

System Architecture Support Libraries (SASL)

Copyright © 1997-2022 Ericsson AB. All Rights Reserved.
System Architecture Support Libraries (SASL) 4.2
October 3, 2022

Copyright © 1997-2022 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..

October 3, 2022

1 SASL User's Guide

The System Architecture Support Libraries SASL application provides support for alarm handling, release handling, and related functions.

1.1 Introduction

1.1.1 Scope

The SASL application provides support for:

- Error logging
- Alarm handling
- Release handling
- Report browsing

Section SASL Error Logging describes the error handler that produces the supervisor, progress, and crash reports, which can be written to screen or to a specified file. It also describes the Report Browser (RB).

The sections about release structure and release handling have been moved to section OTP Design Principles in **System Documentation**.

1.1.2 Prerequisites

It is assumed that the reader is familiar with the Erlang programming language.

1.2 SASL Error Logging

Note:

The SASL error logging concept described in this section is deprecated since Erlang/OTP 21.0, when the new logging API was introduced.

The new default behaviour is that the SASL application no longer affects which log events that are logged. Supervisor reports and crash reports are logged via the default logger handler which is setup by Kernel. 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`.

The old SASL error logging behaviour can be re-enabled by setting the Kernel configuration parameter `logger_sasl_compatible` to `true`.

The mechanism for multi-file error report logging as described in this section is also kept for backwards compatibility. However, the new logging API also introduces `logger_disk_log_h(3)`, which is a logger handler that can print to multiple files using `disk_log(3)`.

1.2.1 SASL reports

The SASL application introduces three types of reports:

- Supervisor report

- Progress report
- Crash report

When the SASL application is started, it adds a Logger handler that formats and writes these reports, as specified in the configuration parameters for SASL.

Supervisor Report

A supervisor report is issued when a supervised child terminates unexpectedly. A supervisor report contains the following items:

`Supervisor`

Name of the reporting supervisor.

`Context`

Indicates in which phase the child terminated from the supervisor's point of view. This can be `start_error`, `child_terminated`, or `shutdown_error`.

`Reason`

Termination reason.

`Offender`

Start specification for the child.

Progress Report

A progress report is issued when a supervisor starts or restarts a child. A progress report contains the following items:

`Supervisor`

Name of the reporting supervisor.

`Started`

Start specification for the successfully started child.

Crash Report

Processes started with functions `proc_lib:spawn` or `proc_lib:spawn_link` are wrapped within a `catch`. A crash report is issued when such a process terminates with an unexpected reason, which is any reason other than `normal`, `shutdown`, or `{shutdown,Term}`. Processes using behaviors `gen_server`, `gen_fsm` or `gen_statem` are examples of such processes. A crash report contains the following items:

`Crasher`

Information about the crashing process, such as initial function call, exit reason, and message queue.

`Neighbours`

Information about processes that are linked to the crashing process and do not trap exits. These processes are the neighbours that terminate because of this process crash. The information gathered is the same as the information for Crasher, described in the previous item.

Example

The following example shows the reports generated when a process crashes. The example process is a `permanent` process supervised by the `test_sup` supervisor. A division by zero is executed and the error is first reported by the faulty process. A crash report is generated, as the process was started using function `proc_lib:spawn/3`. The supervisor generates a supervisor report showing the crashed process. A progress report is generated when the process is finally restarted.

```
=ERROR REPORT==== 27-May-1996::13:38:56 ===
<0.63.0>: Divide by zero !

=CRASH REPORT==== 27-May-1996::13:38:56 ===
crasher:
pid: <0.63.0>
registered_name: []
error_info: {badarith,{test,s,[]}}
initial_call: {test,s,[]}
ancestors: [test_sup,<0.46.0>]
messages: []
links: [<0.47.0>]
dictionary: []
trap_exit: false
status: running
heap_size: 128
stack_size: 128
reductions: 348
neighbours:

=SUPERVISOR REPORT==== 27-May-1996::13:38:56 ===
Supervisor: {local,test_sup}
Context:    child_terminated
Reason:     {badarith,{test,s,[]}}
Offender:   [{pid,<0.63.0>},
{name,test},
{mfa,{test,t,[]}},
{restart_type,permanent},
{shutdown,200},
{child_type,worker}]

=PROGRESS REPORT==== 27-May-1996::13:38:56 ===
Supervisor: {local,test_sup}
Started:  [{pid,<0.64.0>},
{name,test},
{mfa,{test,t,[]}},
{restart_type,permanent},
{shutdown,200},
{child_type,worker}]
```

1.2.2 Multi-File Error Report Logging

Multi-file error report logging is used to store error messages received by `error_logger`. The error messages are stored in several files and each file is smaller than a specified number of kilobytes. No more than a specified number of files exist at the same time. The logging is very fast, as each error message is written as a binary term.

For more details, see the `sasl(6)` application in the Reference Manual.

1.2.3 Report Browser

The report browser is used to browse and format error reports written by the error logger handler `log_mf_h` defined in STDLIB.

The `log_mf_h` handler writes all reports to a report logging directory, which is specified when configuring the SASL application.

If the report browser is used offline, the reports can be copied to another directory specified when starting the browser. If no such directory is specified, the browser reads reports from the SASL `error_logger_mf_dir`.

Starting Report Browser

Start the `rb_server` with function `rb:start([Options])` as shown in the following example:

```
5> rb:start([{max, 20}]).
rb: reading report...done.
rb: reading report...done.
rb: reading report...done.
rb: reading report...done.
{ok,<0.199.0>}
```

Online Help

Enter command `rb:help()` to access the report browser online help system.

List Reports in Server

Use function `rb:list()` to list all loaded reports:

```
4> rb:list().
No                Type          Process       Date     Time
==                ====          =======       ====     ====
20            progress         <0.17.0> 1996-10-16 16:14:54
19            progress         <0.14.0> 1996-10-16 16:14:55
18               error         <0.15.0> 1996-10-16 16:15:02
17            progress         <0.14.0> 1996-10-16 16:15:06
16            progress         <0.38.0> 1996-10-16 16:15:12
15            progress         <0.17.0> 1996-10-16 16:16:14
14            progress         <0.17.0> 1996-10-16 16:16:14
13            progress         <0.17.0> 1996-10-16 16:16:14
12            progress         <0.14.0> 1996-10-16 16:16:14
11               error         <0.17.0> 1996-10-16 16:16:21
10               error         <0.17.0> 1996-10-16 16:16:21
9        crash_report  release_handler 1996-10-16 16:16:21
8   supervisor_report         <0.17.0> 1996-10-16 16:16:21
7             progress         <0.17.0> 1996-10-16 16:16:21
6             progress         <0.17.0> 1996-10-16 16:16:36
5             progress         <0.17.0> 1996-10-16 16:16:36
4             progress         <0.17.0> 1996-10-16 16:16:36
3             progress         <0.14.0> 1996-10-16 16:16:36
2                error         <0.15.0> 1996-10-16 16:17:04
1             progress         <0.14.0> 1996-10-16 16:17:09
ok
```

Show Reports

Use function `rb:show(Number)` to show details of a specific report:

```
7> rb:show(4).

PROGRESS REPORT  <0.20.0>                                  1996-10-16 16:16:36
===============================================================================
supervisor                                                    {local,sasl_sup}
started
[{pid,<0.24.0>},
{name,release_handler},
{mfa,{release_handler,start_link,[]}},
{restart_type,permanent},
{shutdown,2000},
{child_type,worker}]

ok
8> rb:show(9).

CRASH REPORT  <0.24.0>                                     1996-10-16 16:16:21
===============================================================================
Crashing process
pid                                                                <0.24.0>
registered_name                                             release_handler
error_info                            {undef,{release_handler,mbj_func,[]}}
initial_call
{gen,init_it,
[gen_server,
<0.20.0>,
<0.20.0>,
{erlang,register},
release_handler,
release_handler,
[],
[]]}
ancestors                                                [sasl_sup,<0.18.0>]
messages                                                                  []
links                                                    [<0.23.0>,<0.20.0>]
dictionary                                                                []
trap_exit                                                              false
status                                                               running
heap_size                                                                610
stack_size                                                               142
reductions                                                                54

ok
```

Search Reports

All reports containing a common pattern can be shown. Suppose a process crashes because it tries to call a non-existing function `release_handler:mbj_func/1`. The reports can then be shown as follows:

```
12> rb:grep("mbj_func").
Found match in report number 11

ERROR REPORT  <0.24.0>                                        1996-10-16 16:16:21
==============================================================================

** undefined function: release_handler:mbj_func[] **
Found match in report number 10

ERROR REPORT  <0.24.0>                                        1996-10-16 16:16:21
==============================================================================

** Generic server release_handler terminating
** Last message in was {unpack_release,hej}
** When Server state == {state,[],
"/home/dup/otp2/otp_beam_sunos5_p1g_7",
[{release,
"OTP  APN 181 01",
"P1G",
undefined,
[],
permanent}],
undefined}
** Reason for termination ==
** {undef,{release_handler,mbj_func,[]}}
Found match in report number 9

CRASH REPORT  <0.24.0>                                        1996-10-16 16:16:21
==============================================================================
Crashing process
pid                                                                 <0.24.0>
registered_name                                               release_handler
error_info                              {undef,{release_handler,mbj_func,[]}}
initial_call
{gen,init_it,
[gen_server,
<0.20.0>,
<0.20.0>,
{erlang,register},
release_handler,
release_handler,
[],
[]]}
ancestors                                                [sasl_sup,<0.18.0>]
messages                                                                []
links                                                    [<0.23.0>,<0.20.0>]
dictionary                                                              []
trap_exit                                                          false
status                                                           running
heap_size                                                            610
stack_size                                                           142
reductions                                                            54

Found match in report number 8

SUPERVISOR REPORT  <0.20.0>                                   1996-10-16 16:16:21
==============================================================================
Reporting supervisor                                          {local,sasl_sup}

Child process
errorContext                                                 child_terminated
reason                                  {undef,{release_handler,mbj_func,[]}}
pid                                                                 <0.24.0>
name                                                          release_handler
```

```
start_function                              {release_handler,start_link,[]}
restart_type                                                     permanent
shutdown                                                              2000
child_type                                                          worker

ok
```

Stop Server

Use function `rb:stop()` to stop the `rb_server`:

```
13> rb:stop().
ok
```

2 Reference Manual

The SASL application provides support for alarm handling, release handling, and related functions.

sasl

Application

The SASL application provides the following services:

- `alarm_handler`
- `release_handler`
- `systools`

> **Note:**
>
> The SASL application in OTP has nothing to do with "Simple Authentication and Security Layer" (RFC 4422).

Configuration

The following configuration parameters are defined for the SASL application. For more information about configuration parameters, see `app(4)` in Kernel.

All configuration parameters are optional.

`start_prg = string()`

Specifies the program to be used when restarting the system during release installation. Default is `$OTP_ROOT/bin/start`.

`masters = [atom()]`

Specifies the nodes used by this node to read/write release information. This parameter is ignored if parameter `client_directory` is not set.

`client_directory = string()`

This parameter specifies the client directory at the master nodes. For details, see Release Handling in **OTP Design Principles**. This parameter is ignored if parameter `masters` is not set.

`static_emulator = true | false`

Indicates if the Erlang emulator is statically installed. A node with a static emulator cannot switch dynamically to a new emulator, as the executable files are written into memory statically. This parameter is ignored if parameters `masters` and `client_directory` are not set.

`releases_dir = string()`

Indicates where the `releases` directory is located. The release handler writes all its files to this directory. If this parameter is not set, the OS environment parameter `RELDIR` is used. By default, this is `$OTP_ROOT/releases`.

Deprecated Error Logger Event Handlers and Configuration

In Erlang/OTP 21.0, a new API for logging was added. The old `error_logger` event manager, and event handlers running on this manager, still work, but they are not used by default.

The error logger event handlers `sasl_report_tty_h` and `sasl_report_file_h`, were earlier used for printing the so called SASL reports, i.e. **supervisor reports**, **crash reports**, and **progress reports**. These reports are now also printed by the default logger handler started by the Kernel application. Progress reports are by default stopped by the primary log level, but can be enabled by setting this level to `info`, for example by using the Kernel configuration parameter `logger_level`.

If the old error logger event handlers are still desired, they must be added by calling `error_logger:add_report_handler/1,2`.

`sasl_report_tty_h`

Formats and writes **supervisor reports**, **crash reports**, and **progress reports** to `stdio`. This error logger event handler uses `error_logger_format_depth` in the Kernel application to limit how much detail is printed in crash and supervisor reports.

`sasl_report_file_h`

Formats and writes **supervisor reports**, **crash report**, and **progress report** to a single file. This error logger event handler uses `error_logger_format_depth` in the Kernel application to limit the details printed in crash and supervisor reports.

A similar behaviour, but still using the new logger API, can be obtained by setting the Kernel application environment variable `logger_sasl_compatible` to `true`. This adds a second instance of the standard Logger handler, named `sasl`, which only prints the SASL reports. No SASL reports are then printed by the Kernel logger handler.

The `sasl` handler is configured according to the values of the following SASL application environment variables.

`sasl_error_logger = Value`

`Value` is one of the following:

`tty`

Installs `sasl_report_tty_h` in the error logger. This is the default option.

`{file,FileName}`

Installs `sasl_report_file_h` in the error logger. All reports go to file `FileName`, which is a string. The file is opened in `write` mode with encoding `utf8`.

`{file,FileName,Modes}`

Same as `{file,FileName}`, except that `Modes` allows you to specify the modes used for opening the `FileName` given to the file:open/2 call. By default, the file is opened in `write` mode with encoding `utf8`. Use `[append]` to have the `FileName` open in append mode. A different encoding can also be specified. `FileName` is a string.

`false`

No SASL error logger handler is installed.

`errlog_type = error | progress | all`

Restricts the error logging performed by the specified `sasl_error_logger` to error reports or progress reports, or both. Default is `all`.

`utc_log = true | false`

If set to `true`, all dates in textual log outputs are displayed in Universal Coordinated Time with the string `UTC` appended.

The error logger event handler `log_mf_h` can also still be used. This event handler writes **all** events sent to the error logger to disk. Multiple files and log rotation are used. For efficiency reasons, each event is written as a binary. For more information about this handler, see the STDLIB Reference Manual.

To activate this event handler, three SASL configuration parameters must be set:

`error_logger_mf_dir = string() | false`

Specifies in which directory `log_mf_h` is to store its files. If this parameter is undefined or `false`, the `log_mf_h` handler is not installed.

`error_logger_mf_maxbytes = integer()`

Specifies the maximum size of each individual file written by `log_mf_h`. If this parameter is undefined, the `log_mf_h` handler is not installed.

`error_logger_mf_maxfiles = 0<integer()<256`

Specifies the number of files used by `log_mf_h`. If this parameter is undefined, the `log_mf_h` handler is not installed.

The new `logger_disk_log_h` might be an alternative to `log_mf_h` if log rotation is desired. This does, however, write the log events in clear text and not as binaries.

See Also

`alarm_handler(3)`, `error_logger(3)`, `logger(3)`, `log_mf_h(3)`, `rb(3)`, `release_handler(3)`, `systools(3)`

alarm_handler

Erlang module

The alarm handler process is a `gen_event` event manager process that receives alarms in the system. This process is not intended to be a complete alarm handler. It defines a place to which alarms can be sent. One simple event handler is installed in the alarm handler at startup, but users are encouraged to write and install their own handlers.

The simple event handler sends all alarms as info reports to the error logger, and saves all in a list. This list can be passed to a user-defined event handler, which can be installed later. The list can grow large if many alarms are generated. This is a good reason to install a better user-defined handler.

Functions are provided to set and clear alarms. The alarm format is defined by the user. For example, an event handler for SNMP can be defined, together with an alarm Management Information Base (MIB).

The alarm handler is part of the SASL application.

When writing new event handlers for the alarm handler, the following events must be handled:

`{set_alarm, {AlarmId, AlarmDescr}}`

This event is generated by `alarm_handler:set_alarm({AlarmId, AlarmDecsr})`.

`{clear_alarm, AlarmId}`

This event is generated by `alarm_handler:clear_alarm(AlarmId)`.

The default simple handler is called `alarm_handler` and it can be exchanged by calling `gen_event:swap_handler/3` as `gen_event:swap_handler(alarm_handler, {alarm_handler, swap}, {NewHandler, Args})`. `NewHandler:init({Args, {alarm_handler, Alarms}})` is called. For more details, see `gen_event(3)` in STDLIB.

Exports

`clear_alarm(AlarmId) -> void()`

Types:

`AlarmId = term()`

Sends event `clear_alarm` to all event handlers.

When receiving this event, the default simple handler clears the latest received alarm with id `AlarmId`.

`get_alarms() -> [alarm()]`

Returns a list of all active alarms. This function can only be used when the simple handler is installed.

`set_alarm(alarm())`

Types:

`alarm() = {AlarmId, AlarmDescription}`
`AlarmId = term()`
`AlarmDescription = term()`

Sends event `set_alarm` to all event handlers.

When receiving this event, the default simple handler stores the alarm. `AlarmId` identifies the alarm and is used when the alarm is cleared.

See Also

`error_logger(3)`, `gen_event(3)`

rb

Erlang module

The Report Browser (RB) tool is used to browse and format error reports written by the error logger handler `log_mf_h` in STDLIB.

Exports

`filter(Filters)`
`filter(Filters, Dates)`
Types:

```
Filters = [filter()]
filter() = {Key, Value} | {Key, Value, no} | {Key, RegExp, re} | {Key,
RegExp, re, no}
Key = term()
Value = term()
RegExp = string() | {string(), Options} | re:mp() | {re:mp(), Options}
Dates = {DateFrom, DateTo} | {DateFrom, from} | {DateTo, to}
DateFrom = DateTo = calendar:datetime()
```

Displays the reports that match the provided filters.

When a filter includes the `no` atom, it excludes the reports that match that filter.

The reports are matched using the `proplists` module in STDLIB. The report must be a proplist to be matched against any of the filters.

If the filter has the form `{Key, RegExp, re}`, the report must contain an element with key equal to `Key` and the value must match the regular expression `RegExp`.

If parameter `Dates` is specified, the reports are filtered according to the date when they occurred. If `Dates` has the form `{DateFrom, from}`, reports that occurred after `DateFrom` are displayed.

If `Dates` has the form `{DateTo, to}`, reports that occurred before `DateTo` are displayed.

If two `Dates` are specified, reports that occurred between those dates are returned.

To filter only by dates, specify the empty list as the `Filters` parameter.

For details about parameter `RegExp`, see `rb:grep/1`.

For details about data type `mp()`, see `re:mp()`.

For details about data type `datetime()`, see `calendar:datetime()`.

`grep(RegExp)`
Types:

```
RegExp = string() | {string(), Options} | re:mp() | {re:mp(), Options}
```

All reports matching the regular expression `RegExp` are displayed. `RegExp` can be any of the following:

- A string containing the regular expression
- A tuple with the string and the options for compilation
- A compiled regular expression

- A compiled regular expression and the options for running it

For a definition of valid regular expressions and options, see the `re` module in STDLIB and in particular function `re:run/3`.

For details about data type `mp()`, see `re:mp()`.

`h()`
`help()`

Displays online help information.

`list()`
`list(Type)`

Types:

```
Type = type()
type() = error | error_report | info_msg | info_report | warning_msg |
warning_report | crash_report | supervisor_report | progress
```

Lists all reports loaded in `rb_server`. Each report is given a unique number that can be used as a reference to the report in function `show/1`.

If no `Type` is specified, all reports are listed.

`log_list()`
`log_list(Type)`

Types:

```
Type = type()
type() = error | error_report | info_msg | info_report | warning_msg |
warning_report | crash_report | supervisor_report | progress
```

Same as functions `list/0` or `list/1`, but the result is printed to a log file, if set; otherwise to `standard_io`.

If no `Type` is specified, all reports are listed.

`rescan()`
`rescan(Options)`

Types:

```
Options = [opt()]
```

Rescans the report directory. `Options` is the same as for function `start/1`.

`show()`
`show(Report)`

Types:

```
Report = integer() | type()
```

If argument `type` is specified, all loaded reports of this type are displayed. If an integer argument is specified, the report with this reference number is displayed. If no argument is specified, all reports are displayed.

`start()`

`start(Options)`

Types:

```
Options = [opt()]
opt() = {start_log, FileName} | {max, MaxNoOfReports} | {report_dir,
DirString} | {type, ReportType} | {abort_on_error, Bool}
FileName = string() | atom() | pid()
MaxNoOfReports = integer() | all
DirString = string()
ReportType = type() | [type()] | all
Bool = boolean()
```

Function `start/1` starts `rb_server` with the specified options, whereas function `start/0` starts with default options. `rb_server` must be started before reports can be browsed. When `rb_server` is started, the files in the specified directory are scanned. The other functions assume that the server has started.

Options:

`{start_log, FileName}`

Starts logging to file, registered name, or `io_device`. All reports are printed to the specified destination. Default is `standard_io`. Option `{start_log, standard_error}` is not allowed and will be replaced by default `standard_io`.

`{max, MaxNoOfReports}`

Controls how many reports `rb_server` is to read at startup. This option is useful, as the directory can contain a large amount of reports. If this option is specified, the `MaxNoOfReports` latest reports are read. Default is `all`.

`{report_dir, DirString}`

Defines the directory where the error log files are located. Default is the directory specified by application environment variable `error_logger_mf_dir`, see sasl(6).

`{type, ReportType}`

Controls what kind of reports `rb_server` is to read at startup. `ReportType` is a supported type, `all`, or a list of supported types. Default is `all`.

`{abort_on_error, Bool}`

Specifies if logging is to be ended if `rb` encounters an unprintable report. (You can get a report with an incorrect form if function `error_logger`, `error_msg`, or `info_msg` has been called with an invalid format string)

- If `Bool` is `true`, `rb` stops logging (and prints an error message to `stdout`) if it encounters a badly formatted report. If logging to file is enabled, an error message is appended to the log file as well.
- If `Bool` is `false` (the default value), `rb` prints an error message to `stdout` for every bad report it encounters, but the logging process is never ended. All printable reports are written. If logging to file is enabled, `rb` prints `* UNPRINTABLE REPORT *` in the log file at the location of an unprintable report.

`start_log(FileName)`

Types:

```
FileName = string() | atom() | pid()
```

Redirects all report output from the RB tool to the specified file, registered name, or `io_device`.

`stop()`

Stops `rb_server`.

`stop_log()`

Closes the log file. The output from the RB tool is directed to `standard_io`.

release_handler

Erlang module

The **release handler** process belongs to the SASL application, which is responsible for **release handling**, that is, unpacking, installation, and removal of release packages.

An introduction to release handling and an example is provided in OTP Design Principles in **System Documentation**.

A **release package** is a compressed tar file containing code for a certain version of a release, created by calling `systools:make_tar/1,2`. The release package is to be located in the `$ROOT/releases` directory of the previous version of the release, where `$ROOT` is the installation root directory, `code:root_dir()`. Another `releases` directory can be specified using the SASL configuration parameter `releases_dir` or the OS environment variable `RELDIR`. The release handler must have write access to this directory to install the new release. The persistent state of the release handler is stored there in a file called `RELEASES`.

A release package is always to contain:

- A release resource file, `Name.rel`
- A boot script, `Name.boot`

The `.rel` file contains information about the release: its name, version, and which ERTS and application versions it uses.

A release package can also contain:

- A release upgrade file, `relup`
- A system configuration file, `sys.config`
- A system configuration source file, `sys.config.src`

The `relup` file contains instructions for how to upgrade to, or downgrade from, this version of the release.

The release package can be **unpacked**, which extracts the files. An unpacked release can be **installed**. The currently used version of the release is then upgraded or downgraded to the specified version by evaluating the instructions in the `relup` file. An installed release can be made **permanent**. Only one permanent release can exist in the system, and this release is used if the system is restarted. An installed release, except the permanent one, can be **removed**. When a release is removed, all files belonging to that release only are deleted.

Each release version has a status, which can be `unpacked`, `current`, `permanent`, or `old`. There is always one latest release, which either has status `permanent` (normal case) or `current` (installed, but not yet made permanent). The meaning of the status values are illustrated in the following table:

```
Status     Action               NextStatus
-------------------------------------------
-          unpack               unpacked
unpacked   install              current
           remove               -
current    make_permanent       permanent
           install other        old
           remove               -
permanent  make other permanent old
           install              permanent
old        reboot_old           permanent
           install              current
           remove               -
```

The release handler process is a locally registered process on each node. When a release is installed in a distributed system, the release handler on each node must be called. The release installation can be synchronized between nodes.

From an operator view, it can be unsatisfactory to specify each node. The aim is to install one release package in the system, no matter how many nodes there are. It is recommended that software management functions are written that take care of this problem. Such a function can have knowledge of the system architecture, so it can contact each individual release handler to install the package.

For release handling to work properly, the runtime system must know which release it is running. It must also be able to change (in runtime) which boot script and system configuration file are to be used if the system is restarted. This is taken care of automatically if Erlang is started as an embedded system. Read about this in Embedded System in **System Documentation**. In this case, the system configuration file `sys.config` is mandatory.

The installation of a new release can restart the system. Which program to use is specified by the SASL configuration parameter `start_prg`, which defaults to `$ROOT/bin/start`.

The emulator restart on Windows NT expects that the system is started using the `erlsrv` program (as a service). Furthermore, the release handler expects that the service is named `NodeName_Release`, where `NodeName` is the first part of the Erlang node name (up to, but not including the "@") and `Release` is the current release version. The release handler furthermore expects that a program like `start_erl.exe` is specified as "machine" to `erlsrv`. During upgrading with restart, a new service is registered and started. The new service is set to automatic and the old service is removed when the new release is made permanent.

The release handler at a node running on a diskless machine, or with a read-only file system, must be configured accordingly using the following SASL configuration parameters (for details, see sasl(6)):

`masters`

This node uses some master nodes to store and fetch release information. All master nodes must be operational whenever release information is written by this node.

`client_directory`

The `client_directory` in the directory structure of the master nodes must be specified.

`static_emulator`

This parameter specifies if the Erlang emulator is statically installed at the client node. A node with a static emulator cannot dynamically switch to a new emulator, as the executable files are statically written into memory.

The release handler can also be used to unpack and install release packages when not running Erlang as an embedded system. However, in this case the user must somehow ensure that correct boot scripts and configuration files are used if the system must be restarted.

Functions are provided for using another file structure than the structure defined in OTP. These functions can be used to test a release upgrade locally.

Exports

```
check_install_release(Vsn) -> {ok, OtherVsn, Descr} | {error, Reason}
check_install_release(Vsn,Opts) -> {ok, OtherVsn, Descr} | {error, Reason}
```

Types:

```
Vsn = OtherVsn = string()
Opts = [Opt]
Opt = purge
Descr = term()
Reason = term()
```

Checks if the specified version `Vsn` of the release can be installed. The release must not have status `current`. Issues warnings if `relup` file or `sys.config` is not present. If `relup` file is present, its contents are checked and

{error,Reason} is returned if an error is found. Also checks that all required applications are present and that all new code can be loaded; {error,Reason} is returned if an error is found.

Evaluates all instructions that occur before the point_of_no_return instruction in the release upgrade script.

Returns the same as install_release/1. Descr defaults to "" if no relup file is found.

If option purge is specified, all old code that can be soft-purged is purged after all other checks are successfully completed. This can be useful to reduce the time needed by install_release/1.

```
create_RELEASES(RelDir, RelFile, AppDirs) -> ok | {error, Reason}
create_RELEASES(Root, RelDir, RelFile, AppDirs) -> ok | {error, Reason}
```

Types:

```
Root = RelDir = RelFile = string()
AppDirs = [{App, Vsn, Dir}]
 App = atom()
 Vsn = Dir = string()
Reason = term()
```

Creates an initial RELEASES file to be used by the release handler. This file must exist to install new releases.

Root is the root of the installation ($ROOT) as described earlier. RelDir is the directory where the RELEASES file is to be created (normally $ROOT/releases). RelFile is the name of the .rel file that describes the initial release, including the extension .rel. If Root is not given, the RELEASES file will be location independent (i.e, it will not contain absolute paths unless there are absolute paths in AppDirs). A RELEASES file should be made location independent if the installation's $ROOT is unknown. The release_handler module will interpret relative paths in a running system's RELEASES file as being relative to $ROOT.

AppDirs can be used to specify from where the modules for the specified applications are to be loaded. App is the name of an application, Vsn is the version, and Dir is the name of the directory where App-Vsn is located. The corresponding modules are to be located under Dir/App-Vsn/ebin. The directories for applications not specified in AppDirs are assumed to be located in $ROOT/lib.

```
install_file(Vsn, File) -> ok | {error, Reason}
```

Types:

```
Vsn = File = string()
Reason = term()
```

Installs a release-dependent file in the release structure. The release-dependent file must be in the release structure when a new release is installed: start.boot, relup, and sys.config.

The function can be called, for example, when these files are generated at the target. The function is to be called after set_unpacked/2 has been called.

```
install_release(Vsn) -> {ok, OtherVsn, Descr} | {error, Reason}
install_release(Vsn, [Opt]) -> {ok, OtherVsn, Descr} |
{continue_after_restart, OtherVsn, Descr} | {error, Reason}
```

Types:

```
Vsn = OtherVsn = string()
Opt = {error_action, Action} | {code_change_timeout, Timeout}
   | {suspend_timeout, Timeout} | {update_paths, Bool}
 Action = restart | reboot
```

```
 Timeout = default | infinity | pos_integer()
 Bool = boolean()
Descr = term()
Reason = {illegal_option, Opt} | {already_installed, Vsn} |
{change_appl_data, term()} | {missing_base_app, OtherVsn, App} |
{could_not_create_hybrid_boot, term()} | term()
App = atom()
```

Installs the specified version `Vsn` of the release. Looks first for a `relup` file for `Vsn` and a script `{UpFromVsn,Descr1,Instructions1}` in this file for upgrading from the current version. If not found, the function looks for a `relup` file for the current version and a script `{Vsn,Descr2,Instructions2}` in this file for downgrading to `Vsn`.

If a script is found, the first thing that happens is that the application specifications are updated according to the `.app` files and `sys.config` belonging to the release version `Vsn`.

After the application specifications have been updated, the instructions in the script are evaluated and the function returns `{ok,OtherVsn,Descr}` if successful. `OtherVsn` and `Descr` are the version (`UpFromVsn` or `Vsn`) and description (`Descr1` or `Descr2`) as specified in the script.

If `{continue_after_restart,OtherVsn,Descr}` is returned, the emulator is restarted before the upgrade instructions are executed. This occurs if the emulator or any of the applications Kernel, STDLIB, or SASL are updated. The new emulator version and these core applications execute after the restart. For all other applications the old versions are started and the upgrade is performed as normal by executing the upgrade instructions.

If a recoverable error occurs, the function returns `{error,Reason}` and the original application specifications are restored. If a non-recoverable error occurs, the system is restarted.

Options:

`error_action`

Defines if the node is to be restarted (`init:restart()`) or rebooted (`init:reboot()`) if there is an error during the installation. Default is `restart`.

`code_change_timeout`

Defines the time-out for all calls to `sys:change_code`. If no value is specified or `default` is specified, the default value defined in `sys` is used.

`suspend_timeout`

Defines the time-out for all calls to `sys:suspend`. If no value is specified, the values defined by the `Timeout` parameter of the `upgrade` or `suspend` instructions are used. If `default` is specified, the default value defined in `sys` is used.

`{update_paths,Bool}`

Indicates if all application code paths are to be updated (`Bool==true`) or if only code paths for modified applications are to be updated (`Bool==false`, default). This option has only effect for other application directories than the default `$ROOT/lib/App-Vsn`, that is, application directories specified in argument `AppDirs` in a call to `create_RELEASES/4` or `set_unpacked/2`.

Example:

In the current version `CurVsn` of a release, the application directory of `myapp` is `$ROOT/lib/myapp-1.0`. A new version `NewVsn` is unpacked outside the release handler and the release handler is informed about this with a call as follows:

```
release_handler:set_unpacked(RelFile, [{myapp,"1.0","/home/user"},...]).
=> {ok,NewVsn}
```

If `NewVsn` is installed with option `{update_paths,true}`, then `code:lib_dir(myapp)` returns `/home/user/myapp-1.0`.

Note:

Installing a new release can be time consuming if there are many processes in the system. The reason is that each process must be checked for references to old code before a module can be purged. This check can lead to garbage collections and copying of data.

To speed up the execution of `install_release`, first call `check_install_release`, using option `purge`. This does the same check for old code. Then purges all modules that can be soft-purged. The purged modules do then no longer have any old code, and `install_release` does not need to do the checks.

This does not reduce the overall time for the upgrade, but it allows checks and purge to be executed in the background before the real upgrade is started.

Note:

When upgrading the emulator from a version older than OTP R15, an attempt is made to load new application beam code into the old emulator. Sometimes the new beam format cannot be read by the old emulator, so the code loading fails and the complete upgrade is terminated. To overcome this problem, the new application code is to be compiled with the old emulator. For more information about emulator upgrade from pre OTP R15 versions, see Design Principles in **System Documentation**.

```
make_permanent(Vsn) -> ok | {error, Reason}
```

Types:

```
Vsn = string()
Reason = {bad_status, Status} | term()
```

Makes the specified release version `Vsn` permanent.

```
remove_release(Vsn) -> ok | {error, Reason}
```

Types:

```
Vsn = string()
Reason = {permanent, Vsn} | client_node | term()
```

Removes a release and its files from the system. The release must not be the permanent release. Removes only the files and directories not in use by another release.

```
reboot_old_release(Vsn) -> ok | {error, Reason}
```

Types:

```
Vsn = string()
Reason = {bad_status, Status} | term()
```

Reboots the system by making the old release permanent, and calls `init:reboot()` directly. The release must have status `old`.

```
set_removed(Vsn) -> ok | {error, Reason}
```

Types:

```
Vsn = string()
```

Reason = {permanent, Vsn} | term()

Makes it possible to handle removal of releases outside the release handler. Tells the release handler that the release is removed from the system. This function does not delete any files.

`set_unpacked(RelFile, AppDirs) -> {ok, Vsn} | {error, Reason}`

Types:

RelFile = string()
AppDirs = [{App, Vsn, Dir}]
App = atom()
Vsn = Dir = string()
Reason = term()

Makes it possible to handle unpacking of releases outside the release handler. Tells the release handler that the release is unpacked. `Vsn` is extracted from the release resource file `RelFile`.

`AppDirs` can be used to specify from where the modules for the specified applications are to be loaded. `App` is the name of an application, `Vsn` is the version, and `Dir` is the name of the directory where `App-Vsn` is located. The corresponding modules are to be located under `Dir/App-Vsn/ebin`. The directories for applications not specified in `AppDirs` are assumed to be located in `$ROOT/lib`.

`unpack_release(Name) -> {ok, Vsn} | {error, Reason}`

Types:

Name = Vsn = string()
Reason = client_node | term()

Unpacks a release package `Name.tar.gz` located in the `releases` directory.

Performs some checks on the package, for example, checks that all mandatory files are present, and extracts its contents.

`which_releases() -> [{Name, Vsn, Apps, Status}]`

Types:

Name = Vsn = string()
Apps = ["App-Vsn"]
Status = unpacked | current | permanent | old

Returns all releases known to the release handler.

`which_releases(Status) -> [{Name, Vsn, Apps, Status}]`

Types:

Name = Vsn = string()
Apps = ["App-Vsn"]
Status = unpacked | current | permanent | old

Returns all releases, known to the release handler, of a specific status.

The following functions can be used to test upgrade and downgrade of single applications (instead of upgrading/downgrading an entire release). A script corresponding to the instructions in the `relup` file is created on-the-fly, based on the `.appup` file for the application, and evaluated exactly in the same way as `release_handler` does.

Warning:

These functions are primarily intended for simplified testing of `.appup` files. They are not run within the context of the `release_handler` process. They must therefore **not** be used together with calls to `install_release/1,2`, as this causes the `release_handler` to end up in an inconsistent state.

No persistent information is updated, so these functions can be used on any Erlang node, embedded or not. Also, using these functions does not affect which code is loaded if there is a reboot.

If the upgrade or downgrade fails, the application can end up in an inconsistent state.

Exports

`upgrade_app(App, Dir) -> {ok, Unpurged} | restart_emulator | {error, Reason}`

Types:

```
App = atom()
Dir = string()
Unpurged = [Module]
 Module = atom()
Reason = term()
```

Upgrades an application `App` from the current version to a new version located in `Dir` according to the `.appup` file.

`App` is the name of the application, which must be started. `Dir` is the new library directory of `App`. The corresponding modules as well as the `.app` and `.appup` files are to be located under `Dir/ebin`.

The function looks in the `.appup` file and tries to find an upgrade script from the current version of the application using `upgrade_script/2`. This script is evaluated using `eval_appup_script/4`, exactly in the same way as `install_release/1,2` does.

Returns one of the following:

- `{ok, Unpurged}` if evaluating the script is successful, where `Unpurged` is a list of unpurged modules
- `restart_emulator` if this instruction is encountered in the script
- `{error, Reason}` if an error occurred when finding or evaluating the script

If the `restart_new_emulator` instruction is found in the script, `upgrade_app/2` returns `{error,restart_new_emulator}`. This because `restart_new_emulator` requires a new version of the emulator to be started before the rest of the upgrade instructions can be executed, and this can only be done by `install_release/1,2`.

`downgrade_app(App, Dir) ->`
`downgrade_app(App, OldVsn, Dir) -> {ok, Unpurged} | restart_emulator | {error, Reason}`

Types:

```
App = atom()
Dir = OldVsn = string()
Unpurged = [Module]
 Module = atom()
Reason = term()
```

Downgrades an application App from the current version to a previous version OldVsn located in Dir according to the .appup file.

App is the name of the application, which must be started. OldVsn is the previous application version and can be omitted if Dir is of the format "App-OldVsn". Dir is the library directory of the previous version of App. The corresponding modules and the old .app file are to be located under Dir/ebin. The .appup file is to be located in the ebin directory of the **current** library directory of the application (code:lib_dir(App)).

The function looks in the .appup file and tries to find a downgrade script to the previous version of the application using downgrade_script/3. This script is evaluated using eval_appup_script/4, exactly in the same way as install_release/1,2 does.

Returns one of the following:

- {ok, Unpurged} if evaluating the script is successful, where Unpurged is a list of unpurged modules
- restart_emulator if this instruction is encountered in the script
- {error, Reason} if an error occurred when finding or evaluating the script

```
upgrade_script(App, Dir) -> {ok, NewVsn, Script}
```

Types:

App = atom()
Dir = string()
NewVsn = string()
Script = Instructions

Tries to find an application upgrade script for App from the current version to a new version located in Dir.

The upgrade script can then be evaluated using eval_appup_script/4. It is recommended to use upgrade_app/2 instead, but this function (upgrade_script) is useful to inspect the contents of the script.

App is the name of the application, which must be started. Dir is the new library directory of App. The corresponding modules as well as the .app and .appup files are to be located under Dir/ebin.

The function looks in the .appup file and tries to find an upgrade script from the current application version. High-level instructions are translated to low-level instructions. The instructions are sorted in the same manner as when generating a relup file.

Returns {ok, NewVsn, Script} if successful, where NewVsn is the new application version. For details about Script, see appup(4).

Failure: If a script cannot be found, the function fails with an appropriate error reason.

```
downgrade_script(App, OldVsn, Dir) -> {ok, Script}
```

Types:

App = atom()
OldVsn = Dir = string()
Script = Instructions

Tries to find an application downgrade script for App from the current version to a previous version OldVsn located in Dir.

The downgrade script can then be evaluated using eval_appup_script/4. It is recommended to use downgrade_app/2,3 instead, but this function (downgrade_script) is useful to inspect the contents of the script.

App is the name of the application, which must be started. Dir is the previous library directory of App. The corresponding modules and the old .app file are to be located under Dir/ebin. The .appup file is to be located in the ebin directory of the **current** library directory of the application (code:lib_dir(App)).

The function looks in the .appup file and tries to find a downgrade script from the current application version. High-level instructions are translated to low-level instructions. The instructions are sorted in the same manner as when generating a relup file.

Returns {ok, Script} if successful. For details about Script, see appup(4).

Failure: If a script cannot be found, the function fails with an appropriate error reason.

```
eval_appup_script(App, ToVsn, ToDir, Script) -> {ok, Unpurged} |
restart_emulator | {error, Reason}
```

Types:

App = atom()

ToVsn = ToDir = string()

Script

See upgrade_script/2, downgrade_script/3

Unpurged = [Module]

Module = atom()

Reason = term()

Evaluates an application upgrade or downgrade script Script, the result from calling upgrade_script/2 or downgrade_script/3, exactly in the same way as install_release/1,2 does.

App is the name of the application, which must be started. ToVsn is the version to be upgraded/downgraded to, and ToDir is the library directory of this version. The corresponding modules as well as the .app and .appup files are to be located under Dir/ebin.

Returns one of the following:

- {ok, Unpurged} if evaluating the script is successful, where Unpurged is a list of unpurged modules
- restart_emulator if this instruction is encountered in the script
- {error, Reason} if an error occurred when finding or evaluating the script

If the restart_new_emulator instruction is found in the script, eval_appup_script/4 returns {error,restart_new_emulator}. This because restart_new_emulator requires a new version of the emulator to be started before the rest of the upgrade instructions can be executed, and this can only be done by install_release/1,2.

Typical Error Reasons

{bad_masters, Masters}

The master nodes Masters are not alive.

{bad_rel_file, File}

Specified .rel file File cannot be read or does not contain a single term.

{bad_rel_data, Data}

Specified .rel file does not contain a recognized release specification, but another term Data.

{bad_relup_file, File}

Specified relup file Relup contains bad data.

`{cannot_extract_file, Name, Reason}`

Problems when extracting from a tar file, `erl_tar:extract/2` returned `{error, {Name, Reason}}`.

`{existing_release, Vsn}`

Specified release version `Vsn` is already in use.

`{Master, Reason, When}`

Some operation, indicated by the term `When`, failed on the master node `Master` with the specified error reason `Reason`.

`{no_matching_relup, Vsn, CurrentVsn}`

Cannot find a script for upgrading/downgrading between `CurrentVsn` and `Vsn`.

`{no_such_directory, Path}`

The directory `Path`does not exist.

`{no_such_file, Path}`

The path `Path` (file or directory) does not exist.

`{no_such_file, {Master, Path}}`

The path `Path` (file or directory) does not exist at the master node `Master`.

`{no_such_release, Vsn}`

The specified release version `Vsn` does not exist.

`{not_a_directory, Path}`

`Path` exists but is not a directory.

`{Posix, File}`

Some file operation failed for `File`. `Posix` is an atom named from the Posix error codes, such as `enoent`, `eacces`, or `eisdir`. See `file(3)` in Kernel.

`Posix`

Some file operation failed, as for the previous item in the list.

See Also

OTP Design Principles, `config(4)`, `rel(4)`, `relup(4)`, `script(4)`, `sys(3)`, `systools(3)`

systools

Erlang module

This module contains functions to generate boot scripts (`.boot`, `.script`), a release upgrade file (`relup`), and release packages.

Exports

```
make_relup(Name, UpFrom, DownTo) -> Result
make_relup(Name, UpFrom, DownTo, [Opt]) -> Result
```

Types:

```
Name = string()
UpFrom = DownTo = [Name | {Name,Descr}]
 Descr = term()
Opt = {path,[Dir]} | restart_emulator | silent | noexec | {outdir,Dir} |
warnings_as_errors
 Dir = string()
Result = ok | error | {ok,Relup,Module,Warnings} | {error,Module,Error}
 Relup, see relup(4)
 Module = atom()
 Warnings = Error = term()
```

Generates a release upgrade file `relup` containing instructions for upgrading from or downgrading to one or more previous releases. The instructions are used by `release_handler` when installing a new version of a release in runtime.

By default, `relup` file is located in the current working directory. If option `{outdir,Dir}` is specified, the `relup` file is located in `Dir` instead.

The release resource file `Name.rel` is compared with all release resource files `Name2.rel`, specified in `UpFrom` and `DownTo`. For each such pair, the following is deducted:

- Which applications to be deleted, that is, applications listed in `Name.rel` but not in `Name2.rel`
- Which applications to be added, that is, applications listed in `Name2.rel` but not in `Name.rel`
- Which applications to be upgraded/downgraded, that is, applications listed in both `Name.rel` and `Name2.rel` but with different versions
- If the emulator needs to be restarted after upgrading or downgrading, that is, if the ERTS version differs between `Name.rel` and `Name2.rel`

Instructions for this are added to the `relup` file in the above order. Instructions for upgrading or downgrading between application versions are fetched from the relevant application upgrade files `App.appup`, sorted in the same order as when generating a boot script, see `make_script/1,2`. High-level instructions are translated into low-level instructions and the result is printed to the `relup` file.

The optional `Descr` parameter is included "as is" in the `relup` file, see `relup(4)`. Defaults to the empty list.

All the files are searched for in the code path. It is assumed that the `.app` and `.appup` files for an application are located in the same directory.

If option `{path,[Dir]}` is specified, this path is appended to the current path. Wildcard `*` is expanded to all matching directories, for example, `lib/*/ebin`.

If option `restart_emulator` is specified, a low-level instruction to restart the emulator is appended to the `relup` file. This ensures that a complete reboot of the system is done when the system is upgraded or downgraded.

If an upgrade includes a change from an emulator earlier than OTP R15 to OTP R15 or later, the warning `pre_R15_emulator_upgrade` is issued. For more information about this, see Design Principles in **System Documentation**.

By default, errors and warnings are printed to tty and the function returns `ok` or `error`. If option `silent` is specified, the function instead either returns `{ok,Relup,Module,Warnings}`, where `Relup` is the release upgrade file, or `{error,Module,Error}`. Warnings and errors can be converted to strings by calling `Module:format_warning(Warnings)` or `Module:format_error(Error)`.

If option `noexec` is specified, the function returns the same values as for `silent` but no `relup` file is created.

If option `warnings_as_errors` is specified, warnings are treated as errors.

```
make_script(Name) -> Result
make_script(Name, [Opt]) -> Result
```

Types:

```
Name = string()
Opt = src_tests | {path,[Dir]} | local | {variables,[Var]} | exref |
{exref,[App]}] | silent | {outdir,Dir} | no_dot_erlang | no_warn_sasl |
warnings_as_errors | {script_name, Name}
 Dir = string()
 Var = {VarName,Prefix}
  VarName = Prefix = string()
 App = atom()
Result = ok | error | {ok,Module,Warnings} | {error,Module,Error}
 Module = atom()
 Warnings = Error = term()
```

Generates a boot script `Name.script` and its binary version, the boot file `Name.boot`, unless the `{script_name, ScriptName}` option is given, in which case the names are `ScriptName.script` and `ScriptName.boot` The boot file specifies which code to be loaded and which applications to be started when the Erlang runtime system is started. See `script(4)`.

The release resource file `Name.rel` is read to determine which applications are included in the release. Then the relevant application resource files `App.app` are read to determine which modules to be loaded, and if and how the applications are to be started. (Keys `modules` and `mod`, see `app(4)`.

By default, the boot script and boot file are located in the same directory as `Name.rel`. That is, in the current working directory unless `Name` contains a path. If option `{outdir,Dir}` is specified, they are located in `Dir` instead.

The correctness of each application is checked as follows:

- The version of an application specified in the `.rel` file is to be the same as the version specified in the `.app` file.
- There are to be no undefined applications, that is, dependencies to applications that are not included in the release. (Key `applications` in the `.app` file).
- There are to be no circular dependencies among the applications.
- There are to be no duplicated modules, that is, modules with the same name but belonging to different applications.
- If option `src_tests` is specified, a warning is issued if the source code for a module is missing or is newer than the object code.

The applications are sorted according to the dependencies between the applications. Where there are no dependencies, the order in the `.rel` file is kept.

The function fails if the mandatory applications Kernel and STDLIB are not included in the `.rel` file and have start type `permanent` (which is default).

If SASL is not included as an application in the `.rel` file, a warning is issued because such a release cannot be used in an upgrade. To turn off this warning, add option `no_warn_sasl`.

All files are searched for in the current path. It is assumed that the `.app` and `.beam` files for an application are located in the same directory. The `.erl` files are also assumed to be located in this directory, unless it is an `ebin` directory in which case they can be located in the corresponding `src` directory.

If option `{path,[Dir]}` is specified, this path is appended to the current path. A directory in the path can be specified with a wildcard `*`, this is expanded to all matching directories. Example: `"lib/*/ebin"`.

In the generated boot script all application directories are structured as `App-Vsn/ebin`. They are assumed to be located in `$ROOT/lib`, where `$ROOT` is the root directory of the installed release. If option `local` is specified, the actual directories where the applications were found are used instead. This is a useful way to test a generated boot script locally.

Option `variables` can be used to specify an installation directory other than `$ROOT/lib` for some of the applications. If a variable `{VarName,Prefix}` is specified and an application is found in a directory `Prefix/Rest/App[-Vsn]/ebin`, this application gets the path `VarName/Rest/App-Vsn/ebin` in the boot script. If an application is found in a directory `Prefix/Rest`, the path is `VarName/Rest/App-Vsn/ebin`. When starting Erlang, all variables `VarName` are given values using command-line flag `boot_var`.

Example: If option `{variables,[{"TEST","lib"}]}` is specified and `myapp.app` is found in `lib/myapp/ebin`, the path to this application in the boot script is `"$TEST/myapp-1/ebin"`. If `myapp.app` is found in `lib/test`, the path is `$TEST/test/myapp-1/ebin`.

The checks performed before the boot script is generated can be extended with some cross reference checks by specifying option `exref`. These checks are performed with the Xref tool. All applications, or the applications specified with `{exref,[App]}`, are checked by Xref and warnings are issued for calls to undefined functions.

By default, errors and warnings are printed to tty and the function returns `ok` or `error`. If option `silent` is specified, the function instead returns `{ok,Module,Warnings}` or `{error,Module,Error}`. Warnings and errors can be converted to strings by calling `Module:format_warning(Warnings)` or `Module:format_error(Error)`.

If option `warnings_as_errors` is specified, warnings are treated as errors.

If option `no_dot_erlang` is specified, the instruction to load the `.erlang` file during boot is **not** included.

```
make_tar(Name) -> Result
make_tar(Name, Opts) -> Result
```

Types:

```
   Name = string()
   Opts = [Opt]
   Opt =
       {dirs, [IncDir]} |
       {path, [Dir]} |
       {variables, [Var]} |
       {var_tar, VarTar} |
       {erts, Dir} |
       erts_all | src_tests | exref |
       {exref, [App]} |
```

```
      silent |
      {outdir, Dir} |
      no_warn_sasl | warnings_as_errors |
      {extra_files, ExtraFiles}
  Dir = file:filename_all()
  IncDir = src | include | atom()
  Var = {VarName, PreFix}
  VarName = PreFix = string()
  VarTar = include | ownfile | omit
  App = atom()
  Result =
      ok | error |
      {ok, Module :: module(), Warnings :: term()} |
      {error, Module :: module(), Error :: term()}
  ExtraFiles = [{NameInArchive, file:filename_all()}]
  NameInArchive = string()
```

Creates a release package file `Name.tar.gz`. This file must be uncompressed and unpacked on the target system using `release_handler` before the new release can be installed.

The release resource file `Name.rel` is read to determine which applications are included in the release. Then the relevant application resource files `App.app` are read to determine the version and modules of each application (keys `vsn` and `modules`, see `app(4)`).

By default, the release package file is located in the same directory as `Name.rel`. That is, in the current working directory unless `Name` contains a path. If option `{outdir,Dir}` is specified, it is located in `Dir` instead.

If SASL is not included as an application in the `.rel` file, a warning is issued because such a release cannot be used in an upgrade. To turn off this warning, add option `no_warn_sasl`.

By default, the release package contains the directories `lib/App-Vsn/ebin` and `lib/App-Vsn/priv` for each included application. If more directories are to be included, option `dirs` is specified, for example, `{dirs, [src,examples]}`.

All these files are searched for in the current path. If option `{path,[Dir]}` is specified, this path is appended to the current path. Wildcard `*` is expanded to all matching directories. Example: `"lib/*/ebin"`.

If the `{extra_files, ExtraFiles}` option is given then the `ExtraFiles` are added to the tarball after everything else to be included has been added. The `ExtraFiles` list is a list of files or directories in the same format as the `add_type()` tuple for erl_tar:add/3,4

Option `variables` can be used to specify an installation directory other than `lib` for some of the applications. If variable `{VarName,Prefix}` is specified and an application is found in directory `Prefix/Rest/App[-Vsn]/ebin`, this application is packed into a separate `VarName.tar.gz` file as `Rest/App-Vsn/ebin`.

Example: If option `{variables,[{"TEST","lib"}]}` is specified and `myapp.app` is located in `lib/myapp-1/ebin`, application `myapp` is included in `TEST.tar.gz`:

```
% tar tf TEST.tar
myapp-1/ebin/myapp.app
...
```

Option `{var_tar,VarTar}` can be used to specify if and where a separate package is to be stored. In this option `VarTar` is one of the following:

`include`

Each separate (variable) package is included in the main `ReleaseName.tar.gz` file. This is the default.

`ownfile`

Each separate (variable) package is generated as a separate file in the same directory as the `ReleaseName.tar.gz` file.

`omit`

No separate (variable) packages are generated. Applications that are found underneath a variable directory are ignored.

A directory `releases` is also included in the release package, containing `Name.rel` and a subdirectory `RelVsn`. `RelVsn` is the release version as specified in `Name.rel`.

`releases/RelVsn` contains the boot script `Name.boot` renamed to `start.boot` and, if found, the files `relup` and `sys.config` or `sys.config.src`. These files are searched for in the same directory as `Name.rel`, in the current working directory, and in any directories specified using option `path`. In the case of `sys.config` it is not included if `sys.config.src` is found.

If the release package is to contain a new Erlang runtime system, the `erts-ErtsVsn/bin` directory of the specified runtime system `{erts,Dir}` is copied to `erts-ErtsVsn/bin`. Some erts executables are not copied by default, if you want to include all executables you can give the `erts_all` option.

All checks with function `make_script` are performed before the release package is created. Options `src_tests` and `exref` are also valid here.

The return value and the handling of errors and warnings are the same as described for `make_script`.

`script2boot(File) -> ok | error`

Types:

`File = string()`

The Erlang runtime system requires that the contents of the script used to boot the system is a binary Erlang term. This function transforms the `File.script` boot script to a binary term, which is stored in the `File.boot` file.

A boot script generated using `make_script` is already transformed to the binary form.

See Also

`app(4)`, `appup(4)`, `erl(1)`, `rel(4)`, `release_handler(3)`, `relup(4)`, `script(4)`

appup

Name

The **application upgrade file** defines how an application is upgraded or downgraded in a running system.

This file is used by the functions in `systools` when generating a release upgrade file `relup`.

File Syntax

The application upgrade file is to be called `Application.appup`, where `Application` is the application name. The file is to be located in the `ebin` directory for the application.

The `.appup` file contains one single Erlang term, which defines the instructions used to upgrade or downgrade the application. The file has the following syntax:

```
{Vsn,
  [{UpFromVsn, Instructions}, ...],
  [{DownToVsn, Instructions}, ...]}.
```

`Vsn = string()`

Current application version.

`UpFromVsn = string() | binary()`

An earlier application version to upgrade from. If it is a string, it is interpreted as a specific version number. If it is a binary, it is interpreted as a regular expression that can match multiple version numbers.

`DownToVsn = string() | binary()`

An earlier application version to downgrade to. If it is a string, it is interpreted as a specific version number. If it is a binary, it is interpreted as a regular expression that can match multiple version numbers.

`Instructions`

A list of **release upgrade instructions**, see Release Upgrade Instructions. It is recommended to use high-level instructions only. These are automatically translated to low-level instructions by `systools` when creating the `relup` file.

To avoid duplication of upgrade instructions, it is allowed to use regular expressions to specify `UpFromVsn` and `DownToVsn`. To be considered a regular expression, the version identifier must be specified as a binary. For example, the following match all versions `2.1.x`, where `x` is any number:

```
<<"2\\.1\\.[0-9]+">>
```

Notice that the regular expression must match the complete version string, so this example works for, for example, `2.1.1`, but not for `2.1.1.1`.

Release Upgrade Instructions

Release upgrade instructions are interpreted by the release handler when an upgrade or downgrade is made. For more information about release handling, see OTP Design Principles in **System Documentation**.

A process is said to **use** a module `Mod` if `Mod` is listed in the `Modules` part of the child specification used to start the process, see `supervisor(3)`. In the case of `gen_event`, an event manager process is said to use `Mod` if `Mod` is an installed event handler.

High-Level Instructions

```
{update, Mod}
{update, Mod, supervisor}
{update, Mod, Change}
{update, Mod, DepMods}
{update, Mod, Change, DepMods}
{update, Mod, Change, PrePurge, PostPurge, DepMods}
{update, Mod, Timeout, Change, PrePurge, PostPurge, DepMods}
{update, Mod, ModType, Timeout, Change, PrePurge, PostPurge, DepMods}
  Mod = atom()
  ModType = static | dynamic
  Timeout = int()>0 | default | infinity
  Change = soft | {advanced,Extra}
    Extra = term()
  PrePurge = PostPurge = soft_purge | brutal_purge
  DepMods = [Mod]
```

Synchronized code replacement of processes using module `Mod`.

All those processes are suspended using `sys:suspend`, the new module version is loaded, and then the processes are resumed using `sys:resume`.

`Change`

Defaults to `soft` and defines the type of code change. If it is set to `{advanced,Extra}`, implemented processes using `gen_server`, `gen_fsm`, `gen_statem`, or `gen_event` transform their internal state by calling the callback function `code_change`. Special processes call the callback function `system_code_change/4`. In both cases, the term `Extra` is passed as an argument to the callback function.

`PrePurge`

Defaults to `brutal_purge`. It controls what action to take with processes executing old code before loading the new module version. If the value is `brutal_purge`, the processes are killed. If the value is `soft_purge`, `release_handler:install_release/1` returns `{error,{old_processes,Mod}}`.

`PostPurge`

Defaults to `brutal_purge`. It controls what action to take with processes that are executing old code when the new module version has been loaded. If the value is `brutal_purge`, the code is purged when the release is made permanent and the processes are killed. If the value is `soft_purge`, the release handler purges the old code when no remaining processes execute the code.

`DepMods`

Defaults to `[]` and defines other modules that `Mod` is dependent on. In the `relup` file, instructions for suspending processes using `Mod` come before instructions for suspending processes using modules in `DepMods` when upgrading, and conversely when downgrading. In case of circular dependencies, the order of the instructions in the `appup` file is kept.

`Timeout`

Defines the time-out when suspending processes. If no value or `default` is specified, the default value for `sys:suspend` is used.

`ModType`

Defaults to `dynamic`. It specifies if the code is "dynamic", that is, if a process using the module spontaneously switches to new code, or if it is "static". When doing an advanced update and upgrade, the new version of a dynamic module is loaded before the process is asked to change code. When downgrading, the process is asked to change code before loading the new version. For static modules, the new version is loaded before the process is asked to change code, both in the case of upgrading and downgrading. Callback modules are dynamic.

`update` with argument `supervisor` is used when changing the start specification of a supervisor.

```
{load_module, Mod}
{load_module, Mod, DepMods}
{load_module, Mod, PrePurge, PostPurge, DepMods}
  Mod = atom()
  PrePurge = PostPurge = soft_purge | brutal_purge
  DepMods = [Mod]
```

Simple code replacement of the module `Mod`.

For a description of `PrePurge` and `PostPurge`, see `update` above.

`DepMods` defaults to `[]` and defines which other modules `Mod` is dependent on. In the `relup` file, instructions for loading these modules come before the instruction for loading `Mod` when upgrading, and conversely when downgrading.

```
{add_module, Mod}
{add_module, Mod, DepMods}
  Mod = atom()
  DepMods = [Mod]
```

Loads a new module `Mod`.

`DepMods` defaults to `[]` and defines which other modules `Mod` is dependent on. In the `relup` file, instructions related to these modules come before the instruction for loading `Mod` when upgrading, and conversely when downgrading.

```
{delete_module, Mod}
{delete_module, Mod, DepMods}
  Mod = atom()
```

Deletes a module `Mod` using the low-level instructions `remove` and `purge`.

`DepMods` defaults to `[]` and defines which other modules `Mod` is dependent on. In the `relup` file, instructions related to these modules come before the instruction for removing `Mod` when upgrading, and conversely when downgrading.

```
{add_application, Application}
{add_application, Application, Type}
  Application = atom()
  Type = permanent | transient | temporary | load | none
```

Adding an application means that the modules defined by the `modules` key in the `.app` file are loaded using `add_module`.

`Type` defaults to `permanent` and specifies the start type of the application. If `Type = permanent | transient | temporary`, the application is loaded and started in the corresponding way, see `application(3)`. If `Type = load`, the application is only loaded. If `Type = none`, the application is not loaded and not started, although the code for its modules is loaded.

```
{remove_application, Application}
  Application = atom()
```

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

```
{restart_application, Application}
  Application = atom()
```

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. Note that, even if the application has been started before the release upgrade is performed, `restart_application` may only `load` it rather than `start` it, depending on the application's `start type`: If `Type = load`, the application is only loaded. If `Type = none`, the application is not loaded and not started, although the code for its modules is loaded.

Low-Level Instructions

```
{load_object_code, {App, Vsn, [Mod]}}
  App = Mod = atom()
  Vsn = string()
```

Reads each `Mod` from directory `App-Vsn/ebin` as a binary. It does not load the modules. The instruction is to be placed first in the script to read all new code from the file to make the suspend-load-resume cycle less time-consuming.

```
point_of_no_return
```

If a crash occurs after this instruction, the system cannot recover and is restarted from the old release version. The instruction must only occur once in a script. It is to be placed after all `load_object_code` instructions.

```
{load, {Mod, PrePurge, PostPurge}}
  Mod = atom()
  PrePurge = PostPurge = soft_purge | brutal_purge
```

Before this instruction occurs, `Mod` must have been loaded using `load_object_code`. This instruction loads the module. `PrePurge` is ignored. For a description of `PostPurge`, see the high-level instruction `update` earlier.

```
{remove, {Mod, PrePurge, PostPurge}}
  Mod = atom()
  PrePurge = PostPurge = soft_purge | brutal_purge
```

Makes the current version of `Mod` old. `PrePurge` is ignored. For a description of `PostPurge`, see the high-level instruction `update` earlier.

```
{purge, [Mod]}
  Mod = atom()
```

Purges each module `Mod`, that is, removes the old code. Notice that any process executing purged code is killed.

```
{suspend, [Mod | {Mod, Timeout}]}
  Mod = atom()
  Timeout = int()>0 | default | infinity
```

Tries to suspend all processes using a module `Mod`. If a process does not respond, it is ignored. This can cause the process to die, either because it crashes when it spontaneously switches to new code, or as a result of a purge operation. If no `Timeout` is specified or `default` is specified, the default value for `sys:suspend` is used.

```
{resume, [Mod]}
  Mod = atom()
```

Resumes all suspended processes using a module `Mod`.

```
{code_change, [{Mod, Extra}]}
{code_change, Mode, [{Mod, Extra}]}
  Mod = atom()
  Mode = up | down
  Extra = term()
```

`Mode` defaults to `up` and specifies if it is an upgrade or downgrade. This instruction sends a `code_change` system message to all processes using a module `Mod` by calling function `sys:change_code`, passing term `Extra` as argument.

```
{stop, [Mod]}
  Mod = atom()
```

Stops all processes using a module `Mod` by calling `supervisor:terminate_child/2`. This instruction is useful when the simplest way to change code is to stop and restart the processes that run the code.

```
{start, [Mod]}
  Mod = atom()
```

Starts all stopped processes using a module `Mod` by calling `supervisor:restart_child/2`.

```
{sync_nodes, Id, [Node]}
{sync_nodes, Id, {M, F, A}}
  Id = term()
  Node = node()
  M = F = atom()
  A = [term()]
```

`apply(M, F, A)` must return a list of nodes.

This instruction synchronizes the release installation with other nodes. Each `Node` must evaluate this command with the same `Id`. The local node waits for all other nodes to evaluate the instruction before execution continues. If a node goes down, it is considered to be an unrecoverable error, and the local node is restarted from the old release. There is no time-out for this instruction, which means that it can hang forever.

```
{apply, {M, F, A}}
  M = F = atom()
  A = [term()]
```

Evaluates `apply(M, F, A)`.

If the instruction appears before instruction `point_of_no_return`, a failure is caught. `release_handler:install_release/1` then returns `{error,{'EXIT',Reason}}`, unless `{error,Error}` is thrown or returned. Then it returns `{error,Error}`.

If the instruction appears after instruction `point_of_no_return` and the function call fails, the system is restarted.

```
restart_new_emulator
```

This instruction is used when the application ERTS, Kernel, STDLIB, or SASL is upgraded. It shuts down the current emulator and starts a new one. All processes are terminated gracefully, and the new version of ERTS, Kernel, STDLIB, and SASL are used when the emulator restarts. Only one `restart_new_emulator` instruction is allowed in

the `relup` file, and it must be placed first. `systools:make_relup/3,4` ensures this when the `relup` file is generated. The rest of the instructions in the `relup` file is executed after the restart as a part of the boot script.

An info report is written when the upgrade is completed. To programmatically determine if the upgrade is complete, call `release_handler:which_releases/0,1` and check if the expected release has status `current`.

The new release must still be made permanent after the upgrade is completed, otherwise the old emulator is started if there is an emulator restart.

Warning:

As stated earlier, instruction `restart_new_emulator` causes the emulator to be restarted with new versions of ERTS>, Kernel, STDLIB, and SASL. However, all other applications do at startup run their old versions in this new emulator. This is usually no problem, but every now and then incompatible changes occur to the core applications, which can cause trouble in this setting. Such incompatible changes (when functions are removed) are normally preceded by a deprecation over two major releases. To ensure that your application is not crashed by an incompatible change, always remove any call to deprecated functions as soon as possible.

```
restart_emulator
```

This instruction is similar to `restart_new_emulator`, except it must be placed at the end of the `relup` file. It is not related to an upgrade of the emulator or the core applications, but can be used by any application when a complete reboot of the system is required.

When generating the `relup` file, `systools:make_relup/3,4` ensures that there is only one `restart_emulator` instruction and that it is the last instruction in the `relup` file.

See Also

`release_handler(3)`, `relup(4)`, `supervisor(3)`, `systools(3)`

rel

Name

The **release resource file** specifies which applications are included in a release (system) based on Erlang/OTP.

This file is used by the functions in `systools` when generating start scripts (`.script`, `.boot`) and release upgrade files (`relup`).

File Syntax

The release resource file is to be called `Name.rel`.

The `.rel` file contains one single Erlang term, which is called a **release specification**. The file has the following syntax:

```
{release, {RelName,Vsn}, {erts, EVsn},
  [{Application, AppVsn} |
   {Application, AppVsn, Type} |
   {Application, AppVsn, IncApps} |
   {Application, AppVsn, Type, IncApps}]}.
```

`RelName = string()`

Release name.

`Vsn = string()`

Release version.

`EVsn = string()`

ERTS version the release is intended for.

`Application = atom()`

Name of an application included in the release.

`AppVsn = string()`

Version of an application included in the release.

`Type = permanent | transient | temporary | load | none`

Start type of an application included in the release.

If `Type = permanent | transient | temporary`, the application is loaded and started in the corresponding way, see `application(3)`.

If `Type = load`, the application is only loaded.

If `Type = none`, the application is not loaded and not started, although the code for its modules is loaded.

Defaults to `permanent`

`IncApps = [atom()]`

A list of applications that are included by an application included in the release. The list must be a subset of the included applications specified in the application resource file (`Application.app`) and overrides this value. Defaults to the same value as in the application resource file.

> **Note:**
>
> The list of applications must contain the Kernel and STDLIB applications.

See Also

`application(3)`, `relup(4)`, `systools(3)`

relup

Name

The **release upgrade file** describes how a release is upgraded in a running system.

This file is automatically generated by `systools:make_relup/3,4`, using a release resource file (`.rel`), application resource files (`.app`), and application upgrade files (`.appup`) as input.

File Syntax

In a target system, the release upgrade file is to be located in directory `$ROOT/releases/Vsn`.

The `relup` file contains one single Erlang term, which defines the instructions used to upgrade the release. The file has the following syntax:

```
{Vsn,
  [{UpFromVsn, Descr, Instructions}, ...],
  [{DownToVsn, Descr, Instructions}, ...]}.
```

`Vsn = string()`

Current release version.

`UpFromVsn = string()`

Earlier version of the release to upgrade from.

`Descr = term()`

A user-defined parameter passed from the function `systools:make_relup/3,4`. It is used in the return value of `release_handler:install_release/1,2`.

`Instructions`

A list of low-level release upgrade instructions, see `appup(4)`. It consists of the release upgrade instructions from the respective application upgrade files (high-level instructions are translated to low-level instructions), in the same order as in the start script.

`DownToVsn = string()`

Earlier version of the release to downgrade to.

See Also

`app(4)`, `appup(4)`, `rel(4)`, `release_handler(3)`, `systools(3)`

script

Name

The **boot script** describes how the Erlang runtime system is started. It contains instructions on which code to load and which processes and applications to start.

Command `erl -boot Name` starts the system with a boot file called `Name.boot`, which is generated from the `Name.script` file, using `systools:script2boot/1`.

The `.script` file is generated by `systools` from a `.rel` file and from `.app` files.

File Syntax

The boot script is stored in a file with extension `.script`. The file has the following syntax:

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

`Name = string()`

Defines the system name.

`Vsn = string()`

Defines the system version.

`{progress, Term}`

Sets the "progress" of the initialization program. The `init:get_status/0` function returns the current value of the progress, which is `{InternalStatus,Term}`.

`{path, [Dir]}`

`Dir` is a string. This argument sets the load path of the system to `[Dir]`. The load path used to load modules is obtained from the initial load path, which is given in the script file, together with any path flags that were supplied in the command-line arguments. The command-line arguments modify the path as follows:

- `-pa Dir1 Dir2 ... DirN` adds the directories `DirN, DirN-1, ..., Dir2, Dir1` to the front of the initial load path.
- `-pz Dir1 Dir2 ... DirN` adds the directories `Dir1, Dir2, ..., DirN` to the end of the initial load path.
- `-path Dir1 Dir2 ... DirN` defines a set of directories `Dir1, Dir2, ..., DirN`, which replace the search path given in the script file. Directory names in the path are interpreted as follows:
  - Directory names starting with `/` are assumed to be absolute path names.
  - Directory names not starting with `/` are assumed to be relative the current working directory.

- The special `$ROOT` variable can only be used in the script, not as a command-line argument. The given directory is relative the Erlang installation directory.

`{primLoad, [Mod]}`

Loads the modules `[Mod]` from the directories specified in `Path`. The script interpreter fetches the appropriate module by calling `erl_prim_loader:get_file(Mod)`. A fatal error that terminates the system occurs if the module cannot be located.

`{kernel_load_completed}`

Indicates that all modules that **must** be loaded **before** any processes are started are loaded. In interactive mode, all `{primLoad,[Mod]}` commands interpreted after this command are ignored, and these modules are loaded on demand. In embedded mode, `kernel_load_completed` is ignored, and all modules are loaded during system start.

`{kernelProcess, Name, {Mod, Func, Args}}`

Starts the "kernel process" `Name` by evaluating `apply(Mod, Func, Args)`. The start function is to return `{ok, Pid}` or `ignore`. The `init` process monitors the behavior of `Pid` and terminates the system if `Pid` dies. Kernel processes are key components of the runtime system. Users do not normally add new kernel processes.

`{apply, {Mod, Func, Args}}`.

The init process evaluates `apply(Mod, Func, Args)`. The system terminates if this results in an error. The boot procedure hangs if this function never returns.

Note:

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

See Also

`systools(3)`