System Application Support Libraries (SASL)

System Application Support Libraries (SASL) 2.3.1
February 25, 2013

February 25, 2013

1 SASL User's Guide

The System Architecture Support Libraries, *SASL*, provides support for alarm and release handling etc.

1.1 Introduction

1.1.1 About This Document

The SASL (System Architecture Support Libraries) application provides support for:

- error logging
- alarm handling
- overload regulation
- release handling
- report browsing.

In this document, "SASL Error Logging" describes the error handler which 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 chapters about release structure and release handling have been moved to *OTP Design Principles*.

1.2 SASL Error Logging

The SASL application introduces three types of reports:

- supervisor report
- progress report
- crash report.

When the SASL application is started, it adds a handler that formats and writes these reports, as specified in the configuration parameters for sasl, i.e the environment variables in the SASL application specification, which is found in the `.app` file of SASL. See *sasl(Application)*, and app(File) in the Kernel Reference Manual for the details.

1.2.1 Supervisor Report

A supervisor report is issued when a supervised child terminates in an unexpected way. A supervisor report contains the following items:

Supervisor.
: The 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.
: The termination reason.

Offender.
: The start specification for the child.

1.2.2 Progress Report

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

Supervisor.
: The name of the reporting supervisor.

Started.
: The start specification for the successfully started child.

1.2.3 Crash Report

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

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

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

An Example

The following example shows the reports which are generated when a process crashes. The example process is an `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 the `proc_lib:spawn/3` function. The supervisor generates a supervisor report showing the process that has crashed, and then a progress report is generated when the process is finally re-started.

```
=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}]
```

1.2.4 Multi-File Error Report Logging

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

Refer to `sasl` application in the Reference Manual for more details.

1.2.5 Report Browser

The report browser is used to browse and format error reports written by the error logger handler `error_logger_mf_h`.

The `error_logger_mf_h` handler writes all reports to a report logging directory. This directory is specified when configuring the SASL application.

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

Starting the Report Browser

Start the `rb_server` with the 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.
```

On-line Help

Enter the command *rb:help().* to access the report browser on-line help system.

List Reports in the Server

The function `rb:list()` lists 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

To show details of a specific report, use the function `rb:show(Number)`:

```
10> rb:show(1).
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 the Reports

It is possible to show all reports which contain a common pattern. Suppose a process crashes because it tries to call a non-existing function `release_handler:mbj_func.` We could then show reports 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 the Server

Stop the `rb_server` with the function `rb:stop()`:

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

2 Reference Manual

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

sasl

Application

This section describes the SASL (System Architecture Support Libraries) application which provides the following services:

- `alarm_handler`
- `overload`
- `rb`
- `release_handler`
- `systools`

The SASL application also includes `error_logger` event handlers for formatting SASL error and crash reports.

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

Error Logger Event Handlers

The following error logger event handlers are defined in the SASL application.

`sasl_report_tty_h`

Formats and writes *supervisor reports*, *crash reports* and *progress reports* to `stdio`.

`sasl_report_file_h`

Formats and writes *supervisor reports*, *crash report* and *progress report* to a single file.

`error_logger_mf_h`

This error logger writes *all* events sent to the error logger to disk. It installs the `log_mf_h` event handler in the `error_logger` process.

To activate this event handler, the following three sasl configuration parameters must be set: `error_logger_mf_dir`, `error_logger_mf_maxbytes` and `error_logger_mf_maxfiles`. See below for more information about the configuration parameters.

Configuration

The following configuration parameters are defined for the SASL application. See `app(4)` for more information about configuration parameters:

`sasl_error_logger = Value <optional>`

`Value` is one of:

`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. This makes all reports go to the file `FileName`. `FileName` is a string.

`false`

No SASL error logger handler is installed.

`errlog_type = error | progress | all <optional>`

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

`error_logger_mf_dir = string() | false<optional>`

Specifies in which directory the files are stored. If this parameter is undefined or `false`, the `error_logger_mf_h` is not installed.

`error_logger_mf_maxbytes = integer() <optional>`

Specifies how large each individual file can be. If this parameter is undefined, the `error_logger_mf_h` is not installed.

`error_logger_mf_maxfiles = 0<integer()<256 <optional>`

Specifies how many files are used. If this parameter is undefined, the `error_logger_mf_h` is not installed.

`overload_max_intensity = float() > 0 <optional>`

Specifies the maximum intensity for `overload`. Default is `0.8`.

`overload_weight = float() > 0 <optional>`

Specifies the `overload` weight. Default is `0.1`.

`start_prg = string() <optional>`

Specifies which program should be used when restarting the system. Default is `$OTP_ROOT/bin/start`.

`masters = [atom()] <optional>`

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

`client_directory = string() <optional>`

This parameter specifies the client directory at the master nodes. Refer to Release Handling in *OTP Design Principles* for more information. This parameter is ignored if the `masters` parameter is not set.

`static_emulator = true | false <optional>`

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 the `masters` and `client_directory` parameters are not set.

`releases_dir = string() <optional>`

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`.

`utc_log = true | false <optional>`

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

See Also

alarm_handler(3), error_logger(3), log_mf_h(3), *overload(3)*, *rb(3)*, *release_handler(3)*, *systools(3)*

alarm_handler

Erlang module

The alarm handler process is a `gen_event` event manager process which 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 start-up, 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 of them in a list which can be passed to a user defined event handler, which may be installed at a later stage. The list can grow large if many alarms are generated. So it is a good reason to install a better user defined handler.

There are functions to set and clear alarms. The format of alarms are defined by the user. For example, an event handler for SNMP could be defined, together with an alarm 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 may 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. Refer to gen_event(3) for further details.

Exports

`clear_alarm(AlarmId) -> void()`

Types:

`AlarmId = term()`

Clears all alarms 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()`

Sets an alarm with id `AlarmId`. This id is used at a later stage when the alarm is cleared.

See Also

error_logger(3), gen_event(3)

overload

Erlang module

`overload` is a process which indirectly regulates CPU usage in the system. The idea is that a main application calls the `request/0` function before starting a major job, and proceeds with the job if the return value is positive; otherwise the job must not be started.

`overload` is part of the `sasl` application, and all configuration parameters are defined there.

A set of two intensities are maintained, the `total intensity` and the `accept intensity`. For that purpose there are two configuration parameters, the `MaxIntensity` and the `Weight` value (both are measured in 1/second).

Then total and accept intensities are calculated as follows. Assume that the time of the current call to `request/0` is `T(n)`, and that the time of the previous call was `T(n-1)`.

- The current `total intensity`, denoted `TI(n)`, is calculated according to the formula,

  `TI(n) = exp(-Weight*(T(n) - T(n-1)) * TI(n-1) + Weight`,

  where `TI(n-1)` is the previous total intensity.
- The current `accept intensity`, denoted `AI(n)`, is determined by the formula,

  `AI(n) = exp(-Weight*(T(n) - T(n-1)) * AI(n-1) + Weight`,

  where `AI(n-1)` is the previous accept intensity, provided that the value of `exp(-Weight*(T(n) - T(n-1)) * AI(n-1)` is less than `MaxIntensity`; otherwise the value is

  `AI(n) = exp(-Weight*(T(n) - T(n-1)) * AI(n-1)`.

The value of configuration parameter `Weight` controls the speed with which the calculations of intensities will react to changes in the underlying input intensity. The inverted value of `Weight`,

```
T = 1/Weight
```

can be thought of as the "time constant" of the intensity calculation formulas. For example, if `Weight = 0.1`, then a change in the underlying input intensity will be reflected in the `total` and `accept intensities` within approximately 10 seconds.

The overload process defines one alarm, which it sets using `alarm_handler:set_alarm(Alarm)`. `Alarm` is defined as:

`{overload, []}`

This alarm is set when the current accept intensity exceeds `MaxIntensity`.

A new overload alarm is not set until the current accept intensity has fallen below `MaxIntensity`. To prevent the overload process from generating a lot of set/reset alarms, the alarm is not reset until the current accept intensity has fallen below 75% of `MaxIntensity`, and it is not until then that the alarm can be set again.

Exports

```
request() -> accept | reject
```

Returns `accept` or `reject` depending on the current value of the accept intensity.

The application calling this function should be processed with the job in question if the return value is `accept`; otherwise it should not continue with that job.

```
get_overload_info() -> OverloadInfo
```

Types:

```
OverloadInfo = [{total_intensity, TotalIntensity}, {accept_intensity,
AcceptIntensity}, {max_intensity, MaxIntensity}, {weight, Weight},
{total_requests, TotalRequests}, {accepted_requests, AcceptedRequests}].
TotalIntensity = float() > 0
AcceptIntensity = float() > 0
MaxIntensity = float() > 0
Weight = float() > 0
TotalRequests = integer()
AcceptedRequests = integer()
```

Returns the current total and accept intensities, the configuration parameters, and absolute counts of the total number of requests, and accepted number of requests (since the overload process was started).

See Also

alarm_handler(3), sasl(3)

rb

Erlang module

The Report Browser (RB) tool makes it possible to browse and format error reports written by the error logger handler `log_mf_h`.

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} | mp(), {mp(), Options}
Dates = {DateFrom, DateTo} | {DateFrom, from} | {DateTo, to}
DateFrom = DateTo = {date(), time()}
date() and time() are the same type as in the calendar module
```

This function displays the reports that match the provided filters.

When a filter includes the `no` atom it will exclude the reports that match that filter.

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

If the filter is of the form `{Key, RegExp, re}` the report must contain an element with `key = Key` and `Value` must match the RegExp regular expression.

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

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

If two Dates are provided, then reports that occurred between those dates are returned.

If you only want to filter only by dates, then you can provide the empty list as the Filters parameter.

See `rb:grep/1` for more information on the RegExp parameter.

`grep(RegExp)`

Types:

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

All reports containing the regular expression `RegExp` are printed.

`RegExp` can be 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. Refer to the module `re` and specially the function `re:run/3` for a definition of valid regular expressions and options.

`h()`
`help()`

Prints the on-line 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
```

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

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

`rescan()`
`rescan(Options)`

Types:

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

Rescans the report directory. `Options` is the same as for `start()`.

`show()`
`show(Report)`

Types:

```
Report = int() | type()
```

If a type argument is given, all loaded reports of this type are printed. If an integer argument is given, the report with this reference number is printed. If no argument is given, all reports are shown.

`start()`
`start(Options)`

Types:

```
Options = [opt()]
opt() = {start_log, FileName} | {max, MaxNoOfReports} | {report_dir,
DirString} | {type, ReportType} | {abort_on_error, Bool}
FileName = string() | standard_io
MaxNoOfReports = int() | all
DirString = string()
ReportType = type() | [type()] | all
Bool = true | false
```

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

`{start_log, FileName}` starts logging to file. All reports will be printed to the named file. The default is `standard_io`.

`{max, MaxNoOfReports}`. Controls how many reports the `rb_server` should read on start-up. This option is useful as the directory may contain 20.000 reports. If this option is given, the `MaxNoOfReports` latest reports will be read. The default is 'all'.

`{report_dir, DirString}`. Defines the directory where the error log files are located. The default is `{sasl, error_logger_mf_dir}`.

`{type, ReportType}`. Controls what kind of reports the `rb_server` should read on start-up. `ReportType` is a supported type, 'all', or a list of supported types. The default is 'all'.

`{abort_on_error, Bool}`. This option specifies whether or not logging should be aborted if rb encounters an unprintable report. (You may get a report on incorrect form if the `error_logger` function `error_msg` or `info_msg` has been called with an invalid format string). If `Bool` is `true`, rb will stop logging (and print an error message to stdout) if it encounters a badly formatted report. If logging to file is enabled, an error message will be appended to the log file as well. If `Bool` is `false` (which is the default value), rb will print an error message to stdout for every bad report it encounters, but the logging process is never aborted. All printable reports will be 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()`

Redirects all report output from the RB tool to the specified file.

`stop()`

Stops the `rb_server`.

`stop_log()`

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

release_handler

Erlang module

The *release handler* is a process belonging 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 a usage example can be found in *Design Principles*.

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 should be placed 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 in order to install the new release. The persistent state of the release handler is stored there in a file called `RELEASES`.

A release package should always contain the release resource file `Name.rel` and a boot script `Name.boot`. It may contain a release upgrade file `relup` and a system configuration file `sys.config`. The `.rel` file contains information about the release: its name, version, and which ERTS and application versions it uses. The `relup` file contains scripts 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 `relup`. An installed release can be made *permanent*. There can only be one permanent release in the system, and this is the release that is used if the system is restarted. An installed release, except the permanent one, can be *removed*. When a release is removed, all files that belong to that release only are deleted.

Each version of the release has a status. The status 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 following table illustrates the meaning of the status values:

```
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 may be synchronized between nodes. From an operator view, it may be unsatisfactory to specify each node. The aim is to install one release package in the system, no matter how many nodes there are. If this is the case, it is recommended that software management functions are written which take care of this problem. Such a function may 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 needs to have knowledge about which release it is currently running. It must also be able to change (in run-time) which boot script and system configuration file should be used

if the system is restarted. This is taken care of automatically if Erlang is started as an embedded system. Read about this in *Embedded System*. In this case, the system configuration file `sys.config` is mandatory.

The installation of a new release may 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 nodename (up to, but not including the "@") and *Release* is the current version of the release. 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 will be registered and started. The new service will be set to automatic and the old service removed as soon as the new release is made permanent.

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

`masters`

This node uses a number of master nodes in order to store and fetch release information. All master nodes must be up and running 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 because the executable files are statically written into memory.

It is also possible to use the release handler to unpack and install release packages when not running Erlang as an embedded system, but in this case the user must somehow make sure that correct boot scripts and configuration files are used if the system needs to be restarted.

There are additional functions 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` or `sys.config` are not present. If `relup` 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, or `{error,Reason}` is returned.

This function 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 the option `purge` is given, all old code that can be soft purged will be purged after all other checks are successfully completed. This can be useful in order to reduce the time needed by *install_release*.

```
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 in order to install new releases.

`Root` is the root of the installation (`$ROOT`) as described above. `RelDir` is the the directory where the `RELEASES` file should be created (normally `$ROOT/releases`). `RelFile` is the name of the `.rel` file that describes the initial release, including the extension `.rel`.

`AppDirs` can be used to specify from where the modules for the specified applications should 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 should 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. A release dependent file is a file that 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. It should 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 | int()>0
 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 applications 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, it means that the emulator will be restarted before the upgrade instructions are executed. This will happen if the emulator or any of the applications kernel, stdlib or sasl are updated. The new version of the emulator and these core applications will execute after the restart, but for all other applications the old versions will be started and the upgrade will be 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.

The option `error_action` defines if the node should be restarted (`init:restart()`) or rebooted (`init:reboot()`) in case of an error during the installation. Default is `restart`.

The option `code_change_timeout` defines the timeout for all calls to `sys:change_code`. If no value is specified or `default` is given, the default value defined in `sys` is used.

The option `suspend_timeout` defines the timeout 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.

The option `{update_paths,Bool}` indicates if all application code paths should be updated (`Bool==true`), or if only code paths for modified applications should be updated (`Bool==false`, default). This option only has effect for other application directories than the default `$ROOT/lib/App-Vsn`, that is, application directories provided in the `AppDirs` argument 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 to:

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

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

> Note:
>
> Installing a new release might be quite 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 might lead to garbage collections and copying of data.
>
> If you wish to speed up the execution of `install_release`, then you may call *check_install_release* first, using the option `purge`. This will do the same check for old code, and then purge all modules that can be soft purged. The purged modules will then no longer have any old code, and `install_release` will not need to do the checks.
>
> Obviously, this will not reduce the overall time for the upgrade, but it will allow 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, there will be an attempt to load new application beam code into the old emulator. In some cases, the new beam format can not be read by the old emulator, and so the code loading will fail and terminate the complete upgrade. To overcome this problem, the new application code should be compiled with the old emulator. See *Design Principles* for more information about emulator upgrade from pre OTP R15 versions.

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

Types:

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

Makes the specified version `Vsn` of the release 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 should 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 should 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.

Application Upgrade/Downgrade

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 `relup` 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 will cause `release_handler` to end up in an inconsistent state.
>
> No persistent information is updated, why these functions can be used on any Erlang node, embedded or not. Also, using these functions does not affect which code will be loaded in case of a reboot.
>
> If the upgrade or downgrade fails, the application may 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` 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 should 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 `{ok, Unpurged}` if evaluating the script is successful, where `Unpurged` is a list of unpurged modules, or `restart_emulator` if this instruction is encountered in the script, or `{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` will return `{error,restart_new_emulator}`. The reason for this is that this instruction requires that a new version of the emulator is 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` script.

App is the name of the application, which must be started. OldVsn is the previous version of the application and can be omitted if Dir is of the format "App-OldVsn". Dir is the library directory of this previous version of App, the corresponding modules as well as the old .app file should be located under Dir/ebin. The .appup file should 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 an 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 {ok, Unpurged} if evaluating the script is successful, where Unpurged is a list of unpurged modules, or restart_emulator if this instruction is encountered in the script, or {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 -- see appup(4)
```

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 is useful in order 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 should 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. High-level instructions are translated to low-level instructions and the instructions are sorted in the same manner as when generating a relup script.

Returns {ok, NewVsn, Script} if successful, where NewVsn is the new application version.

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 -- see appup(4)
```

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 is useful in order 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 as well as the old .app file should be located under Dir/ebin. The .appup file should 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 an downgrade script from the current version of the application. High-level instructions are translated to low-level instructions and the instructions are sorted in the same manner as when generating a relup script.

Returns {ok, Script} if successful.

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 should be located under `Dir/ebin`.

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

If the `restart_new_emulator` instruction is found in the script, `eval_appup_script/4` will return `{error,restart_new_emulator}`. The reason for this is that this instruction requires that a new version of the emulator is 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` can not 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 up/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 version `Vsn` of the release 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)`.

- `Posix` - Some file operation failed, as above.

SEE ALSO

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

systools

Erlang module

This module contains functions to generate boot scripts (`.boot`, `.script`), release upgrade scripts (`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 a script which describes how to upgrade the system from a number of previous releases, and how to downgrade to a number of previous releases. The script is used by `release_handler` when installing a new version of a release in run-time.

By default, `relup` is placed in the current working directory. If the option `{outdir,Dir}` is provided, `relup` is placed 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, it is deducted:

- Which applications should be deleted, that is applications which are listed in `Name.rel` but not in `Name2.rel`.
- Which applications should be added, that is applications which are listed in `Name2.rel` but not in `Name.rel`.
- Which applications should 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` script 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 `relup`.

The optional `Descr` parameter is included as-is in the `relup` script, 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` file for an application is located in the same directory.

If the option `{path,[Dir]}` is provided, this path is appended to the current path. The wildcard `*` is expanded to all matching directories. Example: `lib/*/ebin`.

If the `restart_emulator` option is supplied, a low-level instruction to restart the emulator is appended to the relup scripts. 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. See *Design Principles* for more information about this.

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

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

If the option `warnings_as_errors` is provided, 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_warn_sasl | warnings_as_errors
 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`. The boot file specifies which code should be loaded and which applications should be started when the Erlang runtime system is started. See `script(4)`.

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

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

The correctness of each application is checked:

- The version of an application specified in the `.rel` file should be the same as the version specified in the `.app` file.
- There should be no undefined applications, that is, dependencies to applications which are not included in the release. (Key `applications` in `.app` file).
- There should be no circular dependencies among the applications.
- There should be no duplicated modules, that is, modules with the same name but belonging to different applications.
- If the `src_tests` option is specified, a warning is issued if the source code for a module is missing or 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 will fail if the mandatory applications `kernel` and `stdlib` are not included in the `.rel` file and have start type `permanent` (default).

If `sasl` is not included as an application in the `.rel` file, a warning is emitted because such a release can not be used in an upgrade. To turn off this warning, add the 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 is 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 may be located in the corresponding `src` directory.

If the option `{path,[Dir]}` is provided, this path is appended to the current path. A directory in the path can be given 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` and assumed to be located in `$ROOT/lib`, where `$ROOT` is the root directory of the installed release. If the `local` option is supplied, the actual directories where the applications were found are used instead. This is a useful way to test a generated boot script locally.

The `variables` option 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 will get the path `VarName/Rest/App-Vsn/ebin` in the boot script. If an application is found in a directory `Prefix/Rest`, the path will be `VarName/Rest/App-Vsn/ebin`. When starting Erlang, all variables `VarName` are given values using the `boot_var` command line flag.

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

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

By default, errors and warnings are printed to tty and the function returns `ok` or `error`. If the option `silent` is provided, 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 the option `warnings_as_errors` is provided, warnings are treated as errors.

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

Types:

```
Name = string()
Opt = {dirs,[IncDir]} | {path,[Dir]} | {variables,[Var]} |
{var_tar,VarTar} | {erts,Dir} | src_tests | exref | {exref,[App]} | silent
| {outdir,Dir}
 Dir = string()
 IncDir = src | include | atom()
 Var = {VarName,PreFix}
  VarName = Prefix = string()
 VarTar = include | ownfile | omit
 Machine = atom()
 App = atom()
Result = ok | error | {ok,Module,Warnings} | {error,Module,Error}
```

`Module = atom()`
`Warning = Error = term()`

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

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

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

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

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

The `variables` option can be used to specify an installation directory other than `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 will be packed into a separate `VarName.tar.gz` file as `Rest/App-Vsn/ebin`.

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

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

The `{var_tar,VarTar}` option can be used to specify if and where a separate package should be stored. In this option, `VarTar` is:

- `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 separate files in the same directory as the `ReleaseName.tar.gz` file.
- `omit`. No separate (variable) packages are generated and applications which are found underneath a variable directory are ignored.

A directory called `releases` is also included in the release package, containing `Name.rel` and a subdirectory called `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`. These files are searched for in the same directory as `Name.rel`, in the current working directory, and in any directories specified using the `path` option.

If the release package should contain a new Erlang runtime system, the `bin` directory of the specified runtime system `{erts,Dir}` is copied to `erts-ErtsVsn/bin`.

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

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

`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 `File.boot`.

A boot script generated using the `make_script` function 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 should be called `Application.appup` where `Application` is the name of the application. The file should 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()` is the current version of the application.
- `UpFromVsn = string() | binary()` is an earlier version of the application to upgrade from. If it is a string, it will be interpreted as a specific version number. If it is a binary, it will be interpreted as a regular expression which can match multiple version numbers.
- `DownToVsn = string() | binary()` is an earlier version of the application to downgrade to. If it is a string, it will be interpreted as a specific version number. If it is a binary, it will be interpreted as a regular expression which can match multiple version numbers.
- `Instructions` is a list of *release upgrade instructions*, see below. It is recommended to use high-level instructions only. These are automatically translated to low-level instructions by `systools` when creating the `relup` file.

In order to avoid duplication of upgrade instructions it is allowed to use regular expressions to specify the `UpFromVsn` and `DownToVsn`. To be considered a regular expression, the version identifier must be specified as a binary, e.g.

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

will match all versions `2.1.x`, where x is any number.

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, refer to *OTP Design Principles*.

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 the module `Mod`. All those processes are suspended using `sys:suspend`, the new version of the module 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}`, processes implemented using gen_server, gen_fsm or gen_event will transform their internal state by calling the callback function `code_change`. Special processes will 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` and controls what action to take with processes that are executing old code before loading the new version of the module. 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` and controls what action to take with processes that are executing old code when the new version of the module 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 will purge the old code when no remaining processes execute the code.

`DepMods` defaults to [] and defines which other modules `Mod` is dependent on. In `relup`, instructions for suspending processes using `Mod` will come before instructions for suspending processes using modules in `DepMods` when upgrading, and vice versa when downgrading. In case of circular dependencies, the order of the instructions in the `appup` script is kept.

`Timeout` defines the timeout when suspending processes. If no value or `default` is given, the default value for `sys:suspend` is used.

`ModType` defaults to `dynamic` and specifies if the code is "dynamic", that is if a process using the module does spontaneously switch to new code, or if it is "static". When doing an advanced update and upgrading, 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`.

See `update` above for a description of `PrePurge` and `PostPurge`.

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

```
{add_module, Mod}
  Mod = atom()
```

Loads a new module `Mod`.

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

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

```
{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 will be loaded and started in the corresponding way, see `application(3)`. If `Type = load`, the application will only be loaded. If `Type = none`, the application will be neither loaded nor started, although the code for its modules will be 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.

Low-level instructions

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

Reads each `Mod` from the directory `App-Vsn/ebin` as a binary. It does not load the modules. The instruction should be placed first in the script in order to read all new code from file to make the suspend-load-resume cycle less time consuming. After this instruction has been executed, the code server with the new version of `App`.

```
point_of_no_return
```

If a crash occurs after this instruction, the system cannot recover and is restarted from the old version of the release. The instruction must only occur once in a script. It should 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. See the high-level instruction `update` for a description of `PostPurge`.

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

Makes the current version of `Mod` old. `PrePurge` is ignored. See the high-level instruction `update` for a description of `PostPurge`.

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

Purges each module `Mod`, that is removes the old code. Note 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 may cause the process to die, either because it crashes when it spontaneously switches to new code, or as a result of a purge operation. If no `Timeout` is specified or `default` is given, 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 the function `sys:change_code`, passing the term `Extra` as argument.

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

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

The 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. In case a node goes down, it is considered to be an unrecoverable error, and the local node is restarted from the old release. There is no timeout for this instruction, which means that it may hang forever.

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

Evaluates `apply(M, F, A)`. If the instruction appears before the `point_of_no_return` instruction, 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 the `point_of_no_return` instruction, and the function call fails, the system is restarted.

```
restart_new_emulator
```

This instruction is used when 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, and it shall be placed first. *systools:make_relup3,4* will ensure this when the relup is generated. The rest of the relup script is executed after the restart as a part of the boot script.

An info report will be written when the upgrade is completed. To programatically find out if the upgrade is complete, call *release_handler:which_releases* 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 in case of an emulator restart.

```
restart_emulator
```

This instruction is similar to `restart_new_emulator`, except it shall be placed at the end of the relup script. 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 reqiured. When generating the relup, *systools:make_relup/3,4* ensures that there is only one `restart_emulator` instruction and that it is the last instruction of the relup.

SEE ALSO

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

rel

Name

The *release resource file* specifies which applications are 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 should 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()` is the name of the release.
- `Vsn = string()` is the version of the release.
- `EVsn = string()` is the version of ERTS the release is intended for.
- `Application = atom()` is the name of an application included in the release.
- `AppVsn = string()` is the version of an application included in the release.
- `Type = permanent | transient | temporary | load | none` is the start type of an application included in the release.

  If `Type = permanent | transient | temporary`, the application will be loaded and started in the corresponding way, see `application(3)`. If `Type = load`, the application will only be loaded. If `Type = none`, the application will be neither loaded nor started, although the code for its modules will be loaded. Defaults to `permanent`
- `IncApps = [atom()]` is 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 should be located in the `OTP_ROOT/erts-EVsn/Vsn` directory.

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()` is the current version of the release.
- `UpFromVsn = string()` is an earlier version of the release to upgrade from.
- `Descr = term()` is a user defined parameter passed from the `systools:make_relup/3,4` function. It will be used in the return value of `release_handler:install_release/1,2`.
- `Instructions` is 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()` is an earlier version of the release to downgrade to.

When upgrading from `UpFromVsn` with `release_handler:install_release/1,2`, there does not have to be an exact match of versions, but `UpFromVsn` can be a sub-string of the current release version.

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.

The 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 `.app` files.

FILE SYNTAX

The boot script is stored in a file with the 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 name of the system.
- `Vsn = string()` defines the version of the system.
- `{progress, Term}` sets the "progress" of the initialization program. The function `init:get_status()` returns the current value of the progress, which is `{InternalStatus,Term}`.
- `{path, [Dir]}` where `Dir` is a string. This argument sets the load path of the system to `[Dir]`. The load path used to load modules is obtained from the initial load path, which is given in the script file, together with any path flags which were supplied in the command line arguments. The command line arguments modify the path as follows:
  - `-pa Dir1 Dir2 ... DirN` adds the directories `Dir1, Dir2, ..., DirN` to the front of the initial load path.
  - `-pz Dir1 Dir2 ... DirN` adds the directories `Dir1, Dir2, ..., DirN` to the end of the initial load path.
  - `-path Dir1 Dir2 ... DirN` defines a set of directories `Dir1, Dir2, ..., DirN` which replaces the search path given in the script file. Directory names in the path are interpreted as follows:
    - Directory names starting with `/` are assumed to be absolute path names.
    - Directory names not starting with `/` are assumed to be relative 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 the function `erl_prim_loader:get_file(Mod)`. A fatal error which terminates the system will occur if the module cannot be located.
- `{kernel_load_completed}` indicates that all modules which *must* be loaded *before* any processes are started are loaded. In interactive mode, all `{primLoad,[Mod]}` commands interpreted after this command are ignored, and these modules are loaded on demand. In embedded mode, `kernel_load_completed` is ignored, and all modules are loaded during system start.
- `{kernelProcess, Name, {Mod, Func, Args}}` starts a "kernel process". The kernel process `Name` is started by evaluating `apply(Mod, Func, Args)` which is expected to return `{ok, Pid}` or `ignore`. The `init` process monitors the behaviour of `Pid` and terminates the system if `Pid` dies. Kernel processes are key components of the runtime system. Users do not normally add new kernel processes.
- `{apply, {Mod, Func, Args}}`. The init process simply evaluates `apply(Mod, Func, Args)`. The system terminates if this results in an error. The boot procedure hangs if this function never returns.

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

SEE ALSO

systools(3)