System Application Support Libraries (SASL)

version 1.8

OTP Team

1999-04-22

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

Contents

Chapter 1

SASL User's Guide

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

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

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

1.1 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.

Prerequisites

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.

1.2 About This Book

In addition to this introductory part, the 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 file. 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

1.3 Where to Find More Information

Refer to the following documentation for more information:

- the Erlang Development Environment User's Guide
- the Erlang Extensions Since 4.4 User's Guide
- the Erlang Development Environment 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.

Chapter 2

SASL Error Logging

The SASL application introduces three types of reports:

- supervisor report
- progress report
- crash report.

When the SASL application is started, it adds a handler that formats and writes these reports, as specified in the configuration parameters for sasl, i.e the environment variables in the SASL application specification, which is found in the `.app` file of SASL. See sasl(Application) [page 63], and app(File) in the Kernel Reference Manual for the details.

2.1 Supervisor Report

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

Supervisor. The name of the reporting supervisor.

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

Reason. The termination reason.

Offender. The start specification for the child.

2.2 Progress Report

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

Supervisor. The name of the reporting supervisor.

Started. The start specification for the successfully started child.

2.3 Crash Report

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

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

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

An Example

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

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

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

=SUPERVISOR REPORT==== 27-May-1996::13:38:56 ===
Supervisor: {local,test_sup}
Context:    child_terminated
Reason:     {badarith,{test,s,[]}}
Offender:   [{pid,<0.63.0>},
{name,test},
```

```
{mfa,{test,t,[]}},
{restart_type,permanent},
{shutdown,200},
{child_type,worker}]


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

2.4 Multi-File Error Report Logging

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

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

2.5 Report Browser

The report browser is used to browse and format error reports written by the error logger handler `error_logger_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().* to access the report browser on-line help system.

List Reports in the Server

The function `rb:list()` lists all loaded reports:

```
4> rb:list().
No                Type          Process       Date     Time
==                ====          =======       ====     ====
20            progress        <0.17.0> 1996-10-16 16:14:54
19            progress        <0.14.0> 1996-10-16 16:14:55
18               error        <0.15.0> 1996-10-16 16:15:02
17            progress        <0.14.0> 1996-10-16 16:15:06
16            progress        <0.38.0> 1996-10-16 16:15:12
15            progress        <0.17.0> 1996-10-16 16:16:14
14            progress        <0.17.0> 1996-10-16 16:16:14
13            progress        <0.17.0> 1996-10-16 16:16:14
12            progress        <0.14.0> 1996-10-16 16:16:14
11               error        <0.17.0> 1996-10-16 16:16:21
```

```
10              error         <0.17.0> 1996-10-16 16:16:21
9       crash_report  release_handler 1996-10-16 16:16:21
8   supervisor_report         <0.17.0> 1996-10-16 16:16:21
7           progress         <0.17.0> 1996-10-16 16:16:21
6           progress         <0.17.0> 1996-10-16 16:16:36
5           progress         <0.17.0> 1996-10-16 16:16:36
4           progress         <0.17.0> 1996-10-16 16:16:36
3           progress         <0.14.0> 1996-10-16 16:16:36
2              error         <0.15.0> 1996-10-16 16:17:04
1           progress         <0.14.0> 1996-10-16 16:17:09
ok
```

Show Reports

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

```
10> rb:show(1).
7> rb:show(4).

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

ok
8> rb:show(9).

CRASH REPORT  <0.24.0>                                     1996-10-16 16:16:21
===============================================================================
Crashing process
pid                                                                 <0.24.0>
registered_name                                            release_handler
error_info                              {undef,{release_handler,mbj_func,[]}}
initial_call
{gen,init_it,
[gen_server,
<0.20.0>,
<0.20.0>,
{erlang,register},
release_handler,
release_handler,
[],
[]]}
ancestors                                             [sasl_sup,<0.18.0>]
```

```
messages                                                        []
links                                          [<0.23.0>,<0.20.0>]
dictionary                                                      []
trap_exit                                                    false
status                                                     running
heap_size                                                      610
stack_size                                                     142
reductions                                                      54

ok
```

Search the Reports

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

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

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

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

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

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

CRASH REPORT  <0.24.0>                                  1996-10-16 16:16:21
==============================================================================
Crashing process
pid                                                          <0.24.0>
registered_name                                       release_handler
error_info                          {undef,{release_handler,mbj_func,[]}}
```

```
initial_call
{gen,init_it,
[gen_server,
<0.20.0>,
<0.20.0>,
{erlang,register},
release_handler,
release_handler,
[],
[]]}
ancestors                                          [sasl_sup,<0.18.0>]
messages                                                           []
links                                              [<0.23.0>,<0.20.0>]
dictionary                                                         []
trap_exit                                                       false
status                                                        running
heap_size                                                         610
stack_size                                                        142
reductions                                                         54

Found match in report number 8

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

Child process
errorContext                                                child_terminated
reason                                  {undef,{release_handler,mbj_func,[]}}
pid                                                             <0.24.0>
name                                                     release_handler
start_function                          {release_handler,start_link,[]}
restart_type                                                  permanent
shutdown                                                           2000
child_type                                                       worker

        ok
```

Stop the Server

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

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

Chapter 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.

3.1 Naming of Modules, Applications and Releases

Each module in the system has a version number. An Erlang module should start with

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

and should be stored in a file named as `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.

Also the applications 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. Hence the following is also valid contents for an application resource file:

```
{application, snmp,
  [{vsn, Va},
   {modules,
     [lists,
      ordsets,
      ...
```

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. An `.appup` file contains the following:

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

- `Vsn` is the 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 38]
- `DownToVsn` is a version to which we can downgrade
- `DownToScript` is the script which describes the sequence of downgrade instructions.

In the case of `UpFromScript` 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 `ReleaseFileName.rel`, which describes the release contains the following:

```
{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` needs not to be the same as `ReleaseFileName` 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).
- The application is started by a call to `application:start(AppName, AppType)`, if the `AppType` is `permanent`, `transient` or `temporary`. If `AppType` is `load` the application is loaded but not started, and if it is `none` the application is neither loaded or 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`. It 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 31]). The tedious work of writing the `relup` file is automated and in most cases, the file will be automatically generated from the `.appup` files for the applications in the release.

3.2 Release Tools

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

The following functions are 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 perform 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.

3.3 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. The `priv` directory of an application is found by a call to `code:priv_dir(AppName)`.
- 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 `ReleaseFileName.rel` files, and new release packages are installed here.
- The `bin` directory contains the top level Erlang executable program `erl`.

Applications are not required 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 are introduced as variables in the call to the systools:make_script/2 function.

Disk-less and/or Read-Only Clients

If a complete system consists of some disk-less and/or read-only client nodes, a clients directory should be added to the $ROOTDIR directory. By 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 contains the following:

```
$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. Alternatively, 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.

3.4 Example

Suppose we have a system called “test”, which consists of the three applications: `snmp`, `kernel` and `stdlib`. The `snmp` application is described in the application resource file `snmp.app` 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’s Guide, Chapter 3: Design Principles, section *Applications* for a more detailed description of the contents of 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 SNMP application call-back module for the application complying with the behavior `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.

The application resource file `stdlib.app` for `stdlib` version “6” contains the following:

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

```
]}.
```

> **Note:**
> `stdlib` is a “code only” application and has no call-back module.

Finally, the `kernel.app` file of the `kernel` application version “2” contains the following:

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

We can now in the `test1.rel` file 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("test1")
```

where `test1` refers to the `test1.rel` file.

This command reads the `test1.rel` file and 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.

All required application resource files and all required Erlang files must be located somewhere within the current code path, {`path, Path`}.

If there were no errors, a start script called `test1.script` and a boot file called `test1.boot` are created. The latter is a binary version of the former, and is used when starting the system (e.g. by issuing the command `erl -boot test1`).

Changing an Application

Suppose now that we make a change to `snmp` which results in new versions of the modules `snmp_map` and `snmp_stuff`. This is specified as follows in a new version of `snmp.app`:

```
{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 in the file `test2.rel` as follows:

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

As before we generate the `test2.script` and `test2.boot` file by calling

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

So far we have version “5” and “6” of the “test” release defined in the `test1.rel` and `test2.rel` files, and the generated script and boot files `test1.script`, `test1.boot`, `test2.script`, and `test2.boot`. In our example two versions of the “test” release only differ in the contents of the `snmp` application. In order to be able to update from version “5” to version “6” of the “test” release, we have to provide a specification of the upgrade of the `snmp` application in the form of an application upgrade file `snmp.appup`.

The contents of the `snmp.appup` file is as follows:

```
{"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, []}]}]
}.
```

The `snmp` application is upgraded by changing code for the `snmp_map` and `snmp_stuff` modules. It is downgraded by changing code for the same two modules.

Since only the `snmp` application was changed between version “5” and version “6” of the “test” release, no `.appup` files are needed for the other applications.

In order to finish the specification of the upgrade of the complete “test” release, a release upgrade file, `relup`, has to be created (`relup` is not a file suffix; the complete name of the file is `relup`). The `relup` file is created by evaluating:

```
systools:make_relup("test2", ["test1"], ["test1"]).
```

Here the first argument is name of the `.rel` file we upgrade to or downgrade from. The second and third arguments are lists of `.rel` files, specifying releases to upgrade from, and downgrade to, respectively.

A `relup` file contains low-level code change instructions for the whole release, based on all application `.appup` files.

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 it by evaluating

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

A release package file, named `test2.tar.gz` is generated. The release package file may be installed in a target system by using the release_handler. [page 27]

In this example, the release package file will contain all applications of version “6” of the “test” release, and also a `releases` directory with the following contents:

```
$ROOTDIR/releases/test2.rel
                /6/relup
                  /start.boot
```

where `start.boot` is a copy of the original `test2.boot`.

Chapter 4

Release Handling

4.1 Introduction

A new release is assembled into a *release package*. Such a package is installed in a running system by giving commands to the *release handler*, which is an SASL process. A system has a unique *system version*, which is updated whenever a new release is installed. The system version 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.

Changes may require a node to be brought down. If that is the case and the system consists of several nodes, the release upgrade can be done 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. Can only be done by taking down the node. This kind of change is not supported by the release handler and therefore has to be performed manually. It is not possible to roll back automatically to a previous release, if there is an error.

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

Erlang emulator change. Can only be made by taking down the node. However, the release handler supports this type of change.

4.2 Administering Releases

This section describes how to build and install releases. Also refer to the SASL Reference Manual, `release_handler`, for more details.

The following steps are involved in administering releases:

1. A release package is built by using release building commands in the `systools` module. The package is assembled from application specification files, code files, data files, and a file, which describes how the release is installed in the system.
2. The release package is transferred to the target machine, e.g. by using `ftp`.
3. The release package is unpacked, which makes the system version in the release package available for installation by the `release_handler`, which interprets the *release upgrade script*, containing instructions for updating to the new version. If an installation fails in some way, the entire system is restarted from the old system 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 reinstall an old version, or reboot the system from an old version. There are functions to remove old releases from disk as well.

4.3 File Structure

The file structure used in an OTP system is described in Release Directories [page 19]. 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 `$OTP_ROOT/lib` in the same way as other OTP applications. The release handler takes care of everything, from unpacking a release to the removal of it. The release packages should be stored in the *releases directory* (default `$OTP_ROOT/releases`). This is where `release_handler:unpack_release/1` searches for the packages, and where the release handler stores its files. Each package 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 package, 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` to handle this case:

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

4.4 Release Installation Files

The following files must be present when a release is installed. All file names are relative to the *releases* directory.

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

The location of the *releases* directory is specified with the configuration parameter `releases_dir` (default `$OTP_ROOT/releases`). In a target system, the default location is preferred, but during testing it may be more convenient to let the release handler write its files in a user specified directory, than in the `$OTP_ROOT` directory.

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

`Vsn` is the system version string.

ReleaseFileName.rel

The `ReleaseFileName.rel` file contains the name of the system, version of the release, the version of `erts` (the Erlang runtime system) and the applications, which are parts of the release. The file must contain the following 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 `AType` are atoms. `ReleaseFileName` is a string given in the call to `release_handler:unpack_release(ReleaseFileName)`. `Name` is the name of the system (the same as found in the boot file). This file is further described in Release Structure [page 17].

relup

The `relup` file contains instructions on how to install the new version in the system. It 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. Finally, 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 the new version, and one tuple {ToVsn, Descr, RuScript} for each old version to which the new version can be downgraded.

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

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

sys.config

The sys.config is the system configuration file.

4.5 Release Handling Principles

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

Erlang Code

The code change feature in Erlang is made possible because Erlang allows two versions of a module to be present in the system: the *current* version and the *old* version. There is always a current version of a loaded module, but an old version of a module only exists if the module has been replaced in run-time by loading a new version. When a new version is loaded, the previously current version becomes the old version, and the new version becomes the current version. However, if there are both a current and old version of a module, a new version cannot be loaded, unless the old version is first explicitly purged.

A global function call is a call where a qualified module name is used, i.e. the call is of the form `M:F(A)` (or `apply(M, F, A)`). A global call causes `M:F` to be dynamically linked into the run-time code, which means that `M:F(A)` will be evaluated using the latest available version of the module, i.e. the current version.

A local function call is a call without a qualified module name, i.e. the call is of the form `F(A)`. The reference to `F` is resolved at compile time (irrespective of whether `F` is exported or not). By the very nature of `F(A)` being a local function call, `F` can only be called by a function that is defined in the very same module as that where `F` is defined. Hence a local function call is always evaluated in the same version of a module as that of the caller.

A *fun* is a function without a name. Like ordinary functions (i.e. functions which have names) its implementation is always bound to some module, and therefore funs are affected by code change as well. A reference to a fun is always indirect, as is the case for a global function call, where the reference is `M:F` (through an export table entry for the module), but the reference is not necessarily global. In fact, if a fun is called in the same module where it is defined, its reference will be resolved in the same way as a local function call is resolved. If a fun is called from a different module, its reference will be resolved as if the call was a global call, but with the additional requirement that the reference also match the particular implementation of the module where the fun was defined.

For each process there is a current function, i.e. the function that the process is currently evaluating. That function resides in some module. Hence a process has always a reference to at least one module. It may of course have references to other modules as well, because of nested, not yet finished calls.

Before a new version of a module can be loaded, the current version must be made old. If there is no old version, the new version is merely loaded, making the previously current version to the old version, and the new version becomes current. All processes that execute the version, which became old, will continue to do so, until they have no unfinished calls within the old version.

If there is an old version, it must first be purged to make room for the current version to become old. However, an old version should not be purged if there are processes that have references to it. Such processes must either be terminated, or the loading of the new version must be postponed until they have terminated by themselves or no longer have references to the old version. There are options for controlling this in release upgrade scripts.

To prevent processes from making calls to other processes during the release installation, they may be *suspended*. All processes implemented with the standard behaviors, or with `sys`, can be suspended. When suspended a 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 possibly to transform its internal state. After the code change a process is *resumed*, i.e. it returns to its main loop.

We highlight here three different types of modules.

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

Process module. A module, which contains a process loop, i.e. some process has constant reference to the module. `init` is an example of a process module.

Call-back module. A special case of a functional module which serves as a call-back module for a generic behavior such as `gen_server`. `file` is an example of a call-back module. A call to a call-back module is always a global call (i.e. it refers to the latest version of the module). This has some impacts upon how updates must be handled.

Modules of the above types are handled differently when changing code.

Functional Module

If the API of a new version of a functional module is backward compatible, as may be the case of a bug fix or new functionality, we 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 complicated situation arises if the API of a functional module is changed so it is not longer backwards compatible. We must then make sure that no processes, directly or indirectly, try to call functions that have changed. We do this by writing new versions of all modules that use the API. Then, when performing the code change, all potential caller processes are suspended, new versions of the modules that uses the API are loaded, the new version of the functional module is loaded, and finally all suspended processes are resumed.

There are two alternatives available to manage this type of change:

1. Find all calls to the module, change them, and write dependencies in your release upgrade script. This may be manageable, if a function that has been incompatibly changed is called from only a few other functions.
2. Avoid this type of change. This is the only reasonable solution, if an incompatible function is called from many other modules. Instead a completely new function should be introduced, and the original function should be kept for backward compatibility. In the next release, when all other modules are changed as well, the original function can be deleted.

Process Module

A process module should never contain global calls to itself (except for code that makes explicit code change). Therefore, a new version of a process module is merely loaded and all processes which are executing the 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:map/2` in a process module, there will be global calls to the module. Therefore, we cannot merely load the module because a process might, still running the old version of the module, make a call to the new version, which might be incompatible.

The only safe way to change code for a process module, is to have its implementation to understand system messages, and to change code by first suspending all processes that run the module, then order them to change code, and finally resume them.

Call-back Module

As long as the type of the internal state of a call-back module has not changed, we can just simply load the new version of the module without suspending and resuming the processes involved in the code change. This case is similar to the case of a functional module.

If the type of the internal state has 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 case of a process module.

Dependencies Between 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 that 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 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 the processes are 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 manage, it may be better to entirely shut down and restart all affected processes. This reduces the problem of introducing new code and removes the need to do a synchronized change.

Finding Processes

For each application the `.appup` file specifies how the application is upgraded. The file contains specifications of which modules to change, and how to change them. The `relup` file is an assembly of all the `.appup` files.

For each application the release handler searches for all processes that have to perform a code change. It traverses the application supervision tree to find all child specifications of every supervisor in the tree. Each child specification lists all modules of the application that the child uses.

Hence it is by combining the list of modules to change with all children of supervisors that the release handler finds all processes that are subject to code change.

Port Programs

A port program runs as an external program in the operating system. The simplest way to do code change for a port program is to terminate it, and then start a new version of it.

If that is not adequate, code change may be performed by sending the port program a message telling it to return any data that must survive the termination. Then the program is terminated, and the new version is started and the survived data is to the new version of the port program.

Changing code for port programs is very application dependent. There is no special support for it in SASL.

Application Specification and Configuration Parameters

In each release, each application specification (i.e. the contents of the `.app` file of the application) is known to the release handler. Before any code change is performed for an application, the new environment variables are are made available for the application, i.e. those parameters specified by the `env` tag in the application specification. When the new version of an application is running it will be informed of any changed, new or removed environment variables (see application(Module) in the KERNEL Reference Manual). This means that old processes may read new variables before they are informed of the new release. We advise against the immediate removal of the old variables. 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 it is known that no processes will read the old variables.

Mnesia Data or Schema Changes

Changing data or schemas in Mnesia is similar to changing code for functional modules. Many processes may read or write in the same table at the same time. If we change a table definition, we must make sure that all code which uses the table is changed at the same time.

One way of doing it is to let one process be responsible for one or several tables. This process creates the tables and changes the table definitions or table data. In this way a set of tables is connected with a module (process module or call-back module). When the process performs a code change, the tables are changed as well.

Upgrade vs. Downgrade

When a new release is installed, the system is *upgraded* to the new release. The release handler reads the `relup` file of the new release, and finds the upgrade script that corresponds to an upgrade from the current version to the new version of the system.

When an old release is reinstalled, the release handler reads the `relup` in the current release, and finds the *downgrade* script that corresponds to an downgrade from the current version to the old version of the system.

Usually a `relup` file for a new release contains one upgrade script and one downgrade script for each old version. If a soft downgrade is not wanted (an alternative is to reboot the system from the old release) the downgrade script is left out.

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 switch to new code.
4. Tell the processes to change the internal state. This usually involves calling, *in the new module*, a `code_change` function that is responsible for state updates, e.g. transforming the state from the old format to the new.
5. Resume the processes.

The code change step is always performed when new code has been loaded and all processes are running the new code. 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 transform the new state to the old version: the new format is unknown to the old code. Therefore, it is the responsibility of 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, *in the current module*, a `code_change` function that is responsible for state reversals, i.e. 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 a process change its 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 difference between the two cases is still in the order of switching code and changing state.

For a call-back module it is not actually necessary to tell the processes to switch code, since all calls to the call-back module are global calls. The difference between upgrade and downgrade is still in the order of loading code and performing state change.

The difference between how process modules and a call-back modules are handled in the downgrade 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 a process module does not spontaneously switch 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. A call-back module is always dynamic, and a process module static. A functional module is always dynamic.

4.6 Release Handling Instructions

This section describes the release upgrade and downgrade scripts. A script is a list of instructions which are interpreted by the release handler when an upgrade or downgrade is made.

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 by the `systools:make_relup/3` command, because the release handler understands only low-level instructions.

Scripts have to be placed in the `.appup` file for each application. `systools:make_relup/3` assembles the scripts in all `.appup` files to form a `relup` file containing low-level instructions.

High-level Instructions

The high-level instructions are:

- `{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.

  The instruction is used to update a process module or a call-back module. All processes that run the code of `Module` are suspended, and if the change is `advanced` they have to transform their states into the new states. Then the processes are resumed. If `Module` is dependent on other modules, the release handler will suspend processes in `Module` before suspending processes in the `[Mod]` 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, which is used in the argument list of the `code_change` function in `Module` (call-back module); otherwise it becomes part of a code change message (process module).
  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 to 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 36] above. It needs to be specified only in the case of soft downgrades. Its value defaults to `dynamic`. Note; 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 upgrade (or downgrade).

Usually there are no such processes. If the value is `soft_purge` and such processes are found, the release will not be installed and the `install_release/1` function returns {`error`, {`old_processes`, `Module`}}. If the value 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 the value is `soft_purge`, the release handler will purge the old code when no remaining processes execute the code. If the value is `brutal_purge`, the code is purged when the release is made permanent. All processes, which still are running old code are killed.

The `update` instruction can 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 perform 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.

  The instruction is used to update a functional module or a call-back module. It only loads the module. A call-back 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`} The instruction adds a new module to the system. It loads the module.
- {`remove_application`, `Appl`} Removes an application. It calls `application:stop` and `application:unload` for the application.
- {`add_application`, `Appl`} Adds a new application. It calls `application:load` and `application:start` for the application.
- {`restart_application`, `Appl`} Restarts an existing application. The current version of the application is stopped and removed, and the new version of the application is loaded and started. The instruction is useful when the simplest way to change code for an application is to stop and restart the whole application.

Low-level instructions

The low-level instructions are:

- {`load_object_code`, {`Lib`, `LibVsn`, [`Module`]}} Reads each `Module` from the library `Lib-LibVsn` as a binary. It does not install the code, it just reads the files. The instruction 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 instruction has been executed, the code server is updated with the new version of `Lib`. Calls to `code:priv_dir(Lib)` which are made after this instruction return the new `priv` dir.

  `Lib` is typically the application name.
- `point_of_no_return` If a crash occurs after this instruction, the system cannot recover and is restarted from the old version. The instruction 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 this instruction occurs, the Module object code must have been loaded with with the load_object_code instruction. This instruction makes code out of the binary. PrePurge = soft_purge | brutal_purge, and PostPurge = soft_purge | brutal_purge.
- {remove, {Module, PrePurge, PostPurge}} Makes the current version of Module old. When it has been executed, there is no current version in the system. PrePurge = soft_purge | brutal_purge, and PostPurge = soft_purge | brutal_purge.
- {purge, [Module]} Kills all processes that run the old versions of the modules in [Module] and deletes all old versions.
- {suspend, [Module | {Module, Timeout}]} Tries to suspend all processes that execute Module. If a process does not respond, it is ignored. This may cause the process to die, either because it crashes when it spontaneously switches to new code, or as a result of a purge operation. If no Timeout is specified (or 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 that 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). In case of 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 it obtains its value from 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]} Resumes all previously suspended processes which execute in any of the modules in the list [Module].
- {stop, [Module]} Stops all processes which are in any of the modules in the list [Module]. The instruction 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]} Starts all previously stopped processes which are in any member of [Module]. The processes will regain their positions 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. The instruction synchronizes the release installation with other nodes. Each node in the list of nodes must evaluate this command, with the same Id. The local node waits for all other nodes to evaluate the instruction before execution continues. In case a node goes down, it is considered to be an unrecoverable error, and the local node is restarted from the old release. There is no time-out for this instruction, which implies that it 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 to crash.
- {apply, {M, F, A}} Applies the function to the arguments. If the instruction appears before the point_of_no_return instruction, a failure of the application M:F(A) is caught, causing release_handler:install_release/1 to return {error, {'EXIT', Reason}}. If {error, Error} is thrown or returned by M:F, install_release/1 returns {error, Error}.

If the instruction appears after the `point_of_no_return` instruction, and if the application `M:F(A)` fails, the system is restarted.

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

4.7 Release Handling Examples

This section includes several examples that show how different types of upgrades are handled. In call-back modules having the `gen_server` behavior, all call-back functions have been provided for reasons of clarity.

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 is about a pure functional module, i.e. a module the functions of which have no side effects. The original version of the module `lists2` has the following contents:

```
-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

Here we have a functional module `bar` that uses the module `lists2` of the previous example. The original version is only dependent on the original version of `lists2`.

```
-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 `bar` uses the new functionality of `lists2` in order to simplify the implementation of the useful function `complicated_sum/1`. It does not change its 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` to make the release handler to load `lists2` before it loads `bar`.

The low-level 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 modify the return value of `lists2:assoc/2` from `{ok, Val}` to `{Key, Val}`. In order to do an upgrade, we would have to find all modules that call `lists2:assoc/2` directly or indirectly, and specify that these modules are dependent on `lists2`. In practice this might an unweildy task, if if many other modules are using the `lists2` module, and the only reasonable way to perform an upgrade which restarts the whole system.

If we insist on doing a soft upgrade, the modification should be made backward compatible by introducing an new function (`assoc2/2`, say) that has the new return value, and not make any changes to the original function at all.

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 contents of the original module are as follows:

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

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

handle_cast(_Request, State) ->
    {noreply, State}.
```

```
handle_info(_Info, State) ->
    {noreply, State}.

terminate(_Reason, _State) ->
    ok.

code_change(_OldVsn, State, _Extra) ->
    {ok, 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, handle_cast/2, handle_info/2,
         terminate/2, 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}.

handle_cast(_Request, State) ->
    {noreply, State}.

handle_info(_Info, State) ->
    {noreply, State}.

terminate(_Reason, _State) ->
    ok.

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

The release upgrade instructions are 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 be as follows:

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

-export([get_data/0, get_time/0]).
-export([init/1, handle_call/3, handle_cast/2, handle_info/2,
         terminate/2, 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}.

handle_cast(_Request, State) ->
    {noreply, State}.

handle_info(_Info, State) ->
    {noreply, State}.

terminate(_Reason, _State) ->
    ok.

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 low-level instructions are:

```
[{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 assumes that we have gen_server process that uses the in gs1 as defined in the previous example.

The contents of the original module are as follows:

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

-export([is_operation_ok/1]).
-export([init/1, handle_call/3, handle_cast/2, handle_info/2,
         terminate/2, code_change/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}.

handle_cast(_Request, State) ->
    {noreply, State}.

handle_info(_Info, State) ->
    {noreply, State}.

terminate(_Reason, _State) ->
    ok.

code_change(_OldVsn, State, _Extra) ->
    {ok, State}.
```

The new version does not have to transform the internal state, hence the code_change/3 function is not really needed (it will not be called since the upgrade of gs2 is soft).

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

-export([is_operation_ok/1]).
-export([init/1, handle_call/3, handle_cast/2, handle_info/2,
         terminate/2, code_change/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 = do_things(lists2:assoc(Op, Data), Time),
    {reply, Reply, State}.

handle_cast(_Request, State) ->
    {noreply, State}.

handle_info(_Info, State) ->
    {noreply, State}.

terminate(_Reason, _State) ->
    ok.

code_change(_OldVsn, State, _Extra) ->
    {ok, State}.

do_things({ok, Val}, Time) ->
    Val;
do_things(false, Time) ->
    {false, Time}.
```

The release upgrade instructions are:

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

The corresponding low-level 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 processes of the types `gen_event`, `gen_fsm`, and processes implemented by using `proc_lib` and `sys`, are handled in exactly the same way as processes of type `gen_server` are handled. Examples follow.

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 contents of the original module is as follows:

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

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

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) ->
    {ok, [LastEvent], LastEvent}.

handle_info(Info, LastEvent) ->
    {ok, LastEvent}.

terminate(Arg, LastEvent) ->
    ok.

code_change(_OldVsn, LastEvent, _Extra) ->
    {ok, 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, handle_info/2,
         terminate/2, 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.

handle_info(Info, Events) ->
    {ok, Events}.

terminate(Arg, Events) ->
    ok.

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 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

Processes implemented with sys and proc_lib are changed in the same way as processes that are implemented according to the gen_server behavior (which should not come as surprise, since gen_server et al. are implemented on top of sys and proc_lib). However, the code change function is defined differently. The original is 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(?MODULE, init, [self()]),
```

```
    {ok, Pid}.

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

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

loop(State, Parent) ->
    receive
        {system, From, Request} ->
            sys:handle_system_msg(Request, From, Parent, ?MODULE, [], State);
        {'EXIT', Parent, Reason} ->
            cleanup(State),
            exit(Reason);
        {From, get_data} ->
            From ! {sp_server, 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 is 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(?MODULE, init, [self()]),
    {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) ->
    register(sp_server, self()),
    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,
                                  ?MODULE, [], 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 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 is 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 is 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 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]}}]
```

High-level update instruction for a supervisor is mapped to a low-level advanced code change instruction. In the code_change function of the supervisor, the new child specification is installed, but no children are explicitly terminated or started. Therefore, children must be terminated, deleted and started by using the apply instruction.

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, individual processes cannot simply be killed, since their supervisors will restart them again. Instead supervisors must first be ordered to stop their children before now code can be loaded. Then supervisors are ordered to restart their children. All this is done by issuing the stop and start instructions.

The following 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 want to restart all processes except for a. The upgrade instructions are 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 stop d, this is done by the supervisor c.

A whole application cannot be stopped and started with the stop and start instructions. The instruction restart_application has to be used instead.

New Application

The examples shown so far have dealt with changing an existing application. In order to introduce a completely new application we just have to have an add_application instruction, but we also have to make sure that the boot file of the new release contains enough in order to start it. 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 corresponding low-level instructions are as follows (note that the application specification is used as argument to application:start_application/1):

```
[{load_object_code, {new_appl, "1.0", [new_mod]}},
 point_of_no_return,
 {load, {new_mod, soft_purge, soft_purge}},
 {apply, {application, start,
           [{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 way as new applications are introduced. This example assumes that we want to remove the `new_appl` application:

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

The corresponding low_level 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

In this example we have a port controller process, where we must take care of the termination and restart of the port program ourselves. Also, we may prepare for the possibility of changing the Erlang code of the port controller only. The `gen_server` behavior is used to implement the port controller. The contents of the original module is 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]}]
```

Here we used low-level instructions only. In this example we also make use of the `Extra` argument of the `code_change/3` function.

Suppose now that we wish to change only the Erlang code. The new version of `portc` is 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 high-level instruction is:

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

The corresponding low-level instructions are:

```
[{load_object_code, {portc, 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 66] Clears the specified alarms
- `get_alarms() -> [alarm()]`
  [page 66] Gets all active alarms
- `set_alarm(alarm())`
  [page 66]

overload

The following functions are exported:

- `request() -> accept | reject`
  [page 69] Requests to proceed with current job
- `get_overload_info() -> OverloadInfo`
  [page 69] Returns current overload information data

rb

The following functions are exported:

- `grep(RegExp)`
  [page 70] Searches the reports for a regular expression
- `h()`
  [page 70] Prints help information
- `help()`
  [page 70] Prints help information
- `list()`
  [page 70] Lists all reports
- `list(Type)`
  [page 70] Lists all reports
- `rescan()`
  [page 70] Rescans the report directory
- `rescan(Options)`
  [page 70] Rescans the report directory
- `show()`
  [page 70] Shows reports
- `show(Report)`
  [page 70] Shows reports
- `start()`
  [page 71] Starts the RB server
- `start(Options)`
  [page 71] Starts the RB server
- `start_log(FileName)`
  [page 71] Redirects all output to `FileName`
- `stop()`
  [page 71] Stops the RB server
- `stop_log()`
  [page 71] Stops logging to file

release_handler

The following functions are exported:

- `check_install_release(Vsn) -> {ok, FromVsn, Descr} | {error, Reason}`
  [page 73] Checks installation of the release in the system
- `create_RELEASES(Root, RelDir, RelFile, LibDirs) -> ok | {error, Reason}`
  [page 74] Creates an initial RELEASES file
- `install_file(Vsn, FileName) -> ok | {error, Reason}`
  [page 74] Installs a release file in the release handler
- `install_release(Vsn) -> {ok, FromVsn, Descr} | {error, Reason}`
  [page 74] Installs the release in the system
- `install_release(Vsn, Opt) -> {ok, FromVsn, Descr} | {error, Reason}`
  [page 74] Installs the release in the system
- `make_permanent(Vsn) -> ok | {error, Reason}`
  [page 75] Makes the specified release to be used at system start-up
- `remove_release(Vsn) -> ok | {error, Reason}`
  [page 75] Deletes all files unique for this release
- `reboot_old_release(Vsn) -> ok | {error, Reason}`
  [page 75] Reboots the system from an old release
- `set_removed(Vsn) -> ok | {error, Reason}`
  [page 75] Marks a release as removed
- `set_unpacked(RelFile, LibDirs) -> {ok, Vsn} | {error, Reason}`
  [page 75] Marks a release as unpacked
- `unpack_release(ReleaseName) -> {ok, Vsn} | {error, Reason}`
  [page 76] Unpacks and extracts files from the release package
- `which_releases() -> [{Name, Vsn, [Lib], Status}]`
  [page 76] Returns all known releases

systools

The following functions are exported:

- `behaviour_info() -> [Behaviour]`
  [page 77] Lists the system defined behaviours
- `behaviour_info(Behaviour) -> [Function]`
  [page 77] Lists the functions that a behaviour uses
- `make_script(ReleaseName) -> MakeRet`
  [page 77] Creates a boot script from a release file
- `make_script(ReleaseName,Opts) -> MakeRet`
  [page 77] Creates a boot script from a release file
- `make_relup(ReleaseName,UpNameList,DownNameList) -> RelRet`
  [page 80] Gathers release upgrade scripts for a release
- `make_relup(ReleaseName,UpNameList,DownNameList,Opts) -> RelRet`
  [page 80] Gathers release upgrade scripts for a release

- `make_tar(ReleaseName) -> TarRet`
  [page 81] Creates a release package.
- `make_tar(ReleaseName,Opts) -> TarRet`
  [page 81] Creates a release package.
- `script2boot(File) -> ok | error`
  [page 82] 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 not set.

`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 not set.

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

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

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 it is to be used in a system, it must 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, which means that Mnesia must run if the implementation of the MIB should be performed.

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. It defines a place to which alarms can be sent. One simple event handler is installed in the alarm handler at start-up, but users are encouraged to write and install their own handlers.

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

There are functions to set and clear alarms. The format of alarms are defined by the user. For example, an event handler for SNMP could be defined, together with an alarm MIB.

The alarm handler is part of the SASL application.

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

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

{clear_alarm, AlarmId} This event is generated by
    alarm_handler:clear_alarm(AlarmId).

The default simple handler is called alarm_handler and it may be exchanged by calling gen_event:swap_handler/3 as gen_event:swap_handler(alarm_handler, {alarm_handler, swap}, {NewHandler, Args}). NewHandler:init({Args, {alarm_handler, Alarms}}) is called. Refer to gen_event(3) for further details.

Exports

clear_alarm(AlarmId) -> void()

Types:

- AlarmId = term()

Clears all alarms with id AlarmId.

get_alarms() -> [alarm()]

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

set_alarm(alarm())

Types:

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

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

See Also

error_logger(3), gen_event(3)

overload (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 the STDLIB reference manual for a definition of valid regular expressions. They are essentially the same as the UNIX command `egrep`.

`h()`
`help()`

Prints the on-line help information.

`list()`
`list(Type)`

Types:

- Type = type()
- type() = crash_report | supervisor_report | error | progress

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

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

`rescan()`
`rescan(Options)`

Types:

- Options = [opt()]

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

`show()`
`show(Report)`

Types:

- Report = int() | type()

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

`start()`

`start(Options)`

Types:

- Options = [opt()]
- opt() = {start_log, FileName} | {max, MaxNoOfReports} | {report_dir, DirString} | {type, ReportType}
- 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 will 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 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} | {code_change_timeout, Timeout} | {suspend_timeout, Timeout}]
- Error_action = restart | reboot
- Descr = term()
- Timeout = default | infinity | int() > 0

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.

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

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

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 this reference manual or in the Erlang Development Environment Reference Manual.

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, Path`} 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 also 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 `.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 | restart_emulator]
- 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.

If the restart_emulator option is supplied, the low-level instruction restart_new_emulator is appended to the relup scripts. This ensures that a complete reboot of the system is done when the system is upgraded or downgraded.

```
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 must 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 .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 `UpFromScript` 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 a *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 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 applications to start.

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

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

FILE SYNTAX

The boot script is stored in a file with the extension `.script`

The file has the following syntax:

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

- `Name = string()` defines the name of the system.
- `Vsn = string()` defines the version of the system.
- {`progress, Term`} sets the "progress" of the initialization program. The function `init:get_status()` returns the current value of the progress, which is {`InternalStatus,Term`}.
- {`path, [Dir]`} where `Dir` is a string. This argument sets the load path of the system to `[Dir]`. The load path used to load modules is obtained from the initial load path, which is given in the script file, together with any path flags which were supplied in the command line arguments. The command line arguments modify the path as follows:
  - `-pa Dir1 Dir2 ... DirN` adds the directories `Dir1, Dir2, ..., DirN` to the front of the initial load path.

- -pz Dir1 Dir2 ... DirN adds the directories Dir1, Dir2, ..., DirN to the end of the initial load path.
- -path Dir1 Dir2 ... DirN defines a set of directories Dir1, Dir2, ..., DirN which replaces the search path given in the script file. Directory names in the path are interpreted as follows:
  - Directory names starting with / are assumed to be absolute path names.
  - Directory names not starting with / are assumed to be relative the current working directory.
  - The special $ROOT variable can only be used in the script, not as a command line argument. The given directory is relative the Erlang installation directory.

- {primLoad, [Mod]} loads the modules [Mod] from the directories specified in Path. The script interpreter fetches the appropriate module by calling the function erl_prim_loader:get_file(Mod). A fatal error which terminates the system will occur if the module cannot be located.
- {kernel_load_completed} indicates that all modules which *must* be loaded *before* any processes are started are loaded. In interactive mode, all {primLoad,[Mod]} commands interpreted after this command are ignored, and these modules are loaded on demand. In embedded mode, kernel_load_completed is ignored, and all modules are loaded during system start.
- {kernelProcess, Name, {Mod, Func, Args}} starts a "kernel process". The kernel process Name is started by evaluating apply(Mod, Func, Args) which is expected to return {ok, Pid} or ignore. The init process monitors the behaviour of Pid and terminates the system if Pid dies. Kernel processes are key components of the runtime system. Users do not normally add new kernel processes.
- {apply, {Mod, Func, Args}}. The init process simply evaluates apply(Mod, Func, Args). The system terminates if this results in an error. The boot procedure hangs if this function never returns.

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

SEE ALSO

systools(3)

List of Tables

Index

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