

The `TypeSpecifierList` is a list of type specifiers separated by hyphens.

Type
: The most commonly used types are `integer`, `float`, and `binary`. See Bit Syntax Expressions in the Reference Manual for a complete description.

Signedness
: The signedness specification can be either `signed` or `unsigned`. Notice that signedness only matters for matching.

Endianness
: The endianness specification can be either `big`, `little`, or `native`. Native-endian means that the endian is resolved at load time, to be either big-endian or little-endian, depending on what is "native" for the CPU that the Erlang machine is run on.

Unit
: The unit size is given as `unit:IntegerLiteral`. The allowed range is 1-256. It is multiplied by the `Size` specifier to give the effective size of the segment. The unit size specifies the alignment for binary segments without size.

Example:

```
X:4/little-signed-integer-unit:8
```

This element has a total size of 4*8 = 32 bits, and it contains a signed integer in little-endian order.

7.4.4 Defaults

The default type for a segment is integer. The default type does not depend on the value, even if the value is a literal. For example, the default type in `<<3.14>>` is integer, not float.

The default `Size` depends on the type. For integer it is 8. For float it is 64. For binary it is all of the binary. In matching, this default value is only valid for the last element. All other binary elements in matching must have a size specification.

The default unit depends on the type. For `integer`, `float`, and `bitstring` it is 1. For binary it is 8.

The default signedness is `unsigned`.

The default endianness is `big`.

7.4.5 Constructing Binaries and Bitstrings

This section describes the rules for constructing binaries using the bit syntax. Unlike when constructing lists or tuples, the construction of a binary can fail with a `badarg` exception.

There can be zero or more segments in a binary to be constructed. The expression `<<>>` constructs a zero length binary.

Each segment in a binary can consist of zero or more bits. There are no alignment rules for individual segments of type `integer` and `float`. For binaries and bitstrings without size, the unit specifies the alignment. Since the default alignment for the `binary` type is 8, the size of a binary segment must be a multiple of 8 bits, that is, only whole bytes.

Example:

```
<<Bin/binary,Bitstring/bitstring>>
```

The variable `Bin` must contain a whole number of bytes, because the `binary` type defaults to `unit:8`. A `badarg` exception is generated if `Bin` consist of, for example, 17 bits.

The `Bitstring` variable can consist of any number of bits, for example, 0, 1, 8, 11, 17, 42, and so on. This is because the default `unit` for bitstrings is 1.

For clarity, it is recommended not to change the unit size for binaries. Instead, use `binary` when you need byte alignment and `bitstring` when you need bit alignment.

The following example successfully constructs a bitstring of 7 bits, provided that all of X and Y are integers:

```
<<X:1,Y:6>>
```

As mentioned earlier, segments have the following general syntax:

`Value:Size/TypeSpecifierList`

When constructing binaries, `Value` and `Size` can be any Erlang expression. However, for syntactical reasons, both `Value` and `Size` must be enclosed in parenthesis if the expression consists of anything more than a single literal or a variable. The following gives a compiler syntax error:

```
<<X+1:8>>
```

This expression must be rewritten into the following, to be accepted by the compiler:

```
<<(X+1):8>>
```

Including Literal Strings

A literal string can be written instead of an element:

```
<<"hello">>
```

This is syntactic sugar for the following:

```
<<$h,$e,$l,$l,$o>>
```

7.4.6 Matching Binaries

This section describes the rules for matching binaries, using the bit syntax.

There can be zero or more segments in a binary pattern. A binary pattern can occur wherever patterns are allowed, including inside other patterns. Binary patterns cannot be nested. The pattern `<<>>` matches a zero length binary.

Each segment in a binary can consist of zero or more bits. A segment of type `binary` must have a size evenly divisible by 8 (or divisible by the unit size, if the unit size has been changed). A segment of type `bitstring` has no restrictions on the size. A segment of type `float` must have size 64 or 32.

As mentioned earlier, segments have the following general syntax:

`Value:Size/TypeSpecifierList`

When matching `Value`, value must be either a variable or an integer, or a floating point literal. Expressions are not allowed.

`Size` must be a guard expression, which can use literals and previously bound variables. The following is not allowed:

```
foo(N, <<X:N,T/binary>>) ->
   {X,T}.
```

The two occurrences of `N` are not related. The compiler will complain that the `N` in the size field is unbound.

The correct way to write this example is as follows:

```
foo(N, Bin) ->
   <<X:N,T/binary>> = Bin,
   {X,T}.
```

Note:

Before OTP 23, `Size` was restricted to be an integer or a variable bound to an integer.

Binding and Using a Size Variable

There is one exception to the rule that a variable that is used as size must be previously bound. It is possible to match and bind a variable, and use it as a size within the same binary pattern. For example:

```
bar(<<Sz:8,Payload:Sz/binary-unit:8,Rest/binary>>) ->
   {Payload,Rest}.
```

Here `Sz` is bound to the value in the first byte of the binary. `Sz` is then used at the number of bytes to match out as a binary.

Starting in OTP 23, the size can be a guard expression:

```
bar(<<Sz:8,Payload:((Sz-1)*8)/binary,Rest/binary>>) ->
   {Payload,Rest}.
```

Here `Sz` is the combined size of the header and the payload, so we will need to subtract one byte to get the size of the payload.

Getting the Rest of the Binary or Bitstring

To match out the rest of a binary, specify a binary field without size:

```
foo(<<A:8,Rest/binary>>) ->
```

The size of the tail must be evenly divisible by 8.

To match out the rest of a bitstring, specify a field without size:

```
foo(<<A:8,Rest/bitstring>>) ->
```

There are no restrictions on the number of bits in the tail.

7.4.7 Appending to a Binary

Appending to a binary in an efficient way can be done as follows:

```
triples_to_bin(T) ->
    triples_to_bin(T, <<>>).

triples_to_bin([{X,Y,Z} | T], Acc) ->
    triples_to_bin(T, <<Acc/binary,X:32,Y:32,Z:32>>);
triples_to_bin([], Acc) ->
    Acc.
```

8 Efficiency Guide

8.1 Introduction

8.1.1 Purpose

"Premature optimization is the root of all evil" (D.E. Knuth)

Efficient code can be well-structured and clean, based on a sound overall architecture and sound algorithms. Efficient code can be highly implementation-code that bypasses documented interfaces and takes advantage of obscure quirks in the current implementation.

Ideally, your code only contains the first type of efficient code. If that turns out to be too slow, profile the application to find out where the performance bottlenecks are and optimize only the bottlenecks. Let other code stay as clean as possible.

This Efficiency Guide cannot really teach you how to write efficient code. It can give you a few pointers about what to avoid and what to use, and some understanding of how certain language features are implemented. This guide does not include general tips about optimization that works in any language, such as moving common calculations out of loops.

8.1.2 Prerequisites

It is assumed that you are familiar with the Erlang programming language and the OTP concepts.

8.2 The Seven Myths of Erlang Performance

Some truths seem to live on well beyond their best-before date, perhaps because "information" spreads faster from person-to-person than a single release note that says, for example, that body-recursive calls have become faster.

This section tries to kill the old truths (or semi-truths) that have become myths.

8.2.1 Myth: Tail-Recursive Functions are Much Faster Than Recursive Functions

According to the myth, using a tail-recursive function that builds a list in reverse followed by a call to `lists:reverse/1` is faster than a body-recursive function that builds the list in correct order; the reason being that body-recursive functions use more memory than tail-recursive functions.

That was true to some extent before R12B. It was even more true before R7B. Today, not so much. A body-recursive function generally uses the same amount of memory as a tail-recursive function. It is generally not possible to predict whether the tail-recursive or the body-recursive version will be faster. Therefore, use the version that makes your code cleaner (hint: it is usually the body-recursive version).

For a more thorough discussion about tail and body recursion, see **Erlang's Tail Recursion is Not a Silver Bullet**.

> **Note:**
>
> A tail-recursive function that does not need to reverse the list at the end is faster than a body-recursive function, as are tail-recursive functions that do not construct any terms at all (for example, a function that sums all integers in a list).

8.2.2 Myth: Operator "++" is Always Bad

The ++ operator has, somewhat undeservedly, got a bad reputation. It probably has something to do with code like the following, which is the most inefficient way there is to reverse a list:

DO NOT

```
naive_reverse([H|T]) ->
    naive_reverse(T)++[H];
naive_reverse([]) ->
    [].
```

As the ++ operator copies its left operand, the result is copied repeatedly, leading to quadratic complexity.

But using ++ as follows is not bad:

OK

```
naive_but_ok_reverse([H|T], Acc) ->
    naive_but_ok_reverse(T, [H]++Acc);
naive_but_ok_reverse([], Acc) ->
    Acc.
```

Each list element is copied only once. The growing result `Acc` is the right operand for the ++ operator, and it is **not** copied.

Experienced Erlang programmers would write as follows:

DO

```
vanilla_reverse([H|T], Acc) ->
    vanilla_reverse(T, [H|Acc]);
vanilla_reverse([], Acc) ->
    Acc.
```

This is slightly more efficient because here you do not build a list element only to copy it directly. (Or it would be more efficient if the compiler did not automatically rewrite `[H]++Acc` to `[H|Acc]`.)

8.2.3 Myth: Strings are Slow

String handling can be slow if done improperly. In Erlang, you need to think a little more about how the strings are used and choose an appropriate representation. If you use regular expressions, use the re module in STDLIB instead of the obsolete `regexp` module.

8.2.4 Myth: Repairing a Dets File is Very Slow

The repair time is still proportional to the number of records in the file, but Dets repairs used to be much slower in the past. Dets has been massively rewritten and improved.

8.2.5 Myth: BEAM is a Stack-Based Byte-Code Virtual Machine (and Therefore Slow)

BEAM is a register-based virtual machine. It has 1024 virtual registers that are used for holding temporary values and for passing arguments when calling functions. Variables that need to survive a function call are saved to the stack.

BEAM is a threaded-code interpreter. Each instruction is word pointing directly to executable C-code, making instruction dispatching very fast.

8.2.6 Myth: Use "_" to Speed Up Your Program When a Variable is Not Used

That was once true, but from R6B the BEAM compiler can see that a variable is not used.

Similarly, trivial transformations on the source-code level such as converting a `case` statement to clauses at the top-level of the function seldom makes any difference to the generated code.

8.2.7 Myth: A NIF Always Speeds Up Your Program

Rewriting Erlang code to a NIF to make it faster should be seen as a last resort. It is only guaranteed to be dangerous, but not guaranteed to speed up the program.

Doing too much work in each NIF call will degrade responsiveness of the VM. Doing too little work may mean that the gain of the faster processing in the NIF is eaten up by the overhead of calling the NIF and checking the arguments.

Be sure to read about Long-running NIFs before writing a NIF.

8.3 Common Caveats

This section lists a few modules and BIFs to watch out for, not only from a performance point of view.

8.3.1 Timer Module

Creating timers using erlang:send_after/3 and erlang:start_timer/3, is more efficient than using the timers provided by the timer module in STDLIB.

The `timer` module uses a separate process to manage the timers. Before OTP 25, this management overhead was substantial and increasing with the number of timers, especially when they were short-lived, so the timer server process could easily become overloaded and unresponsive. In OTP 25, the timer module was improved by removing most of the management overhead and the resulting performance penalty. Still, the timer server remains a single process, and it may at some point become a bottleneck of an application.

The functions in the `timer` module that do not manage timers (such as `timer:tc/3` or `timer:sleep/1`), do not call the timer-server process and are therefore harmless.

8.3.2 Accidental Copying and Loss of Sharing

When spawning a new process using a fun, one can accidentally copy more data to the process than intended. For example:

DO NOT

```
accidental1(State) ->
    spawn(fun() ->
                  io:format("~p\n", [State#state.info])
          end).
```

The code in the fun will extract one element from the record and print it. The rest of the `state` record is not used. However, when the `spawn/1` function is executed, the entire record is copied to the newly created process.

The same kind of problem can happen with a map:

DO NOT

```
accidental2(State) ->
    spawn(fun() ->
                  io:format("~p\n", [map_get(info, State)])
          end).
```

In the following example (part of a module implementing the gen_server behavior) the created fun is sent to another process:

DO NOT

```
handle_call(give_me_a_fun, _From, State) ->
    Fun = fun() -> State#state.size =:= 42 end,
    {reply, Fun, State}.
```

How bad that unnecessary copy is depends on the contents of the record or the map.

For example, if the `state` record is initialized like this:

```
init1() ->
    #state{data=lists:seq(1, 10000)}.
```

a list with 10000 elements (or about 20000 heap words) will be copied to the newly created process.

An unnecessary copy of 10000 element list can be bad enough, but it can get even worse if the `state` record contains **shared subterms**. Here is a simple example of a term with a shared subterm:

```
{SubTerm, SubTerm}
```

When a term is copied to another process, sharing of subterms will be lost and the copied term can be many times larger than the original term. For example:

```
init2() ->
    SharedSubTerms = lists:foldl(fun(_, A) -> [A|A] end, [0], lists:seq(1, 15)),
    #state{data=Shared}.
```

In the process that calls `init2/0`, the size of the `data` field in the `state` record will be 32 heap words. When the record is copied to the newly created process, sharing will be lost and the size of the copied `data` field will be 131070 heap words. More details about loss off sharing are found in a later section.

To avoid the problem, outside of the fun extract only the fields of the record that are actually used:

DO

```
fixed_accidental1(State) ->
    Info = State#state.info,
    spawn(fun() ->
                  io:format("~p\n", [Info])
          end).
```

Similarly, outside of the fun extract only the map elements that are actually used:

DO

```
fixed_accidental2(State) ->
    Info = map_get(info, State),
    spawn(fun() ->
                  io:format("~p\n", [Info])
          end).
```

8.3.3 list_to_atom/1

Atoms are not garbage-collected. Once an atom is created, it is never removed. The emulator terminates if the limit for the number of atoms (1,048,576 by default) is reached.

Therefore, converting arbitrary input strings to atoms can be dangerous in a system that runs continuously. If only certain well-defined atoms are allowed as input, list_to_existing_atom/1 can be used to guard against a denial-of-

service attack. (All atoms that are allowed must have been created earlier, for example, by simply using all of them in a module and loading that module.)

Using `list_to_atom/1` to construct an atom that is passed to `apply/3` as follows, is quite expensive and not recommended in time-critical code:

```
apply(list_to_atom("some_prefix"++Var), foo, Args)
```

8.3.4 length/1

The time for calculating the length of a list is proportional to the length of the list, as opposed to `tuple_size/1`, `byte_size/1`, and `bit_size/1`, which all execute in constant time.

Normally, there is no need to worry about the speed of `length/1`, because it is efficiently implemented in C. In time-critical code, you might want to avoid it if the input list could potentially be very long.

Some uses of `length/1` can be replaced by matching. For example, the following code:

```
foo(L) when length(L) >= 3 ->
    ...
```

can be rewritten to:

```
foo([_,_,_|_]=L) ->
   ...
```

One slight difference is that `length(L)` fails if `L` is an improper list, while the pattern in the second code fragment accepts an improper list.

8.3.5 setelement/3

setelement/3 copies the tuple it modifies. Therefore, updating a tuple in a loop using `setelement/3` creates a new copy of the tuple every time.

There is one exception to the rule that the tuple is copied. If the compiler clearly can see that destructively updating the tuple would give the same result as if the tuple was copied, the call to `setelement/3` is replaced with a special destructive `setelement` instruction. In the following code sequence, the first `setelement/3` call copies the tuple and modifies the ninth element:

```
multiple_setelement(T0) ->
    T1 = setelement(9, T0, bar),
    T2 = setelement(7, T1, foobar),
    setelement(5, T2, new_value).
```

The two following `setelement/3` calls modify the tuple in place.

For the optimization to be applied, **all** the following conditions must be true:

- The indices must be integer literals, not variables or expressions.
- The indices must be given in descending order.
- There must be no calls to another function in between the calls to `setelement/3`.
- The tuple returned from one `setelement/3` call must only be used in the subsequent call to `setelement/3`.

If the code cannot be structured as in the `multiple_setelement/1` example, the best way to modify multiple elements in a large tuple is to convert the tuple to a list, modify the list, and convert it back to a tuple.

8.3.6 size/1

`size/1` returns the size for both tuples and binaries.

Using the BIFs `tuple_size/1` and `byte_size/1` gives the compiler and the runtime system more opportunities for optimization. Another advantage is that the BIFs give Dialyzer more type information.

8.3.7 split_binary/2

It is usually more efficient to split a binary using matching instead of calling the `split_binary/2` function. Furthermore, mixing bit syntax matching and `split_binary/2` can prevent some optimizations of bit syntax matching.

DO

```
        <<Bin1:Num/binary,Bin2/binary>> = Bin,
```

DO NOT

```
        {Bin1,Bin2} = split_binary(Bin, Num)
```

8.4 Constructing and Matching Binaries

Binaries can be efficiently built in the following way:

DO

```
my_list_to_binary(List) ->
    my_list_to_binary(List, <<>>).

my_list_to_binary([H|T], Acc) ->
    my_list_to_binary(T, <<Acc/binary,H>>);
my_list_to_binary([], Acc) ->
    Acc.
```

Binaries can be efficiently matched like this:

DO

```
my_binary_to_list(<<H,T/binary>>) ->
    [H|my_binary_to_list(T)];
my_binary_to_list(<<>>) -> [].
```

8.4.1 How Binaries are Implemented

Internally, binaries and bitstrings are implemented in the same way. In this section, they are called **binaries** because that is what they are called in the emulator source code.

Four types of binary objects are available internally:

- Two are containers for binary data and are called:
  - **Refc binaries** (short for **reference-counted binaries**)
  - **Heap binaries**
- Two are merely references to a part of a binary and are called:
  - **sub binaries**
  - **match contexts**

Refc Binaries

Refc binaries consist of two parts:

- An object stored on the process heap, called a **ProcBin**
- The binary object itself, stored outside all process heaps

The binary object can be referenced by any number of ProcBins from any number of processes. The object contains a reference counter to keep track of the number of references, so that it can be removed when the last reference disappears.

All ProcBin objects in a process are part of a linked list, so that the garbage collector can keep track of them and decrement the reference counters in the binary when a ProcBin disappears.

Heap Binaries

Heap binaries are small binaries, up to 64 bytes, and are stored directly on the process heap. They are copied when the process is garbage-collected and when they are sent as a message. They do not require any special handling by the garbage collector.

Sub Binaries

The reference objects **sub binaries** and **match contexts** can reference part of a refc binary or heap binary.

A **sub binary** is created by `split_binary/2` and when a binary is matched out in a binary pattern. A sub binary is a reference into a part of another binary (refc or heap binary, but never into another sub binary). Therefore, matching out a binary is relatively cheap because the actual binary data is never copied.

Match Context

A **match context** is similar to a sub binary, but is optimized for binary matching. For example, it contains a direct pointer to the binary data. For each field that is matched out of a binary, the position in the match context is incremented.

The compiler tries to avoid generating code that creates a sub binary, only to shortly afterwards create a new match context and discard the sub binary. Instead of creating a sub binary, the match context is kept.

The compiler can only do this optimization if it knows that the match context will not be shared. If it would be shared, the functional properties (also called referential transparency) of Erlang would break.

8.4.2 Constructing Binaries

Appending to a binary or bitstring is specially optimized by the **runtime system**:

```
<<Binary/binary, ...>>
<<Binary/bitstring, ...>>
```

As the runtime system handles the optimization (instead of the compiler), there are very few circumstances in which the optimization does not work.

To explain how it works, let us examine the following code line by line:

```
Bin0 = <<0>>,                   %% 1
Bin1 = <<Bin0/binary,1,2,3>>,   %% 2
Bin2 = <<Bin1/binary,4,5,6>>,   %% 3
Bin3 = <<Bin2/binary,7,8,9>>,   %% 4
Bin4 = <<Bin1/binary,17>>,      %% 5 !!!
{Bin4,Bin3}                     %% 6
```

- Line 1 (marked with the `%% 1` comment), assigns a heap binary to the `Bin0` variable.
- Line 2 is an append operation. As `Bin0` has not been involved in an append operation, a new refc binary is created and the contents of `Bin0` is copied into it. The **ProcBin** part of the refc binary has its size set to the size

of the data stored in the binary, while the binary object has extra space allocated. The size of the binary object is either twice the size of `Bin1` or 256, whichever is larger. In this case it is 256.

- Line 3 is more interesting. `Bin1` **has** been used in an append operation, and it has 252 bytes of unused storage at the end, so the 3 new bytes are stored there.
- Line 4. The same applies here. There are 249 bytes left, so there is no problem storing another 3 bytes.
- Line 5. Here, something **interesting** happens. Notice that the result is not appended to the previous result in `Bin3`, but to `Bin1`. It is expected that `Bin4` will be assigned the value `<<0,1,2,3,17>>`. It is also expected that `Bin3` will retain its value (`<<0,1,2,3,4,5,6,7,8,9>>`). Clearly, the runtime system cannot write byte `17` into the binary, because that would change the value of `Bin3` to `<<0,1,2,3,4,17,6,7,8,9>>`.

The runtime system sees that `Bin1` is the result from a previous append operation (not from the latest append operation), so it **copies** the contents of `Bin1` to a new binary, reserve extra storage, and so on. (Here is not explained how the runtime system can know that it is not allowed to write into `Bin1`; it is left as an exercise to the curious reader to figure out how it is done by reading the emulator sources, primarily `erl_bits.c`.)

Circumstances That Force Copying

The optimization of the binary append operation requires that there is a **single** ProcBin and a **single reference** to the ProcBin for the binary. The reason is that the binary object can be moved (reallocated) during an append operation, and when that happens, the pointer in the ProcBin must be updated. If there would be more than one ProcBin pointing to the binary object, it would not be possible to find and update all of them.

Therefore, certain operations on a binary mark it so that any future append operation will be forced to copy the binary. In most cases, the binary object will be shrunk at the same time to reclaim the extra space allocated for growing.

When appending to a binary as follows, only the binary returned from the latest append operation will support further cheap append operations:

```
Bin = <<Bin0,...>>
```

In the code fragment in the beginning of this section, appending to `Bin` will be cheap, while appending to `Bin0` will force the creation of a new binary and copying of the contents of `Bin0`.

If a binary is sent as a message to a process or port, the binary will be shrunk and any further append operation will copy the binary data into a new binary. For example, in the following code fragment `Bin1` will be copied in the third line:

```
Bin1 = <<Bin0,...>>,
PortOrPid ! Bin1,
Bin = <<Bin1,...>>  %% Bin1 will be COPIED
```

The same happens if you insert a binary into an Ets table, send it to a port using `erlang:port_command/2`, or pass it to enif_inspect_binary in a NIF.

Matching a binary will also cause it to shrink and the next append operation will copy the binary data:

```
Bin1 = <<Bin0,...>>,
<<X,Y,Z,T/binary>> = Bin1,
Bin = <<Bin1,...>>  %% Bin1 will be COPIED
```

The reason is that a match context contains a direct pointer to the binary data.

If a process simply keeps binaries (either in "loop data" or in the process dictionary), the garbage collector can eventually shrink the binaries. If only one such binary is kept, it will not be shrunk. If the process later appends to a binary that has been shrunk, the binary object will be reallocated to make place for the data to be appended.

8.4.3 Matching Binaries

Let us revisit the example in the beginning of the previous section:

DO

```
my_binary_to_list(<<H,T/binary>>) ->
    [H|my_binary_to_list(T)];
my_binary_to_list(<<>>) -> [].
```

The first time `my_binary_to_list/1` is called, a match context is created. The match context points to the first byte of the binary. 1 byte is matched out and the match context is updated to point to the second byte in the binary.

At this point it would make sense to create a sub binary, but in this particular example the compiler sees that there will soon be a call to a function (in this case, to `my_binary_to_list/1` itself) that immediately will create a new match context and discard the sub binary.

Therefore `my_binary_to_list/1` calls itself with the match context instead of with a sub binary. The instruction that initializes the matching operation basically does nothing when it sees that it was passed a match context instead of a binary.

When the end of the binary is reached and the second clause matches, the match context will simply be discarded (removed in the next garbage collection, as there is no longer any reference to it).

To summarize, `my_binary_to_list/1` only needs to create **one** match context and no sub binaries.

Notice that the match context in `my_binary_to_list/1` was discarded when the entire binary had been traversed. What happens if the iteration stops before it has reached the end of the binary? Will the optimization still work?

```
after_zero(<<0,T/binary>>) ->
    T;
after_zero(<<_,T/binary>>) ->
    after_zero(T);
after_zero(<<>>) ->
    <<>>.
```

Yes, it will. The compiler will remove the building of the sub binary in the second clause:

```
...
after_zero(<<_,T/binary>>) ->
    after_zero(T);
...
```

But it will generate code that builds a sub binary in the first clause:

```
after_zero(<<0,T/binary>>) ->
    T;
...
```

Therefore, `after_zero/1` builds one match context and one sub binary (assuming it is passed a binary that contains a zero byte).

Code like the following will also be optimized:

```
all_but_zeroes_to_list(Buffer, Acc, 0) ->
    {lists:reverse(Acc),Buffer};
all_but_zeroes_to_list(<<0,T/binary>>, Acc, Remaining) ->
    all_but_zeroes_to_list(T, Acc, Remaining-1);
all_but_zeroes_to_list(<<Byte,T/binary>>, Acc, Remaining) ->
    all_but_zeroes_to_list(T, [Byte|Acc], Remaining-1).
```

The compiler removes building of sub binaries in the second and third clauses, and it adds an instruction to the first clause that converts `Buffer` from a match context to a sub binary (or do nothing if `Buffer` is a binary already).

But in more complicated code, how can one know whether the optimization is applied or not?

Option bin_opt_info

Use the `bin_opt_info` option to have the compiler print a lot of information about binary optimizations. It can be given either to the compiler or `erlc`:

```
erlc +bin_opt_info Mod.erl
```

or passed through an environment variable:

```
export ERL_COMPILER_OPTIONS=bin_opt_info
```

Notice that the `bin_opt_info` is not meant to be a permanent option added to your `Makefile`s, because all messages that it generates cannot be eliminated. Therefore, passing the option through the environment is in most cases the most practical approach.

The warnings look as follows:

```
./efficiency_guide.erl:60: Warning: NOT OPTIMIZED: binary is returned from the function
./efficiency_guide.erl:62: Warning: OPTIMIZED: match context reused
```

To make it clearer exactly what code the warnings refer to, the warnings in the following examples are inserted as comments after the clause they refer to, for example:

```
after_zero(<<0,T/binary>>) ->
        %% BINARY CREATED: binary is returned from the function
    T;
after_zero(<<_,T/binary>>) ->
        %% OPTIMIZED: match context reused
    after_zero(T);
after_zero(<<>>) ->
    <<>>.
```

The warning for the first clause says that the creation of a sub binary cannot be delayed, because it will be returned. The warning for the second clause says that a sub binary will not be created (yet).

Unused Variables

The compiler figures out if a variable is unused. The same code is generated for each of the following functions:

```
count1(<<_,T/binary>>, Count) -> count1(T, Count+1);
count1(<<>>, Count) -> Count.

count2(<<H,T/binary>>, Count) -> count2(T, Count+1);
count2(<<>>, Count) -> Count.

count3(<<_H,T/binary>>, Count) -> count3(T, Count+1);
count3(<<>>, Count) -> Count.
```

In each iteration, the first 8 bits in the binary will be skipped, not matched out.

8.4.4 Historical Note

Binary handling was significantly improved in R12B. Because code that was efficient in R11B might not be efficient in R12B, and vice versa, earlier revisions of this Efficiency Guide contained some information about binary handling in R11B.

8.5 Maps

This guide to using maps efficiently starts with a brief section on the choice between records or maps, followed by three sections giving concrete (but brief) advice on using maps as an alternative to records, as dictionaries, and as sets.

The remaining sections dig deeper, looking at how maps are implemented, the map syntax, and finally the functions in the maps module.

Terminology used in this chapter:

- A map with at most 32 elements will informally be called a **small map**.
- A map with more than 32 elements will informally be called a **large map**.

8.5.1 Maps or Records?

If the advice in this chapter is followed, the performance of records compared to using small maps instead of records is expected to be similar. Therefore, the choice between records and maps should be based on the desired properties of the data structure and not performance.

The advantages of records compared to maps are:

- If the name of a record field is misspelled, there will be a compilation error. If a map key is misspelled, the compiler will give no warning and program will fail in some way when it is run.
- Records will use slightly less memory than maps, and performance is expected to be **slightly** better than maps in most circumstances.

The disadvantage of records compared to maps is that if a new field is added to a record, all code that uses that record must be recompiled. Because of that, it is recommended to only use records within a unit of code that can easily be recompiled all at once, for example within a single application or single module.

8.5.2 Using Maps as an Alternative to Records

- Use the map syntax instead of the functions in the maps module.
- Avoid having more than 32 elements in the map. As soon as there are more than 32 elements in the map, it will require more memory and keys can no longer be shared with other instances of the map.
- When creating a new map, always create it with all keys that will ever be used. To maximize sharing of keys (thus minimizing memory use), create a single function that constructs the map using the map syntax and always use it.
- Always update the map using the `:=` operator (that is, requiring that an element with that key already exists). The `:=` operator is slightly more efficient, and it helps catching mispellings of keys.
- Whenever possible, match multiple map elements at once.
- Whenever possible, update multiple map elements at once.
- Avoid default values and the maps:get/3 function. If there are default values, sharing of keys between different instances of the map will be less effective, and it is not possible to match multiple elements having default values in one go. The `maps:get/3` function is implemented in Erlang, making it less efficient than `maps:get/2` or the map matching syntax.
- To avoid having to deal with a map that may lack some keys, maps:merge/2 can efficiently add multiple default values. For example:

```
DefaultMap = #{shoe_size => 42, editor => emacs},
MapWithDefaultsApplied = maps:merge(DefaultMap, OtherMap)
```

8.5.3 Using Maps as Dictionaries

Using a map as a dictionary implies the following usage pattern:

- Keys are usually variables not known at compile-time.
- There can be any number of elements in the map.
- Usually, no more than one element is looked up or updated at once.

Given that usage pattern, the difference in performance between using the map syntax and the maps module is usually small. Therefore, which one to use is mostly a matter of taste.

Maps are usually the most efficient dictionary data structure, with a few exceptions:

- If it is necessary to frequently convert a dictionary to a sorted list, or from a sorted list to a dictionary, using gb_trees can be a better choice.
- If all keys are non-negative integers, the array module can be a better choice.

8.5.4 Using Maps as Sets

Starting in OTP 24, the sets module has an option to represent sets as maps. Examples:

```
1> sets:new([{version,2}]).
#{}
2> sets:from_list([x,y,z], [{version,2}]).
#{x => [],y => [],z => []}
```

`sets` backed by maps is generally the most efficient set representation, with a few possible exceptions:

- ordsets:intersection/2 can be more efficient than sets:intersection/2. If the intersection operation is frequently used and operations that operate on a single element in a set (such as `is_element/2`) are avoided, ordsets can be a better choice than sets.
- If the intersection operation is frequently used and operations that operate on a single element in a set (such as `is_element/2`) must also be efficient, gb_sets can potentially be a better choice than sets.
- If the elements of the set are integers in a fairly compact range, the set can be represented as an integer where each bit represents an element in the set. The union operation is performed by `bor` and the intersection operation by `band`.

8.5.5 How Maps are Implemented

Internally, maps have two distinct representations depending on the number of elements in the map. The representation changes when a map grows beyond 32 elements, or when it shrinks to 32 elements or less.

- A map with at most 32 elements has a compact representation, making it suitable as an alternative to records.
- A map with more than 32 elements is represented as a tree that can be efficiently searched and updated regardless of how many elements there are.

How Small Maps are Implemented

A small map looks like this inside the runtime system:

FLATMAP	**N**	**Keys**	**Value1**	**...**	**ValueN**

Table 5.1: The representation of a small map

FLATMAP
: The tag for a small map (called **flat map** in the source code for the runtime system).

N
: The number of elements in the map.

Keys
: A tuple with keys of the map: `{Key1,...,KeyN}`. The keys are sorted.

Value1
: The value corresponding to the first key in the key tuple.

ValueN
: The value corresponding to the last key in the key tuple.

As an example, let us look at how the map `#{a => foo, z => bar}` is represented:

FLATMAP	2	{a,z}	foo	bar

Table 5.2: `#{a => foo, z => bar}`

Let us update the map: `M#{q => baz}`. The map now looks like this:

FLATMAP	3	{a,q,z}	foo	baz	bar

Table 5.3: `#{a => foo, q => baz, z => bar}`

Finally, change the value of one element: `M#{z := bird}`. The map now looks like this:

FLATMAP	3	{a,q,z}	foo	baz	bird

Table 5.4: `#{a => foo, q => baz, z => bird}`

When the value for an existing key is updated, the key tuple is not updated, allowing the key tuple to be shared with other instances of the map that have the same keys. In fact, the key tuple can be shared between all maps with the same keys with some care. To arrange that, define a function that returns a map. For example:

```
new() ->
    #{a => default, b => default, c => default}.
```

Defined like this, the key tuple `{a,b,c}` will be a global literal. To ensure that the key tuple is shared when creating an instance of the map, always call `new()` and modify the returned map:

```
    (SOME_MODULE:new())#{a := 42}.
```

Using the map syntax with small maps is particularly efficient. As long as the keys are known at compile-time, the map is updated in one go, making the time to update a map essentially constant regardless of the number of keys updated. The same goes for matching. (When the keys are variables, one or more of the keys could be identical, so the operations need to be performed sequentially from left to right.)

The memory size for a small map is the size of all keys and values plus 5 words. See Advanced for more information about memory sizes.

How Large Maps are Implemented

A map with more than 32 elements is implemented as a **Hash array mapped trie (HAMT)**. A large map can be efficiently searched and updated regardless of the number of elements in the map.

There is less performance to be gained by matching or updating multiple elements using the map syntax on a large map compared to a small map. The execution time is roughly proportional to the number of elements matched or updated.

The storage overhead for a large map is higher than for a small map. For a large map, the extra number of words besides the keys and values is roughly proportional to the number of elements. For a map with 33 elements the overhead is at least 53 heap words according to the formula in Advanced (compared to 5 extra words for a small map regardless of the number of elements).

When a large map is updated, the updated map and the original map will share common parts of the HAMT, but sharing will never be as effective as the best possible sharing of the key tuple for small maps.

Therefore, if maps are used instead of records and it is expected that many instances of the map will be created, it is more efficient from a memory standpoint to avoid using large maps (for example, by grouping related map elements into sub maps to reduce the number of elements).

8.5.6 Using the Map Syntax

Using the map syntax is usually slightly more efficient than using the corresponding function in the maps module.

The gain in efficiency for the map syntax is more noticeable for the following operations that can only be achieved using the map syntax:

- Matching multiple literal keys
- Updating multiple literal keys
- Adding multiple literal keys to a map

For example:

DO

```
Map = Map1#{x := X, y := Y, z := Z}
```

DO NOT

```
Map2 = maps:update(x, X, Map1),
Map3 = maps:update(y, Y, Map2),
Map = maps:update(z, Z, Map3)
```

If the map is a small map, the first example runs roughly three times as fast.

Note that for variable keys, the elements are updated sequentially from left to right. For example, given the following update with variable keys:

```
Map = Map1#{Key1 := X, Key2 := Y, Key3 := Z}
```

the compiler rewrites it like this to ensure that the updates are applied from left to right:

```
Map2 = Map1#{Key1 := X},
Map3 = Map2#{Key2 := Y},
Map = Map3#{Key3 := Z}
```

If a key is known to exist in a map, using the `:=` operator is slightly more efficient than using the `=>` operator for a small map.

8.5.7 Using the Functions in the maps Module

Here follows some notes about most of the functions in the `maps` module. For each function, the implementation language (C or Erlang) is stated. The reason we mention the language is that it gives an hint about how efficient the function is:

- If a function is implemented in C, it is pretty much impossible to implement the same functionality more efficiently in Erlang.
- However, it might be possible to beat the `maps` modules functions implemented in Erlang, because they are generally implemented in a way that attempts to make the performance reasonable for all possible inputs.

  For example, maps:map/2 iterates over all elements of the map, calling the mapping fun, collects the updated map elements in a list, and finally converts the list back to a map using maps:from_list/1. If it is known that at most one percent of the values in the map will change, it can be more efficient to update only the changed values.

Note:

The implementation details given in this section can change in the future.

maps:filter/2

maps:filter/2 is implemented in Erlang. It creates a new map using maps:from_list/1. If it is known that only a minority of the values will be removed, it can be more efficient to avoid `maps:filter/2` and write a function that will use maps:remove/3 to remove the unwanted values.

maps:filtermap/2

maps:filtermap/2 is implemented in Erlang. It creates a new map using maps:from_list/1. See the notes for `maps:map/2` and `maps:filter/2` for hints on how to implement a more efficient version.

maps:find/2

maps:find/2 is implemented in C.

Using the map matching syntax instead of `maps:find/2` will be slightly more efficient since building an `{ok,Value}` tuple will be avoided.

maps:get/2

As an optimization, the compiler will rewrite a call to maps:get/2 to a call to the guard BIF map_get/2. A call to a guard BIF is more efficient than calls to other BIFs, making the performance similar to using the map matching syntax.

If the map is small and the keys are constants known at compile-time, using the map matching syntax will be more efficient than multiple calls to `maps:get/2`.

maps:get/3

maps:get/3 is implemented in Erlang essentially like this:

```
get(Key, Map, Default) ->
    case Map of
        #{Key := Value} -> Value;
        #{} -> Default
    end.
```

Therefore, a call `maps:get/3` is more expensive than a call to `maps:get/2`.

If a small map is used as alternative to using a record, instead of calling `maps:get/3` multiple times to handle default values, consider putting the default values in a map and merging that map with the other map:

```
    DefaultMap = #{Key1 => Value2, Key2 => Value2, ..., KeyN => ValueN},
    MapWithDefaultsApplied = maps:merge(DefaultMap, OtherMap)
```

Whether that is faster than calling `maps:get/3` multiple times depends on the size of the map and the number of default values.

maps:intersect/2, maps:intersect_with/3

maps:intersect/2 and maps:intersect_with/3 are implemented in Erlang. They both create new maps using maps:from_list/1.

Note:

A map is usually the most efficient way to implement a set, but an exception is the intersection operation, where ordsets:intersection/2 used on ordsets can be more efficient than `maps:intersect/2` on sets implemented as maps.

maps:from_list/1

maps:from_list/1 is implemented in C.

maps:from_keys/2

maps:from_keys/2 is implemented in C.

maps:is_key/2

As an optimization, the compiler rewrites calls to maps:is_key/2 to calls to the guard BIF is_map_key/2. A call to a guard BIF is more efficient than calls to other BIFs, making the performance similar to using the map matching syntax.

maps:iterator/1

maps:iterator/1 is efficiently implemented in C and Erlang.

maps:keys/1

maps:keys/1 is implemented in C. If the resulting list needs to be ordered, use lists:sort/1 to sort the result.

maps:map/2

maps:map/2 is implemented in Erlang. It creates a new map using maps:from_list/1. If it is known that only a minority of the values will be updated, it can be more efficient to avoid `maps:map/2` and write a function that will call maps:update/3 to update only the values that have changed.

maps:merge/2

maps:merge/2 is implemented in C.

maps:merge_with/3

maps:merge_with/3 is implemented in Erlang. It updates and returns the larger of the two maps.

maps:new/0

The compiler rewrites a call to maps:new/0 to using the syntax `#{}` for constructing an empty map.

maps:next/1

maps:next/1 is efficiently implemented in C and Erlang.

maps:put/3

maps:put/3 is implemented in C.

If the key is known to already exist in the map, maps:update/3 is slightly more efficient than `maps:put/3`.

If the keys are constants known at compile-time, using the map update syntax with the `=>` operator is more efficient than multiple calls to `maps:put/3`, especially for small maps.

maps:remove/2

maps:remove/2 is implemented in C.

maps:size/1

As an optimization, the compiler rewrites calls to maps:size/1 to calls to the guard BIF map_size/1. Calls to guard BIFs are more efficient than calls to other BIFs.

maps:take/2

maps:take/2 is implemented in C.

maps:to_list/1

maps:to_list/1 is efficiently implemented in C and Erlang. If the resulting list needs to be ordered, use lists:sort/1 to sort the result.

> **Note:**
>
> Maps are usually more performant than gb_trees, but if it is necessary to frequently convert to and from sorted lists, `gb_trees` can be a better choice.

maps:update/3

maps:update/3 is implemented in C.

If the keys are constants known at compile-time, using the map update syntax with the `:=` operator is more efficient than multiple calls to `maps:update/3`, especially for small maps.

maps:values/1

maps:values/1 is implemented in C.

maps:with/2

maps:with/2 is implemented in Erlang. It creates a new map using maps:from_list/1.

maps:without/2

maps:without/2 is implemented in Erlang. It returns a modified copy of the input map.

8.6 List Handling

8.6.1 Creating a List

Lists can only be built starting from the end and attaching list elements at the beginning. If you use the "++" operator as follows, a new list is created that is a copy of the elements in `List1`, followed by `List2`:

```
List1 ++ List2
```

Looking at how `lists:append/1` or ++ would be implemented in plain Erlang, clearly the first list is copied:

```
append([H|T], Tail) ->
    [H|append(T, Tail)];
append([], Tail) ->
    Tail.
```

When recursing and building a list, it is important to ensure that you attach the new elements to the beginning of the list. In this way, you will build **one** list, not hundreds or thousands of copies of the growing result list.

Let us first see how it is not to be done:

DO NOT

```
bad_fib(N) ->
    bad_fib(N, 0, 1, []).

bad_fib(0, _Current, _Next, Fibs) ->
    Fibs;
bad_fib(N, Current, Next, Fibs) ->
    bad_fib(N - 1, Next, Current + Next, Fibs ++ [Current]).
```

Here more than one list is built. In each iteration step a new list is created that is one element longer than the new previous list.

To avoid copying the result in each iteration, build the list in reverse order and reverse the list when you are done:

DO

```
tail_recursive_fib(N) ->
    tail_recursive_fib(N, 0, 1, []).

tail_recursive_fib(0, _Current, _Next, Fibs) ->
    lists:reverse(Fibs);
tail_recursive_fib(N, Current, Next, Fibs) ->
    tail_recursive_fib(N - 1, Next, Current + Next, [Current|Fibs]).
```

8.6.2 List Comprehensions

Lists comprehensions still have a reputation for being slow. They used to be implemented using funs, which used to be slow.

A list comprehension:

```
[Expr(E) || E <- List]
```

is basically translated to a local function:

```
'lc^0'([E|Tail], Expr) ->
    [Expr(E)|'lc^0'(Tail, Expr)];
'lc^0'([], _Expr) -> [].
```

If the result of the list comprehension will **obviously** not be used, a list will not be constructed. For example, in this code:

```
[io:put_chars(E) || E <- List],
ok.
```

or in this code:

```
...
case Var of
    ... ->
        [io:put_chars(E) || E <- List];
    ... ->
end,
some_function(...),
...
```

the value is not assigned to a variable, not passed to another function, and not returned. This means that there is no need to construct a list and the compiler will simplify the code for the list comprehension to:

```
'lc^0'([E|Tail], Expr) ->
    Expr(E),
    'lc^0'(Tail, Expr);
'lc^0'([], _Expr) -> [].
```

The compiler also understands that assigning to '_' means that the value will not used. Therefore, the code in the following example will also be optimized:

```
_ = [io:put_chars(E) || E <- List],
ok.
```

8.6.3 Deep and Flat Lists

lists:flatten/1 builds an entirely new list. It is therefore expensive, and even **more** expensive than the ++ operator (which copies its left argument, but not its right argument).

In the following situations, you can easily avoid calling `lists:flatten/1`:

- When sending data to a port. Ports understand deep lists so there is no reason to flatten the list before sending it to the port.
- When calling BIFs that accept deep lists, such as list_to_binary/1 or iolist_to_binary/1.
- When you know that your list is only one level deep, you can use lists:append/1.

Port Example

DO

```
    ...
    port_command(Port, DeepList)
    ...
```

DO NOT

```
    ...
    port_command(Port, lists:flatten(DeepList))
    ...
```

A common way to send a zero-terminated string to a port is the following:

DO NOT

```
    ...
    TerminatedStr = String ++ [0], % String="foo" => [$f, $o, $o, 0]
    port_command(Port, TerminatedStr)
    ...
```

Instead:

DO

```
    ...
    TerminatedStr = [String, 0], % String="foo" => [[$f, $o, $o], 0]
    port_command(Port, TerminatedStr)
    ...
```

Append Example

DO

```
    > lists:append([[1], [2], [3]]).
    [1,2,3]
    >
```

DO NOT

```
> lists:flatten([[1], [2], [3]]).
[1,2,3]
>
```

8.6.4 Recursive List Functions

In section about myths, the following myth was exposed: Tail-Recursive Functions are Much Faster Than Recursive Functions.

There is usually not much difference between a body-recursive list function and tail-recursive function that reverses the list at the end. Therefore, concentrate on writing beautiful code and forget about the performance of your list functions. In the time-critical parts of your code (and only there), **measure** before rewriting your code.

Note:

This section is about list functions that **construct** lists. A tail-recursive function that does not construct a list runs in constant space, while the corresponding body-recursive function uses stack space proportional to the length of the list.

For example, a function that sums a list of integers, is **not** to be written as follows:

DO NOT

```
recursive_sum([H|T]) -> H+recursive_sum(T);
recursive_sum([])    -> 0.
```

Instead:

DO

```
sum(L) -> sum(L, 0).

sum([H|T], Sum) -> sum(T, Sum + H);
sum([], Sum)    -> Sum.
```

8.7 Functions

8.7.1 Pattern Matching

Pattern matching in function head as well as in `case` and `receive` clauses are optimized by the compiler. With a few exceptions, there is nothing to gain by rearranging clauses.

One exception is pattern matching of binaries. The compiler does not rearrange clauses that match binaries. Placing the clause that matches against the empty binary **last** is usually slightly faster than placing it **first**.

The following is a rather unnatural example to show another exception:

DO NOT

```
atom_map1(one) -> 1;
atom_map1(two) -> 2;
atom_map1(three) -> 3;
atom_map1(Int) when is_integer(Int) -> Int;
atom_map1(four) -> 4;
atom_map1(five) -> 5;
atom_map1(six) -> 6.
```

The problem is the clause with the variable `Int`. As a variable can match anything, including the atoms `four`, `five`, and `six`, which the following clauses also match, the compiler must generate suboptimal code that executes as follows:

- First, the input value is compared to `one`, `two`, and `three` (using a single instruction that does a binary search; thus, quite efficient even if there are many values) to select which one of the first three clauses to execute (if any).
- If none of the first three clauses match, the fourth clause match as a variable always matches.
- If the guard test `is_integer(Int)` succeeds, the fourth clause is executed.
- If the guard test fails, the input value is compared to `four`, `five`, and `six`, and the appropriate clause is selected. (There is a `function_clause` exception if none of the values matched.)

Rewriting to either:

DO

```
atom_map2(one) -> 1;
atom_map2(two) -> 2;
atom_map2(three) -> 3;
atom_map2(four) -> 4;
atom_map2(five) -> 5;
atom_map2(six) -> 6;
atom_map2(Int) when is_integer(Int) -> Int.
```

or:

DO

```
atom_map3(Int) when is_integer(Int) -> Int;
atom_map3(one) -> 1;
atom_map3(two) -> 2;
atom_map3(three) -> 3;
atom_map3(four) -> 4;
atom_map3(five) -> 5;
atom_map3(six) -> 6.
```

gives slightly more efficient matching code.

Another example:

DO NOT

```
map_pairs1(_Map, [], Ys) ->
    Ys;
map_pairs1(_Map, Xs, [] ) ->
    Xs;
map_pairs1(Map, [X|Xs], [Y|Ys]) ->
    [Map(X, Y)|map_pairs1(Map, Xs, Ys)].
```

The first argument is **not** a problem. It is variable, but it is a variable in all clauses. The problem is the variable in the second argument, `Xs`, in the middle clause. Because the variable can match anything, the compiler is not allowed to rearrange the clauses, but must generate code that matches them in the order written.

If the function is rewritten as follows, the compiler is free to rearrange the clauses:

DO

```
map_pairs2(_Map, [], Ys) ->
    Ys;
map_pairs2(_Map, [_|_]=Xs, [] ) ->
    Xs;
map_pairs2(Map, [X|Xs], [Y|Ys]) ->
    [Map(X, Y)|map_pairs2(Map, Xs, Ys)].
```

The compiler will generate code similar to this:

DO NOT (already done by the compiler)

```
explicit_map_pairs(Map, Xs0, Ys0) ->
    case Xs0 of
 [X|Xs] ->
     case Ys0 of
  [Y|Ys] ->
      [Map(X, Y)|explicit_map_pairs(Map, Xs, Ys)];
  [] ->
      Xs0
     end;
 [] ->
     Ys0
    end.
```

This is slightly faster for probably the most common case that the input lists are not empty or very short. (Another advantage is that Dialyzer can deduce a better type for the `Xs` variable.)

8.7.2 Function Calls

This is a rough hierarchy of the performance of the different types of function calls:

- Calls to local or external functions (`foo()`, `m:foo()`) are the fastest calls.
- Calling or applying a fun (`Fun()`, `apply(Fun, [])`) is just a little slower than external calls.
- Applying an exported function (`Mod:Name()`, `apply(Mod, Name, [])`) where the number of arguments is known at compile time is next.
- Applying an exported function (`apply(Mod, Name, Args)`) where the number of arguments is not known at compile time is the least efficient.

Notes and Implementation Details

Calling and applying a fun does not involve any hash-table lookup. A fun contains an (indirect) pointer to the function that implements the fun.

`apply/3` must look up the code for the function to execute in a hash table. It is therefore always slower than a direct call or a fun call.

Caching callback functions into funs may be more efficient in the long run than apply calls for frequently-used callbacks.

8.7.3 Memory Usage in Recursion

When writing recursive functions, it is preferable to make them tail-recursive so that they can execute in constant memory space:

DO

```
list_length(List) ->
    list_length(List, 0).

list_length([], AccLen) ->
    AccLen; % Base case

list_length([_|Tail], AccLen) ->
    list_length(Tail, AccLen + 1). % Tail-recursive
```

DO NOT

```
list_length([]) ->
    0. % Base case
list_length([_ | Tail]) ->
    list_length(Tail) + 1. % Not tail-recursive
```

8.8 Tables and Databases

8.8.1 Ets, Dets, and Mnesia

Every example using Ets has a corresponding example in Mnesia. In general, all Ets examples also apply to Dets tables.

Select/Match Operations

Select/match operations on Ets and Mnesia tables can become very expensive operations. They usually need to scan the complete table. Try to structure the data to minimize the need for select/match operations. However, if you require a select/match operation, it is still more efficient than using `tab2list`. Examples of this and of how to avoid select/match are provided in the following sections. The functions `ets:select/2` and `mnesia:select/3` are to be preferred over `ets:match/2`, `ets:match_object/2`, and `mnesia:match_object/3`.

In some circumstances, the select/match operations do not need to scan the complete table. For example, if part of the key is bound when searching an `ordered_set` table, or if it is a Mnesia table and there is a secondary index on the field that is selected/matched. If the key is fully bound, there is no point in doing a select/match, unless you have a bag table and are only interested in a subset of the elements with the specific key.

When creating a record to be used in a select/match operation, you want most of the fields to have the value "_". The easiest and fastest way to do that is as follows:

```
#person{age = 42, _ = '_'}.
```

Deleting an Element

The `delete` operation is considered successful if the element was not present in the table. Hence all attempts to check that the element is present in the Ets/Mnesia table before deletion are unnecessary. Here follows an example for Ets tables:

DO

```
...
ets:delete(Tab, Key),
...
```

DO NOT

```
...
case ets:lookup(Tab, Key) of
    [] ->
        ok;
    [_|_] ->
        ets:delete(Tab, Key)
end,
...
```

Fetching Data

Do not fetch data that you already have.

Consider that you have a module that handles the abstract data type `Person`. You export the interface function `print_person/1`, which uses the internal functions `print_name/1`, `print_age/1`, and `print_occupation/1`.

Note:

If the function `print_name/1`, and so on, had been interface functions, the situation would have been different, as you do not want the user of the interface to know about the internal data representation.

DO

```
%%% Interface function
print_person(PersonId) ->
    %% Look up the person in the named table person,
    case ets:lookup(person, PersonId) of
        [Person] ->
            print_name(Person),
            print_age(Person),
            print_occupation(Person);
        [] ->
            io:format("No person with ID = ~p~n", [PersonID])
    end.

%%% Internal functions
print_name(Person) ->
    io:format("No person ~p~n", [Person#person.name]).

print_age(Person) ->
    io:format("No person ~p~n", [Person#person.age]).

print_occupation(Person) ->
    io:format("No person ~p~n", [Person#person.occupation]).
```

DO NOT

```
%%% Interface function
print_person(PersonId) ->
    %% Look up the person in the named table person,
    case ets:lookup(person, PersonId) of
        [Person] ->
            print_name(PersonID),
            print_age(PersonID),
            print_occupation(PersonID);
        [] ->
            io:format("No person with ID = ~p~n", [PersonID])
    end.

%%% Internal functions
print_name(PersonID) ->
    [Person] = ets:lookup(person, PersonId),
    io:format("No person ~p~n", [Person#person.name]).

print_age(PersonID) ->
    [Person] = ets:lookup(person, PersonId),
    io:format("No person ~p~n", [Person#person.age]).

print_occupation(PersonID) ->
    [Person] = ets:lookup(person, PersonId),
    io:format("No person ~p~n", [Person#person.occupation]).
```

Non-Persistent Database Storage

For non-persistent database storage, prefer Ets tables over Mnesia `local_content` tables. Even the Mnesia `dirty_write` operations carry a fixed overhead compared to Ets writes. Mnesia must check if the table is replicated or has indices, this involves at least one Ets lookup for each `dirty_write`. Thus, Ets writes is always faster than Mnesia writes.

tab2list

Assuming an Ets table that uses `idno` as key and contains the following:

```
[#person{idno = 1, name = "Adam",  age = 31, occupation = "mailman"},
 #person{idno = 2, name = "Bryan", age = 31, occupation = "cashier"},
 #person{idno = 3, name = "Bryan", age = 35, occupation = "banker"},
 #person{idno = 4, name = "Carl",  age = 25, occupation = "mailman"}]
```

If you **must** return all data stored in the Ets table, you can use `ets:tab2list/1`. However, usually you are only interested in a subset of the information in which case `ets:tab2list/1` is expensive. If you only want to extract one field from each record, for example, the age of every person, then:

DO

```
...
ets:select(Tab,[{ #person{idno='_',
                          name='_',
                          age='$1',
                          occupation = '_'},
                [],
                ['$1']}]),
...
```

DO NOT

```
...
TabList = ets:tab2list(Tab),
lists:map(fun(X) -> X#person.age end, TabList),
...
```

If you are only interested in the age of all persons named "Bryan", then:

DO

```
...
ets:select(Tab,[{ #person{idno='_',
                          name="Bryan",
                          age='$1',
                          occupation = '_'},
                [],
                ['$1']}]),
...
```

DO NOT

```
...
TabList = ets:tab2list(Tab),
lists:foldl(fun(X, Acc) -> case X#person.name of
                               "Bryan" ->
                                   [X#person.age|Acc];
                               _ ->
                                   Acc
                           end
            end, [], TabList),
...
```

REALLY DO NOT

```
...
TabList = ets:tab2list(Tab),
BryanList = lists:filter(fun(X) -> X#person.name == "Bryan" end,
                         TabList),
lists:map(fun(X) -> X#person.age end, BryanList),
...
```

If you need all information stored in the Ets table about persons named "Bryan", then:

DO

```
...
ets:select(Tab, [{#person{idno='_',
                          name="Bryan",
                          age='_',
                          occupation = '_'}, [], ['$_']}]),
...
```

DO NOT

```
...
TabList = ets:tab2list(Tab),
lists:filter(fun(X) -> X#person.name == "Bryan" end, TabList),
...
```

Ordered_set Tables

If the data in the table is to be accessed so that the order of the keys in the table is significant, the table type `ordered_set` can be used instead of the more usual `set` table type. An `ordered_set` is always traversed in Erlang term order regarding the key field so that the return values from functions such as `select`, `match_object`, and `foldl` are ordered by the key values. Traversing an `ordered_set` with the `first` and `next` operations also returns the keys ordered.

Note:

An `ordered_set` only guarantees that objects are processed in **key** order. Results from functions such as `ets:select/2` appear in **key** order even if the key is not included in the result.

8.8.2 Ets-Specific

Using Keys of Ets Table

An Ets table is a single-key table (either a hash table or a tree ordered by the key) and is to be used as one. In other words, use the key to look up things whenever possible. A lookup by a known key in a `set` Ets table is constant and for an `ordered_set` Ets table it is O(logN). A key lookup is always preferable to a call where the whole table has to be scanned. In the previous examples, the field `idno` is the key of the table and all lookups where only the name is known result in a complete scan of the (possibly large) table for a matching result.

A simple solution would be to use the `name` field as the key instead of the `idno` field, but that would cause problems if the names were not unique. A more general solution would be to create a second table with `name` as key and `idno` as data, that is, to index (invert) the table regarding the `name` field. Clearly, the second table would have to be kept consistent with the master table. Mnesia can do this for you, but a home brew index table can be very efficient compared to the overhead involved in using Mnesia.

An index table for the table in the previous examples would have to be a bag (as keys would appear more than once) and can have the following contents:

```
[#index_entry{name="Adam", idno=1},
 #index_entry{name="Bryan", idno=2},
 #index_entry{name="Bryan", idno=3},
 #index_entry{name="Carl", idno=4}]
```

Given this index table, a lookup of the `age` fields for all persons named "Bryan" can be done as follows:

```
...
MatchingIDs = ets:lookup(IndexTable,"Bryan"),
lists:map(fun(#index_entry{idno = ID}) ->
                 [#person{age = Age}] = ets:lookup(PersonTable, ID),
                 Age
          end,
          MatchingIDs),
...
```

Notice that this code never uses `ets:match/2` but instead uses the `ets:lookup/2` call. The `lists:map/2` call is only used to traverse the `idno`s matching the name "Bryan" in the table; thus the number of lookups in the master table is minimized.

Keeping an index table introduces some overhead when inserting records in the table. The number of operations gained from the table must therefore be compared against the number of operations inserting objects in the table. However, notice that the gain is significant when the key can be used to lookup elements.

8.8.3 Mnesia-Specific

Secondary Index

If you frequently do a lookup on a field that is not the key of the table, you lose performance using "mnesia:select/match_object" as this function traverses the whole table. You can create a secondary index instead and use "mnesia:index_read" to get faster access, however this requires more memory.

Example

```
-record(person, {idno, name, age, occupation}).
        ...
{atomic, ok} =
mnesia:create_table(person, [{index,[#person.age]},
                             {attributes,
                                   record_info(fields, person)}]),
{atomic, ok} = mnesia:add_table_index(person, age),
...

PersonsAge42 =
     mnesia:dirty_index_read(person, 42, #person.age),
...
```

Transactions

Using transactions is a way to guarantee that the distributed Mnesia database remains consistent, even when many different processes update it in parallel. However, if you have real-time requirements it is recommended to use `dirty` operations instead of transactions. When using `dirty` operations, you lose the consistency guarantee; this is usually solved by only letting one process update the table. Other processes must send update requests to that process.

Example

```
...
% Using transaction

Fun = fun() ->
          [mnesia:read({Table, Key}),
           mnesia:read({Table2, Key2})]
      end,

{atomic, [Result1, Result2]}  = mnesia:transaction(Fun),
...

% Same thing using dirty operations
...

Result1 = mnesia:dirty_read({Table, Key}),
Result2 = mnesia:dirty_read({Table2, Key2}),
...
```

8.9 Processes

8.9.1 Creating an Erlang Process

An Erlang process is lightweight compared to threads and processes in operating systems.

A newly spawned Erlang process uses 326 words of memory. The size can be found as follows:

```
Erlang/OTP 24 [erts-12.0] [64-bit] [smp:8:8] [ds:8:8:10] [async-threads:1] [jit]

Eshell V5.6  (abort with ^G)
1> Fun = fun() -> receive after infinity -> ok end end.
#Fun<...>
2> {_,Bytes} = process_info(spawn(Fun), memory).
{memory,1232}
3> Bytes div erlang:system_info(wordsize).
309
```

The size includes 233 words for the heap area (which includes the stack). The garbage collector increases the heap as needed.

The main (outer) loop for a process **must** be tail-recursive. Otherwise, the stack grows until the process terminates.

DO NOT

```
loop() ->
  receive
     {sys, Msg} ->
         handle_sys_msg(Msg),
         loop();
     {From, Msg} ->
          Reply = handle_msg(Msg),
          From ! Reply,
          loop()
  end,
  io:format("Message is processed~n", []).
```

The call to `io:format/2` will never be executed, but a return address will still be pushed to the stack each time `loop/0` is called recursively. The correct tail-recursive version of the function looks as follows:

DO

```
   loop() ->
      receive
         {sys, Msg} ->
            handle_sys_msg(Msg),
            loop();
         {From, Msg} ->
            Reply = handle_msg(Msg),
            From ! Reply,
            loop()
    end.
```

Initial Heap Size

The default initial heap size of 233 words is quite conservative to support Erlang systems with hundreds of thousands or even millions of processes. The garbage collector grows and shrinks the heap as needed.

In a system that use comparatively few processes, performance **might** be improved by increasing the minimum heap size using either the `+h` option for erl or on a process-per-process basis using the `min_heap_size` option for spawn_opt/4.

The gain is twofold:

- Although the garbage collector grows the heap, it grows it step-by-step, which is more costly than directly establishing a larger heap when the process is spawned.
- The garbage collector can also shrink the heap if it is much larger than the amount of data stored on it; setting the minimum heap size prevents that.

Warning:

The emulator probably uses more memory, and because garbage collections occur less frequently, huge binaries can be kept much longer.

In systems with many processes, computation tasks that run for a short time can be spawned off into a new process with a higher minimum heap size. When the process is done, it sends the result of the computation to another process and terminates. If the minimum heap size is calculated properly, the process might not have to do any garbage collections at all. **This optimization is not to be attempted without proper measurements.**

8.9.2 Sending Messages

All data in messages sent between Erlang processes is copied, except for refc binaries and literals on the same Erlang node.

When a message is sent to a process on another Erlang node, it is first encoded to the Erlang External Format before being sent through a TCP/IP socket. The receiving Erlang node decodes the message and distributes it to the correct process.

8.9.3 Receiving messages

The cost of receiving messages depends on how complicated the `receive` expression is. A simple expression that matches any message is very cheap because it retrieves the first message in the message queue:

DO

```
receive
    Message -> handle_msg(Message)
end.
```

However, this is not always convenient: we can receive a message that we do not know how to handle at this point, so it is common to only match the messages we expect:

```
receive
    {Tag, Message} -> handle_msg(Message)
end.
```

While this is convenient it means that the entire message queue must be searched until it finds a matching message. This is very expensive for processes with long message queues, so we have added an optimization for the common case of sending a request and waiting for a response shortly after:

DO

```
MRef = monitor(process, Process),
Process ! {self(), MRef, Request},
receive
    {MRef, Reply} ->
        erlang:demonitor(MRef, [flush]),
        handle_reply(Reply);
    {'DOWN', MRef, _, _, Reason} ->
        handle_error(Reason)
end.
```

Since the compiler knows that the reference created by `monitor/2` cannot exist before the call (since it is a globally unique identifier), and that the `receive` only matches messages that contain said reference, it will tell the emulator to search only the messages that arrived after the call to `monitor/2`.

The above is a simple example where one is but guaranteed that the optimization will take, but what about more complicated code?

Option recv_opt_info

Use the `recv_opt_info` option to have the compiler print information about receive optimizations. It can be given either to the compiler or `erlc`:

```
erlc +recv_opt_info Mod.erl
```

or passed through an environment variable:

```
export ERL_COMPILER_OPTIONS=recv_opt_info
```

Notice that `recv_opt_info` is not meant to be a permanent option added to your `Makefile`s, because all messages that it generates cannot be eliminated. Therefore, passing the option through the environment is in most cases the most practical approach.

The warnings look as follows:

```
efficiency_guide.erl:194: Warning: INFO: receive matches any message, this is always fast
efficiency_guide.erl:200: Warning: NOT OPTIMIZED: all clauses do not match a suitable reference
efficiency_guide.erl:206: Warning: OPTIMIZED: reference used to mark a message queue position
efficiency_guide.erl:208: Warning: OPTIMIZED: all clauses match reference created by monitor/2 at efficiency_gu
efficiency_guide.erl:219: Warning: INFO: passing reference created by make_ref/0 at efficiency_guide.erl:218
efficiency_guide.erl:222: Warning: OPTIMIZED: all clauses match reference in function parameter 1
```

To make it clearer exactly what code the warnings refer to, the warnings in the following examples are inserted as comments after the clause they refer to, for example:

```
%% DO
simple_receive() ->
%% efficiency_guide.erl:194: Warning: INFO: not a selective receive, this is always fast
receive
    Message -> handle_msg(Message)
end.

%% DO NOT, unless Tag is known to be a suitable reference: see
%% cross_function_receive/0 further down.
selective_receive(Tag, Message) ->
%% efficiency_guide.erl:200: Warning: NOT OPTIMIZED: all clauses do not match a suitable reference
receive
    {Tag, Message} -> handle_msg(Message)
end.

%% DO
optimized_receive(Process, Request) ->
%% efficiency_guide.erl:206: Warning: OPTIMIZED: reference used to mark a message queue position
    MRef = monitor(process, Process),
    Process ! {self(), MRef, Request},
    %% efficiency_guide.erl:208: Warning: OPTIMIZED: matches reference created by monitor/2 at efficiency_guide
    receive
        {MRef, Reply} ->
        erlang:demonitor(MRef, [flush]),
        handle_reply(Reply);
    {'DOWN', MRef, _, _, Reason} ->
    handle_error(Reason)
    end.

%% DO
cross_function_receive() ->
    %% efficiency_guide.erl:218: Warning: OPTIMIZED: reference used to mark a message queue position
    Ref = make_ref(),
    %% efficiency_guide.erl:219: Warning: INFO: passing reference created by make_ref/0 at efficiency_guide.erl
    cross_function_receive(Ref).

cross_function_receive(Ref) ->
    %% efficiency_guide.erl:222: Warning: OPTIMIZED: all clauses match reference in function parameter 1
    receive
        {Ref, Message} -> handle_msg(Message)
    end.
```

8.9.4 Literal Pool

Constant Erlang terms (hereafter called **literals**) are kept in **literal pools**; each loaded module has its own pool. The following function does not build the tuple every time it is called (only to have it discarded the next time the garbage collector was run), but the tuple is located in the module's literal pool:

DO

```
days_in_month(M) ->
    element(M, {31,28,31,30,31,30,31,31,30,31,30,31}).
```

If a literal, or a term that contains a literal, is inserted into an Ets table, it is **copied**. The reason is that the module containing the literal can be unloaded in the future.

When a literal is sent to another process, it is **not** copied. When a module holding a literal is unloaded, the literal will be copied to the heap of all processes that hold references to that literal.

There also exists a global literal pool that is managed by the persistent_term module.

By default, 1 GB of virtual address space is reserved for all literal pools (in BEAM code and persistent terms). The amount of virtual address space reserved for literals can be changed by using the `+MIscs option` when starting the emulator.

Here is an example how the reserved virtual address space for literals can be raised to 2 GB (2048 MB):

```
    erl +MIscs 2048
```

8.9.5 Loss of Sharing

An Erlang term can have shared subterms. Here is a simple example:

```
{SubTerm, SubTerm}
```

Shared subterms are **not** preserved in the following cases:

- When a term is sent to another process
- When a term is passed as the initial process arguments in the `spawn` call
- When a term is stored in an Ets table

That is an optimization. Most applications do not send messages with shared subterms.

The following example shows how a shared subterm can be created:

```
kilo_byte() ->
    kilo_byte(10, [42]).

kilo_byte(0, Acc) ->
    Acc;
kilo_byte(N, Acc) ->
    kilo_byte(N-1, [Acc|Acc]).
```

`kilo_byte/1` creates a deep list. If `list_to_binary/1` is called, the deep list can be converted to a binary of 1024 bytes:

```
1> byte_size(list_to_binary(efficiency_guide:kilo_byte())).
1024
```

Using the `erts_debug:size/1` BIF, it can be seen that the deep list only requires 22 words of heap space:

```
2> erts_debug:size(efficiency_guide:kilo_byte()).
22
```

Using the `erts_debug:flat_size/1` BIF, the size of the deep list can be calculated if sharing is ignored. It becomes the size of the list when it has been sent to another process or stored in an Ets table:

```
3> erts_debug:flat_size(efficiency_guide:kilo_byte()).
4094
```

It can be verified that sharing will be lost if the data is inserted into an Ets table:

```
4> T = ets:new(tab, []).
#Ref<0.1662103692.2407923716.214181>
5> ets:insert(T, {key,efficiency_guide:kilo_byte()}).
true
6> erts_debug:size(element(2, hd(ets:lookup(T, key)))).
4094
7> erts_debug:flat_size(element(2, hd(ets:lookup(T, key)))).
4094
```

When the data has passed through an Ets table, `erts_debug:size/1` and `erts_debug:flat_size/1` return the same value. Sharing has been lost.

It is possible to build an **experimental** variant of the runtime system that will preserve sharing when copying terms by giving the `--enable-sharing-preserving` option to the `configure` script.

8.9.6 SMP Emulator

The emulator takes advantage of a multi-core or multi-CPU computer by running several Erlang scheduler threads (typically, the same as the number of cores).

To gain performance from a multi-core computer, your application **must have more than one runnable Erlang process** most of the time. Otherwise, the Erlang emulator can still only run one Erlang process at the time.

Benchmarks that appear to be concurrent are often sequential. The estone benchmark, for example, is entirely sequential. So is the most common implementation of the "ring benchmark"; usually one process is active, while the others wait in a `receive` statement.

8.10 Drivers

This section provides a brief overview on how to write efficient drivers.

It is assumed that you have a good understanding of drivers.

8.10.1 Drivers and Concurrency

The runtime system always takes a lock before running any code in a driver.

By default, that lock is at the driver level, that is, if several ports have been opened to the same driver, only code for one port at the same time can be running.

A driver can be configured to have one lock for each port instead.

If a driver is used in a functional way (that is, holds no state, but only does some heavy calculation and returns a result), several ports with registered names can be opened beforehand, and the port to be used can be chosen based on the scheduler ID as follows:

```
-define(PORT_NAMES(),
 {some_driver_01, some_driver_02, some_driver_03, some_driver_04,
  some_driver_05, some_driver_06, some_driver_07, some_driver_08,
  some_driver_09, some_driver_10, some_driver_11, some_driver_12,
  some_driver_13, some_driver_14, some_driver_15, some_driver_16}).

client_port() ->
    element(erlang:system_info(scheduler_id) rem tuple_size(?PORT_NAMES()) + 1,
     ?PORT_NAMES()).
```

As long as there are no more than 16 schedulers, there will never be any lock contention on the port lock for the driver.

8.10.2 Avoiding Copying Binaries When Calling a Driver

There are basically two ways to avoid copying a binary that is sent to a driver:

- If the `Data` argument for port_control/3 is a binary, the driver will be passed a pointer to the contents of the binary and the binary will not be copied. If the `Data` argument is an iolist (list of binaries and lists), all binaries in the iolist will be copied.

  Therefore, if you want to send both a pre-existing binary and some extra data to a driver without copying the binary, you must call `port_control/3` twice; once with the binary and once with the extra data. However, that will only work if there is only one process communicating with the port (because otherwise another process can call the driver in-between the calls).
- Implement an `outputv` callback (instead of an `output` callback) in the driver. If a driver has an `outputv` callback, refc binaries passed in an iolist in the `Data` argument for port_command/2 will be passed as references to the driver.

8.10.3 Returning Small Binaries from a Driver

The runtime system can represent binaries up to 64 bytes as heap binaries. They are always copied when sent in messages, but they require less memory if they are not sent to another process and garbage collection is cheaper.

If you know that the binaries you return are always small, you are advised to use driver API calls that do not require a pre-allocated binary, for example, driver_output() or erl_drv_output_term(), using the `ERL_DRV_BUF2BINARY` format, to allow the runtime to construct a heap binary.

8.10.4 Returning Large Binaries without Copying from a Driver

To avoid copying data when a large binary is sent or returned from the driver to an Erlang process, the driver must first allocate the binary and then send it to an Erlang process in some way.

Use driver_alloc_binary() to allocate a binary.

There are several ways to send a binary created with `driver_alloc_binary()`:

- From the `control` callback, a binary can be returned if set_port_control_flags() has been called with the flag value `PORT_CONTROL_FLAG_BINARY`.
- A single binary can be sent with driver_output_binary().
- Using erl_drv_output_term() or erl_drv_send_term(), a binary can be included in an Erlang term.

8.11 Advanced

8.11.1 Memory

A good start when programming efficiently is to know how much memory different data types and operations require. It is implementation-dependent how much memory the Erlang data types and other items consume, but the following table shows some figures for the `erts-8.0` system in OTP 19.0.

The unit of measurement is memory words. There exists both a 32-bit and a 64-bit implementation. A word is therefore 4 bytes or 8 bytes, respectively. The value for a running system can be determined by calling `erlang:system_info(wordsize)`.

Data Type	**Memory Size**
Small integer	1 word. On 32-bit architectures: -134217729 < i < 134217728 (28 bits). On 64-bit architectures: -576460752303423489 < i < 576460752303423488 (60 bits).
Large integer	3..N words.
Atom	1 word. An atom refers into an atom table, which also consumes memory. The atom text is stored once for each unique atom in this table. The atom table is **not** garbage-collected.
Float	On 32-bit architectures: 4 words. On 64-bit architectures: 3 words.
Binary	3..6 words + data (can be shared).
List	1 word + 1 word per element + the size of each element.
String (is the same as a list of integers)	1 word + 2 words per character.
Tuple	2 words + the size of each element.
Small Map	5 words + the size of all keys and values.
Large Map (> 32 keys)	`N` x `F` words + the size of all keys and values. `N` is the number of keys in the Map. `F` is a sparsity factor that can vary between 1.6 and 1.8 due to the probabilistic nature of the internal HAMT data structure.
Pid	1 word for a process identifier from the current local node. On 32-bit: 6 words for a process identifier from another node. On 64-bit: 5 words for a process identifier from another node.

	A process identifier refers into a process table and a node table, which also consumes memory.
Port	1 word for a port identifier from the current local node. 5 words for a port identifier from another node. A port identifier refers into a port table and a node table, which also consumes memory.
Reference	On 32-bit architectures: 4-7 words for a reference from the current local node, and 7-9 words for a reference from another node. On 64-bit architectures: 4-6 words for a reference from the current local node, and 6-7 words for a reference from another node. A reference also refers into more or less emulator internal data structures which also consumes memory. At a minimum it refers into the node tables.
Fun	9..13 words + the size of environment. A fun refers into a fun table, which also consumes memory.
Ets table	Initially 768 words + the size of each element (6 words + the size of Erlang data). The table grows when necessary.
Erlang process	338 words when spawned, including a heap of 233 words.

Table 11.1: Memory Size of Different Data Types

8.11.2 System Limits

The Erlang language specification puts no limits on the number of processes, length of atoms, and so on. However, for performance and memory saving reasons, there will always be limits in a practical implementation of the Erlang language and execution environment.

Processes	The maximum number of simultaneously alive Erlang processes is by default 262,144. This limit can be configured at startup. For more information, see the `+P` command-line flag in the `erl(1)` manual page in ERTS.
Known nodes	A remote node Y must be known to node X if there exists any pids, ports, references, or funs (Erlang data types) from Y on X, or if X and Y are connected. The maximum number of remote nodes simultaneously/ever known to a node is limited by the maximum number of atoms available for node names. All data concerning remote nodes, except for the node name atom, are garbage-collected.

Connected nodes	The maximum number of simultaneously connected nodes is limited by either the maximum number of simultaneously known remote nodes, the maximum number of (Erlang) ports available, or the maximum number of sockets available.
Characters in an atom	255.
Atoms	By default, the maximum number of atoms is 1,048,576. This limit can be raised or lowered using the `+t` option.
Elements in a tuple	The maximum number of elements in a tuple is 16,777,215 (24-bit unsigned integer).
Size of binary	In the 32-bit implementation of Erlang, 536,870,911 bytes is the largest binary that can be constructed or matched using the bit syntax. In the 64-bit implementation, the maximum size is 2,305,843,009,213,693,951 bytes. If the limit is exceeded, bit syntax construction fails with a `system_limit` exception, while any attempt to match a binary that is too large fails. This limit is enforced starting in R11B-4. In earlier Erlang/OTP releases, operations on too large binaries in general either fail or give incorrect results. In future releases, other operations that create binaries (such as `list_to_binary/1`) will probably also enforce the same limit.
Total amount of data allocated by an Erlang node	The Erlang runtime system can use the complete 32-bit (or 64-bit) address space, but the operating system often limits a single process to use less than that.
Length of a node name	An Erlang node name has the form host@shortname or host@longname. The node name is used as an atom within the system, so the maximum size of 255 holds also for the node name.
Open ports	The maximum number of simultaneously open Erlang ports is often by default 16,384. This limit can be configured at startup. For more information, see the `+Q` command-line flag in the `erl(1)` manual page in ERTS.
Open files and sockets	The maximum number of simultaneously open files and sockets depends on the maximum number of Erlang ports available, as well as on operating system-specific settings and limits.
Number of arguments to a function or fun	255

Unique References on a Runtime System Instance	Each scheduler thread has its own set of references, and all other threads have a shared set of references. Each set of references consist of `2## - 1` unique references. That is, the total amount of unique references that can be produced on a runtime system instance is `(NoSchedulers + 1) × (2## - 1)`. If a scheduler thread create a new reference each nano second, references will at earliest be reused after more than 584 years. That is, for the foreseeable future they are unique enough.
Unique Integers on a Runtime System Instance	There are two types of unique integers both created using the erlang:unique_integer() BIF: **1.** Unique integers created **with** the `monotonic` modifier consist of a set of `2## - 1` unique integers. **2.** Unique integers created **without** the `monotonic` modifier consist of a set of `2## - 1` unique integers per scheduler thread and a set of `2## - 1` unique integers shared by other threads. That is, the total amount of unique integers without the `monotonic` modifier is `(NoSchedulers + 1) × (2## - 1)`. If a unique integer is created each nano second, unique integers will at earliest be reused after more than 584 years. That is, for the foreseeable future they are unique enough.

Table 11.2: System Limits

8.12 Profiling

8.12.1 Do Not Guess About Performance - Profile

Even experienced software developers often guess wrong about where the performance bottlenecks are in their programs. Therefore, profile your program to see where the performance bottlenecks are and concentrate on optimizing them.

Erlang/OTP contains several tools to help finding bottlenecks:

- `fprof` provides the most detailed information about where the program time is spent, but it significantly slows down the program it profiles.
- `eprof` provides time information of each function used in the program. No call graph is produced, but `eprof` has considerably less impact on the program it profiles.

  If the program is too large to be profiled by `fprof` or `eprof`, `cprof` can be used to locate code parts that are to be more thoroughly profiled using `fprof` or `eprof`.
- `cprof` is the most lightweight tool, but it only provides execution counts on a function basis (for all processes, not per process).
- `dbg` is the generic erlang tracing frontend. By using the `timestamp` or `cpu_timestamp` options it can be used to time how long function calls in a live system take.

- `lcnt` is used to find contention points in the Erlang Run-Time System's internal locking mechanisms. It is useful when looking for bottlenecks in interaction between process, port, ets tables and other entities that can be run in parallel.

The tools are further described in Tools.

There are also several open source tools outside of Erlang/OTP that can be used to help profiling. Some of them are:

- **erlgrind** can be used to visualize fprof data in kcachegrind.
- **eflame** is an alternative to fprof that displays the profiling output as a flamegraph.
- **recon** is a collection of Erlang profiling and debugging tools. This tool comes with an accompanying E-book called **Erlang in Anger**.

8.12.2 Memory profiling

```
eheap_alloc: Cannot allocate 1234567890 bytes of memory (of type "heap").
```

The above slogan is one of the more common reasons for Erlang to terminate. For unknown reasons the Erlang Run-Time System failed to allocate memory to use. When this happens a crash dump is generated that contains information about the state of the system as it ran out of memory. Use the `crashdump_viewer` to get a view of the memory being used. Look for processes with large heaps or many messages, large ets tables, etc.

When looking at memory usage in a running system the most basic function to get information from is `erlang:memory()`. It returns the current memory usage of the system. `instrument(3)` can be used to get a more detailed breakdown of where memory is used.

Processes, ports and ets tables can then be inspected using their respective info functions, i.e. `erlang:process_info/2` , `erlang:port_info/2` and `ets:info/1`.

Sometimes the system can enter a state where the reported memory from `erlang:memory(total)` is very different from the memory reported by the OS. This can be because of internal fragmentation within the Erlang Run-Time System. Data about how memory is allocated can be retrieved using `erlang:system_info(allocator)`. The data you get from that function is very raw and not very pleasant to read. **recon_alloc** can be used to extract useful information from system_info statistics counters.

8.12.3 Large Systems

For a large system, it can be interesting to run profiling on a simulated and limited scenario to start with. But bottlenecks have a tendency to appear or cause problems only when many things are going on at the same time, and when many nodes are involved. Therefore, it is also desirable to run profiling in a system test plant on a real target system.

For a large system, you do not want to run the profiling tools on the whole system. Instead you want to concentrate on central processes and modules, which account for a big part of the execution.

There are also some tools that can be used to get a view of the whole system with more or less overhead.

- `observer` is a GUI tool that can connect to remote nodes and display a variety of information about the running system.
- `etop` is a command line tool that can connect to remote nodes and display information similar to what the UNIX tool top shows.
- `msacc` allows the user to get a view of what the Erlang Run-Time system is spending its time doing. Has a very low overhead, which makes it useful to run in heavily loaded systems to get some idea of where to start doing more granular profiling.

8.12.4 What to Look For

When analyzing the result file from the profiling activity, look for functions that are called many times and have a long "own" execution time (time excluding calls to other functions). Functions that are called a lot of times can also

be interesting, as even small things can add up to quite a bit if repeated often. Also ask yourself what you can do to reduce this time. The following are appropriate types of questions to ask yourself:

- Is it possible to reduce the number of times the function is called?
- Can any test be run less often if the order of tests is changed?
- Can any redundant tests be removed?
- Does any calculated expression give the same result each time?
- Are there other ways to do this that are equivalent and more efficient?
- Can another internal data representation be used to make things more efficient?

These questions are not always trivial to answer. Some benchmarks might be needed to back up your theory and to avoid making things slower if your theory is wrong. For details, see Benchmarking.

8.12.5 Tools

fprof

`fprof` measures the execution time for each function, both own time, that is, how much time a function has used for its own execution, and accumulated time, that is, including called functions. The values are displayed per process. You also get to know how many times each function has been called.

`fprof` is based on trace to file to minimize runtime performance impact. Using `fprof` is just a matter of calling a few library functions, see the fprof manual page in Tools.

eprof

`eprof` is based on the Erlang `trace_info` BIFs. `eprof` shows how much time has been used by each process, and in which function calls this time has been spent. Time is shown as a percentage of total time and absolute time. For more information, see the eprof manual page in Tools.

cprof

`cprof` is something in between `fprof` and `cover` regarding features. It counts how many times each function is called when the program is run, on a per module basis. `cprof` has a low performance degradation effect (compared with `fprof`) and does not need to recompile any modules to profile (compared with `cover`). For more information, see the cprof manual page in Tools.

Tool Summary

Tool	Results	Size of Result	Effects on Program Execution Time	Records Number of Calls	Records Execution Time	Records Called by	Records Garbage Collection
`fprof`	Per process to screen/ file	Large	Significant slowdown	Yes	Total and own	Yes	Yes
`eprof`	Per process/ function to screen/file	Medium	Small slowdown	Yes	Only total	No	No

`cprof`	Per module to caller	Small	Small slowdown	Yes	No	No	No

Table 12.1: Tool Summary

dbg

`dbg` is a generic Erlang trace tool. By using the `timestamp` or `cpu_timestamp` options it can be used as a precision instrument to profile how long time a function call takes for a specific process. This can be very useful when trying to understand where time is spent in a heavily loaded system as it is possible to limit the scope of what is profiled to be very small. For more information, see the dbg manual page in Runtime Tools.

lcnt

`lcnt` is used to profile interactions in between entities that run in parallel. For example if you have a process that all other processes in the system needs to interact with (maybe it has some global configuration), then `lcnt` can be used to figure out if the interaction with that process is a problem.

In the Erlang Run-time System entities are only run in parallel when there are multiple schedulers. Therefore `lcnt` will show more contention points (and thus be more useful) on systems using many schedulers on many cores.

For more information, see the lcnt manual page in Tools.

8.12.6 Benchmarking

The main purpose of benchmarking is to find out which implementation of a given algorithm or function is the fastest. Benchmarking is far from an exact science. Today's operating systems generally run background tasks that are difficult to turn off. Caches and multiple CPU cores do not facilitate benchmarking. It would be best to run UNIX computers in single-user mode when benchmarking, but that is inconvenient to say the least for casual testing.

Benchmarks can measure wall-clock time or CPU time.

- timer:tc/3 measures wall-clock time. The advantage with wall-clock time is that I/O, swapping, and other activities in the operating system kernel are included in the measurements. The disadvantage is that the measurements vary a lot. Usually it is best to run the benchmark several times and note the shortest time, which is to be the minimum time that is possible to achieve under the best of circumstances.
- statistics/1 with argument `runtime` measures CPU time spent in the Erlang virtual machine. The advantage with CPU time is that the results are more consistent from run to run. The disadvantage is that the time spent in the operating system kernel (such as swapping and I/O) is not included. Therefore, measuring CPU time is misleading if any I/O (file or socket) is involved.

It is probably a good idea to do both wall-clock measurements and CPU time measurements.

Some final advice:

- The granularity of both measurement types can be high. Therefore, ensure that each individual measurement lasts for at least several seconds.
- To make the test fair, each new test run is to run in its own, newly created Erlang process. Otherwise, if all tests run in the same process, the later tests start out with larger heap sizes and therefore probably do fewer garbage collections. Also consider restarting the Erlang emulator between each test.
- Do not assume that the fastest implementation of a given algorithm on computer architecture X is also the fastest on computer architecture Y.

8.13 Retired Myths

We believe that the truth finally has caught with the following, retired myths.

8.13.1 Myth: Funs are Slow

Funs used to be very slow, slower than `apply/3`. Originally, funs were implemented using nothing more than compiler trickery, ordinary tuples, `apply/3`, and a great deal of ingenuity.

But that is history. Funs was given its own data type in R6B and was further optimized in R7B. Now the cost for a fun call falls roughly between the cost for a call to a local function and `apply/3`.

8.13.2 Myth: List Comprehensions are Slow

List comprehensions used to be implemented using funs, and in the old days funs were indeed slow.

Nowadays, the compiler rewrites list comprehensions into an ordinary recursive function. Using a tail-recursive function with a reverse at the end would be still faster. Or would it? That leads us to the myth that tail-recursive functions are faster than body-recursive functions.

8.13.3 Myth: List subtraction ("--" operator) is slow

List subtraction used to have a run-time complexity proportional to the product of the length of its operands, so it was extremely slow when both lists were long.

As of OTP 22 the run-time complexity is "n log n" and the operation will complete quickly even when both lists are very long. In fact, it is faster and uses less memory than the commonly used workaround to convert both lists to ordered sets before subtracting them with `ordsets:subtract/2`.

9 Interoperability Tutorial

9.1 Introduction

This section informs on interoperability, that is, information exchange, between Erlang and other programming languages. The included examples mainly treat interoperability between Erlang and C.

9.1.1 Purpose

The purpose of this tutorial is to describe different interoperability mechanisms that can be used when integrating a program written in Erlang with a program written in another programming language, from the Erlang programmer's perspective.

9.1.2 Prerequisites

It is assumed that you are a skilled Erlang programmer, familiar with concepts such as Erlang data types, processes, messages, and error handling.

To illustrate the interoperability principles, C programs running in a UNIX environment have been used. It is assumed that you have enough knowledge to apply these principles to the relevant programming languages and platforms.

Note:

For readability, the example code is kept as simple as possible. For example, it does not include error handling, which might be vital in a real-life system.

9.2 Overview

9.2.1 Built-In Mechanisms

Two interoperability mechanisms are built into the Erlang runtime system, **distributed Erlang** and **ports**. A variation of ports is **linked-in drivers**.

Distributed Erlang

An Erlang runtime system is made a distributed Erlang node by giving it a name. A distributed Erlang node can connect to, and monitor, other nodes. It can also spawn processes at other nodes. Message passing and error handling between processes at different nodes are transparent. A number of useful STDLIB modules are available in a distributed Erlang system. For example, `global`, which provides global name registration. The distribution mechanism is implemented using TCP/IP sockets.

When to use: Distributed Erlang is primarily used for Erlang-Erlang communication. It can also be used for communication between Erlang and C, if the C program is implemented as a C node, see C and Java Libraries.

Where to read more: Distributed Erlang and some distributed programming techniques are described in the Erlang book.

For more information, see Distributed Programming.

Relevant manual pages are the following:

- erlang manual page in ERTS (describes the BIFs)

- global manual page in Kernel
- net_adm manual page in Kernel
- pg manual page in Kernel
- rpc manual page in Kernel
- pool manual page in STDLIB
- slave manual page in STDLIB

Ports and Linked-In Drivers

Ports provide the basic mechanism for communication with the external world, from Erlang's point of view. The ports provide a byte-oriented interface to an external program. When a port is created, Erlang can communicate with it by sending and receiving lists of bytes (not Erlang terms). This means that the programmer might have to invent a suitable encoding and decoding scheme.

The implementation of the port mechanism depends on the platform. For UNIX, pipes are used and the external program is assumed to read from standard input and write to standard output. The external program can be written in any programming language as long as it can handle the interprocess communication mechanism with which the port is implemented.

The external program resides in another OS process than the Erlang runtime system. In some cases this is not acceptable. Consider, for example, drivers with very hard time requirements. It is therefore possible to write a program in C according to certain principles, and dynamically link it to the Erlang runtime system. This is called a **linked-in driver**.

When to use: Ports can be used for all kinds of interoperability situations where the Erlang program and the other program runs on the same machine. Programming is fairly straight-forward.

Linked-in drivers involves writing certain call-back functions in C. This requires very good skills as the code is linked to the Erlang runtime system.

> **Warning:**
>
> A faulty linked-in driver causes the entire Erlang runtime system to leak memory, hang, or crash.

Where to read more: Ports are described in section "Miscellaneous Items" of the Erlang book. Linked-in drivers are described in Appendix E.

The BIF `open_port/2` is documented in the erlang manual page in ERTS.

For linked-in drivers, the programmer needs to read the erl_ddll manual page in Kernel.

Examples: Port example in Ports.

9.2.2 C and Java Libraries

Erl_Interface

The program at the other side of a port is often a C program. To help the C programmer, the Erl_Interface library has been developed

The Erlang external term format is a representation of an Erlang term as a sequence of bytes, that is, a binary. Conversion between the two representations is done using the following BIFs:

```
Binary = term_to_binary(Term)
Term = binary_to_term(Binary)
```

A port can be set to use binaries instead of lists of bytes. It is then not necessary to invent any encoding/decoding scheme. Erl_Interface functions are used for unpacking the binary and convert it into a struct similar to an Erlang term. Such a struct can be manipulated in different ways, be converted to the Erlang external format, and sent to Erlang.

When to use: In C code, in conjunction with Erlang binaries.

Where to read more: See the Erlang Interface User's Guide, Command Reference, and Library Reference. In Erlang/OTP R5B, and earlier versions, the information is part of the Kernel application.

Examples: Erl_Interface example in Erl_Interface.

C Nodes

A C program that uses the Erl_Interface functions for setting up a connection to, and communicating with, a distributed Erlang node is called a **C node**, or a **hidden node**. The main advantage with a C node is that the communication from the Erlang programmer's perspective is extremely easy, as the C program behaves as a distributed Erlang node.

When to use: C nodes can typically be used on device processors (as opposed to control processors) where C is a better choice than Erlang due to memory limitations or application characteristics, or both.

Where to read more: See the `ei_connect` part of the Erl_Interface documentation. The programmer also needs to be familiar with TCP/IP sockets, see Sockets in Standard Protocols and Distributed Erlang in Built-In Mechanisms.

Example: C node example in C Nodes.

Jinterface

In Erlang/OTP R6B, a library similar to Erl_Interface for Java was added called **jinterface**. It provides a tool for Java programs to communicate with Erlang nodes.

9.2.3 Standard Protocols

Sometimes communication between an Erlang program and another program using a standard protocol is desirable. Erlang/OTP currently supports TCP/IP and UDP **sockets**: as follows:

- SNMP
- HTTP
- IIOP (CORBA)

Using one of the latter three requires good knowledge about the protocol and is not covered by this tutorial. See the SNMP, Inets, and Orber applications, respectively.

Sockets

Simply put, connection-oriented socket communication (TCP/IP) consists of an initiator socket ("server") started at a certain host with a certain port number. A connector socket ("client"), which is aware of the initiator host name and port number, can connect to it and data can be sent between them.

Connection-less socket communication (UDP) consists of an initiator socket at a certain host with a certain port number and a connector socket sending data to it.

For a detailed description of the socket concept, refer to a suitable book about network programming. A suggestion is **UNIX Network Programming, Volume 1: Networking APIs - Sockets and XTI** by W. Richard Stevens, ISBN: 013490012X.

In Erlang/OTP, access to TCP/IP and UDP sockets is provided by the modules `gen_tcp` and `gen_udp` in Kernel. Both are easy to use and do not require detailed knowledge about the socket concept.

When to use: For programs running on the same or on another machine than the Erlang program.

Where to read more: See the gen_tcp and the gen_udp manual pages in Kernel.

9.2.4 IC and CORBA

IC (Erlang IDL Compiler) is an interface generator that, given an IDL interface specification, automatically generates stub code in Erlang, C, or Java. See the IC User's Guide and IC Reference Manual.

For details, see the **corba repository**.

9.2.5 Old Applications

Two old applications are of interest regarding interoperability. Both have been replaced by IC and are mentioned here for reference only:

- IG - Removed from Erlang/OTP R6B.

  IG (Interface Generator) automatically generated code for port or socket communication between an Erlang program and a C program, given a C header file with certain keywords.
- Jive - Removed from Erlang/OTP R7B.

  Jive provided a simple interface between an Erlang program and a Java program.

9.3 Problem Example

9.3.1 Description

A common interoperability situation is when you want to incorporate a piece of code, solving a complex problem, in your Erlang program. Suppose for example, that you have the following C functions that you would like to call from Erlang:

```
/* complex.c */

int foo(int x) {
  return x+1;
}

int bar(int y) {
  return y*2;
}
```

The functions are deliberately kept as simple as possible, for readability reasons.

From an Erlang perspective, it is preferable to be able to call `foo` and `bar` without having to bother about that they are C functions:

```
% Erlang code
...
Res = complex:foo(X),
...
```

Here, the communication with C is hidden in the implementation of `complex.erl`. In the following sections, it is shown how this module can be implemented using the different interoperability mechanisms.

9.4 Ports

This section outlines an example of how to solve the example problem in the previous section by using a port.

The scenario is illustrated in the following figure:

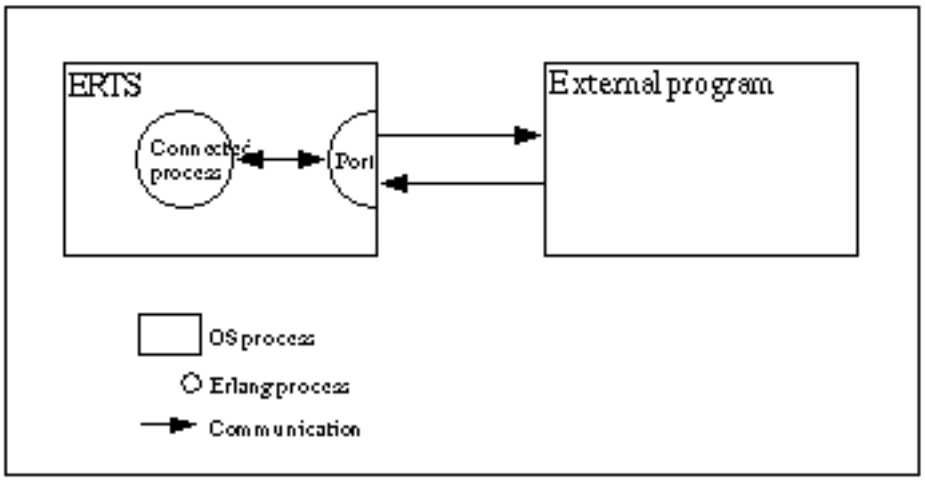


Figure 4.1: Port Communication

9.4.1 Erlang Program

All communication between Erlang and C must be established by creating the port. The Erlang process that creates a port is said to be **the connected process** of the port. All communication to and from the port must go through the connected process. If the connected process terminates, the port also terminates (and the external program, if it is written properly).

The port is created using the BIF `open_port/2` with `{spawn,ExtPrg}` as the first argument. The string `ExtPrg` is the name of the external program, including any command line arguments. The second argument is a list of options, in this case only `{packet,2}`. This option says that a 2 byte length indicator is to be used to simplify the communication between C and Erlang. The Erlang port automatically adds the length indicator, but this must be done explicitly in the external C program.

The process is also set to trap exits, which enables detection of failure of the external program:

```
-module(complex1).
-export([start/1, init/1]).

start(ExtPrg) ->
  spawn(?MODULE, init, [ExtPrg]).

init(ExtPrg) ->
  register(complex, self()),
  process_flag(trap_exit, true),
  Port = open_port({spawn, ExtPrg}, [{packet, 2}]),
  loop(Port).
```

Now `complex1:foo/1` and `complex1:bar/1` can be implemented. Both send a message to the `complex` process and receive the following replies:

```
foo(X) ->
  call_port({foo, X}).
bar(Y) ->
  call_port({bar, Y}).

call_port(Msg) ->
  complex ! {call, self(), Msg},
  receive
    {complex, Result} ->
      Result
  end.
```

The `complex` process does the following:

- Encodes the message into a sequence of bytes.
- Sends it to the port.
- Waits for a reply.
- Decodes the reply.
- Sends it back to the caller:

```
loop(Port) ->
  receive
    {call, Caller, Msg} ->
      Port ! {self(), {command, encode(Msg)}},
      receive
        {Port, {data, Data}} ->
          Caller ! {complex, decode(Data)}
      end,
      loop(Port)
  end.
```

Assuming that both the arguments and the results from the C functions are less than 256, a simple encoding/decoding scheme is employed. In this scheme, `foo` is represented by byte 1, `bar` is represented by 2, and the argument/result is represented by a single byte as well:

```
encode({foo, X}) -> [1, X];
encode({bar, Y}) -> [2, Y].

decode([Int]) -> Int.
```

The resulting Erlang program, including functionality for stopping the port and detecting port failures, is as follows:

```
-module(complex1).
-export([start/1, stop/0, init/1]).
-export([foo/1, bar/1]).

start(ExtPrg) ->
    spawn(?MODULE, init, [ExtPrg]).
stop() ->
    complex ! stop.

foo(X) ->
    call_port({foo, X}).
bar(Y) ->
    call_port({bar, Y}).

call_port(Msg) ->
    complex ! {call, self(), Msg},
    receive
 {complex, Result} ->
     Result
    end.

init(ExtPrg) ->
    register(complex, self()),
    process_flag(trap_exit, true),
    Port = open_port({spawn, ExtPrg}, [{packet, 2}]),
    loop(Port).

loop(Port) ->
    receive
 {call, Caller, Msg} ->
     Port ! {self(), {command, encode(Msg)}},
     receive
  {Port, {data, Data}} ->
      Caller ! {complex, decode(Data)}
     end,
     loop(Port);
 stop ->
     Port ! {self(), close},
     receive
  {Port, closed} ->
      exit(normal)
     end;
 {'EXIT', Port, Reason} ->
     exit(port_terminated)
    end.

encode({foo, X}) -> [1, X];
encode({bar, Y}) -> [2, Y].

decode([Int]) -> Int.
```

9.4.2 C Program

On the C side, it is necessary to write functions for receiving and sending data with 2 byte length indicators from/to Erlang. By default, the C program is to read from standard input (file descriptor 0) and write to standard output (file descriptor 1). Examples of such functions, `read_cmd/1` and `write_cmd/2`, follows:

```
/* erl_comm.c */

#include <stdio.h>
#include <unistd.h>

typedef unsigned char byte;

int read_exact(byte *buf, int len)
{
  int i, got=0;

  do {
      if ((i = read(0, buf+got, len-got)) <= 0){
          return(i);
      }
    got += i;
  } while (got<len);

  return(len);
}

int write_exact(byte *buf, int len)
{
  int i, wrote = 0;

  do {
    if ((i = write(1, buf+wrote, len-wrote)) <= 0)
      return (i);
    wrote += i;
  } while (wrote<len);

  return (len);
}

int read_cmd(byte *buf)
{
  int len;

  if (read_exact(buf, 2) != 2)
    return(-1);
  len = (buf[0] << 8) | buf[1];
  return read_exact(buf, len);
}

int write_cmd(byte *buf, int len)
{
  byte li;

  li = (len >> 8) & 0xff;
  write_exact(&li, 1);

  li = len & 0xff;
  write_exact(&li, 1);

  return write_exact(buf, len);
}
```

Notice that `stdin` and `stdout` are for buffered input/output and must **not** be used for the communication with Erlang.

In the `main` function, the C program is to listen for a message from Erlang and, according to the selected encoding/decoding scheme, use the first byte to determine which function to call and the second byte as argument to the function. The result of calling the function is then to be sent back to Erlang:

```
/* port.c */

typedef unsigned char byte;

int main() {
  int fn, arg, res;
  byte buf[100];

  while (read_cmd(buf) > 0) {
    fn = buf[0];
    arg = buf[1];

    if (fn == 1) {
      res = foo(arg);
    } else if (fn == 2) {
      res = bar(arg);
    }

    buf[0] = res;
    write_cmd(buf, 1);
  }
}
```

Notice that the C program is in a `while`-loop, checking for the return value of `read_cmd/1`. This is because the C program must detect when the port closes and terminates.

9.4.3 Running the Example

Step 1. Compile the C code:

```
unix> gcc -o extprg complex.c erl_comm.c port.c
```

Step 2. Start Erlang and compile the Erlang code:

```
unix> erl
Erlang (BEAM) emulator version 4.9.1.2

Eshell V4.9.1.2 (abort with ^G)
1> c(complex1).
{ok,complex1}
```

Step 3. Run the example:

```
2> complex1:start("./extprg").
<0.34.0>
3> complex1:foo(3).
4
4> complex1:bar(5).
10
5> complex1:stop().
stop
```

9.5 Erl_Interface

This section outlines an example of how to solve the example problem in Problem Example by using a port and Erl_Interface. It is necessary to read the port example in Ports before reading this section.

9.5.1 Erlang Program

The following example shows an Erlang program communicating with a C program over a plain port with home made encoding:

```
-module(complex1).
-export([start/1, stop/0, init/1]).
-export([foo/1, bar/1]).

start(ExtPrg) ->
    spawn(?MODULE, init, [ExtPrg]).
stop() ->
    complex ! stop.

foo(X) ->
    call_port({foo, X}).
bar(Y) ->
    call_port({bar, Y}).

call_port(Msg) ->
    complex ! {call, self(), Msg},
    receive
 {complex, Result} ->
     Result
    end.

init(ExtPrg) ->
    register(complex, self()),
    process_flag(trap_exit, true),
    Port = open_port({spawn, ExtPrg}, [{packet, 2}]),
    loop(Port).

loop(Port) ->
    receive
 {call, Caller, Msg} ->
     Port ! {self(), {command, encode(Msg)}},
     receive
  {Port, {data, Data}} ->
      Caller ! {complex, decode(Data)}
     end,
     loop(Port);
 stop ->
     Port ! {self(), close},
     receive
  {Port, closed} ->
      exit(normal)
     end;
 {'EXIT', Port, Reason} ->
     exit(port_terminated)
    end.

encode({foo, X}) -> [1, X];
encode({bar, Y}) -> [2, Y].

decode([Int]) -> Int.
```

There are two differences when using Erl_Interface on the C side compared to the example in Ports, using only the plain port:

- As Erl_Interface operates on the Erlang external term format, the port must be set to use binaries.
- Instead of inventing an encoding/decoding scheme, the `term_to_binary/1` and `binary_to_term/1` BIFs are to be used.

That is:

```
open_port({spawn, ExtPrg}, [{packet, 2}])
```

is replaced with:

```
open_port({spawn, ExtPrg}, [{packet, 2}, binary])
```

And:

```
Port ! {self(), {command, encode(Msg)}},
receive
  {Port, {data, Data}} ->
    Caller ! {complex, decode(Data)}
end
```

is replaced with:

```
Port ! {self(), {command, term_to_binary(Msg)}},
receive
  {Port, {data, Data}} ->
    Caller ! {complex, binary_to_term(Data)}
end
```

The resulting Erlang program is as follows:

```
-module(complex2).
-export([start/1, stop/0, init/1]).
-export([foo/1, bar/1]).

start(ExtPrg) ->
    spawn(?MODULE, init, [ExtPrg]).
stop() ->
    complex ! stop.

foo(X) ->
    call_port({foo, X}).
bar(Y) ->
    call_port({bar, Y}).

call_port(Msg) ->
    complex ! {call, self(), Msg},
    receive
 {complex, Result} ->
     Result
    end.

init(ExtPrg) ->
    register(complex, self()),
    process_flag(trap_exit, true),
    Port = open_port({spawn, ExtPrg}, [{packet, 2}, binary]),
    loop(Port).

loop(Port) ->
    receive
 {call, Caller, Msg} ->
     Port ! {self(), {command, term_to_binary(Msg)}},
     receive
  {Port, {data, Data}} ->
      Caller ! {complex, binary_to_term(Data)}
     end,
     loop(Port);
 stop ->
     Port ! {self(), close},
     receive
  {Port, closed} ->
      exit(normal)
     end;
 {'EXIT', Port, Reason} ->
     exit(port_terminated)
    end.
```

Notice that calling `complex2:foo/1` and `complex2:bar/1` results in the tuple `{foo,X}` or `{bar,Y}` being sent to the `complex` process, which codes them as binaries and sends them to the port. This means that the C program must be able to handle these two tuples.

9.5.2 C Program

The following example shows a C program communicating with an Erlang program over a plain port with the Erlang external term format encoding:

```
/* ei.c */

#include "ei.h"
#include <unistd.h>
#include <string.h>
#include <stdlib.h>

typedef unsigned char byte;

int read_cmd(byte *buf);
int write_cmd(byte *buf, int len);
int foo(int x);
int bar(int y);

static void fail(int place) {
    fprintf(stderr, "Something went wrong %d\n", place);
    exit(1);
}

int main() {
    byte buf[100];
    int index = 0;
    int version = 0;
    int arity = 0;
    char atom[128];
    long in = 0;
    int res = 0;
    ei_x_buff res_buf;
    ei_init();
    while (read_cmd(buf) > 0) {
        if (ei_decode_version(buf, &index, &version) != 0)
            fail(1);
        if (ei_decode_tuple_header(buf, &index, &arity) != 0)
            fail(2);
        if (arity != 2)
            fail(3);
        if (ei_decode_atom(buf, &index, atom) != 0)
            fail(4);
        if (ei_decode_long(buf, &index, &in) != 0)
            fail(5);
        if (strncmp(atom, "foo", 3) == 0) {
            res = foo((int)in);
        } else if (strncmp(atom, "bar", 3) == 0) {
            res = bar((int)in);
        }
        if (ei_x_new_with_version(&res_buf) != 0)
            fail(6);
        if (ei_x_encode_long(&res_buf, res) != 0)
            fail(7);
        write_cmd(res_buf.buff, res_buf.index);

        if (ei_x_free(&res_buf) != 0)
            fail(8);
        index = 0;
    }
}
```

The following functions, `read_cmd()` and `write_cmd()`, from the `erl_comm.c` example in Ports can still be used for reading from and writing to the port:

```
/* erl_comm.c */

#include <stdio.h>
#include <unistd.h>

typedef unsigned char byte;

int read_exact(byte *buf, int len)
{
  int i, got=0;

  do {
      if ((i = read(0, buf+got, len-got)) <= 0){
          return(i);
      }
    got += i;
  } while (got<len);

  return(len);
}

int write_exact(byte *buf, int len)
{
  int i, wrote = 0;

  do {
    if ((i = write(1, buf+wrote, len-wrote)) <= 0)
      return (i);
    wrote += i;
  } while (wrote<len);

  return (len);
}

int read_cmd(byte *buf)
{
  int len;

  if (read_exact(buf, 2) != 2)
    return(-1);
  len = (buf[0] << 8) | buf[1];
  return read_exact(buf, len);
}

int write_cmd(byte *buf, int len)
{
  byte li;

  li = (len >> 8) & 0xff;
  write_exact(&li, 1);

  li = len & 0xff;
  write_exact(&li, 1);

  return write_exact(buf, len);
}
```

9.5.3 Running the Example

Step 1. Compile the C code. This provides the paths to the include file `ei.h`, and also to the library `ei`:

```
unix> gcc -o extprg -I/usr/local/otp/lib/erl_interface-3.9.2/include \
      -L/usr/local/otp/lib/erl_interface-3.9.2/lib \
      complex.c erl_comm.c ei.c -lei -lpthread
```

In Erlang/OTP R5B and later versions of OTP, the `include` and `lib` directories are situated under `OTPROOT/lib/erl_interface-VSN`, where `OTPROOT` is the root directory of the OTP installation (`/usr/local/otp` in the recent example) and `VSN` is the version of the Erl_interface application (3.2.1 in the recent example).

In R4B and earlier versions of OTP, `include` and `lib` are situated under `OTPROOT/usr`.

Step 2. Start Erlang and compile the Erlang code:

```
unix> erl
Erlang (BEAM) emulator version 4.9.1.2

Eshell V4.9.1.2 (abort with ^G)
1> c(complex2).
{ok,complex2}
```

Step 3. Run the example:

```
2> complex2:start("./extprg").
<0.34.0>
3> complex2:foo(3).
4
4> complex2:bar(5).
10
5> complex2:bar(352).
704
6> complex2:stop().
stop
```

9.6 Port Drivers

This section outlines an example of how to solve the example problem in Problem Example by using a linked-in port driver.

A port driver is a linked-in driver that is accessible as a port from an Erlang program. It is a shared library (SO in UNIX, DLL in Windows), with special entry points. The Erlang runtime system calls these entry points when the driver is started and when data is sent to the port. The port driver can also send data to Erlang.

As a port driver is dynamically linked into the emulator process, this is the fastest way of calling C-code from Erlang. Calling functions in the port driver requires no context switches. But it is also the least safe way, because a crash in the port driver brings the emulator down too.

The scenario is illustrated in the following figure:

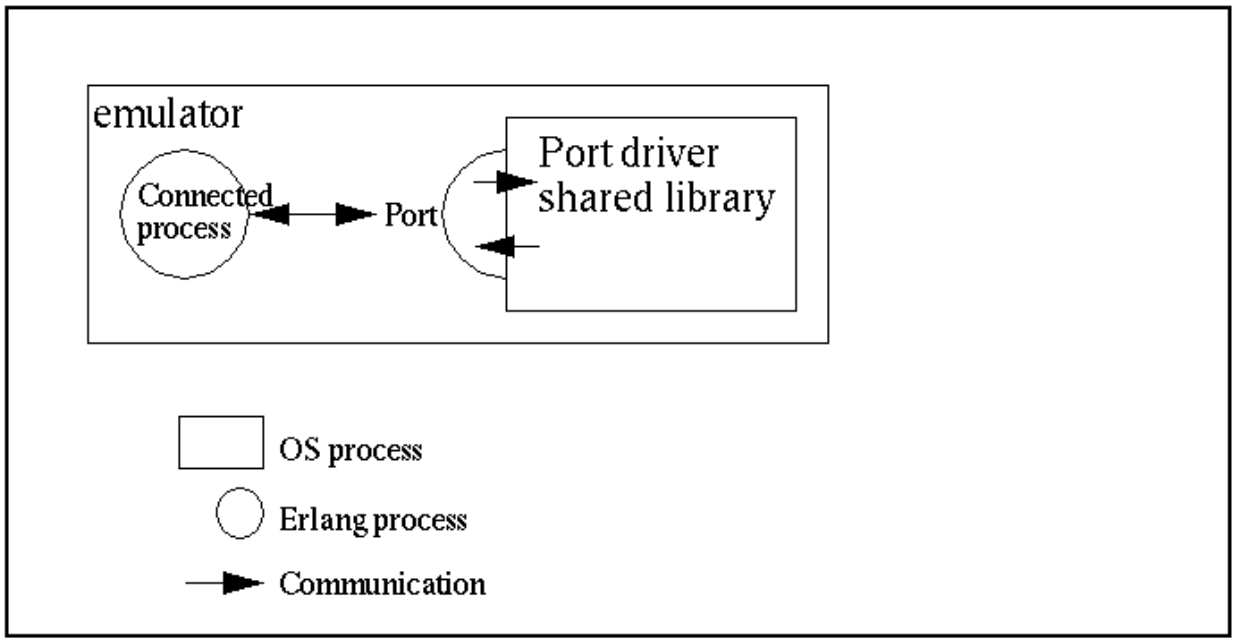


Figure 6.1: Port Driver Communication

9.6.1 Erlang Program

Like a port program, the port communicates with an Erlang process. All communication goes through one Erlang process that is the **connected process** of the port driver. Terminating this process closes the port driver.

Before the port is created, the driver must be loaded. This is done with the function `erl_ddll:load_driver/1`, with the name of the shared library as argument.

The port is then created using the BIF `open_port/2`, with the tuple `{spawn, DriverName}` as the first argument. The string `SharedLib` is the name of the port driver. The second argument is a list of options, none in this case:

```
-module(complex5).
-export([start/1, init/1]).

start(SharedLib) ->
    case erl_ddll:load_driver(".", SharedLib) of
        ok -> ok;
        {error, already_loaded} -> ok;
        _ -> exit({error, could_not_load_driver})
    end,
    spawn(?MODULE, init, [SharedLib]).

init(SharedLib) ->
  register(complex, self()),
  Port = open_port({spawn, SharedLib}, []),
  loop(Port).
```

Now `complex5:foo/1` and `complex5:bar/1` can be implemented. Both send a message to the `complex` process and receive the following reply:

```
foo(X) ->
    call_port({foo, X}).
bar(Y) ->
    call_port({bar, Y}).

call_port(Msg) ->
    complex ! {call, self(), Msg},
    receive
        {complex, Result} ->
            Result
    end.
```

The `complex` process performs the following:

- Encodes the message into a sequence of bytes.
- Sends it to the port.
- Waits for a reply.
- Decodes the reply.
- Sends it back to the caller:

```
loop(Port) ->
    receive
        {call, Caller, Msg} ->
            Port ! {self(), {command, encode(Msg)}},
            receive
                {Port, {data, Data}} ->
                    Caller ! {complex, decode(Data)}
            end,
            loop(Port)
    end.
```

Assuming that both the arguments and the results from the C functions are less than 256, a simple encoding/decoding scheme is employed. In this scheme, `foo` is represented by byte 1, `bar` is represented by 2, and the argument/result is represented by a single byte as well:

```
encode({foo, X}) -> [1, X];
encode({bar, Y}) -> [2, Y].

decode([Int]) -> Int.
```

The resulting Erlang program, including functions for stopping the port and detecting port failures, is as follows:

```
-module(complex5).
-export([start/1, stop/0, init/1]).
-export([foo/1, bar/1]).

start(SharedLib) ->
    case erl_ddll:load_driver(".", SharedLib) of
 ok -> ok;
 {error, already_loaded} -> ok;
 _ -> exit({error, could_not_load_driver})
    end,
    spawn(?MODULE, init, [SharedLib]).

init(SharedLib) ->
    register(complex, self()),
    Port = open_port({spawn, SharedLib}, []),
    loop(Port).

stop() ->
    complex ! stop.

foo(X) ->
    call_port({foo, X}).
bar(Y) ->
    call_port({bar, Y}).

call_port(Msg) ->
    complex ! {call, self(), Msg},
    receive
 {complex, Result} ->
     Result
    end.

loop(Port) ->
    receive
 {call, Caller, Msg} ->
     Port ! {self(), {command, encode(Msg)}},
     receive
  {Port, {data, Data}} ->
      Caller ! {complex, decode(Data)}
     end,
     loop(Port);
 stop ->
     Port ! {self(), close},
     receive
  {Port, closed} ->
      exit(normal)
     end;
 {'EXIT', Port, Reason} ->
     io:format("~p ~n", [Reason]),
     exit(port_terminated)
    end.

encode({foo, X}) -> [1, X];
encode({bar, Y}) -> [2, Y].

decode([Int]) -> Int.
```

9.6.2 C Driver

The C driver is a module that is compiled and linked into a shared library. It uses a driver structure and includes the header file `erl_driver.h`.

The driver structure is filled with the driver name and function pointers. It is returned from the special entry point, declared with the macro `DRIVER_INIT(<driver_name>)`.

The functions for receiving and sending data are combined into a function, pointed out by the driver structure. The data sent into the port is given as arguments, and the replied data is sent with the C-function `driver_output`.

As the driver is a shared module, not a program, no main function is present. All function pointers are not used in this example, and the corresponding fields in the `driver_entry` structure are set to NULL.

All functions in the driver takes a handle (returned from `start`) that is just passed along by the Erlang process. This must in some way refer to the port driver instance.

The `example_drv_start`, is the only function that is called with a handle to the port instance, so this must be saved. It is customary to use an allocated driver-defined structure for this one, and to pass a pointer back as a reference.

It is not a good idea to use a global variable as the port driver can be spawned by multiple Erlang processes. This driver-structure is to be instantiated multiple times:

```
/* port_driver.c */

#include <stdio.h>
#include "erl_driver.h"

typedef struct {
    ErlDrvPort port;
} example_data;

static ErlDrvData example_drv_start(ErlDrvPort port, char *buff)
{
    example_data* d = (example_data*)driver_alloc(sizeof(example_data));
    d->port = port;
    return (ErlDrvData)d;
}

static void example_drv_stop(ErlDrvData handle)
{
    driver_free((char*)handle);
}

static void example_drv_output(ErlDrvData handle, char *buff,
          ErlDrvSizeT bufflen)
{
    example_data* d = (example_data*)handle;
    char fn = buff[0], arg = buff[1], res;
    if (fn == 1) {
      res = foo(arg);
    } else if (fn == 2) {
      res = bar(arg);
    }
    driver_output(d->port, &res, 1);
}

ErlDrvEntry example_driver_entry = {
    NULL,   /* F_PTR init, called when driver is loaded */
    example_drv_start,  /* L_PTR start, called when port is opened */
    example_drv_stop,  /* F_PTR stop, called when port is closed */
    example_drv_output,  /* F_PTR output, called when erlang has sent */
    NULL,   /* F_PTR ready_input, called when input descriptor ready */
    NULL,   /* F_PTR ready_output, called when output descriptor ready */
    "example_drv",  /* char *driver_name, the argument to open_port */
    NULL,   /* F_PTR finish, called when unloaded */
    NULL,                       /* void *handle, Reserved by VM */
    NULL,   /* F_PTR control, port_command callback */
    NULL,   /* F_PTR timeout, reserved */
    NULL,   /* F_PTR outputv, reserved */
    NULL,                        /* F_PTR ready_async, only for async drivers */
    NULL,                        /* F_PTR flush, called when port is about
       to be closed, but there is data in driver
       queue */
    NULL,                        /* F_PTR call, much like control, sync call
       to driver */
    NULL,                        /* unused */
    ERL_DRV_EXTENDED_MARKER,    /* int extended marker, Should always be
       set to indicate driver versioning */
    ERL_DRV_EXTENDED_MAJOR_VERSION, /* int major_version, should always be
          set to this value */
    ERL_DRV_EXTENDED_MINOR_VERSION, /* int minor_version, should always be
          set to this value */
    0,                           /* int driver_flags, see documentation */
    NULL,                        /* void *handle2, reserved for VM use */
    NULL,                        /* F_PTR process_exit, called when a
       monitored process dies */
    NULL                         /* F_PTR stop_select, called to close an
```

```
       event object */
};

DRIVER_INIT(example_drv) /* must match name in driver_entry */
{
    return &example_driver_entry;
}
```

9.6.3 Running the Example

Step 1. Compile the C code:

```
unix> gcc -o example_drv.so -fpic -shared complex.c port_driver.c
windows> cl -LD -MD -Fe example_drv.dll complex.c port_driver.c
```

Step 2. Start Erlang and compile the Erlang code:

```
> erl
Erlang (BEAM) emulator version 5.1

Eshell V5.1 (abort with ^G)
1> c(complex5).
{ok,complex5}
```

Step 3. Run the example:

```
2> complex5:start("example_drv").
<0.34.0>
3> complex5:foo(3).
4
4> complex5:bar(5).
10
5> complex5:stop().
stop
```

9.7 C Nodes

The reader is referred to the erl_interface users guide for information about how to create C nodes.

9.8 NIFs

This section outlines an example of how to solve the example problem in Problem Example by using Native Implemented Functions (NIFs).

NIFs are a simpler and more efficient way of calling C-code than using port drivers. NIFs are most suitable for synchronous functions, such as `foo` and `bar` in the example, that do some relatively short calculations without side effects and return the result.

A NIF is a function that is implemented in C instead of Erlang. NIFs appear as any other functions to the callers. They belong to a module and are called like any other Erlang functions. The NIFs of a module are compiled and linked into a dynamic loadable, shared library (SO in UNIX, DLL in Windows). The NIF library must be loaded in runtime by the Erlang code of the module.

As a NIF library is dynamically linked into the emulator process, this is the fastest way of calling C-code from Erlang (alongside port drivers). Calling NIFs requires no context switches. But it is also the least safe, because a crash in a NIF brings the emulator down too.

9.8.1 Erlang Program

Even if all functions of a module are NIFs, an Erlang module is still needed for two reasons:

- The NIF library must be explicitly loaded by Erlang code in the same module.
- All NIFs of a module must have an Erlang implementation as well.

Normally these are minimal stub implementations that throw an exception. But they can also be used as fallback implementations for functions that do not have native implementations on some architectures.

NIF libraries are loaded by calling `erlang:load_nif/2`, with the name of the shared library as argument. The second argument can be any term that will be passed on to the library and used for initialization:

```
-module(complex6).
-export([foo/1, bar/1]).
-nifs([foo/1, bar/1]).
-on_load(init/0).

init() ->
    ok = erlang:load_nif("./complex6_nif", 0).

foo(_X) ->
    exit(nif_library_not_loaded).
bar(_Y) ->
    exit(nif_library_not_loaded).
```

Here, the directive `on_load` is used to get function `init` to be automatically called when the module is loaded. If `init` returns anything other than `ok`, such when the loading of the NIF library fails in this example, the module is unloaded and calls to functions within it, fail.

Loading the NIF library overrides the stub implementations and cause calls to `foo` and `bar` to be dispatched to the NIF implementations instead.

9.8.2 NIF Library Code

The NIFs of the module are compiled and linked into a shared library. Each NIF is implemented as a normal C function. The macro `ERL_NIF_INIT` together with an array of structures defines the names, arity, and function pointers of all the NIFs in the module. The header file `erl_nif.h` must be included. As the library is a shared module, not a program, no main function is to be present.

The function arguments passed to a NIF appears in an array `argv`, with `argc` as the length of the array, and thus the arity of the function. The Nth argument of the function can be accessed as `argv[N-1]`. NIFs also take an environment argument that serves as an opaque handle that is needed to be passed on to most API functions. The environment contains information about the calling Erlang process:

```
#include <erl_nif.h>

extern int foo(int x);
extern int bar(int y);

static ERL_NIF_TERM foo_nif(ErlNifEnv* env, int argc, const ERL_NIF_TERM argv[])
{
    int x, ret;
    if (!enif_get_int(env, argv[0], &x)) {
 return enif_make_badarg(env);
    }
    ret = foo(x);
    return enif_make_int(env, ret);
}

static ERL_NIF_TERM bar_nif(ErlNifEnv* env, int argc, const ERL_NIF_TERM argv[])
{
    int y, ret;
    if (!enif_get_int(env, argv[0], &y)) {
 return enif_make_badarg(env);
    }
    ret = bar(y);
    return enif_make_int(env, ret);
}

static ErlNifFunc nif_funcs[] = {
    {"foo", 1, foo_nif},
    {"bar", 1, bar_nif}
};

ERL_NIF_INIT(complex6, nif_funcs, NULL, NULL, NULL, NULL)
```

Here, `ERL_NIF_INIT` has the following arguments:

- The first argument must be the name of the Erlang module as a C-identifier. It will be stringified by the macro.
- The second argument is the array of `ErlNifFunc` structures containing name, arity, and function pointer of each NIF.
- The remaining arguments are pointers to callback functions that can be used to initialize the library. They are not used in this simple example, hence they are all set to `NULL`.

Function arguments and return values are represented as values of type `ERL_NIF_TERM`. Here, functions like `enif_get_int` and `enif_make_int` are used to convert between Erlang term and C-type. If the function argument `argv[0]` is not an integer, `enif_get_int` returns false, in which case it returns by throwing a `badarg`-exception with `enif_make_badarg`.

9.8.3 Running the Example

Step 1. Compile the C code:

```
unix> gcc -o complex6_nif.so -fpic -shared complex.c complex6_nif.c
windows> cl -LD -MD -Fe complex6_nif.dll complex.c complex6_nif.c
```

Step 2: Start Erlang and compile the Erlang code:

```
> erl
Erlang R13B04 (erts-5.7.5) [64-bit] [smp:4:4] [rq:4] [async-threads:0] [kernel-poll:false]

Eshell V5.7.5  (abort with ^G)
1> c(complex6).
{ok,complex6}
```

Step 3: Run the example:

```
3> complex6:foo(3).
4
4> complex6:bar(5).
10
5> complex6:foo("not an integer").
** exception error: bad argument
     in function  complex6:foo/1
        called as comlpex6:foo("not an integer")
```

9.9 Debugging NIFs and Port Drivers

9.9.1 With great power comes great responsibilty

NIFs and port driver code run inside the Erlang VM OS process (the "Beam"). To maximize performance the code is called directly by the same threads executing Erlang beam code and has full access to all the memory of the OS process. A buggy NIF/driver can thus make severe damage by corrupting memory.

In a best case scenario such memory corruption is detected immediately causing the Beam to crash generating a core dump file which can be analyzed to find the bug. However, it is very common for memory corruption bugs to not be immediately detected when the faulty write happens, but instead much later, for example when the calling Erlang process is garbage collected. When that happens it can be very hard to find the root cause of the memory corruption by analysing the core dump. All traces that could have indicated which specific buggy NIF/driver that caused the corruption may be long gone.

9.9.2 The debug emulator

One way to make debugging easier is to run an emulator built with target `debug`. It will

- **Increase probability of detecting bugs earlier**. It contains a lot more runtime checks to ensure correct use of internal interfaces and data structures.
- **Generate a core dump that is easier to analyze**. Compiler optimizations are turned off, which stops the compiler from "optimizing away" variables, thus making it easier/possible to inspect their state.
- **Detect lock order violations**. A runtime lock checker will verify that the locks in the `erl_nif` and `erl_driver` APIs are seized in a consistent order that cannot result in deadlock bugs.

In fact, we recommend to use the debug emulator as default during development of NIFs and drivers, regardless if you are troubleshooting bugs or not. Some subtle bugs may not be detected by the normal emulator and just happen to work anyway by chance. However, another version of the emulator, or even different circumstances within the same emulator, may cause the bug to later provoke all kinds of problems.

The main disadvantage of the `debug` emulator is its reduced performance. The extra runtime checks and lack of compiler optimizations may result in a slowdown with a factor of two or more depending on load. The memory footprint should be about the same.

If the `debug` emulator is part of the Erlang/OTP installation, it can be started with the `-emu_type` option.

```
> erl -emu_type debug
Erlang/OTP 25 [erts-13.0.2] ... [type-assertions] [debug-compiled] [lock-checking]

Eshell V13.0.2  (abort with ^G)
1>
```

If the `debug` emulator is not part of the installation, you need to build it from the Erlang/OTP source code. After building from source either make an Erlang/OTP installation or you can run the debug emulator directly in the source tree with the `cerl` script:

```
> $ERL_TOP/bin/cerl -debug
Erlang/OTP 25 [erts-13.0.2] ... [type-assertions] [debug-compiled] [lock-checking]

Eshell V13.0.2  (abort with ^G)
1>
```

The `cerl` script can also be used as a convenient way to start the debugger `gdb` for core dump analysis:

```
> $ERL_TOP/bin/cerl -debug -core core.12345
or
> $ERL_TOP/bin/cerl -debug -rcore core.12345
```

The first variant starts Emacs and runs `gdb` within, while the other `-rcore` runs `gdb` directly in the terminal. Apart from starting `gdb` with the correct `beam.debug.smp` executable file it will also read the file `$ERL_TOP/erts/etc/unix/etp-commands` which contains a lot of `gdb` command for inspecting a beam core dump. For example, the command `etp` that will print the content of an Erlang term (`Eterm`) in plain Erlang syntax.

9.9.3 Address Sanitizer

AddressSanitizer (asan) is an open source programming tool that detects memory corruption bugs such as buffer overflows, use-after-free and memory leaks. AddressSanitizer is based on compiler instrumentation and is supported by both gcc and clang.

Similar to the `debug` emulator, the `asan` emulator runs slower than normal, about 2-3 times slower. However, it also has a larger memory footprint, about 3 times more memory than normal.

To get full effect you should compile both your own NIF/driver code as well as the Erlang emulator with AddressSanitizer instrumentation. Compile your own code by passing option `-fsanitize=address` to gcc or clang. Other recommended options that will improve the fault identification are `-fno-common` and `-fno-omit-frame-pointer`.

Build and run the emulator with AddressSanitizer support by using the same procedure as for the debug emulator, except use the `asan` build target instead of `debug`.

Run in source tree

If you run the `asan` emulator directly in the source tree with the `cerl` script you only need to set environment variable `ASAN_LOG_DIR` to the directory where the error log files will be generated.

```
> export ASAN_LOG_DIR=/my/asan/log/dir
> $ERL_TOP/bin/cerl -asan
Erlang/OTP 25 [erts-13.0.2] ... [address-sanitizer]

Eshell V13.0.2  (abort with ^G)
1>
```

You may however also want to set `ASAN_OPTIONS="halt_on_error=true"` if you want the emulator to crash when an error is detected.

Run installed Erlang/OTP

If you run the `asan` emulator in an installed Erlang/OTP with `erl -emu_type asan` you need to set the path to the error log **file** with

```
> export ASAN_OPTIONS="log_path=/my/asan/log/file"
```

To avoid false positive memory leak reports from the emulator itself set `LSAN_OPTIONS` (LSAN=LeakSanitizer):

```
> export LSAN_OPTIONS="suppressions=$ERL_TOP/erts/emulator/asan/suppress"
```

The `suppress` file is currently not installed but can be copied manually from the source tree to wherever you want it.

Memory corruption errors are reported by AddressSanitizer when they happen, but memory leaks are only checked and reported by default then the emulator terminates.

9.9.4 Valgrind

An even more heavy weight debugging tool is **Valgrind**. It can also find memory corruption bugs and memory leaks similar to `asan`. Valgrind is not as good at buffer overflow bugs, but it will find use of undefined data, which is a type of error that `asan` cannot detect.

Valgrind is much slower than `asan` and it is incapable at exploiting CPU multicore processing. We therefore recommend `asan` as the first choice before trying valgrind.

Valgrind runs as a virtual machine itself, emulating execution of hardware machine instructions. This means you can run almost any program unchanged on valgrind. However, we have found that the beam executable benefits from being compiled with special adaptions for running on valgrind.

Build the emulator with `valgrind` target the same as is done for `debug` and `asan`. Note that `valgrind` needs to be installed on the machine before the build starts.

Run the `valgrind` emulator directly in the source tree with the `cerl` script. Set environment variable `VALGRIND_LOG_DIR` to the directory where the error log files will be generated.

```
> export VALGRIND_LOG_DIR=/my/valgrind/log/dir
> $ERL_TOP/bin/cerl -valgrind
Erlang/OTP 25 [erts-13.0.2] ... [valgrind-compiled]

Eshell V13.0.2  (abort with ^G)
1>
```

9.9.5 rr - Record and Replay

Last but not least, the fantastic interactive debugging tool **rr**, developed by Mozilla as open source. `rr` stands for Record and Replay. While a core dump represents only a static snapshot of the OS process when it crashed, with `rr` you instead record the entire session, from start of the OS process to the end (the crash). You can then replay that session from within `gdb`. Single step, set breakpoints and watchpoints, and even **execute backwards**.

Considering its powerful utility, `rr` is remarkably light weight. It runs on Linux with any reasonably modern x86 CPU. You may get a two times slowdown when executing in recording mode. The big weakness is its inability to

exploite CPU multicore processing. If the bug is a race condition between concurrently running threads, it may be hard to reproduce with `rr`.

`rr` does not require any special instrumented compilation. However, if possible, run it together with the `debug` emulator, as that will result in a much nicer debugging experience. You run `rr` in the source tree using the `cerl` script.

Here is an example of a typical session. First we catch the crash in an rr recording session:

```
> $ERL_TOP/bin/cerl -debug -rr
rr: Saving execution to trace directory /home/foobar/.local/share/rr/beam.debug.smp-1.
Erlang/OTP 25 [erts-13.0.2]

Eshell V13.0.2  (abort with ^G)
1> mymod:buggy_nif().
Segmentation fault
```

Now we can replay that session with `rr replay`:

```
> rr replay
GNU gdb (Ubuntu 9.2-0ubuntu1~20.04.1) 9.2
:
(rr) continue
:
Thread 2 received signal SIGSEGV, Segmentation fault.
(rr) backtrace
```

You get the call stack at the moment of the crash. Bad luck, it is somewhere deep down in the garbage collection of the beam. But you manage to figure out that variable `hp` points to a broken Erlang term.

Set a watch point on that memory position and resume execution **backwards**. The debugger will then stop at the exact position when that memory position `*hp` was written.

```
(rr) watch -l *hp
Hardware watchpoint 1: -location *hp
(rr) reverse-continue
Continuing.

Thread 2 received signal SIGSEGV, Segmentation fault.
```

This is a quirk to be aware about. We started by executing forward until it crashed with SIGSEGV. We are now executing backwards from that point, so we are hitting the same SIGSEGV again but from the other direction. Just continue backwards once more to move past it.

```
(rr) reverse-continue
Continuing.

Thread 2 hit Hardware watchpoint 1: -location *hp

Old value = 42
New value = 0
```

And here we are at the position when someone wrote a broken term on the process heap. Note that "Old value" and "New value" are reversed when we execute backwards. In this case the value 42 was written on the heap. Let's see who the guilty one is:

```
(rr) backtrace
```

10 OTP Design Principles

10.1 Overview

The **OTP design principles** define how to structure Erlang code in terms of processes, modules, and directories.

10.1.1 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 that perform computations, that is, they do the actual work.
- Supervisors are processes that 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, which makes it possible to design and program fault-tolerant software.

In the following figure, square boxes represents supervisors and circles represent workers:

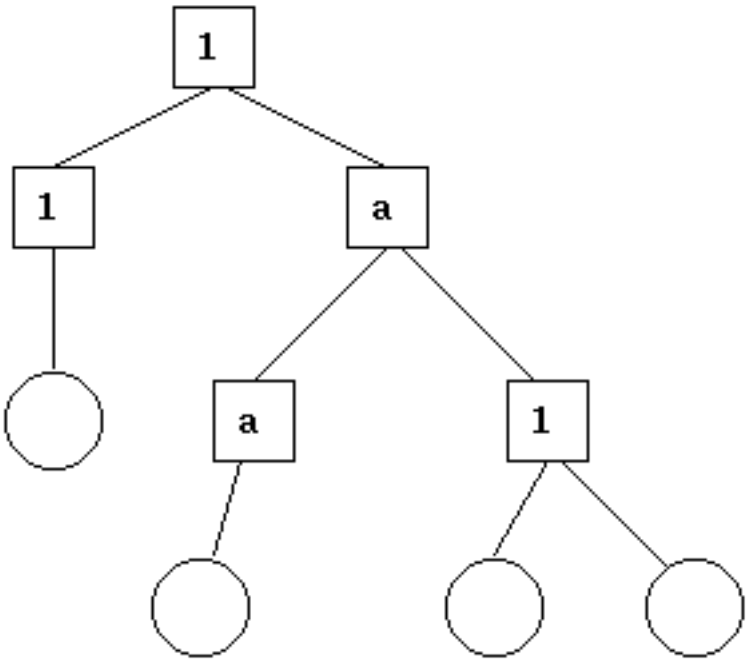


Figure 1.1: Supervision Tree

10.1.2 Behaviours

In a supervision tree, many of the processes have similar structures, they follow similar patterns. For example, the supervisors are similar in structure. The only difference between them is which child processes they supervise. Many of the workers are servers in a server-client relation, finite-state machines, or event handlers.

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 is to export a pre-defined set of functions, the **callback functions**.

The following example 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
```

Notice the following:

- The code in `server` can be reused to build many different servers.

- The server name, in this example the atom `ch2`, is hidden from the users of the client functions. This means that the name can be changed without affecting them.
- The protocol (messages sent to and received from the server) is also hidden. This is good programming practice and allows one to change the protocol without changing the code using the interface functions.
- The functionality of `server` can be extended without having to change `ch2` or any other callback module.

In `ch1.erl` and `ch2.erl` above, the implementation of `channels/0`, `alloc/1`, and `free/2` has been intentionally left out, as it is not relevant to the example. For completeness, one way to write these functions is given below. This is an example only, a realistic implementation must be able to handle situations like running out of channels to allocate, and so on.

```
channels() ->
   {_Allocated = [], _Free = lists:seq(1,100)}.

alloc({Allocated, [H|T] = _Free}) ->
   {H, {[H|Allocated], T}}.

free(Ch, {Alloc, Free} = Channels) ->
   case lists:member(Ch, Alloc) of
      true ->
         {lists:delete(Ch, Alloc), [Ch|Free]};
      false ->
         Channels
   end.
```

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

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

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

The standard Erlang/OTP behaviours are:

- gen_server

  For implementing the server of a client-server relation
- gen_statem

  For implementing state machines
- gen_event

  For implementing event handling functionality
- supervisor

  For implementing a supervisor in a supervision tree

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

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

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

10.1.3 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 following two applications:

- Kernel - Functionality necessary to run Erlang
- STDLIB - Erlang standard libraries

The application concept applies both to program structure (processes) and directory structure (modules).

The simplest applications do not have any processes, but consist 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.

10.1.4 Releases

A **release** is a complete system made out from a subset of Erlang/OTP applications and a set of user-specific applications.

How to program releases is described in Releases.

How to install a release in a target environment is described in the section about target systems in Section 2 System Principles.

10.1.5 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.

10.2 gen_server Behaviour

This section is to be read with the gen_server(3) manual page in `stdlib`, where all interface functions and callback functions are described in detail.

10.2.1 Client-Server Principles

The client-server model is characterized by a central server and an arbitrary number of clients. The client-server model is used for resource management operations, where several different clients want to share a common resource. The server is responsible for managing this resource.

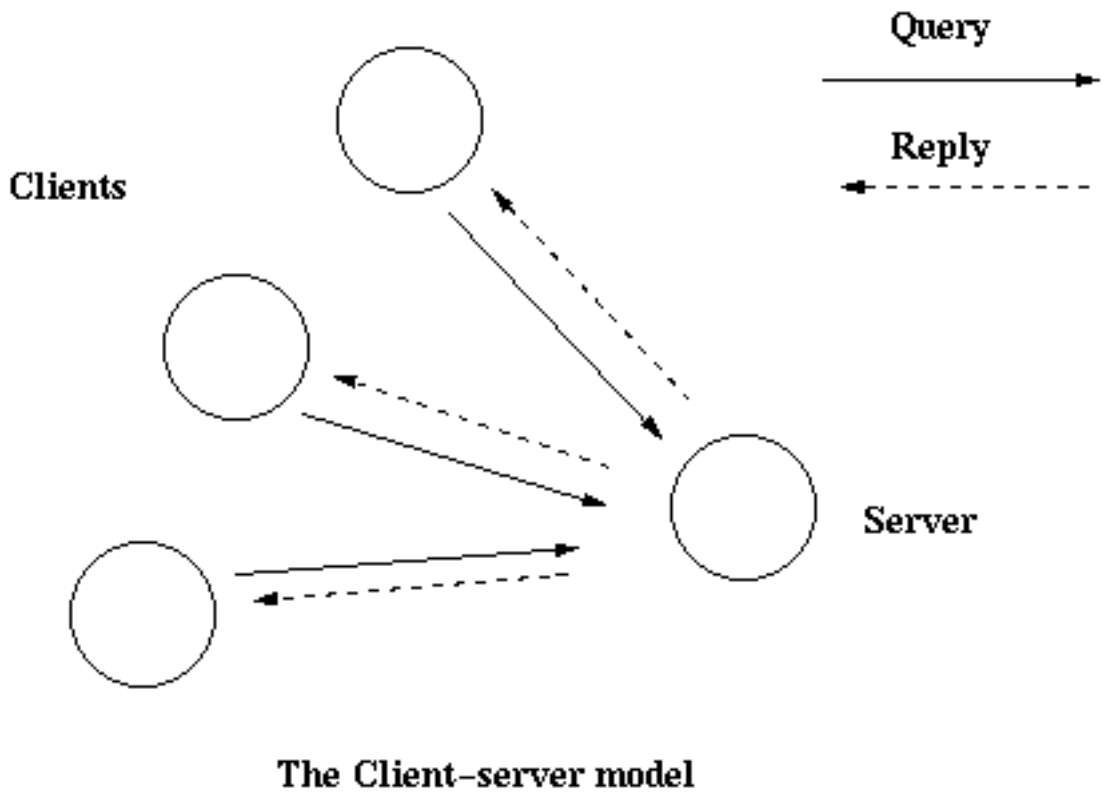


Figure 2.1: Client-Server Model

10.2.2 Example

An example of a simple server written in plain Erlang is provided in Overview. The server can be reimplemented 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.

10.2.3 Starting a Gen_Server

In the example in the previous section, `gen_server` is started by calling `ch3:start_link()`:

```
start_link() ->
    gen_server:start_link({local, ch3}, ch3, [], []) => {ok, Pid}
```

`start_link` calls 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. The gen_server is then locally registered as `ch3`.

  If the name is omitted, the `gen_server` is not registered. Instead its pid must be used. The name can 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.

  The interface functions (`start_link`, `alloc`, and `free`) are then 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 that 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 the `gen_server(3)` manual page 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()}.
```

`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, that is, started by a supervisor. There is another function, `gen_server:start`, to start a standalone `gen_server`, that is, a `gen_server` that is not part of a supervision tree.

10.2.4 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 that is to 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.

10.2.5 Asynchronous Requests - Cast

The asynchronous request `free(Ch)` is implemented using `gen_server:cast/2`:

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

`ch3` is the name of the `gen_server`. `{free, Ch}` is the actual request.

The request is made into a message and sent to the `gen_server`. `cast`, and thus `free`, then returns `ok`.

When the request is received, the `gen_server` calls `handle_cast(Request, State)`, which is expected to return a tuple `{noreply,State1}`. `State1` is a new value for the state of the `gen_server`.

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

In this case, the new state is the updated list of available channels `Chs2`. The `gen_server` is now ready for new requests.

10.2.6 Stopping

In a Supervision Tree

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

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

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

...

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

Standalone Gen_Servers

If the `gen_server` is not part of a supervision tree, a stop function can 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 causes the `gen_server` to call `terminate(normal, State1)` and then it terminates gracefully.

10.2.7 Handling Other Messages

If the `gen_server` is to 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}.
```

The `code_change` method must also be implemented.

```
code_change(OldVsn, State, Extra) ->
    ..code to convert state (and more) during code change
    {ok, NewState}.
```

10.3 gen_statem Behaviour

This section is to be read with the `gen_statem(3)` manual page in STDLIB, where all interface functions and callback functions are described in detail.

10.3.1 Event-Driven State Machines

Established Automata Theory does not deal much with how a **state transition** is triggered, but assumes that the output is a function of the input (and the state) and that they are some kind of values.

For an Event-Driven State Machine, the input is an event that triggers a **state transition** and the output is actions executed during the **state transition**. Analogously to the mathematical model of a Finite State Machine, it can be described as a set of relations of the following form:

```
State(S) x Event(E) -> Actions(A), State(S')
```

These relations are interpreted as follows: if we are in state `S` and event `E` occurs, we are to perform actions `A`, and make a transition to state `S'`. Notice that `S'` can be equal to `S`, and that `A` can be empty.

In `gen_statem` we define a **state change** as a **state transition** in which the new state `S'` is different from the current state `S`, where "different" means Erlang's strict inequality: `=/=` also known as "does not match". `gen_statem` does more things during **state changes** than during other **state transitions**.

As `A` and `S'` depend only on `S` and `E`, the kind of state machine described here is a Mealy machine (see, for example, the Wikipedia article "Mealy machine").

Like most `gen_` behaviours, `gen_statem` keeps a server `Data` besides the state. Because of this, and as there is no restriction on the number of states (assuming that there is enough virtual machine memory) or on the number of distinct input events, a state machine implemented with this behaviour is in fact Turing complete. But it feels mostly like an Event-Driven Mealy machine.

10.3.2 When to use gen_statem

If your process logic is convenient to describe as a state machine, and you want any of these `gen_statem` key features:

- Co-located callback code for each state, for all **Event Types** (such as **call**, **cast** and **info**)
- Postponing Events (a substitute for selective receive)
- Inserted Events (that is, events from the state machine to itself; for purely internal events in particular)
- **State Enter Calls** (callback on state entry co-located with the rest of each state's callback code)
- Easy-to-use time-outs (State Time-Outs, Event Time-Outs and Generic Time-Outs (named time-outs))

If so, or if possibly needed in future versions, then you should consider using `gen_statem` over `gen_server`.

For simple state machines not needing these features `gen_server` works just fine. It also has got smaller call overhead, but we are talking about something like 2 vs 3.3 microseconds call roundtrip time here, so if the server callback does just a little bit more than just replying, or if the call is not extremely frequent, that difference will be hard to notice.

10.3.3 Callback Module

The **callback module** contains functions that implement the state machine. When an event occurs, the `gen_statem` behaviour engine calls a function in the **callback module** with the event, current state and server data. This function performs the actions for this event, and returns the new state and server data and also actions to be performed by the behaviour engine.

The behaviour engine holds the state machine state, server data, timer references, a queue of postponed messages and other metadata. It receives all process messages, handles the system messages, and calls the **callback module** with machine specific events.

The **callback module** can be changed for a running server using any of the transition actions `{change_callback_module, NewModule}`, `{push_callback_module, NewModule}` or

`pop_callback_module`. Note that this is a pretty esoteric thing to do... The origin for this feature is a protocol that after version negotiation branches off into quite different state machines depending on the protocol version. There *might* be other use cases. *Beware* that the new callback module completely replaces the previous behaviour module, so all relevant callback functions have to handle the state and data from the previous callback module.

10.3.4 Callback Modes

The `gen_statem` behaviour supports two **callback modes**:

`state_functions`

Events are handled by one callback function per state.

`handle_event_function`

Events are handled by one single callback function.

The **callback mode** is a property of the **callback module** and is set at server start. It may be changed due to a code upgrade/downgrade, or when changing the **callback module**.

See the section **State Callback** that describes the event handling callback function(s).

The **callback mode** is selected by implementing a mandatory callback function `Module:callback_mode()` that returns one of the **callback modes**.

The `Module:callback_mode()` function may also return a list containing the **callback mode** and the atom `state_enter` in which case **state enter calls** are activated for the **callback mode**.

Choosing the Callback Mode

The short version: choose `state_functions` - it is the one most like `gen_fsm`. But if you do not want the restriction that the state must be an atom, or if you do not want to write one **state callback** function per state; please read on...

The two **callback modes** give different possibilities and restrictions, with one common goal: to handle all possible combinations of events and states.

This can be done, for example, by focusing on one state at the time and for every state ensure that all events are handled. Alternatively, you can focus on one event at the time and ensure that it is handled in every state. You can also use a mix of these strategies.

With `state_functions`, you are restricted to use atom-only states, and the `gen_statem` engine branches depending on state name for you. This encourages the **callback module** to co-locate the implementation of all event actions particular to one state in the same place in the code, hence to focus on one state at the time.

This mode fits well when you have a regular state diagram, like the ones in this chapter, which describes all events and actions belonging to a state visually around that state, and each state has its unique name.

With `handle_event_function`, you are free to mix strategies, as all events and states are handled in the same callback function.

This mode works equally well when you want to focus on one event at the time or on one state at the time, but function `Module:handle_event/4` quickly grows too large to handle without branching to helper functions.

The mode enables the use of non-atom states, for example, complex states or even hierarchical states. See section Complex State. If, for example, a state diagram is largely alike for the client side and the server side of a protocol, you can have a state `{StateName,server}` or `{StateName,client}`, and make `StateName` determine where in the code to handle most events in the state. The second element of the tuple is then used to select whether to handle special client-side or server-side events.

10.3.5 State Callback

The **state callback** is the callback function that handles an event in the current state, and which function that is depends on the **callback mode**:

`state_functions`
: The event is handled by:
`Module:StateName(EventType, EventContent, Data)`

This form is the one mostly used in the Example section.

`handle_event_function`
: The event is handled by:
`Module:handle_event(EventType, EventContent, State, Data)`

See section **One State Callback** for an example.

The state is either the name of the function itself or an argument to it. The other arguments are the `EventType` and the event dependent `EventContent`, both described in section Event Types and Event Content, and the current server `Data`.

State enter calls are also handled by the event handler and have slightly different arguments. See section State Enter Calls.

The **state callback** return values are defined in the description of `Module:StateName/3` in the `gen_statem` manual page, but here is a more readable list:

```
{next_state, NextState, NewData, Actions}
{next_state, NextState, NewData}
```

Set next state and update the server data. If the `Actions` field is used, execute **transition actions**. An empty `Actions` list is equivalent to not returning the field.

See section **Transition Actions** for a list of possible **transition actions**.

If `NextState =/= State` this is a **state change** so the extra things `gen_statem` does are: the event queue is restarted from the oldest postponed event, any current state time-out is cancelled, and a state enter call is performed, if enabled.

```
{keep_state, NewData, Actions}
{keep_state, NewData}
```

Same as the `next_state` values with `NextState =:= State`, that is, no **state change**.

```
{keep_state_and_data, Actions}
keep_state_and_data
```

Same as the `keep_state` values with `NextData =:= Data`, that is, no change in server data.

```
{repeat_state, NewData, Actions}
{repeat_state, NewData}
{repeat_state_and_data, Actions}
repeat_state_and_data
```

Same as the `keep_state` or `keep_state_and_data` values, and if State Enter Calls are enabled, repeat the **state enter call** as if this state was entered again.

If these return values are used from a **state enter call** the `OldState` does not change, but if used from an event handling **state callback** the new **state enter call's** `OldState` will be the current state.

```
{stop, Reason, NewData}
{stop, Reason}
```

Stop the server with reason `Reason`. If the `NewData` field is used, first update the server data.

`{stop_and_reply, Reason, NewData, ReplyActions}`
`{stop_and_reply, Reason, ReplyActions}`

Same as the `stop` values, but first execute the given **transition actions** that may only be reply actions.

The First State

To decide the first state the `Module:init(Args)` callback function is called before any **state callback** is called. This function behaves like a **state callback** function, but gets its only argument `Args` from the `gen_statem start/3,4` or `start_link/3,4` function, and returns `{ok, State, Data}` or `{ok, State, Data, Actions}`. If you use the `postpone` action from this function, that action is ignored, since there is no event to postpone.

10.3.6 Transition Actions

In the first section (Event-Driven State Machines), actions were mentioned as a part of the general state machine model. These general actions are implemented with the code that **callback module** `gen_statem` executes in an event-handling callback function before returning to the `gen_statem` engine.

There are more specific **transition actions** that a callback function can command the `gen_statem` engine to do after the callback function return. These are commanded by returning a list of actions in the return value from the callback function. These are the possible **transition actions**:

`postpone`
`{postpone, Boolean}`
If set postpone the current event, see section Postponing Events.

`hibernate`
`{hibernate, Boolean}`
If set hibernate the `gen_statem`, treated in section Hibernation.

`{state_timeout, Time, EventContent}`
`{state_timeout, Time, EventContent, Opts}`
`{state_timeout, update, EventContent}`
`{state_timeout, cancel}`
Start, update or cancel a state time-out, read more in sections Time-Outs and State Time-Outs.

`{{timeout, Name}, Time, EventContent}`
`{{timeout, Name}, Time, EventContent, Opts}`
`{{timeout, Name}, update, EventContent}`
`{{timeout, Name}, cancel}`
Start, update or cancel a generic time-out, read more in sections Time-Outs and Generic Time-Outs.

`{timeout, Time, EventContent}`
`{timeout, Time, EventContent, Opts}`
`Time`
Start an event time-out, see more in sections Time-Outs and Event Time-Outs.

`{reply, From, Reply}`
Reply to a caller, mentioned at the end of section All State Events.

`{next_event, EventType, EventContent}`
Generate the next event to handle, see section Inserted Events.

`{change_callback_module, NewModule}`
Change the **callback module** for the running server. This can be done during any **state transition**, whether it is a **state change** or not, but it can *not* be done from a **state enter call**.

`{push_callback_module, NewModule}`
Push the current **callback module** to the top of an internal stack of callback modules and set the new **callback module** for the running server. Otherwise like `{change_callback_module, NewModule}` above.

`pop_callback_module`
: Pop the top module from the internal stack of callback modules and set it to be the new **callback module** for the running server. If the stack is empty the server fails. Otherwise like `{change_callback_module, NewModule}` above.

For details, see the `gen_statem(3)` manual page for type `action()`. You can, for example, reply to many callers, generate multiple next events, and set a time-out to use absolute instead of relative time (using the `Opts` field).

Among these **transition actions** only to reply to a caller is an immediate action. The others are collected and handled later during the **state transition**. Inserted Events are stored and inserted all together, and the rest set transition options where the last of a specific type override the previous. See the description of a **state transition** in the `gen_statem(3)` manual page for type `transition_option()`.

The different Time-Outs and `next_event` actions generate new events with corresponding Event Types and Event Content .

10.3.7 Event Types and Event Content

Events are categorized in different **event types**. Events of all types are for a given state handled in the same callback function, and that function gets `EventType` and `EventContent` as arguments. The meaning of the `EventContent` depends on the `EventType`.

The following is a complete list of **event types** and where they come from:

`cast`
: Generated by `gen_statem:cast(ServerRef, Msg)` where `Msg` becomes the `EventContent`.

`{call,From}`
: Generated by `gen_statem:call(ServerRef, Request)` where `Request` becomes the `EventContent`. `From` is the reply address to use when replying either through the **transition action** `{reply,From,Reply}` or by calling `gen_statem:reply(From, Reply)` from the **callback module**.

`info`
: Generated by any regular process message sent to the `gen_statem` process. The process message becomes the `EventContent`.

`state_timeout`
: Generated by **transition action** `{state_timeout,Time,EventContent}` state timer timing out. Read more in sections Time-Outs and State Time-Outs.

`{timeout,Name}`
: Generated by **transition action** `{{timeout,Name},Time,EventContent}` generic timer timing out. Read more in sections Time-Outs and Generic Time-Outs.

`timeout`
: Generated by **transition action** `{timeout,Time,EventContent}` (or its short form `Time`) event timer timing out. Read more in sections Time-Outs and Event Time-Outs.

`internal`
: Generated by **transition action** `{next_event,internal,EventContent}`. All **event types** above can also be generated using the `next_event` action: `{next_event,EventType,EventContent}`.

10.3.8 State Enter Calls

The `gen_statem` behaviour can if this is enabled, regardless of **callback mode**, automatically call the state callback with special arguments whenever the state changes so you can write state enter actions near the rest of the **state transition** rules. It typically looks like this:

```
StateName(enter, OldState, Data) ->
    ... code for state enter actions here ...
    {keep_state, NewData};
StateName(EventType, EventContent, Data) ->
    ... code for actions here ...
    {next_state, NewStateName, NewData}.
```

Since the **state enter call** is not an event there are restrictions on the allowed return value and State Transition Actions. You may not change the state, postpone this non-event, insert any events, or change the **callback module**.

The first state that is entered will get a **state enter call** with `OldState` equal to the current state.

You may repeat the **state enter call** using the `{repeat_state,...}` return value from the state callback. In this case `OldState` will also be equal to the current state.

Depending on how your state machine is specified, this can be a very useful feature, but it forces you to handle the **state enter calls** in all states. See also the State Enter Actions section.

10.3.9 Time-Outs

Time-outs in `gen_statem` are started from a **transition action** during a state transition that is when exiting from the **state callback**.

There are 3 types of time-outs in `gen_statem`:

`state_timeout`
: There is one State Time-Out that is automatically cancelled by a **state change**.

`{timeout, Name}`
: There are any number of Generic Time-Outs differing by their `Name`. They have no automatic cancelling.

`timeout`
: There is one Event Time-Out that is automatically cancelled by any event. Note that postponed and inserted events cancel this time-out just as external events.

When a time-out is started any running time-out of the same type; `state_timeout`, `{timeout, Name}` or `timeout`, is cancelled, that is, the time-out is restarted with the new time.

All time-outs has got an `EventContent` that is part of the **transition action** that starts the time-out. Different `EventContent`s does not create different time-outs. The `EventContent` is delivered to the **state callback** when the time-out expires.

Cancelling a Time-Out

If a time-out is started with the time `infinity` it will never time out, in fact it will not even be started, and any running time-out with the same tag will be cancelled. The `EventContent` will in this case be ignored, so why not set it to `undefined`.

A more explicit way to cancel a timer is to use a **transition action** on the form `{TimeoutType, cancel}` which is a feature introduced in OTP 22.1.

Updating a Time-Out

While a time-out is running, its `EventContent` can be updated using a **transition action** on the form `{TimeoutType, update, NewEventContent}` which is a feature introduced in OTP 22.1.

If this feature is used while no such `TimeoutType` is running then a time-out event is immediately delivered as when starting a Time-Out Zero.

Time-Out Zero

If a time-out is started with the time `0` it will actually not be started. Instead the time-out event will immediately be inserted to be processed after any events already enqueued, and before any not yet received external events. Note that some time-outs are automatically cancelled so if you for example combine postponing an event in a **state change** with starting an event time-out with time `0` there will be no time-out event inserted since the event time-out is cancelled by the postponed event that is delivered due to the state change.

10.3.10 Example

A door with a code lock can be seen as a state machine. Initially, the door is locked. When someone presses a button, an event is generated. The pressed buttons are collected, up to the number of buttons in the correct code. If correct, the door is unlocked for 10 seconds. If not correct, we wait for a new button to be pressed.

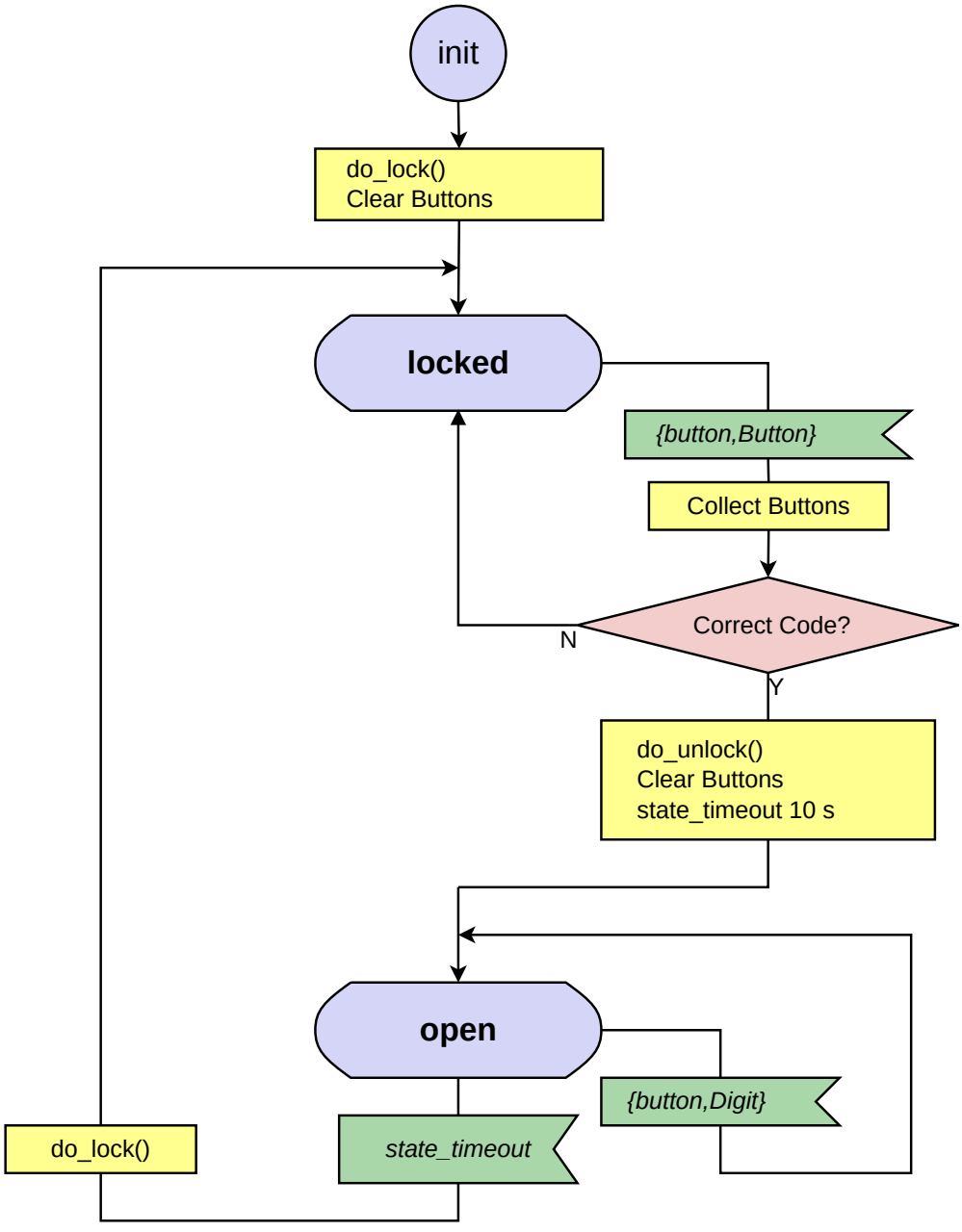


Figure 3.1: Code Lock State Diagram

This code lock state machine can be implemented using `gen_statem` with the following **callback module**:

```
-module(code_lock).
-behaviour(gen_statem).
-define(NAME, code_lock).

-export([start_link/1]).
-export([button/1]).
-export([init/1,callback_mode/0,terminate/3]).
-export([locked/3,open/3]).

start_link(Code) ->
    gen_statem:start_link({local,?NAME}, ?MODULE, Code, []).

button(Button) ->
    gen_statem:cast(?NAME, {button,Button}).

init(Code) ->
    do_lock(),
    Data = #{code => Code, length => length(Code), buttons => []},
    {ok, locked, Data}.

callback_mode() ->
    state_functions.
```

```
locked(
  cast, {button,Button},
  #{code := Code, length := Length, buttons := Buttons} = Data) ->
    NewButtons =
        if
            length(Buttons) < Length ->
                Buttons;
            true ->
                tl(Buttons)
        end ++ [Button],
    if
        NewButtons =:= Code -> % Correct
     do_unlock(),
            {next_state, open, Data#{buttons := []},
             [{state_timeout,10000,lock}]}; % Time in milliseconds
 true -> % Incomplete | Incorrect
            {next_state, locked, Data#{buttons := NewButtons}}
    end.
```

```
open(state_timeout, lock,  Data) ->
    do_lock(),
    {next_state, locked, Data};
open(cast, {button,_}, Data) ->
    {next_state, open, Data}.
```

```
do_lock() ->
    io:format("Lock~n", []).
do_unlock() ->
    io:format("Unlock~n", []).

terminate(_Reason, State, _Data) ->
    State =/= locked andalso do_lock(),
    ok.
```

The code is explained in the next sections.

10.3.11 Starting gen_statem

In the example in the previous section, `gen_statem` is started by calling `code_lock:start_link(Code)`:

```
start_link(Code) ->
    gen_statem:start_link({local,?NAME}, ?MODULE, Code, []).
```

`start_link` calls function `gen_statem:start_link/4`, which spawns and links to a new process, a `gen_statem`.

- The first argument, `{local,?NAME}`, specifies the name. In this case, the `gen_statem` is locally registered as `code_lock` through the macro `?NAME`.

  If the name is omitted, the `gen_statem` is not registered. Instead its pid must be used. The name can also be specified as `{global,Name}`, then the `gen_statem` is registered using `global:register_name/2` in Kernel.
- The second argument, `?MODULE`, is the name of the **callback module**, that is, the module where the callback functions are located, which is this module.

  The interface functions (`start_link/1` and `button/1`) are located in the same module as the callback functions (`init/1`, `locked/3`, and `open/3`). It is normally good programming practice to have the client-side code and the server-side code contained in one module.
- The third argument, `Code`, is a list of digits, which is the correct unlock code that is passed to callback function `init/1`.
- The fourth argument, `[]`, is a list of options. For the available options, see `gen_statem:start_link/3`.

If name registration succeeds, the new `gen_statem` process calls callback function `code_lock:init(Code)`. This function is expected to return `{ok, State, Data}`, where `State` is the initial state of the `gen_statem`, in this case `locked`; assuming that the door is locked to begin with. `Data` is the internal server data of the `gen_statem`. Here the server data is a map with key `code` that stores the correct button sequence, key `length` store its length, and key `buttons` that stores the collected buttons up to the same length.

```
init(Code) ->
    do_lock(),
    Data = #{code => Code, length => length(Code), buttons => []},
    {ok, locked, Data}.
```

Function `gen_statem:start_link` is synchronous. It does not return until the `gen_statem` is initialized and is ready to receive events.

Function `gen_statem:start_link` must be used if the `gen_statem` is part of a supervision tree, that is, started by a supervisor. Another function, `gen_statem:start` can be used to start a standalone `gen_statem`, that is, a `gen_statem` that is not part of a supervision tree.

Function `Module:callback_mode/0` selects the `CallbackMode` for the **callback module**, in this case `state_functions`. That is, each state has got its own handler function:

```
callback_mode() ->
    state_functions.
```

10.3.12 Handling Events

The function notifying the code lock about a button event is implemented using `gen_statem:cast/2`:

```
button(Button) ->
    gen_statem:cast(?NAME, {button,Button}).
```

The first argument is the name of the `gen_statem` and must agree with the name used to start it. So, we use the same macro `?NAME` as when starting. `{button,Button}` is the event content.

The event is sent to the `gen_statem`. When the event is received, the `gen_statem` calls `StateName(cast, Event, Data)`, which is expected to return a tuple `{next_state, NewStateName, NewData}`, or

`{next_state, NewStateName, NewData, Actions}`. `StateName` is the name of the current state and `NewStateName` is the name of the next state to go to. `NewData` is a new value for the server data of the `gen_statem`, and `Actions` is a list of actions to be performed by the `gen_statem` engine.

```
locked(
  cast, {button,Button},
  #{code := Code, length := Length, buttons := Buttons} = Data) ->
    NewButtons =
        if
            length(Buttons) < Length ->
                Buttons;
            true ->
                tl(Buttons)
        end ++ [Button],
    if
        NewButtons =:= Code -> % Correct
     do_unlock(),
            {next_state, open, Data#{buttons := []},
             [{state_timeout,10000,lock}]}; % Time in milliseconds
 true -> % Incomplete | Incorrect
            {next_state, locked, Data#{buttons := NewButtons}}
    end.
```

In state `locked`, when a button is pressed, it is collected with the last pressed buttons up to the length of the correct code, and compared with the correct code. Depending on the result, the door is either unlocked and the `gen_statem` goes to state `open`, or the door remains in state `locked`.

When changing to state `open`, the collected buttons are reset, the lock unlocked, and a state timer for 10 s is started.

```
open(cast, {button,_}, Data) ->
    {next_state, open, Data}.
```

In state `open`, a button event is ignored by staying in the same state. This can also be done by returning `{keep_state, Data}` or in this case since `Data` unchanged even by returning `keep_state_and_data`.

10.3.13 State Time-Outs

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

```
{next_state, open, Data#{buttons := []},
 [{state_timeout,10000,lock}]}; % Time in milliseconds
```

10,000 is a time-out value in milliseconds. After this time (10 seconds), a time-out occurs. Then, `StateName(state_timeout, lock, Data)` is called. The time-out occurs when the door has been in state `open` for 10 seconds. After that the door is locked again:

```
open(state_timeout, lock,  Data) ->
    do_lock(),
    {next_state, locked, Data};
```

The timer for a state time-out is automatically cancelled when the state machine does a **state change**.

You can restart, cancel or update a state time-out. See section Time-Outs for details.

10.3.14 All State Events

Sometimes events can arrive in any state of the `gen_statem`. It is convenient to handle these in a common state handler function that all state functions call for events not specific to the state.

Consider a `code_length/0` function that returns the length of the correct code. We dispatch all events that are not state-specific to the common function `handle_common/3`:

```
...
-export([button/1,code_length/0]).
...

code_length() ->
    gen_statem:call(?NAME, code_length).

...
locked(...) -> ... ;
locked(EventType, EventContent, Data) ->
    handle_common(EventType, EventContent, Data).

...
open(...) -> ... ;
open(EventType, EventContent, Data) ->
    handle_common(EventType, EventContent, Data).

handle_common({call,From}, code_length, #{code := Code} = Data) ->
    {keep_state, Data,
     [{reply,From,length(Code)}]}.
```

Another way to do it is through a convenience macro `?HANDLE_COMMON/0`:

```
...
-export([button/1,code_length/0]).
...

code_length() ->
    gen_statem:call(?NAME, code_length).

-define(HANDLE_COMMON,
    ?FUNCTION_NAME(T, C, D) -> handle_common(T, C, D)).
%%
handle_common({call,From}, code_length, #{code := Code} = Data) ->
    {keep_state, Data,
     [{reply,From,length(Code)}]}.

...
locked(...) -> ... ;
?HANDLE_COMMON.

...
open(...) -> ... ;
?HANDLE_COMMON.
```

This example uses `gen_statem:call/2`, which waits for a reply from the server. The reply is sent with a `{reply,From,Reply}` tuple in an action list in the `{keep_state, ...}` tuple that retains the current state. This return form is convenient when you want to stay in the current state but do not know or care about what it is.

If the common **state callback** needs to know the current state a function `handle_common/4` can be used instead:

```
-define(HANDLE_COMMON,
    ?FUNCTION_NAME(T, C, D) -> handle_common(T, C, ?FUNCTION_NAME, D)).
```

10.3.15 One State Callback

If **callback mode** `handle_event_function` is used, all events are handled in `Module:handle_event/4` and we can (but do not have to) use an event-centered approach where we first branch depending on event and then depending on state:

```
...
-export([handle_event/4]).

...
callback_mode() ->
    handle_event_function.

handle_event(cast, {button,Button}, State, #{code := Code} = Data) ->
    case State of
 locked ->
            #{length := Length, buttons := Buttons} = Data,
            NewButtons =
                if
                    length(Buttons) < Length ->
                        Buttons;
                    true ->
                        tl(Buttons)
                end ++ [Button],
            if
                NewButtons =:= Code -> % Correct
                    do_unlock(),
                    {next_state, open, Data#{buttons := []},
                     [{state_timeout,10000,lock}]}; % Time in milliseconds
                true -> % Incomplete | Incorrect
                    {keep_state, Data#{buttons := NewButtons}}
            end;
 open ->
            keep_state_and_data
    end;
handle_event(state_timeout, lock, open, Data) ->
    do_lock(),
    {next_state, locked, Data};
handle_event(
  {call,From}, code_length, _State, #{code := Code} = Data) ->
    {keep_state, Data,
     [{reply,From,length(Code)}]}.

...
```

10.3.16 Stopping

In a Supervision Tree

If the `gen_statem` is part of a supervision tree, no stop function is needed. The `gen_statem` is automatically terminated by its supervisor. Exactly how this is done is defined by a shutdown strategy set in the supervisor.

If it is necessary to clean up before termination, the shutdown strategy must be a time-out value and the `gen_statem` must in function `init/1` set itself to trap exit signals by calling `process_flag(trap_exit, true)`:

```
init(Args) ->
    process_flag(trap_exit, true),
    do_lock(),
    ...
```

When ordered to shut down, the `gen_statem` then calls callback function `terminate(shutdown, State, Data)`.

In this example, function `terminate/3` locks the door if it is open, so we do not accidentally leave the door open when the supervision tree terminates:

```
terminate(_Reason, State, _Data) ->
    State =/= locked andalso do_lock(),
    ok.
```

Standalone gen_statem

If the `gen_statem` is not part of a supervision tree, it can be stopped using `gen_statem:stop`, preferably through an API function:

```
...
-export([start_link/1,stop/0]).

...
stop() ->
    gen_statem:stop(?NAME).
```

This makes the `gen_statem` call callback function `terminate/3` just like for a supervised server and waits for the process to terminate.

10.3.17 Event Time-Outs

A time-out feature inherited from `gen_statem`'s predecessor `gen_fsm`, is an event time-out, that is, if an event arrives the timer is cancelled. You get either an event or a time-out, but not both.

It is ordered by the **transition action** `{timeout,Time,EventContent}`, or just an integer `Time`, even without the enclosing actions list (the latter is a form inherited from `gen_fsm`.

This type of time-out is useful for example to act on inactivity. Let us restart the code sequence if no button is pressed for say 30 seconds:

```
...

locked(timeout, _, Data) ->
    {next_state, locked, Data#{buttons := []}};
locked(
  cast, {button,Button},
  #{code := Code, length := Length, buttons := Buttons} = Data) ->
...
 true -> % Incomplete | Incorrect
            {next_state, locked, Data#{buttons := NewButtons},
             30000} % Time in milliseconds
...
```

Whenever we receive a button event we start an event time-out of 30 seconds, and if we get an **event type** of `timeout` we reset the remaining code sequence.

An event time-out is cancelled by any other event so you either get some other event or the time-out event. It is therefore not possible nor needed to cancel, restart or update an event time-out. Whatever event you act on has already cancelled the event time-out, so there is never a running event time-out while the **state callback** executes.

Note that an event time-out does not work well when you have for example a status call as in section All State Events, or handle unknown events, since all kinds of events will cancel the event time-out.

10.3.18 Generic Time-Outs

The previous example of state time-outs only work if the state machine stays in the same state during the time-out time. And event time-outs only work if no disturbing unrelated events occur.

You may want to start a timer in one state and respond to the time-out in another, maybe cancel the time-out without changing states, or perhaps run multiple time-outs in parallel. All this can be accomplished with generic time-outs.

They may look a little bit like event time-outs but contain a name to allow for any number of them simultaneously and they are not automatically cancelled.

Here is how to accomplish the state time-out in the previous example by instead using a generic time-out named for example `open`:

```
...
locked(
  cast, {button,Button},
  #{code := Code, length := Length, buttons := Buttons} = Data) ->
...
    if
        NewButtons =:= Code -> % Correct
     do_unlock(),
            {next_state, open, Data#{buttons := []},
             [{{timeout,open},10000,lock}]}; % Time in milliseconds
...

open({timeout,open}, lock, Data) ->
    do_lock(),
    {next_state,locked,Data};
open(cast, {button,_}, Data) ->
    {keep_state,Data};
...
```

Specific generic time-outs can just as state time-outs be restarted or cancelled by setting it to a new time or `infinity`.

In this particular case we do not need to cancel the time-out since the time-out event is the only possible reason to do a **state change** from `open` to `locked`.

Instead of bothering with when to cancel a time-out, a late time-out event can be handled by ignoring it if it arrives in a state where it is known to be late.

You can restart, cancel or update a generic time-out. See section Time-Outs for details.

10.3.19 Erlang Timers

The most versatile way to handle time-outs is to use Erlang Timers; see `erlang:start_timer/3,4`. Most time-out tasks can be performed with the time-out features in `gen_statem`, but an example of one that cannot is if you should need the return value from `erlang:cancel_timer(Tref)`, that is; the remaining time of the timer.

Here is how to accomplish the state time-out in the previous example by instead using an Erlang Timer:

```
...
locked(
  cast, {button,Button},
  #{code := Code, length := Length, buttons := Buttons} = Data) ->
...
    if
        NewButtons =:= Code -> % Correct
     do_unlock(),
     Tref =
                 erlang:start_timer(
                     10000, self(), lock), % Time in milliseconds
            {next_state, open, Data#{buttons := [], timer => Tref}};
...

open(info, {timeout,Tref,lock}, #{timer := Tref} = Data) ->
    do_lock(),
    {next_state,locked,maps:remove(timer, Data)};
open(cast, {button,_}, Data) ->
    {keep_state,Data};
...
```

Removing the `timer` key from the map when we do a **state change** to `locked` is not strictly necessary since we can only get into state `open` with an updated `timer` map value. But it can be nice to not have outdated values in the state `Data`!

If you need to cancel a timer because of some other event, you can use `erlang:cancel_timer(Tref)`. Note that no time-out message will arrive after this (because the timer has been explicitly canceled), unless you have already postponed one earlier (see the next section), so ensure that you do not accidentally postpone such messages. Also note that a time-out message may arrive during a **state callback** that is cancelling the timer, so you may have to read out such a message from the process mailbox, depending on the return value from `erlang:cancel_timer(Tref)`.

Another way to handle a late time-out can be to not cancel it, but to ignore it if it arrives in a state where it is known to be late.

10.3.20 Postponing Events

If you want to ignore a particular event in the current state and handle it in a future state, you can postpone the event. A postponed event is retried after a **state change**, that is, `OldState =/= NewState`.

Postponing is ordered by the **transition action** `postpone`.

In this example, instead of ignoring button events while in the `open` state, we can postpone them and they are queued and later handled in the `locked` state:

```
...
open(cast, {button,_}, Data) ->
    {keep_state,Data,[postpone]};
...
```

Since a postponed event is only retried after a **state change**, you have to think about where to keep a state data item. You can keep it in the server `Data` or in the `State` itself, for example by having two more or less identical states to keep a boolean value, or by using a complex state (see section Complex State) with **callback mode** `handle_event_function`. If a change in the value changes the set of events that is handled, then the value should be kept in the State. Otherwise no postponed events will be retried since only the server Data changes.

This is not important if you do not postpone events. But if you later decide to start postponing some events, then the design flaw of not having separate states when they should be, might become a hard-to-find bug.

Fuzzy State Diagrams

It is not uncommon that a state diagram does not specify how to handle events that are not illustrated in a particular state in the diagram. Hopefully this is described in an associated text or from the context.

Possible actions: ignore as in drop the event (maybe log it) or deal with the event in some other state as in postpone it.

Selective Receive

Erlang's selective receive statement is often used to describe simple state machine examples in straightforward Erlang code. The following is a possible implementation of the first example:

```
-module(code_lock).
-define(NAME, code_lock_1).
-export([start_link/1,button/1]).

start_link(Code) ->
    spawn(
      fun () ->
       true = register(?NAME, self()),
       do_lock(),
       locked(Code, length(Code), [])
      end).

button(Button) ->
    ?NAME ! {button,Button}.
```

```
locked(Code, Length, Buttons) ->
    receive
        {button,Button} ->
            NewButtons =
                if
                    length(Buttons) < Length ->
                        Buttons;
                    true ->
                        tl(Buttons)
                end ++ [Button],
            if
                NewButtons =:= Code -> % Correct
                    do_unlock(),
      open(Code, Length);
                true -> % Incomplete | Incorrect
                    locked(Code, Length, NewButtons)
            end
    end.
```

```
open(Code, Length) ->
    receive
    after 10000 -> % Time in milliseconds
     do_lock(),
     locked(Code, Length, [])
    end.

do_lock() ->
    io:format("Locked~n", []).
do_unlock() ->
    io:format("Open~n", []).
```

The selective receive in this case causes `open` to implicitly postpone any events to the `locked` state.

A selective receive cannot be used from a `gen_statem` behaviour (or from any `gen_*` behaviour), as the receive statement is within the `gen_*` engine itself. It must be there because all `sys` compatible behaviours must respond to system messages and therefore do that in their engine receive loop, passing non-system messages to the **callback module**.

The **transition action** `postpone` is designed to model selective receives. A selective receive implicitly postpones any not received events, but the `postpone` **transition action** explicitly postpones one received event.

Both mechanisms have the same theoretical time and memory complexity, while the selective receive language construct has smaller constant factors.

10.3.21 State Enter Actions

Say you have a state machine specification that uses state enter actions. Although you can code this using inserted events (described in the next section), especially if just one or a few states has got state enter actions, this is a perfect use case for the built in **state enter calls**.

You return a list containing `state_enter` from your `callback_mode/0` function and the `gen_statem` engine will call your **state callback** once with an event `(enter, OldState, ...)` whenever it does a **state change**. Then you just need to handle these event-like calls in all states.

```
...
init(Code) ->
    process_flag(trap_exit, true),
    Data = #{code => Code, length = length(Code)},
    {ok, locked, Data}.

callback_mode() ->
    [state_functions,state_enter].

locked(enter, _OldState, Data) ->
    do_lock(),
    {keep_state,Data#{buttons => []}};
locked(
  cast, {button,Button},
  #{code := Code, length := Length, buttons := Buttons} = Data) ->
...
    if
        NewButtons =:= Code -> % Correct
            {next_state, open, Data};
...

open(enter, _OldState, _Data) ->
    do_unlock(),
    {keep_state_and_data,
     [{state_timeout,10000,lock}]}; % Time in milliseconds
open(state_timeout, lock, Data) ->
    {next_state, locked, Data};
...
```

You can repeat the state enter code by returning one of `{repeat_state, ...}`, `{repeat_state_and_data,_}` or `repeat_state_and_data` that otherwise behaves exactly like their `keep_state` siblings. See the type `state_callback_result()` in the reference manual.

10.3.22 Inserted Events

It can sometimes be beneficial to be able to generate events to your own state machine. This can be done with the **transition action** `{next_event,EventType,EventContent}`.

You can generate events of any existing type, but the `internal` type can only be generated through action `next_event`. Hence, it cannot come from an external source, so you can be certain that an `internal` event is an event from your state machine to itself.

One example for this is to pre-process incoming data, for example decrypting chunks or collecting characters up to a line break.

Purists may argue that this should be modelled with a separate state machine that sends pre-processed events to the main state machine, but to decrease overhead the small pre-processing state machine can be implemented in the common state event handling of the main state machine using a few state data variables that then sends the pre-processed events as internal events to the main state machine. Using internal events also can make it easier to synchronize the state machines.

A variant of this is to use a complex state with **one state callback**. The state is then modeled with for example a tuple `{MainFSMState,SubFSMState}`.

To illustrate this we make up an example where the buttons instead generate down and up (press and release) events, and the lock responds to an up event only after the corresponding down event.

```
...
-export([down/1, up/1]).
...
down(Button) ->
    gen_statem:cast(?NAME, {down,Button}).

up(Button) ->
    gen_statem:cast(?NAME, {up,Button}).

...

locked(enter, _OldState, Data) ->
    do_lock(),
    {keep_state,Data#{buttons => []}};
locked(
  internal, {button,Button},
  #{code := Code, length := Length, buttons := Buttons} = Data) ->
...
```

```
handle_common(cast, {down,Button}, Data) ->
    {keep_state, Data#{button => Button}};
handle_common(cast, {up,Button}, Data) ->
    case Data of
        #{button := Button} ->
            {keep_state,maps:remove(button, Data),
             [{next_event,internal,{button,Button}}]};
        #{} ->
            keep_state_and_data
    end;
...

open(internal, {button,_}, Data) ->
    {keep_state,Data,[postpone]};
...
```

If you start this program with `code_lock:start([17])` you can unlock with `code_lock:down(17), code_lock:up(17).`

10.3.23 Example Revisited

This section includes the example after most of the mentioned modifications and some more using **state enter calls**, which deserves a new state diagram:

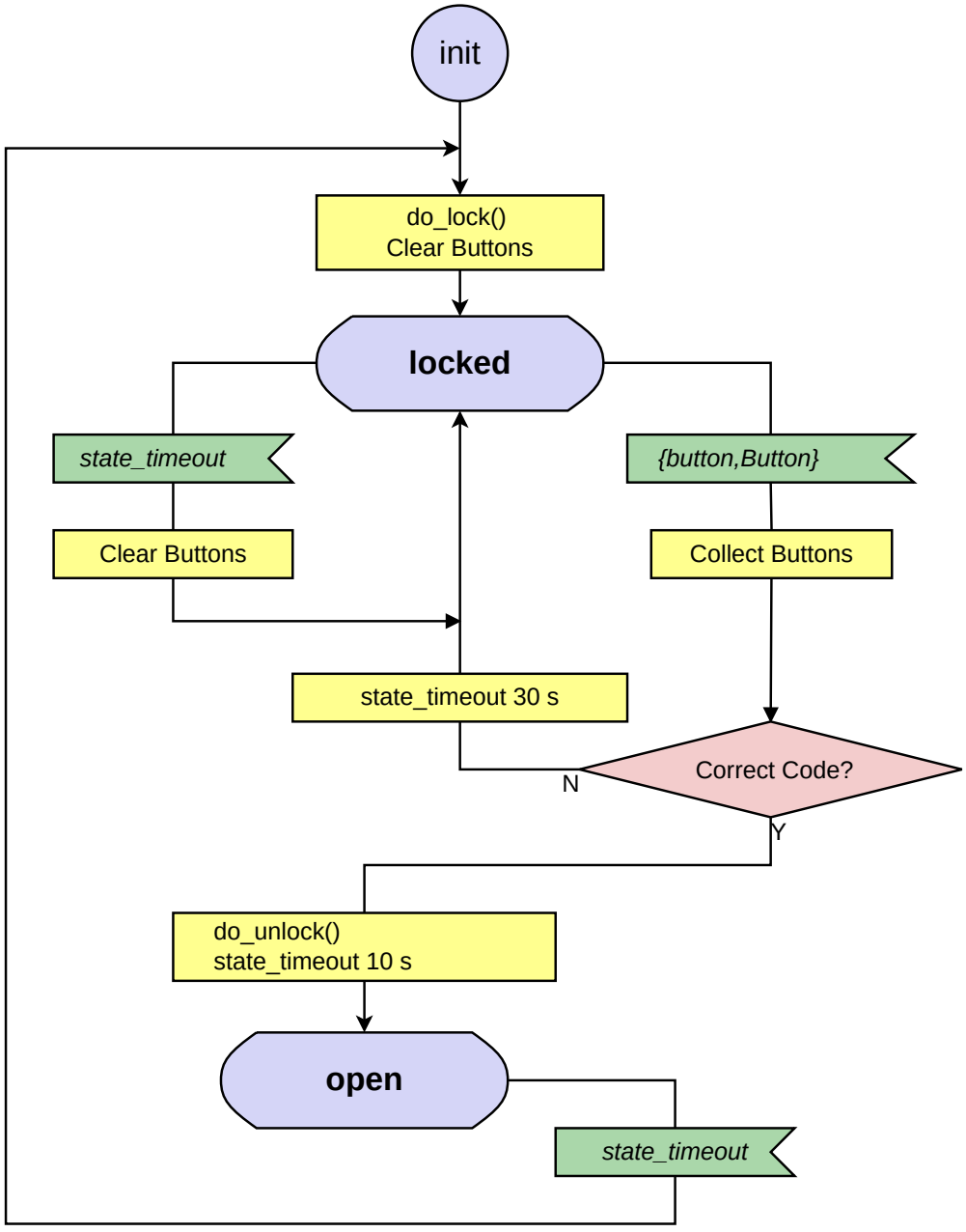


Figure 3.2: Code Lock State Diagram Revisited

Notice that this state diagram does not specify how to handle a button event in the state `open`. So, you need to read in some side notes, that is, here: that unspecified events shall be postponed (handled in some later state). Also, the state diagram does not show that the `code_length/0` call must be handled in every state.

Callback Mode: state_functions

Using state functions:

```
-module(code_lock).
-behaviour(gen_statem).
-define(NAME, code_lock_2).

-export([start_link/1,stop/0]).
-export([down/1,up/1,code_length/0]).
-export([init/1,callback_mode/0,terminate/3]).
-export([locked/3,open/3]).

start_link(Code) ->
    gen_statem:start_link({local,?NAME}, ?MODULE, Code, []).
stop() ->
    gen_statem:stop(?NAME).

down(Button) ->
    gen_statem:cast(?NAME, {down,Button}).
up(Button) ->
    gen_statem:cast(?NAME, {up,Button}).
code_length() ->
    gen_statem:call(?NAME, code_length).
```

```
init(Code) ->
    process_flag(trap_exit, true),
    Data = #{code => Code, length => length(Code), buttons => []},
    {ok, locked, Data}.

callback_mode() ->
    [state_functions,state_enter].

-define(HANDLE_COMMON,
    ?FUNCTION_NAME(T, C, D) -> handle_common(T, C, D)).
%%
handle_common(cast, {down,Button}, Data) ->
    {keep_state, Data#{button => Button}};
handle_common(cast, {up,Button}, Data) ->
    case Data of
        #{button := Button} ->
            {keep_state, maps:remove(button, Data),
             [{next_event,internal,{button,Button}}]};
        #{} ->
            keep_state_and_data
    end;
handle_common({call,From}, code_length, #{code := Code}) ->
    {keep_state_and_data,
     [{reply,From,length(Code)}]}.
```

```
locked(enter, _OldState, Data) ->
    do_lock(),
    {keep_state, Data#{buttons := []}};
locked(state_timeout, button, Data) ->
    {keep_state, Data#{buttons := []}};
locked(
  internal, {button,Button},
  #{code := Code, length := Length, buttons := Buttons} = Data) ->
    NewButtons =
        if
            length(Buttons) < Length ->
                Buttons;
            true ->
                tl(Buttons)
        end ++ [Button],
    if
        NewButtons =:= Code -> % Correct
            {next_state, open, Data};
 true -> % Incomplete | Incorrect
            {keep_state, Data#{buttons := NewButtons},
             [{state_timeout,30000,button}]} % Time in milliseconds
    end;
?HANDLE_COMMON.
```

```
open(enter, _OldState, _Data) ->
    do_unlock(),
    {keep_state_and_data,
     [{state_timeout,10000,lock}]}; % Time in milliseconds
open(state_timeout, lock, Data) ->
    {next_state, locked, Data};
open(internal, {button,_}, _) ->
    {keep_state_and_data, [postpone]};
?HANDLE_COMMON.

do_lock() ->
    io:format("Locked~n", []).
do_unlock() ->
    io:format("Open~n", []).

terminate(_Reason, State, _Data) ->
    State =/= locked andalso do_lock(),
    ok.
```

Callback Mode: handle_event_function

This section describes what to change in the example to use one `handle_event/4` function. The previously used approach to first branch depending on event does not work that well here because of the **state enter calls**, so this example first branches depending on state:

```
-export([handle_event/4]).
```

```
callback_mode() ->
    [handle_event_function,state_enter].
```

```
%%
%% State: locked
handle_event(enter, _OldState, locked, Data) ->
    do_lock(),
    {keep_state, Data#{buttons := []}};
handle_event(state_timeout, button, locked, Data) ->
    {keep_state, Data#{buttons := []}};
handle_event(
  internal, {button,Button}, locked,
  #{code := Code, length := Length, buttons := Buttons} = Data) ->
    NewButtons =
        if
            length(Buttons) < Length ->
                Buttons;
            true ->
                tl(Buttons)
        end ++ [Button],
    if
        NewButtons =:= Code -> % Correct
            {next_state, open, Data};
 true -> % Incomplete | Incorrect
            {keep_state, Data#{buttons := NewButtons},
             [{state_timeout,30000,button}]} % Time in milliseconds
    end;
```

```
%%
%% State: open
handle_event(enter, _OldState, open, _Data) ->
    do_unlock(),
    {keep_state_and_data,
     [{state_timeout,10000,lock}]}; % Time in milliseconds
handle_event(state_timeout, lock, open, Data) ->
    {next_state, locked, Data};
handle_event(internal, {button,_}, open, _) ->
    {keep_state_and_data,[postpone]};
```

```
%% Common events
handle_event(cast, {down,Button}, _State, Data) ->
    {keep_state, Data#{button => Button}};
handle_event(cast, {up,Button}, _State, Data) ->
    case Data of
        #{button := Button} ->
            {keep_state, maps:remove(button, Data),
             [{next_event,internal,{button,Button}},
              {state_timeout,30000,button}]}; % Time in milliseconds
        #{} ->
            keep_state_and_data
    end;
handle_event({call,From}, code_length, _State, #{length := Length}) ->
    {keep_state_and_data,
     [{reply,From,Length}]}.
```

Notice that postponing buttons from the `open` state to the `locked` state feels like a strange thing to do for a code lock, but it at least illustrates event postponing.

10.3.24 Filter the State

The example servers so far in this chapter print the full internal state in the error log, for example, when killed by an exit signal or because of an internal error. This state contains both the code lock code and which digits that remain to unlock.

This state data can be regarded as sensitive, and maybe not what you want in the error log because of some unpredictable event.

Another reason to filter the state can be that the state is too large to print, as it fills the error log with uninteresting details.

To avoid this, you can format the internal state that gets in the error log and gets returned from `sys:get_status/1,2` by implementing function `Module:format_status/2`, for example like this:

```
...
-export([init/1,terminate/3,format_status/2]).
...

format_status(Opt, [_PDict,State,Data]) ->
    StateData =
 {State,
  maps:filter(
    fun (code, _) -> false;
        (_, _) -> true
    end,
    Data)},
    case Opt of
 terminate ->
     StateData;
 normal ->
     [{data,[{"State",StateData}]}]
    end.
```

It is not mandatory to implement a `Module:format_status/2` function. If you do not, a default implementation is used that does the same as this example function without filtering the `Data` term, that is, `StateData = {State,Data}`, in this example containing sensitive information.

10.3.25 Complex State

The **callback mode** `handle_event_function` enables using a non-atom state as described in section Callback Modes, for example, a complex state term like a tuple.

One reason to use this is when you have a state item that when changed should cancel the state time-out, or one that affects the event handling in combination with postponing events. We will go for the latter and complicate the previous example by introducing a configurable lock button (this is the state item in question), which in the `open` state immediately locks the door, and an API function `set_lock_button/1` to set the lock button.

Suppose now that we call `set_lock_button` while the door is open, and we have already postponed a button event that was the new lock button:

```
1> code_lock:start_link([a,b,c], x).
{ok,<0.666.0>}
2> code_lock:button(a).
ok
3> code_lock:button(b).
ok
4> code_lock:button(c).
ok
Open
5> code_lock:button(y).
ok
6> code_lock:set_lock_button(y).
x
% What should happen here?  Immediate lock or nothing?
```

We could say that the button was pressed too early so it is not to be recognized as the lock button. Or we can make the lock button part of the state so when we then change the lock button in the locked state, the change becomes a **state change** and all postponed events are retried, therefore the lock is immediately locked!

We define the state as `{StateName,LockButton}`, where `StateName` is as before and `LockButton` is the current lock button:

```
-module(code_lock).
-behaviour(gen_statem).
-define(NAME, code_lock_3).

-export([start_link/2,stop/0]).
-export([button/1,set_lock_button/1]).
-export([init/1,callback_mode/0,terminate/3]).
-export([handle_event/4]).

start_link(Code, LockButton) ->
    gen_statem:start_link(
        {local,?NAME}, ?MODULE, {Code,LockButton}, []).
stop() ->
    gen_statem:stop(?NAME).

button(Button) ->
    gen_statem:cast(?NAME, {button,Button}).
set_lock_button(LockButton) ->
    gen_statem:call(?NAME, {set_lock_button,LockButton}).
```

```
init({Code,LockButton}) ->
    process_flag(trap_exit, true),
    Data = #{code => Code, length => length(Code), buttons => []},
    {ok, {locked,LockButton}, Data}.

callback_mode() ->
    [handle_event_function,state_enter].

%% State: locked
handle_event(enter, _OldState, {locked,_}, Data) ->
    do_lock(),
    {keep_state, Data#{buttons := []}};
handle_event(state_timeout, button, {locked,_}, Data) ->
    {keep_state, Data#{buttons := []}};
handle_event(
  cast, {button,Button}, {locked,LockButton},
  #{code := Code, length := Length, buttons := Buttons} = Data) ->
    NewButtons =
        if
            length(Buttons) < Length ->
                Buttons;
            true ->
                tl(Buttons)
        end ++ [Button],
    if
        NewButtons =:= Code -> % Correct
            {next_state, {open,LockButton}, Data};
 true -> % Incomplete | Incorrect
            {keep_state, Data#{buttons := NewButtons},
             [{state_timeout,30000,button}]} % Time in milliseconds
    end;
```

```
%%
%% State: open
handle_event(enter, _OldState, {open,_}, _Data) ->
    do_unlock(),
    {keep_state_and_data,
     [{state_timeout,10000,lock}]}; % Time in milliseconds
handle_event(state_timeout, lock, {open,LockButton}, Data) ->
    {next_state, {locked,LockButton}, Data};
handle_event(cast, {button,LockButton}, {open,LockButton}, Data) ->
    {next_state, {locked,LockButton}, Data};
handle_event(cast, {button,_}, {open,_}, _Data) ->
    {keep_state_and_data,[postpone]};
```

```
%%
%% Common events
handle_event(
  {call,From}, {set_lock_button,NewLockButton},
  {StateName,OldLockButton}, Data) ->
    {next_state, {StateName,NewLockButton}, Data,
     [{reply,From,OldLockButton}]}.
```

```
do_lock() ->
    io:format("Locked~n", []).
do_unlock() ->
    io:format("Open~n", []).

terminate(_Reason, State, _Data) ->
    State =/= locked andalso do_lock(),
    ok.
```

10.3.26 Hibernation

If you have many servers in one node and they have some state(s) in their lifetime in which the servers can be expected to idle for a while, and the amount of heap memory all these servers need is a problem, then the memory footprint of a server can be mimimized by hibernating it through `proc_lib:hibernate/3`.

Note:

It is rather costly to hibernate a process; see `erlang:hibernate/3`. It is not something you want to do after every event.

We can in this example hibernate in the `{open,_}` state, because what normally occurs in that state is that the state time-out after a while triggers a transition to `{locked,_}`:

```
...
%%
%% State: open
handle_event(enter, _OldState, {open,_}, _Data) ->
    do_unlock(),
    {keep_state_and_data,
     [{state_timeout,10000,lock}, % Time in milliseconds
      hibernate]};
...
```

The atom `hibernate` in the action list on the last line when entering the `{open,_}` state is the only change. If any event arrives in the `{open,_},` state, we do not bother to rehibernate, so the server stays awake after any event.

To change that we would need to insert action `hibernate` in more places. For example, the state-independent `set_lock_button` operation would have to use `hibernate` but only in the `{open,_}` state, which would clutter the code.

Another not uncommon scenario is to use the event time-out to trigger hibernation after a certain time of inactivity. There is also a server start option `{hibernate_after, Timeout}` for `start/3,4`, `start_link/3,4` or `enter_loop/4,5,6` that may be used to automatically hibernate the server.

This particular server probably does not use heap memory worth hibernating for. To gain anything from hibernation, your server would have to produce non-insignificant garbage during callback execution, for which this example server can serve as a bad example.

10.4 gen_event Behaviour

This section is to be read with the `gen_event(3)` manual page in STDLIB, where all interface functions and callback functions are described in detail.

10.4.1 Event Handling Principles

In OTP, an **event manager** is a named object to which events can be sent. An **event** can be, for example, an error, an alarm, or some information that is to be logged.

In the event manager, zero, one, or many **event handlers** are installed. When the event manager is notified about an event, the event is 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 are to be saved to a file as well, the user adds another event handler that does this. When logging to the 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` is the internal state of that event handler.

10.4.2 Example

The callback module for the event handler writing error messages to the terminal can look as follows:

```
-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 can look as follows:

```
-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.

10.4.3 Starting an Event Manager

To start an event manager for handling errors, as described in the previous example, 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. The event manager is then locally registered as `error_man`.

If the name is omitted, the event manager is not registered. Instead its pid must be used. The name can 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, that is, started by a supervisor. There is another function, `gen_event:start`, to start a standalone event manager, that is, an event manager that is not part of a supervision tree.

10.4.4 Adding an Event Handler

The following example shows how to start an event manager and add an event handler to it by using the shell:

```
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 calls the callback function `terminal_logger:init([])`, where the argument `[]` is the third argument to `add_handler`. `init` is expected to return `{ok, State}`, where `State` is the internal state of the event handler.

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

Here, `init` does not need any input data and ignores its argument. For `terminal_logger`, the internal state is not used. For `file_logger`, the internal state is used to save the open file descriptor.

```
init(File) ->
    {ok, Fd} = file:open(File, read),
    {ok, Fd}.
```

10.4.5 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}.
```

10.4.6 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 calls the callback function `terminal_logger:terminate([], State)`, where the argument `[]` is the third argument to `delete_handler`. `terminate` is to 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` must be closed:

```
terminate(_Args, Fd) ->
    file:close(Fd).
```

10.4.7 Stopping

When an event manager is stopped, it gives 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 is automatically terminated by its supervisor. Exactly how this is done is defined by a shutdown strategy set in the supervisor.

Standalone Event Managers

An event manager can also be stopped by calling:

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

10.4.8 Handling Other Messages

If the `gen_event` is to be able to receive other messages than events, the callback function `handle_info(Info, State)` must be implemented to handle them. Examples of other messages are exit messages, if the `gen_event` is linked to other processes (than the supervisor, for example via `add_sup_handler`) and trapping exit signals.

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

The `code_change` method must also be implemented.

```
code_change(OldVsn, State, Extra) ->
    ..code to convert state (and more) during code change
    {ok, NewState}
```

10.5 Supervisor Behaviour

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

10.5.1 Supervision Principles

A supervisor is responsible for starting, stopping, and monitoring its child processes. The basic idea of a supervisor is that it is to keep its child processes alive by restarting them when necessary.

Which child processes to start and monitor is specified by a list of child specifications. The child processes are started in the order specified by this list, and terminated in the reversed order.

10.5.2 Example

The callback module for a supervisor starting the server from gen_server Behaviour can look as follows:

```
-module(ch_sup).
-behaviour(supervisor).

-export([start_link/0]).
-export([init/1]).

start_link() ->
    supervisor:start_link(ch_sup, []).

init(_Args) ->
    SupFlags = #{strategy => one_for_one, intensity => 1, period => 5},
    ChildSpecs = [#{id => ch3,
                    start => {ch3, start_link, []},
                    restart => permanent,
                    shutdown => brutal_kill,
                    type => worker,
                    modules => [ch3]}],
    {ok, {SupFlags, ChildSpecs}}.
```

The `SupFlags` variable in the return value from `init/1` represents the supervisor flags.

The `ChildSpecs` variable in the return value from `init/1` is a list of child specifications.

10.5.3 Supervisor Flags

This is the type definition for the supervisor flags:

```
sup_flags() = #{strategy => strategy(),           % optional
                intensity => non_neg_integer(),   % optional
                period => pos_integer(),          % optional
                auto_shutdown => auto_shutdown()} % optional
    strategy() = one_for_all
               | one_for_one
               | rest_for_one
               | simple_one_for_one
    auto_shutdown() = never
                    | any_significant
                    | all_significant
```

- `strategy` specifies the restart strategy.
- `intensity` and `period` specify the maximum restart intensity.
- `auto_shutdown` specifies if and when a supervisor should automatically shut itself down.

10.5.4 Restart Strategy

The restart strategy is specified by the `strategy` key in the supervisor flags map returned by the callback function `init`:

```
SupFlags = #{strategy => Strategy, ...}
```

The `strategy` key is optional in this map. If it is not given, it defaults to `one_for_one`.

> **Note:**
>
> For simplicity, the diagrams shown in this section display a setup where all the depicted children are assumed to have a restart type of `permanent`.

one_for_one

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

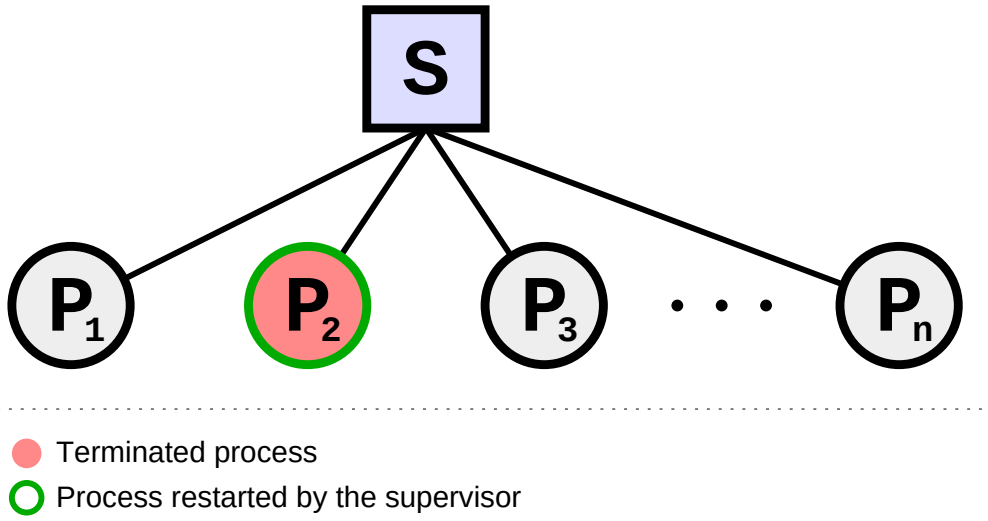


Figure 5.1: One_For_One Supervision

one_for_all

If a child process terminates, all other child processes are terminated, and then all child processes, including the terminated one, are restarted.

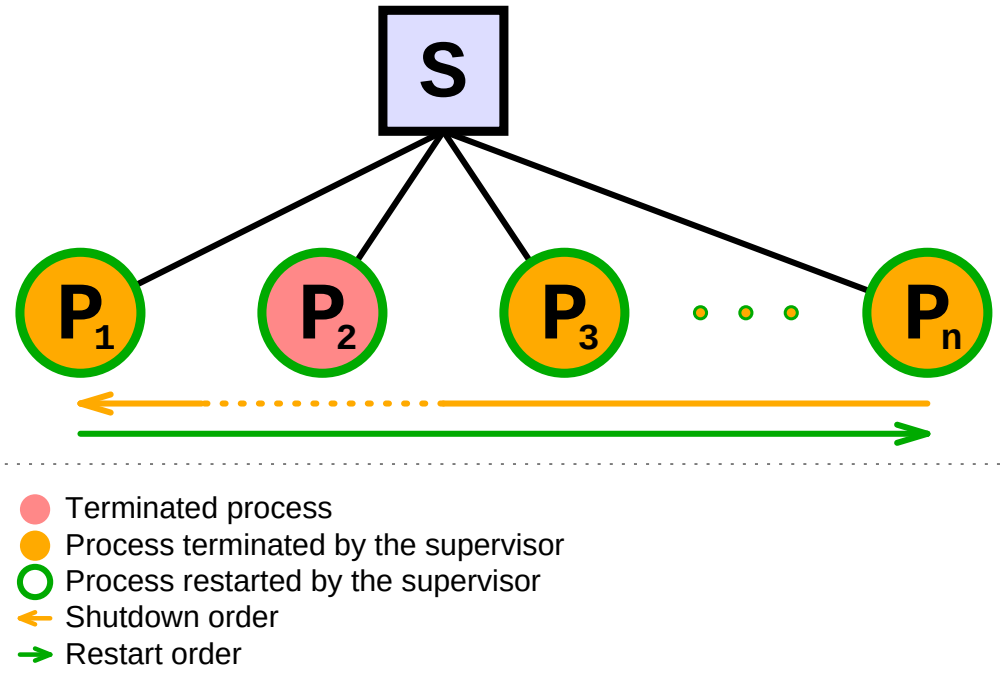


Figure 5.2: One_For_All Supervision

rest_for_one

If a child process terminates, the rest of the child processes (that is, 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.

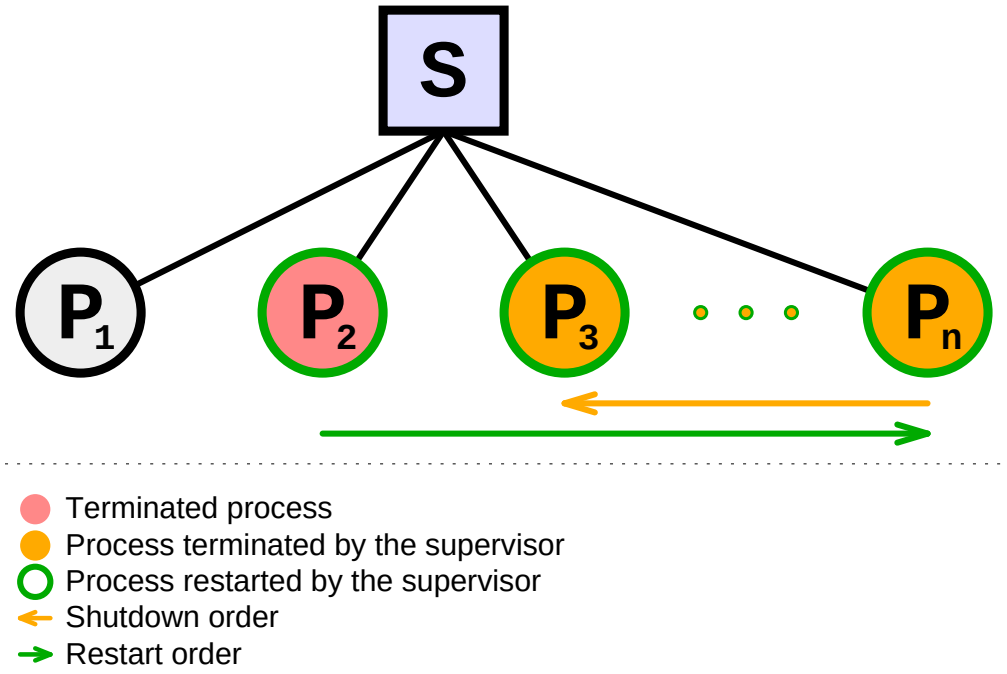


Figure 5.3: Rest_For_One Supervision

simple_one_for_one

See simple-one-for-one supervisors.

10.5.5 Maximum Restart Intensity

The supervisors have a built-in mechanism to limit the number of restarts which can occur in a given time interval. This is specified by the two keys `intensity` and `period` in the supervisor flags map returned by the callback function `init`:

```
SupFlags = #{intensity => MaxR, period => MaxT, ...}
```

If more than `MaxR` number of restarts occur in the last `MaxT` seconds, the supervisor terminates all the child processes and then itself. The termination reason for the supervisor itself in that case will be `shutdown`.

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.

The keys `intensity` and `period` are optional in the supervisor flags map. If they are not given, they default to `1` and `5`, respectively.

Tuning the intensity and period

The default values are 1 restart per 5 seconds. This was chosen to be safe for most systems, even with deep supervision hierarchies, but you will probably want to tune the settings for your particular use case.

First, the intensity decides how big bursts of restarts you want to tolerate. For example, you might want to accept a burst of at most 5 or 10 attempts, even within the same second, if it results in a successful restart.

Second, you need to consider the sustained failure rate, if crashes keep happening but not often enough to make the supervisor give up. If you set intensity to 10 and set the period as low as 1, the supervisor will allow child processes to keep restarting up to 10 times per second, forever, filling your logs with crash reports until someone intervenes manually.

You should therefore set the period to be long enough that you can accept that the supervisor keeps going at that rate. For example, if you have picked an intensity value of 5, then setting the period to 30 seconds will give you at most one restart per 6 seconds for any longer period of time, which means that your logs won't fill up too quickly, and you will have a chance to observe the failures and apply a fix.

These choices depend a lot on your problem domain. If you don't have real time monitoring and ability to fix problems quickly, for example in an embedded system, you might want to accept at most one restart per minute before the supervisor should give up and escalate to the next level to try to clear the error automatically. On the other hand, if it is more important that you keep trying even at a high failure rate, you might want a sustained rate of as much as 1-2 restarts per second.

Avoiding common mistakes:

- Do not forget to consider the burst rate. If you set intensity to 1 and period to 6, it gives the same sustained error rate as 5/30 or 10/60, but will not allow even 2 restart attempts in quick succession. This is probably not what you wanted.
- Do not set the period to a very high value if you want to tolerate bursts. If you set intensity to 5 and period to 3600 (one hour), the supervisor will allow a short burst of 5 restarts, but then gives up if it sees another single restart almost an hour later. You probably want to regard those crashes as separate incidents, so setting the period to 5 or 10 minutes will be more reasonable.
- If your application has multiple levels of supervision, then do not simply set the restart intensities to the same values on all levels. Keep in mind that the total number of restarts (before the top level supervisor gives up and terminates the application) will be the product of the intensity values of all the supervisors above the failing child process.

  For example, if the top level allows 10 restarts, and the next level also allows 10, a crashing child below that level will be restarted 100 times, which is probably excessive. Allowing at most 3 restarts for the top level supervisor might be a better choice in this case.

10.5.6 Automatic Shutdown

A supervisor can be configured to automatically shut itself down when significant children terminate.

This is useful when a supervisor represents a work unit of cooperating children, as opposed to independent workers. When the work unit has finished its work, that is, when any or all significant child processes have terminated, the supervisor should then shut down by terminating all remaining child processes in reverse start order according to the respective shutdown specifications, and then itself.

Automatic shutdown is specified by the `auto_shutdown` key in the supervisor flags map returned by the callback function `init`:

```
SupFlags = #{auto_shutdown => AutoShutdown, ...}
```

The `auto_shutdown` key is optional in this map. If it is not given, it defaults to `never`.

Note:

The automatic shutdown facility only applies when significant children terminate by themselves, that is, when their termination was not caused by means of the supervisor. Specifically, neither the termination of a child as a consequence of a sibling's death in the `one_for_all` or `rest_for_one` strategies nor the manual termination of a child by means of `supervisor:terminate_child/2` will trigger an automatic shutdown.

never

Automatic shutdown is disabled.

In this mode, significant children are not accepted. If the child specs returned from `init` contains significant children, the supervisor will refuse to start. Attempts to start significant children dynamically will be rejected.

This is the default setting.

any_significant

The supervisor will automatically shut itself down when **any** significant child terminates, that is, when a transient significant child terminates normally or when a temporary significant child terminates normally or abnormally.

all_significant

The supervisor will automatically shut itself down when **all** significant children have terminated, that is, when the **last active** significant child terminates. The same rules as for `any_significant` apply.

Warning:

The automatic shutdown feature appeared in OTP 24.0, but applications using this feature will also compile and run with older OTP versions.

However, such applications, when compiled with an OTP version that predates the appearance of the automatic shutdown feature, will leak processes because the automatic shutdowns they rely on will not happen.

It is up to implementors to take proper precautions if they expect that their applications may be compiled with older OTP versions.

Warning:

Top supervisors of Applications should not be configured for automatic shutdown, because when the top supervisor exits, the application terminates. If the application is `permanent`, all other applications and the runtime system are terminated, also.

Warning:

Supervisors configured for automatic shutdown should not be made permanent children of their respective parent supervisors, as they would be restarted immediately after having automatically shut down, only to shut down automatically again after a while, and may thus exhaust the Maximum Restart Intensity of the parent supervisor.

10.5.7 Child Specification

The type definition for a child specification is as follows:

```
child_spec() = #{id => child_id(),             % mandatory
                 start => mfargs(),            % mandatory
                 restart => restart(),         % optional
                 significant => significant(), % optional
                 shutdown => shutdown(),       % optional
                 type => worker(),             % optional
                 modules => modules()}         % optional
    child_id() = term()
    mfargs() = {M :: module(), F :: atom(), A :: [term()]}
    modules() = [module()] | dynamic
    restart() = permanent | transient | temporary
    significant() = boolean()
    shutdown() = brutal_kill | timeout()
    worker() = worker | supervisor
```

- `id` is used to identify the child specification internally by the supervisor.

  The `id` key is mandatory.

  Note that this identifier occasionally has been called "name". As far as possible, the terms "identifier" or "id" are now used but in order to keep backwards compatibility, some occurrences of "name" can still be found, for example in error messages.
- `start` defines the function call used to start the child process. It is a module-function-arguments tuple used as `apply(M, F, A)`.

  It is to be (or result in) a call to any of the following:

  - `supervisor:start_link`
  - `gen_server:start_link`
  - `gen_statem:start_link`
  - `gen_event:start_link`
  - A function compliant with these functions. For details, see the `supervisor(3)` manual page.

  The `start` key is mandatory.
- `restart` defines when a terminated child process is to be restarted.

  - A `permanent` child process is always restarted.
  - A `temporary` child process is never restarted (not even when the supervisor restart strategy is `rest_for_one` or `one_for_all` and a sibling death causes the temporary process to be terminated).
  - A `transient` child process is restarted only if it terminates abnormally, that is, with an exit reason other than `normal`, `shutdown`, or `{shutdown,Term}`.

  The `restart` key is optional. If it is not given, the default value `permanent` will be used.
- `significant` defines if a child is considered significant for automatic self-shutdown of the supervisor.

  It is invalid to set this option to `true` for a child with restart type `permanent` or in a supervisor with auto_shutdown set to `never`.
- `shutdown` defines how a child process is to be terminated.

  - `brutal_kill` means that the child process is unconditionally terminated using `exit(Child, kill)`.
  - An integer time-out 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 must be set to `infinity` to give the subtree enough time to shut down. It is also allowed to set it to `infinity`, if the child process is a worker. See the warning below:

> **Warning:**
>
> Setting the shutdown time to anything other than `infinity` for a child of type `supervisor` can cause a race condition where the child in question unlinks its own children, but fails to terminate them before it is killed.
>
> Be careful when setting the shutdown time to `infinity` when the child process is a worker. Because, in this situation, the termination of the supervision tree depends on the child process; it must be implemented in a safe way and its cleanup procedure must always return.

The `shutdown` key is optional. If it is not given, and the child is of type `worker`, the default value `5000` will be used; if the child is of type `supervisor`, the default value `infinity` will be used.

- `type` specifies if the child process is a supervisor or a worker.

  The `type` key is optional. If it is not given, the default value `worker` will be used.

- `modules` are to 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, gen_statem. If the child process is a gen_event, the value shall be `dynamic`.

  This information is used by the release handler during upgrades and downgrades, see Release Handling.

  The `modules` key is optional. If it is not given, it defaults to `[M]`, where `M` comes from the child's start `{M,F,A}`.

Example: The child specification to start the server `ch3` in the previous example look as follows:

```
#{id => ch3,
  start => {ch3, start_link, []},
  restart => permanent,
  shutdown => brutal_kill,
  type => worker,
  modules => [ch3]}
```

or simplified, relying on the default values:

```
#{id => ch3,
  start => {ch3, start_link, []}
  shutdown => brutal_kill}
```

Example: A child specification to start the event manager from the chapter about gen_event:

```
#{id => error_man,
  start => {gen_event, start_link, [{local, error_man}]},
  modules => dynamic}
```

Both server and event manager are registered processes which can be expected to be always accessible. 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` is sufficient. `error_man` can need some time for the event handlers to clean up, thus the shutdown time is set to 5000 ms (which is the default value).

Example: A child specification to start another supervisor:

```
#{id => sup,
  start => {sup, start_link, []},
  restart => transient,
  type => supervisor} % will cause default shutdown=>infinity
```

10.5.8 Starting a Supervisor

In the previous example, the supervisor is started by calling `ch_sup:start_link()`:

```
start_link() ->
    supervisor:start_link(ch_sup, []).
```

`ch_sup:start_link` calls function `supervisor:start_link/2`, which 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 that 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` shall return `{ok, {SupFlags, ChildSpecs}}`:

```
init(_Args) ->
    SupFlags = #{},
    ChildSpecs = [#{id => ch3,
                    start => {ch3, start_link, []},
                    shutdown => brutal_kill}],
    {ok, {SupFlags, ChildSpecs}}.
```

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`.

`supervisor:start_link` is synchronous. It does not return until all child processes have been started.

10.5.9 Adding a Child Process

In addition to the static supervision tree, dynamic child processes can be added to an existing supervisor with the following call:

```
supervisor:start_child(Sup, ChildSpec)
```

`Sup` is the pid, or name, of the supervisor. `ChildSpec` is a child specification.

Child processes added using `start_child/2` behave in the same way as the other child processes, with the an important exception: if a supervisor dies and is recreated, then all child processes that were dynamically added to the supervisor are lost.

10.5.10 Stopping a Child Process

Any child process, static or dynamic, can be stopped in accordance with the shutdown specification:

```
supervisor:terminate_child(Sup, Id)
```

Stopping a significant child of a supervisor configured for automatic shutdown will not trigger an automatic shutdown.

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 value associated with the `id` key in the child specification.

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

10.5.11 Simplified 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.

The following is an 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) ->
    SupFlags = #{strategy => simple_one_for_one,
                 intensity => 0,
                 period => 1},
    ChildSpecs = [#{id => call,
                    start => {call, start_link, []},
                    shutdown => brutal_kill}],
    {ok, {SupFlags, ChildSpecs}}.
```

When started, the supervisor does 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 are added to the list of arguments specified in the child specification. If the start function is specified as `{M, F, A}`, 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])
```

The result is that the child process is started by calling `apply(call, start_link, []++[id1])`, or actually:

```
call:start_link(id1)
```

A child under a `simple_one_for_one` supervisor can be terminated with the following:

```
supervisor:terminate_child(Sup, Pid)
```

`Sup` is the pid, or name, of the supervisor and `Pid` is the pid of the child.

Because a `simple_one_for_one` supervisor can have many children, it shuts them all down asynchronously. This means that the children will do their cleanup in parallel and therefore the order in which they are stopped is not defined.

10.5.12 Stopping

Since the supervisor is part of a supervision tree, it is automatically terminated by its supervisor. When asked to shut down, it terminates all child processes in reversed start order according to the respective shutdown specifications, and then terminates itself.

If the supervisor is configured for automatic shutdown on termination of any or all significant children, it will shut down itself when any or the last active significant child terminates, respectively. The shutdown itself follows the same procedure as described above, that is, the supervisor terminates all remaining child processes in reversed start order, and then terminates itself.

Manual stopping versus Automatic Shutdown

For several reasons, a supervisor should not be stopped manually via `supervisor:terminate_child/2` from a child located in its own tree.

- The child process will have to know the pids or registered names not only of the supervisor it wants to stop, but also that of the supervisor's parent supervisor, in order to tell the parent supervisor to stop the supervisor it wants to stop. This can make restructuring a supervision tree difficult.
- `supervisor:terminate_child/2` is a blocking call that will only return after the parent supervisor has finished the shutdown of the supervisor that should be stopped. Unless the call is made from a spawned process, this will result in a deadlock, as the supervisor waits for the child to exit as part of its shutdown procedure, whereas the child waits for the supervisor to shut down. If the child is trapping exits, this deadlock will last until the shutdown timeout for the child expires.
- When a supervisor is stopping a child, it will wait for the shutdown to complete before accepting other calls, that is, the supervisor will be unresponsive until then. If the termination takes some time to complete, especially when the considerations outlined in the previous point were not taken into account carefully, said supervisor might become unresponsive for a long time.

Instead, it is generally a better approach to rely on Automatic Shutdown.

- A child process does not need to know anything about its supervisor and its respective parent, not even that it is part of a supervision tree in the first place. It is instead only the supervisor which hosts the child who must know which of its children are significant ones, and when to shut itself down.
- A child process does not need to do anything special to shut down the work unit it is part of. All it needs to do is terminate normally when it has finished the task it was started for.
- A supervisor that is automatically shutting itself down will perform the required shutdown steps fully independent of its parent supervisor. The parent supervisor will only notice that its child supervisor has terminated in the end. As the parent supervisor is not involved in the shutdown process, it will not be blocked.

10.6 sys and proc_lib

The `sys` module has functions for simple debugging of processes implemented using behaviours. It also has functions that, together with functions in the `proc_lib` module, can be used to implement a **special process** that complies to the OTP design principles without using a standard behaviour. These functions can also be used to implement user-defined (non-standard) behaviours.

Both `sys` and `proc_lib` belong to the STDLIB application.

10.6.1 Simple Debugging

The `sys` module has functions for simple debugging of processes implemented using behaviours. The `code_lock` example from gen_statem Behaviour is used to illustrate this:

```
Erlang/OTP 20 [DEVELOPMENT] [erts-9.0] [source-5ace45e] [64-bit] [smp:8:8] [ds:8:8:10] [async-threads:10] [hipe

Eshell V9.0  (abort with ^G)
1>  code_lock:start_link([1,2,3,4]).
Lock
{ok,<0.63.0>}
2> sys:statistics(code_lock, true).
ok
3>  sys:trace(code_lock, true).
ok
4>  code_lock:button(1).
*DBG* code_lock receive cast {button,1} in state locked
ok
*DBG* code_lock consume cast {button,1} in state locked
5>  code_lock:button(2).
*DBG* code_lock receive cast {button,2} in state locked
ok
*DBG* code_lock consume cast {button,2} in state locked
6>  code_lock:button(3).
*DBG* code_lock receive cast {button,3} in state locked
ok
*DBG* code_lock consume cast {button,3} in state locked
7>  code_lock:button(4).
*DBG* code_lock receive cast {button,4} in state locked
ok
Unlock
*DBG* code_lock consume cast {button,4} in state locked
*DBG* code_lock receive state_timeout lock in state open
Lock
*DBG* code_lock consume state_timeout lock in state open
8> sys:statistics(code_lock, get).
{ok,[{start_time,{{2017,4,21},{16,8,7}}},
     {current_time,{{2017,4,21},{16,9,42}}},
     {reductions,2973},
     {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,<0.63.0>,
        {module,gen_statem},
        [[{'$initial_call',{code_lock,init,1}},
          {'$ancestors',[<0.61.0>]}],
         running,<0.61.0>,[],
         [{header,"Status for state machine code_lock"},
          {data,[{"Status",running},
                 {"Parent",<0.61.0>},
                 {"Logged Events",[]},
                 {"Postponed",[]}]},
          {data,[{"State",
                  {locked,#{code => [1,2,3,4],remaining => [1,2,3,4]}}}]}]]}
```

10.6.2 Special Processes

This section describes how to write a process that complies to the OTP design principles, without using a standard behaviour. Such a process is to:

- Be started in a way that makes the process fit into a supervision tree
- Support the `sys` debug facilities

- Take care of system messages.

System messages are messages with a 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 Overview, 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,
         system_get_state/1, system_replace_state/2]).

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, fun ch4:write_debug/3,
                                    ch4, {in, alloc, From}),
            {Ch, Chs2} = alloc(Chs),
            From ! {ch4, Ch},
            Deb3 = sys:handle_debug(Deb2, fun ch4:write_debug/3,
                                    ch4, {out, {ch4, Ch}, From}),
            loop(Chs2, Parent, Deb3);
        {free, Ch} ->
            Deb2 = sys:handle_debug(Deb, fun ch4:write_debug/3,
                                    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).

system_get_state(Chs) ->
    {ok, Chs}.

system_replace_state(StateFun, Chs) ->
    NChs = StateFun(Chs),
    {ok, NChs, NChs}.

write_debug(Dev, Event, Name) ->
```

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

Example on how the simple debugging functions in the `sys` module can also be used for `ch4`:

```
% 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 is to be used to start the process. Several functions are available, 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 stores information (for example, about the ancestors and initial call) that is needed for a process in a supervision tree.

If the process terminates with another reason than `normal` or `shutdown`, a crash report is generated. For more information about the crash report, see the SASL User's Guide.

In the example, synchronous start is used. The process starts 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, spawns, and links to a new process. The new process starts by executing the given function, here `ch4:init(Pid)`, where `Pid` is the pid (`self()`) of the first process, which is the parent process.

All initialization, including name registration, is done in `init`. 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 facilities in `sys`, a **debug structure** is needed. The `Deb` term 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. For information about the possible options, see the `sys(3)` manual page in STDLIB.

Then, for each **system event** to be logged or traced, the following function is to be called.

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

Here:

- `Deb` is the debug structure.
- `Func` is a fun specifying a (user-defined) function used to format trace output. For each system event, the format function is called as `Func(Dev, Event, Info)`, where:
  - `Dev` is the I/O device to which the output is to be printed. See the `io(3)` manual page in STDLIB.
  - `Event` and `Info` are passed as is from `handle_debug`.
- `Info` is used to pass more 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 is to be represented. Typically at least incoming and outgoing messages are considered system events and represented by the tuples `{in,Msg[,From]}` and `{out,Msg,To[,State]}`, 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, fun ch4:write_debug/3,
                                    ch4, {in, alloc, From}),
            {Ch, Chs2} = alloc(Chs),
            From ! {ch4, Ch},
            Deb3 = sys:handle_debug(Deb2, fun ch4:write_debug/3,
                                    ch4, {out, {ch4, Ch}, From}),
            loop(Chs2, Parent, Deb3);
        {free, Ch} ->
            Deb2 = sys:handle_debug(Deb, fun ch4:write_debug/3,
                                    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 is to be called:

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

This function does not return. It handles the system message and then either calls the following if process execution is to continue:

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

Or calls the following if the process is to terminate:

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

A process in a supervision tree is expected to terminate with the same reason as its parent.

- `Request` and `From` are to 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`/ `system_get_state`/`system_replace_state`.

If the process is to return its state, `handle_system_msg` calls:

```
Module:system_get_state(State)
```

If the process is to replace its state using the fun `StateFun`, `handle_system_msg` calls:

```
Module:system_replace_state(StateFun, State)
```

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).

system_get_state(Chs) ->
    {ok, Chs, Chs}.

system_replace_state(StateFun, Chs) ->
    NChs = StateFun(Chs),
    {ok, NChs, NChs}.
```

If the special process is set to trap exits and 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.
```

10.6.3 User-Defined Behaviours

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

If the compiler is to warn for missing callback functions, as it does for the OTP behaviours, add `-callback` attributes in the behaviour module to describe the expected callbacks:

```
-callback Name1(Arg1_1, Arg1_2, ..., Arg1_N1) -> Res1.
-callback Name2(Arg2_1, Arg2_2, ..., Arg2_N2) -> Res2.
...
-callback NameM(ArgM_1, ArgM_2, ..., ArgM_NM) -> ResM.
```

`NameX` are the names of the expected callbacks. `ArgX_Y` and `ResX` are types as they are described in Types and Function Specifications. The whole syntax of the `-spec` attribute is supported by the `-callback` attribute.

Callback functions that are optional for the user of the behaviour to implement are specified by use of the `-optional_callbacks` attribute:

```
-optional_callbacks([OptName1/OptArity1, ..., OptNameK/OptArityK]).
```

where each `OptName/OptArity` specifies the name and arity of a callback function. Note that the `-optional_callbacks` attribute is to be used together with the `-callback` attribute; it cannot be combined with the `behaviour_info()` function described below.

Tools that need to know about optional callback functions can call `Behaviour:behaviour_info(optional_callbacks)` to get a list of all optional callback functions.

Note:

We recommend using the `-callback` attribute rather than the `behaviour_info()` function. The reason is that the extra type information can be used by tools to produce documentation or find discrepancies.

As an alternative to the `-callback` and `-optional_callbacks` attributes you may directly implement and export `behaviour_info()`:

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

where each `{Name, Arity}` specifies the name and arity of a callback function. This function is otherwise automatically generated by the compiler using the `-callback` attributes.

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

Example:

```
%% User-defined behaviour module
-module(simple_server).
-export([start_link/2, init/3, ...]).

-callback init(State :: term()) -> 'ok'.
-callback handle_req(Req :: term(), State :: term()) -> {'ok', Reply :: term()}.
-callback terminate() -> 'ok'.
-callback format_state(State :: term()) -> term().

-optional_callbacks([format_state/1]).

%% Alternatively you may define:
%%
%% -export([behaviour_info/1]).
%% behaviour_info(callbacks) ->
%%     [{init,1},
%%      {handle_req,2},
%%      {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/1, handle_req/2, terminate/0]).

...
```

The contracts specified with `-callback` attributes in behaviour modules can be further refined by adding `-spec` attributes in callback modules. This can be useful as `-callback` contracts are usually generic. The same callback module with contracts for the callbacks:

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

-export([init/1, handle_req/2, terminate/0]).

-record(state, {field1 :: [atom()], field2 :: integer()}).

-type state()   :: #state{}.
-type request() :: {'store', term(), term()};
                   {'lookup', term()}.

...

-spec handle_req(request(), state()) -> {'ok', term()}.

...
```

Each `-spec` contract is to be a subtype of the respective `-callback` contract.

10.7 Applications

This section is to be read with the `app(4)` and `application(3)` manual pages in Kernel.

10.7.1 Application Concept

When you have written code implementing some specific functionality you 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 also be reused in other systems.

To do this, create an application callback module, and describe how the application is to be started and stopped.

Then, an **application specification** is needed, which is put in an application resource file. Among other things, this file specifies which modules the application consists of and the name of the callback module.

If you use `systools`, the Erlang/OTP tools for packaging code (see Releases), the code for each application is placed in a separate directory following a pre-defined directory structure.

10.7.2 Application Callback Module

How to start and stop the code for the application, that is, 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 is to create the supervision tree by starting the top supervisor. It is expected to return the pid of the top supervisor and an optional term, `State`, which defaults to `[]`. This term is passed as is to `stop`.
- `StartType` is usually the atom `normal`. It has other values only in the case of a takeover or failover, see Distributed Applications.

- `StartArgs` is defined by the key `mod` in the application resource file.
- `stop/1` is called **after** the application has been stopped and is to do any necessary cleaning up. The actual stopping of the application, that is, the shutdown of the supervision tree, is handled automatically as described in Starting and Stopping Applications.

Example of an application callback module for packaging the supervision tree from Supervisor Behaviour:

```
-module(ch_app).
-behaviour(application).

-export([start/2, stop/1]).

start(_Type, _Args) ->
    ch_sup:start_link().

stop(_State) ->
    ok.
```

A library application that cannot be started or stopped, does not need any application callback module.

10.7.3 Application Resource File

To define an application, an **application specification** is created, which is put in an **application resource file**, or in short an `.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 defines 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 as follows:

```
{application, libapp, []}.
```

The contents of a minimal `.app` file `ch_app.app` for a supervision tree application like `ch_app` looks as follows:

```
{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 the following is called when the application is to be started:

```
ch_app:start(normal, [])
```

The following is called when the application is stopped.

```
ch_app:stop([])
```

When using `systools`, the Erlang/OTP tools for packaging code (see Section Releases), the keys `description`, `vsn`, `modules`, `registered`, and `applications` are also to 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 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 that must be started before this application is started. `systools` uses this list to generate correct boot scripts. Defaults to `[]`. Notice that all applications have dependencies to at least Kernel and STDLIB.

Note:

For details about the syntax and contents of the application resource file, see the app manual page in Kernel.

10.7.4 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 can be useful to know, even if `systools` is not used, since Erlang/OTP is packaged according to the OTP principles and thus comes with a specific directory structure. The code server (see the `code(3)` manual page in Kernel) automatically uses code from the directory with the highest version number, if more than one version of an application is present.

Directory Structure Guidelines for a Development Environment

Any directory structure for development will suffice as long as the released directory structure adheres to the description below, but it is encouraged that the same directory structure also be used in a development environment. The version number should be omitted from the application directory name since this is an artifact of the release step.

Some sub-directories are **required**. Some sub-directories are **optional**, meaning that it should only be used if the application itself requires it. Finally, some sub-directories are **recommended**, meaning it is encouraged that it is used and used as described here. For example, both documentation and tests are encouraged to exist in an application for it to be deemed a proper OTP application.

```
— ${application}
  ├── doc
  │   ├── internal
  │   ├── examples
  │   └── src
  ├── include
  ├── priv
  ├── src
  │   └── ${application}.app.src
  └── test
```

- `src` - Required. Contains the Erlang source code, the source of the `.app` file and internal include files used by the application itself. Additional sub-directories within `src` can be used as namespaces to organize source files. These directories should never be deeper than one level.
- `priv` - Optional. Used for application specific files.
- `include` - Optional. Used for public include files that must be reachable from other applications.
- `doc` - Recommended. Any source documentation should be placed in sub-directories here.
- `doc/internal` - Recommended. Any documentation that describes implementation details about this application, not intended for publication, should be placed here.

- `doc/examples` - Recommended. Source code for examples on how to use this application should be placed here. It is encouraged that examples are sourced to the public documentation from this directory.
- `doc/src` - Recommended. All source files for documentation, such as Markdown, AsciiDoc or XML-files, should be placed here.
- `test` - Recommended. All files regarding tests, such as test suites and test specifications, should be placed here.

Other directories in the development environment may be needed. If source code from languages other than Erlang is used, for instance C-code for NIFs, that code should be placed in a separate directory. By convention it is recommended to prefix such directories with the language name, for example `c_src` for C, `java_src` for Java or `go_src` for Go. Directories with `_src` suffix indicates that it is a part of the application and the compilation step. The final build artifacts should target the `priv/lib` or `priv/bin` directories.

The `priv` directory holds assets that the application needs during runtime. Executables should reside in `priv/bin` and dynamically-linked libraries should reside in `priv/lib`. Other assets are free to reside within the `priv` directory but it is recommended they do so in a structured manner.

Source files from other languages that generate Erlang code, such as ASN.1 or Mibs, should be placed in directories, at the top level or in `src`, with the same name as the source language, for example `asn1` and `mibs`. Build artifacts should be placed in their respective language directory, such as `src` for Erlang code or `java_src` for Java code.

The `.app` file for release may reside in the `ebin`-directory in a development environment but it is encouraged that this is an artifact of the build step. By convention a `.app.src` file is used, which resides in the `src` directory. This file is nearly identical as the `.app` file but certain fields may be replaced during the build step, such as the application version.

Directory names should not be capitalized.

It is encouraged to omit empty directories.

Directory Structure for a Released System

A released application must follow a certain structure.

```
— ${application}-${version}
  ├── bin
  ├── doc
  │   ├── html
  │   ├── man[1-9]
  │   ├── pdf
  │   ├── internal
  │   └── examples
  ├── ebin
  │   └── ${application}.app
  ├── include
  ├── priv
  │   ├── lib
  │   └── bin
  └── src
```

- `src` - Optional. Contains the Erlang source code and internal include files used by the application itself. This directory is no longer required in a released application.
- `ebin` - Required. Contains the Erlang object code, the `beam` files. The `.app` file must also be placed here.
- `priv` - Optional. Used for application specific files. `code:priv_dir/1` is to be used to access this directory.
- `priv/lib` - Recommended. Any shared-object files that are used by the application, such as NIFs or linked-in-drivers, should be placed here.
- `priv/bin` - Recommended. Any executable that is used by the application, such as port-programs, should be placed here.

- `include` - Optional. Used for public include files that must be reachable from other applications.
- `bin` - Optional. Any executable that is a product of the application, such as escripts or shell-scripts, should be placed here.
- `doc` - Optional. Any released documentation should be placed in sub-directories here.
- `doc/man1` - Recommended. Man pages for Application executables.
- `doc/man3` - Recommended. Man pages for module APIs.
- `doc/man6` - Recommended. Man pages for Application overview.
- `doc/html` - Optional. HTML pages for the entire Application.
- `doc/pdf` - Optional. PDF documentation for the entire Application.

The `src` directory could be useful to release for debugging purposes but is not required. The `include` directory should only be released if the applications has public include files.

The only documentation that is recommended to be released in this way are the man pages. HTML and PDF will normally be distributed in some other manner.

It is encouraged to omit empty directories.

10.7.5 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 interacted with through the functions in the module `application`, see the `application(3)` manual page in Kernel. In particular, applications can be loaded, unloaded, started, and stopped.

10.7.6 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.

10.7.7 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 first loads it using `application:load/1`. It checks the value of the `applications` key, to ensure that all applications that are to be started before this application are running.

The application controller then creates an **application master** for the application. The application master becomes the group leader of all the processes in the application. I/O is forwarded to the previous group leader, though, this is just a way to identify processes that belong to the application. Used for example to find itself from any process, or, reciprocally, to kill them all when it terminates.

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 shut down. The top supervisor tells all its child processes to shut down, and so on; 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.

10.7.8 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` is to 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 the `application(3)` manual page in Kernel.

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 that contains configuration parameters for relevant applications:

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

The system configuration is to be called `Name.config` and Erlang is to be started with the command-line argument `-config Name`. For details, see the `config(4)` manual page in Kernel.

Example:

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

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

The value of `file` overrides the value of `file` as defined in the `.app` file:

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

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

If release handling is used, exactly one system configuration file is to be used and that file is to be called `sys.config`.

The values in the `.app` file and 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"}
```

10.7.9 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.

An application can always be stopped explicitly by calling `application:stop/1`. Regardless of the mode, no other applications are affected.

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

10.8 Included Applications

10.8.1 Introduction

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 that is not included by any other application is called a **primary application**.

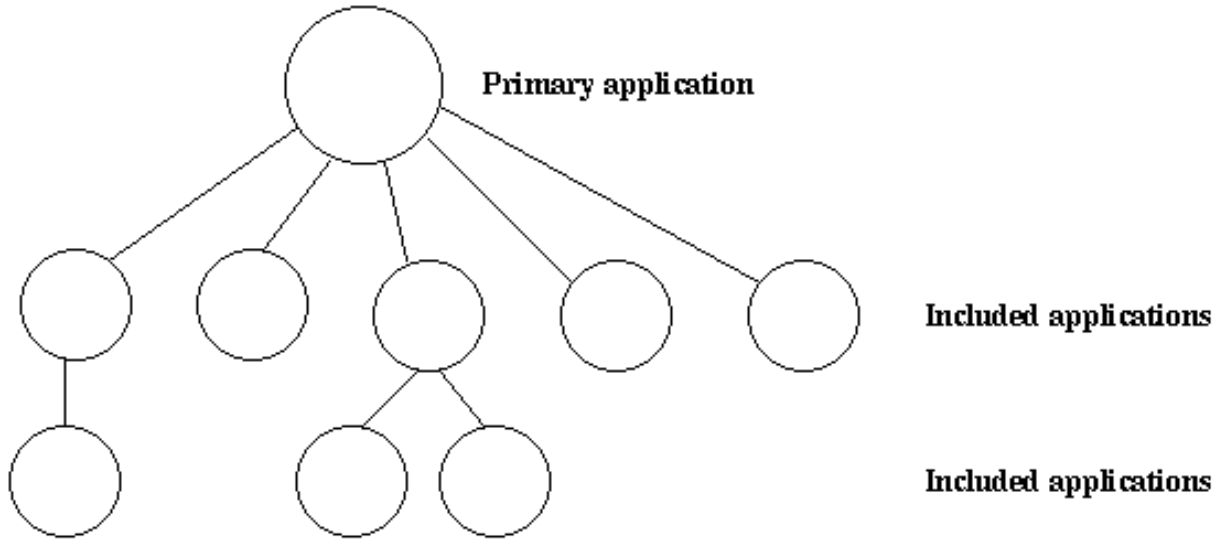


Figure 8.1: Primary Application and Included Applications

The application controller automatically loads any included applications when loading a primary application, but does 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 considers itself belonging to the primary application.

10.8.2 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"}]}
 ]}.
```

10.8.3 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.

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. `StartArgs` is 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, that is:

- The application controller creates an application master for the application
- 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 the primary application, in that order. 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 `{mod, {application_starter, [Module,StartArgs]}}` option 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 calls 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
```

10.9 Distributed Applications

10.9.1 Introduction

In a distributed system with several Erlang nodes, it can be necessary to control applications in a distributed manner. If the node, where a certain application is running, goes down, the application is to be restarted at another node.

Such an application is called a **distributed application**. Notice that it is the control of the application that is distributed. All applications can be distributed in the sense that they, for example, use services on other nodes.

Since a distributed application can 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 `global` or `pg` modules in Kernel can be used for this purpose.

10.9.2 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. Distributed applications are thus 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()` can 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. It defaults to 0.

For distribution of application control to work properly, the nodes where a distributed application can run must contact each other and negotiate where to start the application. This is done using the following configuration parameters in Kernel:

- `sync_nodes_mandatory = [Node]` - Specifies which other nodes must be started (within the time-out specified by `sync_nodes_timeout`).
- `sync_nodes_optional = [Node]` - Specifies which other nodes can be started (within the time-out 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 waits for all nodes specified by `sync_nodes_mandatory` and `sync_nodes_optional` to come up. When all nodes are up, or when all mandatory nodes are up and the time specified by `sync_nodes_timeout` has elapsed, all applications start. If not all mandatory nodes are up, the node terminates.

Example:

An application `myapp` is to run at the node `cp1@cave`. If this node goes down, `myapp` is to be restarted at `cp2@cave` or `cp3@cave`. A system configuration file `cp1.config` for `cp1@cave` can look as follows:

```
[{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 is to 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`. Otherwise the system behaviour is undefined.

10.9.3 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.**

A boot script (see Releases) can be used that automatically starts the application.

The application is started at the first operational node that is listed in the list of nodes in the `distributed` configuration parameter. 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 operational, `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 following figure:



Figure 9.1: Application myapp - Situation 1

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

10.9.4 Failover

If the node where the application is running goes down, the application is restarted (after the specified time-out) at the first operational node that is listed in the list of nodes in the `distributed` configuration parameter. 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)
```

An exception is if the application has the `start_phases` key defined (see Included Applications). The application is then instead started by calling:

```
Module:start({failover, Node}, StartArgs)
```

Here `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`, `myapp` is restarted on `cp2`.

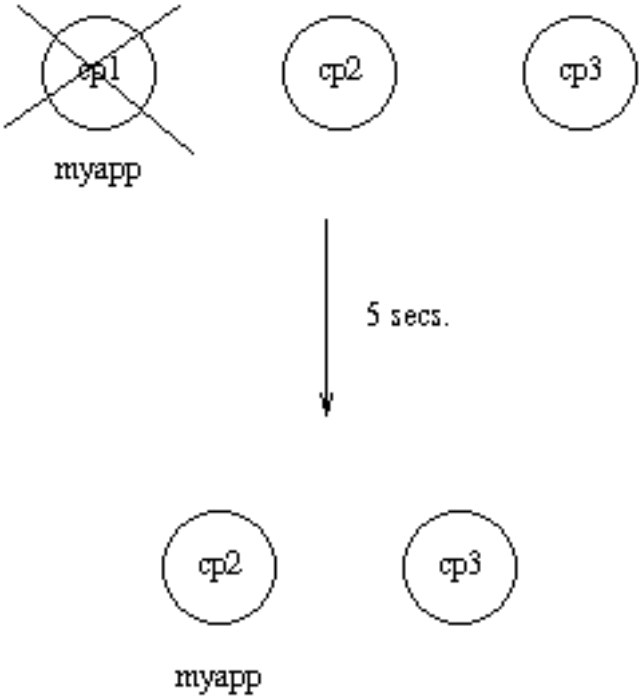


Figure 9.2: Application myapp - Situation 2

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

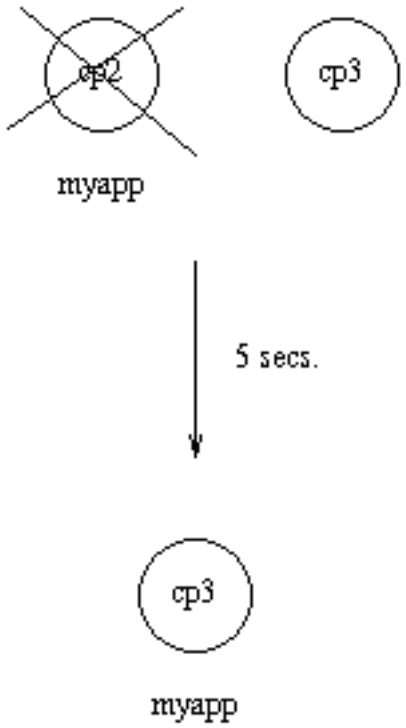


Figure 9.3: Application myapp - Situation 3

10.9.5 Takeover

If a node is started, which has higher priority according to `distributed` than the node where a distributed application is running, the application is 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)
```

Here `Node` is the old node.

Example:

If `myapp` is running at `cp3`, and if `cp2` now restarts, it does not restart `myapp`, as the order between the `cp2` and `cp3` nodes is undefined.

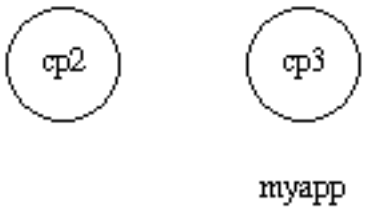


Figure 9.4: Application myapp - Situation 4

However, if `cp1` also restarts, the function `application:takeover/2` moves `myapp` to `cp1`, as `cp1` has a higher priority than `cp3` for this application. In this case, `Module:start({takeover, cp3@cave}, StartArgs)` is executed at `cp1` to start the application.

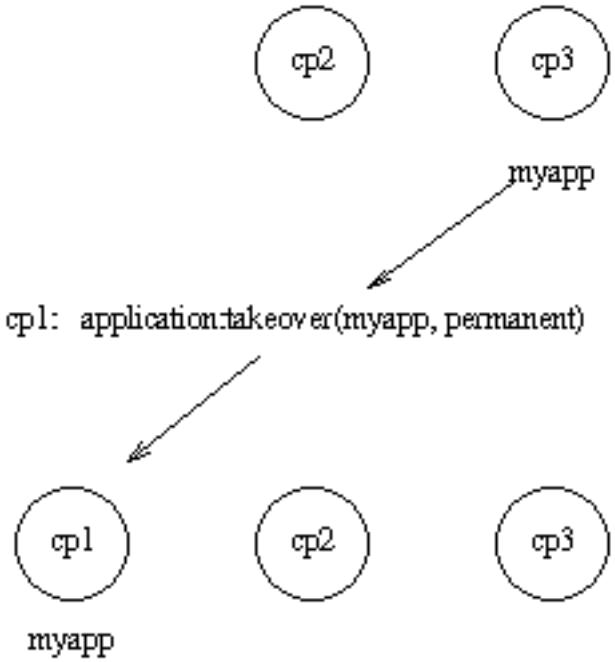


Figure 9.5: Application myapp - Situation 5

10.10 Releases

This section is to be read with the `rel(4)`, `systools(3)`, and `script(4)` manual pages in SASL.

10.10.1 Release Concept

When you have written one or more applications, you might want to create a complete system with these applications and a subset of the Erlang/OTP applications. This is called a **release**.

To do this, create a release resource file that defines which applications are included in the release.

The release resource file is used to generate boot scripts and release packages. A system that is transferred 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.

10.10.2 Release Resource File

To define a release, create a **release resource file**, or in short a `.rel` file. In the file, 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}]}.
```

`Name`, `Vsn`, `EVsn`, and `AppVsn` are strings.

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

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

If the release is to be upgraded, it must also include the SASL application.

Example: A release of `ch_app` from Applications 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`, as these applications are required by `ch_app`. The file is called `ch_rel-1.rel`:

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

10.10.3 Generating Boot Scripts

`systools` in the SASL application includes tools to build and check releases. The functions read the `rel` and `.app` files and perform syntax and dependency checks. The `systools:make_script/1,2` function 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, `ch_rel-1.boot`, used by the runtime system.

- `"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
```

10.10.4 Creating a Release Package

The `systools:make_tar/1,2` function 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
- The `.rel` file
- The object code for all applications, structured according to the application directory structure
- The binary boot script renamed to `start.boot`

```
% 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/A/ch_rel-1.rel
releases/ch_rel-1.rel
```

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`. You do not know where the release package will be installed, so no hard-coded absolute paths are allowed.

The release resource file `mysystem.rel` is duplicated in the tar file. Originally, this file was only stored in the `releases` directory to make it possible for the `release_handler` to extract this file separately. After unpacking the tar file, `release_handler` would automatically copy the file to `releases/FIRST`. However, sometimes the tar file is unpacked without involving the `release_handler` (for example, when unpacking the first target system) and the file is therefore now instead duplicated in the tar file so no manual copying is necessary.

If a `relup` file and/or a system configuration file called `sys.config`, or a `sys.config.src`, is found, these files are also included in the release package. See Release Handling.

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

For information on how to install the first target system, using a release package, see System Principles. For information on how to install a new release package in an existing system, see Release Handling.

10.10.5 Directory Structure

The directory structure for the code installed by the release handler from a release package is as follows:

```
$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` or `sys.config.src`
- `bin` - Top-level Erlang runtime system executables

Applications are not required to be located under directory `$ROOT/lib`. Several installation directories, which contain different parts of a system, can thus exist. For example, the previous example can 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
```

`$SECOND_ROOT` and `$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 disk-less and/or read-only client nodes, a `clients` directory is to be added to the `$ROOT` directory. A read-only node is a node with a read-only file system.

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

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

This structure is to 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 can be divided into one subdirectory per type of Erlang machine. Alternatively, one `$ROOT` can be set up per type of machine. For each type, some of the directories specified for the `$ROOT` directory are to 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`.

10.11 Release Handling

10.11.1 Release Handling Principles

An important feature of the Erlang programming language is the ability to change module code in runtime, **code replacement**, as described in the 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 runtime. This is called **release handling**.

The framework consists of:

- Offline support - `systools` for generating scripts and building release packages
- Online support - `release_handler` for unpacking and installing release packages

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

Release Handling Workflow

Step 1) A release is created as described in Releases.

Step 2) The release is transferred to and installed at target environment. For information of how to install the first target system, see System Principles.

Step 3) Modifications, for example, error corrections, are made to the code in the development environment.

Step 4) 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.

Step 5) For each modified application, an application upgrade file, `.appup`, is created. In this file, it is described how to upgrade and/or downgrade between the old and new version of the application.

Step 6) Based on the `.appup` files, a release upgrade file called `relup`, is created. This file describes how to upgrade and/or downgrade between the old and new version of the entire release.

Step 7) A new release package is made and transferred to the target system.

Step 8) The new release package is unpacked using the release handler.

Step 9) The new version of the release is installed, also using the release handler. This is done by evaluating the instructions in `relup`. Modules can be added, deleted, or reloaded, applications can be started, stopped, or restarted, and so on. In some cases, it is even necessary to restart the entire emulator.

- If the installation fails, the system can be rebooted. The old release version is then automatically used.
- If the installation succeeds, the new version is made the default version, which is to now be used if there is a system reboot.

Release Handling Aspects

Appup Cookbook, contains examples of `.appup` files for typical cases of upgrades/downgrades that are normally easy to handle in runtime. However, many aspects can make release handling complicated, for example:

- Complicated or circular dependencies can make it difficult or even impossible to decide in which order things must be done without risking runtime errors during an upgrade or downgrade. Dependencies can be:
  - Between nodes
  - Between processes
  - Between modules
- During release handling, non-affected processes continue normal execution. This can lead to time-outs 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, can execute old code.

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

10.11.2 Requirements

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

For system reboots to work properly, it is also required that the system is started with heartbeat monitoring, see the `erl(1)` manual page in ERTS and the `heart(3)` manual page in Kernel

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 is fetched from the script when an upgrade or downgrade is performed.

- The system must be configured using 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.

10.11.3 Distributed Systems

If the system consists of several Erlang nodes, each node can 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. A release handling instruction, `sync_nodes`, can be used to synchronize the release handler processes at a number of nodes, see the `appup(4)` manual page in SASL.

10.11.4 Release Handling Instructions

OTP supports a set of **release handling instructions** that are 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`.

Some of the most frequently used instructions are described in this section. The complete list of instructions is included in the `appup(4)` manual page in SASL.

First, some definitions:

- **Residence module** - The module where a process has its tail-recursive loop function(s). If these functions are implemented in several modules, all those modules are residence modules for the process.
- **Functional module** - A module that is not a residence module for any process.

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 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.
- 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 causes behaviour processes to call the callback function `code_change`, passing the term `Extra` and some other information as arguments. See the manual pages for the respective behaviours and Appup Cookbook.

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

When a module is to be updated, the release handler finds which processes that are **using** the module by traversing the supervision tree of each running application and checking all the child specifications:

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

A process uses 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_event`), 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 necessary when running Erlang in embedded mode. It is not strictly required when running Erlang in interactive (default) mode, since the code server then automatically searches for and loads unloaded modules.

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

```
{delete_module, Module}
```

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

Application Instructions

The following is the 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, and then the application is started.

The following is the 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.

The following is the instruction for restarting 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 evaluates `apply(M, F, A)`.

restart_new_emulator (Low-Level)

This instruction is used when changing to a new emulator version, or when any of the core applications Kernel, STDLIB, or SASL is upgraded. If a system reboot is needed for another reason, the `restart_emulator` instruction is to be used instead.

This instruction requires that the system is started with heartbeat monitoring, see the `erl(1)` manual page in ERTS and the `heart(3)` manual page in Kernel.

The `restart_new_emulator` instruction must always be the first instruction in a relup. If the relup is generated by `systools:make_relup/3,4`, this is automatically ensured.

When the release handler encounters the instruction, it first generates a temporary boot file, which starts the new versions of the emulator and the core applications, and the old version of all other applications. Then it shuts down the current emulator by calling `init:reboot()`, see the `init(3)` manual page in Kernel. All processes are terminated gracefully and the system is rebooted by the `heart` program, using the temporary boot file. After the reboot, the rest of the relup instructions are executed. This is done as a part of the temporary boot script.

Warning:

This mechanism causes the new versions of the emulator and core applications to run with the old version of other applications during startup. Thus, take extra care to avoid incompatibility. Incompatible changes in the core applications can in some situations be necessary. If possible, such changes are preceded by deprecation over two major releases before the actual change. To ensure the application is not crashed by an incompatible change, always remove any call to deprecated functions as soon as possible.

An info report is written when the upgrade is completed. To programmatically find out if the upgrade is complete, call `release_handler:which_releases(current)` and check if it returns the expected (that is, the new) release.

The new release version must be made permanent when the new emulator is operational. Otherwise, the old version will be used if there is a new system reboot.

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

restart_emulator (Low-Level)

This instruction is not related to upgrades of ERTS or any of the core applications. It can be used by any application to force a restart of the emulator after all upgrade instructions are executed.

A relup script can only have one `restart_emulator` instruction and it must always be placed at the end. If the relup is generated by `systools:make_relup/3,4`, this is automatically ensured.

When the release handler encounters the instruction, it shuts down the emulator by calling `init:reboot()`, see the `init(3)` manual page in Kernel. All processes are terminated gracefully and the system can then be rebooted by the `heart` program using the new release version. No more upgrade instruction is executed after the restart.

10.11.5 Application Upgrade File

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

```
{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.
- Each `DownToVsn` is a previous version of the application to downgrade to.
- Each `Instructions` is a list of release handling instructions.

For information about the syntax and contents of the `.appup` file, see the `appup(4)` manual page in SASL.

Appup Cookbook includes examples of `.appup` files for typical upgrade/downgrade cases.

Example: Consider the release `ch_rel-1` from Releases. Assume you want to add a function `available/0` to server `ch3`, which returns the number of available channels (when trying out the example, change 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"`), you only need to load the new (old) version of the `ch3` callback module. Create the application upgrade file `ch_app.appup` in the `ebin` directory:

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

10.11.6 Release Upgrade File

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

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 deduced which applications are to be added and deleted, and which applications that must be upgraded and/or downgraded. The instructions for this are 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. It is only to contain low-level instructions.

For details about the syntax and contents of the release upgrade file, see the `relup(4)` manual page in SASL.

Example, continued from the previous section: You have a new version "2" of `ch_app` and an `.appup` file. A new version of the `.rel` file is also needed. This time the file is called `ch_rel-2.rel` and the release version string is 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 generates 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".

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. The code path can be extended 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
```

10.11.7 Installing a Release

When you 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 runtime, the **release handler** is used. This is a process belonging to the SASL application, which handles unpacking, installation, and removal of release packages. It is communicated through the `release_handler` module. For details, see the `release_handler(3)` manual page in SASL.

Assuming there is an operational target system with installation root directory `$ROOT`, the release package with the new version of the release is to be copied to `$ROOT/releases`.

First, **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 `$ROOT/lib/releases/Vsn` is created, where the `.rel` file, the boot script `start.boot`, the system configuration file `sys.config`, and `relup` are placed. For applications with new version numbers, the application directories are placed under `$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 if anything happens and the system is rebooted, it starts 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)

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

```
[].
```

Step 2) Start the system as a simple target system. In reality, it is to 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.

Step 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.

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

The new release package now also contains version "2" of `ch_app` and the `relup` file:

```
% 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/B/ch_rel-2.rel
releases/ch_rel-2.rel
```

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

Step 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`.

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

```
2> ch3:available().
** exception error: undefined function ch3:available/0
```

Step 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"
```

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

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

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

10.11.8 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 is fetched from the boot script included in the release package. Thus, it is important that the boot script is generated from the same `.rel` file as is used to build the release package itself.

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

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

This means that parameter values set in the other system configuration files and 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 compares 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 can be omitted when implementing an application callback module.

10.12 Appup Cookbook

This section includes examples of `.appup` files for typical cases of upgrades/downgrades done in runtime.

10.12.1 Changing a Functional Module

When a functional module has been changed, for example, if a new function has been added or a bug has been corrected, simple code replacement is sufficient, for example:

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

10.12.2 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`, `gen_statem` 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.

10.12.3 Changing a Callback Module

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

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

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

OTP also supports changing the internal state of behaviour processes, see Changing Internal State.

10.12.4 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 `gen_server` `ch3` from gen_server Behaviour. The internal state is a term `Chs` representing the available channels. Assume you want to add a counter `N`, which keeps track of the number of `alloc` requests so far. This means that the format must be changed to `{Chs,N}`.

The `.appup` file can 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 are to 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 the `gen_server(3)` manual page in STDLIB). 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}` if there is a downgrade, or `Vsn` if there is a upgrade. The term `Vsn` is fetched from the 'original' version of the module, that is, the version you 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` must also be modified and perhaps a new interface function must be added, but this is not shown here.

10.12.5 Module Dependencies

Assume that a module is extended by adding an interface function, as in the example in Release Handling, where a function `available/0` is added to `ch3`.

If a call is added to this function, say in module `m1`, a runtime error could can 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`, in the upgrade case, and conversely in the downgrade case. `m1` is said to be **dependent on** `ch3`. In a release handling instruction, this is expressed by the `DepMods` element:

```
{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 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 instead `m1` and `ch3` belong to the same application, the `.appup` file can look as follows:

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

`m1` is dependent on `ch3` also when downgrading. `systools` knows the difference between up- and downgrading and generates a correct `relup`, where `ch3` is loaded before `m1` when upgrading, but `m1` is loaded before `ch3` when downgrading.

10.12.6 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. Otherwise the release handler cannot find the process.

Example: Consider the example `ch4` in sys and proc_lib. When started by a supervisor, the child specification can look as follows:

```
{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 is to be loaded when upgrading from version "1" to "2" of this application, `sp_app.appup` can look as follows:

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

The `update` instruction must contain the tuple `{advanced,Extra}`. The instruction makes 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 function `sys:handle_system_msg(Request, From, Parent, Module, Deb, State)`, and called by the special process when a system message is received. In `ch4`, the internal state is the set of available channels `Chs`.
- The second argument is the name of the module (`ch4`).
- The third argument is `Vsn` or `{down,Vsn}`, as described for `gen_server:code_change/3` in Changing Internal State.

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 instead the internal state is changed (similar to the example in Changing Internal State), this is done in this function and `{ok,Chs2}` returned.

10.12.7 Changing a Supervisor

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

Child processes can be added or deleted, but this is not handled automatically. Instructions must be given by in the `.appup` file.

Changing Properties

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

First, the new version of the callback module must be loaded, 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: To change the restart strategy of `ch_sup` (from Supervisor Behaviour) from `one_for_one` to `one_for_all`, change the callback function `init/1` in `ch_sup.erl`:

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

init(_Args) ->
    {ok, {#{strategy => one_for_all, ...}, ...}}.
```

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 earlier:

```
{"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 is to be restarted, if needed later on.

The id of the child specification cannot be changed.

Changing the `Modules` field of the child specification can 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 earlier, changing child specifications does not affect existing child processes. New child specifications are automatically added, but not deleted. Child processes are not automatically started or terminated, this must be done using `apply` instructions.

Example: Assume a new child process `m1` is to be added to `ch_sup` when upgrading `ch_app` from "1" to "2". This means `m1` is to 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}
   ]}]
}.
```

The order of the instructions is important.

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`, and so on, must be written. This function is then to 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}
   ]}]
}.
```

As stated earlier, the order of the instructions is important. When upgrading, `m1` must be loaded, and the supervisor child specification changed, before the new child process can be started. When downgrading, the child process must be terminated before the child specification is changed and the module is deleted.

10.12.8 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}]}]
```

10.12.9 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 in Changing a Supervisor.

10.12.10 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 the `add_application` and `remove_application` instructions are added automatically.

10.12.11 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 child `m1` to `ch_sup`, as in Adding and Deleting Child Processes in Changing a Supervisor, an alternative to updating the supervisor is to restart the entire application:

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

10.12.12 Changing an Application Specification

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

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

10.12.13 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 the previous section.

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

10.12.14 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 there is a release containing an application `prim_app`, which have a supervisor `prim_sup` in its supervision tree.

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

The workflow is as follows:

Step 1) Edit the code for `prim_sup`:

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

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

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

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

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

The included application can be started in two ways. This is described in the next two sections.

Application Restart

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

However, if this is done and a `relup` file is generated, not only would it contain instructions for restarting (that is, removing and adding) `prim_app`, it would also contain instructions for starting `ch_app` (and stopping it, in the case of downgrade). This is because `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

Step 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` is to 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]}}]}]
}.
```

10.12.15 Changing Non-Erlang Code

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

Example: When 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 is to be updated, the code for the `gen_server` can be extended with a `code_change` function, which closes the old port and opens a new port. (If necessary, the `gen_server` can first request data that must 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}}]}]
].
```

Ensure that the `priv` directory, where the C program is located, is included in the new release package:

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

10.12.16 Emulator Restart and Upgrade

Two upgrade instructions restart the emulator:

- `restart_new_emulator`

  Intended when ERTS, Kernel, STDLIB, or SASL is upgraded. It is automatically added when the `relup` file is generated by `systools:make_relup/3,4`. It is executed before all other upgrade instructions. For more information about this instruction, see restart_new_emulator (Low-Level) in Release Handling Instructions.
- `restart_emulator`

  Used when a restart of the emulator is required after all other upgrade instructions are executed. For more information about this instruction, see restart_emulator (Low-Level) in Release Handling Instructions.

If an emulator restart is necessary and no upgrade instructions are needed, that is, if the restart itself is enough for the upgraded applications to start running the new versions, a simple `relup` file can be created manually:

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

In this case, the release handler framework with automatic packing and unpacking of release packages, automatic path updates, and so on, can be used without having to specify `.appup` files.

10.12.17 Emulator Upgrade From Pre OTP R15

From OTP R15, an emulator upgrade is performed by restarting the emulator with new versions of the core applications (Kernel, STDLIB, and SASL) before loading code and running upgrade instruction for other applications. For this to work, the release to upgrade from must include OTP R15 or later.

For the case where the release to upgrade from includes an earlier emulator version, `systools:make_relup` creates a backwards compatible relup file. This means that all upgrade instructions are executed before the emulator is restarted. The new application code is therefore loaded into the old emulator. If the new code is compiled with the

new emulator, there can be cases where the beam format has changed and beam files cannot be loaded. To overcome this problem, compile the new code with the old emulator.

11 OAM Principles

11.1 Introduction

The Operation and Maintenance (OAM) support in OTP consists of a generic model for management subsystems in OTP, and some components to be used in these subsystems. This section describes the model.

The main idea in the model is that it is not tied to any specific management protocol. An Application Programming Interface (API) is defined, which can be used to write adaptations for specific management protocols.

Each OAM component in OTP is implemented as one sub-application, which can be included in a management application for the system. Notice that such a complete management application is not in the scope of this generic functionality. However, this section includes examples illustrating how such an application can be built.

11.1.1 Terminology

The protocol-independent architectural model on the network level is the well-known client-server model for management operations. This model is based on the client-server principle, where the manager (client) sends a request from a manager to an agent (server) when it accesses management information. The agent sends a reply back to the manager. There are two main differences to the normal client-server model:

- Usually a few managers communicate with many agents.
- The agent can spontaneously send a notification, for example, an alarm, to the manager.

The following picture illustrates the idea:

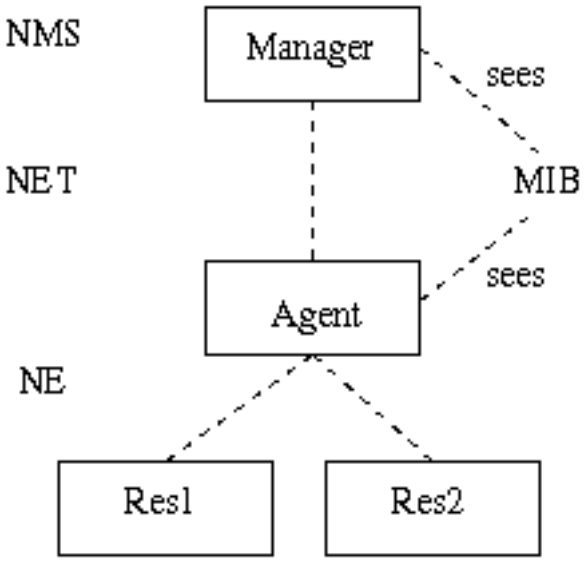


Figure 1.1: Terminology

The manager is often referred to as the **Network Management System (NMS)**, to emphasize that it usually is realized as a program that presents data to an operator.

The agent is an entity that executes within a **Network Element (NE)**. In OTP, the NE can be a distributed system, meaning that the distributed system is managed as one entity. Of course, the agent can be configured to be able to run on one of several nodes, making it a distributed OTP application.

The management information is defined in a **Management Information Base (MIB)**. It is a formal definition of which information the agent makes available to the manager. The manager accesses the MIB through a management protocol, such as SNMP, CMIP, HTTP, or CORBA. Each protocol has its own MIB definition language. In SNMP, it is a subset of ASN.1, in CMIP it is GDMO, in HTTP it is implicit, and using CORBA, it is IDL.

Usually, the entities defined in the MIB are called **Managed Objects (MOs)**, although they do not have to be objects in the object-oriented way. For example, a simple scalar variable defined in a MIB is called an MO. The MOs are logical objects, not necessarily with a one-to-one mapping to the resources.

11.1.2 Model

This section presents the generic protocol-independent model for use within an OTP-based NE. This model is used by all OAM components and can be used by the applications. The advantage of the model is that it clearly separates the resources from the management protocol. The resources do not need to be aware of which management protocol is used to manage the system. The same resources can therefore be managed with different protocols.

The entities involved in this model are the agent, which terminates the management protocol, and the resources, which is to be managed, that is, the actual application entities. The resources should in general have no knowledge of the management protocol used, and the agent should have no knowledge of the managed resources. This implies that a translation mechanism is needed, to translate the management operations to operations on the resources. This translation mechanism is usually called **instrumentation** and the function that implements it is called **instrumentation function**. The instrumentation functions are written for each combination of management protocol and resource to be managed. For example, if an application is to be managed by SNMP and HTTP, two sets of instrumentation functions are defined; one that maps SNMP requests to the resources, and one that, for example, generates an HTML page for some resources.

When a manager makes a request to the agent, the following illustrates the situation:

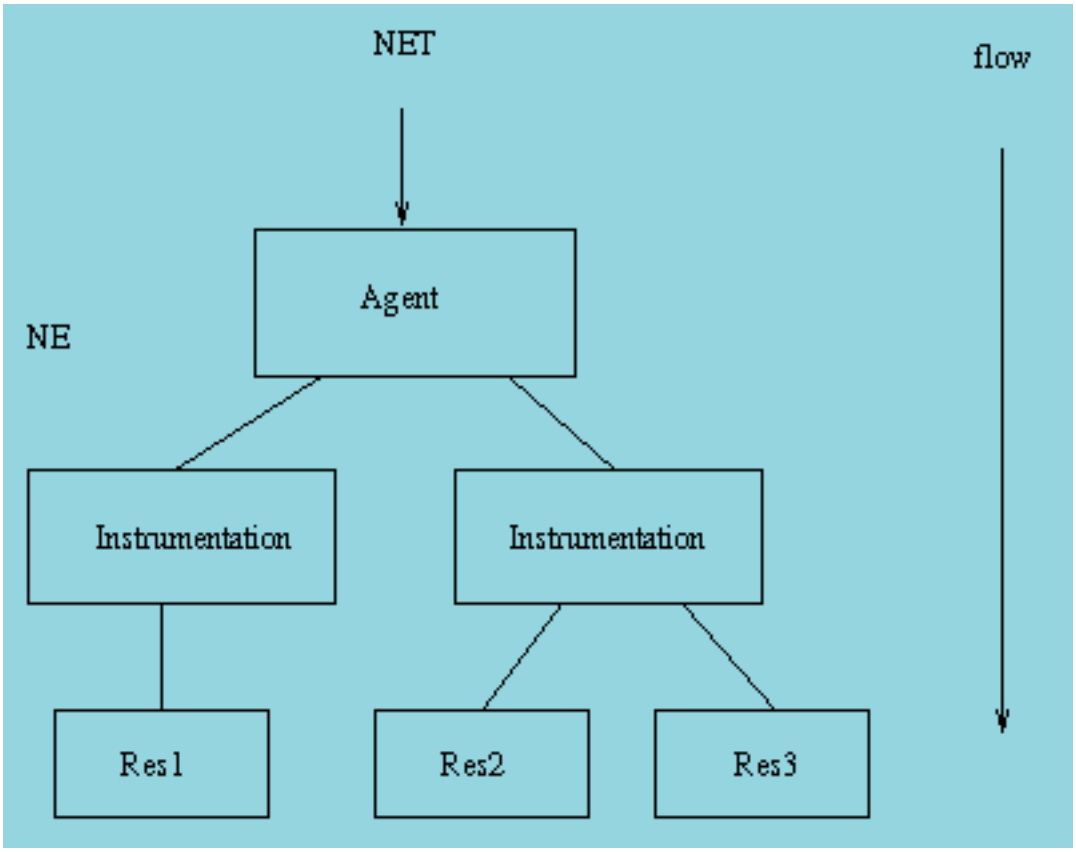


Figure 1.2: Request to An Agent by a Manager

The mapping between an instrumentation function and a resource is not necessarily 1-1. It is also possible to write one instrumentation function for each resource, and use that function from different protocols.

The agent receives a request and maps it to calls to one or more instrumentation functions. These functions perform operations on the resources to implement the semantics associated with the MO.

For example, a system that is managed with SNMP and HTTP can be structured as follows:

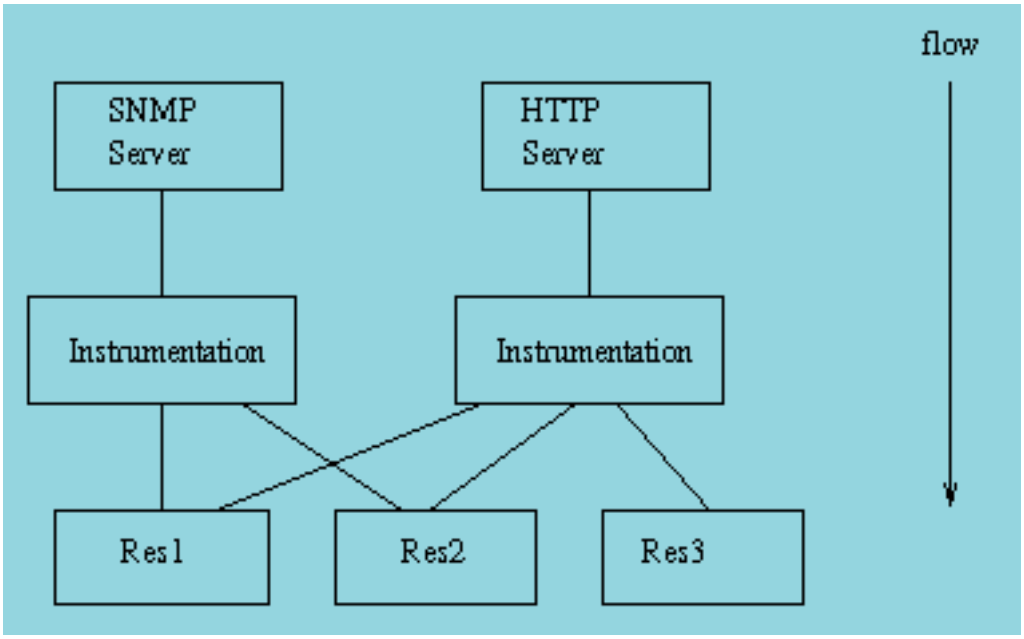


Figure 1.3: Structure of a System Managed with SNMP and HTTP

The resources can send notifications to the manager as well. Examples of notifications are events and alarms. The resource needs to generate protocol-independent notifications. The following picture illustrates how this is achieved:

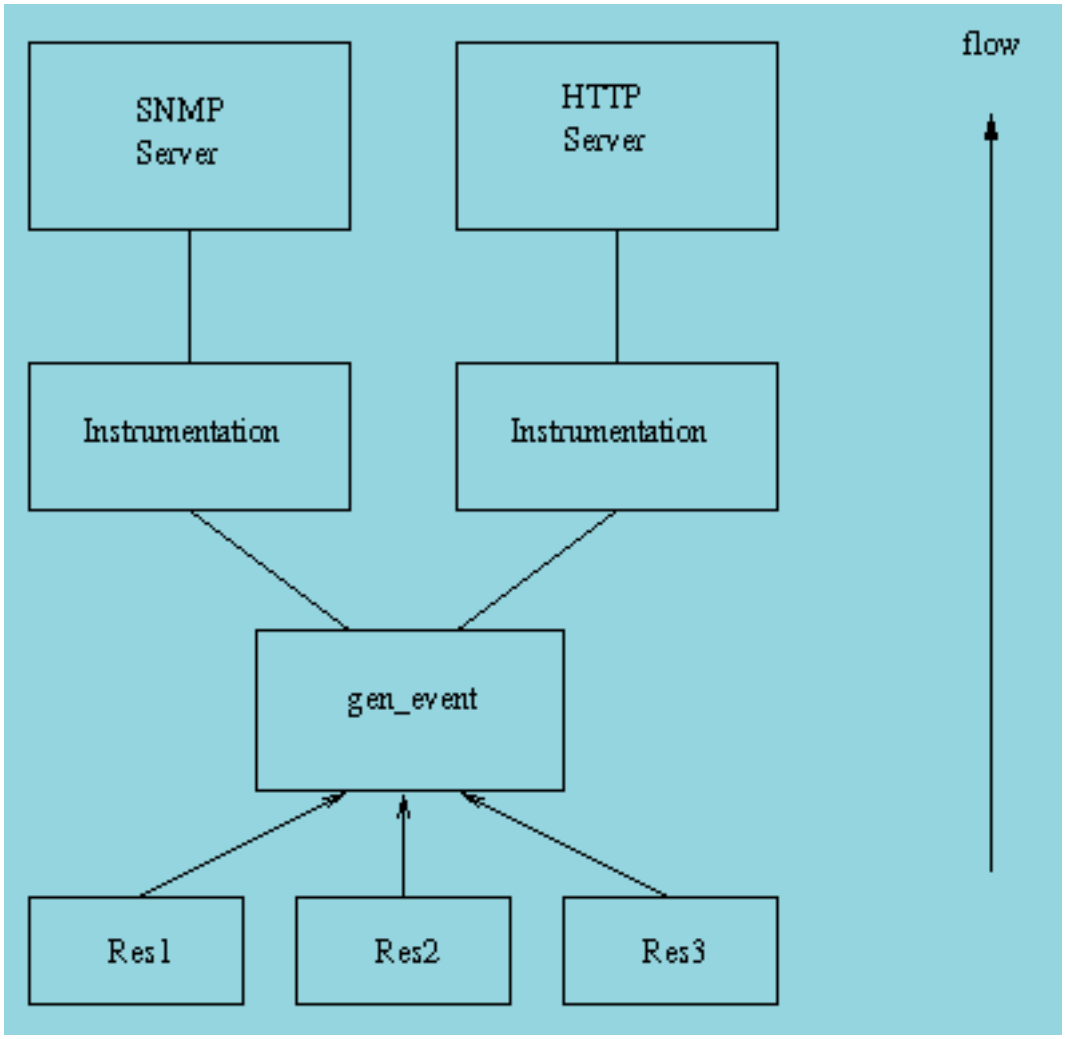


Figure 1.4: Notification Handling

The main idea is that the resource sends the notifications as Erlang terms to a dedicated `gen_event` process. Into this process, handlers for the different management protocols are installed. When an event is received by this process, it is forwarded to each installed handler. The handlers are responsible for translating the event into a notification to be sent over the management protocol. For example, a handler for SNMP translates each event into an SNMP trap.

11.1.3 SNMP-Based OAM

For all OAM components, SNMP adaptations are provided. Other adaptations might be defined in the future.

The OAM components, and some other OTP applications, define SNMP MIBs. These MIBs are written in SNMPv2 SMI syntax, as defined in RFC 1902. For convenience we also deliver the SNMPv1 SMI equivalent. All MIBs are designed to be v1/v2 compatible, that is, the v2 MIBs do not use any construct not available in v1.

MIB Structure

The top-level OTP MIB is called `OTP-REG` and it is included in the SNMP application. All other OTP MIBs import some objects from this MIB.

Each MIB is contained in one application. The MIB text files are stored under `mibs/<MIB>.mib` in the application directory. The generated `.hrl` files with constant declarations are stored under `include/<MIB>.hrl`, and the compiled MIBs are stored under `priv/mibs/<MIB>.bin`.

An application that needs to import an MIB into another MIB is to use the `il` option to the SNMP MIB compiler:

```
snmp:c("MY-MIB", [{il, ["snmp/priv/mibs"]}]).
```

If the application needs to include a generated `.hrl` file, it is to use the `-include_lib` directive to the Erlang compiler:

```
-module(my_mib).
-include_lib("snmp/include/OTP-REG.hrl").
```

Here is a list of some of the MIBs defined in the OTP system:

- `OTP-REG` (in SNMP) contains the top-level OTP registration objects, used by all other MIBs.
- `OTP-TC` (in SNMP) contains the general Textual Conventions, which can be used by any other MIB.
- `OTP-SNMPEA-MIB` (in `snmp`) contains objects for instrumentation and control of the extensible SNMP agent itself. The agent also implements the standard SNMPv2-MIB (or v1 part of MIB-II, if SNMPv1 is used).

The different applications use different strategies for loading the MIBs into the agent. Some MIB implementations are code-only, while others need a server. One way, used by the code-only MIB implementations, is for the user to call a function such as `snmpa:load_mibs(Agent, [Mib])` to load the MIB, and `snmpa:unload_mibs(Agent, [Mib])` to unload the MIB. See the manual page for each application for a description of how to load each MIB.