System Application Support Libraries (SASL)

version 2.1
Typeset in \LaTeX\ from SGML source using the DOCBUILDER 3.3.2 Document System.
## Contents

**1 SASL User's Guide**

1.1 About This Document ........................................... 1

**2 SASL Error Logging**

2.1 Supervisor Report ........................................... 3
2.2 Progress Report ........................................... 3
2.3 Crash Report ............................................. 4
   2.3.1 An Example ...................................... 4
2.4 Multi-File Error Report Logging ................................ 5
2.5 Report Browser ............................................ 5
   2.5.1 Starting the Report Browser .......................... 5
   2.5.2 On-line Help ...................................... 5
   2.5.3 List Reports in the Server ........................... 6
   2.5.4 Show Reports ...................................... 6
   2.5.5 Search the Reports ................................ 7

**3 SASL Reference Manual**

3.1 sasl ......................................................... 15
3.2 alarm_handler ............................................... 17
3.3 overload .................................................... 19
3.4 rb ......................................................... 21
3.5 release_handler ............................................. 24
3.6 systools .................................................... 31
3.7 appup ....................................................... 36
3.8 rel ......................................................... 41
3.9 relup ......................................................... 43
3.10 script ....................................................... 44

System Application Support Libraries (SASL)
Chapter 1

SASL User's Guide

1.1 About This Document

The SASL 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.
Chapter 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) [page 15], and app(File) in the Kernel Reference Manual for the details.

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.

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.
Chapter 2: SASL Error Logging

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.

2.3.1 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}]
```
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.

2.5 Report Browser

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

The error_logger.mfh 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.

2.5.1 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.
```

2.5.2 On-line Help

Enter the command rb:help(). to access the report browser on-line help system.
2.5.3 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
```

2.5.4 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 f local,sasl
          f pid,release_handler, f name,release_handler,start_link,[]
          f mfa,release_handler,start,[]
          f restart_type,permanent
          f shutdown,2000
          f child_type,worker
ok
```

CRASH REPORT <0.24.0> 1996-10-16 16:16:21

```
ok
```
2.5: Report Browser

===============================================================================
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

2.5.5 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_pig7",
[release,
"OTP APN 181 01",
System Application Support Libraries (SASL)
Chapter 2: SASL Error Logging

"P1G",
undefined,
[],
permanent],
undefined}
** Reason for termination ==
** {undefined,[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  {undefined,[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  []
dictionary  [<0.23.0>,<0.20.0>]
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  {undefined,[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
SASL Reference Manual

Short Summaries

- Application sasl [page 15] - The SASL Application
- Erlang Module alarm_handler [page 17] - An Alarm Handling Process
- Erlang Module overload [page 19] - An Overload Regulation Process
- Erlang Module rb [page 21] - The Report Browser Tool
- Erlang Module release_handler [page 24] - Unpacking and Installation of Release Packages
- File rel [page 41] - Release resource file
- File relup [page 43] - Release upgrade file
- File script [page 44] - Boot script

sasl

No functions are exported.

alarm_handler

The following functions are exported:

- clear_alarm(AlarmId) -> void() [page 17] Clear the specified alarms
- get_alarms() -> [alarm()] [page 17] Get all active alarms
- set_alarm(alarm()) [page 17] Set an alarm with an id

overload

The following functions are exported:

- request() -> accept | reject [page 20] Request to proceed with current job
- get_overload_info() -> OverloadInfo [page 20] Return current overload information data
rb

The following functions are exported:

- grep(RegExp)
  [page 21] Search the reports for a regular expression
- h()
  [page 21] Print help information
- help()
  [page 21] Print help information
- list()
  [page 21] List all reports
- list(Type)
  [page 21] List all reports
- rescan()
  [page 21] Rescan the report directory
- rescan(Options)
  [page 21] Rescan the report directory
- show()
  [page 21] Show reports
- show(Report)
  [page 21] Show reports
- start()
  [page 22] Start the RB server
- start(Options)
  [page 22] Start the RB server
- start_log(FileName)
  [page 22] Redirect all output to FileName
- stop()
  [page 23] Stop the RB server
- stop_log()
  [page 23] Stop logging to file

release_handler

The following functions are exported:

- check_install_release(Vsn) -> {ok, OtherVsn, Descr} | {error, Reason}
  [page 26] Check installation of a release in the system.
- create_RELEASES(Root, RelDir, RelFile, AppDirs) -> ok | {error, Reason}
  [page 26] Create an initial RELEASES file.
- install_file(Vsn, File) -> ok | {error, Reason}
  [page 26] Install a release file in the release structure.
- install_release(Vsn) -> {ok, OtherVsn, Descr} | {error, Reason}
  [page 27] Install a release in the system.
install_release(Vsn, [Opt]) -> {ok, OtherVsn, Descr} | {error, Reason}
[page 27] Install a release in the system.

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

remove_release(Vsn) -> ok | {error, Reason}
[page 28] Remove a release from the system.

reboot_old_release(Vsn) -> ok | {error, Reason}
[page 28] Reboot the system from an old release.

set_removed(Vsn) -> ok | {error, Reason}
[page 28] Mark a release as removed.

set_unpacked(RelFile, AppDirs) -> ok, Vsn | {error, Reason}
[page 28] Mark a release as unpacked.

unpack_release(Name) -> ok, Vsn | {error, Reason}
[page 29] Unpack a release package.

which_releases() -> [{Name, Vsn, Apps, Status}]
[page 29] Return all known releases

systools

The following functions are exported:

• make_relup(Name, UpFrom, DownTo) -> Result
  [page 31] Generate a release upgrade file relup.

• make_relup(Name, UpFrom, DownTo, [Opt]) -> Result
  [page 31] Generate a release upgrade file relup.

• make_script(Name) -> Result

• make_script(Name, [Opt]) -> Result

• make_tar(Name) -> Result
  [page 33] Create a release package.

• make_tar(Name, [Opt]) -> Result
  [page 33] Create a release package.

• script2boot(File) -> ok | error
  [page 35] Generate a binary version of a boot script.

appup

No functions are exported.

rel

No functions are exported.
relup
No functions are exported.

script
No functions are exported.
This section describes the SASL 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.

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 stdout.
- error_logger_nf.h: This error logger writes all events sent to the error logger to disk. It installs the log_nf.h event handler in the error_logger process.

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.

errolg 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.

errolog nf dir = string() | false <optional> Specifies in which directory the files are stored. If this parameter is undefined or false, the error_logger_nf.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) [page 17], error_logger(3), log mf h(3), overload(3) [page 19], rb(3) [page 21], release_handler(3) [page 24], systools(3) [page 31]
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, AlarmDescr}).

{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:
Set 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 $T(n-1)$.

1. The current total intensity, denoted $TI(n)$, is calculated according to the formula,

   $$TI(n) = \exp(-Weight \times (T(n) - T(n-1)) \times TI(n-1) + Weight,$$

   where $TI(n-1)$ is the previous total intensity.

2. The current accept intensity, denoted $AI(n)$, is determined by the formula,

   $$AI(n) = \exp(-Weight \times (T(n) - T(n-1)) \times AI(n-1) + Weight,$$

   where $AI(n-1)$ is the previous accept intensity, provided that the value of

   $$\exp(-Weight \times (T(n) - T(n-1)) \times AI(n-1)$$

   is less than `MaxIntensity`; otherwise the value is

   $$AI(n) = \exp(-Weight \times (T(n) - T(n-1)) \times AI(n-1).$$

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

```erlang
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

grep(RegExp)

Types:
  • RegExp = string()
All reports containing the regular expression RegExp are printed.
RegExp is a string containing the regular expression. Refer to the module regexp in the STDLIB reference manual for a definition of valid regular expressions. They are essentially the same as the UNIX command egrep.

h()
help()

Prints the on-line help information.

list()
list(Type)

Types:
  • Type = type()
  • type() = crash_report | supervisor_report | error | 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.
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.

A release package is a compressed tar file containing code for a certain version of a release, see systools(3). The release package should be placed in the $ROOT/releases directory of a 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:

<table>
<thead>
<tr>
<th>Status</th>
<th>Action</th>
<th>NextStatus</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>unpack</td>
<td>unpacked</td>
</tr>
<tr>
<td>unpacked</td>
<td>install</td>
<td>current</td>
</tr>
<tr>
<td></td>
<td>remove</td>
<td>-</td>
</tr>
<tr>
<td>current</td>
<td>make permanent</td>
<td>permanent</td>
</tr>
<tr>
<td></td>
<td>install other</td>
<td>old</td>
</tr>
<tr>
<td></td>
<td>remove</td>
<td>-</td>
</tr>
<tr>
<td>permanent</td>
<td>make other permanent</td>
<td>old</td>
</tr>
<tr>
<td></td>
<td>install</td>
<td>permanent</td>
</tr>
<tr>
<td>old</td>
<td>reboot_old</td>
<td>permanent</td>
</tr>
<tr>
<td></td>
<td>install</td>
<td>current</td>
</tr>
<tr>
<td></td>
<td>remove</td>
<td>-</td>
</tr>
</tbody>
</table>
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.

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 release of the application. 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:

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}

Types:
- Vsn = OtherVsn = string()
- 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.

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 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} | {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()} | term()

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 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(RefFile, [{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`.

```
release_handler
```

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

Types:

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

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

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

**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.

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

**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`.

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

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

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

Types:

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

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

AppDir 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.

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) [page 43], rel(4), script(4) [page 44], sys(3), systools(3) [page 31]
**systools**

Erlang Module

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

**Exports**

```erlang
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
- **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.

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 in the current working directory.

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.

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.

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

**Types:**
- **Name** = string()
- **Opt** = no_module_tests | {path,[Dir]} | local | {variables,[Var]} | exref | {exref,[App]} | silent
- **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 in a readable version Name.script and a binary version Name.boot, specifying 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)).

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 not be duplicated modules, that is, modules with the same name but belonging to different applications.
- A warning is issued if the source code for a module is missing or newer than the object code.
  If the no_module_tests option is specified, this check is omitted.

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 resulting boot script files are created in the current working directory, or in the directory where Name.rel is located if Name contains a path.

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:formatwarning(Warnings) or Module:formaterror(Error).

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

Types:
- Name = string()
Creates a release package file `Name.tar.gz`. 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 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 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/RelVsn is also included in the release package. The release version RelVsn is found in the release package. This directory contains Name.rel, the boot script Name.boot renamed to start.boot and, if found, the files relup and sys.config.

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-ErrtsVsn/bin.

All checks performed with the make_script function are performed before the release package is created. The no_module_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)
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 theebin 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() is an earlier version of the application to upgrade from.
- `DownToVsn` = string() is an earlier version of the application to downgrade to.
- `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.

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
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.
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.

Editions can replace the module Mod.

Load a new module Mod.

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

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

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.

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

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

Shuts down the current emulator and starts a new one. All processes are terminated gracefully. The new release must still be made permanent when the new emulator is up and running. Otherwise, the old emulator is started in case of an emulator restart. This instruction should be used when a new emulator is introduced, or if a complete reboot of the system should be done.

SEE ALSO
relup(4) [page 43], release_handler(3) [page 24], supervisor(3), systools(3) [page 31]
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 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:

```erlang
{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 empty list.

**Note:**
The list of applications must contain the kernel and stdlib applications.
SEE ALSO

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

File

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)`
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:getArgument(mode)` to find out if it should run in demand mode, or non-demand driven mode.

SEE ALSO
systools(3)
Index of Modules and Functions

Modules are typed in this way. Functions are typed in this way.

alarm_handler
   clear_alarm/1,17
   get_alarms/0,17
   set_alarm/1,17

check_install_release/1
   release_handler, 26

clear_alarm/1
   alarm_handler, 17
create_RELEASES/4
   release_handler, 26

get_alarms/0
   alarm_handler, 17
get_overload_info/0
   overload, 20
grep/1
   rb, 21

h/0
   rb, 21
help/0
   rb, 21

install_file/2
   release_handler, 26
install_release/1
   release_handler, 27
install_release/2
   release_handler, 27

list/0
   rb, 21
list/1
   rb, 21

make_permanent/1
   release_handler, 28
make_relup/3
   systools, 31
make_relup/4
   systools, 31
make_script/1
   systools, 32
make_script/2
   systools, 32
make_tar/1
   systools, 33
make_tar/2
   systools, 33

overload
   get_overload_info/0, 20
   request/0, 20

rb
   grep/1, 21
   h/0, 21
   help/0, 21
   list/0, 21
   list/1, 21
   rescan/0, 21
   rescan/1, 21
   show/0, 21
   show/1, 21
   start/0, 22
   start/1, 22
   start_log/1, 22
   stop/0, 23
   stop_log/0, 23

reboot_old_release/1
   release_handler, 28
release_handler
check_install_release/1, 26
create_RELEASES/4, 26
install_file/2, 26
install_release/1, 27
install_release/2, 27
make_permanent/1, 28
reboot_old_release/1, 28
remove_release/1, 28
set_removed/1, 28
set_unpacked/2, 28
unpack_release/1, 29
which_releases/0, 29

remove_release/1
release_handler, 28

request/0
 overload, 20
rescan/0
 rb, 21
rescan/1
 rb, 21

script2boot/1
 systools, 35

set_alarm/1
 alarm_handler, 17
set_removed/1
 release_handler, 28
set_unpacked/2
 release_handler, 28

show/0
 rb, 21
show/1
 rb, 21
start/0
 rb, 22
start/1
 rb, 22
start_log/1
 rb, 22
stop/0
 rb, 23
stop_log/0
 rb, 23

systools
 make_relup/3, 31
 make_relup/4, 31

unpack_release/1
 release_handler, 29

which_releases/0
 release_handler, 29

System Application Support Libraries (SASL)