

OSE

Copyright © 2014-2017 Ericsson AB. All Rights Reserved.
OSE 1.1
December 1, 2017

Copyright © 2014-2017 Ericsson AB. All Rights Reserved.
Licensed under the Apache License, Version 2.0 (the "License"); you may not use this file except in compliance with the License. You may obtain a copy of the License at http://www.apache.org/licenses/LICENSE-2.0 Unless required by applicable law or agreed to in writing, software distributed under the License is distributed on an "AS IS" BASIS, WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied. See the License for the specific language governing permissions and limitations under the License. Ericsson AB. All Rights Reserved..

December 1, 2017

1 OSE User's Guide

OSE.

1.1 Introduction

1.1.1 Features

1.1.2 Starting Erlang/OTP

Starting Erlang/OTP on OSE is not as simple as on Unix/Windows (yet). First of all you have to explicitly use the beam (or beam.smp) executables found in erts-X.Y.Z/bin as the load module that you run. This in turn means that you have to supply the raw beam arguments to the emulator when starting. Fortunately `erl` on Unix/Windows has a undocumented flag called `-emu_args_exit` that can be used to figure out what the arguments to beam look like. For example:

```
# erl +Mut false +A 10 +S 4:4 +Muycs256 +P 2096 +Q 2096 -emu_args_exit
-Mut
false
-A
10
-S
4:4
-Muycs256
-P
2096
-Q
2096
--
-root
/usr/local/lib/erlang
-progname
erl
--
-home
/home/erlang
--
```

The arguments are printed on separate lines to make it possible to know what has to be quoted with ". Each line is one quotable unit. So taking the arguments above you can supply them to pm_create or just execute directly on the command line. For example:

```
rtose@acp3400> pm_install erlang /mst/erlang/erts-6.0/bin/beam.smp
rtose@acp3400> pm_create -c ARGV="-Mut false -A 10 -S 4:4 -Muycs256 -P 2096 -Q 2099 -- -root /mst/erlang -pro
pid: 0x110059
rtose@acp3400> pm_start 0x110059
```

Also note that since we are running erl to figure out the arguments on a separate machine the paths have to be updated. In the example above `/usr/local/lib/erlang` was replaced by `/mst/erlang/`. The goal is to in future releases not have to do the special argument handling but for now (OTP 17.0) you have to do it.

> **Note:**
> Because of a limitation in the way the OSE handles stdio when starting load modules using pm_install/create the Erlang shell only reads every other command from stdin. However if you start Erlang using run_erl you do not have this problem. So it is highly recommended that you start Erlang using run_erl.

1.1.3 run_erl and to_erl

In OSE run_erl and to_erl are combined into a single load module called run_erl_lm. Installing and starting the load module will add two new shell commands called run_erl and to_erl. They work in exactly the same way as the unix variants of run_erl and to_erl, except that the read and write pipes have to be placed under the /pipe vm. One additional option also exists to run_erl on ose:

`-block Name`
: The name of the install handle and block that will be created/used by installing and exectuting the first part of the command. If nothing if given the basename of the load module will be used for this value. Example:

```
pm_install erlang /path/to/erlang/vm/beam.smp
run_erl -daemon -block erlang /pipe/ /mst/erlang_logs/ "beam.smp -A 1 -- -root /mst/erlang -- -home /mst --
```

The same argument munching as when starting Erlang/OTP without run_erl has to be done. If `-daemon` is given then all error printouts are sent to the ramlog. See also *run_erl* for more details.

Below is an example of how to get started with `run_erl_lm`.

```
rtose@acp3400> pm_install run_erl_lm /mst/erlang/erts-6.0/bin/run_erl_lm
rtose@acp3400> pm_create run_erl_lm
pid: 0x1c005d
rtose@acp3400> pm_start 0x1c005d
rtose@acp3400> mkdir /mst/erlang_log
rtose@acp3400> run_erl -daemon /pipe/ /mst/erlang_log/ "/mst/erlang/erts-6.0/bin/beam.smp -A 1 -- -root /mst/er
rtose@acp3400> to_erl
Attaching to /pipe/erlang.pipe.1 (^C to exit)
os:type().
{ose,release}
2>
'to_erl' terminated.
```

Note that Ctrl-C is used instead of Ctrl-D to exit the to_erl shell.

1.1.4 epmd

In OSE epmd will not be started automatically so if you want to use Erlang distribution you have to manually start epmd.

1.1.5 VM Process Priorities

It is possible to set the priorities you want for the OSE processes that thr emulator creates in the lmconf. An example of how to do it can be found in the default lmconf file in $ERL_TOP/erts/emulator/sys/ose/beam.lmconf.

1.2 Interacting with Enea OSE

1.2.1 Introduction

The main way which programs on Enea OSE interact is through the usage of message passing, much the same way as Erlang processes communicate. There are two ways in which an Erlang programmer can interact with the signals sent from other Enea OSE processes; either through the provided `ose` module, or by writing a custom linked-in driver. This User's Guide describes and provides examples for both approaches.

1.2.2 Signals in Erlang

Erlang/OTP on OSE provides a erlang module called *ose* that can be used to interact with other OSE processes using message passing. The api in the module is very similar to the native OSE api, so for details of how the functions work please refer to the official OSE documenation. Below is an example usage of the API.

```
1> P1 = ose:open("p1").
#Port>0.344>
2> ose:hunt(P1,"p2").
{#Port>0.344>,1}
3> P2 = ose:open("p2").
#Port>0.355>
4> flush().
Shell got {mailbox_up,#Port>0.344>,{#Port>0.344>,1},852189}
ok
5> ose:listen(P1,[1234]).
ok
6> ose:send(P2,ose:get_id(P1),1234,>>"hello">>).
ok
7> flush().
Shell got {message,#Port>0.344>,{852189,1245316,1234,>>"hello">>}}
ok
```

1.2.3 Signals in a Linked-in driver

Writing Linked-in drivers for OSE is very similar to how it is done for Unix/Windows. It is only the way in which the driver subscribes and consumed external events that is different. In Unix (and Windows) file descriptiors (and Event Objects) are used to select on. On OSE we use signals to deliver the same functionality. There are two large differences between a signal and an fd.

In OSE it is not possible for a signal number to be a unique identifier for a resource in the same way as an fd is. For example; let's say we implement a driver that does an asynchronous hunt that uses signal number 1234 as the hunt_sig. If we want to be able to have multiple hunt ports running at the same time we have to have someway of routing the signal to the correct port. This is achieved by supplying a secondary id that can be retrieved through the meta-data or payload of the signal, e.g:

```
ErlDrvEvent event = erl_drv_ose_event_alloc(1234,port,resolver);
```

The event you get back from *erl_drv_ose_event_alloc* can then be used by *driver_select* to subscribe to signals. The first argument is just the signal number that we are interested in. The second is the id that we choose to use, in this case the port id that we got in the *start* callback is used. The third argument is a function pointer to a function that can be used to figure out the id from a given signal. The fourth argument can point to any additional data you might want to associate with the event. There is a complete. You can examine the data contained in the event with *erl_drv_ose_event_fetch* , eg:

```
erl_drv_ose_event_fetch(event, &signal, &port, (void **)&extra);
```

example of what this could look like in *the next section*.

> **Note:**
> It is very important to issue the driver_select call before any of the signals you are interested in are sent. If driver_select is called after the signal is sent, there is a high probability that it will be lost.

The other difference from unix is that in OSE the payload of the event (i.e. the signal data) is already received when the ready_output/input callbacks are called. This means that you access the data of a signal by calling *erl_drv_ose_get_signal*. Additionally multiple signals might be associated with the event, so you should call *erl_drv_ose_get_signal* until `NULL` is returned.

1.2.4 Example Linked-in driver

```
#include "erl_driver.h"
#include "ose.h"

struct huntsig {
  SIGSELECT signo;
  ErlDrvPort port;
};

union SIGNAL {
  SIGSELECT signo;
  struct huntsig;
}

/* Here we have to get the id from the signal. In this case we use the
   port id since we have control over the data structure of the signal.
   It is however possible to use anything in here. The only restriction
   is that the same id has to be used for all signals of the same number.*/
ErlDrvOseEventId resolver(union SIGNAL *sig) {
  return (ErlDrvOseEventId)sig->huntsig.port;
}

static int drv_init(void) { return 0; };

static ErlDrvData drv_start(ErlDrvPort port, char *command) {
  return (ErlDrvData)port;
}

static ErlDrvSSizeT control(ErlDrvData driver_data, unsigned int cmd,
                            char *buf, ErlDrvSizeT len,
                            char **rbuf, ErlDrvSizeT rlen) {
  ErlDrvPort port = (ErlDrvPort)driver_data;

  /* An example of extra data to associate with the event */
  char *extra_data = driver_alloc(80);
  snprintf("extra_data, "Event, sig_no: 1234, and port: %d", port);

  /* Create a new event to select on */
  ErlDrvOseEvent evt = erl_drv_ose_event_alloc(1234,port,resolver, extra_data);

  /* Make sure to do the select call _BEFORE_ the signal arrives.
     The signal might get lost if the hunt call is done before the
     select. */
  driver_select(port,evt,ERL_DRV_READ|ERL_DRV_USE,1);
```

```
  union SIGNAL *sig = alloc(sizeof(union SIGNAL),1234);
  sig->huntsig.port = port;
  hunt("testprocess",0,NULL,&sig);
  return 0;
}

static void ready_input(ErlDrvData driver_data, ErlDrvEvent evt) {
  char *extra_data;
  /* Get the first signal payload from the event */
  union SIGNAL *sig = erl_drv_ose_get_signal(evt);
  ErlDrvPort port = (ErlDrvPort)driver_data;
  while (sig != NULL) {
    if (sig->signo == 1234) {
      /* Print out the string we added as the extra parameter */
      erl_drv_ose_event_fetch(evt, NULL, NULL, (void **)&extra_data);
      printf("We've received: %s\n", extra_data);

      /* If it is our signal we send a message with the sender of the signal
         to the controlling erlang process */
      ErlDrvTermData reply[] = { ERL_DRV_UINT, (ErlDrvUInt)sender(&sig) };
      erl_drv_send_term(port,reply,sizeof(reply) / sizeof(reply[0]));
    }

    /* Cleanup the signal and deselect on the event.
       Note that the event itself has to be free'd in the stop_select
       callback. */
    free_buf(&sig);
    driver_select(port,evt,ERL_DRV_READ|ERL_DRV_USE,0);

    /* There could be more than one signal waiting in this event, so
       we have to loop until sig == NULL */
    sig = erl_drv_ose_get_signal(evt);
  }
}

static void stop_select(ErlDrvEvent event, void *reserved)
{
  /* Free the extra_data */
  erl_drv_ose_event_fetch(evt, NULL, NULL, (void **)&extra_data);
  driver_free(extra_data);

  /* Free the event itself */
  erl_drv_ose_event_free(event);
}

/**
 * Setup the driver entry for the Erlang runtime
 **/
ErlDrvEntry ose_signal_driver_entry = {
  .init                       = drv_init,
  .start                      = drv_start,
  .stop                       = drv_stop,
  .ready_input                = ready_input,
  .driver_name                = DRIVER_NAME,
  .control                    = control,
  .extended_marker            = ERL_DRV_EXTENDED_MARKER,
  .major_version              = ERL_DRV_EXTENDED_MAJOR_VERSION,
  .minor_version              = ERL_DRV_EXTENDED_MINOR_VERSION,
  .driver_flags               = ERL_DRV_FLAG_USE_PORT_LOCKING,
  .stop_select                = stop_select
};
```

2 Reference Manual

The Standard Erlang Libraries application, *STDLIB*, contains modules for manipulating lists, strings and files etc.

ose

Application

The OSE application contains modules and documentation that only applies when running Erlang/OTP on Enea OSE.

ose

Erlang module

Interface module for OSE messaging and process monitoring from Erlang

For each mailbox created through *open/1* a OSE phantom process with that name is started. Since phantom processes are used the memory footprint of each mailbox is quite small.

To receive messages you first have to subscribe to the specific message numbers that you are interested in with *listen/2*. The messages will be sent to the Erlang process that created the mailbox.

DATA TYPES

`attach_ref()`

Reference from an attach request. This term will be included in the term returned when the attached mailbox disappears.

`hunt_ref()`

Reference from a hunt request. This term will be included in a successful hunt response.

`mailbox()`

Mailbox handle. Implemented as an erlang port.

`mailbox_id()`

Mailbox ID, this is the same as the process id of an OSE process. An integer.

`message_number() = 0..4294967295`

OSE Signal number

Exports

`attach(Port, Pid) -> Ref`

Types:

`Port = mailbox()`
`Pid = mailbox_id()`
`Ref = attach_ref()`

Attach to an OSE process.

Will send `{mailbox_down, Port, Ref, MboxId}` to the calling process if the OSE process exits.

Returns a reference that can be used to cancel the attachment using *detach/2*.

raises: `badarg` | `enomem`

`close(Port) -> ok`

Types:

`Port = mailbox()`

Close a mailbox

This kills the OSE phantom process associated with this mailbox.

Will also consume any `{'EXIT',Port,_}` message from the port that comes due to the port closing when the calling process traps exits.

raises: `badarg`

`dehunt(Port, Ref) -> ok`

Types:

`Port = mailbox()`
`Ref = hunt_ref()`

Stop hunting for OSE process.

If a message for this hunt has been sent but not received by the calling process, it is removed from the message queue. Note that this only works if the same process that did the hunt does the dehunt.

raises: `badarg`

See also: hunt/2.

`detach(Port, Ref) -> ok`

Types:

`Port = mailbox()`
`Ref = attach_ref()`

Remove attachment to an OSE process.

If a message for this monitor has been sent but not received by the calling process, it is removed from the message queue. Note that this only works of the same process that did the attach does the detach.

raises: `badarg`

See also: attach/2.

`get_id(Port) -> Pid`

Types:

`Port = mailbox()`
`Pid = mailbox_id()`

Get the mailbox id for the given port.

The mailbox id is the same as the OSE process id of the OSE phantom process that this mailbox represents.

raises: `badarg`

`get_name(Port, Pid) -> Name | undefined`

Types:

`Port = mailbox()`
`Pid = mailbox_id()`
`Name = binary()`

Get the mailbox name for the given mailbox id.

The mailbox name is the name of the OSE process with process id Pid.

This call will fail with badarg if the underlying system does not support getting the name from a process id.

raises: `badarg`

hunt(Port, HuntPath) -> Ref

Types:

Port = mailbox()
HuntPath = iodata()
Ref = hunt_ref()

Hunt for OSE process by name.

Will send `{mailbox_up, Port, Ref, MboxId}` to the calling process when the OSE process becomes available.

Returns a reference term that can be used to cancel the hunt using *dehunt/2*.

raises: `badarg`

listen(Port, SigNos) -> ok

Types:

Port = mailbox()
SigNos = [message_number()]

Start listening for specified OSE signal numbers.

The mailbox will send `{message,Port,{FromMboxId,ToMboxId,MsgNo,MsgData}}` to the process that created the mailbox when an OSE message with any of the specified `SigNos` arrives.

Repeated calls to listen will replace the current set of signal numbers to listen to. i.e

```
  1>ose:listen(MsgB,[1234,12345]).
ok
2> ose:listen(MsgB,[1234,123456]).
ok.
```

The above will first listen for signals with numbers 1234 and 12345, and then replace that with only listening to 1234 and 123456.

With the current implementation it is not possible to listen to all signal numbers.

raises: `badarg` | `enomem`

open(Name) -> Port

Types:

Name = iodata()
Port = mailbox()

Create a mailbox with the given name and return a port that handles the mailbox.

An OSE phantom process with the given name will be created that will send any messages sent through this mailbox. Any messages sent to the new OSE process will automatically be converted to an Erlang message and sent to the Erlang process that calls this function. See *listen/2* for details about the format of the message sent.

The caller gets linked to the created mailbox.

raises: `badarg` | `system_limit`

See also: listen/2.

send(Port, Pid, SigNo, SigData) -> ok

Types:

Port = mailbox()
Pid = mailbox_id()
SigNo = message_number()
SigData = iodata()

Send an OSE message.

The message is sent from the OSE process' own ID that is: `get_id(Port)`.

raises: `badarg`

See also: send/5.

send(Port, Pid, SenderPid, SigNo, SigData) -> ok

Types:

Port = mailbox()
Pid = mailbox_id()
SenderPid = mailbox_id()
SigNo = message_number()
SigData = iodata()

Send an OSE message with different sender.

As *send/4* but the sender will be `SenderPid`.

raises: `badarg`

See also: send/4.

ose_erl_driver

C Library

Writing Linked-in drivers that also work on Enea OSE is very similar for how you would do it for Unix. The difference from Unix is that driver_select, ready_input and ready_output all work with signals instead of file descriptors. This means that the driver_select is used to specify which type of signal should trigger calls to ready_input/ready_output. The functions described below are available to driver programmers on Enea OSE to facilitate this.

DATA TYPES

union SIGNAL
: See the Enea OSE SPI documentation for a description.

SIGSELECT
: See the Enea OSE SPI documentation for a description.

ErlDrvEvent
: The `ErlDrvEvent` is a handle to a signal number and id combination. It is passed to *driver_select(3)*.

ErlDrvOseEventId
: This is the id used to associate a specific signal to a certain driver instance.

Exports

`union SIGNAL *erl_drv_ose_get_signal(ErlDrvEvent drv_event)`

Fetch the next signal associated with `drv_event`. Signals will be returned in the order which they were received and when no more signals are available `NULL` will be returned. Use this function in the ready_input/ready_output callbacks to get signals.

`ErlDrvEvent erl_drv_ose_event_alloc(SIGSELECT signo, ErlDrvOseEventId id, ErlDrvOseEventId (*resolve_signal)(union SIGNAL* sig), void *extra)`

Create a new `ErlDrvEvent` associated with `signo`, `id` and uses the `resolve_signal` function to extract the `id` from a signal with `signo`. The `extra` parameter can be used for additional data. See *Signals in a Linked-in driver* in the OSE User's Guide.

`void erl_drv_ose_event_free(ErlDrvEvent drv_event)`

Free a `ErlDrvEvent`. This should always be done in the *stop_select* callback when the event is no longer being used.

`void erl_drv_ose_event_fetch(ErlDrvEvent drv_event, SIGSELECT *signo, ErlDrvOseEventId *id, void **extra)`

Write the signal number, id and any extra data associated with `drv_event` into `*signo` and `*id` respectively. `NULL` can be also passed as `signo` or `id` in order to ignore that field.

SEE ALSO

driver_entry(3), *erl_driver(3)*