System Application Support Libraries (SASL)

version 1.7

OTP Team

1997-11-17

Typeset in LaTeX from SGML source using the DOCBUILDER 3.0 Document System.

Contents

Chapter 1

SASL User's Guide

The System Application Support Libraries (*SASL*) provides support for alarm and release handling etc.

1.1 Introduction

This User's Guide describes the System Architecture Support Libraries (SASL). SASL is itself an Erlang application which currently includes modules for:

- error logging
- release handling
- report browsing.

These services, and the release structure and release handling processes, are described in this Users's Guide.

Scope and Purpose

This manual describes the SASL application as a component of the Erlang/Open Telecom Platform development environment. It is assumed that the reader is familiar with the Erlang Development Environment which is described in a separate user's guide.

Exclusions

This User Guide assumes that the reader is familiar with the Erlang programming language and does not explain how to program in Erlang. References to programming manuals are listed at the end of this chapter.

About This Book

In addition to this introductory chapter, this User's Guide includes the following chapters:

- Chapter 2: "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 fie. It also describes the report browser `rb_server`.
- Chapter 3: "The Release Structure" provides an overview of the Erlang release tools and processes.
- Chapter 4: "Release Handling" describes the administration and principles of release handling in detail.

Typographical Conventions

The following typographical conventions are used in this user's guide.

convention	*where used*
command	To show command line entries To show keyboard entries at system prompts
`code`	To highlight Erlang code, module and function names, arguments, variables, and file names.

Table 1.1: Examples of Typographical Conventions

Where to Find More Information

Refer to the following documentation for more information:

- the Erlang Development Environment User's Guide
- the Erlang 4.4 Extensions User's Guide
- the Reference Manual
- the Erlang Embedded Systems User's Guide
- the Mnesia User's Guide
- the SNMP User's Guide
- the Installation Guide
- Concurrent Programming in Erlang, 2nd Edition, ISBN 0-13-508301-X.

1.2 SASL Error Logging

The SASL application introduces three types of reports:

- supervisor report
- progress report
- crash report.

The SASL application adds a handler for these reports when it is started. According to the SASL configuration these error reports are written on screen, or to a specified file.

Supervisor Report

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

Supervisor. This is the name of the reporting supervisor.

Context. This section 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. This is the termination reason.

Offender. This is the start specification for the child.

Progress Report

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

Supervisor. This is the name of the reporting supervisor.

Started. This is the start specification for the successfully started child.

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 which shows what process crashed, and 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}]
```

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 Applications in the Reference Manual for more details.

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 direcory 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()*. ro 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_jam_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` as follows with the function `rb:stop()`:

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

1.3 The Release Structure

Erlang programs are organized into modules. Each module in a release must have a unique name.

Collections of modules which cooperate to solve a particular problem are organized into applications. Applications are described in an application resource file.

Collections of applications are organized into a release. Releases are described in a release resource file.

Naming of Modules, Applications and Releases

Every module in the system has a version number. An Erlang module should start:

```
-module(Mod).
-vsn(Vsn).
...
```

This module should be stored in a file named `Mod.erl`.

The name of the module is `Mod` and the version of the module is `Vsn`. `Mod` must be an atom while `Vsn` can be any valid Erlang term. For example, the version can be an integer, or a string which represents an Ericsson product number.

Applications also have versions, but the version must be a string. For example, the application resource file for the application named `snmp` must be stored in a file named `snmp.app` and must start:

```
{application, snmp,
  [{vsn, Va},
   {modules,
     [{lists, V1},
      {orddsets, V2}
      ...
```

Here, `Va` is the version of the application (a string). The application uses the Erlang module versions `V1`, `V2`, `...`, where `V1`, `V2`, `...` can be any valid Erlang terms. The only requirement is that the module version types (integers, strings, etc.) agrees with the convention used in the module declarations.

> **Note:**
> In the application resource file, the name of a module must be specified in `modules`, but the version number is not a mandatory requirement.

Applications can be upgraded and the instructions to do this are placed in the `.appup` file for the application. For example, for the `snmp` application these instructions are placed in the `snmp.appup` file. The `.appup` file looks as follows:

```
{Vsn,
    [{UpFromVsn, UpFromScript}, ...],
    [{DownToVsn, DownToScript}, ...]
}.
```

- `Vsn` is the current version of the application
- `UpFromVsn` is a version we can upgrade from
- `UpFromScript` is the script which describes the sequence of release upgrade instructions Refer to the section Release Handling Instructions [page 25]
- `DownToVsn` is a version to which we can downgrade
- `DownToScript` is the script which describes the sequence of downgrade instructions.

In the case of `UpFrom Script` and `DownFromScript`, the scripts typically contain one line for each module in the application.

A release resource file has a structure similar to an application resource file. The file `ReleaseName.rel` which describes the `ReleaseName` release looks as follows:

```
{release, {Name,Vsn}, {erts, EVsn},
 [{AppName, AppVsn}, {AppName, AppVsn, AppType},  {AppName, AppVsn,
 IncApps}, {AppName, AppVsn, AppType, IncApps} ...]}.
```

- `Name` is the name of the release (a string). `Name` need not be the same as `ReleaseName` above.
- `Vsn` is the version of the release (a string).
- `{erts, EVsn}` indicates which Erlang runtime system version `EVsn` the release is intended for, for example "4.4". `EVsn` must be a string.
- `AppName` is the name of an application included in the release (an atom).
- `AppVsn` is the version of the `AppName` application (a string).
- `AppType = permanent | transient | temporary | load | none` is the start type of the `AppName` application. This parameter specifies how the application is treated in the systools-generated start script. If it is `permanent`, `transient` or `temporary`, the application is started with a call to `application:start(AppName, AppType)`. If it is `load`, the application is loaded, but not started. If it is `none`, the application is neither loaded nor started.
- `IncApps` is a list of applications that are included by an application, for example `[AppName, ...]`. This list overrides the `included_applications` key in the application resource file `.app`. This list must be a subset of the list of included applications which are specified in the `.app` file.

> **Note:**
> The list of applications must contain the `kernel` and the `stdlib` applications.

Releases can also be upgraded and instructions for this should be written in the `relup` file (see the definition of the relup file [page 20]). The tedious work of writing the `relup` file has been automated and in most cases the file can be automatically generated from the `.appup` files for the applications in the release.

Release Tools

There are tools available to build and check release packages. These tools read the release resource file, the application resource and upgrade files and they generate a boot script, a release upgrade script, and build a release package.

The following functions exists in the `systools` module:

- `make_script` generates a boot script
- `make_relup` generates a release upgrade script
- `make_tar` generates a release package `.tar` file .

These functions read the `.rel` release resource file from the current directory and performs syntax and dependency checks before the output is generated.

> **Note:**
> The generated files are written to the current directory as well.

Refer to the Reference Manual for more information about these functions.

Release Directories

A release should be divided into the following directories:

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

The release resource file includes one `AppN-AVsnN` directory per application. `AppN` is the name and `AVsnN` is the version of the application.

- The `ebin` directory contains the Erlang object code and the application resource file.
- The `priv` directory contains any application private data. Specifically, port programs should be located in the `priv` directory. A port program is found by using the `code:priv_dir(AppN) ++ "/bin/Pgm"` expression (if a `bin` directory exist under `priv` as well).
- The boot script and relup files should be located in the `releases/Vsn` directory. `Vsn` is the release version found in the release resource file.
- The Erlang runtime system executables are located in the `erts-EVsn/bin` directory.
- The `releases` directory should also contain the `ReleaseName.rel` files, and new release packages are installed here.

- The `bin` directory contains the top level Erlang executable program `erl`.

There is no requirement for applications to be located under the `$ROOTDIR/lib` directory. Accordingly, several installation directories may exist which contain different parts of a system. For example, the previous example could be extended as follows:

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

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

The `$SECOND_ROOT` and `$THIRD_ROOT` variables are introduced to the system through the `systools` module.

Diskless and/or Read-Only Clients

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

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

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

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

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

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

Example

Suppose we have a system called “test”, which consists of the three applications `snmp`, `kernel` and `stdlib`. The first application can be described in an application resource file as follows:

```
{application, snmp,
 [{vsn, "10"},
  {modules,
   [{snmp_table, 2},
    {snmp_map, 3},
    {snmp_stuff,5}]},
  {applications,
   [stdlib,
    kernel]},
  {mod,
   {snmp_stuff, [12,34]}}
 ]}.
```

> **Note:**
> The resource file shown contains only a sub-set of the information available in the actual resource files. Refer to the Erlang Development Environment User Guide, Chapter 3: Design Principles, section *Applications* for a more detailed description of the content in an application resource file.

In the example shown, version “10” of `snmp` uses version 2 of `snmp_table`, version 3 of `snmp_map` an so on. It requires that `stdlib` and `kernel` are started before this application is started. It is started by

evaluating the function `snmp_stuff:start(normal, [12,34])`. `snmp_stuff` is the application callback module for the application.

> **Note:**
> We have used integer version numbers written as strings for the application version. In our further discussion we will simplify things by using integer version numbers. We will also assume that version N+1 is the successor of version N of a system component.

`stdlib` version "6" looks as follows:

```
{application, stdlib,
 [{vsn, "6"},
  {modules,
   [{lists,2},
    {dict,4},
    {ordsets, 7}]},
  {applications,
   []},
 ]}.
```

> **Note:**
> `stdlib` is a "code only" application and has no callback module.

Finally, `kernel` version "2" looks as follows:

```
{application, kernel,
 [{vsn, "2"},
  {modules,
   [{net_kernel, 3},
    {auth, 3},
    {rcp, 5}]},
  {applications,
   [stdlib]},
  {mod,
   {net_kernel,[]}}]}.
```

We can now define release "5" of the "test" release in terms of these applications:

```
{release,
 {"test", "5"},
 {erts, "4.4"},
 [{kernel, "2"},
  {stdlib, "6"},
  {snmp, "10"}
 ]}.
```

> **Note:**
> This means that release “5” of the “test” system is built from `kernel` version “2”, `stdlib` version “6”. The release requires the Erlang runtime system “4.4”.

Making the Start Script

In the example shown, we have defined enough to be able to generate a system. We now have to generate a start script off-line which will be used when the system is loaded. We evaluate:

```
systools:make_script("test")
```

This command checks that all applications required for the release can be found and that all the modules which are required can be located and have the correct version numbers. This is easy because each module only belongs to one application.

If there were no errors, a start script called `test.script` and a boot file called `test.boot` are created. The script first loads all the modules to be used and then starts and loads all the applications in the order given in the release description file. This file may be re-ordered if other dependencies were found among the applications.

All required application resource files and all required Erlang files must be located somewhere within the current code path.

If this succeeds, then the command `erl -boot test` can be issued. This command actually starts the system.

Changing an Application

Suppose now that we make a change to `snmp` which results in new versions of `snmp_map` and `snmp_stuff`. This can be described as follows:

```
{application, snmp,
 [{vsn,"11"},
  {modules,
   [{snmp_table, 2},
    {snmp_map, 4},
    {snmp_stuff,6}]},
  {applications,
   [stdlib,
    kernel]},
  {mod,
   {snmp_stuff, [12,34]}}
 ]}.
```

> **Note:**
> We have changed the two modules `snmp_map` and `snmp_stuff`. Everything else remains the same.

We can now define a new release of the system as follows:

```
{release,
 {"test","6"},
 {erts, "4.4"},
 [{kernel, "2"},
  {stdlib, "6"},
  {snmp, "11"}
 ]}.
```

The .appup file for the snmp application should be:

```
{"11",
 [{"10", [{update, snmp_map, soft, soft_purge, soft_purge, []},
          {update, snmp_stuff, soft, soft_purge, soft_purge, []}]}],

 [{"10", [{update, snmp_map, soft, soft_purge, soft_purge, []},
          {update, snmp_stuff, soft, soft_purge, soft_purge, []}]}]
}.
```

In this file, we state that the snmp application can be upgraded by changing code on the snmp_map and snmp_stuff modules. We also state that the snmp application can be downgraded by changing the same two modules.

We can now compile a new script file named test2 as before:

```
systools:make_script("test2")
```

A relup file is generated by evaluating:

```
systools:make_relup("test2", ["test"], ["test"]).
```

Making a Release Package

Next, we want to generate a release package which can be installed in the target system. After evaluating make_script/1 and make_relup/3 as described above, we do this by evaluating

```
systools:make_tar("test2").
```

This creates a release package file, named test2.tar.gz. This release package file may be installed in a target system by using the release_handler. [page 18]

1.4 Release Handling

Introduction

A new release is produced whenever a change of the system is made. The release consists of a release packet which is identified by its name. The release package is installed in a running system by giving commands to the *release handler*, which is an SASL process. The system has a unique version string, which is updated whenever a new release is installed. This version string is referred to as *the system version*. This is the version of the entire system, not just the OTP version. If the system consists of several nodes, each node has its own system version. Release handling can be synchronized between nodes, or be done at one node at a time.

Some changes require the node to be brought down. If this is the case and the system consists of several nodes, the release upgrade can be implemented as follows:

1. move all applications from the node to be changed to other nodes
2. take down the node
3. do the change
4. restart the node and move the applications back.

There are several different types of releases:

Operating system change. This change can only be made by taking down the node. The installation is not supported by the release handler and is performed manually. It is not possible to roll back to a previous release automatically if there is an error.

Application code or data change. This is the normal type of release. The release is installed without bringing down the running node. Some type of changes, for example change of C-programs, may be done by shutting down and restarting the affected processes.

Erlang emulator change. This change can only be made by taking down the node. However, the release handler supports this type of change and it is done as a normal release.

Administering Releases

This section describes how to build and install releases. Also refer to the Reference Manual, the section SASL, `release_handler` for more details. The following steps are involved in administering releases:

1. Each node in the system needs a release packet. A release building tool in OTP is used to construct this packet. This tool constructs a release packet from application specification files, code files, data files, and a file which describes how the release is installed in the system. The release building tool is invoked with commands in the module `systools`.
2. The release packet must be transferred to the target machine. This can be done with a method like ftp.
3. After transfer to the target machine, the release packet must be unpacked. The unpacking makes the system version in the release packet available for installation. When the system version is installed, a *release upgrade script* is executed, which loads all new code. If an installation fails in some way, the entire system is restarted from the old version.

4. When the installation is complete, the system version must be made *permanent*. When permanent, the new version is used if the system restarts.

It is also possible to install an old version, or reboot the system from an old version. There are functions to remove old releases from the disk as well.

File Structure

The file structure used in an OTP system is described in Release Directories [page 12]. There are two ways of using this file structure together with the release handler.

The simplest way is to store all user-defined applications under `$OTP_ROOT/lib` in the same way as other OTP applications. The release handler takes care of everything, from unpacking a release to the removal of it. The release packets should be stored in the *releases directory* (default `$OTP_ROOT/releases`). This is where `release_handler:unpack_release/1` looks for the packages, and where the release handler stores its files. Each packet is a compressed `tar` file. The files in the `tar` file are named relative to the `$OTP_ROOT` directory. For example, if a new version (say 1.3) of the application `snmp` is contained in the release packet, the files in the `tar` file should be named `lib/snmp-1.3/*`.

The second way is to store all user-defined applications in some other place in the file system. In this case, some more work has to be done outside the release handler. Specifically, the release packages must be unpacked in some way and the release handler must be notified of where the new release is located. The following three functions are available in the module `release_handler` and can be used with this method:

- `set_unpacked/2`
- `set_removed/1`
- `install_file/2`.

Release Installation Files

The following files must be present when a release is installed. All file names are relative to the releases directory. The location of this directory is specified with the configuration parameter `releases_dir` (default `$OTP_ROOT/releases`). In a target system, the default location is preferred, but during testing it may be more convenient to let the release handler write its files at a user specified directory, than in the `$OTP_ROOT` directory.

The files are either present in the release packet, or generated at the target machine and copied to their correct places using `release_handler:install_file/2`.

- `ReleaseName.rel`
- `Vsn/relup`
- `Vsn/start.boot`
- `Vsn/sys.config`

`Vsn` is the system version string, and `EVsn` is the version of the Erlang runtime system.

ReleaseName.rel

This file contains the new name and version of the system, the version of `erts` (the Erlang runtime system) and the libs which are used in the release. The file must contain one Erlang term:

```
{release, {Name, Vsn}, {erts, EVsn},
 [{App, AVsn} | {App, AVsn, AType} | {App, AVsn, [App]} |
    {App, AVsn, AType, [App]}]}.
```

`Name`, `Vsn`, `EVsn` and `AVsn` are strings, `App` and `ATytpe` are atoms. `ReleaseName` is a string given in the call to `release_handler:unpack_release(ReleaseName)`. `Name` is the name of the system (the same as found in the boot file). This file is further described in Release Structure [page 11].

relup

This file contains instructions on how to install this version in the system. This file must contain one Erlang term:

```
{Vsn, [{FromVsn, Descr, RuScript}], [{ToVsn, Descr, RuScript}]}.
```

`Vsn`, `FromVsn` and `ToVsn` are strings, `RuScript` is a release upgrade script. `Descr` is a user defined parameter which is not processed by any release handling functions. It can be used to describe the release to an operator. Eventually, it will be returned by `release_handler:install_release/1` and `release_handler:check_install_release/1`. There is one tuple {`FromVsn, Descr, RuScript`} for each old system version which can be upgraded to this version, and one tuple {`ToVsn, Descr, RuScript`} for each old version that this version can be downgraded to.

When upgrading from `FromVsn` with `release_handler:install_release/1`, there does not have to be an exact match of versions. `FromVsn` can be a sub-string of the current version of the system. For example, if the current version is `"2.1.1"`, we can upgrade from `FromVsn` `"2.1"` or `"2.1.1"`, but not from `"2.0"` or `"2.1.1.2"`. However, if this scheme is used, the same release upgrade script is used to go from both `"2.1"` and `"2.1.1"`. Therefore, `"2.1.1"` must be compatible with `"2.1"`. If you do not want to use this feature, you must make sure that the current version and the new version match before you call `install_release/1`.

start.boot

This file is the compiled `start.script` file. It is used to boot the Erlang machine.

sys.config

This file is the system configuration file.

Release Handling Principles

The following sections describe the principles for updating parts of an OTP system.

Erlang Code

One of the advantages of Erlang is its capability to change code during runtime. This is a somewhat unique feature. However, changing code during runtime cannot be done without considering the application.

The code change feature in Erlang is made possible because Erlang allows two versions of a module to be present in the system. The latest version is called *current* and previous version is called *old*. Any global call to a function, which is a call made by `apply` or pre-fixed with the module name, is always made to the current version of a module. Local calls, which are calls to functions within the same module, always refer to the version of the module in which the call is written.

A process is said to have references to particular versions of a module. A process, of course, has a reference to the version of the module where the function it currently executes is defined, but it also has references to the module versions of all stacked, but not yet finished calls.

Before a new version of a module can be loaded, the current version must be made old. If a previous old version does not exist, this can be done easily. All processes which execute the version which is now old will continue to do so until they have no unfinished calls within the old version. The new version is loaded at the same time as the previous version is made old, and there is no gap when there is no current version. This is all taken care of by Erlang.

If an old version already exists, this version must first be purged to make room for the current version to become old. However, a version cannot be purged if processes still have references to it. If this is the case, these processes must either be terminated, or the loading of the new release must be postponed until these processes have terminated by themselves or no longer have references to the old version. This behaviour can be configured in the release upgrade script for each module.

To prevent processes from making calls to other processes during the release installation, they may be *suspended*. All processes implemented with the standard behaviours, or with `sys`, can be suspended. The process enters a special suspend loop instead of its usual main process loop. In the suspend loop, the process can only receive system messages and shut-down messages from its supervisor. The code change message is a special system message, and this message causes the process to change code to the new version, and maybe to transform its internal state. A suspended process can be *resumed* to continue its execution, possibly in a new module.

There are three different types of modules.

Functional module. A functional module is a module which does not contain a process loop, which means that no process has constant references to a functional module. `lists` is an example of a functional module.

Process module. A process module is a module which contains a process loop, which means that some process has constant references to this module. `init` is an example of a process module.

Callback module. A callback module is a special case of a functional module which serves as a callback module for a generic behaviour such as `gen_server`. These modules are quite similar to process modules in the sense that processes are heavily dependent on them. `file` is an example of a callback module. In the current implementation, a call to a callback module is always a global call (i.e. it refers to the latest version of the module). This has some implications on how updates are handled.

These types of modules are handled differently when changing code.

Functional Module Normally, it is quite easy to change a functional module. If the API is backwards compatible, as may be the case with bug fixes or new functionality, we can simply load the new version. After a short while, when no processes have references to the old version, the old module is purged.

A more tricky situation arises if the API of a functional module is changed so that it is no longer backwards compatible. We must make sure that no processes, directly or indirectly, try to call the functions which have changed. We can do this by suspending the processes. Then, all other modules which are changed to reflect the API change are loaded, the new version of our module is loaded, and finally all the suspended processes are resumed. There are two alternatives available to manage this type of change:

1. Try to find all calls to the module, change them, and write dependencies in your release upgrade script. This may be possible if the function is not called from many other functions.
2. Avoid this type of change. This is the only possible solution if the function is called from many other modules. Instead, introduce a new function and keep the old version for backward compatibility. In the next release, when all other modules are changed as well, you can delete the old version of the function.

Process Module A pure process module never contains global calls to itself. Therefore, the new module can be loaded and all processes which run this module are told to change their code and, if required, to transform their internal state.

In practice, few modules are pure in the sense that they never contain global calls to themselves. If you use higher-order functions such as `lists` in a module, or if you ever use a `fun`, there will be global calls to the module. Therefore, we cannot just load the module because a process might make a spontaneous change to the new version of the module without transforming its state. If the change was totally backwards compatible, as in the case of a bug fix, we can load the module and hope that a process does not run into the buggy code before it performs a spontaneous code change, or we can send a code change message to the process in question. Alternatively, suspend all processes which run the code, tell them to change code, and then resume them.

Callback Module As long as the internal state in a callback module has not changed, we can load the new version of the module. This is the same situation that applies to functional modules.

If the internal state has been changed, we must first suspend the processes, tell them to change code and at the same time give them the possibility to transform their states, and finally resume them. This is similar to the situation that applies to process modules.

Dependencies Among Processes It is possible that a group of processes which communicate must perform code changes while they are suspended. Some of the processes may otherwise use the old protocol while others use the new protocol. On the other hand, there may be time-out dependencies which restrict the number of processes which can perform a synchronized code change as one set. The more processes that are included in the set, the longer the processes are suspended.

There may also be problems with circular dependencies. The following scenario illustrates this situation.

- two modules `a` and `b` are dependent on each other
- each module is executed by one process with the same name as the corresponding module
- both are updated at the same time because the internal protocol between them has changed.

The following sequence of events may occur:

1. `a` is suspended.

2. the release handler then tries to suspend `b`, but some microsecond before this happens, `b` tries to communicate with `a` which is now suspended
3. If `b` hangs in its call to `a`, the suspension of `b` fails and only `a` is updated.
4. If `b` notices that `a` does not answer and is able to deal with it, then `b` receives the suspend message and is suspended. Then both modules are updated and resumed.
5. When `a` resumes, there is a message waiting from `b`. This message may be of an old format which `a` does not recognize.

Situations of the type described, and many others, are highly application dependent. The author of the release upgrade script has to predict and avoid them. If the consequences are too difficult to estimate, it may be better to entirely shut down and restart all affected subsystems. This reduces the problem to introducing new code and removes the need to do a synchronized change.

Finding Processes The release handler finds all processes for you. You only have to specify which modules you want to change, and how to change them, and the release handler searches for the processes. To do this, it checks all processes in the application supervision trees. A supervisor must include a process specification for each child, which lists all modules that the child uses. The release handler checks this list to find the processes.

Port Programs

A port program runs as an external program in the operating system. (This applies unless the program is a linked-in driver, but then it is part of the emulator and we cannot change its code easily.) OTP has mechanisms for changing the code of a port program as well, but this cannot be done as nicely as with Erlang processes. Actually, if the linked-in driver is dynamically linked in using `ddll`, it is possible to change its code.

A port program is changed by sending a special message to the Erlang port controller process. The port controller sends a message to the port program which tells it to terminate and return any data that must survive the termination. When the program has terminated, the new version of the program is started and the termination data is returned from the previous version.

If the Interface Generator (IG) is used, the update is performed automatically by the IG code. An ordinary port controller process which uses `open_port` and communicates directly with the port has to be updated by changing the user code. Refer to Chapter 4 of the Erlang Development Environment User Guide, section *C Interface Generator* for more details.

Application Specification and Configuration Parameters

In each release, the entire application specification is known to the release handler. Before any changes are made to the system, the new configuration parameters are installed. After that the new release is running the applications will be informed of any changed, new or removed configuration parameters, refer to Chapter KERNEL Reference Manual, application(Module), of the Erlang Environment Refence Manual. This means that old processes may read new parameters before they are informed of the new release. We recommend against the immediate removal of the old parameters. Neither do we recommend that they be syntactically changed, although they may of course change their values. They can be safely removed in the next release, by which time you know that no processes will read the old parameters.

Mnesia Data or Schema Changes

The problem with this type of change is the same problem that exists with changing functional modules. Many processes may read or write in the same tables at the same time. If we change a table definition, we must make sure that all code which uses the table changes its code in a synchronized manner.

To best handle this problem, let one process be responsible for one or many tables. This process creates the tables and changes the table definitions or table data. In this way, you can connect a table with a module. When the process performs a code change, the table may be changed.

Upgrade vs. Downgrade

When a new release is installed, the system is *updraged* to the new release. The release handler reads the `relup` file in the new release, and finds the upgrade script that corresponds to an upgrade from the current version to the new version of the system. If an old release is installed, the release handler reads the `relup` in the current release, and finds the downgrade script that corresponds to to an downgrade from the current version to the old version (that is to be installed) of the system. Therefore, when constructing a `relup` file for a new release, there must one upgrade script and one downgrade script for each old version (provided you want to be able to perform soft downgrade of course - an alternative could be to reboot the system from the old release; in that case you don't need a downgrade script).

For each modified module in the new release, there are some instructions that specifies how to install that module in a system. When performing an upgrade, the following steps are typically involved:

1. Suspend the processes running the module
2. Load the new code
3. Tell the processes to change code. This usually involves calling a `code_change` function *in the new module*, which is responsible for state updates, e.g. transforming the state from the old format to the new.
4. Resume the processes

The code change step is always performed when the new code has been loaded. The reason for this is that it is always the new version of the module that knows how to change the state from the old version. When performing a downgrade the situation is different. The old module does not know how to tranform the new state to the old version - the new format is unknown to the old code! Therefore, it is the responsibility of the new code to revert the state back to the old version during downgrade. The following steps are involved:

1. Suspend the processes running the module
2. Tell the processes to change the internal state. This usually involves calling a `code_change` function *in the current module*, which is responsible for state reversals, e.g. transforming the state from the current format to the old.
3. Load the new code
4. Tell the processes to switch code
5. Resume the processes

We note that for a process module, it is possible to load the code before the processes change their internal state (since a process module never contains global calls to itself), thus making the steps needed for downgrade almost the same as for upgrade. The remaining difference with the upgrade case is when the state transformation is performed.

For a callback module it isn't actually necessary to tell the processes to switch to the new code, since all calls to the callback module are global calls. The only difference with the upgrade case the order between the load and state change steps.

The difference between how process modules and a callback modules are handled in the dowgrade case comes from the fact that a process module never contains global calls to itself. The code is thus *static* in the sense that a process executing this module doesn't spontaneously switches to new loaded code. The opposite situation is a *dynamic* module, where a process executing the module spontaneously switches to the new code when it is loaded. Currently, a callback module is always dynamic, and a process module static. A functional module is always dynamic. If you want to handle the downgrade case, this must be specified in the downgrade script.

Release Handling Instructions

This section describes the release upgrade script. A release upgrade script is a list of instructions which are interpreted by the release handler when a new system version is installed.

There are two levels of instructions: the high-level instructions and the low-level instructions. High- and low-level instructions may be mixed in one script. However, the high-level instructions are translated to low-level instructions at compile time, because the release handler only understands low-level instructions.

High-level instructions should be written in an `.appup` file for each application. When a release packet is constructed, `systools:make_relup` is called to generate a `relup` file with low-level instructions.

High-level Instructions

The high-level instructions are defined as follows:

- `{update, Module, Change, PrePurge, PostPurge, [Mod]}` | `{update, Module, Timeout, Change, PrePurge, PostPurge,[Mod]}` | `{update, Module, ModType, Timeout, Change, PrePurge, PostPurge,[Mod]}`
  - `Module = atom()`
  - `Timeout = default | infinity | int() > 0`
  - `ModType = static | dynamic`
  - `Change = soft | {advanced, Extra}`
  - `PrePurge = soft_purge | brutal_purge`
  - `PostPurge = soft_purge | brutal_purge`
  - `Mod = atom()`. If the module is dependent on changes in other modules, these other modules are listed here.

  This instruction is used to update a process module or a callback module. All processes which run the code are suspended, and if the change was `advanced` they may transform their states into the new states. Then the processes are resumed. If the module is dependent on other modules, the release handler will suspend this module before it suspends the other modules. In case of circular dependencies, it will suspend processes in the order that update instructions appear in the script.

  `soft` means backwards compatible changes and `advanced` means internal data changes, or changes which are not backwards compatible. `Extra` is any term, and it is sent to the `code_change` function in all affected modules.

  The optional parameter `Timeout` defines the time-out for the call to `sys:suspend`. It specifies how long to wait for a process to handle a suspend message and get suspended. If no value is specified (or `default` is given), the default value defined in `sys` is used.

  The optional parameter `ModType` specifies if the code is static or dynamic, as defined in Upgrade vs. Downgrade [page 24] above. This information is only necessary if you want to handle soft

downgrades. It defaults to `dynamic`. Note that if this parameter is specified, `Timeout` is needed as well.

`PrePurge` controls what action to take with processes that are executing an old version of this module. These are processes which are left since an earlier release and normally there are no such processes. If it is `soft_purge` and such processes are found, the release will not be installed. In this case, the `install_release/1` function returns `{error, {old_processes, Module}}`. If it is `brutal_purge`, the processes which run old code are killed.

`PostPurge` controls what action to take with processes that are executing old code when the new module has been installed. If it is `soft_purge`, the release handler will purge the old code when no remaining processes execute the code. If it is `brutal_purge`, the code is purged when the release is made permanent. This kills all processes which are still running the old code.

This instruction could also be used for functional modules. However, no processes will be suspended because no processes will have the functional module as its main module. Therefore, no processes may perform a code change.

- `{load_module, Module, PrePurge, PostPurge, [Mod]}`
  - `Module = atom()`.
  - `PrePurge = soft_purge | brutal_purge`
  - `PostPurge = soft_purge | brutal_purge`
  - `Mod = atom()`. If the module is dependent on changes in other modules, these other modules are listed here.

  This instruction is used to update a functional module or a callback module. This instruction only loads the module. A callback module which must perform a code change, or synchronize by being suspended, should use `update` instead.

  The object code is fetched in the beginning of the release upgrade, but the module is loaded when this instruction occurs.
- `{add_module, Mod}` This instruction adds a new module to the system. This instruction loads the module.
- `{remove_application, Appl}` This instruction removes an application. It calls `application:stop` and `application:unload` for the application.
- `{add_application, Appl}` This instruction adds a new application. It calls `application:load` and `application:start` for the application.

Low-level instructions

The low-level instructions are defined as follows:

- `{load_object_code, {Lib, LibVsn, [Module]}}` This instruction reads each `Module` from the library `Lib-LibVsn` as a binary. It does not install the code, it just reads the files. This operation should be placed first in the script in order to read all new code from file. This makes the suspend-load-resume cycle less time consuming. After this operation, the code server is updated with the new version of `Lib`. Calls to `code:priv_dir(Lib)` which are made after this operation return the new `priv` dir.

  `Lib` is normally the application name.
- `point_of_no_return` If a crash occurs after this line, the system cannot recover and is restarted from the old version. This command must only occur once in a script. It should be placed after all `load_object_code` operations, and after user defined checks which are performed with `apply`. The function `check_install_release/1` tries to evaluate all instructions before this command occurs in the script. Therefore, user defined checks must not have side effects, as they may be evaluated many times.

- {load, {Module, PrePurge, PostPurge}} Before using this instruction, the Module object code must be loaded with load_object_code. This instruction makes code out of the binary. PrePurge = soft_purge | brutal_purge, and PostPurge = soft_purge | brutal_purge.
- {remove, {Module, PrePurge, PostPurge}} This instruction makes the current version of the module old. After this operation, there is no current version in the system. PrePurge = soft_purge | brutal_purge, and PostPurge = soft_purge | brutal_purge.
- {purge, [Module]} This instruction kills all processes which run the old versions of the code and deletes the old versions.
- {suspend, [Module | {Module, Timeout}]} This instruction tries to suspend all processes which execute Module. If a process does not respond, it is ignored. This may cause the process to die, either because it crashes if and when it spontaneously switches to new code, or as a result of a purge operation. If no Timeout is specified (or if default is given), the default time-out defined in the module sys is used.
- {code_change, [{Module, Extra}]} | {code_change, Mode, [{Module, Extra}]} This instruction sends a code_change system message using the function change_code in the module sys with the Extra argument to the *suspended* processes which run this code. Mode is either up or down. Default is up. In case of an upgrade, the message is sent to the suspended process, *after* the new code is loaded (the new version must contain functions to convert from the old internal state, to the the new internal state). If it is a downgrade, the message is sent to the suspended process, *before* the new code is loaded (the current version must contain functions to convert from the current internal state, to the the old internal state).

  Module uses the Extra argument internally in its code change function. Refer to the Reference Manual, module sys for further details.

  One of the arguments to the function sys:change_code is OldVsn. In the case of an upgrade, this parameter is the attribute -vsn in the old code, or undefined if no such attribute was defined. In the case of downgrade, it is the tuple {down, Vsn}, where Vsn is the version of the module as defined in the .app file, or undefined otherwise.
- {resume, [Module]} This instruction resumes all previously suspended processes which run this code.
- {stop, [Module]} This instruction stops all processes which run this code. This is useful when the simplest way to change code for the Module is to stop and restart the processes which run the code. If a supervisor is stopped, all its children are stopped as well.
- {start, [Module]} This instruction starts all previously stopped processes which run this code. The processes will keep their places in the supervision tree.
- {sync_nodes, Id, [Node] | {M, F, A}} If {M, F, A} is specified, apply(M, F, A) is evaluated and must return a list of nodes. This operation synchronizes the release installation with other nodes. Each node in the list of nodes must evaluate this command, with the same Id. The node waits for all other nodes to evaluate this operation before execution continues. If a node goes down, this is assumed to be an unrecoverable error and this node is restarted from the old release. There is no time-out for this command. This command may hang forever if a user defined apply enters an infinite loop at some node. It is up to the user to ensure that the apply command eventually returns or makes the node crash.
- {apply, {M, F, A}} This instruction applies the function to the arguments. If this function crashes after the point_of_no_return instruction, the system is restarted. If it is evaluated before the point_of_no_return, it is evaluated within a catch. If the function crashes with {'EXIT', Reason}, the call to release_handler:install_release/1 returns {error, {'EXIT', Reason}}. The function could also do throw({error, Error}), in which case the call to install_release returns {error, Error}.

  Use this instruction with care as this operation makes it possible to do anything.

- `restart_new_emulator` This instruction 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 a crash. This line should be used if a new emulator is introduced, or if a complete reboot of the system should be done.

Release Handling Examples

This section includes several examples to illustrate how common upgrade situations are handled.

Update of Erlang Code

Several update examples are shown. Unless otherwise stated, it is assumed that all original modules are in the application `foo`, version `"1.1"`, and the updated version is `"1.2"`.

Simple Functional Module This example assumes a pure functional module. This is a module which has functions without side effects. The original version of the module `lists2` looks as follows:

```
-module(lists2).
-vsn(1).

-export([assoc/2]).

assoc(Key, [{Key, Val} | _]) -> {ok, Val};
assoc(Key, [H | T]) -> assoc(Key, T);
assoc(Key, []) -> false.
```

The new version of the module adds a new function:

```
-module(lists2).
-vsn(2).

-export([assoc/2, multi_map/2]).

assoc(Key, [{Key, Val} | _]) -> {ok, Val};
assoc(Key, [H | T]) -> assoc(Key, T);
assoc(Key, []) -> false.

multi_map(Func, [[] | ListOfLists]) -> [];
multi_map(Func, ListOfLists) ->
    [apply(Func, lists:map({erlang, hd}, ListOfLists)) |
     multi_map(Func, lists:map({erlang, tl}, ListOfLists))].
```

The release upgrade instructions are:

```
[{load_module, lists2, soft_purge, soft_purge, []}]
```

Alternatively, the low-level instructions are:

```
[{load_object_code, {foo, "1.2", [lists2]}},
 point_of_no_return,
 {load, {lists2, soft_purge, soft_purge}}]
```

A More Complicated Functional Module This example assumes a functional module which uses `lists2`. If it only uses the old functions, we can load the new version independently of `lists2`. The original version looks as follows:

```
-module(bar).
-vsn(1).

-export([simple/1, complicated_sum/1]).

simple(X) ->
    case lists2:assoc(simple, X) of
        {ok, Val} -> Val;
        false -> false
    end.

complicated_sum([X, Y, Z]) -> cs(X, Y, Z).

cs([HX | TX], [HY | TY], [HZ | TZ]) ->
    NewRes = cs(TX, TY, TZ),
    [HX + HY + HZ | NewRes];
cs([], [], []) -> [].
```

The new version of the module uses the new functionality of `lists2` in order to simplify the implementation of the useful function `complicated_sum/1`. It does not change the API in any way.

```
-module(bar).
-vsn(2).

-export([simple/1, complicated_sum/1]).

simple(X) ->
    case lists2:assoc(simple, X) of
        {ok, Val} -> Val;
        false -> false
    end.

complicated_sum(X) ->
    lists2:multi_map(fun(A,B,C) -> A+B+C end, X).
```

The release upgrade instructions, including instructions for `lists2`, are as follows:

```
[{load_module, lists2, soft_purge, soft_purge, []},
 {load_module, bar, soft_purge, soft_purge, [lists2]}]
```

> **Note:**
> We must state that `bar` is dependent on `lists2` so the release handler can load `lists2` before it loads `bar`.

The low-level variant of the instructions are:

```
[{load_object_code, {foo, "1.2", [lists2, bar]}},
 point_of_no_return,
 {load, {lists2, soft_purge, soft_purge}}
 {load, {bar, soft_purge, soft_purge}}]
```

Advanced Functional Module Suppose now that we want to change the return value from `lists2:assoc/2` from `{ok, Val}` to `{Key, Val}`. To solve this, we would have to find all modules (process and callbacks) that may call `lists2:assoc/2` directly or indirectly, and specify that these modules are dependent on `lists2`. In practice, we do not even try to do this, as our code uses `lists2` extensively. Instead, we do this change in a release which restarts the system.

Advanced gen_server This example assumes that we have a `gen_server` process that must be updated because we have introduced a new function, and added a new data field in our internal state. The original module looks as follows:

```
-module(gs1).
-vsn(1).
-behaviour(gen_server).

-export([get_data/0]).
-export([init/1, handle_call/3]).

-record(state, {data}).

get_data() -> gen_server:call(gs1, get_data).

init([Data]) ->
  {ok, #state{data = Data}}.

handle_call(get_data, _From, State) ->
  {reply, {ok, State#state.data}, State}.
```

The new module must translate the old state into the new state. Recall that a record is just syntactic sugar for a tuple:

```
-module(gs1).
-vsn(2).
-behaviour(gen_server).

-export([get_data/0, get_time/0]).
-export([init/1, handle_call/3]).
-export([code_change/3]).
```

```
-record(state, {data, time}).

get_data() -> gen_server:call(gs1, get_data).
get_time() -> gen_server:call(gs1, get_time).

init([Data]) ->
  {ok, #state{data = Data, time = erlang:time()}}.

handle_call(get_data, _From, State) ->
  {reply, {ok, State#state.data}, State};
handle_call(get_time, _From, State) ->
  {reply, {ok, State#state.time}, State}.

code_change(1, {state, Data}, _Extra) ->
  {ok, #state{data = Data, time = erlang:time()}}.
```

The release upgrade instructions look as follows:

```
[{update, gs1, {advanced, []}, soft_purge, soft_purge, []}]
```

The alternative low-level instructions are:

```
[{load_object_code, {foo, "1.2", [gs1]}},
 point_of_no_return,
 {suspend, [gs1]},
 {load, {gs1, soft_purge, soft_purge}},
 {code_change, [{gs1, []}]},
 {resume, [gs1]}]
```

If we want to handle soft downgrade as well, the code would look like:

```
-module(gs1).
-vsn(2).
-behaviour(gen_server).

-export([get_data/0, get_time/0]).
-export([init/1, handle_call/3]).
-export([code_change/3]).

-record(state, {data, time}).

get_data() -> gen_server:call(gs1, get_data).
get_time() -> gen_server:call(gs1, get_time).

init([Data]) ->
  {ok, #state{data = Data, time = erlang:time()}}.

handle_call(get_data, _From, State) ->
  {reply, {ok, State#state.data}, State};
handle_call(get_time, _From, State) ->
  {reply, {ok, State#state.time}, State}.
```

```
code_change(1, {state, Data}, _Extra) ->
  {ok, #state{data = Data, time = erlang:time()}};
code_change({down, 1}, #state{data = Data}, _Extra) ->
  {ok, {state, Data}}.
```

Note that we take care of translating the new state to the old format as well. The corresponding low-level instructions are generated from the same high-level instructions:

```
[{load_object_code, {foo, "1.2", [gs1]}},
 point_of_no_return,
 {suspend, [gs1]},
 {code_change, [{gs1, []}]},
 {load, {gs1, soft_purge, soft_purge}},
 {resume, [gs1]}]
```

Advanced gen_server with Dependencies This example further assumes that we have another gen_server process that uses the new functionality in gs1. Otherwise, it has not changed. This is the same situation as above with the functional modules lists2 and bar, but here there are now processes involved.

The original module looks as follows:

```
-module(gs2).
-vsn(1).
-behaviour(gen_server).

-export([is_operation_ok/1]).
-export([init/1, handle_call/3]).

is_operation_ok(Op) -> gen_server:call(gs2, {is_operation_ok, Op}).

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

handle_call({is_operation_ok, Op}, _From, State) ->
  Data = gs1:get_data(),
  Reply = lists2:assoc(Op, Data),
  {reply, Reply, State}.
```

In this case, the new module must not translate its state as it has not changed. Therefore, it does not have to specify a code_change function.

```
-module(gs2).
-vsn(2).
-behaviour(gen_server).

-export([is_operation_ok/1]).
-export([init/1, handle_call/3]).

is_operation_ok(Op) -> gen_server:call(gs2, {is_operation_ok, Op}).

init([Data]) ->
```

```
  {ok, []}.

handle_call({is_operation_ok, Op}, _From, State) ->
  Data = gs1:get_data(),
  Time = gs1:get_time(),
  Reply = complicated_calculation(lists2:assoc(Op, Data), Time),
  {reply, Reply, State}.
```

The release upgrade instructions are:

```
[{update, gs1, {advanced, []}, soft_purge, soft_purge, []},
 {update, gs2, soft, soft_purge, soft_purge, [gs1]},
```

The low-level alternative instructions are:

```
[{load_object_code, {foo, "1.2", [gs1, gs2]}},
 point_of_no_return,
 {suspend, [gs1, gs2]},
 {load, {gs1, soft_purge, soft_purge}},
 {load, {gs2, soft_purge, soft_purge}},
 {code_change, [{gs1, []}]},    % No gs2 here!
 {resume, [gs1, gs2]}]
```

Other Worker Processes All other worker processes in a supervision tree, such as the process types `gen_event`, `gen_fsm`, and other processes written with `proc_lib` and `sys`, are handled in exactly the same way as gen_server. Some more examples of these kind of processes are shown below.

Simple gen_event This example shows how an event handler may be updated. We do not make any assumptions about which event manager processes the handler is installed in, it is the responsibility of the release handler to find them. The original module looks as follows:

```
-module(ge_h).
-vsn(1).
-behaviour(gen_event).

-export([get_events/1]).
-export([init/1, handle_event/2, handle_call/2]).

get_events(Mgr) -> gen_event:call(Mgr, ge_h, get_events).

init(_) -> {ok, undefined}.

handle_event(Event, _LastEvent) ->
    {ok, Event}.

handle_call(get_events, LastEvent) -> [LastEvent].
```

The new module decides to keep the two latest events in a list and must translate the old state into the new state.

```
-module(ge_h).
-vsn(2).
-behaviour(gen_event).

-export([get_events/1]).
-export([init/1, handle_event/2, handle_call/2]).
-export([code_change/3]).

get_events(Mgr) -> gen_event:call(Mgr, ge_h, get_events).

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

handle_event(Event, []) ->
    {ok, [Event]};
handle_event(Event, [Event1 | _]) ->
    {ok, [Event, Event1]}.

handle_call(get_events, Events) -> Events.

code_change(1, undefined, _Extra) -> {ok, []};
code_change(1, LastEvent, _Extra) -> {ok, [LastEvent]}.
```

The release upgrade instructions are:

```
[{update, ge_h, {advanced, []}, soft_purge, soft_purge, []}]
```

The low-level alternative instructions are:

```
[{load_object_code, {foo, "1.2", [ge_h]}},
 point_of_no_return,
 {suspend, [ge_h]},
 {load, {ge_h, soft_purge, soft_purge}},
 {code_change, [{ge_h, []}]},
 {resume, [ge_h]}]
```

> **Note:**
> These instructions are identical to those used for the `gen_server`.

Process Implemented with sys and proc_lib These processes are changed in the same way as the gen_servers shown above (not surprising since `gen_server` is written using these modules). However, the code change function is defined differently. The original code looks as follows:

```
-module(sp).
-vsn(1).

-export([start/0, get_data/0]).
-export([init/1, system_continue/3, system_terminate/4]).
```

```
-record(state, {data}).

start() ->
    Pid = proc_lib:spawn_link(test, init, [self()]),
    register(sp_server, Pid),
    {ok, Pid}.

get_data() ->
    sp_server ! {self(), get_data},
    receive
        {data, Data} -> Data
    end.

init(Parent) ->
    process_flag(trap_exit, true),
    loop(#state{}, Parent).

loop(State, Parent) ->
    receive
        {system, From, Request} ->
            sys:handle_system_msg(Request, From, Parent, test, [], State);
        {'EXIT', Parent, Reason} ->
            cleanup(State),
            exit(Reason);
        {From, get_data} ->
            From ! {self(), State#state.data},
            loop(State, Parent);
        _Any ->
            loop(State, Parent)
    end.

cleanup(State) -> ok.

%% Here are the sys call back functions
system_continue(Parent, _, State) ->
    loop(State, Parent).

system_terminate(Reason, Parent, _, State) ->
    cleanup(State),
    exit(Reason).
```

The new code, which takes care of up- and downgrade looks as follows:

```
-module(sp).
-vsn(2).

-export([start/0, get_data/0, set_data/1]).
-export([init/1, system_continue/3, system_terminate/4, system_code_change/4]).

-record(state, {data, last_pid}).

start() ->
    Pid = proc_lib:spawn_link(test, init, [self()]),
```

```
    register(sp_server, Pid),
    {ok, Pid}.

get_data() ->
    sp_server ! {self(), get_data},
    receive
        {sp_server, Data} -> Data
    end.

set_data(Data) ->
    sp_server ! {self(), set_data, Data}.

init(Parent) ->
    process_flag(trap_exit, true),
    loop(#state{last_pid = no_one}, Parent).

loop(State, Parent) ->
    receive
        {system, From, Request} ->
            sys:handle_system_msg(Request, From, Parent, test, [], State);
        {'EXIT', Parent, Reason} ->
            cleanup(State),
            exit(Reason);
        {From, get_data} ->
            From ! {sp_server, State#state.data},
            loop(State, Parent);
        {From, set_data, Data} ->
            loop(State#state{data = Data, last_pid = From}, Parent);
        _Any ->
            loop(State, Parent)
    end.

cleanup(State) -> ok.

%% Here are the sys call back functions
system_continue(Parent, _, State) ->
    loop(State, Parent).

system_terminate(Reason, Parent, _, State) ->
    cleanup(State),
    exit(Reason).

system_code_change({state, Data}, _Mod, 1, _Extra) ->
    {ok, #state{data = Data, last_pid = no_one}};
system_code_change(#state{data = Data}, _Mod, {down, 1}, _Extra) ->
    {ok, {state, Data}}.
```

The release upgrade instructions are:

```
[{update, sp, static, default, {advanced, []}, soft_purge, soft_purge, []}]
```

The low-level alternative instructions are the same for upgrade and downgrade:

```
[{load_object_code, {foo, "1.2", [sp]}},
 point_of_no_return,
 {suspend, [sp]},
 {load, {sp, soft_purge, soft_purge}},
 {code_change, [{sp, []}]},
 {resume, [sp]}]
```

Supervisor This example assumes that a new version of an application adds a new process, and deletes one process from a supervisor. The original code looks as follows:

```
-module(sup).
-vsn(1).
-behaviour(supervisor).
-export([init/1]).

init([]) ->
    SupFlags = {one_for_one, 4, 3600},
    Server = {my_server, {my_server, start_link, []},
              permanent, 2000, worker, [my_server]},
    GS1 = {gs1, {gs1, start_link, []}, permanent, 2000, worker, [gs1]},
    {ok, {SupFlags, [Server, GS1]}}.
```

The new code looks as follows:

```
-module(sup).
-vsn(2).
-behaviour(supervisor).
-export([init/1]).

init([]) ->
    SupFlags = {one_for_one, 4, 3600},
    GS1 = {gs1, {gs1, start_link, []}, permanent, 2000, worker, [gs1]},
    GS2 = {gs2, {gs2, start_link, []}, permanent, 2000, worker, [gs2]},
    {ok, {SupFlags, [GS1, GS2]}}.
```

The release upgrade instructions are:

```
[{update, sup, {advanced, []}, soft_purge, soft_purge, []}
 {apply, {supervisor, terminate_child, [sup, my_server]}},
 {apply, {supervisor, delete_child, [sup, my_server]}},
 {apply, {supervisor, restart_child, [sup, gs2]}}]
```

The low-level alternative instructions are:

```
[{load_object_code, {foo, "1.2", [sup]}},
 point_of_no_return,
 {suspend, [sup]},
 {load, {sup, soft_purge, soft_purge}},
 {code_change, [{sup, []}]},
 {resume, [sup]},
 {apply, {supervisor, terminate_child, [sup, my_server]}},
 {apply, {supervisor, delete_child, [sup, my_server]}},
 {apply, {supervisor, restart_child, [sup, gs2]}}]
```

When a supervisor is updated, it is updated as an advanced code change. In the `code_change` function of the supervisor, the new child specification is installed, but no children are explicitly terminated or started. Therefore, we must terminate and start the children with `apply`.

Complex Dependencies As already mentioned, sometimes the simplest and safest way to introduce a new release is to terminate parts of the system, load the new code, and restart that part. However, we cannot just kill the individual processes because their supervisors will probably restart them again before we have loaded the new code. Instead, we must explicitly tell the supervisor not to restart the children. We do this by using the `stop` and `start` instructions.

This example assumes that we have a supervisor `a` with two children `b` and `c`, where `b` is a worker and `c` is a supervisor for `d`. We now want to restart all processes except for `a`. The upgrade instructions look as follows:

```
[{load_object_code, {foo, "1.2", [b,c,d]}},
 point_of_no_return,
 {stop, [b, c]},
 {load, {b, soft_purge, soft_purge}},
 {load, {c, soft_purge, soft_purge}},
 {load, {d, soft_purge, soft_purge}},
 {start, [b, c]}]
```

> **Note:**
> We do not need to explicitly terminate `d`, this is done by the supervisor `c`.

New Application The examples shown so far have dealt with changing an existing application. How do we introduce a new application? We just have to call `application:start_application` and make sure that the application is present in the new boot file that is shipped with each new release. The following example shows how to to introduce the application `new_appl`, which has just one module, `new_mod`.

The release upgrade instructions are:

```
[{add_application, new_appl}]
```

The low-level alternative instructions (in which the application specification is copied) look as follows:

```
[{load_object_code, {new_appl, "1.0", [new_mod]}},
 point_of_no_return,
 {load, {new_mod, soft_purge, soft_purge}},
 {apply, {application, start_application,
           [{application, new_appl,
             [{description, "NEW APPL"},
              {vsn, "1.0"},
              {modules, [new_mod]},
              {registered, []},
              {applications, [kernel, foo]},
              {env, []},
              {mod, {new_mod, start_link, []}}]},
            permanent]}}].
```

Remove an Application An application is removed in the same fashion as new applications are introduced. This example assumes that we want to remove the newly introduced `new_appl`:

```
[{remove_application, new_appl}]
```

The low-variant alternative instructions are:

```
[point_of_no_return,
 {apply, {application, stop, [new_appl]}},
 {remove, {new_mod, soft_purge, soft_purge}}].
```

Update of Port Programs

Each port program is controlled by a Erlang process called the *port controller*. A port program is updated by the port controller process. It is always done by terminating the old port program, and starting the new one.

Port Controller (IG) When the C Interface Generator (IG) is used, Erlang and C code is generated from a header file. In this way, the Erlang module is connected to the port program. Changes are always done simultaneously for the Erlang and C code . This is automatically taken care of by IG in the code change function of the IG process. Suppose that we want to change the IG generated module `portc_ig`. We must make sure that the code change function is called. The release upgrade instructions will look as follows:

```
[{update, portc_ig, {advanced, []}, soft_purge, soft_purge, []}]
```

Or, the low-level variant:

```
[{load_object_code, {foo, "1.2", [portc_ig]}},
 point_of_no_return,
 {suspend, [portc_ig]},
 {load, {portc_ig, soft_purge, soft_purge}},
 {code_change, [{portc_ig, []}]},
 {resume, [portc_ig]}]
```

Port Controller (Ordinary Process) This example assumes that we have an ordinary port controller process. This is a port controller which is not written with IG. In this case, we must take care of the termination and restart of the port program ourselves. Also, we may prepare for the possibility of changing the port program without changing the corresponding Erlang code, and the other way round. We use a `gen_server` to implement the port controller. The original module looks as follows:

```
-module(portc).
-vsn(1).
-behaviour(gen_server).

-export([get_data/0]).
-export([init/1, handle_call/3, handle_info/2, code_change/3]).

-record(state, {port, data}).
```

```
get_data() -> gen_server:call(portc, get_data).

init([]) ->
    PortProg = code:priv_dir(foo) ++ "/bin/portc",
    Port = open_port({spawn, PortProg}, [binary, {packet, 2}])
{ok, #state{port = Port}}.

handle_call(get_data, _From, State) ->
    {reply, {ok, State#state.data}, State}.

handle_info({Port, Cmd}, State) ->
    NewState = do_cmd(Cmd, State),
    {noreply, NewState}.

code_change(_, State, change_port_only) ->
    State#state.port ! close,
    receive
        {Port, closed} -> true
    end,
    NPortProg = code:priv_dir(foo) ++ "/bin/portc",   % get new version
    NPort = open_port({spawn, NPortProg}, [binary, {packet, 2}]),
    {ok, State#state{port = NPort}}.
```

To change the port program without changing the Erlang code, we can use the following code:

```
[point_of_no_return,
 {suspend, [portc]},
 {code_change, [{portc, change_port_only}]},
 {resume, [portc]}]
```

In this situation, we use the low-level instructions directly. It is also the first example that uses the Extra argument.

Suppose now that we wish to change the Erlang code instead. The new Erlang module looks as follows:

```
-module(portc).
-vsn(2).
-behaviour(gen_server).

-export([get_data/0]).
-export([init/1, handle_call/3, handle_info/2, code_change/3]).

-record(state, {port, data}).

get_data() -> gen_server:call(portc, get_data).

init([]) ->
    PortProg = code:priv_dir(foo) ++ "/bin/portc",
    Port = open_port({spawn, PortProg}, [binary, {packet, 2}])
{ok, #state{port = Port}}.

handle_call(get_data, _From, State) ->
    {reply, {ok, State#state.data}, State}.
```

```
handle_info({Port, Cmd}, State) ->
    NewState = do_cmd(Cmd, State),
    {noreply, NewState}.

code_change(_, State, change_port_only) ->
    State#state.port ! close,
    receive
        {Port, closed} -> true
    end,
    NPortProg = code:priv_dir(foo) ++ "/bin/portc",   % get new version
    NPort = open_port({spawn, NPortProg}, [binary, {packet, 2}]),
    {ok, State#state{port = NPort}};
code_change(1, State, change_erl_only) ->
    NState = transform_state(State),
    {ok, NState}.
```

The relup instructions are:

```
[{update, portc, {advanced, change_erl_only}, soft_purge, soft_purge, []}]
```

The low-level alternative instructions are:

```
[{load_object_code, {foo, "1.2", [portc]}},
 point_of_no_return,
 {suspend, [portc]},
 {load, {portc, soft_purge, soft_purge}},
 {code_change, [{portc, change_erl_only}]},
 {resume, [portc]}]
```

SASL Reference Manual

Short Summaries

sasl

No functions are exported

alarm_handler

The following functions are exported:

- `clear_alarm(AlarmId) -> void()`
  [page 50] Clears the specified alarms
- `get_alarms() -> [alarm()]`
  [page 50] Gets all active alarms
- `set_alarm(alarm())`
  [page 50]

overload

The following functions are exported:

- `request() -> accept | reject`
  [page 53] Requests to proceed with current job
- `get_overload_info() -> OverloadInfo`
  [page 53] Returns current overload information data

rb

The following functions are exported:

- `grep(RegExp)`
  [page 54] Searches the reports for a regular expression
- `h()`
  [page 54] Prints help information
- `help()`
  [page 54] Prints help information
- `list()`
  [page 54] Lists all reports
- `list(Type)`
  [page 54] Lists all reports
- `rescan()`
  [page 54] Rescans the report directory
- `rescan(Options)`
  [page 54] Rescans the report directory
- `show()`
  [page 54] Shows reports
- `show(Report)`
  [page 54] Shows reports
- `start()`
  [page 55] Starts the RB server
- `start(Options)`
  [page 55] Starts the RB server
- `start_log(FileName)`
  [page 55] Redirects all output to `FileName`
- `stop()`
  [page 55] Stops the RB server
- `stop_log()`
  [page 55] Stops logging to file

release_handler

The following functions are exported:

- `check_install_release(Vsn) -> {ok, FromVsn, Descr} | {error, Reason}`
  [page 57] Checks installation of the release in the system
- `create_RELEASES(Root, RelDir, RelFile, LibDirs) -> ok | {error, Reason}`
  [page 58] Creates an initial RELEASES file
- `install_file(Vsn, FileName) -> ok | {error, Reason}`
  [page 58] Installs a release file in the release handler
- `install_release(Vsn) -> {ok, FromVsn, Descr} | {error, Reason}`
  [page 58] Installs the release in the system
- `install_release(Vsn, Opt) -> {ok, FromVsn, Descr} | {error, Reason}`
  [page 58] Installs the release in the system
- `make_permanent(Vsn) -> ok | {error, Reason}`
  [page 59] Makes the specified release to be used at system start-up
- `remove_release(Vsn) -> ok | {error, Reason}`
  [page 59] Deletes all files unique for this release
- `reboot_old_release(Vsn) -> ok | {error, Reason}`
  [page 59] Reboots the system from an old release
- `set_removed(Vsn) -> ok | {error, Reason}`
  [page 59] Marks a release as removed
- `set_unpacked(RelFile, LibDirs) -> {ok, Vsn} | {error, Reason}`
  [page 59] Marks a release as unpacked
- `unpack_release(ReleaseName) -> {ok, Vsn} | {error, Reason}`
  [page 59] Unpacks and extracts files from the release package
- `which_releases() -> [{Name, Vsn, [Lib], Status}]`
  [page 60] Returns all known releases

systools

The following functions are exported:

- `behaviour_info() -> [Behaviour]`
  [page 61] Lists the system defined behaviours
- `behaviour_info(Behaviour) -> [Function]`
  [page 61] Lists the functions that a behaviour uses
- `make_script(ReleaseName) -> MakeRet`
  [page 61] Creates a boot script from a release file
- `make_script(ReleaseName,Opts) -> MakeRet`
  [page 61] Creates a boot script from a release file
- `make_relup(ReleaseName,UpNameList,DownNameList) -> RelRet`
  [page 64] Gathers release upgrade scripts for a release
- `make_relup(ReleaseName,UpNameList,DownNameList,Opts) -> RelRet`
  [page 64] Gathers release upgrade scripts for a release

- `make_tar(ReleaseName) -> TarRet`
  [page 65] Creates a release package.
- `make_tar(ReleaseName,Opts) -> TarRet`
  [page 65] Creates a release package.
- `script2boot(File) -> ok | error`
  [page 66] Generate a binary form of a boot script.

appup

No functions are exported

rel

No functions are exported

relup

No functions are exported

script

No functions are exported

sasl (Application)

This section describes the `sasl` application which is included with the Erlang system/Open Telecom Platform. The SASL application provides the following services:

- `alarm_handler`
- `overload`
- `release_handler`

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 report*, *crash report* and *progress report* to `stdio`.

`sasl_report_file_h` Formats and writes *supervisor report*, *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.

Configuration

The following configuration parameters are defined for the SASL application. Refer to application(3) 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 obsolete.

`client_directory = string() <optional>` This parameter specifies the client directory at the master nodes. Refer to Release Handling in the Erlang Development Environment User's Guide for more information. This parameter is ignored if the `masters` parameter is obsolete.

`static_emulator = true | false <optional>` Indicates if the Erlang emulator is statically installed. A node with a static emulator can not 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 obsolete.

`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`.

SNMP MIBs

The following MIBs are defined in SASL:

OTP-REG This MIB contains the top-level OTP registration objects, used by all other MIBs.

OTP-TC This MIB contains the general Textual Conventions, which can be used by any other MIB.

OTP-MIB This MIB contains objects for instrumentation of the Erlang nodes, the Erlang machines and the applications in the system.

The MIBs are stored in the `mibs` directory. All MIBs are defined in SNMPv2 SMI syntax. SNMPv1 versions of the mibs are delivered in the `mibs/v1` directory.

The compiled MIBs are located under `priv/mibs`, and the generated `.hrl` files under the `include` directory. To compile a MIB that IMPORTS the `OTP-MIB`, give the option `{il, ["sasl/priv/mibs"]}` to the MIB compiler.

The only MIB with Managed Objects is `OTP-MIB`. If this MIB should be used in a system, it should be loaded into an agent with a call to `otp_mib:init(Agent)`, where `Agent` is the Pid or registered name of an SNMP agent. Use `otp_mib:stop(Agent)` to unload the MIB. The implementation of this MIB uses Mnesia to store a cache with data needed. This means that Mnesia must run if this implementation of the MIB should be used.

See Also

alarm_handler(3), error_logger(3), log_mf_h(3), overload(3), release_handler(3), systools(3), appup(4), rel(4), relup(4), script(4), application(3), snmp(6)

alarm_handler (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, but rather it defines a place to which alarms can be sent. One simple event handler is installed in the alarm handler at start-up, and 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 can be installed at a later stage. This list can grow large if many alarms are generated. This is one 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 (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 (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 this 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}
- FileName = string() | standard_io
- MaxNoOfReports = int() | all
- DirString = string()
- ReportType = type() | [type()] | all

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'.

`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 with be directed to `standard_io`.

release_handler (Module)

The release handler process is a SASL process that handles unpacking, installation, and removal of release packages. As an example, a release package could contain applications, a new emulator, and new configuration parameters. In this text, the directory `ROOT` refers to the installation root directory (`code:root_dir()`). A release package is a compressed `tar` file that is written to the *releases* directory, for example via ftp. The location of this directory is specified with the configuration parameter `releases_dir`, or the OS environment variable `RELDIR`. Default is `ROOT/releases`. The release handler must have write access to this directory in order to install new releases. The persistent state of the release handler, for example information about installed releases, is stored in a file called `RELEASES` in the releases directory.

The package can be *unpacked*, which extracts the files from the package. When the release is unpacked, it can be *installed*. This operation evaluates the release upgrade script. 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 when the system is started. 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. The system can be *rebooted* from an old release.

Each 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	-

A release package always contains two special files, the `ReleaseName.rel` file and the `relup` file. The `ReleaseName.rel` file contains information about the release, such as its name, version, and which system and library versions it uses. The `relup` file contains release upgrade scripts. There is one release upgrade script for each old version that can be updated to the new version.

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.

A new release may restart the system, using `start_prg`. This is a configuration parameter to the application `sasl`. The default is `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 is 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.

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, FromVsn, Descr} | {error, Reason}`

Types:

- Vsn = FromVsn = string()
- Descr = term()

The release must not have status `current`. Checks that there is a `relup` release upgrade script from the `FromVsn` (current version) to `Vsn`. Checks that all required libs (or applications) are present and that all new code can be loaded. Checks that there is a `start.boot` file and a `sys.config` for the new release.

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`.

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

Types:

- Root = RelDir = RelFile = string()
- LibDirs = [{LibName, LibVsn, Dir}]
- LibName = atom()
- LibVsn = Dir = string()

This function can be called to create an initial RELEASES file to be used by the release_handler. This file must exist in order to install new releases. When the system is installed, a default RELEASES file is created in the default releases directory `ROOT/releases`.

`Root` is the root of the installation as described above. `RelDir` is the the releases directory where the RELASES file should be created. `RelFile` is the name of the `.rel` file that describes the initial release.

`LibDirs` can be used to specify from where the modules for an application should be loaded. `LibName` is the name of the lib (or application), `LibVsn` is the version, and `Dir` is the name of the directory where the lib directory `LibName-LibVsn` is located. The corresponding modules should be located under `Dir/LibName-LibVsn/ebin`.

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

Types:

- FileName = string()
- Vsn = string()

Installs a release dependent file in the release structure. A release dependent file is a file that must be in the release structure when the release is installed. Currently there are three such mandatory files, `start.boot`, `sys.config` and `relup`.

This function should be called to install release dependent files, for example when these files are generated at the target. It should be called when `set_unpacked/2` has been called.

`install_release(Vsn) -> {ok, FromVsn, Descr} | {error, Reason}`
`install_release(Vsn, Opt) -> {ok, FromVsn, Descr} | {error, Reason}`

Types:

- Vsn = FromVsn = string()
- Opt = [{error_action, Error_action}]
- Error_action = restart | reboot
- Descr = term()

The release must not have status `current`. Installs the delivered release in the system by evaluating the release upgrade script found in the `relup` file. This function returns `{ok, FromVsn, Descr}` if successful, or `{error, Reason}` if a recoverable error occurs. `Descr` is a user defined parameter, found in the `relup` file, used to describe the release. The system is restarted if a non-recoverable error occurs. There can be many installed releases at the same time in the system.

It is possible to define if the node should be restarted or rebooted in case of an error during the installation. Default is `restart`.

Note that if an `old` or the `permanent` release is installed, a downgrade will occur. There must a correspnding downgrade script in the `relup` file.

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

Types:

- Vsn = string()

Makes the current release permanent. This causes the specified release to be used at system start-up.

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

Types:

- Vsn = string()

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 = {no_such_release, Vsn}

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 = {no_such_release, Vsn} | {permanent, Vsn}

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

Types:

- RelFile = string()
- LibDirs = [{LibName, LibVsn, Dir}]
- LibName = atom()
- LibVsn = Dir = string()
- Vsn = string()

Makes it possible to handle the unpacking of releases outside the `release_handler`. Makes the `release_handler` aware that the release is unpacked. `Vsn` is extracted from the release file `RelFile` and is used as parameter to the other functions.

`LibDirs` can be used to specify from where the modules for an application should be loaded. `LibName` is the name of the lib (or application), `LibVsn` is the version, and `Dir` is the name of the directory where the lib directory `LibName-LibVsn` is located. The corresponding modules should be located under `Dir/LibName-LibVsn/ebin`.

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

Types:

- ReleaseName = string()
- Vsn = string()

The `ReleaseName` is the name of the release package. This is the name of the package file, without `.tar.gz`. `ReleaseName` may or may not be the same as the release version. `Vsn` is extracted from the release package and is used as parameter to the other functions.

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

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

Types:

- Name = string()
- Vsn = string()
- Lib = string()
- Status = unpacked | current | permanent | old

Returns all releases known to the release handler. `Name` is the name of the system. `Lib` is the name of a library. This name may be the application name followed by its version, for example “kernel-1.0”.

See Also

systools(3)

systools (Module)

This module contains functions to generate boot scripts, release upgrade scripts, and release packages. A release file (.rel), application definition files (.app), and application upgrade files (.appup) are required as input to these functions. The syntax definitions for these files can be found in the Erlang Development Environment User's Guide.

If a boot script is written without using the generator, it can be transformed to a binary form with the `script2boot/1` function, as required by the Erlang system during start-up.

The behaviour functions described below can be used to obtain a list of the system defined behaviours, and information about which callback functions are required for each of them.

Exports

`behaviour_info() -> [Behaviour]`

Types:

- Behaviour = atom()

Returns a list of the behaviours defined in the Erlang system. `gen_server` and `gen_event` are examples of behaviours.

`behaviour_info(Behaviour) -> [Function]`

Types:

- Behaviour = atom()
- Function = {Name, Arity}
- Name = atom()
- Arity = int()

A behaviour calls a number of functions in the callback module. The functions that a callback module has to export are returned by this function. `Behaviour` is the same as returned from the `behaviour_info/0` function.

`make_script(ReleaseName) -> MakeRet`
`make_script(ReleaseName,Opts) -> MakeRet`

Types:

- ReleaseName = string()
- Opts = [{path, Path} | silent | local | no_module_tests | {variables, Vars} | {machine, Machine} | exref | {exref, [AppName]}]

- Path = [Dir]
- Dir = string()
- Vars = [Var]
- Var = {VarName, PreFixDir}
- VarName = atom() | string()
- PreFixDir = string()
- Machine = atom()
- AppName = atom()
- MakeRet = ok | error | {ok, Module, Warnings} | {error, Module, Error}
- Warnings = void()
- Module = atom()
- Error = void()

A boot script file is generated from the `ReleaseName`.rel file. The `ReleaseName`.script and `ReleaseName`.boot files are generated. The release file contains a specification of the version of the release, and the name and version of the applications that are included.

The script generator searches the normal code server path for the `ReleaseName`.rel file and the application files `ApplicationName.app`. A {`path, Path`} option can be specified and appended to the code server path. Each directory in `Path` can be given with the wildcard `*` (* is the only wildcard recognized). A directory given with wildcards is expanded to all matching directories. `*` is translated to "any character except /". If `/*/` is specified - `*` is the only character given between two `/` characters - the corresponding regular expression is `[^/]+` and it represents a directory.

The compiled Erlang modules should be located in the same directory as the `.app` file. The function searches for the source code in the corresponding `src` or `src/e_src` directory if the directory name of the `.app` file directory ends with `/ebin`. Otherwise, it searches for the source code in the `.app` file directory.

The correctness of each application is checked. The following checks are performed:

- The version of the application file found.
- Dependencies to applications not included in the release.
- Circular dependencies among applications.
- Duplicated module names.
- Version compliance between modules and versions specified in the application file.
- Currency of object code for each module.

The boot script is generated if all checks are satisfactory. The applications are loaded and started in the order specified in the release file. The exception to this order are dependencies between applications as specified in the application files. These dependencies specify that applications on which other applications depend must be started first.

If the `no_module_tests` option is specified, the module version and object code checks are excluded. This implies that a boot script can be generated without the requirement that each `.app` file must be located in the same directory as the modules which belong to the application.

The checks performed before the boot script is generated can be extended with some rudimentary cross reference checks by specifying the `exref` option. These checks are performed with the `exref` tool. All modules specified in the application resource files are loaded into the `exref` tool. A warning is generated for each call to an undefined

function, but only explicit function calls are checked. No cross reference checks are performed if the `exref` option is specified in combination with the `no_module_tests` option.

As the cross reference checks can be heavy, the set of modules to be checked can be limited. The `{exref, [AppName]}` option specifies the applications in which modules should be cross referenced checked. One warning only is generated for each application whenever calls are found to functions in applications which are not cross reference checked.

The generated boot script contains a search path to all included applications. By default, all directories in the path are relative to the installation directory of the Erlang system which uses the boot script.

The `variables` option can be used to specify an installation directory other than the Erlang installation directory for user provided applications. If the option `{variables, [{"TEST","/home/xxx/applications"}]}` is supplied, all applications found underneath this directory will have `$TEST` substituted in place of the directory. The variable substitution mechanism needs absolute paths. Therefore, the paths specified (either in the code server path, or with the `path` option) must be absolute. The following example illustrates this:

```
/home/xxx/applications/type1/app1/ebin
                            /app2/ebin
                       type2/app3/ebin
                       app4/ebin
```

The boot script is generated as:

```
systools:make_script(RelName,
                     [{path,["/home/xxx/applications/*/ebin"]},
                      {variables,[{"TEST","/home/xxx/applications"}]}])
```

In the generated boot script, the path looks as follows for the applications `app1` - `app4`:

```
[...
 "$TEST/type1/app1-Vsn/ebin",
 "$TEST/type1/app2-Vsn/ebin",
 "$TEST/type2/app3-Vsn/ebin",
 "$TEST/app4-Vsn/ebin"]
```

When starting the system with the generated boot script, the `TEST` variable is given a value using the `-boot_var Var Value` command line flag. In the previous example, `Var` is `TEST` and `Value` is the name of the directory where these applications are installed. The `-boot_var` flag is described for the `init` module.

The `local` option can be used to change the default path as well. If the `local` option is supplied, the path includes the actual directories where the applications were found. This is a useful way to test a generated boot script *locally*.

The `machine` option can be used to generate a boot script for an Erlang machine other than the running machine. This is important when checking the object code, as the file extension can differ between the machines (for example `.jam` or `.beam`).

By default, this function writes all errors and warnings to the tty and returns `ok` or `error`. Nothing is written to the tty if the `silent` option is supplied, but the function returns `{ok, Module, Warnings}` or `{error, Module, Errors}` instead. To convert the `Warnings` and `Errors` terms to strings, the `Module`:format_warning(`Warnings`) and `Module`:format_error(`Errors`) functions are called respectively.

```
make_relup(ReleaseName,UpNameList,DownNameList) -> RelRet
make_relup(ReleaseName,UpNameList,DownNameList,Opts) -> RelRet
```

Types:

- ReleaseName = string() | atom()
- UpNameList = NameList
- DownNameList = NameList
- NameList = [ReleaseName | {ReleaseName, Description}]
- Description = term()
- Opts = [{path, Path} | silent | noexec]
- Path = [Dir]
- Dir = string()
- RelRet = ok | error | {ok, Relup, Module, Warnings} | {error, Module, Error}
- Relup = {Vsn, UpScript, DownScript}
- UpScript = RelupScript
- DownScript = RelupScript
- RelupScript = [{Vsn, Description, Script}]
- Script = [low_level_release_upgrade_instructions]
- Warnings = void()
- Module = atom()
- Error = void()

A `relup` file is generated which describes how to upgrade the system from a number of previous releases, and also how to downgrade from a number of previous releases.

The `relup` file is built by gathering all the application release upgrade scripts and picking those applicable for each combination of release versions. The scripts are also translated from high level release instructions to low level instructions. The normal code server path is searched for release files (`ReleaseName.rel`) and application files (`ApplicationName.app`), as well as the application upgrade scripts files (`ApplicationName.appup`). The `ApplicationName.app` and `ApplicationName.appup` files must be in the same directory. The code server path can be appended with a path specified with the `{path, Path}` option. `Path` can contain wildcards (*) as described for the `make_script` function.

A `ReleaseName`.rel file must be available for each `UpName` and `DownName` since the versions of the applications are compared. For each change in the application versions, there must be an entry in the `Application`.appup file.

The optional `Description` parameter which can be supplied to either of the input name lists is passed to the correct output script in the `relup` file. The parameter defaults to the empty list `[]`.

Basically, `make_relup` combines a re-ordering of the `ReleaseName`.rel file and the `Application`.appup files, so that the new release version and a target release version is a list of release upgrade scripts for all applications that have changed between the two release versions.

By default, this function writes the relup script to a file named `relup` and all errors and warnings to the tty and returns `ok` or `error`. If the `silent` option is supplied, nothing is written to the tty and the function returns `{ok, Relup, Module, Warnings}` or `{error, Module, Error}` instead, where `Relup` is the structure written to the `relup` file. The Warnings and Errors can be converted to strings with the `Module`:format_warning(Warning) and `Module`:format_error(Error) functions. If the

noexec option is supplied, then nothing is written to the relup file and the function returns one of the verbose return values.

```
make_tar(ReleaseName) -> TarRet
make_tar(ReleaseName,Opts) -> TarRet
```

Types:

- ReleaseName = string()
- Opts = [{path, Path} | silent | {dirs, Dirs} | {erts, ErtsDir} | no_module_tests | {variables, Vars} | {var_tar, VarTar} | {machine, Machine} | exref | {exref, [AppName]}]
- Path = [Dir]
- Dir = string()
- Dirs = [atom()]
- ErtsDir = string()
- Vars = [Var]
- Var = {VarName, PreFixDir}
- VarName = atom() | string()
- PreFixDir = string()
- VarTar = include | ownfile | omit
- Machine = atom()
- AppName = atom()
- TarRet = ok | error | {ok, Module, Warnings} | {error, Module, Error}
- Warnings = void()
- Module = atom()
- Error = void()

A release package file is generated from the ReleaseName.rel file. The ReleaseName.tar.gz file is generated. This file can then be uncompressed and unpacked on the target system before the new release can be activated, using the release_handler.

By default, the generated release package contains a directory under the lib directory for each included application . Each application directory is named ApplicationName-ApplicationVsn. For each application, the ebin and priv directories are included. These directories are copied from where the applications were found. If more directories are needed, it is possible to specify these with the {dirs,Dirs} option. For example, if the src and example directories should be included for each application in the release package, the {dirs,[src,examples]} option should be supplied.

The variables option can be used to specify an installation directory other than the Erlang installation directory for the user provided applications. If the option{variables, [{"TEST","/home/xxx/applications"}]} is supplied, all applications found underneath this directory will be packed into the TEST.tar.gz file. Accordingly, a separate package is created for each defined variable. By default, all these files are included at the top level in the ReleaseName.tar.gz file and should be unpacked to an appropriate installation directory. 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.

The normal code server path is searched for the release file `ReleaseName`.rel and the application files (`ApplicationName.app`). The code server path can be appended with a path specified with the {`path, Path`} option. `Path` can contain wildcards (*) as described for the `make_script` function.

The `machine` option can be used to generate a release package file for an Erlang machine other than the running machine. This ensures that object code files with the expected file extension are included in the package, for example `.jam` or `.beam` files.

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 the boot script (`ReleaseName`.boot copied to `start.boot`), the `relup` file (generated by `make_relup`), and the system configuration file (`sys.config`).

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

Finally, the `releases` directory contains the `ReleaseName`.rel file.

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 `TarRet` and the handling of errors and warnings are as described for the `make_script` function above.

`script2boot(File) -> ok | error`

Types:

- File = string()

The Erlang 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

release_handler(3), init(3), exref(3)

appup (File)

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

This file is used by `systools` to generate release upgrade files.

FILE SYNTAX

Applications can be upgraded and the instructions to do this are placed in the `.appup` file for the application. For example, for the `snmp` application these instructions are placed in the `snmp.appup` file. The `.appup` file looks as follows:

The application upgrade file is called `Name.appup` where `Name` 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 the application. The file has the following syntax:

```
{Vsn,
    [{UpFromVsn, UpFromScript}, ...],
    [{DownToVsn, DownToScript}, ...]}.
```

- `Vsn = string()` is the current version of the application.
- `UpFromVsn = string()` is a version we can upgrade from.
- `UpFromScript` is the script which describes the sequence of release upgrade instructions. Refer to the section Release Handling Instructions in the SASL User's Guide for a description of this script.
- `DownToVsn = string()` is a version to which we can downgrade.
- `DownToScript` is the script which describes the sequence of downgrade instructions. Refer to the section Release Handling Instructions in the SASL User's Guide for a description of this script.

In the case of `UpFrom Script` and `DownFromScript`, the scripts typically contain one line for each module in the application.

SEE ALSO

app(4), relup(4), systools(3)

rel (File)

The *release resource file* describes each release of an entire system based on OTP. This file defines which applications are included in a certain version of the system.

This file is used by `systools` to generate start scripts and release upgrade files.

Releases can also be upgraded and instructions for this should be written in the `relup` file (see the definition of the relup file). The tedious work of writing the `relup` file has been automated and in most cases the file can be automatically generated from the `.appup` files for the applications in the release.

FILE SYNTAX

A release resource file is called `RelName.rel` where `RelName` is the name of the release.

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

```
{release, {Name,Vsn}, {erts, EVsn},
  [{AppName, AppVsn} |
   {AppName, AppVsn, AppType} |
   {AppName, AppVsn, IncApps} |
   {AppName, AppVsn, AppType, IncApps}]}.
```

- `Name = string()` is the name of the release `Name` need not be the same as `RelName` above in the file name.
- `Vsn = string()` is the version of the release.
- `EVsn = string()` indicates which Erlang runtime system version `EVsn` the release is intended for, for example “4.4”.
- `AppName = atom()` is the name of an application included in the release.
- `AppVsn = string()` is the version of the `AppName` application.
- `AppType = permanent | transient | temporary | load | none` is the start type of the `AppName` application. This parameter specifies how the application is treated in the systools-generated start script. If it is `permanent`, `transient` or `temporary`, the application is started with a call to `application:start(AppName, AppType)`. If it is `load`, the application is loaded, but not started. If it is `none`, the application is neither loaded nor started.
- `IncApps = [atom()]` is a list of applications that are included by an application, for example `[AppName, ...]`. This list overrides the `included_applications` key in the application resource file `.app`. This list must be a subset of the list of included applications which are specified in the `.app` file.

> **Note:**
> The list of applications must contain the `kernel` and the `stdlib` applications.

SEE ALSO

app(4), appup(4), relup(4), systools(3)

relup (File)

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

This file is used by `systools` to generate start scripts and release upgrade files.

The tedious work of writing the `relup` file has been automated and in most cases this file can be automatically generated from the `.rel` file and `.appup` files for the applications in the release.

FILE SYNTAX

A release upgrade file is called `relup`. In the target system, this file must be located in the `OTP_ROOT/erts-EVsn/Vsn` directory.

The `relup` file contains one single Erlang term, which is contains instructions on how to upgrade from old versions to this version, and how to downgrade from this version to older versions. The file has the following syntax:

```
{Vsn, [{FromVsn, Descr, RuScript}], [{ToVsn, Descr, RuScript}]}.
```

- `Vsn = string()` is the version of the release.
- `FromVsn = string()` is a version of a release that we can upgrade from. If the current version of the system matches this version, the corresponding upgrade instructions in `RuScript` is used to install the release in the system.
- `ToVsn = string()` is a version of a release that we can downgrade to. If this release (`Vsn`) is the current release, and we are about to downgrade to `ToVsn`, the corresponding upgrade instructions in `RuScript` is used to install the old release in the system.
- `Descr` is a user defined parameter which is not processed by any release handling functions. It can be used to describe the release to an operator. Eventually, it will be returned by `release_handler:install_release/1` and `release_handler:check_install_release/1`.
- `RuScript` is a release upgrade script. Refer to the section Release Handling Instructions in the SASL User's Guide for a description of this script.

There is one tuple {`FromVsn, Descr, RuScript`} for each old system version which can be upgraded to this version, and one tuple {`ToVsn, Descr, RuScript`} for each old version that this version can be downgraded to.

When upgrading from `FromVsn` with `release_handler:install_release/1`, there does not have to be an exact match of versions. `FromVsn` can be a sub-string of the current version of the system. For example, if the current version is `"2.1.1"`, we can upgrade from `FromVsn` `"2.1"` or `"2.1.1"`, but not from `"2.0"` or `"2.1.1.2"`. However, if this scheme is used, the same release upgrade script is used to go from both `"2.1"` and

`"2.1.1"`. Therefore, `"2.1.1"` must be compatible with `"2.1"`. If you do not want to use this feature, you must make sure that the current version and the new version match before you call `install_release/1`.

SEE ALSO

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

script (File)

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

The command `erl -boot Namee` 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)

List of Tables

Index

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