1 ERTS User's Guide

1.1 Introduction

1.1.1 Scope

The Erlang Runtime System Application, ERTS, contains functionality necessary to run the Erlang system.

Note:

By default, ERTS is only guaranteed to be compatible with other Erlang/OTP components from the same release as ERTS itself.

For information on how to communicate with Erlang/OTP components from earlier releases, see the documentation of system flag +R in erl(1).

1.1.2 Prerequisites

It is assumed that the reader is familiar with the Erlang programming language.

1.2 Communication in Erlang

Communication in Erlang is conceptually performed using asynchronous signaling. All different executing entities, such as processes and ports, communicate through asynchronous signals. The most commonly used signal is a message. Other common signals are exit, link, unlink, monitor, and demonitor signals.

1.2.1 Passing of Signals

This information has been moved to the Signals section of the Processes chapter in the Erlang Reference Manual.

1.2.2 Synchronous Communication

This information has been moved to the Signals section of the Processes chapter in the Erlang Reference Manual.

1.2.3 Implementation

The implementation of different asynchronous signals in the virtual machine can vary over time, but the behavior always respects this concept of asynchronous signals being passed between entities as described above.

By inspecting the implementation, you might notice that some specific signal gives a stricter guarantee than described above. It is of vital importance that such knowledge about the implementation is not used by Erlang code, as the implementation can change at any time without prior notice.

Examples of major implementation changes:

• As from ERTS 5.5.2 exit signals to processes are truly asynchronously delivered.
• As from ERTS 5.10 all signals from processes to ports are truly asynchronously delivered.
1.3 Time and Time Correction in Erlang

1.3.1 Extended Time Functionality

As of Erlang/OTP 18 (ERTS 7.0) the time functionality was extended. This includes a new API for time and time warp modes that change the system behavior when system time changes.

Note:

As of Erlang/OTP 26 (ERTS 14.0) the multi time warp mode is enabled by default. This assumes that all code executing on the system is time warp safe.

If you have old code in the system that is not time warp safe, you now explicitly need to start the system in no time warp mode (or singe time warp mode if it is partially time warp safe) in order to avoid problems. When starting the system in no time warp mode, the system behaves as it did prior to the introduction of the extended time functionality introduced in OTP 18.

If you have code that is not time warp safe, you are strongly encouraged to change this so that you can use multi time warp mode. Compared to no time warp mode, multi time warp mode improves scalability and performance as well as accuracy and precision of time measurements.

1.3.2 Terminology

To make it easier to understand this section, some terms are defined. This is a mix of our own terminology (Erlang/OS system time, Erlang/OS monotonic time, time warp) and globally accepted terminology.

Monotonically Increasing

In a monotonically increasing sequence of values, all values that have a predecessor are either larger than or equal to its predecessor.

Strictly Monotonically Increasing

In a strictly monotonically increasing sequence of values, all values that have a predecessor are larger than its predecessor.

UT1

Universal Time. UT1 is based on the rotation of the earth and conceptually means solar time at 0° longitude.

UTC

Coordinated Universal Time. UTC almost aligns with UT1. However, UTC uses the SI definition of a second, which has not exactly the same length as the second used by UT1. This means that UTC slowly drifts from UT1. To keep UTC relatively in sync with UT1, leap seconds are inserted, and potentially also deleted. That is, an UTC day can be 86400, 86401, or 86399 seconds long.

POSIX Time

Time since Epoch. Epoch is defined to be 00:00:00 UTC, 1970-01-01. A day in POSIX time is defined to be exactly 86400 seconds long. Strangely enough, Epoch is defined to be a time in UTC, and UTC has another definition of how long a day is. Quoting the Open Group "POSIX time is therefore not necessarily UTC, despite its appearance". The effect of this is that when an UTC leap second is inserted, POSIX time either stops for a second, or repeats the last second. If an UTC leap second would be deleted (which has not happened yet), POSIX time would make a one second leap forward.
1.3 Time and Time Correction in Erlang

Time Resolution
The shortest time interval that can be distinguished when reading time values.

Time Precision
The shortest time interval that can be distinguished repeatedly and reliably when reading time values. Precision is limited by the resolution, but resolution and precision can differ significantly.

Time Accuracy
The correctness of time values.

Time Warp
A time warp is a leap forwards or backwards in time. That is, the difference of time values taken before and after the time warp does not correspond to the actual elapsed time.

OS System Time
The operating systems view of POSIX time. To retrieve it, call `os:system_time()`. This may or may not be an accurate view of POSIX time. This time may typically be adjusted both backwards and forwards without limitation. That is, time warps may be observed.

To get information about the Erlang runtime system's source of OS system time, call `erlang:system_info(os_system_time_source)`.

OS Monotonic Time
A monotonically increasing time provided by the OS. This time does not leap and has a relatively steady frequency although not completely correct. However, it is not uncommon that OS monotonic time stops if the system is suspended. This time typically increases since some unspecified point in time that is not connected to OS system time. This type of time is not necessarily provided by all OSs.

To get information about the Erlang runtime system's source of OS monotonic time, call `erlang:system_info(os_monotonic_time_source)`.

Erlang System Time
The Erlang runtime systems view of POSIX time. To retrieve it, call `erlang:system_time()`.

This time may or may not be an accurate view of POSIX time, and may or may not align with OS system time. The runtime system works towards aligning the two system times. Depending on the time warp mode used, this can be achieved by letting Erlang system time perform a time warp.

Erlang Monotonic Time
A monotonically increasing time provided by the Erlang runtime system. Erlang monotonic time increases since some unspecified point in time. To retrieve it, call `erlang:monotonic_time()`.

The accuracy and precision of Erlang monotonic time heavily depends on the following:
- Accuracy and precision of OS monotonic time
- Accuracy and precision of OS system time
- Time warp mode used

On a system without OS monotonic time, Erlang monotonic time guarantees monotonicity, but cannot give other guarantees. The frequency adjustments made to Erlang monotonic time depend on the time warp mode used.

Internally in the runtime system, Erlang monotonic time is the “time engine” that is used for more or less everything that has anything to do with time. All timers, regardless of it is a receive ... after timer, BIF timer, or a timer
1.3 Time and Time Correction in Erlang

in the timer(3) module, are triggered relative Erlang monotonic time. Even Erlang system time is based on Erlang monotonic time. By adding current Erlang monotonic time with current time offset, you get current Erlang system time.

To retrieve the current time offset, call `erlang:time_offset/0`.

1.3.3 Introduction

Time is vital to an Erlang program and, more importantly, correct time is vital to an Erlang program. As Erlang is a language with soft real-time properties and we can express time in our programs, the Virtual Machine and the language must be careful about what is considered a correct time and in how time functions behave.

When Erlang was designed, it was assumed that the wall clock time in the system showed a monotonic time moving forward at exactly the same pace as the definition of time. This more or less meant that an atomic clock (or better time source) was expected to be attached to your hardware and that the hardware was then expected to be locked away from any human tinkering forever. While this can be a compelling thought, it is simply never the case.

A “normal” modern computer cannot keep time, not on itself and not unless you have a chip-level atomic clock wired to it. Time, as perceived by your computer, must normally be corrected. Hence the Network Time Protocol (NTP) protocol, together with the `ntpd` process, does its best to keep your computer time in sync with the correct time. Between NTP corrections, usually a less potent time-keeper than an atomic clock is used.

However, NTP is not fail-safe. The NTP server can be unavailable, `ntp.conf` can be wrongly configured, or your computer can sometimes be disconnected from Internet. Furthermore, you can have a user (or even system administrator) who thinks the correct way to handle Daylight Saving Time is to adjust the clock one hour two times a year (which is the incorrect way to do it). To complicate things further, this user fetched your software from Internet and has not considered what the correct time is as perceived by a computer. The user does not care about keeping the wall clock in sync with the correct time. The user expects your program to have unlimited knowledge about the time.

Most programmers also expect time to be reliable, at least until they realize that the wall clock time on their workstation is off by a minute. Then they set it to the correct time, but most probably not in a smooth way.

The number of problems that arise when you always expect the wall clock time on the system to be correct can be immense. Erlang therefore introduced the "corrected estimate of time", or the "time correction", many years ago. The time correction relies on the fact that most operating systems have some kind of monotonic clock, either a real-time extension or some built-in "tick counter" that is independent of the wall clock settings. This counter can have microsecond resolution or much less, but it has a drift that cannot be ignored.

1.3.4 Time Correction

If time correction is enabled, the Erlang runtime system makes use of both OS system time and OS monotonic time, to adjust the frequency of the Erlang monotonic clock. Time correction ensures that Erlang monotonic time does not warp and that the frequency is relatively accurate. The type of frequency adjustments depends on the time warp mode used. Section Time Warp Modes provides more details.

By default time correction is enabled if support for it exists on the specific platform. Support for it includes both OS monotonic time, provided by the OS, and an implementation in the Erlang runtime system using OS monotonic time. To check if your system has support for OS monotonic time, call `erlang:system_info(os_monotonic_time_source)`. To check if time correction is enabled on your system, call `erlang:system_info(time_correction)`.

To enable or disable time correction, pass command-line argument `+c [true|false]` to `erl(1)`.

If time correction is disabled, Erlang monotonic time can warp forwards or stop, or even freeze for extended periods of time. There are then no guarantees that the frequency of the Erlang monotonic clock is accurate or stable.

**You typically never want to disable time correction.** Previously a performance penalty was associated with time correction, but nowadays it is usually the other way around. If time correction is disabled, you probably get bad scalability, bad performance, and bad time measurements.
1.3.5 Time Warp Safe Code

Time warp safe code can handle a time warp of Erlang system time. `erlang:now/0` behaves bad when Erlang system time warps. When Erlang system time does a time warp backwards, the values returned from `erlang:now/0` freeze (if you disregard the microsecond increments made because of the actual call) until OS system time reaches the point of the last value returned by `erlang:now/0`. This freeze can continue for a long time. It can take years, decades, and even longer until the freeze stops.

All uses of `erlang:now/0` are not necessarily time warp unsafe. If you do not use it to get time, it is time warp safe. However, all uses of `erlang:now/0` are suboptimal from a performance and scalability perspective. So you really want to replace the use of it with other functionality. For examples of how to replace the use of `erlang:now/0`, see section How to Work with the New API.

1.3.6 Time Warp Modes

Current Erlang system time is determined by adding the current Erlang monotonic time with current time offset. The time offset is managed differently depending on which time warp mode you use.

To set the time warp mode, pass command-line argument `+C [no_time_warp|single_time_warp|multi_time_warp]` to `erl(1)`.

**No Time Warp Mode**

The time offset is determined at runtime system start and does not change later. This is the same behavior as was default prior to OTP 26 (ERTS 14.0), and the only behavior prior to OTP 18 (ERTS 7.0).

As the time offset is not allowed to change, time correction must adjust the frequency of the Erlang monotonic clock to align Erlang system time with OS system time smoothly. A significant downside of this approach is that we on purpose will use a faulty frequency on the Erlang monotonic clock if adjustments are needed. This error can be as large as 1%. This error will show up in all time measurements in the runtime system.

If time correction is not enabled, Erlang monotonic time freezes when OS system time leaps backwards. The freeze of monotonic time continues until OS system time catches up. The freeze can continue for a long time. When OS system time leaps forwards, Erlang monotonic time also leaps forward.

**Single Time Warp Mode**

This mode is more or less a backward compatibility mode as from its introduction.

On an embedded system it is not uncommon that the system has no power supply, not even a battery, when it is shut off. The system clock on such a system is typically way off when the system boots. If no time warp mode is used, and the Erlang runtime system is started before OS system time has been corrected, Erlang system time can be wrong for a long time, centuries or even longer.

If you need to use Erlang code that is not time warp safe, and you need to start the Erlang runtime system before OS system time has been corrected, you may want to use the single time warp mode.

**Note:**

There are limitations to when you can execute time warp unsafe code using this mode. If it is possible to use time warp safe code only, it is much better to use the multi-time warp mode instead.

Using the single time warp mode, the time offset is handled in two phases:

**Preliminary Phase**

This phase starts when the runtime system starts. A preliminary time offset based on current OS system time is determined. This offset is from now on to be fixed during the whole preliminary phase.
1.3 Time and Time Correction in Erlang

If time correction is enabled, adjustments to the Erlang monotonic clock are made to keep its frequency as correct as possible. However, no adjustments are made trying to align Erlang system time and OS system time. That is, during the preliminary phase Erlang system time and OS system time can diverge from each other, and no attempt is made to prevent this.

If time correction is disabled, changes in OS system time affects the monotonic clock the same way as when the no time warp mode is used.

Final Phase

This phase begins when the user finalizes the time offset by calling `erlang:system_flag(time_offset, finalize)`. The finalization can only be performed once.

During finalization, the time offset is adjusted and fixed so that current Erlang system time aligns with the current OS system time. As the time offset can change during the finalization, Erlang system time can do a time warp at this point. The time offset is from now on fixed until the runtime system terminates. If time correction has been enabled, the time correction from now on also makes adjustments to align Erlang system time with OS system time. When the system is in the final phase, it behaves exactly as in no time warp mode.

In order for this to work properly, the user must ensure that the following two requirements are satisfied:

Forward Time Warp

The time warp made when finalizing the time offset can only be done forwards without encountering problems. This implies that the user must ensure that OS system time is set to a time earlier or equal to actual POSIX time before starting the Erlang runtime system.

If you are not sure that OS system time is correct, set it to a time that is guaranteed to be earlier than actual POSIX time before starting the Erlang runtime system, just to be safe.

Finalize Correct OS System Time

OS system time must be correct when the user finalizes the time offset.

If these requirements are not fulfilled, the system may behave very bad.

Assuming that these requirements are fulfilled, time correction is enabled, and OS system time is adjusted using a time adjustment protocol such as NTP, only small adjustments of Erlang monotonic time are needed to keep system times aligned after finalization. As long as the system is not suspended, the largest adjustments needed are for inserted (or deleted) leap seconds.

**Warning:**

To use this mode, ensure that all Erlang code that will execute in both phases is time warp safe.

Code executing only in the final phase does not have to be able to cope with the time warp.

Multi-Time Warp Mode

Multi-time warp mode in combination with time correction is the preferred configuration. This as the Erlang runtime system have better performance, scale better, and behave better on almost all platforms. Also, the accuracy and precision of time measurements are better. Only Erlang runtime systems executing on ancient platforms benefit from another configuration. As of OTP 26 (ERTS 14.0) this is also the default.

The time offset can change at any time without limitations. That is, Erlang system time can perform time warps both forwards and backwards at any time. As we align Erlang system time with OS system time by changing the time offset, we can enable a time correction that tries to adjust the frequency of the Erlang monotonic clock to be as correct as possible. This makes time measurements using Erlang monotonic time more accurate and precise.
1.3 Time and Time Correction in Erlang

If time correction is disabled, Erlang monotonic time leaps forward if OS system time leaps forward. If OS system time leaps backwards, Erlang monotonic time stops briefly, but it does not freeze for extended periods of time. This as the time offset is changed to align Erlang system time with OS system time.

**Warning:**

To use this mode, ensure that all Erlang code that will execute on the runtime system is time warp safe.

### 1.3.7 New Time API

The old time API is based on `erlang:now/0`. `erlang:now/0` was intended to be used for many unrelated things. This tied these unrelated operations together and caused issues with performance, scalability, accuracy, and precision for operations that did not need to have such issues. To improve this, the new API spreads different functionality over multiple functions.

To be backward compatible, `erlang:now/0` remains "as is", but you are strongly discouraged from using it. Many use cases of `erlang:now/0` prevents you from using the new multi-time warp mode, which is an important part of this new time functionality improvement.

Some of the new BIFs on some systems, perhaps surprisingly, return negative integer values on a newly started runtime system. This is not a bug, but a memory use optimization.

The new API consists of the following new BIFs:

- `erlang:convert_time_unit/3`
- `erlang:monotonic_time/0`
- `erlang:monotonic_time/1`
- `erlang:system_time/0`
- `erlang:system_time/1`
- `erlang:time_offset/0`
- `erlang:time_offset/1`
- `erlang:timestamp/0`
- `erlang:unique_integer/0`
- `erlang:unique_integer/1`
- `os:system_time/0`
- `os:system_time/1`

The new API also consists of extensions of the following existing BIFs:

- `erlang:monitor(time_offset, clock_service)`
- `erlang:system_flag(time_offset, finalize)`
- `erlang:system_info(os_monotonic_time_source)`
- `erlang:system_info(os_system_time_source)`
- `erlang:system_info(time_offset)`
- `erlang:system_info(time_warp_mode)`
- `erlang:system_info(time_correction)`
- `erlang:system_info(start_time)`
- `erlang:system_info(end_time)`
New Erlang Monotonic Time

Erlang monotonic time as such is new as from ERTS 7.0. It is introduced to detach time measurements, such as elapsed time from calendar time. In many use cases there is a need to measure elapsed time or specify a time relative to another point in time without the need to know the involved times in UTC or any other globally defined time scale. By introducing a time scale with a local definition of where it starts, time that do not concern calendar time can be managed on that time scale. Erlang monotonic time uses such a time scale with a locally defined start.

The introduction of Erlang monotonic time allows us to adjust the two Erlang times (Erlang monotonic time and Erlang system time) separately. By doing this, the accuracy of elapsed time does not have to suffer just because the system time happened to be wrong at some point in time. Separate adjustments of the two times are only performed in the time warp modes, and only fully separated in the multi-time warp mode. All other modes than the multi-time warp mode are for backward compatibility reasons. When using these modes, the accuracy of Erlang monotonic time suffer, as the adjustments of Erlang monotonic time in these modes are more or less tied to Erlang system time.

The adjustment of system time could have been made smother than using a time warp approach, but we think that would be a bad choice. As we can express and measure time that is not connected to calendar time by the use of Erlang monotonic time, it is better to expose the change in Erlang system time immediately. This as the Erlang applications executing on the system can react on the change in system time as soon as possible. This is also more or less exactly how most operating systems handle this (OS monotonic time and OS system time). By adjusting system time smoothly, we would just hide the fact that system time changed and make it harder for the Erlang applications to react to the change in a sensible way.

To be able to react to a change in Erlang system time, you must be able to detect that it happened. The change in Erlang system time occurs when the current time offset is changed. We have therefore introduced the possibility to monitor the time offset using `erlang:monitor(time_offset, clock_service)`. A process monitoring the time offset is sent a message on the following format when the time offset is changed:

```
{'CHANGE', MonitorReference, time_offset, clock_service, NewTimeOffset}
```

Unique Values

Besides reporting time, `erlang:now/0` also produces unique and strictly monotonically increasing values. To detach this functionality from time measurements, we have introduced `erlang:unique_integer()`.

How to Work with the New API

Previously `erlang:now/0` was the only option for doing many things. This section deals with some things that `erlang:now/0` can be used for, and how you use the new API.

Retrieve Erlang System Time

<table>
<thead>
<tr>
<th>Don’t:</th>
</tr>
</thead>
<tbody>
<tr>
<td>Use <code>erlang:now/0</code> to retrieve the current Erlang system time.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Do:</th>
</tr>
</thead>
<tbody>
<tr>
<td>Use <code>erlang:system_time/1</code> to retrieve the current Erlang system time on the time unit of your choice.</td>
</tr>
<tr>
<td>If you want the same format as returned by <code>erlang:now/0</code>, use <code>erlang:timestamp/0</code>.</td>
</tr>
</tbody>
</table>
Measure Elapsed Time

**Don’t:**
Take time stamps with `erlang:now/0` and calculate the difference in time with `timer:now_diff/2`.

**Do:**
Take time stamps with `erlang:monotonic_time/0` and calculate the time difference using ordinary subtraction. The result is in native time unit. If you want to convert the result to another time unit, you can use `erlang:convert_time_unit/3`.

An easier way to do this is to use `erlang:monotonic_time/1` with the desired time unit. However, you can then lose accuracy and precision.

Determine Order of Events

**Don’t:**
Determine the order of events by saving a time stamp with `erlang:now/0` when the event occurs.

**Do:**
Determine the order of events by saving the integer returned by `erlang:unique_integer([monotonic])` when the event occurs. These integers are strictly monotonically ordered on current runtime system instance corresponding to creation time.

Determine Order of Events with Time of the Event

**Don’t:**
Determine the order of events by saving a time stamp with `erlang:now/0` when the event occurs.
1.3 Time and Time Correction in Erlang

**Do:**

Determine the order of events by saving a tuple containing monotonic time and a strictly monotonically increasing integer as follows:

```
Time = erlang:monotonic_time(),
UMI = erlang:unique_integer([monotonic]),
EventTag = {Time, UMI}
```

These tuples are strictly monotonically ordered on the current runtime system instance according to creation time. It is important that the monotonic time is in the first element (the most significant element when comparing two-tuples). Using the monotonic time in the tuples, you can calculate time between events.

If you are interested in Erlang system time at the time when the event occurred, you can also save the time offset before or after saving the events using `erlang:time_offset/0`. Erlang monotonic time added with the time offset corresponds to Erlang system time.

If you are executing in a mode where time offset can change, and you want to get the actual Erlang system time when the event occurred, you can save the time offset as a third element in the tuple (the least significant element when comparing three-tuples).

**Create a Unique Name**

**Don't:**

Use the values returned from `erlang:now/0` to create a name unique on the current runtime system instance.

**Do:**

Use the value returned from `erlang:unique_integer/0` to create a name unique on the current runtime system instance. If you only want positive integers, you can use `erlang:unique_integer([positive])`.

**Seed Random Number Generation with a Unique Value**

**Don't:**

Seed random number generation using `erlang:now/0`.

**Do:**

Seed random number generation using a combination of `erlang:monotonic_time()`, `erlang:time_offset()`, `erlang:unique_integer()`, and other functionality.

To sum up this section: **Do not use `erlang:now/0`.**

### 1.3.8 Support of Both New and Old OTP Releases

It can be required that your code must run on a variety of OTP installations of different OTP releases. If so, you cannot use the new API out of the box, as it will not be available on releases before OTP 18. The solution is **not** to avoid using the new API, as your code would then not benefit from the scalability and accuracy improvements made. Instead, use the new API when available, and fall back on `erlang:now/0` when the new API is unavailable.

Fortunately most of the new API can easily be implemented using existing primitives, except for:
1.4 Match Specifications in Erlang

A "match specification" (match_spec) is an Erlang term describing a small "program" that tries to match something. It can be used to either control tracing with erlang:trace_pattern/3 or to search for objects in an ETS table with for example ets:select/2. The match specification in many ways works like a small function in Erlang, but is interpreted/compiled by the Erlang runtime system to something much more efficient than calling an Erlang function. The match specification is also very limited compared to the expressiveness of real Erlang functions.

The most notable difference between a match specification and an Erlang fun is the syntax. Match specifications are Erlang terms, not Erlang code. Also, a match specification has a strange concept of exceptions:

- An exception (such as badarg) in the MatchCondition part, which resembles an Erlang guard, generates immediate failure.
- An exception in the MatchBody part, which resembles the body of an Erlang function, is implicitly caught and results in the single atom 'EXIT'.

1.4.1 Grammar

A match specification used in tracing can be described in the following informal grammar:

- MatchExpression ::= [ MatchFunction, ... ]
- MatchFunction ::= { MatchHead, MatchConditions, MatchBody }
- MatchHead ::= MatchVariable | '_' | [ MatchHeadPart, ... ]
- MatchHeadPart ::= term() | MatchVariable | '_'
- MatchVariable ::= '$<number>'
- MatchConditions ::= [ MatchCondition, ... ] | []
- MatchCondition ::= { GuardFunction } | { GuardFunction, ConditionExpression, ... }
- BoolFunction ::= is_atom | is_float | is_integer | is_list | is_number | is_pid | is_port | is_reference | is_tuple | is_map | is_map_key | is_binary | is_bitstring | is_boolean | is_function | is_record | is_seq_trace | 'and' | 'or' | 'not' | 'xor' | 'andalso' | 'orelse'
- ConditionExpression ::= ExprMatchVariable | { GuardFunction } | { GuardFunction, ConditionExpression, ... } | TermConstruct
- ExprMatchVariable ::= MatchVariable (bound in the MatchHead) | '$_' | '$$'
- TermConstruct = {} | [[]] | {{ ConditionExpression, ... }} | [] | ConditionExpression, ... | %{ term() } => ConditionExpression, ... | | NonCompositeTerm | Constant
- NonCompositeTerm ::= term() (not list or tuple or map)
- Constant ::= { const, term() }
- GuardFunction ::= BoolFunction | abs | element | hd | length | map_get | map_size | max | min | node | float | round | floor | ceil | size | bit_size | byte_size | tuple_size | tl | trunc | binary_part | '+' | '-' | '*' | 'div' | 'rem' | 'band' | 'bor' | 'bxor' | 'bnot' | 'bsl' | 'bsr' | '>=>' | '<=' | '>=<>' | '==<>' | '===' | '=/=' | '/=' | self | get_tcw
1.4 Match Specifications in Erlang

- MatchBody ::= [ ActionTerm ]
- ActionTerm ::= ConditionExpression | ActionCall
- ActionCall ::= {ActionFunction} | {ActionFunction, ActionTerm, ...}
- ActionFunction ::= set_seq_token | get_seq_token | message | return_trace | exception_trace | process_dump | enable_trace | disable_trace | trace | display | caller | caller_line | current_stacktrace | set_tcw | silent

A match specification used in ets(3) can be described in the following informal grammar:

- MatchExpression ::= [ MatchFunction, ... ]
- MatchFunction ::= { MatchHead, MatchConditions, MatchBody }
- MatchHead ::= MatchVariable | '_-' | { MatchHeadPart, ... }
- MatchHeadPart ::= term() | MatchVariable | '_-
- MatchVariable ::= '$<number>'
- MatchConditions ::= [ MatchCondition, ...] | []
- MatchCondition ::= { GuardFunction } | { GuardFunction, ConditionExpression, ... }
- GuardFunction ::= BoolFunction | abs | element | hd | length | map_get | map_size | max | min | node | float | round | floor | ceil | size | bit_size | byte_size | tuple_size | tl | trunc | binary_part | '+' | '-' | '*' | 'div' | 'rem' | 'band' | 'bor' | 'bxor' | 'bnot' | 'bsl' | 'bsr' | '>' | '>=' | '<' | '<=' | '==' | '=/=' | '/=' | 'self
- MatchBody ::= [ ConditionExpression, ... ]

1.4.2 Function Descriptions

Functions Allowed in All Types of Match Specifications

The functions allowed in match_spec work as follows:

- is_atom, is_boolean, is_float, is_integer, is_list, is_number, is_pid, is_port, is_reference, is_tuple, is_map, is_binary, is_bitstring, is_function

  Same as the corresponding guard tests in Erlang, return true or false.

- is_record

  Takes an additional parameter, which must be the result of record_info(size, <record_type>), like in {is_record, '$1', rectype, record_info(size, rectype)}.

- 'not'

  Negates its single argument (anything other than false gives false).
1.4 Match Specifications in Erlang

'and'
Returns true if all its arguments (variable length argument list) evaluate to true, otherwise false. Evaluation order is undefined.

'or'
Returns true if any of its arguments evaluates to true. Variable length argument list. Evaluation order is undefined.

'andalso'
Works as 'and', but quits evaluating its arguments when one argument evaluates to something else than true. Arguments are evaluated left to right.

'orelse'
Works as 'or', but quits evaluating as soon as one of its arguments evaluates to true. Arguments are evaluated left to right.

'xor'
Only two arguments, of which one must be true and the other false to return true; otherwise 'xor' returns false.

abs, element, hd, length, map_get, map_size, max, min, node, round, ceil, floor, float, size, bit_size, byte_size, tuple_size, tl, trunc, binary_part, '+', '-', '*', 'div', 'rem', 'band', 'bor', 'bxor', 'bnot', 'bsl', 'bsr', '>', '>=', '<', '=<', '==', '===', '/=/', '/=/', self

Same as the corresponding Erlang BIFs (or operators). In case of bad arguments, the result depends on the context. In the MatchConditions part of the expression, the test fails immediately (like in an Erlang guard). In the MatchBody part, exceptions are implicitly caught and the call results in the atom 'EXIT'.

Functions Allowed Only for Tracing
The functions allowed only for tracing work as follows:

is_seq_trace
Returns true if a sequential trace token is set for the current process, otherwise false.

set_seq_token
Works as seq_trace:set_token/2, but returns true on success, and 'EXIT' on error or bad argument. Only allowed in the MatchBody part and only allowed when tracing.

get_seq_token
Same as seq_trace:get_token/0 and only allowed in the MatchBody part when tracing.

message
Sets an additional message appended to the trace message sent. One can only set one additional message in the body. Later calls replace the appended message.

As a special case, {message, false} disables sending of trace messages ('call' and 'return_to') for this function call, just like if the match specification had not matched. This can be useful if only the side effects of the MatchBody part are desired.

Another special case is {message, true}, which sets the default behavior, as if the function had no match specification; trace message is sent with no extra information (if no other calls to message are placed before {message, true}, it is in fact a "noop").

Takes one argument: the message. Returns true and can only be used in the MatchBody part and when tracing.
1.4 Match Specifications in Erlang

return_trace
Causes a return_from trace message to be sent upon return from the current function. Takes no arguments, returns true and can only be used in the MatchBody part when tracing. If the process trace flag silent is active, the return_from trace message is inhibited.

Warning: If the traced function is tail-recursive, this match specification function destroys that property. Hence, if a match specification executing this function is used on a perpetual server process, it can only be active for a limited period of time, or the emulator will eventually use all memory in the host machine and crash. If this match specification function is inhibited using process trace flag silent, tail-recursiveness still remains.

eexception_trace
Works as return_trace plus; if the traced function exits because of an exception, an exception_from trace message is generated, regardless of the exception is caught or not.

process_dump
Returns some textual information about the current process as a binary. Takes no arguments and is only allowed in the MatchBody part when tracing.

enable_trace
With one parameter this function turns on tracing like the Erlang call erlang:trace(self(), true, [P2]), where P2 is the parameter to enable_trace.

With two parameters, the first parameter is to be either a process identifier or the registered name of a process. In this case tracing is turned on for the designated process in the same way as in the Erlang call erlang:trace(P1, true, [P2]), where P1 is the first and P2 is the second argument. The process P1 gets its trace messages sent to the same tracer as the process executing the statement uses. P1 cannot be one of the atoms all, new or existing (unless they are registered names). P2 cannot be cpu_timestamp or tracer.

Returns true and can only be used in the MatchBody part when tracing.

disable_trace
With one parameter this function disables tracing like the Erlang call erlang:trace(self(), false, [P2]), where P2 is the parameter to disable_trace.

With two parameters this function works as the Erlang call erlang:trace(P1, false, [P2]), where P1 can be either a process identifier or a registered name and is specified as the first argument to the match specification function. P2 cannot be cpu_timestamp or tracer.

Returns true and can only be used in the MatchBody part when tracing.

trace
With two parameters this function takes a list of trace flags to disable as first parameter and a list of trace flags to enable as second parameter. Logically, the disable list is applied first, but effectively all changes are applied atomically. The trace flags are the same as for erlang:trace/3, not including cpu_timestamp, but including tracer.

If a tracer is specified in both lists, the tracer in the enable list takes precedence. If no tracer is specified, the same tracer as the process executing the match specification is used (not the meta tracer). If that process doesn't have tracer either, then trace flags are ignored.

When using a tracer module, the module must be loaded before the match specification is executed. If it is not loaded, the match fails.

With three parameters to this function, the first is either a process identifier or the registered name of a process to set trace flags on, the second is the disable list, and the third is the enable list.
1.4 Match Specifications in Erlang

Returns true if any trace property was changed for the trace target process, otherwise false. Can only be used in the MatchBody part when tracing.

caller

Returns the calling function as a tuple {Module, Function, Arity} or the atom undefined if the calling function cannot be determined. Can only be used in the MatchBody part when tracing.

Notice that if a "technically built in function" (that is, a function not written in Erlang) is traced, the caller function sometimes returns the atom undefined. The calling Erlang function is not available during such calls.

caller_line

Similar to caller but returns additional information about the source code location of the function call-site within the caller function. Returns the calling function as a tuple {Module, Function, Arity, {File, Line}}. File is the string file name while Line is source line number. If the File and Line cannot be determined, {Module, Function, Arity, undefined} is returned. If the calling function cannot be determined, the atom undefined is returned. Can only be used in the MatchBody part when tracing.

Notice that if a "technically built in function" (that is, a function not written in Erlang) is traced, the caller_line function sometimes returns the atom undefined. The calling Erlang function is not available during such calls.

current_stacktrace

Returns the current call stack back-trace (stacktrace) of the calling function. The stack has the same format as in the catch part of a try. See The call-stack back trace (stacktrace). The depth of the stacktrace is truncated according to the backtrace_depth system flag setting.

Accepts a depth parameter. The depth value will be backtrace_depth if the argument is greater.

display

For debugging purposes only. Displays the single argument as an Erlang term on stdout, which is seldom what is wanted. Returns true and can only be used in the MatchBody part when tracing.

get_tcw

Takes no argument and returns the value of the node's trace control word. The same is done by erlang:system_info(trace_control_word).

The trace control word is a 32-bit unsigned integer intended for generic trace control. The trace control word can be tested and set both from within trace match specifications and with BIFs. This call is only allowed when tracing.

set_tcw

Takes one unsigned integer argument, sets the value of the node's trace control word to the value of the argument, and returns the previous value. The same is done by erlang:system_flag(trace_control_word, Value). It is only allowed to use set_tcw in the MatchBody part when tracing.

silent

Takes one argument. If the argument is true, the call trace message mode for the current process is set to silent for this call and all later calls, that is, call trace messages are inhibited even if {message, true} is called in the MatchBody part for a traced function.

This mode can also be activated with flag silent to erlang:trace/3.

If the argument is false, the call trace message mode for the current process is set to normal (non-silent) for this call and all later calls.

If the argument is not true or false, the call trace message mode is unaffected.
**1.4 Match Specifications in Erlang**

**Note:**
All "function calls" must be tuples, even if they take no arguments. The value of self is the atom() self, but the value of {self} is the pid() of the current process.

### 1.4.3 Match target

Each execution of a match specification is done against a match target term. The format and content of the target term depends on the context in which the match is done. The match target for ETS is always a full table tuple. The match target for call trace is always a list of all function arguments. The match target for event trace depends on the event type, see table below.

<table>
<thead>
<tr>
<th>Context</th>
<th>Type</th>
<th>Match target</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ETS</td>
<td></td>
<td>{Key, Value1, Value2, ...}</td>
<td>A table object</td>
</tr>
<tr>
<td>Trace</td>
<td>call</td>
<td>[Arg1, Arg2, ...]</td>
<td>Function arguments</td>
</tr>
<tr>
<td>Trace</td>
<td>send</td>
<td>[Receiver, Message]</td>
<td>Receiving process/port and message term</td>
</tr>
<tr>
<td>Trace</td>
<td>'receive'</td>
<td>[Node, Sender, Message]</td>
<td>Sending node, process/port and message term</td>
</tr>
</tbody>
</table>

Table 4.1: Match target depending on context

### 1.4.4 Variables and Literals

Variables take the form '$_<number>' where <number> is an integer between 0 and 100,000,000 (1e+8). The behavior if the number is outside these limits is **undefined**. In the MatchHead part, the special variable '_.' matches anything, and never gets bound (like _ in Erlang).

- In the MatchCondition/MatchBody parts, no unbound variables are allowed, so '_.' is interpreted as itself (an atom). Variables can only be bound in the MatchHead part.
- In the MatchBody and MatchCondition parts, only variables bound previously can be used.
- As a special case, the following apply in the MatchCondition/MatchBody parts:
  - The variable '_.' expands to the whole match target term.
  - The variable '$$' expands to a list of the values of all bound variables in order (that is, ['$1', '$2', ...]).

In the MatchHead part, all literals (except the variables above) are interpreted "as is".

In the MatchCondition/MatchBody parts, the interpretation is in some ways different. Literals in these parts can either be written "as is", which works for all literals except tuples, or by using the special form {const, T}, where T is any Erlang term.

For tuple literals in the match specification, double tuple parentheses can also be used, that is, construct them as a tuple of arity one containing a single tuple, which is the one to be constructed. The "double tuple parenthesis" syntax is useful to construct tuples from already bound variables, like in `{{'$1', [a,b,'$2']}ặ`. Examples:

<table>
<thead>
<tr>
<th>Expression</th>
<th>Variable Bindings</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
1.4 Match Specifications in Erlang

<table>
<thead>
<tr>
<th>{ ['$1'; '$2']}</th>
<th>'$1' = a, '$2' = b</th>
<th>{a, b}</th>
</tr>
</thead>
<tbody>
<tr>
<td>{const, ['$1', '$2']}</td>
<td>Irrelevant</td>
<td>{'$1', '$2'}</td>
</tr>
<tr>
<td>a</td>
<td>Irrelevant</td>
<td>a</td>
</tr>
<tr>
<td>'$_1'</td>
<td>'$1' = []</td>
<td>[]</td>
</tr>
<tr>
<td>['$_1']</td>
<td>'$1' = []</td>
<td>[[]]</td>
</tr>
<tr>
<td>[{</td>
<td>{a}</td>
<td>}]</td>
</tr>
<tr>
<td>42</td>
<td>Irrelevant</td>
<td>42</td>
</tr>
<tr>
<td>&quot;hello&quot;</td>
<td>Irrelevant</td>
<td>&quot;hello&quot;</td>
</tr>
<tr>
<td>$1</td>
<td>Irrelevant</td>
<td>49 (the ASCII value for character '1')</td>
</tr>
</tbody>
</table>

Table 4.2: Literals in MatchCondition/MatchBody Parts of a Match Specification

1.4.5 Execution of the Match

The execution of the match expression, when the runtime system decides whether a trace message is to be sent, is as follows:

For each tuple in the MatchExpression list and while no match has succeeded:

- Match the MatchHead part against the match target term, binding the '$<number>' variables (much like in ets:match/2). If the MatchHead part cannot match the arguments, the match fails.
- Evaluate each MatchCondition (where only '$<number>' variables previously bound in the MatchHead part can occur) and expect it to return the atom true. When a condition does not evaluate to true, the match fails. If any BIF call generates an exception, the match also fails.
- Two cases can occur:
  - If the match specification is executing when tracing:
    Evaluate each ActionTerm in the same way as the MatchConditions, but ignore the return values. Regardless of what happens in this part, the match has succeeded.
  - If the match specification is executed when selecting objects from an ETS table:
    Evaluate the expressions in order and return the value of the last expression (typically there is only one expression in this context).

1.4.6 Differences between Match Specifications in ETS and Tracing

ETS match specifications produce a return value. Usually the MatchBody contains one single ConditionExpression that defines the return value without any side effects. Calls with side effects are not allowed in the ETS context.

When tracing there is no return value to produce, the match specification either matches or does not. The effect when the expression matches is a trace message rather than a returned term. The ActionTerms are executed as in an imperative language, that is, for their side effects. Functions with side effects are also allowed when tracing.
1.4.7 Tracing Examples

Match an argument list of three, where the first and third arguments are equal:

\[
[[\{’$1’, ’\_’, ’$1’\},
  [],
  []]]
\]

Match an argument list of three, where the second argument is a number > 3:

\[
[[’\_’, ’$1’, ’\_’],
  [[’>’, ’$1’, 3]],
  []]]
\]

Match an argument list of three, where the third argument is either a tuple containing argument one and two, or a list beginning with argument one and two (that is, \([a, b, \{a, b, c\}]\) or \([a, b, \{a, b\}]\)):

\[
[[’$1’, ’$2’, ’$3’],
  [‘orelse’,
    [’=:=’, ’$3’, \{{’$1’, ’$2’}\}},
    [’and’,
      [’=:=’, ’$1’, \{hd, ’$3’\}],
      [’=:=’, ’$2’, \{hd, \{tl, ’$3’\}\}]}],
  []]]
\]

The above problem can also be solved as follows:

\[
[[’$1’, ’$2’, ’$3’],
  [[’$1’, ’$2’, [’$1’, ’$2’ | ’\_’]]],
  [],
  []]]
\]

Match two arguments, where the first is a tuple beginning with a list that in turn begins with the second argument times two (that is, \([\{(4, x), y\}, 2\}] or \([\{(8), y, z\}, 4]\)):

\[
[[’$1’, ’$2’],\{{’=:=’, ’\*', 2, ’$2’}, \{hd, \{element, 1, ’$1’\}\}}],
  []]]
\]

Match three arguments. When all three are equal and are numbers, append the process dump to the trace message, otherwise let the trace message be "as is", but set the sequential trace token label to 4711:

\[
[[’$1’, ’$1’, ’$1’],
  [[is_number, ’$1’]],
  [[message, (process_dump)]]],
  [’\_', [], [[set_seq_token, label, 4711]]]]
\]

As can be noted above, the parameter list can be matched against a single \texttt{MatchVariable} or an \texttt{’\_’}. To replace the whole parameter list with a single variable is a special case. In all other cases the \texttt{MatchHead} must be a proper list.

Generate a trace message only if the trace control word is set to 1:

\[
[’\_’,
  [[’=:=’,(get_tcw),(const, 1)]],
  []]]
\]

Generate a trace message only if there is a seq\_trace token:

\[
[’\_’,
  [[’=:=’,(is_seq_trace),(const, 1)]],
  []]]
\]

Remove the ’silent’ trace flag when the first argument is ’verbose’, and add it when it is ’silent’:
1.5 How to Interpret the Erlang Crash Dumps

This section describes the `erl_crash.dump` file generated upon abnormal exit of the Erlang runtime system.
The Erlang crash dump had a major facelift in Erlang/OTP R9C. The information in this section is therefore not directly applicable for older dumps. However, if you use crashdump_viewer(3) on older dumps, the crash dumps are translated into a format similar to this.

The system writes the crash dump in the current directory of the emulator or in the file pointed out by the environment variable (whatever that means on the current operating system) ERL_CRASH_DUMP. For a crash dump to be written, a writable file system must be mounted.

Crash dumps are written mainly for one of two reasons: either the built-in function erlang:halt/1 is called explicitly with a string argument from running Erlang code, or the runtime system has detected an error that cannot be handled. The most usual reason that the system cannot handle the error is that the cause is external limitations, such as running out of memory. A crash dump caused by an internal error can be caused by the system reaching limits in the emulator itself (like the number of atoms in the system, or too many simultaneous ETS tables). Usually the emulator or the operating system can be reconfigured to avoid the crash, which is why interpreting the crash dump correctly is important.

On systems that support OS signals, it is also possible to stop the runtime system and generate a crash dump by sending the SIGUSR1 signal.

The Erlang crash dump is a readable text file, but it can be difficult to read. Using the Crashdump Viewer tool in the Observer application simplifies the task. This is a wx-widget-based tool for browsing Erlang crash dumps.

### 1.5.1 General Information

The first part of the crash dump shows the following:

- The creation time for the dump
- A slogan indicating the reason for the dump
- The system version of the node from which the dump originates
- The number of atoms in the atom table
- The runtime system thread that caused the crash dump

#### Reasons for Crash Dumps (Slogan)

The reason for the dump is shown in the beginning of the file as:

```
Slogan: <reason>
```

If the system is halted by the BIF erlang:halt/1, the slogan is the string parameter passed to the BIF, otherwise it is a description generated by the emulator or the (Erlang) kernel. Normally the message is enough to understand the problem, but some messages are described here. Notice that the suggested reasons for the crash are only suggestions. The exact reasons for the errors can vary depending on the local applications and the underlying operating system.

- **<A>: Cannot allocate <N> bytes of memory (of type "<T>")**
  
  The system has run out of memory. `<A>` is the allocator that failed to allocate memory, `<N>` is the number of bytes that `<A>` tried to allocate, and `<T>` is the memory block type that the memory was needed for. The most common case is that a process stores huge amounts of data. In this case `<T>` is most often heap, old_heap, heap_frag, or binary. For more information on allocators, see erts_alloc(3).

- **<A>: Cannot reallocate <N> bytes of memory (of type "<T>")**
  
  Same as above except that memory was reallocated instead of allocated when the system ran out of memory.
1.5 How to Interpret the Erlang Crash Dumps

### Unexpected op code \(<N>\)

Error in compiled code, \(\text{beam}\) file damaged, or error in the compiler.

### Module \(<\text{Name}>\) undefined | Function \(<\text{Name}>\) undefined | No function \(<\text{Name}>:\<\text{Name}>/1\) | No function \(<\text{Name}>:\text{start}/2\)

The Kernel/STDLIB applications are damaged or the start script is damaged.

### Driver_select called with too large file descriptor \(N\)

The number of file descriptors for sockets exceeds 1024 (Unix only). The limit on file descriptors in some Unix flavors can be set to over 1024, but only 1024 sockets/pipes can be used simultaneously by Erlang (because of limitations in the Unix `select` call). The number of open regular files is not affected by this.

### Received SIGUSR1

Sending the `SIGUSR1` signal to an Erlang machine (Unix only) forces a crash dump. This slogan reflects that the Erlang machine crash-dumped because of receiving that signal.

### Kernel pid terminated \((<\text{Who}>)(<\text{Exit reason}>)\)

The kernel supervisor has detected a failure, usually that the `application_controller` has shut down \((\text{Who} = \text{application_controller}, \text{Why} = \text{shutdown})\). The application controller can have shut down for many reasons, the most usual is that the node name of the distributed Erlang node is already in use. A complete supervisor tree "crash" (that is, the top supervisors have exited) gives about the same result. This message comes from the Erlang code and not from the virtual machine itself. It is always because of some failure in an application, either within OTP or a "user-written" one. Looking at the error log for your application is probably the first step to take.

### Init terminating in do_boot ()

The primitive Erlang boot sequence was terminated, most probably because the boot script has errors or cannot be read. This is usually a configuration error; the system can have been started with a faulty `-boot` parameter or with a boot script from the wrong OTP version.

### Could not start kernel pid \((<\text{Who}>)()\)

One of the kernel processes could not start. This is probably because of faulty arguments (like errors in a `-config` argument) or faulty configuration files. Check that all files are in their correct location and that the configuration files (if any) are not damaged. Usually messages are also written to the controlling terminal and/or the error log explaining what is wrong.

Other errors than these can occur, as the `erlang:halt/1` BIF can generate any message. If the message is not generated by the BIF and does not occur in the list above, it can be because of an error in the emulator. There can however be unusual messages, not mentioned here, which are still connected to an application failure. There is much more information available, so a thorough reading of the crash dump can reveal the crash reason. The size of processes, the number of ETS tables, and the Erlang data on each process stack can be useful to find the problem.

### Number of Atoms

The number of atoms in the system at the time of the crash is shown as `Atoms: \(<\text{number}>\)`. Some ten thousands atoms is perfectly normal, but more can indicate that the BIF `erlang:list_to_atom/1` is used to generate many different atoms dynamically, which is never a good idea.

#### 1.5.2 Scheduler Information

Under the tag `=scheduler` is shown information about the current state and statistics of the schedulers in the runtime system. On operating systems that allow suspension of other threads, the data within this section reflects what the runtime system looks like when a crash occurs.

The following fields can exist for a process:
1.5 How to Interpret the Erlang Crash Dumps

=scheduler:id
Heading. States the scheduler identifier.

Scheduler Sleep Info Flags
If empty, the scheduler was doing some work. If not empty, the scheduler is either in some state of sleep, or suspended.

Scheduler Sleep Info Aux Work
If not empty, a scheduler internal auxiliary work is scheduled to be done.

Current Port
The port identifier of the port that is currently executed by the scheduler.

Current Process
The process identifier of the process that is currently executed by the scheduler. If there is such a process, this entry is followed by the State, Internal State, Program Counter, and CP of that same process. The entries are described in section Process Information.

Notice that this is a snapshot of what the entries are exactly when the crash dump is starting to be generated. Therefore they are most likely different (and more telling) than the entries for the same processes found in the =proc section. If there is no currently running process, only the Current Process entry is shown.

Current Process Limited Stack Trace
This entry is shown only if there is a current process. It is similar to =proc_stack, except that only the function frames are shown (that is, the stack variables are omitted). Also, only the top and bottom part of the stack are shown. If the stack is small (< 512 slots), the entire stack is shown. Otherwise the entry skipping ## slots is shown, where ## is replaced by the number of slots that has been skipped.

Run Queue
Shows statistics about how many processes and ports of different priorities are scheduled on this scheduler.

** crashed **
This entry is normally not shown. It signifies that getting the rest of the information about this scheduler failed for some reason.

1.5.3 Memory Information
Under the tag =memory is shown information similar to what can be obtained on a living node with erlang:memory().

1.5.4 Internal Table Information
Under the tags =hash_table:<table_name> and =index_table:<table_name> is shown internal tables. These are mostly of interest for runtime system developers.

1.5.5 Allocated Areas
Under the tag =allocated_areas is shown information similar to what can be obtained on a living node with erlang:system_info(allocated_areas).

1.5.6 Allocator
Under the tag =allocator:<A> is shown various information about allocator <A>. The information is similar to what can be obtained on a living node with erlang:system_info({allocator, <A>}). For more information, see also erts_alloc(3).
1.5.7 Process Information

The Erlang crashdump contains a listing of each living Erlang process in the system. The following fields can exist for a process:

=proc:<pid>

- **Heading**: States the process identifier.

- **State**
  - **Scheduled**: The process was scheduled to run but is currently not running ("in the run queue").
  - **Waiting**: The process was waiting for something (in receive).
  - **Running**: The process was currently running. If the BIF erlang:halt/1 was called, this was the process calling it.
  - **Exiting**: The process was on its way to exit.
  - **Garbing**: This is bad luck, the process was garbage collecting when the crash dump was written. The rest of the information for this process is limited.
  - **Suspended**: The process is suspended, either by the BIF erlang:suspend_process/1 or because it tries to write to a busy port.

- **Registered name**: The registered name of the process, if any.

- **Spawned as**: The entry point of the process, that is, what function was referenced in the spawn or spawn_link call that started the process.

- **Last scheduled in for | Current call**: The current function of the process. These fields do not always exist.

- **Spawned by**: The parent of the process, that is, the process that executed spawn or spawn_link.

- **Started**: The date and time when the process was started.

- **Message queue length**: The number of messages in the process' message queue.

- **Number of heap fragments**: The number of allocated heap fragments.

- **Heap fragment data**: Size of fragmented heap data, in words. This is data either created by messages sent to the process or by the Erlang BIFs. This amount depends on so many things that this field is usually uninteresting.
1.5 How to Interpret the Erlang Crash Dumps

Link list

Process IDs of processes linked to this one. Can also contain ports. If process monitoring is used, this field also tells in which direction the monitoring is in effect. That is, a link "to" a process tells you that the "current" process was monitoring the other, and a link "from" a process tells you that the other process was monitoring the current one.

Reductions

The number of reductions consumed by the process.

Stack+heap

The size of the stack and heap, in words (they share memory segment).

OldHeap

The size of the "old heap", in words. The Erlang virtual machine uses generational garbage collection with two generations. There is one heap for new data items and one for the data that has survived two garbage collections. The assumption (which is almost always correct) is that data surviving two garbage collections can be "tenured" to a heap more seldom garbage collected, as they will live for a long period. This is a usual technique in virtual machines. The sum of the heaps and stack together constitute most of the allocated memory of the process.

Heap unused, OldHeap unused

The amount of unused memory on each heap, in words. This information is usually useless.

Memory

The total memory used by this process, in bytes. This includes call stack, heap, and internal structures. Same as erlang:process_info(Pid,memory).

Program counter

The current instruction pointer. This is only of interest for runtime system developers. The function into which the program counter points is the current function of the process.

CP

The continuation pointer, that is, the return address for the current call. Usually useless for other than runtime system developers. This can be followed by the function into which the CP points, which is the function calling the current function.

Arity

The number of live argument registers. The argument registers if any are live will follow. These can contain the arguments of the function if they are not yet moved to the stack.

Internal State

A more detailed internal representation of the state of this process.

See also section Process Data.

1.5.8 Port Information

This section lists the open ports, their owners, any linked processes, and the name of their driver or external process.

1.5.9 ETS Tables

This section contains information about all the ETS tables in the system. The following fields are of interest for each table:

=ets:<owner>

Heading. States the table owner (a process identifier).
### 1.5 How to Interpret the Erlang Crash Dumps

**Table**
- The identifier for the table. If the table is a named_table, this is the name.

**Name**
- The table name, regardless of if it is a named_table or not.

**Hash table, Buckets**
- If the table is a hash table, that is, if it is not an ordered_set.

**Hash table, Chain Length**
- If the table is a hash table. Contains statistics about the table, such as the maximum, minimum, and average chain length. Having a maximum much larger than the average, and a standard deviation much larger than the expected standard deviation is a sign that the hashing of the terms behaves badly for some reason.

**Ordered set (AVL tree), Elements**
- If the table is an ordered_set. (The number of elements is the same as the number of objects in the table.)

**Fixed**
- If the table is fixed using ets:safe_fixtable/2 or some internal mechanism.

**Objects**
- The number of objects in the table.

**Words**
- The number of words allocated to data in the table.

**Type**
- The table type, that is, set, bag, duplicate_bag, or ordered_set.

**Compressed**
- If the table was compressed.

**Protection**
- The protection of the table.

**Write Concurrency**
- If write_concurrency was enabled for the table.

**Read Concurrency**
- If read_concurrency was enabled for the table.

### 1.5.10 Timers

This section contains information about all the timers started with the BIFs erlang:start_timer/3 and erlang:send_after/3. The following fields exist for each timer:

- **=timer:<owner>**
  - Heading. States the timer owner (a process identifier), that is, the process to receive the message when the timer expires.

- **Message**
  - The message to be sent.
1.5.11 Distribution Information

If the Erlang node was alive, that is, set up for communicating with other nodes, this section lists the connections that were active. The following fields can exist:

=node:<node_name>
   The node name.

no_distribution
   If the node was not distributed.

=visible_node:<channel>
   Heading for a visible node, that is, an alive node with a connection to the node that crashed. States the channel number for the node.

=hidden_node:<channel>
   Heading for a hidden node. A hidden node is the same as a visible node, except that it is started with the "-hidden" flag. States the channel number for the node.

=not_connected:<channel>
   Heading for a node that was connected to the crashed node earlier. References (that is, process or port identifiers) to the not connected node existed at the time of the crash. States the channel number for the node.

Name
   The name of the remote node.

Controller
   The port controlling communication with the remote node.

Creation
   An integer (1-3) that together with the node name identifies a specific instance of the node.

Remote monitoring: <local_proc> <remote_proc>
   The local process was monitoring the remote process at the time of the crash.

Remotely monitored by: <local_proc> <remote_proc>
   The remote process was monitoring the local process at the time of the crash.

Remote link: <local_proc> <remote_proc>
   A link existed between the local process and the remote process at the time of the crash.

1.5.12 Loaded Module Information

This section contains information about all loaded modules.
First, the memory use by the loaded code is summarized:

Current code
   Code that is the current latest version of the modules.

Old code
   Code where there exists a newer version in the system, but the old version is not yet purged.
Then, all loaded modules are listed. The following fields exist:

\[=\text{mod:<module_name>}\]

- **Heading.** States the module name.
- **Current size**
  - Memory use for the loaded code, in bytes.
- **Old size**
  - Memory use for the old code, in bytes.
- **Current attributes**
  - Module attributes for the current code. This field is decoded when looked at by the Crashdump Viewer tool.
- **Old attributes**
  - Module attributes for the old code, if any. This field is decoded when looked at by the Crashdump Viewer tool.
- **Current compilation info**
  - Compilation information (options) for the current code. This field is decoded when looked at by the Crashdump Viewer tool.
- **Old compilation info**
  - Compilation information (options) for the old code, if any. This field is decoded when looked at by the Crashdump Viewer tool.

### 1.5.13 Fun Information

This section lists all funs. The following fields exist for each fun:

\[=\text{fun}\]

- **Heading.**
- **Module**
  - The name of the module where the fun was defined.
- **Uniq, Index**
  - Identifiers.
- **Address**
  - The address of the fun’s code.
- **Refe**
  - The number of references to the fun.

### 1.5.14 Process Data

For each process there is at least one \[=\text{proc_stack}\] and one \[=\text{proc_heap}\] tag, followed by the raw memory information for the stack and heap of the process.

For each process there is also a \[=\text{proc_messages}\] tag if the process message queue is non-empty, and a \[=\text{proc_dictionary}\] tag if the process dictionary (the `put/2` and `get/1` thing) is non-empty.

The raw memory information can be decoded by the Crashdump Viewer tool. You can then see the stack dump, the message queue (if any), and the dictionary (if any).
1.6 How to Implement an Alternative Carrier for the Erlang Distribution

The stack dump is a dump of the Erlang process stack. Most of the live data (that is, variables currently in use) are placed on the stack; thus this can be interesting. One has to “guess” what is what, but as the information is symbolic, thorough reading of this information can be useful. As an example, we can find the state variable of the Erlang primitive loader online (5) and (6) in the following example:

When interpreting the data for a process, it is helpful to know that anonymous function objects (funs) are given the following:

- A name constructed from the name of the function in which they are created
- A number (starting with 0) indicating the number of that fun within that function

1.5.15 Atoms

This section presents all the atoms in the system. This is only of interest if one suspects that dynamic generation of atoms can be a problem, otherwise this section can be ignored.

Notice that the last created atom is shown first.

1.5.16 Disclaimer

The format of the crash dump evolves between OTP releases. Some information described here may not apply to your version. A description like this will never be complete; it is meant as an explanation of the crash dump in general and as a help when trying to find application errors, not as a complete specification.

1.6 How to Implement an Alternative Carrier for the Erlang Distribution

This section describes how to implement an alternative carrier protocol for the Erlang distribution. The distribution is normally carried by TCP/IP. Here is explained a method for replacing TCP/IP with another protocol.

The section is a step-by-step explanation of the uds_dist example application (in the Kernel application examples directory). The uds_dist application implements distribution over Unix domain sockets and is written for the Sun Solaris 2 operating environment. The mechanisms are however general and apply to any operating system Erlang runs on. The reason the C code is not made portable, is simply readability.

1.6.1 Introduction

To implement a new carrier for the Erlang distribution, the main steps are as follows.
1.6 How to Implement an Alternative Carrier for the Erlang Distribution

Note:
As of ERTS version 10.0 support for distribution controller processes has been introduced. That is, the traffic over a distribution channel can be managed by a process instead of only by a port. This makes it possible to implement large parts of the logic in Erlang code, and you perhaps do not even need a new driver for the protocol. One example could be Erlang distribution over UDP using gen_udp (your Erlang code will of course have to take care of retransmissions, etc in this example). That is, depending on what you want to do you perhaps do not need to implement a driver at all and can then skip the driver related sections below. The gen_tcp_dist and erl_uds_dist examples described in the Distribution Module section utilize distribution controller processes and can be worth having a look at if you want to use distribution controller processes.

Writing an Erlang Driver
First, the protocol must be available to the Erlang machine, which involves writing an Erlang driver. A port program cannot be used, an Erlang driver is required. Erlang drivers can be:

- Statically linked to the emulator, which can be an alternative when using the open source distribution of Erlang, or
- Dynamically loaded into the Erlang machines address space, which is the only alternative if a precompiled version of Erlang is to be used

Writing an Erlang driver is not easy. The driver is written as some callback functions called by the Erlang emulator when data is sent to the driver, or the driver has any data available on a file descriptor. As the driver callback routines execute in the main thread of the Erlang machine, the callback functions can perform no blocking activity whatsoever. The callbacks are only to set up file descriptors for waiting and/or read/write available data. All I/O must be non-blocking. Driver callbacks are however executed in sequence, why a global state can safely be updated within the routines.

Writing an Erlang Interface for the Driver
When the driver is implemented, one would preferably write an Erlang interface for the driver to be able to test the functionality of the driver separately. This interface can then be used by the distribution module, which will cover the details of the protocol from the net_kernel.

The easiest path is to mimic the inet and inet_tcp interfaces, but not much functionality in those modules needs to be implemented. In the example application, only a few of the usual interfaces are implemented, and they are much simplified.

Writing a Distribution Module
When the protocol is available to Erlang through a driver and an Erlang interface module, a distribution module can be written. The distribution module is a module with well-defined callbacks, much like a gen_server (there is no compiler support for checking the callbacks, though). This module implements:

- The details of finding other nodes (that is, talking to epmd or something similar)
- Creating a listen port (or similar)
- Connecting to other nodes
- Performing the handshakes/cookie verification

There is however a utility module, dist_util, which does most of the hard work of handling handshakes, cookies, timers, and ticking. Using dist_util makes implementing a distribution module much easier and that is done in the example application.
Creating Boot Scripts

The last step is to create boot scripts to make the protocol implementation available at boot time. The implementation can be debugged by starting the distribution when all the system is running, but in a real system the distribution is to start very early, why a boot script and some command-line parameters are necessary.

This step also implies that the Erlang code in the interface and distribution modules is written in such a way that it can be run in the startup phase. In particular, there can be no calls to the application module or to any modules not loaded at boot time. That is, only Kernel, STDLIB, and the application itself can be used.

1.6.2 Distribution Module

The distribution module exposes an API that net_kernel calls in order to manage connections to other nodes. The module name should have the suffix _dist.

The module needs to create some kind of listening entity (process or port) and an acceptor process that accepts incoming connections using the listening entity. For each connection, the module at least needs to create one connection supervisor process, which also is responsible for the handshake when setting up the connection, and a distribution controller (process or port) responsible for transport of data over the connection. The distribution controller and the connection supervisor process should be linked together so both of them are cleaned up when the connection is taken down.

Note that there need to be exactly one distribution controller per connection. A process or port can only be distribution controller for one connection. The registration as distribution controller cannot be undone. It will stick until the distribution controller terminates. The distribution controller should not ignore exit signals. It is allowed to trap exits, but it should then voluntarily terminate when an exit signal is received.

An example implementation of a distribution module can be found in $ERL_TOP/lib/kernel/examples/gen_tcp_dist/src/gen_tcp_dist.erl. It implements the distribution over TCP/IP using the gen_tcp API with distribution controllers implemented by processes. This instead of using port distribution controllers as the ordinary TCP/IP distribution uses.

Another example implementation of a distribution module can be found in $ERL_TOP/lib/kernel/examples/erl_uds_dist/src/erl_uds_dist.erl. It implements the distribution over Unix domain sockets using the gen_tcp API with distribution controllers implemented by processes. Compared to the original uds_dist example using a port driver written in C, erl_uds_dist is written entirely in Erlang.

Exported Callback Functions

The following functions are mandatory:

```erlang
listen(Name) ->
  {ok, {Listen, Address, Creation}} | {error, Error}
listen(Name, Host) ->
  {ok, {Listen, Address, Creation}} | {error, Error}
```

`listen/2` is called once in order to listen for incoming connection requests. The call is made when the distribution is brought up. The argument `Name` is the part of the node name before the @ sign in the full node name. It can be either an atom or a string. The argument `Host` is the part of the node name after the @ sign in the full node name. It is always a string.

The return value consists of a `Listen` handle (which is later passed to the `accept/1` callback), `Address` which is a `#net_address{}` record with information about the address for the node (the `#net_address{}` record is defined in `kernel/include/net_address.hrl`), and `Creation` which (currently) is an integer 1, 2, or 3.

If `epmd` is to be used for node discovery, you typically want to use the `erl_epmd` module (part of the `kernel` application) in order to register the listen port with `epmd` and retrieve `Creation` to use.

```erlang
address() ->
```
1.6 How to Implement an Alternative Carrier for the Erlang Distribution

Address

address/0 is called in order to get the Address part of the listen/2 function without creating a listen socket. All fields except address have to be set in the returned record.

Example:

```erlang
address() ->
  {ok, Host} = inet:gethostname(),
  #net_address{ host = Host, protocol = tcp, family = inet6 }.
```

accept (Listen) ->

 accept/1 should spawn a process that accepts connections. This process should preferably execute on max priority. The process identifier of this process should be returned.

The Listen argument will be the same as the Listen handle part of the return value of the listen/1 callback above. accept/1 is called only once when the distribution protocol is started.

The caller of this function is a representative for net_kernel (this may or may not be the process registered as net_kernel) and is in this document identified as Kernel. When a connection has been accepted by the acceptor process, it needs to inform Kernel about the accepted connection. This is done by passing a message on the form:

```erlang
Kernel ! {accept, AcceptorPid, DistController, Family, Proto}
```

DistController is either the process or port identifier of the distribution controller for the connection. The distribution controller should be created by the acceptor processes when a new connection is accepted. Its job is to dispatch traffic on the connection.

Kernel responds with one of the following messages:

```erlang
{Kernel, controller, SupervisorPid}
```

The request was accepted and SupervisorPid is the process identifier of the connection supervisor process (which is created in the accept_connection/5 callback).

```erlang
{Kernel, unsupported_protocol}
```

The request was rejected. This is a fatal error. The acceptor process should terminate.

When an accept sequence has been completed the acceptor process is expected to continue accepting further requests.

accept_connection(AcceptorPid, DistCtrl, MyNode, Allowed, SetupTime) ->

 accept_connection/5 should spawn a process that will perform the Erlang distribution handshake for the connection. If the handshake successfully completes it should continue to function as a connection supervisor. This process should preferably execute on max priority and should be linked to the caller. The dist_util:net_ticker_spawn_options() function can be called to get spawn options suitable for this process which can be passed directly to erlang:spawn_opt/4. dist_util:net_ticker_spawn_options() will by default return [link, {priority, max}], but allows the user to configure more options using the net_ticker_spawn_options kernel parameter. The process identifier of this process should be returned.

The arguments:

```erlang
AcceptorPid
```

Process identifier of the process created by the accept/1 callback.
1.6 How to Implement an Alternative Carrier for the Erlang Distribution

DistCtrl
The identifier of the distribution controller identifier created by the acceptor process. To be passed along to
dist_util:handshake_other_started(HsData).

MyNode
Node name of this node. To be passed along to
dist_util:handshake_other_started(HsData).

Allowed
To be passed along to dist_util:handshake_other_started(HsData).

SetupTime
Time used for creating a setup timer by a call to dist_util:start_timer(SetupTime). The timer
should be passed along to dist_util:handshake_other_started(HsData).

The created process should provide callbacks and other information needed for the handshake in a #hs_data{}
record and call dist_util:handshake_other_started(HsData) with this record.

dist_util:handshake_other_started(HsData) will perform the handshake and if the handshake
successfully completes this process will then continue in a connection supervisor loop as long as the connection
is up.

setup(Node, Type, MyNode, LongOrShortNames, SetupTime) ->
ConnectionSupervisorPid
setup/5 should spawn a process that connects to Node. When connection has been established it should
perform the Erlang distribution handshake for the connection. If the handshake successfully completes it should
continue to function as a connection supervisor. This process should preferably execute on max priority and
should be linked to the caller. The dist_util:net_ticker_spawn_options() function can be called
to get spawn options suitable for this process which can be passed directly to erlang:spawn_opt/4.
dist_util:net_ticker_spawn_options() will by default return [link, {priority, max}],
but allows the user to configure more options using the net_ticker_spawn_options kernel parameter.
The process identifier of this process should be returned.

The arguments:
Node
Node name of remote node. To be passed along to dist_util:handshake_we_started(HsData).

Type
Connection type. To be passed along to dist_util:handshake_we_started(HsData).

MyNode
Node name of this node. To be passed along to dist_util:handshake_we_started(HsData).

LongOrShortNames
Either the atom longnames or the atom shortnames indicating whether long or short names is used.

SetupTime
Time used for creating a setup timer by a call to dist_util:start_timer(SetupTime). The timer
should be passed along to dist_util:handshake_we_started(HsData).

The caller of this function is a representative for net_kernel (this may or may not be the process registered
as net_kernel) and is in this document identified as Kernel.

This function should, besides spawning the connection supervisor, also create a distribution controller. The
distribution controller is either a process or a port which is responsible for dispatching traffic.
1.6 How to Implement an Alternative Carrier for the Erlang Distribution

The created process should provide callbacks and other information needed for the handshake in a #hs_data() record and call dist_util:handshake_we_started(HsData) with this record.

dist_util:handshake_we_started(HsData) will perform the handshake and the handshake successfully completes this process will then continue in a connection supervisor loop as long as the connection is up.

close(Listen) ->
    void()

Called in order to close the Listen handle that originally was passed from the listen/1 callback.

select(NodeName) ->
    boolean()

Return true if the host name part of the NodeName is valid for use with this protocol; otherwise, false.

There are also two optional functions that may be exported:

setopts(Listen, Opts) ->
    ok | {error, Error}

The argument Listen is the handle originally passed from the listen/1 callback. The argument Opts is a list of options to set on future connections.

getopts(Listen, Opts) ->
    {ok, OptionValues} | {error, Error}

The argument Listen is the handle originally passed from the listen/1 callback. The argument Opts is a list of options to read for future connections.

The #hs_data{} Record

The dist_util:handshake_we_started/1 and dist_util:handshake_other_started/1 functions takes a #hs_data() record as argument. There are quite a lot of fields in this record that you need to set. The record is defined in kernel/include/dist_util.hrl. Not documented fields should not be set, i.e., should be left as undefined.

The following #hs_data{} record fields need to be set unless otherwise stated:

kernel_pid
    Process identifier of the Kernel process. That is, the process that called either setup/5 or accept_connection/5.

other_node
    Name of the other node. This field is only mandatory when this node initiates the connection. That is, when connection is set up via setup/5.

this_node
    The node name of this node.

socket
    The identifier of the distribution controller.

timer
    The timer created using dist_util:start_timer/1.

allowed
    Information passed as Allowed to accept_connection/5. This field is only mandatory when the remote node initiated the connection. That is, when the connection is set up via accept_connection/5.
1.6 How to Implement an Alternative Carrier for the Erlang Distribution

f_send
A fun with the following signature:

\[
\text{fun (DistCtrlr, Data) -> ok | {error, Error}}
\]

where DistCtrlr is the identifier of the distribution controller and Data is io data to pass to the other side.
Only used during handshake phase.

f_recv
A fun with the following signature:

\[
\text{fun (DistCtrlr, Length) -> {ok, Packet} | {error, Reason}}
\]

where DistCtrlr is the identifier of the distribution controller. If Length is 0, all available bytes should be returned. If Length > 0, exactly Length bytes should be returned, or an error; possibly discarding less than Length bytes of data when the connection is closed from the other side. It is used for passive receive of data from the other end.
Only used during handshake phase.

f_setopts_pre_nodeup
A fun with the following signature:

\[
\text{fun (DistCtrlr) -> ok | {error, Error}}
\]

where DistCtrlr is the identifier of the distribution controller. Called just before the distribution channel is taken up for normal traffic.
Only used during handshake phase.

f_setopts_post_nodeup
A fun with the following signature:

\[
\text{fun (DistCtrlr) -> ok | {error, Error}}
\]

where DistCtrlr is the identifier of the distribution controller. Called just after distribution channel has been taken up for normal traffic.
Only used during handshake phase.

f_getll
A fun with the following signature:

\[
\text{fun (DistCtrlr) -> ID}
\]

where DistCtrlr is the identifier of the distribution controller and ID is the identifier of the low level entity that handles the connection (often DistCtrlr itself).
Only used during handshake phase.

f_address
A fun with the following signature:

\[
\text{fun (DistCtrlr, Node) -> NetAddress}
\]

where DistCtrlr is the identifier of the distribution controller, Node is the node name of the node on the other end, and NetAddress is a #net_address{} record with information about the address for the
1.6 How to Implement an Alternative Carrier for the Erlang Distribution

Node on the other end of the connection. The #net_address{} record is defined in kernel/include/net_address.hrl.

Only used during handshake phase.

mf_tick

A fun with the following signature:

```
fun (DistCtrlr) -> void()
```

where DistCtrlr is the identifier of the distribution controller. This function should send information over the connection that is not interpreted by the other end while increasing the statistics of received packets on the other end. This is usually implemented by sending an empty packet.

**Note:**

It is of vital importance that this operation does not block the caller for a long time. This since it is called from the connection supervisor.

Used when connection is up.

mf_getstat

A fun with the following signature:

```
fun (DistCtrlr) -> {ok, Received, Sent, PendSend}
```

where DistCtrlr is the identifier of the distribution controller, Received is received packets, Sent is sent packets, and PendSend is amount of data in queue to be sent (typically in bytes, but dist_util only checks whether the value is non-zero to know there is data in queue) or a boolean() indicating whether there are packets in queue to be sent.

**Note:**

It is of vital importance that this operation does not block the caller for a long time. This since it is called from the connection supervisor.

Used when connection is up.

request_type

The request Type as passed to setup/5. This is only mandatory when the connection has been initiated by this node. That is, the connection is set up via setup/5.

mf_setopts

A fun with the following signature:

```
fun (DistCtrl, Opts) -> ok | {error, Error}
```

where DistCtrl is the identifier of the distribution controller and Opts is a list of options to set on the connection.

This function is optional. Used when connection is up.

mf_getopts

A fun with the following signature:

```
fun (DistCtrl, Opts) -> {ok, OptionValues} | {error, Error}
```
1.6 How to Implement an Alternative Carrier for the Erlang Distribution

where DistCtrlr is the identifier of the distribution controller and Opts is a list of options to read for the connection. This function is optional. Used when connection is up.

f_handshake_complete

A fun with the following signature:

```erlang
fun (DistCtrlr, Node, DHandle) -> void()
```

where DistCtrlr is the identifier of the distribution controller, Node is the node name of the node connected at the other end, and DHandle is a distribution handle needed by a distribution controller process when calling the following BIFs:

- `erlang:dist_ctrl_get_data/1`
- `erlang:dist_ctrl_get_data_notification/1`
- `erlang:dist_ctrl_input_handler/2`
- `erlang:dist_ctrl_put_data/2`

This function is called when the handshake has completed and the distribution channel is up. The distribution controller can begin dispatching traffic over the channel. This function is optional. Only used during handshake phase.

add_flags

Distribution flags to add to the connection. Currently all (non obsolete) flags will automatically be enabled. This flag field is optional.

reject_flags

Distribution flags to reject. Currently the following distribution flags can be rejected:

- `DFLAG_DIST_HDR_ATOM_CACHE`
  Do not use atom cache over this connection.
- `DFLAG_FRAGMENTS`
  Split large distribution messages into multiple fragments.

This flag field is optional.

See also Distribution Data Delivery

require_flags

Require these distribution flags to be used. The connection will be aborted during the handshake if the other end does not use them.

This flag field is optional.

Distribution Data Delivery

When using the default configuration, the data to pass over a connection needs to be delivered as is to the node on the receiving end in the **exact same order**, with no loss of data whatsoever, as sent from the sending node.

The data delivery order can be relaxed by disabling features that require strict ordering. This is done by passing the distribution flags returned by `dist_util:strict_order_flags/0` in the `reject_flags` field of the `#hs_data()` record used when setting up the connection. When relaxed ordering is used, only the order of signals with the same sender/receiver pair has to be preserved. However, note that disabling the features that require strict ordering may have a negative impact on performance, throughput, and/or latency.
Enable Your Distribution Module

For `net_kernel` to find out which distribution module to use, the `erl` command-line argument `-proto_dist` is used. It is followed by one or more distribution module names, with suffix "_dist" removed. That is, `gen_tcp_dist` as a distribution module is specified as `-proto_dist gen_tcp`.

If no `epmd` (TCP port mapper daemon) is used, also command-line option `-no_epmd` is to be specified, which makes Erlang skip the `epmd` startup, both as an OS process and as an Erlang ditto.

1.6.3 The Driver

Note:

This section was written a long time ago. Most of it is still valid, but some things have changed since then. Some updates have been made to the documentation of the driver presented here, but more can be done and is planned for the future. The reader is encouraged to read the `erl_driver` and `driver_entry` documentation also.

Although Erlang drivers in general can be beyond the scope of this section, a brief introduction seems to be in place.

Drivers in General

An Erlang driver is a native code module written in C (or assembler), which serves as an interface for some special operating system service. This is a general mechanism that is used throughout the Erlang emulator for all kinds of I/O. An Erlang driver can be dynamically linked (or loaded) to the Erlang emulator at runtime by using the `erl_ddll` Erlang module. Some of the drivers in OTP are however statically linked to the runtime system, but that is more an optimization than a necessity.

The driver data types and the functions available to the driver writer are defined in header file `erl_driver.h` seated in Erlang’s include directory. See the `erl_driver` documentation for details of which functions are available.

When writing a driver to make a communications protocol available to Erlang, one should know just about everything worth knowing about that particular protocol. All operation must be non-blocking and all possible situations are to be accounted for in the driver. A non-stable driver will affect and/or crash the whole Erlang runtime system.

The emulator calls the driver in the following situations:

- When the driver is loaded. This callback must have a special name and inform the emulator of what callbacks are to be used by returning a pointer to a `ErlDrvEntry` struct, which is to be properly filled in (see below).
- When a port to the driver is opened (by a `open_port` call from Erlang). This routine is to set up internal data structures and return an opaque data entity of the type `ErlDrvData`, which is a data type large enough to hold a pointer. The pointer returned by this function is the first argument to all other callbacks concerning this particular port. It is usually called the port handle. The emulator only stores the handle and does never try to interpret it, why it can be virtually anything (anything not larger than a pointer that is) and can point to anything if it is a pointer. Usually this pointer refers to a structure holding information about the particular port, as it does in the example.
- When an Erlang process sends data to the port. The data arrives as a buffer of bytes, the interpretation is not defined, but is up to the implementor. This callback returns nothing to the caller, answers are sent to the caller as messages (using a routine called `driver_output` available to all drivers). There is also a way to talk in a synchronous way to drivers, described below. There can be an additional callback function for handling data that is fragmented (sent in a deep io-list). That interface gets the data in a form suitable for Unix `writev` rather than in a single buffer. There is no need for a distribution driver to implement such a callback, so we will not.
- When a file descriptor is signaled for input. This callback is called when the emulator detects input on a file descriptor that the driver has marked for monitoring by using the interface `driver_select`. The mechanism of driver select makes it possible to read non-blocking from file descriptors by calling `driver_select` when reading is needed, and then do the reading in this callback (when reading is possible). The typical scenario is...
that \texttt{driver\_select} is called when an Erlang process orders a read operation, and that this routine sends the answer when data is available on the file descriptor.

- When a file descriptor is signaled for output. This callback is called in a similar way as the previous, but when writing to a file descriptor is possible. The usual scenario is that Erlang orders writing on a file descriptor and that the driver calls \texttt{driver\_select}. When the descriptor is ready for output, this callback is called and the driver can try to send the output. Queuing can be involved in such operations, and there are convenient queue routines available to the driver writer to use.

- When a port is closed, either by an Erlang process or by the driver calling one of the \texttt{driver\_failure\_XXX} routines. This routine is to clean up everything connected to one particular port. When other callbacks call a \texttt{driver\_failure\_XXX} routine, this routine is immediately called. The callback routine issuing the error can make no more use of the data structures for the port, as this routine surely has freed all associated data and closed all file descriptors. If the queue utility available to driver writer is used, this routine is however not called until the queue is empty.

- When an Erlang process calls \texttt{erlang:port\_control/3}, which is a synchronous interface to drivers. The control interface is used to set driver options, change states of ports, and so on. This interface is used a lot in the example.

- When a timer expires. The driver can set timers with the function \texttt{driver\_set\_timer}. When such timers expire, a specific callback function is called. No timers are used in the example.

- When the whole driver is unloaded. Every resource allocated by the driver is to be freed.

**The Data Structures of the Distribution Driver**

The driver used for Erlang distribution is to implement a reliable, order maintaining, variable length packet-oriented protocol. All error correction, resending and such need to be implemented in the driver or by the underlying communications protocol. If the protocol is stream-oriented (as is the case with both TCP/IP and our streamed Unix domain sockets), some mechanism for packaging is needed. We will use the simple method of having a header of four bytes containing the length of the package in a big-endian 32-bit integer. As Unix domain sockets only can be used between processes on the same machine, we do not need to code the integer in some special endianness, but we will do it anyway because in most situation you need to do it. Unix domain sockets are reliable and order maintaining, so we do not need to implement resends and such in the driver.

We start writing the example Unix domain sockets driver by declaring prototypes and filling in a static \texttt{ErlDrvEntry} structure:
On line 1-10 the OS headers needed for the driver are included. As this driver is written for Solaris, we know that the header `stdio.h` exists. So the preprocessor variable `HAVE_UIO_H` can be defined before `erl_driver.h` is included on line 12. The definition of `HAVE_UIO_H` will make the I/O vectors used in Erlang's driver queues to correspond to the operating systems ditto, which is very convenient.

On line 16-23 the different callback functions are declared ("forward declarations").
The driver structure is similar for statically linked-in drivers and dynamically loaded. However, some of the fields are to be left empty (that is, initialized to NULL) in the different types of drivers. The first field (the init function pointer) is always left blank in a dynamically loaded driver, see line 26. NULL on line 37 is always to be there, the field is no longer used and is retained for backward compatibility. No timers are used in this driver, why no callback for timers is needed. The outputv field (line 40) can be used to implement an interface similar to Unix writev for output. The Erlang runtime system could previously not use outputv for the distribution, but it can as from ERTS 5.7.2. As this driver was written before ERTS 5.7.2 it does not use the outputv callback. Using the outputv callback is preferred, as it reduces copying of data. (We will however use scatter/gather I/O internally in the driver.)

As from ERTS 5.5.3 the driver interface was extended with version control and the possibility to pass capability information. Capability flags are present on line 48. As from ERTS 5.7.4 flag ERL_DRV_FLAG_SOFT_BUSY is required for drivers that are to be used by the distribution. The soft busy flag implies that the driver can handle calls to the output and outputv callbacks although it has marked itself as busy. This has always been a requirement on drivers used by the distribution, but no capability information has been available about this previously. For more information, see erl_driver:set_busy_port()).

This driver was written before the runtime system had SMP support. The driver will still function in the runtime system with SMP support, but performance will suffer from lock contention on the driver lock used for the driver. This can be alleviated by reviewing and perhaps rewriting the code so that each instance of the driver safely can execute in parallel. When instances safely can execute in parallel, it is safe to enable instance-specific locking on the driver. This is done by passing ERL_DRV_FLAG_USE_PORT_LOCKING as a driver flag. This is left as an exercise for the reader.

Thus, the defined callbacks are as follows:

uds_start
Must initiate data for a port. We do not create any sockets here, only initialize data structures.

uds_stop
Called when a port is closed.

uds_command
Handles messages from Erlang. The messages can either be plain data to be sent or more subtle instructions to the driver. This function is here mostly for data pumping.

uds_input
Called when there is something to read from a socket.

uds_output
Called when it is possible to write to a socket.

uds_finish
Called when the driver is unloaded. A distribution driver will never be unloaded, but we include this for completeness. To be able to clean up after oneself is always a good thing.

uds_control
The erlang:port_control/3 callback, which is used a lot in this implementation.

The ports implemented by this driver operate in two major modes, named command and data. In command mode, only passive reading and writing (like gen_tcp:recv/gen_tcp:send) can be done. The port is in this mode during the distribution handshake. When the connection is up, the port is switched to data mode and all data is immediately read and passed further to the Erlang emulator. In data mode, no data arriving to uds_command is interpreted, only packaged and sent out on the socket. The uds_control callback does the switching between those two modes.
1.6 How to Implement an Alternative Carrier for the Erlang Distribution

While `net_kernel` informs different subsystems that the connection is coming up, the port is to accept data to send. However, the port should not receive any data, to avoid that data arrives from another node before every kernel subsystem is prepared to handle it. A third mode, named `intermediate`, is used for this intermediate stage.

An enum is defined for the different types of ports:

```c
typedef enum {
  portTypeUnknown,      /* An uninitialized port */
  portTypeListener,     /* A listening port/socket */
  portTypeAcceptor,     /* An intermediate stage when accepting
                         on a listen port */
  portTypeConnector,    /* An intermediate stage when connecting */
  portTypeCommand,      /* A connected open port in command mode */
  portTypeIntermediate, /* A connected open port in special half active mode */
  portTypeData          /* A connected open port in data mode */
} PortType;
```

The different types are as follows:

- **portTypeUnknown**
  
  The type a port has when it is opened, but not bound to any file descriptor.

- **portTypeListener**
  
  A port that is connected to a listen socket. This port does not do much, no data pumping is done on this socket, but read data is available when one is trying to do an accept on the port.

- **portTypeAcceptor**
  
  This port is to represent the result of an accept operation. It is created when one wants to accept from a listen socket, and it is converted to a `portTypeCommand` when the accept succeeds.

- **portTypeConnector**
  
  Very similar to `portTypeAcceptor`, an intermediate stage between the request for a connect operation and that the socket is connected to an accepting ditto in the other end. When the sockets are connected, the port switches type to `portTypeCommand`.

- **portTypeCommand**
  
  A connected socket (or accepted socket) in command mode mentioned earlier.

- **portTypeIntermediate**
  
  The intermediate stage for a connected socket. There is to be no processing of input for this socket.

- **portTypeData**
  
  The mode where data is pumped through the port and the `uds_command` routine regards every call as a call where sending is wanted. In this mode, all input available is read and sent to Erlang when it arrives on the socket, much like in the active mode of a `gen_tcp` socket.

We study the state that is needed for the ports. Notice that not all fields are used for all types of ports. Some space could be saved by using unions, but that would clutter the code with multiple indirections, so here is used one struct for all types of ports, for readability:
1.6 How to Implement an Alternative Carrier for the Erlang Distribution

```c
(1) typedef unsigned char Byte;
(2) typedef unsigned int Word;
(3) typedef struct uds_data {
(4)     int fd; /* File descriptor */
(5)     ErlDrvPort port; /* The port identifier */
(6)     int lockfd; /* The file descriptor for a lock file in case of listen sockets */
(7)     Byte creation; /* The creation serial derived from the lock file */
(8)     PortType type; /* Type of port */
(9)     char *name; /* Short name of socket for unlink */
(10)    Word sent; /* Bytes sent */
(11)    Word received; /* Bytes received */
(12)    struct uds_data *partner; /* The partner in an accept/listen pair */
(13)    struct uds_data *next; /* Next structure in list */
(14)    int buffer_size; /* The allocated size of the input buffer */
(15)    int buffer_pos; /* Current position in input buffer */
(16)    int header_pos; /* Where the current header is in the input buffer */
(17)    Byte *buffer; /* The actual input buffer */
(18) } UdsData;
```

This structure is used for all types of ports although some fields are useless for some types. The least memory consuming solution would be to arrange this structure as a union of structures. However, the multiple indirections in the code to access a field in such a structure would clutter the code too much for an example.

The fields in the structure are as follows:

- **fd**
  - The file descriptor of the socket associated with the port.

- **port**
  - The port identifier for the port that this structure corresponds to. It is needed for most driver_XXX calls from the driver back to the emulator.

- **lockfd**
  - If the socket is a listen socket, we use a separate (regular) file for two purposes:
    - We want a locking mechanism that gives no race conditions, to be sure if another Erlang node uses the listen socket name we require or if the file is only left there from a previous (crashed) session.
    - We store the creation serial number in the file. The creation is a number that is to change between different instances of different Erlang emulators with the same name, so that process identifiers from one emulator do not become valid when sent to a new emulator with the same distribution name. The creation can be from 0 through 3 (two bits) and is stored in every process identifier sent to another node.
  
  In a system with TCP-based distribution, this data is kept in the Erlang port mapper daemon (epmd), which is contacted when a distributed node starts. The lock file and a convention for the UDS listen socket's name remove the need for epmd when using this distribution module. UDS is always restricted to one host, why avoiding a port mapper is easy.

- **creation**
  - The creation number for a listen socket, which is calculated as (the value found in the lock-file + 1) rem 4. This creation value is also written back into the lock file, so that the next invocation of the emulator finds our value in the file.

- **type**
  - The current type/state of the port, which can be one of the values declared above.
name

The name of the socket file (the path prefix removed), which allows for deletion (unlink) when the socket is closed.

sent

How many bytes that have been sent over the socket. This can wrap, but that is no problem for the distribution, as the Erlang distribution is only interested in if this value has changed. (The Erlang net_kernel ticker uses this value by calling the driver to fetch it, which is done through the erlang:port_control/3 routine.)

received

How many bytes that are read (received) from the socket, used in similar ways as sent.

partner

A pointer to another port structure, which is either the listen port from which this port is accepting a connection or conversely. The “partner relation” is always bidirectional.

next

Pointer to next structure in a linked list of all port structures. This list is used when accepting connections and when the driver is unloaded.

buffer_size, buffer_pos, header_pos, buffer

Data for input buffering. For details about the input buffering, see the source code in directory kernel/examples. That certainly goes beyond the scope of this section.

Selected Parts of the Distribution Driver Implementation

The implementation of the distribution driver is not completely covered here, details about buffering and other things unrelated to driver writing are not explained. Likewise are some peculiarities of the UDS protocol not explained in detail. The chosen protocol is not important.

Prototypes for the driver callback routines can be found in the erl_driver.h header file.

The driver initialization routine is (usually) declared with a macro to make the driver easier to port between different operating systems (and flavors of systems). This is the only routine that must have a well-defined name. All other callbacks are reached through the driver structure. The macro to use is named DRIVER_INIT and takes the driver name as parameter:

```
(1) /* Beginning of linked list of ports */
(2) static UdsData *first_data;
(3) DRIVER_INIT(uds_drv)
(4) {
(5)     first_data = NULL;
(6)     return &uds_driver_entry;
(7) }
```

The routine initializes the single global data structure and returns a pointer to the driver entry. The routine is called when erl_ddll:load_driver is called from Erlang.

The uds_start routine is called when a port is opened from Erlang. In this case, we only allocate a structure and initialize it. Creating the actual socket is left to the uds_command routine.
1.6 How to Implement an Alternative Carrier for the Erlang Distribution

(1) static ErlDrvData uds_start(ErlDrvPort port, char *buf) {
(2)     UdsData *ud;
(3)     ud = ALLOC(sizeof(UdsData));
(4)     ud->fd = -1;
(5)     ud->lockfd = -1;
(6)     ud->creation = 0;
(7)     ud->port = port;
(8)     ud->type = portTypeUnknown;
(9)     ud->name = NULL;
(10)    ud->buffer_size = 0;
(11)    ud->buffer_pos = 0;
(12)    ud->header_pos = 0;
(13)    ud->buffer = NULL;
(14)    ud->sent = 0;
(15)    ud->received = 0;
(16)    ud->partner = NULL;
(17)    ud->next = first_data;
(18)    first_data = ud;
(19)    return((ErlDrvData) ud);
(20) }

Every data item is initialized, so that no problems arise when a newly created port is closed (without there being any corresponding socket). This routine is called when open_port({'spawn', "uds_drv"}, []) is called from Erlang.

The uds_command routine is the routine called when an Erlang process sends data to the port. This routine handles all asynchronous commands when the port is in command mode and the sending of all data when the port is in data mode:
The command routine takes three parameters; the handle returned for the port by `uds_start`, which is a pointer to
the internal port structure, the data buffer, and the length of the data buffer. The buffer is the data sent from Erlang
(a list of bytes) converted to an C array (of bytes).

If Erlang sends, for example, the list `[a, b, c]` to the port, the `bufflen` variable is 3 and the `buff` variable
contains `{'a', 'b', 'c'}` (no NULL termination). Usually the first byte is used as an opcode, which is the case in
this driver too (at least when the port is in `command` mode). The opcodes are defined as follows:

'L'<socket name>

Creates and listens on socket with the specified name.
1.6 How to Implement an Alternative Carrier for the Erlang Distribution

'A'<listen number as 32-bit big-endian>
Accepts from the listen socket identified by the specified identification number. The identification number is retrieved with the uds_control routine.

'C'<socket name>
Connects to the socket named <socket name>.

'S'<data>
Sends the data <data> on the connected/accepted socket (in command mode). The sending is acknowledged when the data has left this process.

'R'
Receives one packet of data.

"One packet of data" in command 'R' can be explained as follows. This driver always sends data packaged with a 4 byte header containing a big-endian 32-bit integer that represents the length of the data in the packet. There is no need for different packet sizes or some kind of streamed mode, as this driver is for the distribution only. Why is the header word coded explicitly in big-endian when a UDS socket is local to the host? It is good practice when writing a distribution driver, as distribution in practice usually crosses the host boundaries.

On line 4-8 is handled the case where the port is in data mode or intermediate mode and the remaining routine handles the different commands. The routine uses the driver_failure_posix() routine to report errors (see, for example, line 15). Notice that the failure routines make a call to the uds_stop routine, which will remove the internal port data. The handle (and the casted handle ud) is therefore invalid pointers after a driver_failure call and we should return immediately. The runtime system will send exit signals to all linked processes.

The uds_input routine is called when data is available on a file descriptor previously passed to the driver_select routine. This occurs typically when a read command is issued and no data is available. The do_recv routine is as follows:

```c
(1) static void do_recv(UdsData *ud)
(2) {
(3)     int res;
(4)     char *ibuf;
(5)     for(;;) {
(6)         if ((res = buffered_read_package(ud,&ibuf)) < 0) {
(7)             if (res == NORMAL_READ_FAILURE) {
(8)                 driver_select(ud->port, (ErlDrvEvent) ud->fd, DO_READ, 1);
(9)             } else {
(10)                driver_failure_eof(ud->port);
(11)            }
(12)            return;
(13)        }
(14) /* Got a package */
(15)     if (ud->type == portTypeCommand) {
(16)         ibuf[-1] = 'R'; /* There is always room for a single byte
(17)                 opcode before the actual buffer
(18)                 (where the packet header was) */
(19)         driver_output(ud->port,ibuf - 1, res + 1);
(20)         driver_select(ud->port, (ErlDrvEvent) ud->fd, DO_READ,0);
(21)         return;
(22)     } else {
(23)         ibuf[-1] = DIST_MAGIC_RECV_TAG; /* XXX */
(24)         driver_output(ud->port,ibuf - 1, res + 1);
(25)         driver_select(ud->port, (ErlDrvEvent) ud->fd, DO_READ,1);
(26)     }
(27) }
(28) }
```

46 | Ericsson AB. All Rights Reserved.: Erlang Run-Time System Application (ERTS)
The routine tries to read data until a packet is read or the `buffered_read_package` routine returns a `NORMAL_READ_FAILURE` (an internally defined constant for the module, which means that the read operation resulted in an `EWOULDBLOCK`). If the port is in `command` mode, the reading stops when one package is read. If the port is in `data` mode, the reading continues until the socket buffer is empty (read failure). If no more data can be read and more is wanted (which is always the case when the socket is in `data` mode), `driver_select` is called to make the `uds_input` callback be called when more data is available for reading.

When the port is in `data` mode, all data is sent to Erlang in a format that suits the distribution. In fact, the raw data will never reach any Erlang process, but will be translated/interpreted by the emulator itself and then delivered in the correct format to the correct processes. In the current emulator version, received data is to be tagged with a single byte of 100. That is what the macro `DIST_MAGIC_RECV_TAG` is defined to. The tagging of data in the distribution can be changed in the future.

The `uds_input` routine handles other input events (like non-blocking `accept`), but most importantly handle data arriving at the socket by calling `do_recv`:

```c
( 1) static void uds_input(ErlDrvData handle, ErlDrvEvent event)
( 2) {
( 3)     UdsData *ud = (UdsData *) handle;
( 4)     if (ud->type == portTypeListener) {
( 5)         UdsData *ad = ud->partner;
( 6)         struct sockaddr_un peer;
( 7)         int pl = sizeof(struct sockaddr_un);
( 8)         int fd;
( 9)         if ((fd = accept(ud->fd, (struct sockaddr *) &peer, &pl)) < 0) {
(10)             if (errno != EWOULDBLOCK) {
(11)                 driver_failure_posix(ud->port, errno);
(12)                 return;
(13)             }
(14)             return;
(15)         }
(16)         SET_NONBLOCKING(fd);
(17)         ad->fd = fd;
(18)         ad->partner = NULL;
(19)         ad->type = portTypeCommand;
(20)         ud->partner = NULL;
(21)         driver_select(ud->port, (ErlDrvEvent) ud->fd, DO_READ, 0);
(22)         driver_output(ad->port, "Aok",3);
(23)         return;
(24)     } else {
(25)         do_recv(ud);
(26)     }
```

The important line is the last line in the function: the `do_read` routine is called to handle new input. The remaining function handles input on a listen socket, which means that it is to be possible to do an `accept` on the socket, which is also recognized as a read event.

The output mechanisms are similar to the input. The `do_send` routine is as follows:
1.6 How to Implement an Alternative Carrier for the Erlang Distribution

This driver uses the `writev` system call to send data onto the socket. A combination of `writev` and the driver output queues is very convenient. An `ErlIOVec` structure contains a `SysIOVec` (which is equivalent to the `struct iovec` structure defined in `uio.h`). The `ErlIOVec` also contains an array of `ErlDrvBinary` pointers, of the same length as the number of buffers in the I/O vector itself. One can use this to allocate the binaries for the queue "manually" in the driver, but here the binary array is filled with `NULL` values (line 7). The runtime system then allocates its own buffers when `driver_enqv` is called (line 37).

The routine builds an I/O vector containing the header bytes and the buffer (the opcode has been removed and the buffer length decreased by the output routine). If the queue is empty, we write the data directly to the socket (or at least try to). If any data is left, it is stored in the queue and then we try to send the queue (line 38). An acknowledgement is sent when the message is delivered completely (line 22). The `send_out_queue` sends acknowledgements if the sending is completed there. If the port is in command mode, the Erlang code serializes the send operations so that only one packet can be waiting for delivery at a time. Therefore the acknowledgement can be sent whenever the queue is empty.

The `send_out_queue` routine is as follows:

```c
(1) static void do_send(UdsData *ud, char *buff, int bufflen)
(2) {
(3)     char header[4];
(4)     int written;
(5)     SysIOVec iov[2];
(6)     ErlIOVec eio;
(7)     ErlDrvBinary *binv[] = {NULL,NULL};
(8)     put_packet_length(header, bufflen);
(9)     iov[0].iov_base = (char *) header;
(10)    iov[0].iov_len = 4;
(11)    iov[1].iov_base = buff;
(12)    iov[1].iov_len = bufflen;
(13)    eio.iov = iov;
(14)    eio.binv = binv;
(15)    eio.vsize = 2;
(16)    eio.size = bufflen + 4;
(17)    written = 0;
(18)    if (driver_sizeq(ud->port) == 0) {
(19)        if ((written = writev(ud->fd, iov, 2)) == eio.size) {
(20)            ud->sent += written;
(21)            if (ud->type == portTypeCommand) {
(22)                driver_output(ud->port, "Sok", 3);
(23)            }
(24)        } else if (written < 0) {
(25)            if (errno != EWOULDBLOCK) {
(26)                driver_failure_eof(ud->port);
(27)                return;
(28)            } else {
(29)                written = 0;
(30)            }
(31)        } else {
(32)            ud->sent += written;
(33)        }
(34)    }
(35)    /* Enqueue remaining */
(36)    driver_enqv(ud->port, &eio, written);
(37)    send_out_queue(ud);
(38) }
```
1.6 How to Implement an Alternative Carrier for the Erlang Distribution

We simply pick out an I/O vector from the queue (which is the whole queue as a `SysIOVec`). If the I/O vector is too long (`IO_VECTOR_MAX` is defined to 16), the vector length is decreased (line 15), otherwise the `writev` call (line 17) fails. Writing is tried and anything written is dequeued (line 27). If the write fails with `EWOULDBLOCK` (notice that all sockets are in non-blocking mode), `driver_select` is called to make the `uds_output` routine be called when there is space to write again.

We continue trying to write until the queue is empty or the writing blocks.

The routine above is called from the `uds_output` routine:

```c
(1) static void uds_output(ErlDrvData handle, ErlDrvEvent event)
(2) {
(3)    UdsData *ud = (UdsData *) handle;
(4)    if (ud->type == portTypeConnector) {
(5)        ud->type = portTypeCommand;
(6)        driver_select(ud->port, (ErlDrvEvent) ud->fd, DO_WRITE, 0);
(7)        driver_output(ud->port, "Cok", 3);
(8)    }
(9)    send_out_queue(ud);
(10) }
```

The routine is simple: it first handles the fact that the output select will concern a socket in the business of connecting (and the connecting blocked). If the socket is in a connected state, it simply sends the output queue. This routine is called when it is possible to write to a socket where we have an output queue, so there is no question what to do.

The driver implements a control interface, which is a synchronous interface called when Erlang calls `erlang:port_control/3`. Only this interface can control the driver when it is in `data` mode. It can be called with the following opcodes:
1.6 How to Implement an Alternative Carrier for the Erlang Distribution

'C'
Sets port in command mode.

'I'
Sets port in intermediate mode.

'D'
Sets port in data mode.

'N'
Gets identification number for listen port. This identification number is used in an accept command to the driver. It is returned as a big-endian 32-bit integer, which is the file identifier for the listen socket.

'S'
Gets statistics, which is the number of bytes received, the number of bytes sent, and the number of bytes pending in the output queue. This data is used when the distribution checks that a connection is alive (ticking). The statistics is returned as three 32-bit big-endian integers.

'T'
Sends a tick message, which is a packet of length 0. Ticking is done when the port is in data mode, so the command for sending data cannot be used (besides it ignores zero length packages in command mode). This is used by the ticker to send dummy data when no other traffic is present.

Note: It is important that the interface for sending ticks is not blocking. This implementation uses erlang:port_control/3, which does not block the caller. If erlang:port_command is used, use erlang:port_command/3 and pass [force] as option list; otherwise the caller can be blocked indefinitely on a busy port and prevent the system from taking down a connection that is not functioning.

'R'
Gets creation number of a listen socket, which is used to dig out the number stored in the lock file to differentiate between invocations of Erlang nodes with the same name.

The control interface gets a buffer to return its value in, but is free to allocate its own buffer if the provided one is too small. The uds_control code is as follows:
1.6 How to Implement an Alternative Carrier for the Erlang Distribution

```c
static int uds_control(ErlDrvData handle, unsigned int command,
                        char* buf, int count, char** res, int res_size)
{
    UdsData *ud = (UdsData *) handle;

    switch (command) {
    case 'S':
        {
            unsigned int i;
            **res = 0;
            put_packet_length((*res) + 1, ud->received);
            put_packet_length((*res) + 5, ud->sent);
            put_packet_length((*res) + 9, driver_sizeq(ud->port));
            return 13;
        }
    case 'C':
        if (ud->type < portTypeCommand) {
            return report_control_error(res, res_size, "einval");
        }
        ud->type = portTypeCommand;
        driver_select(ud->port, (ErlDrvEvent) ud->fd, DO_READ, 0);
        ENSURE(1);
        **res = 0;
        return 1;
    case 'I':
        if (ud->type < portTypeCommand) {
            return report_control_error(res, res_size, "einval");
        }
        ud->type = portTypeIntermediate;
        driver_select(ud->port, (ErlDrvEvent) ud->fd, DO_READ, 0);
        ENSURE(1);
        **res = 0;
        return 1;
    case 'D':
        if (ud->type < portTypeCommand) {
            return report_control_error(res, res_size, "einval");
        }
        ud->type = portTypeData;
        do_recv(ud);
        ENSURE(1);
        **res = 0;
        return 1;
    case 'N':
        if (ud->type != portTypeListener) {
            return report_control_error(res, res_size, "einval");
        }
        ENSURE(5);
        (*res)[0] = 0;
        put_packet_length((*res) + 1, ud->fd);
        return 5;
    case 'T': /* tick */
        if (ud->type != portTypeData) {
            return report_control_error(res, res_size, "einval");
        }
        do_send(ud,"",0);
        ENSURE(1);
        **res = 0;
```

Ericsson AB. All Rights Reserved.: Erlang Run-Time System Application (ERTS) | 51
1.6 How to Implement an Alternative Carrier for the Erlang Distribution

The macro `ENSURE` (line 5-10) is used to ensure that the buffer is large enough for the answer. We switch on the command and take actions. We always have read select active on a port in data mode (achieved by calling `do_recv` on line 45), but we turn off read selection in intermediate and command modes (line 27 and 36).

The rest of the driver is more or less UDS-specific and not of general interest.

1.6.4 Putting It All Together

To test the distribution, the `net_kernel:start/1` function can be used. It is useful, as it starts the distribution on a running system, where tracing/debugging can be performed. The `net_kernel:start/1` routine takes a list as its single argument. The list first element in the list is to be the node name (without the "@hostname") as an atom. The second (and last) element is to be one of the atoms `shortnames` or `longnames`. In the example case, `shortnames` is preferred.

For `net_kernel` to find out which distribution module to use, command-line argument `-proto_dist` is used. It is followed by one or more distribution module names, with suffix "_dist" removed, that is, `uds_dist` as a distribution module is specified as `-proto_dist uds`.

If no `epmd` (TCP port mapper daemon) is used, also command-line option `-no_epmd` is to be specified, which makes Erlang skip the `epmd` startup, both as an OS process and as an Erlang ditto.

The path to the directory where the distribution modules reside must be known at boot. This can be achieved either by specifying `-pa <path>` on the command line or by building a boot script containing the applications used for your distribution protocol. (In the `uds_dist` protocol, only the `uds_dist` application needs to be added to the script.)

The distribution starts at boot if all the above is specified and an `-sname <name>` flag is present at the command line.

Example 1:

$ erl -pa $ERL_TOP/lib/kernel/examples/uds_dist/ebin -proto_dist uds -no_epmd
Erlang (BEAM) emulator version 5.0
Eshell V5.0 (abort with ^G)
1> net_kernel:start([bing,shortnames]).
{ok,<{0.30.0}>}

Example 2:

$ erl -pa $ERL_TOP/lib/kernel/examples/uds_dist/ebin -proto_dist uds \   -no_epmd -sname bong
Erlang (BEAM) emulator version 5.0
Eshell V5.0 (abort with ^G)
(bong@hador)1>
1.7 How to Implement an Alternative Node Discovery for Erlang Distribution

The `ERL_FLAGS` environment variable can be used to store the complicated parameters in:

```
$ ERL_FLAGS=-pa $ERL_TOP/lib/kernel/examples/uds_dist/ebin \
    -proto dist uds -no_epmd
$ export ERL_FLAGS
$ erl -sname bang
Erlang (BEAM) emulator version 5.0
Eshell V5.0  (abort with ^G)
(bang@hador)1>
```

ERL_FLAGS should not include the node name.

1.7 How to Implement an Alternative Node Discovery for Erlang Distribution

This section describes how to implement an alternative node discovery mechanism for Erlang distribution. Node discovery is normally done using DNS and the Erlang Port Mapper Daemon (EPMD) for port registration and lookup.

**Note:**
Support for alternative node discovery mechanisms was added in Erlang/OTP 21.

1.7.1 Introduction

To implement your own node discovery module you have to write your own EPMD module. The EPMD module is responsible for providing the location of another node. The distribution modules (`inet_tcp_dist/inet_tls_dist`) call the EPMD module to get the IP address and port of the other node. The EPMD module that is part of Erlang/OTP will resolve the hostname using DNS and uses the EPMD unix process to get the port of another node. The EPMD unix process does this by connecting to the other node on a well-known port, port 4369.

1.7.2 Discovery module

The discovery module needs to implement the same API as the regular EPMD module. However, instead of communicating with EPMD you can connect to any service to find out connection details of other nodes. A discovery module is enabled by setting `-epmd_module` when starting erlang. The discovery module must implement the following callbacks:

- `start_link/0`
  - Start any processes needed by the discovery module.
- `names/1`
  - Return node names held by the registrar for the given host.
- `register_node/2`
  - Register the given node name with the registrar.
- `port_please/3`
  - Return the distribution port used by the given node.

The discovery module may implement the following callback:

- `address_please/3`
  - Return the address of the given node. If not implemented, `erl_epmd:address_please/3` will be used instead.
  - This callback may also return the port of the given node. In that case `port_please/3` may be omitted.
1.8 The Abstract Format

This section describes the standard representation of parse trees for Erlang programs as Erlang terms. This representation is known as the abstract format. Functions dealing with such parse trees are `compile:forms/1,2` and functions in the following modules:

- `epp(3)`
- `erl_eval(3)`
- `erl_lint(3)`
- `erl_parse(3)`
- `erl_pp(3)`
- `io(3)`

The functions are also used as input and output for parse transforms, see the `compile(3)` module.

We use the function `Rep` to denote the mapping from an Erlang source construct `C` to its abstract format representation `R`, and write `R = Rep(C)`.

The word `ANNO` in this section represents an annotation, and denotes among other things the number of the line in the source file where the construction occurred. See `erl_anno(3)` for details. Several instances of `ANNO` in the same construction can denote different annotations.

As operators are not terms in their own right, when operators are mentioned below, the representation of an operator is to be taken to be the atom with a printname consisting of the same characters as the operator.

1.8.1 Module Declarations and Forms

A module declaration consists of a sequence of forms, which are either function declarations or attributes.

- If `D` is a module declaration consisting of the forms `F_1, ..., F_k`, then `Rep(D) = [Rep(F_1), ..., Rep(F_k)]`.
- If `F` is an attribute `-export([Fun_1/A_1, ..., Fun_k/A_k])`, then `Rep(F) = {attribute,ANNO,export,[{Fun_1,A_1}], ..., {Fun_k,A_k}]}`.
- If `F` is an attribute `-import(Mod,[Fun_1/A_1, ..., Fun_k/A_k])`, then `Rep(F) = {attribute,ANNO,import,{Mod,[{Fun_1,A_1}], ..., {Fun_k,A_k}]}}`.
- If `F` is an attribute `-module(Mod), then Rep(F) = {attribute,ANNO,module,Mod}`.
- If `F` is an attribute `-file(File,Line), then Rep(F) = {attribute,ANNO,file,{File,Line}}`.
- If `F` is a function declaration `Name Fc_1 ; ... ; Name Fc_k`, where each `Fc_i` is a function clause with a pattern sequence of the same length `Arity`, then `Rep(F) = {function,ANNO,Name,Arity, [Rep(Fc_1), ..., Rep(Fc_k)]}`.
- If `F` is a function specification `-spec Name Ft_1; ...; Ft_k`, where Spec is either the atom `spec` or the atom `callback`, and each `Ft_i` is a possibly constrained function type with an argument sequence of the same length `Arity`, then `Rep(F) = {attribute,ANNO,Spec,[{Name,Arity},{Rep(Ft_1), ..., Rep(Ft_k)]}, ...]}`.
- If `F` is a function specification `-spec Mod:Name Ft_1; ...; Ft_k`, where each `Ft_i` is a possibly constrained function type with an argument sequence of the same length `Arity`, then `Rep(F) = {attribute,ANNO,spec,[{Mod,Name,Arity},{Rep(Ft_1), ..., Rep(Ft_k)]}}`.
1.8 The Abstract Format

- If F is a record declaration -record(Name,(V_1, ..., V_k)), where each V_i is a record field, then Rep(F) = {attribute,ANNO,record,{Name,[Rep(V_1), ..., Rep(V_k)]}}. For Rep(V), see below.
- If F is a type declaration -Type Name(V_1, ..., V_k) :: T, where Type is either the atom type or the atom opaque, each V_i is a type variable, and T is a type, then Rep(F) = {attribute,ANNO,Type, {Name,Rep(T),[Rep(V_1), ..., Rep(V_k)]}}.
- If F is a wild attribute -A(T), then Rep(F) = {attribute,ANNO,A,T}.

Record Fields

Each field in a record declaration can have an optional, explicit, default initializer expression, and an optional type.

- If V is A, then Rep(V) = {record_field,ANNO,Rep(A)}.
- If V is A = E, where E is an expression, then Rep(V) = {record_field,ANNO,Rep(A),Rep(E)}.
- If V is A :: T, where T is a type, then Rep(V) = {typed_record_field, {record_field,ANNO,Rep(A)},Rep(T)}.
- If V is A = E :: T, where E is an expression and T is a type, then Rep(V) = {typed_record_field, {record_field,ANNO,Rep(A),Rep(E)},Rep(T)}.

Representation of Parse Errors and End-of-File

In addition to the representations of forms, the list that represents a module declaration (as returned by functions in epp(3) and erl_parse(3)) can contain the following:

- Tuples {error,E} and {warning,W}, denoting syntactically incorrect forms and warnings.
- {eof,LOCATION}, denoting an end-of-stream encountered before a complete form had been parsed. The word LOCATION represents a location, and denotes the number of the last line, and possibly the number of the last column on that line, in the source file. See erl_anno(3) for details.

See the form_info/0 type in erl_parse(3) for more details about these values.

1.8.2 Atomic Literals

There are five kinds of atomic literals, which are represented in the same way in patterns, expressions, and guards:

- If L is an atom literal, then Rep(L) = {atom,ANNO,L}.
- If L is a character literal, then Rep(L) = {char,ANNO,L}.
- If L is a float literal, then Rep(L) = {float,ANNO,L}.
- If L is an integer literal, then Rep(L) = {integer,ANNO,L}.
- If L is a string literal consisting of the characters C_1, ..., C_k, then Rep(L) = {string,ANNO,[C_1, ..., C_k]}.

Notice that negative integer and float literals do not occur as such; they are parsed as an application of the unary negation operator.

1.8.3 Patterns

If Ps is a sequence of patterns P_1, ..., P_k, then Rep(Ps) = [Rep(P_1), ..., Rep(P_k)]. Such sequences occur as the list of arguments to a function or fun.

Individual patterns are represented as follows:

- If P is an atomic literal L, then Rep(P) = Rep(L).
- If P is a bitstring pattern <<P_1:Size_1/TSL_1, ..., P_k:Size_k/TSL_k>>, where each Size_i is an expression that can be evaluated to an integer, and each TSL_i is a type specifier list, then Rep(P) = {bin,ANNO,[(bin_element,ANNO,Rep(P_1),Rep(Size_1),Rep(TSL_1)), ...,}.
1.8 The Abstract Format

{bin_element, ANNO, Rep(P_k), Rep(Size_k), Rep(TSL_k))}. For Rep(TSL), see below. An omitted Size_i is represented by default. An omitted TSL_i is represented by default.

• If P is a compound pattern \( P_1 = P_2 \), then Rep(P) = \{match, ANNO, Rep(P_1), Rep(P_2)\}.

• If P is a cons pattern \([P_h \mid P_t]\), then Rep(P) = \{cons, ANNO, Rep(P_h), Rep(P_t)\}.

• If P is a map pattern \(#(A_1, \ldots, A_k)\), where each \( A_i \) is an association \( P_i_1 \Rightarrow P_i_2 \), then Rep(P) = \{map, ANNO, [Rep(A_1), \ldots, Rep(A_k)]\}. For Rep(A), see below.

• If P is a nil pattern [], then Rep(P) = \{nil, ANNO\}.

• If P is an operator pattern \( P_1 \ Op P_2 \), where \( Op \) is a binary operator (this is either an occurrence of ++ applied to a literal string or character list, or an occurrence of an expression that can be evaluated to a number at compile time), then Rep(P) = \{op, ANNO, Op, Rep(P_1), Rep(P_2)\}.

• If P is an operator pattern \( Op P_0 \), where \( Op \) is a unary operator (this is an occurrence of an expression that can be evaluated to a number at compile time), then Rep(P) = \{op, ANNO, Op, Rep(P_0)\}.

• If P is a parenthesized pattern \( (P_0) \), then Rep(P) = Rep(P_0), that is, parenthesized patterns cannot be distinguished from their bodies.

• If P is a record field index pattern \#Name.Field, where Field is an atom, then Rep(P) = \{record_index, ANNO, Name, Rep(Field)\}.

• If P is a record pattern \#Name{Field_1=P_1, \ldots, Field_k=P_k}, where each Field_i is an atom or _, then Rep(P) = \{record, ANNO, Name, [{record_field, ANNO, Rep(Field_1), Rep(P_1)}, \ldots, {record_field, ANNO, Rep(Field_k), Rep(P_k)}]\}.

• If P is a tuple pattern \( \{P_1, \ldots, P_k\} \), then Rep(P) = \{tuple, ANNO, [Rep(P_1), \ldots, Rep(P_k)]\}.

• If P is a universal pattern _, then Rep(P) = \{var, ANNO, _\}.

• If P is a variable pattern V, then Rep(P) = \{var, ANNO, A\}, where A is an atom with a printname consisting of the same characters as V.

Notice that every pattern has the same source form as some expression, and is represented in the same way as the corresponding expression.

1.8.4 Expressions

A body B is a non-empty sequence of expressions \( E_1, \ldots, E_k \), and Rep(B) = [Rep(E_1), \ldots, Rep(E_k)].

An expression E is one of the following:

• If E is an atomic literal L, then Rep(E) = Rep(L).

• If E is a bitstring comprehension <<E_0 || Q_1, \ldots, Q_k>>, where each Q_i is a qualifier, then Rep(E) = \{bc, ANNO, Rep(E_0), [Rep(Q_1), \ldots, Rep(Q_k)]\}. For Rep(Q), see below.

• If E is a bitstring constructor <<E_1:Size_1/TSL_1, \ldots, E_k:Size_k/TSL_k>>, where each Size_i is an expression and each TSL_i is a type specifier list, then Rep(E) = \{bin, ANNO, [{bin_element, ANNO, Rep(E_1), Rep(Size_1), Rep(TSL_1)}, \ldots, {bin_element, ANNO, Rep(E_k), Rep(Size_k), Rep(TSL_k)}]\}. For Rep(TSL), see below. An omitted Size_i is represented by default. An omitted TSL_i is represented by default.

• If E is a block expression begin B end, where B is a body, then Rep(E) = \{block, ANNO, Rep(B)\}.

• If E is a case expression case E_0 of Cc_1 ; \ldots ; Cc_k end, where E_0 is an expression and each Cc_i is a case clause, then Rep(E) = \{'case', ANNO, Rep(E_0), [Rep(Cc_1), \ldots, Rep(Cc_k)]\}.

• If E is a cons expression cons E_0 then Rep(E) = \{'cons', ANNO, Rep(E_0)\}.

• If E is a fun expression fun Name/Arity, then Rep(E) = \{'fun', ANNO, {function, Name, Arity}\}.
1.8 The Abstract Format

- If $E$ is a fun expression `fun Module:Name/Arity`, then $\text{Rep}(E) = \{\text{'fun'}, \text{ANNO}, \{\text{function}, \text{Rep(Module)}, \text{Rep(Name)}, \text{Rep(Arity)}\}\}$.
- If $E$ is a fun expression `fun Fc_1 ; ... ; Fc_k end`, where each $Fc_i$ is a function clause, then $\text{Rep}(E) = \{\text{'fun'}, \text{ANNO}, \{\text{clauses}, \{\text{Rep(Fc_1)}, \ldots, \text{Rep(Fc_k)}\}\}\}$.
- If $E$ is a fun expression `fun Name Fc_1 ; ... ; Name Fc_k end`, where Name is a variable and each $Fc_i$ is a function clause, then $\text{Rep}(E) = \{\text{name_fun}, \text{ANNO}, \text{Name}, \{\text{Rep(Fc_1)}, \ldots, \text{Rep(Fc_k)}\}\}$.
- If $E$ is a function call $E_0(E_1, \ldots, E_k)$, then $\text{Rep}(E) = \{\text{call}, \text{ANNO}, \{\text{Rep(E_0)}, \{\text{Rep(E_1)}, \ldots, \text{Rep(E_k)}\}\}\}$.
- If $E$ is a function call $E_m:E_0(E_1, \ldots, E_k)$, then $\text{Rep}(E) = \{\text{call}, \text{ANNO}, \{\text{remote}, \text{ANNO}, \text{Rep(E_m)}, \text{Rep(E_0)}, \{\text{Rep(E_1)}, \ldots, \text{Rep(E_k)}\}\}\}$.
- If $E$ is an if expression `if Ic_1 ; ... ; Ic_k end`, where each $Ic_i$ is an if clause, then $\text{Rep}(E) = \{\text{'if'}, \text{ANNO}, \{\text{Rep(Ic_1)}, \ldots, \text{Rep(Ic_k)}\}\}$.
- If $E$ is a list comprehension `$[E_0 || Q_1, \ldots, Q_k]$`, where each $Q_i$ is a qualifier, then $\text{Rep}(E) = \{\text{lc}, \text{ANNO}, \text{Rep(E_0)}, \{\text{Rep(Q_1)}, \ldots, \text{Rep(Q_k)}\}\}$.
- For $\text{Rep}(Q)$, see below.
- If $E$ is a map comprehension `#{E_0 || Q_1, \ldots, Q_k}`, where $E_0$ is an association $K => V$ and each $Q_i$ is a qualifier, then $\text{Rep}(E) = \{\text{mc}, \text{ANNO}, \text{Rep(E_0)}, \{\text{Rep(Q_1)}, \ldots, \text{Rep(Q_k)}\}\}$. For $\text{Rep(E_0)}$ and $\text{Rep(Q)}$, see below.
- If $E$ is a map creation `#{A_1, \ldots, A_k}`, where each $A_i$ is an association $E_i_1 => E_i_2$, then $\text{Rep}(E) = \{\text{map}, \text{ANNO}, \{\text{Rep(A_1)}, \ldots, \text{Rep(A_k)}\}\}$. For $\text{Rep(A)}$, see below.
- If $E$ is a map update $E_0#{A_1, \ldots, A_k}$, where each $A_i$ is an association $E_i_1 => E_i_2$ or $E_i_1 := E_i_2$, then $\text{Rep}(E) = \{\text{map}, \text{ANNO}, \text{Rep(E_0)}, \{\text{Rep(A_1)}, \ldots, \text{Rep(A_k)}\}\}$. For $\text{Rep(A)}$, see below.
- If $E$ is a match operator expression $P = E_0$, where $P$ is a pattern, then $\text{Rep}(E) = \{\text{match}, \text{ANNO}, \text{Rep(P)}, \text{Rep(E_0)}\}$.
- If $E$ is a conditional match operator expression $P ?= E_0$, where $P$ is a pattern, then $\text{Rep}(E) = \{\text{maybe_match}, \text{ANNO}, \text{Rep(P)}, \text{Rep(E_0)}\}$.
- If $E$ is a maybe expression `maybe B end`, where $B$ is a body then $\text{Rep}(E) = \{\text{'maybe'}, \text{ANNO}, \text{Rep(B)}\}$.
- If $E$ is a maybe expression `maybe B else Ec_1 ; ... ; Ec_k end`, where $B$ is a body and each $Ec_i$ is an else clause then $\text{Rep}(E) = \{\text{'maybe'}, \text{ANNO}, \text{Rep(B)}, \{'else', \text{ANNO}, \{\text{Rep(Ec_1)}, \ldots, \text{Rep(Tc_k)}\}\}\}$.
- If $E$ is nil `[]`, then $\text{Rep}(E) = \{\text{nil}, \text{ANNO}\}$.
- If $E$ is an operator expression $E_1 \op E_2$, where $\op$ is a binary operator other than match operator $\= $, then $\text{Rep}(E) = \{\text{op}, \text{ANNO}, \text{Rep(E_1)}, \text{Rep(E_2)}\}$.
- If $E$ is an operator expression $\op E_0$, where $\op$ is a unary operator, then $\text{Rep}(E) = \{\text{op}, \text{ANNO}, \text{Rep(E_0)}\}$.
- If $E$ is a parenthesized expression `( E_0 )`, then $\text{Rep}(E) = \text{Rep}(E_0)$, that is, parenthesized expressions cannot be distinguished from their bodies.
- If $E$ is a receive expression `receive Cc_1 ; ... ; Cc_k end`, where each $Cc_i$ is a case clause, then $\text{Rep}(E) = \{\text{'receive'}, \text{ANNO}, \{\text{Rep(Cc_1)}, \ldots, \text{Rep(Cc_k)}\}\}$.
- If $E$ is a receive expression `receive Cc_1 ; ... ; Cc_k after E_0 -> B_t end`, where each $Cc_i$ is a case clause, $E_0$ is an expression, and $B_t$ is a body, then $\text{Rep}(E) = \{\text{'receive'}, \text{ANNO}, \{\text{Rep(Cc_1)}, \ldots, \text{Rep(Cc_k)}\}, \text{Rep(E_0)}, \text{Rep(B_t)}\}$.
- If $E$ is a record creation `#Name{Field_1=E_1, \ldots, Field_k=E_k}`, where each $Field_i$ is an atom or an atom or `__`, then $\text{Rep}(E) = \{\text{record}, \text{ANNO}, \text{Name}, \{\text{record_field}, \text{ANNO}, \text{Rep(Field_1)}, \text{Rep(E_1)}\}, \ldots, \{\text{record_field}, \text{ANNO}, \text{Rep(Field_k)}, \text{Rep(E_k)}\}\}$. If $E$ is a record field access `E_0#Name.Field`, where $Field$ is an atom, then $\text{Rep}(E) = \{\text{record_field}, \text{ANNO}, \text{Rep(E_0)}, \text{Name}, \text{Rep(Field)}\}$.
1.8 The Abstract Format

- If \( E \) is a record field index \(#\text{Name.Field}\), where \( \text{Name} \) is an atom, then \( \text{Rep}(E) = \{\text{record_index}, \text{ANNO}, \text{Name}, \text{Rep}(\text{Field})\} \).

- If \( E \) is a record update \( E_0\#\text{Name}\{\text{Field}_1=\text{E}_1, \ldots, \text{Field}_k=\text{E}_k\} \), where each \( \text{Field}_i \) is an atom, then \( \text{Rep}(E) = \{\text{record}, \text{ANNO}, \text{Rep}(\text{E}_0), \text{Name}, \{\{\text{record_field}, \text{ANNO}, \text{Rep}(\text{Field}_1), \text{Rep}(\text{E}_1)\}, \ldots, \{\text{record_field}, \text{ANNO}, \text{Rep}(\text{Field}_k), \text{Rep}(\text{E}_k)\}\}\} \).

- If \( E \) is a tuple skeleton \( \{\text{E}_1, \ldots, \text{E}_k\} \), then \( \text{Rep}(E) = \{\text{tuple}, \text{ANNO}, \{\text{Rep}(\text{E}_1), \ldots, \text{Rep}(\text{E}_k)\}\} \).

- If \( E \) is a try expression \( \text{try B catch Tc}_1 \ldots; \text{Tc}_k \text{ end} \), where \( B \) is a body and each \( \text{Tc}_i \) is a catch clause, then \( \text{Rep}(E) = \{\text{'try'}, \text{ANNO}, \text{Rep}(B), [], \{\text{Rep}(\text{Tc}_1), \ldots, \text{Rep}(\text{Tc}_k)\}\} \).

- If \( E \) is a try expression \( \text{try B of Cc}_1 \ldots; \text{Cc}_k \text{ catch Tc}_1 \ldots; \text{Tc}_n \text{ end} \), where \( B \) is a body, each \( \text{Cc}_i \) is a case clause, and each \( \text{Tc}_j \) is a catch clause, then \( \text{Rep}(E) = \{\text{'try'}, \text{ANNO}, \text{Rep}(B), \{\text{Rep}(\text{Cc}_1), \ldots, \text{Rep}(\text{Cc}_k)\}, \{\text{Rep}(\text{Tc}_1), \ldots, \text{Rep}(\text{Tc}_n)\}\} \).

- If \( E \) is a try expression \( \text{try B after A end} \), where \( B \) and \( A \) are bodies, then \( \text{Rep}(E) = \{\text{'try'}, \text{ANNO}, \text{Rep}(B), [], [], \text{Rep}(A)\} \).

- If \( E \) is a try expression \( \text{try B of Cc}_1 \ldots; \text{Cc}_k \text{ after A end} \), where \( B \) and \( A \) are bodies, and each \( \text{Cc}_i \) is a case clause, then \( \text{Rep}(E) = \{\text{'try'}, \text{ANNO}, \text{Rep}(B), \{\text{Rep}(\text{Cc}_1), \ldots, \text{Rep}(\text{Cc}_k)\}, [], \text{Rep}(A)\} \).

- If \( E \) is a try expression \( \text{try B catch Tc}_1 \ldots; \text{Tc}_k \text{ after A end} \), where \( B \) and \( A \) are bodies, and each \( \text{Tc}_i \) is a catch clause, then \( \text{Rep}(E) = \{\text{'try'}, \text{ANNO}, \text{Rep}(B), [], [], \text{Rep}(\text{Tc}_1), \ldots, \text{Rep}(\text{Tc}_k)\}, \text{Rep}(A)\} \).

- If \( E \) is a variable \( V \), then \( \text{Rep}(E) = \{\text{var}, \text{ANNO}, A\} \), where \( A \) is an atom with a printname consisting of the same characters as \( V \).

Qualifiers

A qualifier \( Q \) is one of the following:

- If \( Q \) is a filter \( E \), where \( E \) is an expression, then \( \text{Rep}(Q) = \text{Rep}(E) \).

- If \( Q \) is a list generator \( P \leftarrow E \), where \( P \) is a pattern and \( E \) is an expression, then \( \text{Rep}(Q) = \{\text{generate}, \text{ANNO}, \text{Rep}(P), \text{Rep}(E)\} \).

- If \( Q \) is a bitstring generator \( P \leftarrow E \), where \( P \) is a pattern and \( E \) is an expression, then \( \text{Rep}(Q) = \{\text{b_generate}, \text{ANNO}, \text{Rep}(P), \text{Rep}(E)\} \).

- If \( Q \) is a map generator \( P \leftarrow E \), where \( P \) is an association pattern \( P_1 := P_2 \) and \( E \) is an expression, then \( \text{Rep}(Q) = \{\text{m_generate}, \text{ANNO}, \text{Rep}(P), \text{Rep}(E)\} \). For \( \text{Rep}(P) \), see below.

Bitstring Element Type Specifiers

A type specifier list \( \text{TSL} \) for a bitstring element is a sequence of type specifiers \( \text{TS}_1 \ldots \text{TS}_k \) and \( \text{Rep} (\text{TSL}) = \{\text{Rep}(\text{TS}_1), \ldots, \text{Rep}(\text{TS}_k)\} \).

- If \( \text{TS} \) is a type specifier \( A \), where \( A \) is an atom, then \( \text{Rep}(\text{TS}) = A \).

- If \( \text{TS} \) is a type specifier \( A : \text{Value} \), where \( A \) is an atom and \( \text{Value} \) is an integer, then \( \text{Rep}(\text{TS}) = \{A, \text{Value}\} \).

Associations

An association \( A \) is one of the following:

- If \( A \) is an association \( K \Rightarrow V \), then \( \text{Rep}(A) = \{\text{map_field_assoc}, \text{ANNO}, \text{Rep}(K), \text{Rep}(V)\} \).
• If A is an association \(K := V\), then \(\text{Rep}(A) = \{\text{map\_field\_exact, ANNO, Rep}(K), \text{Rep}(V)\}\).

### 1.8.5 Clauses

There are function clauses, if clauses, case clauses, and catch clauses.

A clause \(C\) is one of the following:

• If \(C\) is a case clause \(P -> B\), where \(P\) is a pattern and \(B\) is a body, then \(\text{Rep}(C) = \{\text{clause, ANNO, [Rep}(P)], [\], \text{Rep}(B)\}\).

• If \(C\) is a case clause \(P \text{ when } Gs \rightarrow B\), where \(P\) is a pattern, \(Gs\) is a guard sequence, and \(B\) is a body, then \(\text{Rep}(C) = \{\text{clause, ANNO, [Rep}(\{\text{throw, P, _}\}), [\], \text{Rep}(Gs), \text{Rep}(B)\}\), that is, a catch clause with an explicit exception class \text{throw} and with or without an explicit stacktrace variable \text{_} cannot be distinguished from a catch clause without an explicit exception class and without an explicit stacktrace variable.

• If \(C\) is a catch clause \(X : P \to B\), where \(X\) is an atomic literal or a variable pattern, \(P\) is a pattern, and \(B\) is a body, then \(\text{Rep}(C) = \{\text{clause, ANNO, [Rep}(\{X, P, S\}), [\], \text{Rep}(Ps), \text{Rep}(B)\}\), that is, a catch clause with an explicit exception class and with an explicit stacktrace variable \text{_} cannot be distinguished from a catch clause with an explicit exception class and without an explicit stacktrace variable.

• If \(C\) is a catch clause \(X : P \rightarrow B\), where \(X\) is an atomic literal or a variable pattern, \(P\) is a pattern, \(S\) is a variable, and \(B\) is a body, then \(\text{Rep}(C) = \{\text{clause, ANNO, [Rep}(\{X, P, S\}), [\], \text{Rep}(Ps), \text{Rep}(B)\}\).

• If \(C\) is a catch clause \(P \text{ when } Gs \rightarrow B\), where \(P\) is a pattern, \(Gs\) is a guard sequence, and \(B\) is a body, then \(\text{Rep}(C) = \{\text{clause, ANNO, [Rep}(\{\text{throw, P, _}\}), \text{Rep}(Gs), \text{Rep}(B)\}\), that is, a catch clause with an explicit exception class and with an explicit stacktrace variable \text{_} cannot be distinguished from a catch clause without an explicit exception class and without an explicit stacktrace variable.

• If \(C\) is a catch clause \(X : P \text{ when } Gs \rightarrow B\), where \(X\) is an atomic literal or a variable pattern, \(P\) is a pattern, \(Gs\) is a guard sequence, and \(B\) is a body, then \(\text{Rep}(C) = \{\text{clause, ANNO, [Rep}(\{\text{throw, P, _}\}), \text{Rep}(Gs), \text{Rep}(B)\}\), that is, a catch clause with an explicit exception class and with an explicit stacktrace variable \text{_} cannot be distinguished from a catch clause with an explicit exception class and with an explicit stacktrace variable.

• If \(C\) is a catch clause \(X : P : S \rightarrow B\), where \(X\) is an atomic literal or a variable pattern, \(P\) is a pattern, \(S\) is a variable, and \(B\) is a body, then \(\text{Rep}(C) = \{\text{clause, ANNO, [Rep}(\{X, P, S\}), [\], \text{Rep}(Ps), \text{Rep}(B)\}\).

• If \(C\) is a function clause \(( Ps ) \rightarrow B\), where \(Ps\) is a pattern sequence and \(B\) is a body, then \(\text{Rep}(C) = \{\text{clause, ANNO, Rep}(Ps), [\], \text{Rep}(B)\}\).

• If \(C\) is a function clause \(( Ps ) \text{ when } Gs \rightarrow B\), where \(Ps\) is a pattern sequence, \(Gs\) is a guard sequence and \(B\) is a body, then \(\text{Rep}(C) = \{\text{clause, ANNO, Rep}(Ps), \text{Rep}(Gs), \text{Rep}(B)\}\).

• If \(C\) is an \(if\) clause \(( Gs ) \rightarrow B\), where \(Gs\) is a guard sequence and \(B\) is a body, then \(\text{Rep}(C) = \{\text{clause, ANNO, [\], Rep}(Gs), \text{Rep}(B)\}\).

### 1.8.6 Guards

A guard sequence \(Gs\) is a sequence of guards \(G_1; \ldots; G_k\), and \(\text{Rep}(Gs) = [\text{Rep}(G_1), \ldots, \text{Rep}(G_k)]\). If the guard sequence is empty, then \(\text{Rep}(Gs) = [\]\).

A guard \(G\) is a non-empty sequence of guard tests \(Gt_1, \ldots, Gt_k\), and \(\text{Rep}(G) = [\text{Rep}(Gt_1), \ldots, \text{Rep}(Gt_k)]\).

A guard test \(Gt\) is one of the following:

• If \(Gt\) is an atomic literal \(L\), then \(\text{Rep}(Gt) = \text{Rep}(L)\).
1.8 The Abstract Format

- If $Gt$ is a bitstring constructor $\langle \langle Gt_1 : Size_1 / TSL_1, \ldots, Gt_k : Size_k / TSL_k \rangle \rangle$, where each $Size_i$ is a guard test and each $TSL_i$ is a type specifier list, then $Rep(Gt) = \{bin, ANNO, [\{bin_element, ANNO, Rep(Gt_1), Rep(Size_1), Rep(TSL_1)\}, \ldots, \{bin_element, ANNO, Rep(Gt_k), Rep(Size_k), Rep(TSL_k)\}]\}$. For $Rep(TSL)$, see above. An omitted $Size_i$ is represented by default. An omitted $TSL_i$ is represented by default.

- If $Gt$ is a cons skeleton $\langle Gt_h \mid Gt_t \rangle$, then $Rep(Gt) = \{cons, ANNO, Rep(Gt_h), Rep(Gt_t)\}$.

- If $Gt$ is a function call $A(Gt_1, \ldots, Gt_k)$, where $A$ is an atom, then $Rep(Gt) = \{call, ANNO, Rep(A), [Rep(Gt_1), \ldots, Rep(Gt_k)]\}$.

- If $Gt$ is a function call $A_m : A(Gt_1, \ldots, Gt_k)$, where $A_m$ is the atom erlang and $A$ is an atom or an operator, then $Rep(Gt) = \{call, ANNO, \{remote, ANNO, Rep(A_m), Rep(A)\}, [Rep(Gt_1), \ldots, Rep(Gt_k)]\}$.

- If $Gt$ is a map creation $\#\{A_1, \ldots, A_k\}$, where each $A_i$ is an association $Gt_i_1 \mapsto Gt_i_2$, then $Rep(Gt) = \{map, ANNO, [Rep(A_1), \ldots, Rep(A_k)]\}$. For $Rep(A)$, see above.

- If $Gt$ is a map update $Gt_0\#\{A_1, \ldots, A_k\}$, where each $A_i$ is an association $Gt_i_1 \mapsto Gt_i_2$ or $Gt_i_1 := Gt_i_2$, then $Rep(Gt) = \{map, ANNO, Rep(Gt_0), [Rep(A_1), \ldots, Rep(A_k)]\}$.

- If $Gt$ is nil, $\langle\rangle$, then $Rep(Gt) = \{nil, ANNO\}$.

- If $Gt$ is an operator guard test $Gt_1 Op Gt_2$, where $Op$ is a binary operator other than match operator =, then $Rep(Gt) = \{op, ANNO, Op, Rep(Gt_1), Rep(Gt_2)\}$.

- If $Gt$ is an operator guard test $Op Gt_0$, where $Op$ is a unary operator, then $Rep(Gt) = \{op, ANNO, Op, Rep(Gt_0)\}$.

- If $Gt$ is a parenthesized guard test ( $Gt_0$ ), then $Rep(Gt) = Rep(Gt_0)$, that is, parenthesized guard tests cannot be distinguished from their bodies.

- If $Gt$ is a record creation $\#Name\{Field_1=Gt_1, \ldots, Field_k=Gt_k\}$, where each $Field_i$ is an atom or _, then $Rep(Gt) = \{record, ANNO, Name, [\{record_field, ANNO, Rep(Field_1), Rep(Gt_1)\}, \ldots, \{record_field, ANNO, Rep(Field_k), Rep(Gt_k)\}]\}$.

- If $Gt$ is a record field access $Gt_0\#Name.Field$, where $Field$ is an atom, then $Rep(Gt) = \{record_field, ANNO, Rep(Gt_0), Name, Rep(Field)\}$.

- If $Gt$ is a record field index $\#Name.Field$, where $Field$ is an atom, then $Rep(Gt) = \{record_index, ANNO, Name, Rep(Field)\}$.

- If $Gt$ is a tuple skeleton $\langle Gt_1, \ldots, Gt_k \rangle$, then $Rep(Gt) = \{tuple, ANNO, [Rep(Gt_1), \ldots, Rep(Gt_k)]\}$.

- If $Gt$ is a variable pattern $V$, then $Rep(Gt) = \{var, ANNO, A\}$, where $A$ is an atom with a printname consisting of the same characters as $V$.

Notice that every guard test has the same source form as some expression, and is represented in the same way as the corresponding expression.

1.8.7 Types

- If $T$ is an annotated type $A :: T_0$, where $A$ is a variable, then $Rep(T) = \{ann_type, ANNO, [Rep(A), Rep(T_0)]\}$.

- If $T$ is an atom, a character, or an integer literal $L$, then $Rep(T) = Rep(L)$.

- If $T$ is a bitstring type $\langle\langle M, _:*N\rangle\rangle$, where $M$ and $N$ are singleton integer types, then $Rep(T) = \{type, ANNO, binary, [Rep(M), Rep(N)]\}$.

- If $T$ is the empty list type $\langle\rangle$, then $Rep(T) = \{type, ANNO, nil, []\}$, that is, the empty list type $\langle\rangle$ cannot be distinguished from the predefined type $nil()$.

- If $T$ is a fun type $\langle fun() \rangle$, then $Rep(T) = \{type, ANNO, 'fun', []\}$.
1.8 The Abstract Format

• If $T$ is a fun type $\text{fun}((...) \rightarrow T_0)$, then $\text{Rep}(T) = \{\text{type,ANNO,'fun',}\{\{\text{type,ANNO,any},\text{Rep}(T_0)\}\}\}$.

• If $T$ is a fun type $\text{fun}(Ft)$, where $Ft$ is a function type, then $\text{Rep}(T) = \text{Rep}(Ft)$. For $\text{Rep}(Ft)$, see below.

• If $T$ is an integer range type $L \ldots H$, where $L$ and $H$ are singleton integer types, then $\text{Rep}(T) = \{\text{type,ANNO,range,}\{\text{Rep}(L),\text{Rep}(H)\}\}$.

• If $T$ is a map type $\text{map()}$, then $\text{Rep}(T) = \{\text{type,ANNO,map,}\{\}\}$.

• If $T$ is a map type $\#\{A_1, \ldots, A_k\}$, where each $A_i$ is an association type, then $\text{Rep}(T) = \{\text{type,ANNO,map,}\{\text{Rep}(A_1), \ldots, \text{Rep}(A_k)\}\}$.

• If $T$ is an operator type $T_1 \; \text{Op} \; T_2$, where $\text{Op}$ is a binary operator (this is an occurrence of an expression that can be evaluated to an integer at compile time), then $\text{Rep}(T) = \{\text{op,ANNO,Op,}\text{Rep}(T_1),\text{Rep}(T_2)\}$.

• If $T$ is an operator type $\text{Op} \; T_0$, where $\text{Op}$ is a unary operator (this is an occurrence of an expression that can be evaluated to an integer at compile time), then $\text{Rep}(T) = \{\text{op,ANNO,Op,}\text{Rep}(T_0)\}$.

• If $T$ is $\langle T_0 \rangle$, then $\text{Rep}(T) = \text{Rep}(T_0)$, that is, parenthesized types cannot be distinguished from their bodies.

• If $T$ is a predefined (or built-in) type $N(T_1, \ldots, T_k)$, then $\text{Rep}(T) = \{\text{type,ANNO,N,}\{\text{Rep}(T_1), \ldots, \text{Rep}(T_k)\}\}$.

• If $T$ is a record type $\#\text{Name}{F_1, \ldots, F_k}$, where each $F_i$ is a record field type, then $\text{Rep}(T) = \{\text{type,ANNO,record,}\{\text{Rep}(F_1), \ldots, \text{Rep}(F_k)\}\}$.

• If $T$ is a remote type $M:N(T_1, \ldots, T_k)$, then $\text{Rep}(T) = \{\text{remote_type,ANNO,}\{\text{Rep}(M),\text{Rep}(N),\{\text{Rep}(T_1), \ldots, \text{Rep}(T_k)\}\}\}$.

• If $T$ is a tuple type $\text{tuple()}$, then $\text{Rep}(T) = \{\text{type,ANNO,tuple,}\{\}\}$.

• If $T$ is a tuple type $\{T_1, \ldots, T_k\}$, then $\text{Rep}(T) = \{\text{type,ANNO,tuple,}\{\text{Rep}(T_1), \ldots, \text{Rep}(T_k)\}\}$.

• If $T$ is a union $T_1 \mid \ldots \mid T_k$, then $\text{Rep}(T) = \{\text{type,ANNO,union,}\{\text{Rep}(T_1), \ldots, \text{Rep}(T_k)\}\}$.

• If $T$ is a type union $T_1 \mid \ldots \mid T_k$, then $\text{Rep}(T) = \{\text{type,ANNO,union,}\{\text{Rep}(T_1), \ldots, \text{Rep}(T_k)\}\}$.

• If $T$ is a tuple type $\{T_1, \ldots, T_k\}$, then $\text{Rep}(T) = \{\text{type,ANNO,union,}\{\text{Rep}(T_1), \ldots, \text{Rep}(T_k)\}\}$.

• If $T$ is a user-defined type $N(T_1, \ldots, T_k)$, then $\text{Rep}(T) = \{\text{user_type,ANNO,N,}\{\text{Rep}(T_1), \ldots, \text{Rep}(T_k)\}\}$.

Function Types
A function type $Ft$ is one of the following:

• If $Ft$ is a constrained function type $Ft_1 \; \text{when} \; Fc$, where $Ft_1$ is a function type and $Fc$ is a function constraint, then $\text{Rep}(T) = \{\text{type,ANNO,bounded_fun,}\{\text{Rep}(Ft_1),\text{Rep}(Fc)\}\}$.

Function Constraints
A function constraint $Fc$ is a non-empty sequence of constraints $C_1, \ldots, C_k$, and $\text{Rep}(Fc) = \{\text{Rep}(C_1), \ldots, \text{Rep}(C_k)\}$.

Association Types
If $A$ is an association type $K \Rightarrow V$, where $K$ and $V$ are types, then $\text{Rep}(A) = \{\text{type,ANNO,map_field_assoc,}\{\text{Rep}(K),\text{Rep}(V)\}\}$.

Association Types
If $A$ is an association type $K \Rightarrow V$, where $K$ and $V$ are types, then $\text{Rep}(A) = \{\text{type,ANNO,map_field_assoc,}\{\text{Rep}(K),\text{Rep}(V)\}\}$. 

Ericsson AB. All Rights Reserved.: Erlang Run-Time System Application (ERTS) | 61
Record Field Types

- If F is a record field type Name :: Type, where Type is a type, then Rep(F) = {type, ANNO, field_type, [Rep(Name), Rep(Type)]}.

1.8.8 The Abstract Format after Preprocessing

The compilation option debug_info can be specified to the compiler to have the abstract code stored in the abstract_code chunk in the Beam file (for debugging purposes).

As from Erlang/OTP R9C, the abstract_code chunk contains {raw_abstract_v1, AbstractCode}, where AbstractCode is the abstract code as described in this section.

In OTP releases before R9C, the abstract code after some more processing was stored in the Beam file. The first element of the tuple would be either abstract_v1 (in OTP R7B) or abstract_v2 (in OTP R8B).

1.9 tty - A Command-Line Interface

tty is a simple command-line interface program where keystrokes are collected and interpreted. Completed lines are sent to the shell for interpretation. A simple history mechanism saves previous lines, which can be edited before sending them to the shell. tty is started when Erlang is started with the following command:

```
erl
tty```

tty operates in several different modes:

- Normal mode, in which text lines can be edited and sent to the shell.
- Search mode, in which the user can search for previous commands in the history buffer.
- Shell break mode, which allows the user to kill the current shell, start multiple shells, and so on.

1.9.1 Normal Mode

In normal mode keystrokes from the user are collected and interpreted by tty. Most of the Emacs line-editing commands are supported. The following is a complete list of the supported line-editing commands.

Typographic conventions:

- C-a means pressing the Ctrl key and the letter a simultaneously.
- C-S-a means pressing the Ctrl key, the Shift key, and the letter a simultaneously.
- M-f means pressing the Esc key and the letter f in sequence or pressing the Alt key and the letter f simultaneously.
- Home and End represent the keys with the same name on the keyboard.
- Left, Right, Up and Down represent the corresponding arrow keys.
- When a function has multiple possible key sequences they are listed on individual lines in the Key Sequence column.

<table>
<thead>
<tr>
<th>Key Sequence</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>Home</td>
<td>Beginning of line</td>
</tr>
<tr>
<td>C-a</td>
<td>Beginning of line</td>
</tr>
<tr>
<td>C-b</td>
<td>Backward character</td>
</tr>
<tr>
<td>Keys</td>
<td>Function</td>
</tr>
<tr>
<td>-----------</td>
<td>-----------------------------------</td>
</tr>
<tr>
<td>C-Left</td>
<td>Backward word</td>
</tr>
<tr>
<td>M-Left</td>
<td>Backward word</td>
</tr>
<tr>
<td>M-b</td>
<td>Backward word</td>
</tr>
<tr>
<td>C-d</td>
<td>Delete character</td>
</tr>
<tr>
<td>M-d</td>
<td>Delete word</td>
</tr>
<tr>
<td>End</td>
<td>End of line</td>
</tr>
<tr>
<td>C-e</td>
<td>End of line</td>
</tr>
<tr>
<td>C-f</td>
<td>Forward character</td>
</tr>
<tr>
<td>C-Right</td>
<td>Forward word</td>
</tr>
<tr>
<td>M-Right</td>
<td>Forward word</td>
</tr>
<tr>
<td>M-f</td>
<td>Forward word</td>
</tr>
<tr>
<td>C-g</td>
<td>Enter shell break mode</td>
</tr>
<tr>
<td>C-k</td>
<td>Kill line</td>
</tr>
<tr>
<td>C-u</td>
<td>Backward kill line</td>
</tr>
<tr>
<td>C-l</td>
<td>Clears the screen</td>
</tr>
<tr>
<td>M-c</td>
<td>Clears the current expression</td>
</tr>
<tr>
<td>M-l</td>
<td>Redraw line</td>
</tr>
<tr>
<td>C-n</td>
<td>Fetch next line from the history buffer</td>
</tr>
<tr>
<td>C-o</td>
<td>Edit the current line using the editor specified in the environment variable VISUAL or EDITOR. The environment variables can contain arguments to the editor if needed, for example VISUAL=&quot;emacs -nw&quot;. On Windows the editor cannot be a console based editor.</td>
</tr>
<tr>
<td>M-o</td>
<td>Fetch previous line from the history buffer</td>
</tr>
<tr>
<td>C-p</td>
<td>Fetch previous line from the history buffer</td>
</tr>
<tr>
<td>C-r</td>
<td>Enter search mode and then search backward in the shell history</td>
</tr>
<tr>
<td>C-s</td>
<td>In search mode, search forward in the shell history</td>
</tr>
<tr>
<td>C-t</td>
<td>Transpose characters</td>
</tr>
<tr>
<td>C-w</td>
<td>Backward kill word</td>
</tr>
<tr>
<td>C-y</td>
<td>Insert previously killed text</td>
</tr>
</tbody>
</table>
1.10 How to Implement a Driver

<table>
<thead>
<tr>
<th>Key</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>C-[]</td>
<td>Insert matching closing bracket</td>
</tr>
<tr>
<td>C-Up</td>
<td>Navigate one row up when editing multiple lines</td>
</tr>
<tr>
<td>M-Up</td>
<td></td>
</tr>
<tr>
<td>C-Down</td>
<td>Navigate one row down when editing multiple lines</td>
</tr>
<tr>
<td>M-Down</td>
<td></td>
</tr>
<tr>
<td>M-Enter</td>
<td>Insert a new line at cursor</td>
</tr>
<tr>
<td>M-&lt;</td>
<td>Navigate to the start of the current expression</td>
</tr>
<tr>
<td>M-S-Up</td>
<td></td>
</tr>
<tr>
<td>M-&gt;</td>
<td>Navigate to the end of the current expression</td>
</tr>
<tr>
<td>M-S-Down</td>
<td></td>
</tr>
<tr>
<td>Tab</td>
<td>Autocomplete current expression, or show completion suggestions</td>
</tr>
<tr>
<td>C-i</td>
<td></td>
</tr>
<tr>
<td>M-c</td>
<td>Clear current expression</td>
</tr>
</tbody>
</table>

Table 9.1: tty Text Editing

1.9.2 Shell Break Mode

In this mode the following can be done:
- Kill or suspend the current shell
- Connect to a suspended shell
- Start a new shell

1.10 How to Implement a Driver

**Note:**

This section was written a long time ago. Most of it is still valid, as it explains important concepts, but this was written for an older driver interface so the examples do not work anymore. The reader is encouraged to read the `erl_driver` and `driver_entry` documentation also.

1.10.1 Introduction

This section describes how to build your own driver for Erlang.

A driver in Erlang is a library written in C, which is linked to the Erlang emulator and called from Erlang. Drivers can be used when C is more suitable than Erlang, to speed up things, or to provide access to OS resources not directly accessible from Erlang.

A driver can be dynamically loaded, as a shared library (known as a DLL on Windows), or statically loaded, linked with the emulator when it is compiled and linked. Only dynamically loaded drivers are described here, statically linked drivers are beyond the scope of this section.
Warning:

When a driver is loaded it is executed in the context of the emulator, shares the same memory and the same thread. This means that all operations in the driver must be non-blocking, and that any crash in the driver brings the whole emulator down. In short, be careful.

1.10.2 Sample Driver

This section describes a simple driver for accessing a postgres database using the libpq C client library. Postgres is used because it is free and open source. For information on postgres, see www.postgres.org.

The driver is synchronous, it uses the synchronous calls of the client library. This is only for simplicity, but not good, as it halts the emulator while waiting for the database. This is improved below with an asynchronous sample driver.

The code is straightforward: all communication between Erlang and the driver is done with `port_control/3`, and the driver returns data back using the `rbuf`.

An Erlang driver only exports one function: the driver entry function. This is defined with a macro, `DRIVER_INIT`, which returns a pointer to a C struct containing the entry points that are called from the emulator. The struct defines the entries that the emulator calls to call the driver, with a NULL pointer for entries that are not defined and used by the driver.

The `start` entry is called when the driver is opened as a port with `open_port/2`. Here we allocate memory for a user data structure. This user data is passed every time the emulator calls us. First we store the driver handle, as it is needed in later calls. We allocate memory for the connection handle that is used by LibPQ. We also set the port to return allocated driver binaries, by setting flag PORT_CONTROL_FLAG_BINARY, calling `set_port_control_flags`. (This is because we do not know if our data will fit in the result buffer of `control`, which has a default size, 64 bytes, set up by the emulator.)

An entry `init` is called when the driver is loaded. However, we do not use this, as it is executed only once, and we want to have the possibility of several instances of the driver.

The `stop` entry is called when the port is closed.

The `control` entry is called from the emulator when the Erlang code calls `port_control/3`, to do the actual work. We have defined a simple set of commands: `connect` to log in to the database, `disconnect` to log out, and `select` to send a SQL-query and get the result. All results are returned through `rbuf`. The library `ei` in `erl_interface` is used to encode data in binary term format. The result is returned to the emulator as binary terms, so `binary_to_term` is called in Erlang to convert the result to term form.

The code is available in `pg_sync.c` in the sample directory of erts.

The driver entry contains the functions that will be called by the emulator. In this example, only `start`, `stop`, and `control` are provided:
1.10 How to Implement a Driver

/* Driver interface declarations */
static ErlDrvData start(ErlDrvPort port, char *command);
static void stop(ErlDrvData drv_data);
static int control(ErlDrvData drv_data, unsigned int command, char *buf,
                   int len, char **rbuf, int rlen);

static ErlDrvEntry pq_driver_entry = {
    NULL, /* init */
    start,
    stop,
    NULL, /* output */
    NULL, /* ready_input */
    NULL, /* ready_output */
    "pg sync", /* the name of the driver */
    NULL, /* finish */
    NULL, /* handle */
    control,
    NULL, /* timeout */
    NULL, /* outputv */
    NULL, /* ready_async */
    NULL, /* flush */
    NULL, /* call */
    NULL /* event */
};

We have a structure to store state needed by the driver, in this case we only need to keep the database connection:

typedef struct our_data_s {
    PGconn* conn;
} our_data_t;

The control codes that we have defined are as follows:

/* Keep the following definitions in alignment with the
 * defines in erl_pq_sync.erl
 */
#define DRV_CONNECT             'C'
#define DRV_DISCONNECT          'D'
#define DRV_SELECT              'S'

This returns the driver structure. The macro DRIVER_INIT defines the only exported function. All the other functions are static, and will not be exported from the library.

/* INITIALIZATION AFTER LOADING */

/*
 * This is the init function called after this driver has been loaded.
 * It must *not* be declared static. Must return the address to
 * the driver entry.
 */
DRIVER_INIT(pq_drv)
{    return &pq_driver_entry;
}

Here some initialization is done, start is called from open_port. The data will be passed to control and stop.
How to Implement a Driver

1.10 How to Implement a Driver

```c
/* DRIVER INTERFACE */
static ErlDrvData start(ErlDrvPort port, char *command)
{
    our_data_t* data;
    data = (our_data_t*)driver_alloc(sizeof(our_data_t));
    data->conn = NULL;
    set_port_control_flags(port, PORT_CONTROL_FLAG_BINARY);
    return (ErlDrvData)data;
}
```

We call disconnect to log out from the database. (This should have been done from Erlang, but just in case.)

```c
static int do_disconnect(our_data_t* data, ei_x_buff* x);

static void stop(ErlDrvData drv_data)
{
    our_data_t* data = (our_data_t*)drv_data;
    do_disconnect(data, NULL);
    driver_free(data);
}
```

We use the binary format only to return data to the emulator; input data is a string parameter for connect and select. The returned data consists of Erlang terms.

```c
We use the binary format only to return data to the emulator; input data is a string parameter for connect and select. The returned data consists of Erlang terms.
```

The functions `get_s` and `ei_x_to_new_binary` are utilities that are used to make the code shorter. `get_s` duplicates the string and zero-terminates it, as the postgres client library wants that. `ei_x_to_new_binary` takes an `ei_x_buff` buffer, allocates a binary, and copies the data there. This binary is returned in `*rbuf`. (Notice that this binary is freed by the emulator, not by us.)

```c
static char* get_s(const char* buf, int len);
static int do_connect(const char *s, our_data_t* data, ei_x_buff* x);
static int do_select(const char* s, our_data_t* data, ei_x_buff* x);
```

```c
/* As we are operating in binary mode, the return value from control * is irrelevant, as long as it is not negative. */
static int control(ErlDrvData drv_data, unsigned int command, char *buf, int len, char **rbuf, int rlen)
{
    int r;
    ei_x_buff x;
    our_data_t* data = (our_data_t*)drv_data;
    char* s = get_s(buf, len);
    ei_x_new_with_version(&x);
    switch (command) {
        case DRV_CONNECT:    r = do_connect(s, data, &x);  break;
        case DRV_DISCONNECT: r = do_disconnect(data, &x);  break;
        case DRV_SELECT:     r = do_select(s, data, &x);   break;
        default:             r = -1;        break;
    }
    *rbuf = (char*)ei_x_to_new_binary(&x);
    ei_x_free(&x);
    driver_free(s);
    return r;
}
```

do_connect is where we log in to the database. If the connection was successful, we store the connection handle in the driver data, and return '+ok'. Otherwise, we return the error message from postgres and store NULL in the driver data.
1.10 How to Implement a Driver

```c
static int do_connect(const char *s, our_data_t* data, ei_x_buff* x)
{
    PGconn* conn = PQconnectdb(s);
    if (PQstatus(conn) != CONNECTION_OK) {
        encode_error(x, conn);
        PQfinish(conn);
        conn = NULL;
    } else {
        encode_ok(x);
    }
    data->conn = conn;
    return 0;
}
```

If we are connected (and if the connection handle is not NULL), we log out from the database. We need to check if we should encode an 'ok', as we can get here from function stop, which does not return data to the emulator:

```c
static int do_disconnect(our_data_t* data, ei_x_buff* x)
{
    if (data->conn == NULL)
        return 0;
    PQfinish(data->conn);
    data->conn = NULL;
    if (x != NULL)
        encode_ok(x);
    return 0;
}
```

We execute a query and encode the result. Encoding is done in another C module, pg_encode.c, which is also provided as sample code.

```c
static int do_select(const char* s, our_data_t* data, ei_x_buff* x)
{
    PGresult* res = PQexec(data->conn, s);
    encode_result(x, res, data->conn);
    PQclear(res);
    return 0;
}
```

Here we check the result from postgres. If it is data, we encode it as lists of lists with column data. Everything from postgres is C strings, so we use ei_x_encode_string to send the result as strings to Erlang. (The head of the list contains the column names.)
# 1.10 How to Implement a Driver

```c
void encode_result(ei_x_buff* x, PGresult* res, PGconn* conn)
{
    int row, n_rows, col, n_cols;
    switch (PQresultStatus(res)) {
    case PGRES_TUPLES_OK:
        n_rows = PQntuples(res);
        n_cols = PQnfields(res);
        ei_x_encode_tuple_header(x, 2);
        encode_ok(x);
        ei_x_encode_list_header(x, n_rows+1);
        ei_x_encode_list_header(x, n_cols);
        for (col = 0; col < n_cols; ++col) {
            ei_x_encode_string(x, PQfname(res, col));
        }
        ei_x_encode_empty_list(x);
        for (row = 0; row < n_rows; ++row) {
            ei_x_encode_list_header(x, n_cols);
            for (col = 0; col < n_cols; ++col) {
                ei_x_encode_string(x, PQgetvalue(res, row, col));
            }
        }
        ei_x_encode_empty_list(x);
    break;
    case PGRES_COMMAND_OK:
        ei_x_encode_tuple_header(x, 2);
        encode_ok(x);
        ei_x_encode_string(x, PQcmdTuples(res));
    break;
    default:
        encode_error(x, conn);
        break;
    }
}
```

## 1.10.3 Compiling and Linking the Sample Driver

The driver is to be compiled and linked to a shared library (DLL on Windows). With gcc, this is done with link flags `-shared` and `-fpic`. As we use the `ei` library, we should include it too. There are several versions of `ei`, compiled for debug or non-debug and multi-threaded or single-threaded. In the makefile for the samples, the `obj` directory is used for the `ei` library, meaning that we use the non-debug, single-threaded version.

## 1.10.4 Calling a Driver as a Port in Erlang

Before a driver can be called from Erlang, it must be loaded and opened. Loading is done using the `erl_ddll` module (the `erl_ddll` driver that loads dynamic driver is actually a driver itself). If loading is successful, the port can be opened with `open_port/2`. The port name must match the name of the shared library and the name in the driver entry structure.

When the port has been opened, the driver can be called. In the `pg_sync` example, we do not have any data from the port, only the return value from the `port_control`.

The following code is the Erlang part of the synchronous postgres driver, `pg_sync.erl`:
1.10 How to Implement a Driver

-module(pg_sync).
-define(DRV_CONNECT, 1).
-define(DRV_DISCONNECT, 2).
-define(DRV_SELECT, 3).
-export([connect/1, disconnect/1, select/2]).

connect(ConnectStr) ->
    case erl_ddll:load_driver(".", "pg_sync") of
        ok -> ok;
        {error, already_loaded} -> ok;
        E -> exit({error, E})
    end,
    Port = open_port({spawn, ?MODULE}, []),
    case binary_to_term(port_control(Port, ?DRV_CONNECT, ConnectStr)) of
        ok -> {ok, Port};
        Error -> Error
    end.

disconnect(Port) ->
    R = binary_to_term(port_control(Port, ?DRV_DISCONNECT, "")),
    port_close(Port),
    R.

select(Port, Query) ->
    binary_to_term(port_control(Port, ?DRV_SELECT, Query)).

The API is simple:

• connect/1 loads the driver, opens it, and logs on to the database, returning the Erlang port if successful.
• select/2 sends a query to the driver and returns the result.
• disconnect/1 closes the database connection and the driver. (However, it does not unload it.)

The connection string is to be a connection string for postgres.

The driver is loaded with erl_ddll:load_driver/2. If this is successful, or if it is already loaded, it is opened. This will call the start function in the driver.

We use the port_control/3 function for all calls into the driver. The result from the driver is returned immediately and converted to terms by calling binary_to_term/1. (We trust that the terms returned from the driver are well-formed, otherwise the binary_to_term calls could be contained in a catch.)

1.10.5 Sample Asynchronous Driver

Sometimes database queries can take a long time to complete, in our pg_sync driver, the emulator halts while the driver is doing its job. This is often not acceptable, as no other Erlang process gets a chance to do anything. To improve on our postgres driver, we re-implement it using the asynchronous calls in LibPQ.

The asynchronous version of the driver is in the sample files pg_async.c and pg_asyng.erl.
1.10 How to Implement a Driver

```c
/* Driver interface declarations */
static ErlDrvData start(ErlDrvPort port, char *command);
static void stop(ErlDrvData drv_data);
static int control(ErlDrvData drv_data, unsigned int command, char *buf,
                   int len, char **rbuf, int rlen);
static void ready_io(ErlDrvData drv_data, ErlDrvEvent event);

static ErlDrvEntry pq_driver_entry = {
    NULL,                     /* init */
    start,
    stop,
    NULL,                     /* output */
    ready_io,                /* ready_input */
    ready_io,                /* ready_output */
    "pg_async",                /* the name of the driver */
    NULL,                     /* finish */
    NULL,                     /* handle */
    control,
    NULL,                     /* timeout */
    NULL,                     /* outputv */
    NULL,                     /* ready_async */
    NULL,                     /* flush */
    NULL,                     /* call */
    NULL,                     /* event */
};

typedef struct our_data_t {
    PGconn* conn;
    ErlDrvPort port;
    int socket;
    int connecting;
} our_data_t;
```

Some things have changed from `pg_sync.c`: we use the entry `ready_io` for `ready_input` and `ready_output`, which is called from the emulator only when there is input to be read from the socket. (Actually, the socket is used in a `select` function inside the emulator, and when the socket is signaled, indicating there is data to read, the `ready_input` entry is called. More about this below.)

Our driver data is also extended, we keep track of the socket used for communication with postgres, and also the port, which is needed when we send data to the port with `driver_output`. We have a flag `connecting` to tell whether the driver is waiting for a connection or waiting for the result of a query. (This is needed, as the entry `ready_io` is called both when connecting and when there is a query result.)
1.10 How to Implement a Driver

```c
static int do_connect(const char *s, our_data_t* data)
{
    PGconn* conn = PQconnectStart(s);
    if (PQstatus(conn) == CONNECTION_BAD) {
        ei_x_buff x;
        ei_x_new_with_version(&x);
        encode_error(&x, conn);
        PQfinish(conn);
        conn = NULL;
        driver_output(data->port, x.buff, x.index);
        ei_x_free(&x);
    }
    PQconnectPoll(conn);
    int socket = PQsocket(conn);
    data->socket = socket;
    driver_select(data->port, (ErlDrvEvent)socket, DO_READ, 1);
    driver_select(data->port, (ErlDrvEvent)socket, DO_WRITE, 1);
    data->conn = conn;
    data->connecting = 1;
    return 0;
}
```

The `connect` function looks a bit different too. We connect using the asynchronous `PQconnectStart` function. After the connection is started, we retrieve the socket for the connection with `PQsocket`. This socket is used with the `driver_select` function to wait for connection. When the socket is ready for input or for output, the `ready_io` function is called.

Notice that we only return data (with `driver_output`) if there is an error here, otherwise we wait for the connection to be completed, in which case our `ready_io` function is called.

```c
static int do_select(const char* s, our_data_t* data)
{
    data->connecting = 0;
    PGconn* conn = data->conn;
    /* if there's an error return it now */
    if (PQsendQuery(conn, s) == 0) {
        ei_x_buff x;
        ei_x_new_with_version(&x);
        encode_error(&x, conn);
        driver_output(data->port, x.buff, x.index);
        ei_x_free(&x);
    }
    /* else wait for ready_output to get results */
    return 0;
}
```

The `do_select` function initiates a select, and returns if there is no immediate error. The result is returned when `ready_io` is called.
The `ready_io` function is called when the socket we got from postgres is ready for input or output. Here we first check if we are connecting to the database. In that case, we check connection status and return OK if the connection is successful, or error if it is not. If the connection is not yet established, we simply return; `ready_io` is called again.

If we have a result from a connect, indicated by having data in the `x` buffer, we no longer need to select on output (`ready_output`), so we remove this by calling `driver_select`.

If we are not connecting, we wait for results from a `PQsendQuery`, so we get the result and return it. The encoding is done with the same functions as in the earlier example.

Error handling is to be added here, for example, checking that the socket is still open, but this is only a simple example.

The Erlang part of the asynchronous driver consists of the sample file `pg_async.erl`. 

```c
static void ready_io(ErlDrvData drv_data, ErlDrvEvent event)
{
  PGresult* res = NULL;
  our_data_t* data = (our_data_t*)drv_data;
  PGconn* conn = data->conn;
  ei_x_buffer x;
  ei_x_new_with_version(&x);
  if (data->connecting) {
    ConnStatusType status;
    PQconnectPoll(conn);
    status = PQstatus(conn);
    if (status == CONNECTION_OK)
      encode_ok(&x);
    else if (status == CONNECTION_BAD)
      encode_error(&x, conn);
  } else {
    PQconsumeInput(conn);
    if (PQisBusy(conn))
      return;
    res = PQgetResult(conn);
    encode_result(&x, res, conn);
    PQclear(res);
    for (;;) {
      res = PQgetResult(conn);
      if (res == NULL)
        break;
      PQclear(res);
    }
  }
  if (x.index > 1) {
    driver_output(data->port, x.buff, x.index);
    if (data->connecting)
      driver_select(data->port, (ErlDrvEvent)data->socket, DO_WRITE, 0);
  }
  ei_x_free(&x);
}
```
The Erlang code is slightly different, as we do not return the result synchronously from `port_control`, instead we get it from `driver_output` as data in the message queue. The function `return_port_data` above receives data from the port. As the data is in binary format, we use `binary_to_term/1` to convert it to an Erlang term. Notice that the driver is opened in binary mode (`open_port/2` is called with option `[binary]`). This means that data sent from the driver to the emulator is sent as binaries. Without option `binary`, they would have been lists of integers.

### An Asynchronous Driver Using `driver_async`

As a final example we demonstrate the use of `driver_async`. We also use the driver term interface. The driver is written in C++. This enables us to use an algorithm from STL. We use the `next_permutation` algorithm to get the next permutation of a list of integers. For large lists (> 100,000 elements), this takes some time, so we perform this as an asynchronous task.

The asynchronous API for drivers is complicated. First, the work must be prepared. In the example, this is done in `output`. We could have used `control`, but we want some variation in the examples. In our driver, we allocate a structure that contains anything that is needed for the asynchronous task to do the work. This is done in the main emulator thread. Then the asynchronous function is called from a driver thread, separate from the main emulator thread. Notice that the driver functions are not re-entrant, so they are not to be used. Finally, after the function is completed, the driver callback `ready_async` is called from the main emulator thread, this is where we return the result to Erlang. (We cannot return the result from within the asynchronous function, as we cannot call the driver functions.)
The following code is from the sample file `next_perm.cc`. The driver entry looks like before, but also contains the callback `ready_async`.

```c++
static ErlDrvEntry next_perm_driver_entry = {
    NULL,                        /* init */
    start,
    NULL,                        /* stop */
    output,
    NULL,                        /* ready_input */
    NULL,                        /* ready_output */
    "next_perm",                 /* the name of the driver */
    NULL,                        /* finish */
    NULL,                        /* handle */
    NULL,                        /* control */
    NULL,                        /* timeout */
    NULL,                        /* outputv */
    ready_async,
    NULL,                        /* flush */
    NULL,                        /* call */
    NULL                         /* event */
};
```

The `output` function allocates the work area of the asynchronous function. As we use C++, we use a struct, and stuff the data in it. We must copy the original data, it is not valid after we have returned from the `output` function, and the `do_perm` function is called later, and from another thread. We return no data here, instead it is sent later from the `ready_async` callback.

The `async_data` is passed to the `do_perm` function. We do not use a `async_free` function (the last argument to `driver_async`), it is only used if the task is cancelled programmatically.

```c++
struct our_async_data {
    bool prev;
    vector<int> data;
    our_async_data(ErlDrvPort p, int command, const char* buf, int len);
};
our_async_data::our_async_data(ErlDrvPort p, int command,
                                const char* buf, int len)
    : prev(command == 2),
      data((int*)buf, (int*)buf + len / sizeof(int))
{
}

static void do_perm(void* async_data);
static void output(ErlDrvData drv_data, char *buf, int len)
{
    if (*buf < 1 || *buf > 2) return;
    ErlDrvPort port = reinterpret_cast<ErlDrvPort>(drv_data);
    void* async_data = new our_async_data(port, *buf, buf+1, len);
    driver_async(port, NULL, do_perm, async_data, do_free);
}
```

In the `do_perm` we do the work, operating on the structure that was allocated in `output`.

```c++
static void do_perm(void* async_data)
{
    our_async_data* d = reinterpret_cast<our_async_data*>(async_data);
    if (!d->prev)
        prev_permutation(d->data.begin(), d->data.end());
    else
        next_permutation(d->data.begin(), d->data.end());
}
```
1.10 How to Implement a Driver

In the `ready_async` function the output is sent back to the emulator. We use the driver term format instead of `ei`. This is the only way to send Erlang terms directly to a driver, without having the Erlang code to call `binary_to_term/1`. In the simple example this works well, and we do not need to use `ei` to handle the binary term format.

When the data is returned, we deallocate our data.

```c
static void ready_async(ErlDrvData drv_data, ErlDrvThreadData async_data)
{
    ErlDrvPort port = reinterpret_cast<ErlDrvPort>(drv_data);
    our_async_data* d = reinterpret_cast<our_async_data*>(async_data);
    int n = d->data.size(), result_n = n*2 + 3;
    ErlDrvTermData *result = new ErlDrvTermData[result_n], *rp = result;
    for (vector<int>::iterator i = d->data.begin(); i != d->data.end(); ++i) {
        *rp++ = ERL_DRV_INT;
        *rp++ = *i;
    }
    *rp++ = ERL_DRV_UINT;
    *rp++ = *i;
}
```

This driver is called like the others from Erlang. However, as we use `driver_output_term`, there is no need to call `binary_to_term`. The Erlang code is in the sample file `next_perm.erl`.

The input is changed into a list of integers and sent to the driver.
1.11 Inet Configuration

1.11.1 Introduction

This section describes how the Erlang runtime system is configured for IP communication. It also explains how you can configure it for your needs by a configuration file. The information is primarily intended for users with special configuration needs or problems. There is normally no need for specific settings for Erlang to function properly on a correctly IP-configured platform.

When Erlang starts up it reads the Kernel variable inetrc, which, if defined, is to specify the location and name of a user configuration file. Example:

% erl -kernel inetrc "./cfg_files/erl_inetrc"

Notice that the use of an .inetrc file, which was supported in earlier Erlang/OTP versions, is now obsolete.
A second way to specify the configuration file is to set environment variable ERL_INETRC to the full name of the file. Example (bash):

```
% export ERL_INETRC=./cfg_files/erl_inetrc
```

Notice that the Kernel variable inetrc overrides this environment variable.

If no user configuration file is specified and Erlang is started in non-distributed or short name distributed mode, Erlang uses default configuration settings and a native lookup method that works correctly under most circumstances. Erlang reads no information from system inet configuration files (such as /etc/host.conf and /etc/ nsswitch.conf) in these modes, except for /etc/resolv.conf and /etc/hosts that is read and monitored for changes on Unix platforms for the internal DNS client inet_res(3).

If Erlang is started in long name distributed mode, it needs to get the domain name from somewhere and reads system inet configuration files for this information. Any hosts and resolver information found is also recorded, but not used as long as Erlang is configured for native lookups. The information becomes useful if the lookup method is changed to 'file' or 'dns', see below.

Native lookup (system calls) is always the default resolver method. This is true for all platforms, except OSE Delta where 'file' or 'dns' is used in that priority order.

On Windows platforms, Erlang searches the system registry rather than looks for configuration files when started in long name distributed mode.

### 1.11.2 Configuration Data

Erlang records the following data in a local database if found in system inet configuration files (or system registry):

- Hostnames and host addresses
- Domain name
- Nameservers
- Search domains
- Lookup method

This data can also be specified explicitly in the user configuration file. This file is to contain lines of configuration parameters (each terminated with a full stop). Some parameters add data to the configuration (such as host and nameserver), others overwrite any previous settings (such as domain and lookup). The user configuration file is always examined last in the configuration process, making it possible for the user to override any default values or previously made settings. Call inet:get_rc() to view the state of the inet configuration database.

The valid configuration parameters are as follows:

```
{file, Format, File}.
```

- **Format** = atom()
- **File** = string()

Specify a system file that Erlang is to read configuration data from. Format tells the parser how the file is to be interpreted:

- `resolv` (Unix resolv.conf)
- `host_conf_freebsd` (FreeBSD host.conf)
- `host_conf_bsdos` (BSDOS host.conf)
- `host_conf_linux` (Linux host.conf)
- `nsswitch_conf` (Unix nsswitch.conf)
- `hosts` (Unix hosts)

File is to specify the filename with full path.
1.11 Inet Configuration

(resolv_conf, File).
File = string()

Specify a system file that Erlang is to read resolver configuration from for the internal DNS client
inet_res(3), and monitor for changes, even if it does not exist. The path must be absolute.

This can override the configuration parameters nameserver and search depending on the contents of the
specified file. They can also change any time in the future reflecting the file contents.

If the file is specified as an empty string "", no file is read or monitored in the future. This emulates the old
behavior of not configuring the DNS client when the node is started in short name distributed mode.

If this parameter is not specified, it defaults to /etc/resolv.conf unless environment variable
ERL_INET_ETC_DIR is set, which defines the directory for this file to some maybe other than /etc.

(hosts_file, File).
File = string()

Specify a system file that Erlang is to read resolver configuration from for the internal hosts file resolver, and
monitor for changes, even if it does not exist. The path must be absolute.

These host entries are searched after all added with \{file, hosts, File\} above or \{host, IP, Alias\} below when lookup option file is used.

If the file is specified as an empty string "", no file is read or monitored in the future. This emulates the old
behavior of not configuring the DNS client when the node is started in short name distributed mode.

If this parameter is not specified, it defaults to /etc/hosts unless environment variable
ERL_INET_ETC_DIR is set, which defines the directory for this file to some maybe other than /etc.

(registry, Type).
Type = atom()

Specify a system registry that Erlang is to read configuration data from. win32 is the only valid option.

(host, IP, Aliases).
IP = tuple()
Aliases = [string()]

Add host entry to the hosts table.

domain, Domain).
Domain = string()
Set domain name.

(nameserver, IP [,Port]).
IP = tuple()
Port = integer()

Add address (and port, if other than default) of the primary nameserver to use for inet_res(3).

(alt_nameserver, IP [,Port]).
IP = tuple()
Port = integer()

Add address (and port, if other than default) of the secondary nameserver for inet_res(3).
1.11 Inet Configuration

{search, Domains}.
   Domains = [string()]
   Add search domains for inet_res(3).

{lookup, Methods}.
   Methods = [atom()]
   Specify lookup methods and in which order to try them. The valid methods are as follows:
   • native (use system calls)
   • file (use host data retrieved from system configuration files and/or the user configuration file)
   • dns (use the Erlang DNS client inet_res(3) for nameserver queries)

   The lookup method string tries to parse the hostname as an IPv4 or IPv6 string and return the resulting IP address. It is automatically tried first when native is not in the Methods list. To skip it in this case, the pseudo lookup method nostring can be inserted anywhere in the Methods list.

{cache_size, Size}.
   Size = integer()
   Set the resolver cache size for dns lookups. native lookups are not cached. Defaults to 100 DNS records.

{cache_refresh, Time}.
   Time = integer()
   Set how often (in milliseconds) the resolver cache for inet_res(3) is refreshed (that is, expired DNS records are deleted). Defaults to 1 hour.

{timeout, Time}.
   Time = integer()
   Set the time to wait until retry (in milliseconds) for DNS queries made by inet_res(3). Defaults to 2 seconds.

{retry, N}.
   N = integer()
   Set the number of DNS queries inet_res(3) will try before giving up. Defaults to 3.

{servfail_retry_timeout, Time}.
   Time = non_neg_integer()
   After all name servers have been tried, there is a timeout before the name servers are tried again. This is to prevent the server from answering the query with what's in the servfail cache, inet_res(3). Defaults to 1500 milliseconds.

{inet6, Bool}.
   Bool = true | false
   Tells the DNS client inet_res(3) to look up IPv6 addresses. Defaults to false.

{usevc, Bool}.
   Bool = true | false
   Tells the DNS client inet_res(3) to use TCP (Virtual Circuit) instead of UDP. Defaults to false.

{edns, Version}.
   Version = false | 0
1.11 Inet Configuration

Sets the EDNS version that \texttt{inet_res(3)} will use. The only allowed version is zero. Defaults to \texttt{false}, which means not to use EDNS.

\begin{verbatim}
{udp_payload_size, Size}.
N = integer()
Sets the allowed UDP payload size \texttt{inet_res(3)} will advertise in EDNS queries. Also sets the limit when the DNS query will be deemed too large for UDP forcing a TCP query instead; this is not entirely correct, as the advertised UDP payload size of the individual nameserver is what is to be used, but this simple strategy will do until a more intelligent (probing, caching) algorithm needs to be implemented. Default to 1280, which stems from the standard Ethernet MTU size.
\end{verbatim}

\begin{verbatim}
{udp, Module}.
Module = atom()
Tell Erlang to use another primitive UDP module than \texttt{inet_udp}.
\end{verbatim}

\begin{verbatim}
{tcp, Module}.
Module = atom()
Tell Erlang to use another primitive TCP module than \texttt{inet_tcp}.
\end{verbatim}

\begin{verbatim}
clear_hosts.
    Clear the hosts table.
\end{verbatim}

\begin{verbatim}
clear_ns.
    Clear the list of recorded nameservers (primary and secondary).
\end{verbatim}

\begin{verbatim}
clear_search.
    Clear the list of search domains.
\end{verbatim}

1.11.3 User Configuration Example

Assume that a user does not want Erlang to use the native lookup method, but wants Erlang to read all information necessary from start and use that for resolving names and addresses. If lookup fails, Erlang is to request the data from a nameserver (using the Erlang DNS client, set to use EDNS allowing larger responses). The resolver configuration is updated when its configuration file changes. Also, DNS records are never to be cached. The user configuration file (in this example named \texttt{erl_inetrc}, stored in directory \texttt{./cfg_files}) can then look as follows (Unix):

\begin{verbatim}
%% -- ERLANG INET CONFIGURATION FILE --
%% read the hosts file
{file, hosts, "/etc/hosts"}.
%% add a particular host
{host, \{134,138,177,105\}, ["finwe"]}.
%% do not monitor the hosts file
{hosts_file, "}

%% read and monitor nameserver config from here
{resolv_conf, "/usr/local/etc/resolv.conf"}.
%% enable EDNS
{edns, 0}.
%% disable caching
{cache_size, 0}.
%% specify lookup method
{lookup, [file, dns]}.
\end{verbatim}

And Erlang can, for example, be started as follows:

\begin{verbatim}
% erl -sname my_node -kernel inetrc "./cfg_files/erl_inetrc"
\end{verbatim}
1.12 External Term Format

1.12.1 Introduction

The external term format is mainly used in the distribution mechanism of Erlang.

As Erlang has a fixed number of types, there is no need for a programmer to define a specification for the external format used within some application. All Erlang terms have an external representation and the interpretation of the different terms is application-specific.

In Erlang the BIF `erlang:term_to_binary/1,2` is used to convert a term into the external format. To convert binary data encoding to a term, the BIF `erlang:binary_to_term/1` is used.

The distribution does this implicitly when sending messages across node boundaries.

The overall format of the term format is as follows:

<p>| | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>N</td>
<td></td>
</tr>
<tr>
<td>131</td>
<td>Tag</td>
<td>Data</td>
<td></td>
</tr>
</tbody>
</table>

Table 12.1: Term Format

**Note:**

When messages are passed between connected nodes and a distribution header is used, the first byte containing the version number (131) is omitted from the terms that follow the distribution header. This is because the version number is implied by the version number in the distribution header.

The compressed term format is as follows:

<p>| | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>4</td>
<td>N</td>
</tr>
<tr>
<td>131</td>
<td>80</td>
<td>UncompressedSize</td>
<td>Zlib-compressedData</td>
</tr>
</tbody>
</table>

Table 12.2: Compressed Term Format

Uncompressed size (unsigned 32-bit integer in big-endian byte order) is the size of the data before it was compressed. The compressed data has the following format when it has been expanded:

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Uncompressed Size</td>
</tr>
<tr>
<td>Tag</td>
<td>Data</td>
</tr>
</tbody>
</table>

Table 12.3: Compressed Data Format when Expanded

1.12.2 Encoding atoms

As from ERTS 9.0 (OTP 20), atoms may contain any Unicode characters.
Atoms sent over node distribution are always encoded in UTF-8 using either \texttt{ATOM\_UTF8\_EXT}, \texttt{SMALL\_ATOM\_UTF8\_EXT} or \texttt{ATOM\_CACHE\_REF}.

Atoms encoded with \texttt{erlang:term\_to\_binary/1,2} or \texttt{erlang:term\_to\_iovec/1,2} are by default still using the old deprecated Latin-1 format \texttt{ATOM\_EXT} for atoms that only contain Latin-1 characters (Unicode code points 0-255). Atoms with higher code points will be encoded in UTF-8 using either \texttt{ATOM\_UTF8\_EXT} or \texttt{SMALL\_ATOM\_UTF8\_EXT}.

The maximum number of allowed characters in an atom is 255. In the UTF-8 case, each character can need 4 bytes to be encoded.

### 1.12.3 Distribution Header

The distribution header is sent by the erlang distribution to carry metadata about the coming control message and potential payload. It is primarily used to handle the atom cache in the Erlang distribution. Since OTP-22 it is also used to fragment large distribution messages into multiple smaller fragments. For more information about how the distribution uses the distribution header, see the documentation of the protocol between connected nodes in the distribution protocol documentation.

Any \texttt{ATOM\_CACHE\_REF} entries with corresponding \texttt{AtomCacheReferenceIndex} in terms encoded on the external format following a distribution header refer to the atom cache references made in the distribution header. The range is $0 \leq \texttt{AtomCacheReferenceIndex} < 255$, that is, at most 255 different atom cache references from the following terms can be made.

#### Normal Distribution Header

The non-fragmented distribution header format is as follows:

<p>| | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>2+1</td>
</tr>
<tr>
<td>131</td>
<td>68</td>
<td>NumberOfAtomCacheRefs</td>
<td>Flags</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>AtomCacheRefs</td>
</tr>
</tbody>
</table>

Table 12.4: Normal Distribution Header Format

Flags consist of \texttt{NumberOfAtomCacheRefs/2+1} bytes, unless \texttt{NumberOfAtomCacheRefs} is 0. If \texttt{NumberOfAtomCacheRefs} is 0, \texttt{Flags} and \texttt{AtomCacheRefs} are omitted. Each atom cache reference has a half byte flag field. Flags corresponding to a specific \texttt{AtomCacheReferenceIndex} are located in flag byte number \texttt{AtomCacheReferenceIndex}/2. Flag byte 0 is the first byte after the \texttt{NumberOfAtomCacheRefs} byte. Flags for an even \texttt{AtomCacheReferenceIndex} are located in the least significant half byte and flags for an odd \texttt{AtomCacheReferenceIndex} are located in the most significant half byte.

The flag field of an atom cache reference has the following format:

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1 bit</td>
<td>3 bits</td>
</tr>
<tr>
<td>NewCacheEntryFlag</td>
<td>SegmentIndex</td>
</tr>
</tbody>
</table>

Table 12.5:

The most significant bit is the \texttt{NewCacheEntryFlag}. If set, the corresponding cache reference is new. The three least significant bits are the \texttt{SegmentIndex} of the corresponding atom cache entry. An atom cache consists of 8 segments, each of size 256, that is, an atom cache can contain 2048 entries.
Another half byte flag field is located along with flag fields for atom cache references. When
NumberOfAtomCacheRefs is even, this half byte is the least significant half byte of the byte that follows the
atom cache references. When NumberOfAtomCacheRefs is odd, this half byte is the most significant half byte
of the last byte of the atom cache references (on the wire, it will appear before the last cache reference). It has the
following format:

<table>
<thead>
<tr>
<th>3 bits</th>
<th>1 bit</th>
</tr>
</thead>
<tbody>
<tr>
<td>CurrentlyUnused</td>
<td>LongAtoms</td>
</tr>
</tbody>
</table>

Table 12.6:

The least significant bit in that half byte is flag LongAtoms. If it is set, 2 bytes are used for atom lengths instead
of 1 byte in the distribution header.

After the Flags field follow the AtomCacheRefs. The first AtomCacheRef is the one corresponding to
AtomCacheReferenceIndex 0. Higher indices follow in sequence up to index NumberOfAtomCacheRefs - 1.

If the NewCacheEntryFlag for the next AtomCacheRef has been set, a NewAtomCacheRef on the following
format follows:

<table>
<thead>
<tr>
<th>1</th>
<th>1</th>
<th>2</th>
<th>Length</th>
</tr>
</thead>
<tbody>
<tr>
<td>InternalSegmentIndex</td>
<td>Length</td>
<td>AtomText</td>
<td></td>
</tr>
</tbody>
</table>

Table 12.7:

InternalSegmentIndex together with the SegmentIndex completely identify the location of an atom
cache entry in the atom cache. Length is the number of bytes that AtomText consists of. Length is a 2
byte big-endian integer if flag LongAtoms has been set, otherwise a 1 byte integer. When distribution flag
DFLAG_UTF8_ATOMS has been exchanged between both nodes in the distribution handshake, characters in
AtomText are encoded in UTF-8, otherwise in Latin-1. The following CachedAtomRefs with the same
SegmentIndex and InternalSegmentIndex as this NewAtomCacheRef refer to this atom until a new
NewAtomCacheRef with the same SegmentIndex and InternalSegmentIndex appear.

For more information on encoding of atoms, see the section on UTF-8 encoded atoms above.

If the NewCacheEntryFlag for the next AtomCacheRef has not been set, a CachedAtomRef on the following
format follows:

<table>
<thead>
<tr>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td>InternalSegmentIndex</td>
</tr>
</tbody>
</table>

Table 12.8:

InternalSegmentIndex together with the SegmentIndex identify the location of the atom cache entry in the
atom cache. The atom corresponding to this CachedAtomRef is the latest NewAtomCacheRef preceding this
CachedAtomRef in another previously passed distribution header.
Distribution Header for fragmented messages

Messages sent between Erlang nodes can sometimes be quite large. Since OTP-22 it is possible to split large messages into smaller fragments in order to allow smaller messages to be interleaved between large messages. It is only the message part of each distributed message that may be split using fragmentation. Therefore it is recommended to use the PAYLOAD control messages introduced in OTP-22.

Fragmented distribution messages are only used if the receiving node signals that it supports them via the DFLAG_FRAGMENTS distribution flag.

A process must complete the sending of a fragmented message before it can start sending any other message on the same distribution channel.

The start of a sequence of fragmented messages looks like this:

<table>
<thead>
<tr>
<th></th>
<th></th>
<th>8</th>
<th>8</th>
<th>NumberOfAtomCacheRefs/2+1</th>
<th>N</th>
</tr>
</thead>
<tbody>
<tr>
<td>131</td>
<td>69</td>
<td>SequenceId</td>
<td>FragmentId</td>
<td>NumberOfAtomCacheRefs</td>
<td>Flags</td>
</tr>
</tbody>
</table>

Table 12.9: Starting Fragmented Distribution Header Format

The continuation of a sequence of fragmented messages looks like this:

<table>
<thead>
<tr>
<th></th>
<th></th>
<th>8</th>
<th>8</th>
</tr>
</thead>
<tbody>
<tr>
<td>131</td>
<td>70</td>
<td>SequenceId</td>
<td>FragmentId</td>
</tr>
</tbody>
</table>

Table 12.10: Continuing Fragmented Distribution Header Format

The starting distribution header is very similar to a non-fragmented distribution header. The atom cache works the same as for normal distribution header and is the same for the entire sequence. The additional fields added are the sequence id and fragment id.

Sequence ID

The sequence id is used to uniquely identify a fragmented message sent from one process to another on the same distributed connection. This is used to identify which sequence a fragment is a part of as the same process can be in the process of receiving multiple sequences at the same time.

As one process can only be sending one fragmented message at once, it can be convenient to use the local PID as the sequence id.

Fragments ID

The Fragment ID is used to number the fragments in a sequence. The id starts at the total number of fragments and then decrements to 1 (which is the final fragment). So if a sequence consists of 3 fragments the fragment id in the starting header will be 3, and then fragments 2 and 1 are sent.

The fragments must be delivered in the correct order, so if an unordered distribution carrier is used, they must be ordered before delivered to the Erlang run-time.

Example:

As an example, let say that we want to send `{call, <0.245.2>, {set_get_state, <<0:1024>>}} to registered process `reg` using a fragment size of 128. To send this message we need a distribution header, atom cache
updates, the control message (which would be \{6, <0.245.2>, [], reg\} in this case) and finally the actual message. This would all be encoded into:

```
131,69,0,0,2,168,0,0,5,83,0,0,0,0,0,0,2,            %% Header with seq and frag id
5,4,137,9,10,5,236,3,114,101,103,9,4,99,97,108,108,
104,4,97,6,103,82,0,0,0,85,0,0,0,0,0,2,82,1,82,2,
104,3,82,3,103,82,0,0,0,0,245,0,0,0,2,2,           %% Atom cache updates
104,2,82,4,109,0,0,0,128,0,0,0,0,0,0,0,0,0,0,0,0,0,0,
0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,
0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,
0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,
0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,
0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,
0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,
131,70,0,0,2,168,0,0,5,83,0,0,0,0,0,0,0,0,0,0,0,1,
0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,
0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,
131,69,                   %% Start fragment header
0,0,2,168,0,0,5,83,       %% The sequence ID
0,0,0,0,0,0,0,2,           %% The fragment ID
5,4,137,9, % 5 atoms and their flags
10,5, % The already cached atom ids
236,3,114,101,103, % The atom 'reg'
9,4,99,97,108,108, % The atom 'call'
```

Let us break that apart into its components. First we have the distribution header tags together with the sequence id
and a fragment id of 2.

```
Then we have the updates to the atom cache:
```

```
5,4,137,9, % 5 atoms and their flags
10,5, % The already cached atom ids
236,3,114,101,103, % The atom 'reg'
9,4,99,97,108,108, % The atom 'call'
```

The first byte says that we have 5 atoms that are part of the cache. Then follows three bytes that are the atom cache ref flags. Each of the flags uses 4 bits so they are a bit hard to read in decimal byte form. In binary half-byte form they look like this:

```
0000, 0100, 1000, 1001
```

As the high bit of the first two atoms in the cache are not set we know that they are already in the cache, so they do not have to be sent again (this is the node name of the receiving and sending node). Then follows the atoms that have to be sent, together with their segment ids.

Then the listing of the atoms comes, starting with 10 and 5 which are the atom refs of the already cached atoms. Then the new atoms are sent.

When the atom cache is setup correctly the control message is sent.

```
104,4,97,6,103,82,0,0,0,85,0,0,0,0,2,82,1,82,2,
```

Note that up until here it is not allowed to fragments the message. The entire atom cache and control message has to be part of the starting fragment. After the control message the payload of the message is sent using 128 bytes:

```
104,3,82,3,103,82,0,0,0,245,0,0,0,2,2,           %% Atom cache updates
104,2,82,4,109,0,0,0,128,0,0,0,0,0,0,0,0,0,0,0,0,0,
0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,
0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,
0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,
0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,
0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,
0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,
104,3,82,3,103,82,0,0,0,245,0,0,0,2,2,
104,2,82,4,109,0,0,0,128,0,0,0,0,0,0,0,0,0,0,0,0,
0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,
0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,
0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,
0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,
0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,
0,0,0,0,0,0,0,0,0,0,0,0,0,0,0
```

Since the payload is larger than 128-bytes it is split into two fragments. The second fragment does not have any atom cache update instructions so it is a lot simpler:
Note:
The fragment size of 128 is only used as an example. Any fragments size may be used when sending fragmented messages.

1.12.4 ATOM_CACHE_REF

<table>
<thead>
<tr>
<th>1</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td>82</td>
<td>AtomCacheReferenceIndex</td>
</tr>
</tbody>
</table>

Table 12.11: ATOM_CACHE_REF

Refers to the atom with AtomCacheReferenceIndex in the distribution header.

1.12.5 SMALL_INTEGER_EXT

<table>
<thead>
<tr>
<th>1</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td>97</td>
<td>Int</td>
</tr>
</tbody>
</table>

Table 12.12: SMALL_INTEGER_EXT

Unsigned 8-bit integer.

1.12.6 INTEGER_EXT

<table>
<thead>
<tr>
<th>1</th>
<th>4</th>
</tr>
</thead>
<tbody>
<tr>
<td>98</td>
<td>Int</td>
</tr>
</tbody>
</table>

Table 12.13: INTEGER_EXT

Signed 32-bit integer in big-endian format.

1.12.7 FLOAT_EXT

<table>
<thead>
<tr>
<th>1</th>
<th>31</th>
</tr>
</thead>
<tbody>
<tr>
<td>99</td>
<td>Float string</td>
</tr>
</tbody>
</table>

Table 12.14: FLOAT_EXT
1.12 External Term Format

A finite float (i.e. not inf, -inf or NaN) is stored in string format. The format used in sprintf to format the float is "%.20e" (there are more bytes allocated than necessary). To unpack the float, use sscanf with format "%lf".

This term is used in minor version 0 of the external format; it has been superseded by NEW_FLOAT_EXT.

### 1.12.8 PORT_EXT

<table>
<thead>
<tr>
<th></th>
<th>N</th>
<th>4</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td>102</td>
<td>Node</td>
<td>ID</td>
<td>Creation</td>
</tr>
</tbody>
</table>

Table 12.15: PORT_EXT

Same as NEW_PORT_EXT except the Creation field is only one byte and only two bits are significant, the rest are to be 0.

### 1.12.9 NEW_PORT_EXT

<table>
<thead>
<tr>
<th></th>
<th>N</th>
<th>4</th>
<th>4</th>
</tr>
</thead>
<tbody>
<tr>
<td>89</td>
<td>Node</td>
<td>ID</td>
<td>Creation</td>
</tr>
</tbody>
</table>

Table 12.16: NEW_PORT_EXT

Same as V4_PORT_EXT except the ID field is only four bytes. Only 28 bits are significant; the rest are to be 0.

NEW_PORT_EXT was introduced in OTP 19, but only to be decoded and echoed back. Not encoded for local ports.

In OTP 23 distribution flag DFLAG_BIG_CREATION became mandatory. All ports are now encoded using NEW_PORT_EXT, even external ports received as PORT_EXT from older nodes.

### 1.12.10 V4_PORT_EXT

<table>
<thead>
<tr>
<th></th>
<th>N</th>
<th>8</th>
<th>4</th>
</tr>
</thead>
<tbody>
<tr>
<td>120</td>
<td>Node</td>
<td>ID</td>
<td>Creation</td>
</tr>
</tbody>
</table>

Table 12.17: V4_PORT_EXT

Encodes a port identifier (obtained from erlang:open_port/2). Node is the originating node, encoded as an atom. ID is a 64-bit big endian unsigned integer. The Creation works just like in NEW_PID_EXT. Port operations are not allowed across node boundaries.

In OTP 26 distribution flag DFLAG_V4_NC as well as V4_PORT_EXT became mandatory accepting full 64-bit ports to be decoded and echoed back.

### 1.12.11 PID_EXT

<table>
<thead>
<tr>
<th></th>
<th>N</th>
<th>4</th>
<th>4</th>
<th>1</th>
</tr>
</thead>
</table>

88 | Ericsson AB. All Rights Reserved.: Erlang Run-Time System Application (ERTS)
1.12 External Term Format

<table>
<thead>
<tr>
<th>103</th>
<th>Node</th>
<th>ID</th>
<th>Serial</th>
<th>Creation</th>
</tr>
</thead>
</table>

Table 12.18: PID_EXT

Same as NEW_PID_EXT except the Creation field is only one byte and only two bits are significant, the rest are to be 0.

### 1.12.12 NEW_PID_EXT

<table>
<thead>
<tr>
<th>1</th>
<th>N</th>
<th>4</th>
<th>4</th>
<th>4</th>
</tr>
</thead>
<tbody>
<tr>
<td>88</td>
<td>Node</td>
<td>ID</td>
<td>Serial</td>
<td>Creation</td>
</tr>
</tbody>
</table>

Table 12.19: NEW_PID_EXT

Encodes an Erlang process identifier object.

- **Node**
  - The name of the originating node, encoded as an atom.

- **ID**
  - A 32-bit big endian unsigned integer.

- **Serial**
  - A 32-bit big endian unsigned integer.

- **Creation**
  - A 32-bit big endian unsigned integer. All identifiers originating from the same node incarnation must have identical Creation values. This makes it possible to separate identifiers from old (crashed) nodes from a new one. The value zero is reserved and must be avoided for normal operations.

NEW_PID_EXT was introduced in OTP 19, but only to be decoded and echoed back. Not encoded for local processes.

In OTP 23 distribution flag DFLAG_BIG_CREATION became mandatory. All pids are now encoded using NEW_PID_EXT, even external pids received as PID_EXT from older nodes.

In OTP 26 distribution flag DFLAG_V4_NC became mandatory accepting full 64-bit pids to be decoded and echoed back.

### 1.12.13 SMALL_TUPLE_EXT

<table>
<thead>
<tr>
<th>1</th>
<th>N</th>
</tr>
</thead>
<tbody>
<tr>
<td>104</td>
<td>Arity</td>
</tr>
</tbody>
</table>

Table 12.20: SMALL_TUPLE_EXT

Encodes a tuple. The Arity field is an unsigned byte that determines how many elements that follows in section Elements.
1.12.14 LARGE_TUPLE_EXT

<table>
<thead>
<tr>
<th>1</th>
<th>4</th>
<th>N</th>
</tr>
</thead>
<tbody>
<tr>
<td>105</td>
<td>Arity</td>
<td>Elements</td>
</tr>
</tbody>
</table>

Table 12.21: LARGE_TUPLE_EXT

Same as SMALL_TUPLE_EXT except that Arity is an unsigned 4 byte integer in big-endian format.

1.12.15 MAP_EXT

<table>
<thead>
<tr>
<th>1</th>
<th>4</th>
<th>N</th>
</tr>
</thead>
<tbody>
<tr>
<td>116</td>
<td>Arity</td>
<td>Pairs</td>
</tr>
</tbody>
</table>

Table 12.22: MAP_EXT

Encodes a map. The Arity field is an unsigned 4 byte integer in big-endian format that determines the number of key-value pairs in the map. Key and value pairs (K_i => V_i) are encoded in section Pairs in the following order: K_1, V_1, K_2, V_2,..., K_n, V_n. Duplicate keys are not allowed within the same map.

As from Erlang/OTP 17.0

1.12.16 NIL_EXT

<table>
<thead>
<tr>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td>106</td>
</tr>
</tbody>
</table>

Table 12.23: NIL_EXT

The representation for an empty list, that is, the Erlang syntax [].

1.12.17 STRING_EXT

<table>
<thead>
<tr>
<th>1</th>
<th>2</th>
<th>Len</th>
</tr>
</thead>
<tbody>
<tr>
<td>107</td>
<td>Length</td>
<td>Characters</td>
</tr>
</tbody>
</table>

Table 12.24: STRING_EXT

String does not have a corresponding Erlang representation, but is an optimization for sending lists of bytes (integer in the range 0-255) more efficiently over the distribution. As field Length is an unsigned 2 byte integer (big-endian), implementations must ensure that lists longer than 65535 elements are encoded as LIST_EXT.
1.12.18 LIST_EXT

<table>
<thead>
<tr>
<th></th>
<th>4</th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Length</td>
<td>Elements</td>
<td>Tail</td>
</tr>
<tr>
<td>108</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table 12.25: LIST_EXT

Length is the number of elements that follows in section Elements. Tail is the final tail of the list; it is NIL_EXT for a proper list, but can be any type if the list is improper (for example, [a | b]).

1.12.19 BINARY_EXT

<table>
<thead>
<tr>
<th></th>
<th>4</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Len</td>
<td>Data</td>
</tr>
<tr>
<td>109</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table 12.26: BINARY_EXT

Binaries are generated with bit syntax expression or with erlang:list_to_binary/1, erlang:term_to_binary/1, or as input from binary ports. The Len length field is an unsigned 4 byte integer (big-endian).

1.12.20 SMALL_BIG_EXT

<table>
<thead>
<tr>
<th></th>
<th>1</th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Sign</td>
<td></td>
<td></td>
</tr>
<tr>
<td>110</td>
<td>n</td>
<td>d(0) ... d(n-1)</td>
<td></td>
</tr>
</tbody>
</table>

Table 12.27: SMALL_BIG_EXT

Bignums are stored in unary form with a Sign byte, that is, 0 if the bignum is positive and 1 if it is negative. The digits are stored with the least significant byte stored first. To calculate the integer, the following formula can be used:

\[ B = 256 \]

\[ (d0*B^0 + d1*B^1 + d2*B^2 + ... d(N-1)*B^(n-1)) \]

1.12.21 LARGE_BIG_EXT

<table>
<thead>
<tr>
<th></th>
<th>4</th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Sign</td>
<td></td>
<td></td>
</tr>
<tr>
<td>111</td>
<td>n</td>
<td>d(0) ... d(n-1)</td>
<td></td>
</tr>
</tbody>
</table>

Table 12.28: LARGE_BIG_EXT

Same as SMALL_BIG_EXT except that the length field is an unsigned 4 byte integer.
1.12.22 REFERENCE_EXT (deprecated)

<p>| | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>N</td>
<td>4</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>101</td>
<td>Node</td>
<td>ID</td>
<td>Creation</td>
<td></td>
</tr>
</tbody>
</table>

Table 12.29: REFERENCE_EXT

The same as NEW_REFERENCE_EXT except ID is only one word (Len = 1).

1.12.23 NEW_REFERENCE_EXT

<p>| | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2</td>
<td>N</td>
<td>1</td>
<td>N'</td>
</tr>
<tr>
<td>114</td>
<td>Len</td>
<td>Node</td>
<td>Creation</td>
<td>ID ...</td>
</tr>
</tbody>
</table>

Table 12.30: NEW_REFERENCE_EXT

The same as NEWER_REFERENCE_EXT except:

ID
- In the first word (4 bytes) of ID, only 18 bits are significant, the rest must be 0.

Creation
- Only one byte long and only two bits are significant, the rest must be 0.

1.12.24 NEWER_REFERENCE_EXT

<p>| | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2</td>
<td>N</td>
<td>4</td>
<td>N'</td>
</tr>
<tr>
<td>90</td>
<td>Len</td>
<td>Node</td>
<td>Creation</td>
<td>ID ...</td>
</tr>
</tbody>
</table>

Table 12.31: NEWER_REFERENCE_EXT

Encodes a reference term generated with erlang:make_ref/0.

Node
- The name of the originating node, encoded as an atom.

Len
- A 16-bit big endian unsigned integer not larger than 5.

ID
- A sequence of Len big-endian unsigned integers (4 bytes each, so \( N' = 4 \times \text{Len} \)), but is to be regarded as uninterpreted data.

Creation
- Works just like in NEW_PID_EXT.
NEWER_REFERENCE_EXT was introduced in OTP 19, but only to be decoded and echoed back. Not encoded for local references.

In OTP 23 distribution flag DFLAG_BIG_CREATION became mandatory. All references are now encoded using NEWER_REFERENCE_EXT, even external references received as NEW_REFERENCE_EXT from older nodes.

In OTP 26 distribution flag DFLAG_V4_NC became mandatory. References now can contain up to 5 ID words.

### 1.12.25 FUN_EXT (removed)

<table>
<thead>
<tr>
<th>1</th>
<th>4</th>
<th>N1</th>
<th>N2</th>
<th>N3</th>
<th>N4</th>
<th>N5</th>
</tr>
</thead>
<tbody>
<tr>
<td>117</td>
<td></td>
<td>NumFree</td>
<td>Pid</td>
<td>Module</td>
<td>Index</td>
<td>Uniq</td>
</tr>
</tbody>
</table>

Table 12.32: FUN_EXT

Not emitted since OTP R8, and not decoded since OTP 23.

### 1.12.26 NEW_FUN_EXT

<table>
<thead>
<tr>
<th>1</th>
<th>4</th>
<th>1</th>
<th>16</th>
<th>4</th>
<th>4</th>
<th>N1</th>
<th>N2</th>
<th>N3</th>
<th>N4</th>
<th>N5</th>
</tr>
</thead>
<tbody>
<tr>
<td>112</td>
<td></td>
<td>Size</td>
<td>Arity</td>
<td>Uniq</td>
<td>Index</td>
<td>NumFree</td>
<td>Module</td>
<td>OldIndex</td>
<td>OldUniq</td>
<td>Pid</td>
</tr>
</tbody>
</table>

Table 12.33: NEW_FUN_EXT

This is the encoding of internal funs: `fun F/A` and `fun(Arg1,..) -> ... end`.

**Size**

The total number of bytes, including field `Size`.

**Arity**

The arity of the function implementing the fun.

**Uniq**

The 16 bytes MD5 of the significant parts of the Beam file.

**Index**

An index number. Each fun within a module has an unique index. `Index` is stored in big-endian byte order.

**NumFree**

The number of free variables.

**Module**

The module that the fun is implemented in, encoded as an atom.

**OldIndex**

An integer encoded using SMALL_INTEGER_EXT or INTEGER_EXT. Is typically a small index into the module's fun table.
1.12 External Term Format

OldUniq
An integer encoded using SMALL_INTEGER_EXT or INTEGER_EXT. Uniq is the hash value of the parse tree for the fun.

Pid
A process identifier as in PID_EXT. Represents the process in which the fun was created.

Free vars
NumFree number of terms, each one encoded according to its type.

1.12.27 EXPORT_EXT

<table>
<thead>
<tr>
<th>I</th>
<th>N1</th>
<th>N2</th>
<th>N3</th>
</tr>
</thead>
<tbody>
<tr>
<td>113</td>
<td>Module</td>
<td>Function</td>
<td>Arity</td>
</tr>
</tbody>
</table>

Table 12.34: EXPORT_EXT

This term is the encoding for external funs: fun M:F/A.
Module and Function are encoded as atoms.
Arity is an integer encoded using SMALL_INTEGER_EXT.

1.12.28 BIT_BINARY_EXT

<table>
<thead>
<tr>
<th>I</th>
<th>4</th>
<th>1</th>
<th>Len</th>
</tr>
</thead>
<tbody>
<tr>
<td>77</td>
<td>Len</td>
<td>Bits</td>
<td>Data</td>
</tr>
</tbody>
</table>

Table 12.35: BIT_BINARY_EXT

This term represents a bitstring whose length in bits does not have to be a multiple of 8. The Len field is an unsigned 4 byte integer (big-endian). The Bits field is the number of bits (1-8) that are used in the last byte in the data field, counting from the most significant bit to the least significant.

1.12.29 NEW_FLOAT_EXT

<table>
<thead>
<tr>
<th>I</th>
<th>8</th>
</tr>
</thead>
<tbody>
<tr>
<td>70</td>
<td>IEEE float</td>
</tr>
</tbody>
</table>

Table 12.36: NEW_FLOAT_EXT

A finite float (i.e. not inf, -inf or NaN) is stored as 8 bytes in big-endian IEEE format.
This term is used in minor version 1 of the external format.
1.12.30 ATOM_UTF8_EXT

An atom is stored with a 2 byte unsigned length in big-endian order, followed by \( \text{Len} \) bytes containing the \( \text{AtomName} \) encoded in UTF-8.

For more information, see the section on encoding atoms in the beginning of this page.

1.12.31 SMALL_ATOM_UTF8_EXT

An atom is stored with a 1 byte unsigned length, followed by \( \text{Len} \) bytes containing the \( \text{AtomName} \) encoded in UTF-8. Longer atoms encoded in UTF-8 can be represented using ATOM_UTF8_EXT.

For more information, see the section on encoding atoms in the beginning of this page.

1.12.32 ATOM_EXT (deprecated)

An atom is stored with a 2 byte unsigned length in big-endian order, followed by \( \text{Len} \) numbers of 8-bit Latin-1 characters that forms the \( \text{AtomName} \). The maximum allowed value for \( \text{Len} \) is 255.

1.12.33 SMALL_ATOM_EXT (deprecated)

An atom is stored with a 1 byte unsigned length, followed by \( \text{Len} \) numbers of 8-bit Latin-1 characters that forms the \( \text{AtomName} \).
### 1.13 Distribution Protocol

This description is far from complete. It will be updated if the protocol is updated. However, the protocols, both from Erlang nodes to the Erlang Port Mapper Daemon (EPMD) and between Erlang nodes are stable since many years.

The distribution protocol can be divided into four parts:

- Low-level socket connection (1)
- Handshake, interchange node name, and authenticate (2)
- Authentication (done by net_kernel(3)) (3)
- Connected (4)

A node fetches the port number of another node through the EPMD (at the other host) to initiate a connection request. For each host, where a distributed Erlang node is running, also an EPMD is to be running. The EPMD can be started explicitly or automatically as a result of the Erlang node startup.

By default the EPMD listens on port 4369.

(3) and (4) above are performed at the same level but the net_kernel disconnects the other node if it communicates using an invalid cookie (after 1 second).

The integers in all multibyte fields are in big-endian order.

---

### Note:

| SMALL_ATOM_EXT was introduced in ERTS 5.7.2 and require an exchange of distribution flag DFLAG_SMALL_ATOM_TAGS in the distribution handshake. |

<table>
<thead>
<tr>
<th>LOCAL_EXT</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>...</td>
</tr>
<tr>
<td>121</td>
<td>...</td>
</tr>
</tbody>
</table>

Table 12.41: LOCAL_EXT

Marks that this is encoded on an alternative local external term format intended to only be decoded by a specific local decoder. The bytes following from here on may contain any unspecified type of encoding of terms. It is the responsibility of the user to only attempt to decode terms on the local external term format which has been produced by a matching encoder.

This tag is used by the Erlang runtime system upon encoding the local external term format when the local option is passed to term_to_binary/2, but can be used by other encoders as well providing similar functionality. The Erlang runtime system adds a hash immediately following the LOCAL_EXT tag which is verified on decoding in order to verify that encoder and decoder match which might be a good practice. This will very likely catch mistakes made by users, but is not guaranteed to, and is not intended to, prevent decoding of an intentionally forged encoding on the local external term format.

LOCAL_EXT was introduced in OTP 26.0.
Warning:

The Erlang Distribution protocol is not by itself secure and does not aim to be so. In order to get secure distribution the distributed nodes should be configured to use distribution over tls. See the Using SSL for Erlang Distribution User's Guide for details on how to setup a secure distributed node.

1.13.1 EPMD Protocol

The requests served by the EPMD are summarized in the following figure.

![Diagram of EPMD Requests](image)

Each request \(*_\text{REQ}\) is preceded by a 2 byte length field. Thus, the overall request format is as follows:

| 2 | \(n\) |
Register a Node in EPMD

When a distributed node is started it registers itself in the EPMD. The message ALIVE2_REQ described below is sent from the node to the EPMD. The response from the EPMD is ALIVE2_X_RESP (or ALIVE2_RESP).

<table>
<thead>
<tr>
<th>Length</th>
<th>Request</th>
</tr>
</thead>
<tbody>
<tr>
<td>120</td>
<td>PortNo NodeType Protocol HighestVersion Nlen NodeName Elen Extra</td>
</tr>
</tbody>
</table>

Table 13.2: ALIVE2_REQ (120)

PortNo

The port number on which the node accept connection requests.

NodeType

77 = normal Erlang node, 72 = hidden node (C-node), ...

Protocol

0 = TCP/IPv4, ...

HighestVersion

The highest distribution protocol version this node can handle. The value in OTP 23 and later is 6. Older nodes only support version 5.

LowestVersion

The lowest distribution version that this node can handle. The value in OTP 25 and later is 6 as support for connections to nodes older than OTP 23 has been dropped.

Nlen

The length (in bytes) of field NodeName.

NodeName

The node name as an UTF-8 encoded string of Nlen bytes.

Elen

The length of field Extra.

Extra

Extra field of Elen bytes.

The connection created to the EPMD must be kept as long as the node is a distributed node. When the connection is closed, the node is automatically unregistered from the EPMD.

The response message is either ALIVE2_X_RESP or ALIVE2_RESP depending on distribution version. If both the node and EPMD support distribution version 6 then the response is ALIVE2_X_RESP otherwise it is the older ALIVE2_RESP:

| 1 | 1 | 4 |
1.13 Distribution Protocol

Table 13.3: ALIVE2_X_RESP (118) with 32 bit creation

<table>
<thead>
<tr>
<th>118</th>
<th>Result</th>
<th>Creation</th>
</tr>
</thead>
</table>

Table 13.4: ALIVE2_RESP (121) with 16-bit creation

Result = 0 -> ok, result > 0 -> error.

Unregister a Node from EPMD

A node unregisters itself from the EPMD by closing the TCP connection to EPMD established when the node was registered.

Get the Distribution Port of Another Node

When one node wants to connect to another node it starts with a PORT_PLEASE2_REQ request to the EPMD on the host where the node resides to get the distribution port that the node listens to.

Table 13.5: PORT_PLEASE2_REQ (122)

<table>
<thead>
<tr>
<th>1</th>
<th>N</th>
</tr>
</thead>
<tbody>
<tr>
<td>122</td>
<td>NodeName</td>
</tr>
</tbody>
</table>

where N = Length - 1.

Table 13.6: PORT2_RESP (119) Response Indicating Error, Result > 0

<table>
<thead>
<tr>
<th>1</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td>119</td>
<td>Result</td>
</tr>
</tbody>
</table>

or

Table 13.7: PORT2_RESP, Result = 0

<table>
<thead>
<tr>
<th>1</th>
<th>1</th>
<th>2</th>
<th>1</th>
<th>1</th>
<th>2</th>
<th>2</th>
<th>2</th>
<th>Nlen</th>
<th>2</th>
<th>Elen</th>
</tr>
</thead>
<tbody>
<tr>
<td>119</td>
<td>Result</td>
<td>PortNo</td>
<td>NodeType</td>
<td>HighestVersion</td>
<td>LowestVersion</td>
<td>len</td>
<td>NodeName</td>
<td>Elen</td>
<td>&gt;Extra</td>
<td></td>
</tr>
</tbody>
</table>

If Result > 0, the packet only consists of [119, Result].
The EPMD closes the socket when it has sent the information.
Get All Registered Names from EPMD

This request is used through the Erlang function `net_adm:names/1,2`. A TCP connection is opened to the EPMD and this request is sent.

![Table 13.8: NAMES_REQ (110)]

Table 13.8: NAMES_REQ (110)

The response for a NAMES_REQ is as follows:

![Table 13.9: NAMES_RESP](image)

Table 13.9: NAMES_RESP

NodeInfo is a string written for each active node. When all NodeInfo has been written the connection is closed by the EPMD.

NodeInfo is, as expressed in Erlang:

```
io:format("name ~ts at port ~p-n", [NodeName, Port]).
```

Dump All Data from EPMD

This request is not really used, it is to be regarded as a debug feature.

![Table 13.10: DUMP_REQ](image)

Table 13.10: DUMP_REQ

The response for a DUMP_REQ is as follows:

![Table 13.11: DUMP_RESP](image)

Table 13.11: DUMP_RESP

NodeInfo is a string written for each node kept in the EPMD. When all NodeInfo has been written the connection is closed by the EPMD.

NodeInfo is, as expressed in Erlang:

```
io:format("active name ~ts at port ~p, fd = ~p-n", [NodeName, Port, Fd]).
```
or

```erlang
io:format("old/unused name ~ts at port ~p, fd = ~p ~n", [NodeName, Port, Fd]).
```

**Kill EPMD**

This request kills the running EPMD. It is almost never used.

<table>
<thead>
<tr>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td>107</td>
</tr>
</tbody>
</table>

Table 13.12: KILL_REQ

The response for a KILL_REQ is as follows:

<table>
<thead>
<tr>
<th>2</th>
</tr>
</thead>
<tbody>
<tr>
<td>OKString</td>
</tr>
</tbody>
</table>

Table 13.13: KILL_RESP

where OKString is "OK".

**STOP_REQ (Not Used)**

<table>
<thead>
<tr>
<th>1</th>
<th>n</th>
</tr>
</thead>
<tbody>
<tr>
<td>115</td>
<td>NodeName</td>
</tr>
</tbody>
</table>

Table 13.14: STOP_REQ

where n = Length - 1.

The response for a STOP_REQ is as follows:

<table>
<thead>
<tr>
<th>7</th>
</tr>
</thead>
<tbody>
<tr>
<td>OKString</td>
</tr>
</tbody>
</table>

Table 13.15: STOP_RESP

where OKString is "STOPPED".

A negative response can look as follows:

| 7 |
1.13 Distribution Protocol

Table 13.16: STOP_NOTOK_RESP

where NOKString is "NOEXIST".

1.13.2 Distribution Handshake

This section describes the distribution handshake protocol used between nodes to establishing a connection. The protocol was introduced in Erlang/OTP R6 and amended in OTP 23. From OTP 25 support for the older protocol was dropped. Therefore an OTP 25 node can not connect to nodes older than OTP 23. This documentation only describes the part of the protocol used by OTP 25.

Note:

A bug introduced in OTP 25.0 may cause OTP 25 nodes to reject connection attempts from OTP 23 and 24 nodes that are not using epmd to gain version information about the remote node. This is fixed in OTP 25.3.

General

The TCP/IP distribution uses a handshake that expects a connection-based protocol, that is, the protocol does not include any authentication after the handshake procedure.

This is not entirely safe, as it is vulnerable against takeover attacks, but it is a tradeoff between fair safety and performance.

The cookies are never sent in cleartext and the handshake procedure expects the client (called A) to be the first one to prove that it can generate a sufficient digest. The digest is generated with the MD5 message digest algorithm and the challenges are expected to be random numbers.

Definitions

A challenge is a 32-bit integer in big-endian order. Below the function gen_challenge() returns a random 32-bit integer used as a challenge.

A digest is a (16 bytes) MD5 hash of the challenge (as text) concatenated with the cookie (as text). Below, the function gen_digest(Challenge, Cookie) generates a digest as described above.

An out_cookie is the cookie used in outgoing communication to a certain node, so that A's out_cookie for B is to correspond with B's in_cookie for A and conversely. A's out_cookie for B and A's in_cookie for B need not be the same. Below the function out_cookie(Node) returns the current node's out_cookie for Node.

An in_cookie is the expected cookie to be used by another node when communicating with us, so that A's in_cookie for B corresponds with B's out_cookie for A. Below the function in_cookie(Node) returns the current node's in_cookie for Node.

The cookies are text strings that can be viewed as passwords.

Every message in the handshake starts with a 16-bit big-endian integer, which contains the message length (not counting the two initial bytes). In Erlang this corresponds to option {packet, 2} in gen_tcp(3). Notice that after the handshake, the distribution switches to 4 byte packet headers.

The Handshake in Detail

Imagine two nodes, A that initiates the handshake and B that accepts the connection.
1.13 Distribution Protocol

1) **connect/accept**

   A connects to B through TCP/IP and B accepts the connection.

2) **send_name/receive_name**

   A sends an initial identification to B, which receives the message. The message can have two different formats which looks as follows (the packet headers are removed):

<table>
<thead>
<tr>
<th>1</th>
<th>2</th>
<th>4</th>
<th>Nlen</th>
</tr>
</thead>
<tbody>
<tr>
<td>'n'</td>
<td>Version=5</td>
<td>Flags</td>
<td>Name</td>
</tr>
</tbody>
</table>

   **Table 17: Old send_name ('n') for protocol version 5**

<table>
<thead>
<tr>
<th>1</th>
<th>8</th>
<th>4</th>
<th>2</th>
<th>Nlen</th>
</tr>
</thead>
<tbody>
<tr>
<td>'N'</td>
<td>Flags</td>
<td>Creation</td>
<td>Nlen</td>
<td>Name</td>
</tr>
</tbody>
</table>

   **Table 18: New send_name ('N') for protocol version 6**

   The old send_name format is only sent from OTP 23 and 24 nodes that are not using epmd and therefore do not know if the remote node only supports protocol version 5. The Version is a 16-bit big endian integer and **must** always have the value 5 (even though node A supports version 6). Flags are the capability flags of node A in 32-bit big endian. The flag bit DFLAG_HANDSHAKE_23 must be set (as node A must supports version 6). **Name** is the full node name of A, as a string of bytes (the packet length denotes how long it is).

   The new send_name is sent to nodes known to support version 6. Flags are the capability flags of node A in 64-bit big endian. The flag bit DFLAG_HANDSHAKE_23 must always be set. **Creation** is the node incarnation identifier used by node A to create its pids, ports and references. **Name** is the full node name of A, as a string of bytes. **Nlen** is the byte length of the node name in 16-bit big endian. Any extra data after the node **Name** must be accepted and ignored.

   The **Name** must be just the host name (without @) when DFLAG_NAME_ME is set.

3) **recv_status/send_status**

   B sends a status message to A, which indicates if the connection is allowed.

<table>
<thead>
<tr>
<th>1</th>
<th>Slen</th>
</tr>
</thead>
<tbody>
<tr>
<td>'s'</td>
<td>Status</td>
</tr>
</tbody>
</table>

   **Table 19: The format of the status message**

   's' is the message tag. **Status** is the status code as a string (not null terminated). The following status codes are defined:

   **ok**

   The handshake will continue.
1.13 Distribution Protocol

ok_simultaneous
The handshake will continue, but A is informed that B has another ongoing connection attempt that will be shut down (simultaneous connect where A’s name is greater than B’s name, compared literally).
nok
The handshake will not continue, as B already has an ongoing handshake, which it itself has initiated (simultaneous connect where B’s name is greater than A’s).
not_allowed
The connection is disallowed for some (unspecified) security reason.
alive
A connection to the node is already active, which either means that node A is confused or that the TCP connection breakdown of a previous node with this name has not yet reached node B. See step 3B below.
named:
The handshake will continue, but A requested a dynamic node name by setting flag DFLAG_NAME_ME. The dynamic node name of A is supplied at the end of the status message from B. The host name of A which was sent as Name in send_name will be used by node B to generate the full dynamic node name.

<p>| | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Slen=6</td>
<td>2</td>
<td>Nlen</td>
</tr>
<tr>
<td>'s'</td>
<td>Status='named:'</td>
<td>Nlen</td>
<td>Name</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Creation</td>
</tr>
</tbody>
</table>

Table .20: The format of the ‘named:’ status message

Name is the full dynamic node name of A, as a string of bytes. Nlen is the byte length of the node name in 16-bit big endian. Creation is the incarnation identifier of node A generated by node B. Any extra data after the node Creation must be accepted and ignored.

3B) send_status/recv_status
If status was alive, node A answers with another status message containing either true, which means that the connection is to continue (the old connection from this node is broken), or false, which means that the connection is to be closed (the connection attempt was a mistake).

4) recv_challenge/send_challenge
If the status was ok or ok_simultaneous, the handshake continues with B sending A another message, the challenge. The challenge contains the same type of information as the “name” message initially sent from A to B, plus a 32-bit challenge:

<p>| | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>8</td>
<td>4</td>
<td>4</td>
<td>2</td>
<td>Nlen</td>
</tr>
<tr>
<td>'N'</td>
<td>Flags</td>
<td>Challenge</td>
<td>Creation</td>
<td>Nlen</td>
<td>Name</td>
</tr>
</tbody>
</table>

Table .21: The new challenge message format (version 6)

Challenge is a 32-bit big-endian integer. The other fields are node B’s flags, creation and full node name, similar to the send_name message. Any extra data after the node Name must be accepted and ignored.
4B) send_complement/recv_complement

The complement message, from A to B, is only sent if node A initially sent an old name message. It contains complementary information missing in the initial old name message from node A.

<table>
<thead>
<tr>
<th>1</th>
<th>4</th>
<th>4</th>
</tr>
</thead>
<tbody>
<tr>
<td>'c'</td>
<td>FlagsHigh</td>
<td>Creation</td>
</tr>
</tbody>
</table>

Table 22: The complement message

FlagsHigh are the high capability flags (bit 33-64) of node A as a 32-bit big endian integer. Creation is the incarnation identifier of node A.

5) send_challenge_reply/recv_challenge_reply

Now A has generated a digest and its own challenge. Those are sent together in a package to B:

<table>
<thead>
<tr>
<th>1</th>
<th>4</th>
<th>16</th>
</tr>
</thead>
<tbody>
<tr>
<td>'r'</td>
<td>Challenge</td>
<td>Digest</td>
</tr>
</tbody>
</table>

Table 23: The challenge_reply message

Challenge is A’s challenge for B to handle. Digest is the MD5 digest that A constructed from the challenge B sent in the previous step.

6) recv_challenge_ack/send_challenge_ack

B checks that the digest received from A is correct and generates a digest from the challenge received from A. The digest is then sent to A. The message is as follows:

<table>
<thead>
<tr>
<th>1</th>
<th>16</th>
</tr>
</thead>
<tbody>
<tr>
<td>'a'</td>
<td>Digest</td>
</tr>
</tbody>
</table>

Table 24: The challenge_ack message

Digest is the digest calculated by B for A’s challenge.

7) check

A checks the digest from B and the connection is up.
Semigraphic View

A (initiator)                                      B (acceptor)
TCP connect ------------------------------------> TCP accept
send_name -------------------------------------> recv_name
recv_status (if status was 'alive'
  send_status - - - - - - - - - - - - - - - - - -> recv_status)
  (ChB)                             ChB = gen_challenge()
recv_challenge
  (if old send_name
    send_complement - - - - - - - - - - - - - - - -> recv_complement)
  ChA = gen_challenge(),
  OCA = out_cookie(B),
  DiA = gen_digest(ChB, OCA)
  (ChA, DiA)
send_challenge_reply --------------------------->
  recv_challenge_reply
    ICB = in_cookie(A),
    check:
    DiA == gen_digest (ChB, ICB)?
    - if OK:
      OCB = out_cookie(A),
      DiB = gen_digest (ChA, OCB)
  (DiB)                              send_challenge_ack
recv_challenge_ack
  ICA = in_cookie(B),
  check:
  DiB == gen_digest(ChA, ICA)?
  - if OK:
    DONE
  - else:
    CLOSE

Distribution Flags

Early in the distribution handshake the two participating nodes exchange capability flags. This is done in order to
determine how the communication between the two nodes should be performed. The intersection of the capabilities
presented by the two nodes defines the capabilities that will be used. The following capability flags are defined:

-define(DFLAG_PUBLISHED,16#1).
  The node is to be published and part of the global namespace.
-define(DFLAG_ATOM_CACHE,16#2).
  The node implements an atom cache (obsolete).
-define(DFLAG_EXTENDED_REFERENCES,16#4).

The node implements extended (3 × 32 bits) references. This flag is mandatory. If not present, the connection is refused.

-define(DFLAG_DIST_MONITOR,16#8).

The node implements distributed process monitoring.

-define(DFLAG_FUN_TAGS,16#10).

The node uses separate tags for funs (lambdas) in the distribution protocol. This flag is mandatory. If not present, the connection is refused.

-define(DFLAG_DIST_MONITOR_NAME,16#20).

The node implements distributed named process monitoring.

-define(DFLAG_HIDDEN_ATOM_CACHE,16#40).

The (hidden) node implements atom cache (obsolete).

-define(DFLAG_NEW_FUN_TAGS,16#80).

The node understands the new fun tag. This flag is mandatory. If not present, the connection is refused.

-define(DFLAG_EXTENDED_PIDS_PORTS,16#100).

The node can handle extended pids and ports. This flag is mandatory. If not present, the connection is refused.

-define(DFLAG_EXPORT_PTR_TAG,16#200).

The node understands the export ptr tag. This flag is mandatory. If not present, the connection is refused.

-define(DFLAG_BIT_BINARY_EXT,16#400).

The node understands the bit binary tag. This flag is mandatory. If not present, the connection is refused.

-define(DFLAG_NEW_FLOATS,16#800).

The node understands the new float tag. This flag is mandatory. If not present, the connection is refused.

-define(DFLAG_UNICODE_IO,16#1000).

-define(DFLAG_DIST_HDR_ATOM_CACHE,16#2000).

The node implements atom cache in distribution header.

-define(DFLAG_SMALL_ATOM_TAGS, 16#4000).

The node understands the small atom tag.

-define(DFLAG_UTF8_ATOMS, 16#10000).

The node understands UTF-8 atoms encoded with ATOM_UTF8_EXT and SMALL_ATOM_UTF8_EXT. This flag is mandatory. If not present, the connection is refused.

-define(DFLAG_MAP_TAG, 16#20000).

The node understands the map tag MAP_EXT. This flag is mandatory. If not present, the connection is refused.

-define(DFLAG_BIG_CREATION, 16#40000).

The node understands big node creation tags NEW_PID_EXT, NEW_PORT_EXT and NEWER_REFERENCE_EXT. This flag is mandatory. If not present, the connection is refused.

-define(DFLAG_SEND_SENDER, 16#80000).

Use the SEND_SENDER control message instead of the SEND control message and use the SEND_SENDER_TT control message instead of the SEND_TT control message.
1.13 Distribution Protocol

-define(DFLAG_BIG_SEQTRACE_LABELS, 16#100000).
   The node understands any term as the seqtrace label.
-define(DFLAG_EXIT_PAYLOAD, 16#400000).
   Use the PAYLOAD_EXIT, PAYLOAD_EXIT_TT, PAYLOAD_EXIT2, PAYLOAD_EXIT2_TT and
   PAYLOAD_MONITOR_P_EXIT control messages instead of the non-PAYLOAD variants.
-define(DFLAG_FRAGMENTS, 16#800000).
   Use fragmented distribution messages to send large messages.
-define(DFLAG_HANDSHAKE_23, 16#1000000).
   The node supports the new connection setup handshake (version 6) introduced in OTP 23. This flag is mandatory
   (from OTP 25). If not present, the connection is refused.
-define(DFLAG_UNLINK_ID, 16#2000000).
   Use the new link protocol.

Note:
This flag is mandatory as of OTP 26.

-define(DFLAG_MANDATORY_25_DIGEST, (1 bsl 36)).
   The node supports all capabilities that are mandatory in OTP 25. Introduced in OTP 25.

Note:
This flag will become mandatory in OTP 27.

-define(DFLAG_SPAWN, (1 bsl 32)).
   Set if the SPAWN_REQUEST, SPAWN_REQUEST_TT, SPAWN_REPLY, SPAWN_REPLY_TT control messages
   are supported.
-define(DFLAG_NAME_ME, (1 bsl 33)).
   Dynamic node name. This is not a capability but rather used as a request from the connecting node to receive its
   node name from the accepting node as part of the handshake.
-define(DFLAG_V4_NC, (1 bsl 34)).
   The node accepts a larger amount of data in pids, ports and references (node container types version 4). In the pid
   case full 32-bit ID and Serial fields in NEW_PID_EXT, in the port case a 64-bit integer in V4_PORT_EXT,
   and in the reference case up to 5 32-bit ID words are now accepted in NEWER_REFERENCE_EXT. This flag was
   introduced in OTP 24 and became mandatory in OTP 26.
-define(DFLAG_ALIAS, (1 bsl 35)).
   The node supports process alias and can by this handle the ALIAS_SEND and ALIAS_SEND_TT control
   messages. Introduced in OTP 24.

There is also function dist_util:strict_order_flags/0 returning all flags (bitwise or-ed together)
   corresponding to features that require strict ordering of data over distribution channels.

1.13.3 Protocol between Connected Nodes

Since ERTS 5.7.2 (OTP R13B) the runtime system passes a distribution flag in the handshake stage that enables the use
   of a distribution header on all messages passed. Messages passed between nodes have in this case the following format:
Distribution Protocol

<table>
<thead>
<tr>
<th>4</th>
<th>d</th>
<th>n</th>
<th>m</th>
</tr>
</thead>
<tbody>
<tr>
<td>Length</td>
<td>DistributionHeader</td>
<td>ControlMessage</td>
<td>Message</td>
</tr>
</tbody>
</table>

Table 13.25: Format of Messages Passed between Nodes (as from ERTS 5.7.2 (OTP R13B))

Length
Equal to \(d + n + m\).

DistributionHeader
Distribution header describing the atom cache and fragmented distribution messages.

ControlMessage
A tuple passed using the external format of Erlang.

Message
The message sent to another node using the '!\)' or the reason for a EXIT, EXIT2 or DOWN signal using the external term format.

Notice that the version number is omitted from the terms that follow a distribution header.

Nodes with an ERTS version earlier than 5.7.2 (OTP R13B) does not pass the distribution flag that enables the distribution header. Messages passed between nodes have in this case the following format:

<table>
<thead>
<tr>
<th>4</th>
<th>1</th>
<th>n</th>
<th>m</th>
</tr>
</thead>
<tbody>
<tr>
<td>Length</td>
<td>Type</td>
<td>ControlMessage</td>
<td>Message</td>
</tr>
</tbody>
</table>

Table 13.26: Format of Messages Passed between Nodes (before ERTS 5.7.2 (OTP R13B))

Length
Equal to \(1 + n + m\).

Type
Equal to 112 (pass through).

ControlMessage
A tuple passed using the external format of Erlang.

Message
The message sent to another node using the '!\)' (in external format). Notice that Message is only passed in combination with a ControlMessage encoding a send ('!\)').

The ControlMessage is a tuple, where the first element indicates which distributed operation it encodes:

LINK
{1, FromPid, ToPid}
This signal is sent by FromPid in order to create a link between FromPid and ToPid.

SEND
{2, Unused, ToPid}
1.13 Distribution Protocol

Followed by Message.
Unused is kept for backward compatibility.

EXIT

{3, FromPid, ToPid, Reason}
This signal is sent when a link has been broken

UNLINK (obsolete)

{4, FromPid, ToPid}

Warning:
This signal is obsolete and not supported as of OTP 26. For more information see the documentation of the link protocol.

NODE_LINK

{5}

REG_SEND

{6, FromPid, Unused, ToName}
Followed by Message.
Unused is kept for backward compatibility.

GROUP_LEADER

{7, FromPid, ToPid}

EXIT2

{8, FromPid, ToPid, Reason}
This signal is sent by a call to the erlang:exit/2 bif

SEND_TT

{12, Unused, ToPid, TraceToken}
Followed by Message.
Unused is kept for backward compatibility.

EXIT_TT

{13, FromPid, ToPid, TraceToken, Reason}

REG_SEND_TT

{16, FromPid, Unused, ToName, TraceToken}
Followed by Message.
Unused is kept for backward compatibility.

EXIT2_TT

{18, FromPid, ToPid, TraceToken, Reason}

MONITOR_P

{19, FromPid, ToProc, Ref}, where FromPid = monitoring process and ToProc = monitored process pid or name (atom)
DEMONTOR_P

{20, FromPid, ToProc, Ref}, where FromPid = monitoring process and ToProc = monitored process
pid or name (atom)

We include FromPid just in case we want to trace this.

MONITOR_P_EXIT

{21, FromProc, ToPid, Ref, Reason}, where FromProc = monitored process pid or name (atom),
ToPid = monitoring process, and Reason = exit reason for the monitored process

New Ctrlmessages for Erlang/OTP 21

SEND_SENDER

{22, FromPid, ToPid}
Followed by Message.

This control message replaces the SEND control message and will be sent when the distribution flag
DFLAG_SEND_SENDER has been negotiated in the connection setup handshake.

Note:

Messages encoded before the connection has been set up may still use the SEND control message. However,
once a SEND_SENDER or SEND_SENDER_TT control message has been sent, no more SEND control
messages will be sent in the same direction on the connection.

SEND_SENDER_TT

{23, FromPid, ToPid, TraceToken}
Followed by Message.

This control message replaces the SEND_TT control message and will be sent when the distribution flag
DFLAG_SEND_SENDER has been negotiated in the connection setup handshake.

Note:

Messages encoded before the connection has been set up may still use the SEND_TT control message.
However, once a SEND_SENDER or SEND_SENDER_TT control message has been sent, no more SEND_TT
control messages will be sent in the same direction on the connection.

New Ctrlmessages for Erlang/OTP 22

Note:

Messages encoded before the connection has been set up may still use the non-PAYLOAD variant. However, once
a PAYLOAD control message has been sent, no more non-PAYLOAD control messages will be sent in the same
direction on the connection.

PAYLOAD_EXIT

{24, FromPid, ToPid}
Followed by Reason.

This control message replaces the EXIT control message and will be sent when the distribution flag
DFLAG_EXIT_PAYLOAD has been negotiated in the connection setup handshake.
1.13 Distribution Protocol

PAYLOAD_EXIT_TT
   (25, FromPid, ToPid, TraceToken)
   Followed by Reason.
   This control message replaces the EXIT_TT control message and will be sent when the distribution flag DFLAG_EXIT_PAYLOAD has been negotiated in the connection setup handshake.

PAYLOAD_EXIT2
   (26, FromPid, ToPid)
   Followed by Reason.
   This control message replaces the EXIT2 control message and will be sent when the distribution flag DFLAG_EXIT_PAYLOAD has been negotiated in the connection setup handshake.

PAYLOAD_EXIT2_TT
   (27, FromPid, ToPid, TraceToken)
   Followed by Reason.
   This control message replaces the EXIT2_TT control message and will be sent when the distribution flag DFLAG_EXIT_PAYLOAD has been negotiated in the connection setup handshake.

PAYLOAD_MONITOR_P_EXIT
   (28, FromProc, ToPid, Ref)
   Followed by Reason.
   This control message replaces the MONITOR_P_EXIT control message and will be sent when the distribution flag DFLAG_EXIT_PAYLOAD has been negotiated in the connection setup handshake.

New Controlmessages for Erlang/OTP 23

SPAWN_REQUEST
   (29, ReqId, From, GroupLeader, {Module, Function, Arity}, OptList)
   Followed by ArgList.
   This signal is sent by the spawn_request() BIF.

ReqId :: reference()
   Request identifier. Also used as monitor reference in case the monitor option has been passed.

From :: pid()
   Process identifier of the process making the request. That is, the parent process to be.

GroupLeader :: pid()
   Process identifier of the group leader of the newly created process.

{Module :: atom(), Function :: atom(), Arity :: integer() >= 0}
   Entry point for the new process.

OptList :: [term()]
   A proper list of spawn options to use when spawning.

ArgList :: [term()]
   A proper list of arguments to use in the call to the entry point.
   Only supported when the DFLAG_SPAWN distribution flag has been passed.
1.13 Distribution Protocol

**SPAWN_REQUEST_TT**

{30, ReqId, From, GroupLeader, {Module, Function, Arity}, OptList, Token}
Followed by ArgList.
Same as SPAWN_REQUEST, but also with a sequential trace Token.
Only supported when the DFLAG_SPAWN distribution flag has been passed.

**SPAWN_REPLY**

{31, ReqId, To, Flags, Result}
This signal is sent as a reply to a process previously sending a SPAWN_REQUEST signal.

- **ReqId** :: reference()
  Request identifier. Also used as monitor reference in case the monitor option has been passed.

- **To** :: pid()
  Process identifier of the process making the spawn request.

- **Flags** :: integer() >= 0
  A bit flag field of bit flags bitwise or:ed together. Currently the following flags are defined:

  1. A link between To and Result was set up on the node where Result resides.

  2. A monitor from To to Result was set up on the node where Result resides.

- **Result** :: pid() | atom()
  Result of the operation. If Result is a process identifier, the operation succeeded and the process identifier is the identifier of the newly created process. If Result is an atom, the operation failed and the atom identifies failure reason.

Only supported when the DFLAG_SPAWN distribution flag has been passed.

**SPAWN_REPLY_TT**

{32, ReqId, To, Flags, Result, Token}
Same as SPAWN_REPLY, but also with a sequential trace Token.
Only supported when the DFLAG_SPAWN distribution flag has been passed.

**UNLINK_ID**

{35, Id, FromPid, ToPid}
This signal is sent by FromPid in order to remove a link between FromPid and ToPid. This unlink signal replaces the UNLINK signal. Besides process identifiers of the sender and receiver the UNLINK_ID signal also contains an integer identifier Id. Valid range of Id is [1, (1 bsl 64) - 1]. Id is to be passed back to the sender by the receiver in an UNLINK_ID_ACK signal. Id must uniquely identify the UNLINK_ID signal among all not yet acknowledged UNLINK_ID signals from FromPid to ToPid.
This signal is part of the new link protocol which became mandatory as of OTP 26.

**UNLINK_ID_ACK**

{36, Id, FromPid, ToPid}
1.13 Distribution Protocol

An unlink acknowledgement signal. This signal is sent as an acknowledgement of the reception of an UNLINK_ID signal. The Id element should be the same Id as present in the UNLINK_ID signal. FromPid identifies the sender of the UNLINK_ID_ACK signal and ToPid identifies the sender of the UNLINK_ID signal.

This signal is part of the new link protocol which became mandatory as of OTP 26.

New Ctrlmessages for Erlang/OTP 24

ALIAS_SEND

{33, FromPid, Alias}

Followed by Message.

This control message is used when sending the message Message to the process identified by the process alias Alias. Nodes that can handle this control message sets the distribution flag DFLAG_ALIAS in the connection setup handshake.

ALIAS_SEND_TT

{34, FromPid, Alias, Token}

Followed by Message.

Same as ALIAS_SEND, but also with a sequential trace Token.

Link Protocol

The new link protocol introduced in OTP 23.3 became mandatory as of OTP 26. As of OTP 26, OTP nodes will therefor refuse to connect to nodes that do not indicate that they support the new link protocol using the DFLAG_UNLINK_ID distribution flag.

The new link protocol introduced two new signals, UNLINK_ID and UNLINK_ID_ACK, which replaced the old UNLINK signal. The old LINK signal is still sent in order to set up a link, but handled differently upon reception.

In order to set up a link, a LINK signal is sent, from the process initiating the operation, to the process that it wants to link to. In order to remove a link, an UNLINK_ID signal is sent, from the process initiating the operation, to the linked process. The receiver of an UNLINK_ID signal responds with an UNLINK_ID_ACK signal. Upon reception of an UNLINK_ID signal, the corresponding UNLINK_ID_ACK signal must be sent before any other signals are sent to the sender of the UNLINK_ID signal. Together with the signal ordering guarantee of Erlang this makes it possible for the sender of the UNLINK_ID signal to know the order of other signals which is essential for the protocol. The UNLINK_ID_ACK signal should contain the same Id as the Id contained in the UNLINK_ID signal being acknowledged.

Processes also need to maintain process local information about links. The state of this process local information is changed when the signals above are sent and received. This process local information also determines if a signal should be sent when a process calls link/1 or unlink/1. A LINK signal is only sent if there does not currently exist an active link between the processes according to the process local information and an UNLINK_ID signal is only sent if there currently exists an active link between the processes according to the process local information.

The process local information about a link contains:

Pid
Process identifier of the linked process.

Active Flag
If set, the link is active and the process will react on incoming exit signals issued due to the link. If not set, the link is inactive and incoming exit signals, issued due to the link, will be ignored. That is, the processes are considered as not linked.

Unlink Id
Identifier of an outstanding unlink operation. That is, an unlink operation that has not yet been acknowledged. This information is only used when the active flag is not set.
A process is only considered linked to another process if it has process local information about the link containing the process identifier of the other process and with the active flag set.

The process local information about a link is updated as follows:

A **LINK** signal is sent

- Link information is created if not already existing. The active flag is set, and unlink id is cleared. That is, if we had an outstanding unlink operation we will ignore the result of that operation and enable the link.

A **LINK** signal is received

- If no link information already exists, it is created, the active flag is set and unlink id is cleared. If the link information already exists, the signal is silently ignored, regardless of whether the active flag is set or not. That is, if we have an outstanding unlink operation we will **not** activate the link. In this scenario, the sender of the **LINK** signal has not yet sent an **UNLINK_ID_ACK** signal corresponding to our **UNLINK_ID** signal which means that it will receive our **UNLINK_ID** signal after it sent its **LINK** signal. This in turn means that both processes in the end will agree that there is no link between them.

An **UNLINK_ID** signal is sent

- Link information already exists and the active flag is set (otherwise the signal would not be sent). The active flag is unset, and the unlink id of the signal is saved in the link information.

An **UNLINK_ID** signal is received

- If the active flag is set, information about the link is removed. If the active flag is not set (that is, we have an outstanding unlink operation), the information about the link is left unchanged.

An **UNLINK_ID_ACK** signal is sent

- This is done when an **UNLINK_ID** signal is received and causes no further changes of the link information.

An **UNLINK_ID_ACK** signal is received

- If information about the link exists, the active flag is not set, and the unlink id in the link information equals the Id in the signal, the link information is removed; otherwise, the signal is ignored.

When a process receives an exit signal due to a link, the process will first react to the exit signal if the link is active and then remove the process local information about the link.

In case the connection is lost between two nodes, exit signals with exit reason **noconnection** are sent to all processes with links over the connection. This will cause all process local information about links over the connection to be removed.

Exactly the same link protocol is also used internally on an Erlang node. The signals however have different formats since they do not have to be sent over the wire.
2 Reference Manual
atomics

Erlang module

This module provides a set of functions to do atomic operations towards mutable atomic variables. The implementation utilizes only atomic hardware instructions without any software level locking, which makes it very efficient for concurrent access. The atomics are organized into arrays with the following semantics:

• Atomics are 64 bit integers.
• Atomics can be represented as either signed or unsigned.
• Atomics wrap around at overflow and underflow operations.
• All operations guarantee atomicity. No intermediate results can be seen. The result of one mutation can only be the input to one following mutation.
• All atomic operations are mutually ordered. If atomic \texttt{B} is updated \texttt{after} atomic \texttt{A}, then that is how it will appear to any concurrent readers. No one can read the new value of \texttt{B} and then read the old value of \texttt{A}.
• Indexes into atomic arrays are one-based. An atomic array of arity \texttt{N} contains \texttt{N} atomics with index from 1 to \texttt{N}.

Data Types

\texttt{atomics\_ref()}

Identifies an atomic array returned from \texttt{new/2}.

Exports

\texttt{new(Arity, Opts) -> atomics\_ref()}

Types:

\texttt{Arity = integer() >= 1}
\texttt{Opts = [Opt]}
\texttt{Opt = \{signed, boolean()\}}

Create a new array of \texttt{Arity} number of atomics. All atomics in the array are initially set to zero.

Argument \texttt{Opt} is a list of the following possible options:

\texttt{(signed, boolean())}

Indicate if the elements of the array will be treated as signed or unsigned integers. Default is \texttt{true} (signed).

The integer interval for signed atomics are from \texttt{-(1 bsl 63)} to \texttt{(1 bsl 63)-1} and for unsigned atomics from \texttt{0} to \texttt{(1 bsl 64)-1}.

Atomics are not tied to the current process and are automatically garbage collected when they are no longer referenced.

\texttt{put(Ref, Ix, Value) -> ok}

Types:

\texttt{Ref = atomics\_ref()}
\texttt{Ix = Value = integer()}

Set atomic to \texttt{Value}.

\texttt{get(Ref, Ix) -> integer()}

Types:
atomics

Ref = atomics_ref()
Ix = integer()
Read atomic value.

add(Ref, Ix, Incr) -> ok
Types:
    Ref = atomics_ref()
    Ix = Incr = integer()
Add Incr to atomic.

add_get(Ref, Ix, Incr) -> integer()
Types:
    Ref = atomics_ref()
    Ix = Incr = integer()
Atomic addition and return of the result.

sub(Ref, Ix, Decr) -> ok
Types:
    Ref = atomics_ref()
    Ix = Decr = integer()
Subtract Decr from atomic.

sub_get(Ref, Ix, Decr) -> integer()
Types:
    Ref = atomics_ref()
    Ix = Decr = integer()
Atomic subtraction and return of the result.

exchange(Ref, Ix, Desired) -> integer()
Types:
    Ref = atomics_ref()
    Ix = Desired = integer()
Atomically replaces the value of the atomic with Desired and returns the value it held previously.

compare_exchange(Ref, Ix, Expected, Desired) -> ok | integer()
Types:
    Ref = atomics_ref()
    Ix = Expected = Desired = integer()
Atomically compares the atomic with Expected, and if those are equal, set atomic to Desired. Returns ok if Desired was written. Returns the actual atomic value if not equal to Expected.
info(Ref) -> Info
Types:
    Ref = atomics_ref()
    Info =
        #{size := Size, max := Max, min := Min, memory := Memory}
    Size = integer() >= 0
    Max = Min = integer()
    Memory = integer() >= 0

Return information about an atomic array in a map. The map has the following keys:

size
    The number of atomics in the array.

max
    The highest possible value an atomic in this array can hold.

min
    The lowest possible value an atomic in this array can hold.

memory
    Approximate memory consumption for the array in bytes.
This module provides a set of functions to do operations towards shared mutable counter variables. The implementation does not utilize any software level locking, which makes it very efficient for concurrent access. The counters are organized into arrays with the following semantics:

- Counters are 64 bit signed integers.
- Counters wrap around at overflow and underflow operations.
- Counters are initialized to zero.
- Write operations guarantee atomicity. No intermediate results can be seen from a single write operation.
- Two types of counter arrays can be created with options `atomics` or `write_concurrency`. The `atomics` counters have good allround performance with nice consistent semantics while `write_concurrency` counters offers even better concurrent write performance at the expense of some potential read inconsistencies. See `new/2`.
- Indexes into counter arrays are one-based. A counter array of size N contains N counters with index from 1 to N.

### Data Types

counters_ref()  
Identifies a counter array returned from `new/2`.

### Exports

counters

**new(Size, Opts) -> counters_ref()**

Types:

- **Size** = integer() >= 1  
- **Opts** = [Opt]  
  - **Opt** = atomics | write_concurrency

Create a new counter array of `Size` counters. All counters in the array are initially set to zero.

Argument **`Opts`** is a list of the following possible options:

- `atomics` (Default)  
  Counters will be sequentially consistent. If write operation A is done sequentially before write operation B, then a concurrent reader may see the result of none of them, only A, or both A and B. It cannot see the result of only B.

- `write_concurrency`  
  This is an optimization to achieve very efficient concurrent `add` and `sub` operations at the expense of potential read inconsistency and memory consumption per counter.

  Read operations may see sequentially inconsistent results with regard to concurrent write operations. Even if write operation A is done sequentially before write operation B, a concurrent reader may see any combination of A and B, including only B. A read operation is only guaranteed to see all writes done sequentially before the read. No writes are ever lost, but will eventually all be seen.

  The typical use case for `write_concurrency` is when concurrent calls to `add` and `sub` toward the same counters are very frequent, while calls to `get` and `put` are much less frequent. The lack of absolute read consistency must also be acceptable.

Counters are not tied to the current process and are automatically garbage collected when they are no longer referenced.
get(Ref, Ix) -> integer()
Types:
   Ref = counters_ref()
   Ix = integer()
Read counter value.

add(Ref, Ix, Incr) -> ok
Types:
   Ref = counters_ref()
   Ix = Incr = integer()
Add Incr to counter at index Ix.

sub(Ref, Ix, Decr) -> ok
Types:
   Ref = counters_ref()
   Ix = Decr = integer()
Subtract Decr from counter at index Ix.

put(Ref, Ix, Value) -> ok
Types:
   Ref = counters_ref()
   Ix = Value = integer()
Write Value to counter at index Ix.

Note:
Despite its name, the write_concurrency optimization does not improve put. A call to put is a relatively heavy operation compared to the very lightweight and scalable add and sub. The cost for a put with write_concurrency is like a get plus a put without write_concurrency.

info(Ref) -> Info
Types:
   Ref = counters_ref()
   Info = #{size := Size, memory := Memory}
   Size = Memory = integer() >= 0
Return information about a counter array in a map. The map has the following keys (at least):

size
   The number of counters in the array.

memory
   Approximate memory consumption for the array in bytes.
Warning:

Use this functionality with extreme care.

A driver callback is executed as a direct extension of the native code of the VM. Execution is not made in a safe environment. The VM cannot provide the same services as provided when executing Erlang code, such as preemptive scheduling or memory protection. If the driver callback function does not behave well, the whole VM will misbehave.

- A driver callback that crash will crash the whole VM.
- An erroneously implemented driver callback can cause a VM internal state inconsistency, which can cause a crash of the VM, or miscellaneous misbehaviors of the VM at any point after the call to the driver callback.
- A driver callback doing lengthy work before returning degrades responsiveness of the VM, and can cause miscellaneous strange behaviors. Such strange behaviors include, but are not limited to, extreme memory usage, and bad load balancing between schedulers. Strange behaviors that can occur because of lengthy work can also vary between Erlang/OTP releases.

As from ERTS 5.9 (Erlang/OTP R15B) the driver interface has been changed with larger types for the callbacks output, control, and call. See driver version management in erl_driver.

Note:

Old drivers (compiled with an erl_driver.h from an ERTS version earlier than 5.9) must be updated and have to use the extended interface (with version management).

The driver_entry structure is a C struct that all Erlang drivers define. It contains entry points for the Erlang driver, which are called by the Erlang emulator when Erlang code accesses the driver.

The erl_driver driver API functions need a port handle that identifies the driver instance (and the port in the emulator). This is only passed to the start function, but not to the other functions. The start function returns a driver-defined handle that is passed to the other functions. A common practice is to have the start function allocate some application-defined structure and stash the port handle in it, to use it later with the driver API functions.

The driver callback functions are called synchronously from the Erlang emulator. If they take too long before completing, they can cause time-outs in the emulator. Use the queue or asynchronous calls if necessary, as the emulator must be responsive.

The driver structure contains the driver name and some 15 function pointers, which are called at different times by the emulator.

The only exported function from the driver is driver_init. This function returns the driver_entry structure that points to the other functions in the driver. The driver_init function is declared with a macro, DRIVER_INIT(drivername). (This is because different operating systems have different names for it.)

When writing a driver in C++, the driver entry is to be of "C" linkage. One way to do this is to put the following line somewhere before the driver entry:

```c
extern "C" DRIVER_INIT(drivername);
```
When the driver has passed the driver_entry over to the emulator, the driver is **not** allowed to modify the driver_entry.

If compiling a driver for static inclusion through `--enable-static-drivers`, you must define `STATIC_ERLANG_DRIVER` before the `DRIVER_INIT` declaration.

**Note:**

Do **not** declare the `driver_entry` const. This because the emulator must modify the `handle` and the `handle2` fields. A statically allocated, and const-declared `driver_entry` can be located in read-only memory, which causes the emulator to crash.

**Data Types**

`ErlDrvEntry`
typedef struct erl_drv_entry {
    int (*init)(void);          /* Called at system startup for statically
                                 linked drivers, and after loading for
                                 dynamically loaded drivers */
    #ifndef ERL_SYS_DRV
        ErlDrvData (*start)(ErlDrvPort port, char *command);
        /* Called when open_port/2 is invoked,
         return value -1 means failure */
    #else
        ErlDrvData (*start)(ErlDrvPort port, char *command, SysDriverOpts* opts);
        /* Special options, only for system driver */
    #endif
    void (*stop)(ErlDrvData drv_data);
        /* Called when port is closed, and when the
         emulator is halted */
    void (*output)(ErlDrvData drv_data, char *buf, ErlDrvSizeT len);
        /* Called when we have output from Erlang to
         the port */
    void (*ready_input)(ErlDrvData drv_data, ErlDrvEvent event);
        /* Called when we have input from one of
         the driver's handles */
    void (*ready_output)(ErlDrvData drv_data, ErlDrvEvent event);
        /* Called when output is possible to one of
         the driver's handles */
    char *driver_name;          /* Name supplied as command in
                                 erlang:open_port/2 */
    void (*finish)(void);       /* Called before unloading the driver -
                                 dynamic drivers only */
    void *handle;               /* Reserved, used by emulator internally */
    ErlDrvSSizeT (*control)(ErlDrvData drv_data, unsigned int command,
                            char *buf, ErlDrvSizeT len,
                            char **rbuf, ErlDrvSizeT rlen);
        /* "ioctl" for drivers - invoked by
         port_control/3 */
    void (*timeout)(ErlDrvData drv_data);
        /* Handling of time-out in driver */
    void (*outputv)(ErlDrvData drv_data, ErlIOVec *ev);
        /* Called when we have output from Erlang
to the port */
    void (*ready_async)(ErlDrvData drv_data, ErlDrvThreadData thread_data);
    void (*flush)(ErlDrvData drv_data);
        /* Called when the port is about to be
         closed, and there is data in the
driver queue that must be flushed
before 'stop' can be called */
    void (*call)(ErlDrvData drv_data, unsigned int command,
                 char *buf, ErlDrvSizeT len,
                 char **rbuf, ErlDrvSizeT rlen, unsigned int *flags);
        /* Works mostly like 'control', a synchronous
call into the driver */
    void* unused_event_callback;
    int extended_marker;        /* ERL_DRV_EXTENDED_MARKER */
    int major_version;          /* ERL_DRV_EXTENDED_MAJOR_VERSION */
    int minor_version;          /* ERL_DRV_EXTENDED_MINOR_VERSION */
    int driver_flags;           /* ERL_DRV_FLAGS */
    void *handle2;              /* Reserved, used by emulator internally */
    void (*process_exit)(ErlDrvData drv_data, ErlDrvMonitor *monitor);
        /* Called when a process monitor fires */
    void (*stop_select)(ErlDrvEvent event, void* reserved);
        /* Called to close an event object */
} ErlDrvEntry;
int (*init)(void)
   Called directly after the driver has been loaded by erl_ddll:load_driver/2 (actually when the driver is added to the driver list). The driver is to return 0, or, if the driver cannot initialize, -1.

ErlDrvData (*start)(ErlDrvPort port, char* command)
   Called when the driver is instantiated, when erlang:open_port/2 is called. The driver is to return a number >= 0 or a pointer, or, if the driver cannot be started, one of three error codes:
   
   ERL_DRV_ERROR_GENERAL
      General error, no error code
   ERL_DRV_ERROR_ERRNO
      Error with error code in errno
   ERL_DRV_ERROR_BADARG
      Error, badarg
   
   If an error code is returned, the port is not started.

void (*stop)(ErlDrvData drv_data)
   Called when the port is closed, with erlang:port_close/1 or Port ! {self(), close}. Notice that terminating the port owner process also closes the port. If drv_data is a pointer to memory allocated in start, then stop is the place to deallocate that memory.

void (*output)(ErlDrvData drv_data, char *buf, ErlDrvSizeT len)
   Called when an Erlang process has sent data to the port. The data is pointed to by buf, and is len bytes. Data is sent to the port with Port ! {self(), {command, Data}} or with erlang:port_command/2. Depending on how the port was opened, it is to be either a list of integers 0...255 or a binary. See erlang:open_port/2 and erlang:port_command/2.

void (*ready_input)(ErlDrvData drv_data, ErlDrvEvent event)
void (*ready_output)(ErlDrvData drv_data, ErlDrvEvent event)
   Called when a driver event (specified in parameter event) is signaled. This is used to help asynchronous drivers "wake up" when something occurs.

   On Unix the event is a pipe or socket handle (or something that the select system call understands).

   On Windows the event is an Event or Semaphore (or something that the WaitForMultipleObjects API function understands). (Some trickery in the emulator allows more than the built-in limit of 64 Events to be used.)

   To use this with threads and asynchronous routines, create a pipe on Unix and an Event on Windows. When the routine completes, write to the pipe (use SetEvent on Windows), this makes the emulator call ready_input or ready_output.

   False events can occur. That is, calls to ready_input or ready_output although no real events are signaled. In reality, it is rare (and OS-dependant), but a robust driver must nevertheless be able to handle such cases.

char *driver_name
   The driver name. It must correspond to the atom used in erlang:open_port/2, and the name of the driver library file (without the extension).

void (*finish)(void)
   Called by the erl_ddll driver when the driver is unloaded. (It is only called in dynamic drivers.)
   
   The driver is only unloaded as a result of calling erl_ddll:unload_driver/1, or when the emulator halts.
void *handle

This field is reserved for the emulator's internal use. The emulator will modify this field, so it is important that the driver_entry is not declared const.

ErlDrvSizeT (*control)(ErlDrvData drv_data, unsigned int command, char *buf, ErlDrvSizeT len, char **rbuf, ErlDrvSizeT rlen)

A special routine invoked with erlang:port_control/3. It works a little like an "ioctl" for Erlang drivers. The data specified to port_control/3 arrives in buf and len. The driver can send data back, using *rbuf and rlen.

This is the fastest way of calling a driver and get a response. It makes no context switch in the Erlang emulator and requires no message passing. It is suitable for calling C function to get faster execution, when Erlang is too slow.

If the driver wants to return data, it is to return it in rbuf. When control is called, *rbuf points to a default buffer of rlen bytes, which can be used to return data. Data is returned differently depending on the port control flags (those that are set with erl_driver:set_port_control_flags).

If the flag is set to PORT_CONTROL_FLAG_BINARY, a binary is returned. Small binaries can be returned by writing the raw data into the default buffer. A binary can also be returned by setting *rbuf to point to a binary allocated with erl_driver:driver_alloc_binary.

This binary is freed automatically after control has returned. The driver can retain the binary for read only access with erl_driver:driver_binary_inc_refc to be freed later with erl_driver:driver_free_binary. It is never allowed to change the binary after control has returned.

If *rbuf is set to NULL, an empty list is returned.

If the flag is set to 0, data is returned as a list of integers. Either use the default buffer or set *rbuf to point to a larger buffer allocated with erl_driver:driver_alloc. The buffer is freed automatically after control has returned.

Using binaries is faster if more than a few bytes are returned.

The return value is the number of bytes returned in *rbuf.

void (*timeout)(ErlDrvData drv_data)

Called any time after the driver's timer reaches 0. The timer is activated with erl_driver:driver_set_timer. No priorities or ordering exist among drivers, so if several drivers time out at the same time, anyone of them is called first.

void (*outputv)(ErlDrvData drv_data, ErlIOVec *ev)

Called whenever the port is written to. If it is NULL, the output function is called instead. This function is faster than output, as it takes an ErlIOVec directly, which requires no copying of the data. The port is to be in binary mode, see erlang:open_port/2.

ErlIOVec contains both a SysIOVec, suitable for writev, and one or more binaries. If these binaries are to be retained when the driver returns from outputv, they can be queued (using, for example, erl_driver:driver_enq_bin) or, if they are kept in a static or global variable, the reference counter can be incremented.

void (*ready_async)(ErlDrvData drv_data, ErlDrvThreadData thread_data)

Called after an asynchronous call has completed. The asynchronous call is started with erl_driver:driver_async. This function is called from the Erlang emulator thread, as opposed to the asynchronous function, which is called in some thread (if multi-threading is enabled).

void (*flush)(ErlDrvData drv_data)

Called when the port is about to be closed, and there is data in the driver queue that must be flushed before 'stop' can be called.

126 | Ericsson AB. All Rights Reserved.: Erlang Run-Time System Application (ERTS)
**driver_entry**

ErlDrvSSizeT (*call)(ErlDrvData drv_data, unsigned int command, char *buf, ErlDrvSizeT len, char **rbuf, ErlDrvSizeT rlen, unsigned int *flags)

Called from erlang:port_call/3. It works a lot like the control callback, but uses the external term format for input and output.

command is an integer, obtained from the call from Erlang (the second argument to erlang:port_call/3).

buf and len provide the arguments to the call (the third argument to erlang:port_call/3). They can be decoded using ei functions.

rbuf points to a return buffer, rlen bytes long. The return data is to be a valid Erlang term in the external (binary) format. This is converted to an Erlang term and returned by erlang:port_call/3 to the caller. If more space than rlen bytes is needed to return data, *rbuf can be set to memory allocated with erl_driver:driver_alloc. This memory is freed automatically after call has returned.

The return value is the number of bytes returned in *rbuf. If ERL_DRV_ERROR_GENERAL is returned (or in fact, anything < 0), erlang:port_call/3 throws a BAD_ARG.

**void (*event)(ErlDrvData drv_data, ErlDrvEvent event, ErlDrvEventData event_data)**

Intentionally left undocumented.

**int extended_marker**

This field is either to be equal to ERL_DRV_EXTENDED_MARKER or 0. An old driver (not aware of the extended driver interface) is to set this field to 0. If this field is 0, all the following fields must also be 0, or NULL if it is a pointer field.

**int major_version**

This field is to equal ERL_DRV_EXTENDED_MAJOR_VERSION if field extended_marker equals ERL_DRV_EXTENDED_MARKER.

**int minor_version**

This field is to equal ERL_DRV_EXTENDED_MINOR_VERSION if field extended_marker equals ERL_DRV_EXTENDED_MARKER.

**int driver_flags**

This field is used to pass driver capability and other information to the runtime system. If field extended_marker equals ERL_DRV_EXTENDED_MARKER, it is to contain 0 or driver flags (ERL_DRV_FLAG_*) OR'ed bitwise. The following driver flags exist:

ERL_DRV_FLAG_USE_PORT_LOCKING

The runtime system uses port-level locking on all ports executing this driver instead of driver-level locking. For more information, see erl_driver.

ERL_DRV_FLAG_SOFT_BUSY

Marks that driver instances can handle being called in the output and/or outputv callbacks although a driver instance has marked itself as busy (see erl_driver:set_busy_port). As from ERTS 5.7.4 this flag is required for drivers used by the Erlang distribution (the behavior has always been required by drivers used by the distribution).

ERL_DRV_FLAG_NO_BUSY_MSGQ

Disables busy port message queue functionality. For more information, see erl_driver:erl_drv_busy_msgq_limits.
ERL_DRV_FLAG_USE_INIT_ACK

When this flag is specified, the linked-in driver must manually acknowledge that the port has been successfully started using erl_driver:erl_drv_init_ack(). This allows the implementor to make the erlang:open_port exit with badarg after some initial asynchronous initialization has been done.

void *handle2

This field is reserved for the emulator's internal use. The emulator modifies this field, so it is important that the driver_entry is not declared const.

void (*process_exit)(ErlDrvData drv_data, ErlDrvMonitor *monitor)

Called when a monitored process exits. The drv_data is the data associated with the port for which the process is monitored (using erl_driver:driver_monitor_process) and the monitor corresponds to the ErlDrvMonitor structure filled in when creating the monitor. The driver interface function erl_driver:driver_get_monitored_process can be used to retrieve the process ID of the exiting process as an ErlDrvTermData.

void (*stop_select)(ErlDrvEvent event, void* reserved)

Called on behalf of erl_driver:driver_select when it is safe to close an event object.

A typical implementation on Unix is to do close((int)event).

Argument reserved is intended for future use and is to be ignored.

In contrast to most of the other callback functions, stop_select is called independent of any port. No ErlDrvData argument is passed to the function. No driver lock or port lock is guaranteed to be held. The port that called driver_select can even be closed at the time stop_select is called. But it can also be the case that stop_select is called directly by erl_driver:driver_select.

It is not allowed to call any functions in the driver API from stop_select. This strict limitation is because the volatile context that stop_select can be called.

See Also

erl_driver(3), erlang(3), erl_ddll(3)
epmd

Command

epmd [-d|-debug] [DbgExtra...] [-address Addresses] [-port No] [-daemon] [-relaxed_command_check]

Starts the port mapper daemon.

epmd [-d|-debug] [-port No] [-names|-kill|-stop Name]

Communicates with a running port mapper daemon.

This daemon acts as a name server on all hosts involved in distributed Erlang computations. When an Erlang node starts, the node has a name and it obtains an address from the host OS kernel. The name and address are sent to the epmd daemon running on the local host. In a TCP/IP environment, the address consists of the IP address and a port number. The node name is an atom on the form of Name@Node. The job of the epmd daemon is to keep track of which node name listens on which address. Hence, epmd maps symbolic node names to machine addresses.

The TCP/IP epmd daemon only keeps track of the Name (first) part of an Erlang node name. The Host part (whatever is after the @) is implicit in the node name where the epmd daemon was contacted, as is the IP address where the Erlang node can be reached. Consistent and correct TCP naming services are therefore required for an Erlang network to function correctly.

Note:

On Windows the maximum number of nodes allowed in one epmd instance is 60. This is because of limitations in the current implementation. If you need more nodes, you should look into using an erlang based epmd implementation such as Erlang EPMD.

Starting the port mapper daemon

The daemon is started automatically by command erl(1) if the node is to be distributed and no running instance is present. If automatically launched environment variables must be used to change the behavior of the daemon; see section Environment Variables.

If argument -daemon is not specified, epmd runs as a normal program with the controlling terminal of the shell in which it is started. Normally, it is to be run as a daemon.

Regular startup options are described in section Regular Options.

The DdbgExtra options are described in section DdbgExtra Options.

Communicating with a running port mapper daemon

Communicating with the running epmd daemon by the epmd program is done primarily for debugging purposes.

The different queries are described in section Interactive options.

Regular Options

These options are available when starting the name server. The name server is normally started automatically by command erl(1) (if not already available), but it can also be started at system startup.

-address List

Lets this instance of epmd listen only on the comma-separated list of IP addresses and on the loopback address (which is implicitly added to the list if it has not been specified). This can also be set using environment variable ERL_EPMD_ADDRESS; see section Environment Variables.
-port No

    Lets this instance of epmd listen to another TCP port than default 4369. This can also be set using environment variable ERL_EPMD_PORT; see section Environment Variables.

-d | -debug

    Enables debug output. The more -d flags specified, the more debug output you will get (to a certain limit). This option is most useful when the epmd daemon is not started as a daemon.

-daemon

    Starts epmd detached from the controlling terminal. Logging ends up in syslog when available and correctly configured. If the epmd daemon is started at boot, this option is definitely to be used. It is also used when command erl automatically starts epmd.

-relaxed_command_check

    Starts the epmd program with relaxed command checking (mostly for backward compatibility). This affects the following:

    • With relaxed command checking, the epmd daemon can be killed from the local host with, for example, command epmd -kill even if active nodes are registered. Normally only daemons with an empty node database can be killed with epmd -kill.

    • Command epmd -stop (and the corresponding messages to epmd, as can be specified using erl_interface:ei(3)) is normally always ignored. This because it can cause a strange situation where two nodes of the same name can be alive at the same time. A node unregisters itself by only closing the connection to epmd, which is why command stop was only intended for use in debugging situations.

    With relaxed command checking enabled, you can forcibly unregister live nodes.

    Relaxed command checking can also be enabled by setting environment variable ERL_EPMD_RELAXED_COMMAND_CHECK before starting epmd.

    Use relaxed command checking only on systems with very limited interactive usage.

### DbgExtra Options

**Note:**

These options are only for debugging and testing epmd clients. They are not to be used in normal operation.

-packet_timeout Seconds

    Sets the number of seconds a connection can be inactive before epmd times out and closes the connection. Defaults to 60.

-delay_accept Seconds

    To simulate a busy server, you can insert a delay between when epmd gets notified that a new connection is requested and when the connection gets accepted.

-delay_write Seconds

    Also a simulation of a busy server. Inserts a delay before a reply is sent.

### Interactive Options

These options make epmd run as an interactive command, displaying the results of sending queries to an already running instance of epmd. The epmd contacted is always on the local node, but option -port can be used to select between instances if several are running using different ports on the host.
-port No

Contacts the epmd listening on the specified TCP port number (default 4369). This can also be set using environment variable ERL_EPMD_PORT; see section Environment Variables.

-names

Lists names registered with the currently running epmd.

-kill

Kills the currently running epmd.

Killing the running epmd is only allowed if epmd -names shows an empty database or if -relaxed_command_check was specified when the running instance of epmd was started.

Notice that -relaxed_command_check is specified when starting the daemon that is to accept killing when it has live nodes registered. When running epmd interactively, -relaxed_command_check has no effect. A daemon that is started without relaxed command checking must be killed using, for example, signals or some other OS-specific method if it has active clients registered.

-stop Name

Forcibly unregisters a live node from the epmd database.

This command can only be used when contacting epmd instances started with flag -relaxed_command_check.

Notice that relaxed command checking must enabled for the epmd daemon contacted. When running epmd interactively, -relaxed_command_check has no effect.

Environment Variables

ERL_EPMD_ADDRESS

Can be set to a comma-separated list of IP addresses, in which case the epmd daemon will listen only on the specified address(es) and on the loopback address (which is implicitly added to the list if it has not been specified). The default behavior is to listen on all available IP addresses.

ERL_EPMD_PORT

Can contain the port number epmd will use. The default port will work fine in most cases. A different port can be specified to allow several instances of epmd, representing independent clusters of nodes, to co-exist on the same host. All nodes in a cluster must use the same epmd port number.

ERL_EPMD_RELAXED_COMMAND_CHECK

If set before start, the epmd daemon behaves as if option -relaxed_command_check was specified at startup. Consequently, if this option is set before starting the Erlang virtual machine, the automatically started epmd accepts the -kill and -stop commands without restrictions.

Logging

On some operating systems syslog will be used for error reporting when epmd runs as a daemon. To enable the error logging, you must edit the /etc/syslog.conf file and add an entry:

```
!epmd
*.*/var/log/epmd.log
```

where <TABs> are at least one real tab character. Spaces are silently ignored.
Access Restrictions

The epmd daemon accepts messages from both the local host and remote hosts. However, only the query commands are answered (and acted upon) if the query comes from a remote host. It is always an error to try to register a node name if the client is not a process on the same host as the epmd instance is running on. Such requests are considered hostile and the connection is closed immediately.

The following queries are accepted from remote nodes:

- Port queries, that is, on which port the node with a specified name listens
- Name listing, that is, gives a list of all names registered on the host

To restrict access further, firewall software must be used.
erl

Command

The erl program starts an Erlang runtime system. The exact details (for example, whether erl is a script or a program and which other programs it calls) are system-dependent.

Windows users probably want to use the werl program instead, which runs in its own window with scrollbars and supports command-line editing. The erl program on Windows provides no line editing in its shell, and on Windows 95 there is no way to scroll back to text that has scrolled off the screen. The erl program must be used, however, in pipelines or if you want to redirect standard input or output.

Note:

As from ERTS 5.9 (Erlang/OTP R15B) the runtime system does by default not bind schedulers to logical processors. For more information, see system flag +sbt.

Exports

erl <arguments>

Starts an Erlang runtime system.

The arguments can be divided into emulator flags, flags, and plain arguments:

- Any argument starting with character + is interpreted as an emulator flag.
  
  As indicated by the name, emulator flags control the behavior of the emulator.

- Any argument starting with character - (hyphen) is interpreted as a flag, which is to be passed to the Erlang part of the runtime system, more specifically to the init system process, see init(3).

  The init process itself interprets some of these flags, the init flags. It also stores any remaining flags, the user flags. The latter can be retrieved by calling init:get_argument/1.

  A small number of "-" flags exist, which now actually are emulator flags, see the description below.

- Plain arguments are not interpreted in any way. They are also stored by the init process and can be retrieved by calling init:get_plain_arguments/0. Plain arguments can occur before the first flag, or after a -- flag. Also, the -extra flag causes everything that follows to become plain arguments.

Examples:

% erl +W w -sname arnie +R 9 -s my_init -extra +bertie
(arnie@host)> init:get_argument(sname).
{ok,[["arnie"]]} 
(arnie@host)> init:get_plain_arguments().
["+bertie"]

Here +W w and +R 9 are emulator flags. -s my_init is an init flag, interpreted by init. -sname arnie is a user flag, stored by init. It is read by Kernel and causes the Erlang runtime system to become distributed. Finally, everything after -extra (that is, +bertie) is considered as plain arguments.
% erl -myflag 1
1> init:get_argument(myflag).
{ok,[["1"]]}  
2> init:get_plain_arguments().
[]

Here the user flag -myflag 1 is passed to and stored by the init process. It is a user-defined flag, presumably used by some user-defined application.

Flags

In the following list, init flags are marked "(init flag)". Unless otherwise specified, all other flags are user flags, for which the values can be retrieved by calling init:get_argument/1. Notice that the list of user flags is not exhaustive, there can be more application-specific flags that instead are described in the corresponding application documentation.

-- (init flag)

Everything following -- up to the next flag (-flag or +flag) is considered plain arguments and can be retrieved using init:get_plain_arguments/0.

-Application Par Val

Sets the application configuration parameter Par to the value Val for the application Application; see app(4) and application(3).

-args_file FileName

Command-line arguments are read from the file FileName. The arguments read from the file replace flag '-args_file FileName' on the resulting command line.

The file FileName is to be a plain text file and can contain comments and command-line arguments. A comment begins with a # character and continues until the next end of line character. Backslash (\) is used as quoting character. All command-line arguments accepted by erl are allowed, also flag -args_file FileName. Be careful not to cause circular dependencies between files containing flag -args_file, though.

The flag -extra is treated in special way. Its scope ends at the end of the file. Arguments following an -extra flag are moved on the command line into the -extra section, that is, the end of the command line following after an -extra flag.

-async_shell_start

The initial Erlang shell does not read user input until the system boot procedure has been completed (Erlang/OTP 5.4 and later). This flag disables the start synchronization feature and lets the shell start in parallel with the rest of the system.

-boot File

Specifies the name of the boot file, File.boot, which is used to start the system; see init(3). Unless File contains an absolute path, the system searches for File.boot in the current and $ROOT/bin directories.

Defaults to $ROOT/bin/start.boot.

-boot_var Var Dir

If the boot script contains a path variable Var other than $ROOT, this variable is expanded to Dir. Used when applications are installed in another directory than $ROOT/lib; see systools:make_script/1,2 in SASL.

-code_path_cache

Enables the code path cache of the code server; see code(3).
-compile Mod1 Mod2 ...

Compiles the specified modules and then terminates (with non-zero exit code if the compilation of some file did not succeed). Implies -noinput.

Not recommended; use erlc instead.

-config Config [Config ...]

Specifies the name of one or more configuration files, Config.config, which is used to configure applications; see app(4) and application(3). See the documentation for the configuration file format for a description of the configuration format and the order in which configuration parameters are read.

-configfd FD [FD ...]

Specifies the name of one or more file descriptors (called configuration file descriptors from here on) with configuration data for applications; see app(4) and application(3). See the documentation for the configuration file format for a description of the configuration format and the order in which configuration parameters are read.

A configuration file descriptor will be read until its end and will then be closed.

The content of a configuration file descriptor is stored so that it can be reused when init:restart/0 or init:restart/1 is called.

The parameter -configfd 0 implies -noinput.

Note:

It is not recommended to use file descriptors 1 (standard output), and 2 (standard error) together with -configfd as these file descriptors are typically used to print information to the console the program is running in.

Examples (Unix shell):

```
$ erl \n  -noshell \n  -configfd 3 \n  -eval \n  \"io:format("\n\3<\n\{ok,warning}\n\"Test 1\"
\n$ echo '\{kernel, [\{logger_level, warning\}\]}.' > test1.config
$ echo '\{kernel, [\{logger_level, error\}\]}.' > test2.config
$ erl \n  -noshell \n  -configfd 3 \n  -configfd 4 \n  -eval \n  \"io:format("\n\3<\n\{ok,error\}
```

-connect_all false

This flag is deprecated and has been replaced by the kernel application parameter connect_all.

-cookie Cookie

Obsolete flag without any effect and common misspelling for -setcookie. Use -setcookie instead.
-detached

Starts the Erlang runtime system detached from the system console. Useful for running daemons and backgrounds processes. Implies -noinput.

-disable-feature feature

Disables the feature feature in the runtime system. The special feature all can be used to disable all non permanent features.

-dist_listen true|false

Specifies whether this node should be listening for incoming distribution connections or not. By default a node will listen for incoming connections. Setting this option to false implies -hidden.

-emu_args

Useful for debugging. Prints the arguments sent to the emulator.

-emu_flavor emu|jit|smp

Start an emulator of a different flavor. Normally only one flavor is available, more can be added by building specific flavors. The currently available flavors are: emu and jit. The smp flavor is an alias for the current default flavor. You can combine this flag with --emu_type. You can get the current flavor at run-time using erlang:system_info(emu_flavor). (The emulator with this flavor must be built. You can build a specific flavor by doing FLAVOR=$FLAVOR in the Erlang/OTP source repository.)

-emu_type Type

Start an emulator of a different type. For example, to start the lock-counter emulator, use -emu_type lcnt. You can get the current type at run-time using erlang:system_info(build_type). (The emulator of this type must already be built. Use the configure option --enable-lock-counter to build the lock-counter emulator.)

-enable-feature feature

Enables the feature feature in the runtime system. The special feature all can be used to enable all features.

-env Variable Value

Sets the host OS environment variable Variable to the value Value for the Erlang runtime system. Example:

% erl -env DISPLAY gin:0

In this example, an Erlang runtime system is started with environment variable DISPLAY set to gin:0.

-epmd_module Module (init flag)

Configures the module responsible to communicate to epmd. Defaults to erl_epmd.

-erl_epmd_port Port (init flag)

Configures the port used by erl_epmd to listen for connection and connect to other nodes. See erl_epmd for more details. Defaults to 0.

-eval Expr (init flag)

Makes init evaluate the expression Expr; see init(3).

-extra (init flag)

Everything following -extra is considered plain arguments and can be retrieved using init:get_plain_arguments/0.
-heart

Starts heartbeat monitoring of the Erlang runtime system; see `heart(3)`.

-hidden

Starts the Erlang runtime system as a hidden node, if it is run as a distributed node. Hidden nodes always establish hidden connections to all other nodes except for nodes in the same global group. Hidden connections are not published on any of the connected nodes, that is, none of the connected nodes are part of the result from `nodes/0` on the other node. See also hidden global groups; `global_group(3)`.

-hosts Hosts

Specifies the IP addresses for the hosts on which Erlang boot servers are running, see `erl_boot_server(3)`. This flag is mandatory if flag `-loader inet` is present.

The IP addresses must be specified in the standard form (four decimal numbers separated by periods, for example, "150.236.20.74"). Hosts names are not acceptable, but a broadcast address (preferably limited to the local network) is.

-id Id

Specifies the identity of the Erlang runtime system. If it is run as a distributed node, Id must be identical to the name supplied together with flag `-sname` or `-name`.

-init_debug

Makes `init` write some debug information while interpreting the boot script.

-instr (emulator flag)

Selects an instrumented Erlang runtime system (virtual machine) to run, instead of the ordinary one. When running an instrumented runtime system, some resource usage data can be obtained and analyzed using the `instrument` module. Functionally, it behaves exactly like an ordinary Erlang runtime system.

-loader Loader

Specifies the method used by `erl_prim_loader` to load Erlang modules into the system; see `erl_prim_loader(3)`. Two Loader methods are supported:

- `efile`, which means use the local file system, this is the default.
- `inet`, which means use a boot server on another machine. The flags `-id`, `-hosts` and `-setcookie` must also be specified.

If Loader is something else, the user-supplied Loader port program is started.

-make

Makes the Erlang runtime system invoke `make:all()` in the current working directory and then terminate; see `make(3)`. Implies `-noinput`.

-man Module

Displays the manual page for the Erlang module Module. Only supported on Unix.

-mode interactive | embedded

Modules are auto loaded when they are first referenced if the runtime system runs in `interactive` mode, which is the default. In `embedded` mode modules are not auto loaded. The latter is recommended when the boot script preloads all modules, as conventionally happens in OTP releases. See `code(3)`.

-name Name

Makes the Erlang runtime system into a distributed node. This flag invokes all network servers necessary for a node to become distributed; see `net_kernel(3)`. It also ensures that `epmd` runs on the current host before
Erlang is started (see `epmd(1)` and the `-start_epmd` option) and that a magic cookie has been set (see `-setcookie`).

The node name will be `Name@Host`, where `Host` is the fully qualified host name of the current host. For short names, use flag `-sname` instead.

If `Name` is set to **undefined** the node will be started in a special mode optimized to be the temporary client of another node. The node will then request a dynamic node name from the first node it connects to. Read more in Dynamic Node Name.

**Warning:**

Starting a distributed node without also specifying `-proto_dist inet_tls` will expose the node to attacks that may give the attacker complete access to the node and in extension the cluster. When using unsecure distributed nodes, make sure that the network is configured to keep potential attackers out.

**-no_epmd**

Specifies that the distributed node does not need epmd at all.

This option ensures that the Erlang runtime system does not start epmd and does not start the erl_epmd process for distribution either.

This option only works if Erlang is started as a distributed node with the `-proto_dist` option using an alternative protocol for Erlang distribution which does not rely on epmd for node registration and discovery. For more information, see How to implement an Alternative Carrier for the Erlang Distribution.

**-noinput**

Ensures that the Erlang runtime system never tries to read any input. Implies `-noshell`.

**-noshell**

Starts an Erlang runtime system with no shell. This flag makes it possible to have the Erlang runtime system as a component in a series of Unix pipes.

**-nostick**

Disables the sticky directory facility of the Erlang code server; see `code(3)`.

**-oldshell**

Invokes the old Erlang shell from Erlang/OTP 3.3. The old shell can still be used.

**-pa Dir1 Dir2 ...**

Adds the specified directories to the beginning of the code path, similar to `code:add_pathsa/1`. Note that the order of the given directories will be reversed in the resulting path.

As an alternative to `-pa`, if several directories are to be prepended to the code path and the directories have a common parent directory, that parent directory can be specified in environment variable `ERL_LIBS`; see `code(3)`.

**-pz Dir1 Dir2 ...**

Adds the specified directories to the end of the code path, similar to `code:add_pathsz/1`; see `code(3)`.

**-path Dir1 Dir2 ...**

Replaces the path specified in the boot script; see `script(4)`.

**-proto_dist Proto**

Specifies a protocol for Erlang distribution:
**inet_tcp**
TCP over IPv4 (the default)

**inet_tls**
Distribution over TLS/SSL. See the Using SSL for Erlang Distribution User's Guide for details on how to setup a secure distributed node.

**inet6_tcp**
TCP over IPv6

For example, to start up IPv6 distributed nodes:

```
% erl -name test@ipv6node.example.com -proto_dist inet6_tcp
```

**-remsh Node**
Starts Erlang with a remote shell connected to Node.

If no `-name` or `-sname` is given the node will be started using `-sname undefined`. If Node does not contain a hostname, one is automatically taken from `-name` or `-sname`

**Note:**
Before OTP-23 the user **needed** to supply a valid `-sname` or `-name` for `-remsh` to work. This is still the case if the target node is not running OTP-23 or later.

**-rsh Program**
Specifies an alternative to ssh for starting a slave node on a remote host; see slave(3).

**-run Mod [Func [Arg1, Arg2, ...]] (init flag)**
Makes `init` call the specified function. `Func` defaults to `start`. If no arguments are provided, the function is assumed to be of arity 0. Otherwise it is assumed to be of arity 1, taking the list `[Arg1, Arg2, ...]` as argument. All arguments are passed as strings. See `init(3)`.

**-s Mod [Func [Arg1, Arg2, ...]] (init flag)**
Makes `init` call the specified function. `Func` defaults to `start`. If no arguments are provided, the function is assumed to be of arity 0. Otherwise it is assumed to be of arity 1, taking the list `[Arg1, Arg2, ...]` as argument. All arguments are passed as atoms. See `init(3)`.

**-setcookie Cookie**
Sets the magic cookie of the node to `Cookie`; see `erlang:set_cookie/1`. See section Distributed Erlang in the Erlang Reference Manual for more details.

**-setcookie Node Cookie**
Sets the magic cookie for `Node` to `Cookie`; see `erlang:set_cookie/2`.

**-shutdown_time Time**
Specifies how long time (in milliseconds) the `init` process is allowed to spend shutting down the system. If `Time` milliseconds have elapsed, all processes still existing are killed. Defaults to infinity.
-sname Name

Makes the Erlang runtime system into a distributed node, similar to -name, but the host name portion of the node name Name@Host will be the short name, not fully qualified.

This is sometimes the only way to run distributed Erlang if the Domain Name System (DNS) is not running. No communication can exist between nodes running with flag -sname and those running with flag -name, as node names must be unique in distributed Erlang systems.

If Name is set to undefined the node will be started in a special mode optimized to be the temporary client of another node. The node will then request a dynamic node name from the first node it connects to. Read more in Dynamic Node Name.

**Warning:**

Starting a distributed node without also specifying -proto_dist inet_tls will expose the node to attacks that may give the attacker complete access to the node and in extension the cluster. When using un-secure distributed nodes, make sure that the network is configured to keep potential attackers out.

-start_epmd true | false

Specifies whether Erlang should start epmd on startup. By default this is true, but if you prefer to start epmd manually, set this to false.

This only applies if Erlang is started as a distributed node, i.e. if -name or -sname is specified. Otherwise, epmd is not started even if -start_epmd true is given.

Note that a distributed node will fail to start if epmd is not running.

-version (emulator flag)

Makes the emulator print its version number. The same as erl +V.

**Emulator Flags**

erl invokes the code for the Erlang emulator (virtual machine), which supports the following flags:

+a size

Suggested stack size, in kilowords, for threads in the async thread pool. Valid range is 16-8192 kilowords. The default suggested stack size is 16 kilowords, that is, 64 kilobyte on 32-bit architectures. This small default size has been chosen because the number of async threads can be large. The default size is enough for drivers delivered with Erlang/OTP, but might not be large enough for other dynamically linked-in drivers that use the driver_async() functionality. Notice that the value passed is only a suggestion, and it can even be ignored on some platforms.

+A size

Sets the number of threads in async thread pool. Valid range is 1-1024. The async thread pool is used by linked-in drivers to handle work that may take a very long time. Since OTP 21 there are very few linked-in drivers in the default Erlang/OTP distribution that uses the async thread pool. Most of them have been migrated to dirty IO schedulers. Defaults to 1.

+B [c | d | i]

Option c makes Ctrl-C interrupt the current shell instead of invoking the emulator break handler. Option d (same as specifying +B without an extra option) disables the break handler. Option i makes the emulator ignore any break signal.

If option c is used with oldshell on Unix, Ctrl-C will restart the shell process rather than interrupt it.
Notice that on Windows, this flag is only applicable for werl, not erl (oldshell). Notice also that Ctrl-Break is used instead of Ctrl-C on Windows.

```
+c true | false
```

Enables or disables time correction:

```
true
```

Enables time correction. This is the default if time correction is supported on the specific platform.

```
false
```

Disables time correction.

For backward compatibility, the boolean value can be omitted. This is interpreted as +c false.

```
+C no_time_warp | single_time_warp | multi_time_warp
```

Sets time warp mode:

```
no_time_warp
```

No time warp mode (the default)

```
single_time_warp
```

Single time warp mode

```
multi_time_warp
```

Multi-time warp mode

```
+d
```

If the emulator detects an internal error (or runs out of memory), it, by default, generates both a crash dump and a core dump. The core dump is, however, not very useful as the content of process heaps is destroyed by the crash dump generation.

Option +d instructs the emulator to produce only a core dump and no crash dump if an internal error is detected.

Calling erlang:halt/1 with a string argument still produces a crash dump. On Unix systems, sending an emulator process a SIGUSR1 signal also forces a crash dump.

```
+dcg DecentralizedCounterGroupsLimit
```

Limits the number of decentralized counter groups used by decentralized counters optimized for update operations in the Erlang runtime system. By default, the limit is 256.

When the number of schedulers is less than or equal to the limit, each scheduler has its own group. When the number of schedulers is larger than the groups limit, schedulers share groups. Shared groups degrade the performance for updating counters while many reader groups degrade the performance for reading counters. So, the limit is a tradeoff between performance for update operations and performance for read operations. Each group consumes 64 bytes in each counter.

Notice that a runtime system using decentralized counter groups benefits from binding schedulers to logical processors, as the groups are distributed better between schedulers with this option.

This option only affects decentralized counters used for the counters that are keeping track of the memory consumption and the number of terms in ETS tables of type ordered_set with the write_concurrency option activated.

```
+e Number
```

Sets the maximum number of ETS tables. This limit is partially obsolete.

```
+ec
```

Forces option compressed on all ETS tables. Only intended for test and evaluation.
+fnl

The virtual machine works with filenames as if they are encoded using the ISO Latin-1 encoding, disallowing Unicode characters with code points > 255.

For more information about Unicode filenames, see section Unicode Filenames in the STDLIB User's Guide. Notice that this value also applies to command-line parameters and environment variables (see section Unicode in Environment and Parameters in the STDLIB User's Guide).

+fnu([w|i|e])

The virtual machine works with filenames as if they are encoded using UTF-8 (or some other system-specific Unicode encoding). This is the default on operating systems that enforce Unicode encoding, that is, Windows MacOS X and Android.

The +fnu switch can be followed by w, i, or e to control how wrongly encoded filenames are to be reported:

- w means that a warning is sent to the error_logger whenever a wrongly encoded filename is "skipped" in directory listings. This is the default.
- i means that those wrongly encoded filenames are silently ignored.
- e means that the API function returns an error whenever a wrongly encoded filename (or directory name) is encountered.

Notice that file:read_link/1 always returns an error if the link points to an invalid filename.

For more information about Unicode filenames, see section Unicode Filenames in the STDLIB User's Guide. Notice that this value also applies to command-line parameters and environment variables (see section Unicode in Environment and Parameters in the STDLIB User's Guide).

+fna([w|i|e])

Selection between +fnl and +fnu is done based on the current locale settings in the OS. This means that if you have set your terminal for UTF-8 encoding, the filesystem is expected to use the same encoding for filenames. This is the default on all operating systems, except Android, MacOS X and Windows.

The +fna switch can be followed by w, i, or e. This has effect if the locale settings cause the behavior of +fnu to be selected; see the description of +fnu above. If the locale settings cause the behavior of +fnl to be selected, then w, i, or e have no effect.

For more information about Unicode filenames, see section Unicode Filenames in the STDLIB User's Guide. Notice that this value also applies to command-line parameters and environment variables (see section Unicode in Environment and Parameters in the STDLIB User's Guide).

+hms Size

Sets the default heap size of processes to the size Size.

+hmbs Size

Sets the default binary virtual heap size of processes to the size Size.

+hmax Size

Sets the default maximum heap size of processes to the size Size words. Defaults to 0, which means that no maximum heap size is used. For more information, see process_flag(max_heap_size, MaxHeapSize).

+hmaxel true|false

Sets whether to send an error logger message or not for processes reaching the maximum heap size. Defaults to true. For more information, see process_flag(max_heap_size, MaxHeapSize).
+hmaxib true|false
  Sets whether to include the size of shared off-heap binaries in the sum compared against the maximum heap size. Defaults to false. For more information, see process_flag(max_heap_size, MaxHeapSize).

+hmaxk true|false
  Sets whether to kill processes reaching the maximum heap size or not. Default to true. For more information, see process_flag(max_heap_size, MaxHeapSize).

+hpds Size
  Sets the initial process dictionary size of processes to the size Size.

+hmqd off_heap|on_heap
  Sets the default value of the message_queue_data process flag. Defaults to on_heap. If +hmqd is not passed, on_heap will be the default. For more information, see process_flag(message_queue_data, MQD).

+IOp PollSets
  Sets the number of IO pollsets to use when polling for I/O. This option is only used on platforms that support concurrent updates of a pollset, otherwise the same number of pollsets are used as IO poll threads. The default is 1.

+IOt PollThreads
  Sets the number of IO poll threads to use when polling for I/O. The maximum number of poll threads allowed is 1024. The default is 1.
  A good way to check if more IO poll threads are needed is to use microstate accounting and see what the load of the IO poll thread is. If it is high it could be a good idea to add more threads.

+IOPp PollSetsPercentage
  Similar to +IOp but uses percentages to set the number of IO pollsets to create, based on the number of poll threads configured. If both +IOPp and +IOp are used, +IOPp is ignored.

+IOPt PollThreadsPercentage
  Similar to +IOt but uses percentages to set the number of IO poll threads to create, based on the number of schedulers configured. If both +IOPt and +IOt are used, +IOPt is ignored.

+IOs true|false
  Enable or disable scheduler thread poll optimization. Default is true.
  If enabled, file descriptors that are frequently read may be moved to a special pollset used by scheduler threads. The objective is to reduce the number of system calls and thereby CPU load, but it can in some cases increase scheduling latency for individual file descriptor input events.

+JPperf true|false|dump|map|fp|no_fp
  Enables or disables support for the perf profiler when running with the JIT on Linux. Defaults to false.
  This option can be combined multiple times to enable several options:

dump
  Gives perf detailed line information, so that the perf annotate feature works.

map
  Gives perf a map over all module code, letting it translate machine code addresses to Erlang source code locations. This also enables frame pointers for Erlang code so that perf can walk the call stacks of Erlang processes, which costs one extra word per stack frame.

fp
  Enables frame pointers independently of the map option.
no_fp
   Disables the frame pointers added by the map option.
true
   Enables map and dump.
false
   Disables all other options.

For more details about how to run perf see the perf support section in the BeamAsm internal documentation.

+JMsingle true|false

   Enables or disables the use of single-mapped RWX memory for JIT code. The default is to map JIT:ed machine code into two regions sharing the same physical pages, where one region is executable but not writable, and the other writable but not executable. As some tools, such as QEMU user mode emulation, cannot deal with the dual mapping, this flags allows it to be disabled. This flag is automatically enabled by the +JPerf flag.

+L

   Prevents loading information about source filenames and line numbers. This saves some memory, but exceptions do not contain information about the filenames and line numbers.

+MFlag Value

   Memory allocator-specific flags. For more information, see erts_alloc(3).

+pad true|false

   Since: OTP 25.3

   The boolean value used with the +pad parameter determines the default value of the async_dist process flag of newly spawned processes. By default, if no +pad command line option is passed, the async_dist flag will be set to false.

   The value used in runtime can be inspected by calling erlang:system_info(async_dist).

+pc Range

   Sets the range of characters that the system considers printable in heuristic detection of strings. This typically affects the shell, debugger, and io:format functions (when ~tp is used in the format string).

   Two values are supported for Range:

   latin1
      The default. Only characters in the ISO Latin-1 range can be considered printable. This means that a character with a code point > 255 is never considered printable and that lists containing such characters are displayed as lists of integers rather than text strings by tools.

   unicode
      All printable Unicode characters are considered when determining if a list of integers is to be displayed in string syntax. This can give unexpected results if, for example, your font does not cover all Unicode characters.

   See also io:printable_range/0 in STDLIB.

+P Number

   Sets the maximum number of simultaneously existing processes for this system if a Number is passed as value. Valid range for Number is [1024-134217727]

   NOTE: The actual maximum chosen may be much larger than the Number passed. Currently the runtime system often, but not always, chooses a value that is a power of 2. This might, however, be changed in the future. The actual value chosen can be checked by calling erlang:system_info(process_limit).

   The default value is 262144
+Q Number
Sets the maximum number of simultaneously existing ports for this system if a Number is passed as value. Valid range for Number is \([1024-134217727]\)

NOTE: The actual maximum chosen may be much larger than the actual Number passed. Currently the runtime system often, but not always, chooses a value that is a power of 2. This might, however, be changed in the future. The actual value chosen can be checked by calling erlang:system_info(port_limit).

The default value used is normally 65536. However, if the runtime system is able to determine maximum amount of file descriptors that it is allowed to open and this value is larger than 65536, the chosen value will increased to a value larger or equal to the maximum amount of file descriptors that can be opened.

On Windows the default value is set to 8196 because the normal OS limitations are set higher than most machines can handle.

+R ReleaseNumber
Sets the compatibility mode.

The distribution mechanism is not backward compatible by default. This flag sets the emulator in compatibility mode with an earlier Erlang/OTP release ReleaseNumber. The release number must be in the range `<current release>-2..<current release>`. This limits the emulator, making it possible for it to communicate with Erlang nodes (as well as C- and Java nodes) running that earlier release.

Note:
Ensure that all nodes (Erlang-, C-, and Java nodes) of a distributed Erlang system is of the same Erlang/OTP release, or from two different Erlang/OTP releases X and Y, where all Y nodes have compatibility mode X.

+r
Forces ETS memory block to be moved on realloc.

+rg ReaderGroupsLimit
Limits the number of reader groups used by read/write locks optimized for read operations in the Erlang runtime system. By default the reader groups limit is 64.

When the number of schedulers is less than or equal to the reader groups limit, each scheduler has its own reader group. When the number of schedulers is larger than the reader groups limit, schedulers share reader groups. Shared reader groups degrade read lock and read unlock performance while many reader groups degrade write lock performance. So, the limit is a tradeoff between performance for read operations and performance for write operations. Each reader group consumes 64 byte in each read/write lock.

Notice that a runtime system using shared reader groups benefits from binding schedulers to logical processors, as the reader groups are distributed better between schedulers.

+S Schedulers:SchedulerOnline
Sets the number of scheduler threads to create and scheduler threads to set online. The maximum for both values is 1024. If the Erlang runtime system is able to determine the number of logical processors configured and logical processors available, Schedulers defaults to logical processors configured, and SchedulersOnline defaults to logical processors available; otherwise the default values are 1. If the emulator detects that it is subject to a CPU quota, the default value for SchedulersOnline will be limited accordingly.

Schedulers can be omitted if :SchedulerOnline is not and conversely. The number of schedulers online can be changed at runtime through erlang:system_flag(schedulers_online, SchedulersOnline).
If Schedulers or SchedulersOnline is specified as a negative number, the value is subtracted from the default number of logical processors configured or logical processors available, respectively.

Specifying value 0 for Schedulers or SchedulersOnline resets the number of scheduler threads or scheduler threads online, respectively, to its default value.

+S SchedulersPercentage:SchedulersOnlinePercentage

Similar to +S but uses percentages to set the number of scheduler threads to create, based on logical processors configured, and scheduler threads to set online, based on logical processors available. Specified values must be > 0. For example, +SP 50:25 sets the number of scheduler threads to 50% of the logical processors configured, and the number of scheduler threads online to 25% of the logical processors available. SchedulersPercentage can be omitted if :SchedulersOnlinePercentage is not and conversely. The number of schedulers online can be changed at runtime through erlang:system_flag(schedulers_online, SchedulersOnline).

This option interacts with +S settings. For example, on a system with 8 logical cores configured and 8 logical cores available, the combination of the options +S 4:4 +SP 50:25 (in either order) results in 2 scheduler threads (50% of 4) and 1 scheduler thread online (25% of 4).

+SDcpu DirtyCPUSchedulers:DirtyCPUSchedulersOnline

Sets the number of dirty CPU scheduler threads to create and dirty CPU scheduler threads to set online. The maximum for both values is 1024, and each value is further limited by the settings for normal schedulers:

- The number of dirty CPU scheduler threads created cannot exceed the number of normal scheduler threads created.
- The number of dirty CPU scheduler threads online cannot exceed the number of normal scheduler threads online.

For details, see the +S and +SP. By default, the number of dirty CPU scheduler threads created equals the number of normal scheduler threads created, and the number of dirty CPU scheduler threads online equals the number of normal scheduler threads online. DirtyCPUSchedulers can be omitted if :DirtyCPUSchedulersOnline is not and conversely. The number of dirty CPU schedulers online can be changed at runtime through erlang:system_flag(dirty_cpu_schedulers_online, DirtyCPUSchedulersOnline).

The amount of dirty CPU schedulers is limited by the amount of normal schedulers in order to limit the effect on processes executing on ordinary schedulers. If the amount of dirty CPU schedulers was allowed to be unlimited, dirty CPU bound jobs would potentially starve normal jobs.

Typical users of the dirty CPU schedulers are large garbage collections, json protocol encode/decoders written as nifs and matrix manipulation libraries.

You can use msacc(3) in order to see the current load of the dirty CPU schedulers threads and adjust the number used accordingly.

+SDPcpu DirtyCPUSchedulersPercentage:DirtyCPUSchedulersOnlinePercentage

Similar to +SDcpu but uses percentages to set the number of dirty CPU scheduler threads to create and the number of dirty CPU scheduler threads to set online. Specified values must be > 0. For example, +SDPcpu 50:25 sets the number of dirty CPU scheduler threads to 50% of the logical processors configured and the number of dirty CPU scheduler threads online to 25% of the logical processors available. DirtyCPUSchedulersPercentage can be omitted if :DirtyCPUSchedulersOnlinePercentage is not and conversely. The number of dirty CPU schedulers online can be changed at runtime through erlang:system_flag(dirty_cpu_schedulers_online, DirtyCPUSchedulersOnline).

This option interacts with +SDcpu settings. For example, on a system with 8 logical cores configured and 8 logical cores available, the combination of the options +SDcpu 4:4 +SDPcpu 50:25 (in either order) results in 2 dirty CPU scheduler threads (50% of 4) and 1 dirty CPU scheduler thread online (25% of 4).
**+SDio DirtyIOSchedulers**

Sets the number of dirty I/O scheduler threads to create. Valid range is 1-1024. By default, the number of dirty I/O scheduler threads created is 10.

The amount of dirty IO schedulers is not limited by the amount of normal schedulers like the amount of dirty CPU schedulers. This since only I/O bound work is expected to execute on dirty I/O schedulers. If the user should schedule CPU bound jobs on dirty I/O schedulers, these jobs might starve ordinary jobs executing on ordinary schedulers.

Typical users of the dirty IO schedulers are reading and writing to files.

You can use msacc(3) in order to see the current load of the dirty IO schedulers threads and adjust the number used accordingly.

**+sFlag Value**

Scheduling specific flags.

**+sbt BindType**

Sets scheduler bind type.

Schedulers can also be bound using flag +stbt. The only difference between these two flags is how the following errors are handled:

- Binding of schedulers is not supported on the specific platform.
- No available CPU topology. That is, the runtime system was not able to detect the CPU topology automatically, and no user-defined CPU topology was set.

If any of these errors occur when +sbt has been passed, the runtime system prints an error message, and refuses to start. If any of these errors occur when +stbt has been passed, the runtime system silently ignores the error, and start up using unbound schedulers.

Valid BindTypes:

- u unbound - Schedulers are not bound to logical processors, that is, the operating system decides where the scheduler threads execute, and when to migrate them. This is the default.
- ns no_spread - Schedulers with close scheduler identifiers are bound as close as possible in hardware.
- ts thread_spread - Thread refers to hardware threads (such as Intel's hyper-threads). Schedulers with low scheduler identifiers, are bound to the first hardware thread of each core, then schedulers with higher scheduler identifiers are bound to the second hardware thread of each core, and so on.
- ps processor_spread - Schedulers are spread like thread_spread, but also over physical processor chips.
- s spread - Schedulers are spread as much as possible.
- nnts no_node_thread_spread - Like thread_spread, but if multiple Non-Uniform Memory Access (NUMA) nodes exist, schedulers are spread over one NUMA node at a time, that is, all logical processors of one NUMA node are bound to schedulers in sequence.
- nnps no_node_processor_spread - Like processor_spread, but if multiple NUMA nodes exist, schedulers are spread over one NUMA node at a time, that is, all logical processors of one NUMA node are bound to schedulers in sequence.
tnnps
  thread_no_node_processor_spread - A combination of thread_spread, and
  no_node_processor_spread. Schedulers are spread over hardware threads across NUMA
  nodes, but schedulers are only spread over processors internally in one NUMA node at a time.

db
  default_bind - Binds schedulers the default way. Defaults to
  thread_no_node_processor_spread (which can change in the future).

Binding of schedulers is only supported on newer Linux, Solaris, FreeBSD, and Windows systems. If no CPU topology is available when flag +sbt is processed and BindType is any other type than u, the runtime system fails to start. CPU topology can be defined using flag +sct. Notice that flag +sct can have to be passed before flag +sbt on the command line (if no CPU topology has been automatically detected).

The runtime system does by default not bind schedulers to logical processors.

Note:

If the Erlang runtime system is the only operating system process that binds threads to logical processors, this improves the performance of the runtime system. However, if other operating system processes (for example another Erlang runtime system) also bind threads to logical processors, there can be a performance penalty instead. This performance penalty can sometimes be severe. If so, you are advised not to bind the schedulers.

How schedulers are bound matters. For example, in situations when there are fewer running processes than schedulers online, the runtime system tries to migrate processes to schedulers with low scheduler identifiers. The more the schedulers are spread over the hardware, the more resources are available to the runtime system in such situations.

Note:

If a scheduler fails to bind, this is often silently ignored, as it is not always possible to verify valid logical processor identifiers. If an error is reported, it is reported to the error_logger. If you want to verify that the schedulers have bound as requested, call erlang:system_info(scheduler_bindings).

+sbwt none|very_short|short|medium|long|very_long

Sets scheduler busy wait threshold. Defaults to medium. The threshold determines how long schedulers are to busy wait when running out of work before going to sleep.

Note:

This flag can be removed or changed at any time without prior notice.

+sbwtdcpu none|very_short|short|medium|long|very_long

As +sbwt but affects dirty CPU schedulers. Defaults to short.

Note:

This flag can be removed or changed at any time without prior notice.
As +sbwt but affects dirty IO schedulers. Defaults to short.

Note:
This flag can be removed or changed at any time without prior notice.

+scl true|false
Enables or disables scheduler compaction of load. By default scheduler compaction of load is enabled. When enabled, load balancing strives for a load distribution, which causes as many scheduler threads as possible to be fully loaded (that is, not run out of work). This is accomplished by migrating load (for example, runnable processes) into a smaller set of schedulers when schedulers frequently run out of work. When disabled, the frequency with which schedulers run out of work is not taken into account by the load balancing logic.
+scl false is similar to +sub true, but +sub true also balances scheduler utilization between schedulers.

+sct CpuTopology
• <Id> = integer(); when 0 <= <Id> <= 65535
• <IdRange> = <Id>--<Id>
• <IdOrIdRange> = <Id> | <IdRange>
• <IdList> = <IdOrIdRange>,<IdOrIdRange> | <IdOrIdRange>
• <LogicalIds> = L<IdList>
• <ThreadIds> = T<IdList> | t<IdList>
• <CoreIds> = C<IdList> | c<IdList>
• <ProcessorIds> = P<IdList> | p<IdList>
• <NodeIds> = N<IdList> | n<IdList>
• <IdDefs> = <LogicalIds><ThreadIds><CoreIds><ProcessorIds><NodeIds> | <LogicalIds><ThreadIds><CoreIds><NodeIds><ProcessorIds>
• CpuTopology = <IdDefs>:<IdDefs> | <IdDefs>

Sets a user-defined CPU topology. The user-defined CPU topology overrides any automatically detected CPU topology. The CPU topology is used when binding schedulers to logical processors.

Uppercase letters signify real identifiers and lowercase letters signify fake identifiers only used for description of the topology. Identifiers passed as real identifiers can be used by the runtime system when trying to access specific hardware; if they are incorrect the behavior is undefined. Faked logical CPU identifiers are not accepted, as there is no point in defining the CPU topology without real logical CPU identifiers. Thread, core, processor, and node identifiers can be omitted. If omitted, the thread ID defaults to t0, the core ID defaults to c0, the processor ID defaults to p0, and the node ID is left undefined. Either each logical processor must belong to only one NUMA node, or no logical processors must belong to any NUMA nodes.

Both increasing and decreasing <IdRange>s are allowed.

NUMA node identifiers are system wide. That is, each NUMA node on the system must have a unique identifier. Processor identifiers are also system wide. Core identifiers are processor wide. Thread identifiers are core wide.

The order of the identifier types implies the hierarchy of the CPU topology. The valid orders are as follows:
• <LogicalIds><ThreadIds><CoreIds><ProcessorIds><NodeIds>, that is, thread is part of a core that is part of a processor, which is part of a NUMA node.
•  \(<\text{LogicalIds}><\text{ThreadIds}><\text{CoreIds}><\text{NodeIds}><\text{ProcessorIds}>\), that is, thread is part of a core that is part of a NUMA node, which is part of a processor.

A CPU topology can consist of both processor external, and processor internal NUMA nodes as long as each logical processor belongs to only one NUMA node. If \(<\text{ProcessorIds}>\) is omitted, its default position is before \(<\text{NodeIds}>\). That is, the default is processor external NUMA nodes.

If a list of identifiers is used in an \(<\text{IdDefs}>\):
  •  \(<\text{LogicalIds}>\) must be a list of identifiers.
  •  At least one other identifier type besides \(<\text{LogicalIds}>\) must also have a list of identifiers.
  •  All lists of identifiers must produce the same number of identifiers.

A simple example. A single quad core processor can be described as follows:

```erl
% erl +sct L0-3c0-3
1> erlang:system_info(cpu_topology).
[processor,{{core,{{logical,0}},
           {core,{{logical,1}},
           {core,{{logical,2}},
           {core,{{logical,3}}}}}}}
```

A more complicated example with two quad core processors, each processor in its own NUMA node. The ordering of logical processors is a bit weird. This to give a better example of identifier lists:

```erl
% erl +sct L0-1,3-2c0-3p0N0:L7,4-6c0-3p1N1
1> erlang:system_info(cpu_topology).
[{{node,{{processor,{{core,{{logical,0}}},
                        {core,{{logical,1}}},
                        {core,{{logical,3}}}},
                        {core,{{logical,2}}}}}},
   {node,{{processor,{{core,{{logical,7}}},
                        {core,{{logical,4}}},
                        {core,{{logical,6}}}},
                        {core,{{logical,5}}}}}}}
```

As long as real identifiers are correct, it is OK to pass a CPU topology that is not a correct description of the CPU topology. When used with care this can be very useful. This to trick the emulator to bind its schedulers as you want. For example, if you want to run multiple Erlang runtime systems on the same machine, you want to reduce the number of schedulers used and manipulate the CPU topology so that they bind to different logical CPUs. An example, with two Erlang runtime systems on a quad core machine:

```erl
% erl +sct L0-3c0-3 +sbt db +S3:2 -detached -noinput -noshell -sname one
% erl +sct L3-6c0-3 +sbt db +S3:2 -detached -noinput -noshell -sname two
```

In this example, each runtime system have two schedulers each online, and all schedulers online will run on different cores. If we change to one scheduler online on one runtime system, and three schedulers online on the other, all schedulers online will still run on different cores.

Notice that a faked CPU topology that does not reflect how the real CPU topology looks like is likely to decrease the performance of the runtime system.

For more information, see `erlang:system_info(cpu_topology)`.

`+ssrct`

Skips reading CPU topology.
Note:
Reading CPU topology slows down startup when starting many parallel instances of ERTS on systems with large amount of cores, using this flag might speed up execution in such scenarios.

+sffw Interval
Sets scheduler-forced wakeup interval. All run queues are scanned each Interval milliseconds. While there are sleeping schedulers in the system, one scheduler is woken for each non-empty run queue found. Interval default to 0, meaning this feature is disabled.

Note:
This feature has been introduced as a temporary workaround for long-executing native code, and native code that does not bump reductions properly in OTP. When these bugs have been fixed, this flag will be removed.

+spp Bool
Sets default scheduler hint for port parallelism. If set to true, the virtual machine schedules port tasks when it improves parallelism in the system. If set to false, the virtual machine tries to perform port tasks immediately, improving latency at the expense of parallelism. Default to false. The default used can be inspected in runtime by calling erlang:system_info(port_parallelism). The default can be overridden on port creation by passing option parallelism to erlang:open_port/2.

+sps size
Suggested stack size, in kilowords, for scheduler threads. Valid range is 20-8192 kilowords. The default suggested stack size is 128 kilowords.

+spsdcpu size
Suggested stack size, in kilowords, for dirty CPU scheduler threads. Valid range is 20-8192 kilowords. The default suggested stack size is 40 kilowords.

+spsdio size
Suggested stack size, in kilowords, for dirty IO scheduler threads. Valid range is 20-8192 kilowords. The default suggested stack size is 40 kilowords.

+sbs BindType
Tries to set the scheduler bind type. The same as flag +sbt except how some errors are handled. For more information, see +sbt.

+sub true|false
Enables or disables scheduler utilization balancing of load. By default scheduler utilization balancing is disabled and instead scheduler compaction of load is enabled, which strives for a load distribution that causes as many scheduler threads as possible to be fully loaded (that is, not run out of work). When scheduler utilization balancing is enabled, the system instead tries to balance scheduler utilization between schedulers. That is, strive for equal scheduler utilization on all schedulers.
+sub true is only supported on systems where the runtime system detects and uses a monotonically increasing high-resolution clock. On other systems, the runtime system fails to start.
+sub true implies +scl false. The difference between +sub true and +scl false is that +scl false does not try to balance the scheduler utilization.
+swct very_eager|eager|medium|lazy|very_lazy

Sets scheduler wake cleanup threshold. Defaults to medium. Controls how eager schedulers are to be requesting wakeup because of certain cleanup operations. When a lazy setting is used, more outstanding cleanup operations can be left undone while a scheduler is idling. When an eager setting is used, schedulers are more frequently woken, potentially increasing CPU-utilization.

**Note:**
This flag can be removed or changed at any time without prior notice.

+sws default|legacy

Sets scheduler wakeup strategy. Default strategy changed in ERTS 5.10 (Erlang/OTP R16A). This strategy was known as proposal in Erlang/OTP R15. The legacy strategy was used as default from R13 up to and including R15.

**Note:**
This flag can be removed or changed at any time without prior notice.

+swt very_low|low|medium|high|very_high

Sets scheduler wakeup threshold. Defaults to medium. The threshold determines when to wake up sleeping schedulers when more work than can be handled by currently awake schedulers exists. A low threshold causes earlier wakeups, and a high threshold causes later wakeups. Early wakeups distribute work over multiple schedulers faster, but work does more easily bounce between schedulers.

**Note:**
This flag can be removed or changed at any time without prior notice.

+swtdcpu very_low|low|medium|high|very_high

As +swt but affects dirty CPU schedulers. Defaults to medium.

**Note:**
This flag can be removed or changed at any time without prior notice.

+swtdio very_low|low|medium|high|very_high

As +swt but affects dirty IO schedulers. Defaults to medium.

**Note:**
This flag can be removed or changed at any time without prior notice.

+t size

Sets the maximum number of atoms the virtual machine can handle. Defaults to 1,048,576.

+T Level

Enables modified timing and sets the modified timing level. Valid range is 0-9. The timing of the runtime system is changed. A high level usually means a greater change than a low level. Changing the timing can be very useful for finding timing-related bugs.
Modified timing affects the following:

**Process spawning**
A process calling `spawn`, `spawn_link`, `spawn_monitor`, or `spawn_opt` is scheduled out immediately after completing the call. When higher modified timing levels are used, the caller also sleeps for a while after it is scheduled out.

**Context reductions**
The number of reductions a process is allowed to use before it is scheduled out is increased or reduced.

**Input reductions**
The number of reductions performed before checking I/O is increased or reduced.

---

**Note:**

Performance suffers when modified timing is enabled. This flag is **only** intended for testing and debugging. `return_to` and `return_from` trace messages are lost when tracing on the spawn BIFs.

This flag can be removed or changed at any time without prior notice.

---

**+v**

Verbose.

**+V**

Makes the emulator print its version number.

**+W w | i | e**

Sets the mapping of warning messages for `error_logger`. Messages sent to the error logger using one of the warning routines can be mapped to errors (`+W e`), warnings (`+W w`), or information reports (`+W i`). Defaults to warnings. The current mapping can be retrieved using `error_logger:warning_map/0`. For more information, see `error_logger:warning_map/0` in Kernel.

---

**Flag Value**

**Miscellaneous flags:**

**+zdbbl size**

Sets the distribution buffer busy limit (`dist_buf_busy_limit`) in kilobytes. Valid range is 1-2097151. Defaults to 1024.

A larger buffer limit allows processes to buffer more outgoing messages over the distribution. When the buffer limit has been reached, sending processes will be suspended until the buffer size has shrunk. The buffer limit is per distribution channel. A higher limit gives lower latency and higher throughput at the expense of higher memory use.

This limit only affects processes that have disabled *fully asynchronous distributed signaling*.

**+zdntgc time**

Sets the delayed node table garbage collection time (`delayed_node_table_gc`) in seconds. Valid values are either `infinity` or an integer in the range 0-100000000. Defaults to 60.

Node table entries that are not referred linger in the table for at least the amount of time that this parameter determines. The lingering prevents repeated deletions and insertions in the tables from occurring.

**+zosrl limit**

Sets a limit on the amount of outstanding requests made by a system process orchestrating system wide changes. Valid range of this limit is [1, 134217727]. See
erl

erlang:system_flag(outstanding_system_requests_limit, Limit) for more information.

Environment Variables

ERL_CRASH_DUMP

If the emulator needs to write a crash dump, the value of this variable is the filename of the crash dump file. If the variable is not set, the name of the crash dump file is erl_crash.dump in the current directory.

ERL_CRASH_DUMP_NICE

Unix systems: If the emulator needs to write a crash dump, it uses the value of this variable to set the nice value for the process, thus lowering its priority. Valid range is 1-39 (higher values are replaced with 39). The highest value, 39, gives the process the lowest priority.

ERL_CRASH_DUMP_SECONDS

Unix systems: This variable gives the number of seconds that the emulator is allowed to spend writing a crash dump. When the given number of seconds have elapsed, the emulator is terminated.

ERL_CRASH_DUMP_SECONDS=0

If the variable is set to 0 seconds, the runtime system does not even attempt to write the crash dump file. It only terminates. This is the default if option -heart is passed to erl and ERL_CRASH_DUMP_SECONDS is not set.

ERL_CRASH_DUMP_SECONDS=S

If the variable is set to a positive value S, wait for S seconds to complete the crash dump file and then terminates the runtime system with a SIGALRM signal.

ERL_CRASH_DUMP_SECONDS=-1

A negative value causes the termination of the runtime system to wait indefinitely until the crash dump file has been completely written. This is the default if option -heart is not passed to erl and ERL_CRASH_DUMP_SECONDS is not set.

See also heart(3).

ERL_CRASH_DUMP_BYTES

This variable sets the maximum size of a crash dump file in bytes. The crash dump will be truncated if this limit is exceeded. If the variable is not set, no size limit is enforced by default. If the variable is set to 0, the runtime system does not even attempt to write a crash dump file.

Introduced in ERTS 8.1.2 (Erlang/OTP 19.2).

ERL_AFLAGS

The content of this variable is added to the beginning of the command line for erl.

Flag -extra is treated in a special way. Its scope ends at the end of the environment variable content. Arguments following an -extra flag are moved on the command line into section -extra, that is, the end of the command line following an -extra flag.

ERL_ZFLAGS and ERL_FLAGS

The content of these variables are added to the end of the command line for erl.

Flag -extra is treated in a special way. Its scope ends at the end of the environment variable content. Arguments following an -extra flag are moved on the command line into section -extra, that is, the end of the command line following an -extra flag.

ERL_LIBS

Contains a list of additional library directories that the code server searches for applications and adds to the code path; see code(3).
ERL_EPMD_ADDRESS
Can be set to a comma-separated list of IP addresses, in which case the epmd daemon listens only on the specified address(es) and on the loopback address (which is implicitly added to the list if it has not been specified).

ERL_EPMD_PORT
Can contain the port number to use when communicating with epmd. The default port works fine in most cases. A different port can be specified to allow nodes of independent clusters to co-exist on the same host. All nodes in a cluster must use the same epmd port number.

Signals
On Unix systems, the Erlang runtime will interpret two types of signals.

SIGUSR1
A SIGUSR1 signal forces a crash dump.

SIGTERM
A SIGTERM will produce a stop message to the init process. This is equivalent to a init:stop/0 call.

Introduced in ERTS 8.3 (Erlang/OTP 19.3)
The signal SIGUSR2 is reserved for internal usage. No other signals are handled.

Configuration
The standard Erlang/OTP system can be reconfigured to change the default behavior on startup.

The .erlang startup file
When Erlang/OTP is started, the system searches for a file named .erlang in the user's home directory and then filename:basedir(user_config, "erlang").

If an .erlang file is found, it is assumed to contain valid Erlang expressions. These expressions are evaluated as if they were input to the shell.

A typical .erlang file contains a set of search paths, for example:

```
io:format("executing user profile in $HOME/.erlang\n",[]).
code:add_path("/home/calvin/test/ebin").
code:add_path("/home/hobbes/bigappl-1.2/ebin").
io:format(".erlang rc finished\n",[]).
```

user_default and shell_default
Functions in the shell that are not prefixed by a module name are assumed to be functional objects (funs), built-in functions (BIFs), or belong to the module user_default or shell_default.

To include private shell commands, define them in a module user_default and add the following argument as the first line in the .erlang file:

```
code:load_abs("....\user_default").
```

erl
If the contents of .erlang are changed and a private version of user_default is defined, the Erlang/OTP environment can be customized. More powerful changes can be made by supplying command-line arguments in the startup script erl. For more information, see init(3).
See Also

epmd(1), erl_prim_loader(3), erts_alloc(3), init(3), application(3), auth(3), code(3), erl_boot_server(3), heart(3), net_kernel(3), make(3)
erlang

Erlang module

By convention, most Built-In Functions (BIFs) and all predefined types are included in this module. Some of the BIFs and all of the predefined types are viewed more or less as part of the Erlang programming language and are **auto-imported**. Thus, it is not necessary to specify the module name. For example, the calls `atom_to_list(erlang)` and `erlang:atom_to_list(erlang)` are identical.

Auto-imported BIFs are listed without module prefix. BIFs listed with module prefix are not auto-imported.

Predefined types are listed in the Predefined datatypes section of this reference manual and in the Types and Function Specifications section of the Erlang Reference Manual.

BIFs can fail for various reasons. All BIFs fail with reason **badarg** if they are called with arguments of an incorrect type. The other reasons are described in the description of each individual BIF.

Some BIFs can be used in guard tests and are marked with "Allowed in guard tests".

**Data Types**

**Predefined datatypes**

`any() = any()`

All possible Erlang terms. Synonym for `term()`.

`arity() = arity()`

The arity of a function or type.

`atom() = atom()`

An Erlang atom.

`binary() = <<:_:*8>>`

An Erlang binary, that is, a bitstring with a size divisible by 8.

`bitstring() = <<:_:*1>>`

An Erlang bitstring.

`boolean() = true | false`

A boolean value.

`byte() = 0..255`

A byte of data represented by an integer.

`char() = 0..1114111`

An ASCII character or a unicode codepoint presented by an integer.

`float() = float()`

An Erlang float.

`function() = function()`

An Erlang fun.

`identifier() = pid() | port() | reference()`

An unique identifier for some entity, for example a process, port or monitor.
integer() = integer()

An Erlang integer.

iodata() = iolist() | binary()

A binary or list containing bytes and/or iodata. This datatype is used to represent data that is meant to be output using any I/O module. For example: file:write/2 or gen_tcp:send/2.

To convert an iodata() term to binary() you can use iolist_to_binary/2. To transcode a string() or unicode:chardata() to iodata() you can use unicode:characters_to_binary/1.

iolist() =
    maybe_improper_list(byte() | binary() | iolist(),
                       binary() | [])

A list containing bytes and/or iodata. This datatype is used to represent data that is meant to be output using any I/O module. For example: file:write/2 or gen_tcp:send/2.

In most use cases you want to use iodata() instead of this type.

list() = [any()]

An Erlang list containing terms of any type.

list(ContentType) = [ContentType]

An Erlang list containing terms of the type ContentType.

map() = #{any() => any()}

An Erlang map containing any number of key and value associations.

maybe_improper_list() = maybe_improper_list(any(), any())

An Erlang list that is not guaranteed to end with a [], and where the list elements can be of any type.

maybe_improper_list(ContentType, TerminationType) =
    maybe_improper_list(ContentType, TerminationType)

An Erlang list, that is not guaranteed to end with a [], and where the list elements are of the type ContentType.

mfa() = {module(), atom(), arity()}

A three-tuple representing a Module:Function/Arity function signature.

module() = atom()

An Erlang module represented by an atom.

neg_integer() = integer() =< -1

A negative integer.

nil() = []

The empty list().

no_return() = none()

The type used to show that a function will never return a value, that is it will always throw an exception.

node() = atom()

An Erlang node represented by an atom.

non_neg_integer() = integer() >= 0

A non-negative integer, that is any positive integer or 0.
none() = none()

This type is used to show that a function will never return a value; that is it will always throw an exception. In a spec, use no_return() for the sake of clarity.

nonempty_binary() = <<_:8, _:*8>>

A binary() that contains some data.

nonempty_bitstring() = <<_:1, _:*1>>

A bitstring() that contains some data.

nonempty_improper_list(ContentType, TerminationType) =
    nonempty_improper_list(ContentType, TerminationType)

A maybe_improper_list/2 that contains some items.

nonempty_list() = [any(), ...]

A list() that contains some items.

nonempty_list(ContentType) = [ContentType, ...]

A list(ContentType) that contains some items.

nonempty_maybe_improper_list() =
    nonempty_maybe_improper_list(any(), any())

A maybe_improper_list() that contains some items.

nonempty_maybe_improper_list(ContentType, TerminationType) =
    nonempty_maybe_improper_list(ContentType, TerminationType)

A maybe_improper_list(ContentType, TerminationType) that contains some items.

nonempty_string() = [char(), ...]

A string() that contains some characters.

number() = integer() | float()

An Erlang number.

pid() = pid()

An Erlang process identifier.

port() = port()

An Erlang port identifier.

pos_integer() = integer() >= 1

An integer greater than zero.

reference() = reference()

An Erlang reference.

string() = [char()]

A character string represented by a list of ASCII characters or unicode codepoints.

term() = any()

All possible Erlang terms. Synonym for any().

timeout() = infinity | integer() >= 0

A timeout value that can be passed to a receive expression.
tuple() = tuple()
An Erlang tuple.

Data Types

Other Datatypes
ext_binary() = binary()
A binary data object, structured according to the Erlang external term format.
ext_iovec() = iovec()
A term of type iovec(), structured according to the Erlang external term format.
iovec() = [binary()]
A list of binaries. This datatype is useful to use together with enif_inspect_iovec.
message_queue_data() = off_heap | on_heap
See process_flag(message_queue_data, MQD).
monitor_option() =
\{alias, explicit_unalias | demonitor | reply_demonitor\} |
\{tag, term()\}
See monitor/3.
timestamp() =
\{MegaSecs :: integer() >= 0,
  Secs :: integer() >= 0,
  MicroSecs :: integer() >= 0\}
See erlang:timestamp/0.
time_unit() =
  integer() >= 1 |
  second | millisecond | microsecond | nanosecond | native | perf_counter |
  deprecated_time_unit()
Supported time unit representations:
PartsPerSecond :: integer() >= 1
  Time unit expressed in parts per second. That is, the time unit equals 1/PartsPerSecond second.
  second
    Symbolic representation of the time unit represented by the integer 1.
millisecond
    Symbolic representation of the time unit represented by the integer 1000.
microsecond
    Symbolic representation of the time unit represented by the integer 1000_000.
nanosecond
    Symbolic representation of the time unit represented by the integer 1000_000_000.
native
    Symbolic representation of the native time unit used by the Erlang runtime system.
The **native** time unit is determined at runtime system start, and remains the same until the runtime system terminates. If a runtime system is stopped and then started again (even on the same machine), the **native** time unit of the new runtime system instance can differ from the **native** time unit of the old runtime system instance.

One can get an approximation of the **native** time unit by calling `erlang:convert_time_unit(1, second, native)`. The result equals the number of whole **native** time units per second. If the number of **native** time units per second does not add up to a whole number, the result is rounded downwards.

**Note:**

The value of the **native** time unit gives you more or less no information about the quality of time values. It sets a limit for the resolution and for the precision of time values, but it gives no information about the accuracy of time values. The resolution of the **native** time unit and the resolution of time values can differ significantly.

**perf_counter**

Symbolic representation of the performance counter time unit used by the Erlang runtime system. The **perf_counter** time unit behaves much in the same way as the **native** time unit. That is, it can differ between runtime restarts. To get values of this type, call `os:perf_counter/0`.

**deprecated_time_unit()**

Deprecated symbolic representations kept for backwards-compatibility.

The `time_unit/0` type can be extended. To convert time values between time units, use `erlang:convert_time_unit/3`.

```erlang
deprecated_time_unit() = seconds | milli_seconds | micro_seconds | nano_seconds
```

The `time_unit()` type also consist of the following **deprecated** symbolic time units:

- **seconds**
  - Same as `second`.
- **milli_seconds**
  - Same as `millisecond`.
- **micro_seconds**
  - Same as `microsecond`.
- **nano_seconds**
  - Same as `nanosecond`.

**dist_handle()**

An opaque handle identifying a distribution channel.

**nif_resource()**

An opaque handle identifying a NIF resource object.

**spawn_opt_option()**

```erlang
spawn_opt_option() = link | monitor |
  {monitor, MonitorOpts :: [monitor_option()]} | 
  {priority, Level :: priority_level()} | 
  {fullsweep_after, Number :: integer() >= 0} | 
  {min_heap_size, Size :: integer() >= 0} |
```
{min_bin_vheap_size, VSize :: integer() >= 0} |
{max_heap_size, Size :: max_heap_size()} |
{message_queue_data, MQD :: message_queue_data()} |
{async_dist, Enabled :: boolean()}

Options for spawn_opt().

priority_level() = low | normal | high | max

Process priority level. For more info see process_flag(priority, Level)

max_heap_size() =
integer() >= 0 |
#{size => integer() >= 0, |
kill => boolean(), |
error_logger => boolean(), |
include_shared_binaries => boolean()}

Process max heap size configuration. For more info see process_flag(max_heap_size, MaxHeapSize)

message_queue_data() = off_heap | on_heap

Process message queue data configuration. For more information, see process_flag(message_queue_data, MQD)

stacktrace() =
[module(), |
atom(), |
arity() | [term()], |
[stacktrace_extrainfo()]] |
{function(), arity() | [term()], [stacktrace_extrainfo()]]

stacktrace_extrainfo() =
{line, integer() >= 1} |
{file, unicode:chardata()} |
{error_info, |
#{module => module(), function => atom(), cause => term()}} |
{atom(), term()}

An Erlang stacktrace as described by Errors and Error Handling section in the Erlang Reference Manual.

send_destination() =
pid() |
reference() |
port() |
{RegName :: atom()} |
{RegName :: atom(), Node :: node()}

The destination for a send operation, can be a remote or local process identifier, a (local) port, a reference denoting a process alias, a locally registered name, or a tuple {RegName, Node} for a registered name at another node.

Exports

abs(Float) -> float()
abs(Int) -> integer() >= 0

Types:
Int = integer()

Returns an integer or float that is the arithmetical absolute value of Float or Int, for example:
> abs(-3.33).
3.33
> abs(-3).
3

Allowed in guard tests.

erlang:adler32(Data) -> integer() >= 0
Types:
   Data = iodata()
Computes and returns the adler32 checksum for Data.

erlang:adler32(OldAdler, Data) -> integer() >= 0
Types:
   OldAdler = integer() >= 0
   Data = iodata()
Continues computing the adler32 checksum by combining the previous checksum, OldAdler, with the checksum of Data.
The following code:

X = erlang:adler32(Data1),
Y = erlang:adler32(X,Data2).

assigns the same value to Y as this:

Y = erlang:adler32([Data1,Data2]).

erlang:adler32_combine(FirstAdler, SecondAdler, SecondSize) ->
   integer() >= 0
Types:
   FirstAdler = SecondAdler = SecondSize = integer() >= 0
Combines two previously computed adler32 checksums. This computation requires the size of the data object for the second checksum to be known.
The following code:

Y = erlang:adler32(Data1),
Z = erlang:adler32_combine(Y,Data2).

assigns the same value to Z as this:

X = erlang:adler32(Data1),
Y = erlang:adler32(Data2),
Z = erlang:adler32_combine(X,Y,iolist_size(Data2)).

alias() -> Alias
alias(Opts) -> Alias
Types:
Alias = reference()
Opts = [explicit_unalias | reply]

Create an alias which can be used when sending messages to the process that created the alias. When the alias has been deactivated, messages sent using the alias will be dropped. An alias can be deactivated using unalias/1. Currently available options for alias/1:

explicit_unalias
The alias can only be deactivated via a call to unalias/1. This is also the default behaviour if no options are passed or if alias/0 is called.

reply
The alias will be automatically deactivated when a reply message sent via the alias is received. The alias can also still be deactivated via a call to unalias/1.

Example:

```erlang
calendar() ->
    receive
        {request, AliasReqId, Request} ->
            Result = perform_request(Request),
            AliasReqId ! {reply, AliasReqId, Result}
    end,
    calendar().
```

```erlang
calendar(ServerPid, Request) ->
    AliasReqId = alias([reply]),
    ServerPid ! {request, AliasReqId, Request},
    % Alias will be automatically deactivated if we receive a reply
    % since we used the 'reply' option...
    receive
        {reply, AliasReqId, Result} -> Result
    after 5000 ->
        unalias(AliasReqId),
        % Flush message queue in case the reply arrived
        % just before the alias was deactivated...
        receive {reply, AliasReqId, Result} -> Result
            after 0 -> exit(timeout)
        end
    end.
```

Note that both the server and the client in this example must be executing on at least OTP 24 systems in order for this to work.

For more information on process aliases see the Process Aliases section of the Erlang Reference Manual.

erlang:append_element(Tuple1, Term) -> Tuple2
Types:

    Tuple1 = Tuple2 = tuple()
    Term = term()

Returns a new tuple that has one element more than Tuple1, and contains the elements in Tuple1 followed by Term as the last element. Semantically equivalent to list_to_tuple(tuple_to_list(Tuple1) ++ [Term]), but much faster. Example:

```erlang
got = 10000000;

% Should be much faster than the following...
> lists:foldl(fun(X,Y) -> [X,Y] end, [], [1..got]).
% ... but equivalent.
```

`erlang:append_element({one, two}, three).`

`{one, two, three}`
apply(Fun, Args) -> term()

Types:
  Fun = function()
  Args = [term()]

Calls a fun, passing the elements in Args as arguments.

If the number of elements in the arguments are known at compile time, the call is better written as Fun(Arg1, Arg2, ... ArgN).

**Warning:**

Earlier, Fun could also be specified as {Module, Function}, equivalent to apply(Module, Function, Args). This use is deprecated and will stop working in a future release.

apply(Module, Function, Args) -> term()

Types:
  Module = module()
  Function = atom()
  Args = [term()]

Returns the result of applying Function in Module to Args. The applied function must be exported from Module. The arity of the function is the length of Args. Example:

```
> apply(lists, reverse, [[a, b, c]]).
[c,b,a]
> apply(erlang, atom_to_list, ['Erlang']).
"Erlang"
```

If the number of arguments are known at compile time, the call is better written as Module:Function(Arg1, Arg2, ..., ArgN).

Failure: error_handler:undefined_function/3 is called if the applied function is not exported. The error handler can be redefined (see process_flag/2). If error_handler is undefined, or if the user has redefined the default error_handler so the replacement module is undefined, an error with reason undef is generated.

atom_to_binary(Atom) -> binary()

Types:
  Atom = atom()

The same as atom_to_binary(Atom, utf8).

atom_to_binary(Atom, Encoding) -> binary()

Types:
  Atom = atom()
  Encoding = latin1 | unicode | utf8

Returns a binary corresponding to the text representation of Atom. If Encoding is latin1, one byte exists for each character in the text representation. If Encoding is utf8 or unicode, the characters are encoded using UTF-8 where characters may require multiple bytes.
**Change:**

As from Erlang/OTP 20, atoms can contain any Unicode character and `atom_to_binary(Atom, latin1)` may fail if the text representation for `Atom` contains a Unicode character > 255.

Example:

```erlang
code(ex1).

> atom_to_binary('Erlang', latin1).
"Erlang"
```

**atom_to_list(Atom) -> string()**

Types:

- `Atom = atom()`

Returns a list of unicode code points corresponding to the text representation of `Atom`, for example:

```erlang
code(ex2).

> atom_to_list('Erlang').
"Erlang"
> atom_to_list('##').
[20320,22909]
```

See `unicode(3)` for how to convert the resulting list to different formats.

**binary_part(Subject, PosLen) -> binary()**

Types:

- `Subject = binary()`
- `PosLen = {Start :: integer() >= 0, Length :: integer()}`

Extracts the part of the binary described by `PosLen`.

Negative length can be used to extract bytes at the end of a binary, for example:

```erlang
code(ex3).

1> Bin = <<1,2,3,4,5,6,7,8,9,10>>.
2> binary_part(Bin,{byte_size(Bin), -5}).
<<6,7,8,9,10>>
```

Failure: `badarg` if `PosLen` in any way references outside the binary.

Start is zero-based, that is:

```erlang
code(ex4).

1> Bin = <<1,2,3>>
2> binary_part(Bin,{0,2}).
<<1,2>>
```

For details about the `PosLen` semantics, see `binary(3)`.

Allowed in guard tests.

**binary_part(Subject, Start, Length) -> binary()**

Types:
Subject = binary()
Start = integer() >= 0
Length = integer()
The same as binary_part(Subject, {Start, Length}).
Allowed in guard tests.

binary_to_atom(Binary) -> atom()
Types:
   Binary = binary()
The same as binary_to_atom(Binary, utf8).

binary_to_atom(Binary, Encoding) -> atom()
Types:
   Binary = binary()
   Encoding = latin1 | unicode | utf8
Returns the atom whose text representation is Binary. If Encoding is utf8 or unicode, the binary must contain valid UTF-8 sequences.

Change:
As from Erlang/OTP 20, binary_to_atom(Binary, utf8) is capable of decoding any Unicode character. Earlier versions would fail if the binary contained Unicode characters > 255.

Note:
The number of characters that are permitted in an atom name is limited. The default limits can be found in the efficiency guide (section Advanced).

Note:
There is configurable limit on how many atoms that can exist and atoms are not garbage collected. Therefore, it is recommended to consider whether binary_to_existing_atom/2 is a better option than binary_to_atom/2. The default limits can be found in efficiency guide (section Advanced).

Examples:

> binary_to_atom(<<"Erlang">>, latin1).
'Erlang'

> binary_to_atom(<<1024/utf8>>, utf8).
'Ѐ'

binary_to_existing_atom(Binary) -> atom()
Types:
**Binary** = binary()

The same as `binary_to_existing_atom(Binary, utf8)`.

*binary_to_existing_atom*(Binary, Encoding) -> atom()

Types:

- **Binary** = binary()
- Encoding = latin1 | unicode | utf8

As `binary_to_atom/2`, but the atom must exist.

The Erlang system has a configurable limit for the total number of atoms that can exist, and atoms are not garbage collected. Therefore, it is not safe to create many atoms from binaries that come from an untrusted source (for example, a file fetched from the Internet), for example, using `binary_to_atom/2`. This function is thus the appropriate option when the input binary comes from an untrusted source.

An atom exists in an Erlang system when included in a loaded Erlang module or when created programmatically (for example, by `binary_to_atom/2`). See the next note for an example of when an atom exists in the source code for an Erlang module but not in the compiled version of the same module.

Failure: `badarg` if the atom does not exist.

---

**Note:**

Note that the compiler may optimize away atoms. For example, the compiler will rewrite `atom_to_list(some_atom)` to "some_atom". If that expression is the only mention of the atom `some_atom` in the containing module, the atom will not be created when the module is loaded, and a subsequent call to `binary_to_existing_atom(<<"some_atom">>, utf8)` will fail.

---

**Note:**

The number of characters that are permitted in an atom name is limited. The default limits can be found in the efficiency guide (section Advanced).

*binary_to_float*(Binary) -> float()

Types:

- **Binary** = binary()

Returns the float whose text representation is `Binary`, for example:

```
> binary_to_float(<<"2.2017764e+0">>).
2.2017764
```

The float string format is the same as the format for Erlang float literals except for that underscores are not permitted.

Failure: `badarg` if `Binary` contains a bad representation of a float.

*binary_to_integer*(Binary) -> integer()

Types:

- **Binary** = binary()

Returns an integer whose text representation is `Binary`, for example:
> binary_to_integer(<<"123">>).
   
123

binary_to_integer/1 accepts the same string formats as list_to_integer/1.
Failure: badarg if Binary contains a bad representation of an integer.

binary_to_integer(Binary, Base) -> integer()

Types:
   Binary = binary()
   Base = 2..36

Returns an integer whose text representation in base Base is Binary, for example:

> binary_to_integer(<<"3FF">>, 16).
   
1023

binary_to_integer/2 accepts the same string formats as list_to_integer/2.
Failure: badarg if Binary contains a bad representation of an integer.

binary_to_list(Binary) -> [byte()]

Types:
   Binary = binary()

Returns a list of integers corresponding to the bytes of Binary.

binary_to_list(Binary, Start, Stop) -> [byte()]

Types:
   Binary = binary()
   Start = Stop = integer() >= 1
   1..byte_size(Binary)

As binary_to_list/1, but returns a list of integers corresponding to the bytes from position Start to position Stop in Binary. The positions in the binary are numbered starting from 1.

Note:
The one-based indexing for binaries used by this function is deprecated. New code is to use binary:bin_to_list/3 in STDLIB instead. All functions in module binary consistently use zero-based indexing.

binary_to_term(Binary) -> term()

Types:
   Binary = ext_binary()

Returns an Erlang term that is the result of decoding binary object Binary, which must be encoded according to the Erlang external term format.
erlang

> Bin = term_to_binary(hello).
<<131,100,0,5,104,101,108,108,111>>
> hello = binary_to_term(Bin).
hello

**Warning:**

When decoding binaries from untrusted sources, the untrusted source may submit data in a way to create resources, such as atoms and remote references, that cannot be garbage collected and lead to Denial of Service attack. In such cases, consider using `binary_to_term/2` with the `safe` option.

See also `term_to_binary/1` and `binary_to_term/2`.

`binary_to_term(Binary, Opts) -> term() | {term(), Used}`

Types:
- `Binary` = `ext_binary()`
- `Opt` = `safe | used`
- `Opts` = `[Opt]`
- `Used` = `integer() >= 1`

As `binary_to_term/1`, but takes these options:

`safe`

Use this option when receiving binaries from an untrusted source.

When enabled, it prevents decoding data that can be used to attack the Erlang runtime. In the event of receiving unsafe data, decoding fails with a `badarg` error.

This prevents creation of new atoms directly, creation of new atoms indirectly (as they are embedded in certain structures, such as process identifiers, refs, and funs), and creation of new external function references. None of those resources are garbage collected, so unchecked creation of them can exhaust available memory.

> binary_to_term(<<131,100,0,5,"hello">>, [safe]).
** exception error: bad argument
> hello.
hello
> binary_to_term(<<131,100,0,5,"hello">>, [safe]).
hello

**Warning:**

The `safe` option ensures the data is safely processed by the Erlang runtime but it does not guarantee the data is safe to your application. You must always validate data from untrusted sources. If the binary is stored or transits through untrusted sources, you should also consider cryptographically signing it.

`used`

Changes the return value to `{Term, Used}` where `Used` is the number of bytes actually read from `Binary`. 
> Input = <<131,100,0,5,"hello","world">>.
> {Term, Used} = binary_to_term(Input, [used]).
> {hello, 9}
> split_binary(Input, Used).
> {<<131,100,0,5,104,101,108,108,111,114,108,100>>, <<"world">>}

Failure: badarg if safe is specified and unsafe data is decoded.

See also term_to_binary/1, binary_to_term/1, and list_to_existing_atom/1.

define module to do

bit_size(Bitstring) -> integer() >= 0

Types:

Bitstring = bitstring()

Returns an integer that is the size in bits of Bitstring, for example:

> bit_size(<<433:16,3:3>>).
19
> bit_size(<<1,2,3>>).
24

Allowed in guard tests.

bitstring_to_list(Bitstring) -> [byte() | bitstring()]

Types:

Bitstring = bitstring()

Returns a list of integers corresponding to the bytes of Bitstring. If the number of bits in the binary is not divisible by 8, the last element of the list is a bitstring containing the remaining 1-7 bits. Examples:

> bitstring_to_list(<<433:16>>).
[1,177]
> bitstring_to_list(<<433:16,3:3>>).
[1,177,<<3:3>>]

erlang:bump_reductions(Reductions) -> true

Types:

Reductions = integer() >= 1

This implementation-dependent function increments the reduction counter for the calling process. In the Beam emulator, the reduction counter is normally incremented by one for each function and BIF call. A context switch is forced when the counter reaches the maximum number of reductions for a process (4000 reductions in Erlang/OTP 19.2 and later).

Warning:

This BIF can be removed in a future version of the Beam machine without prior warning. It is unlikely to be implemented in other Erlang implementations.
byte_size(Bitstring) -> integer() \= 0

Types:
  Bitstring = bitstring()

Returns an integer that is the number of bytes needed to contain Bitstring. That is, if the number of bits in Bitstring is not divisible by 8, the resulting number of bytes is rounded up. Examples:

> byte_size(<<433:16,3:3>>).
  3
> byte_size(<<1,2,3>>).
  3

Allowed in guard tests.

erlang:cancel_timer(TimerRef) -> Result

Types:
  TimerRef = reference()
  Time = integer() \= 0
  Result = Time \| false

Cancels a timer. The same as calling erlang:cancel_timer(TimerRef, []).

erlang:cancel_timer(TimerRef, Options) -> Result \| ok

Types:
  TimerRef = reference()
  Async = Info = boolean()
  Option = {async, Async} \| {info, Info}
  Options = [Option]
  Time = integer() \= 0
  Result = Time \| false

Cancels a timer that has been created by erlang:start_timer or erlang:send_after. TimerRef identifies the timer, and was returned by the BIF that created the timer.

Options:

{async, Async}

Asynchronous request for cancellation. Async defaults to false, which causes the cancellation to be performed synchronously. When Async is set to true, the cancel operation is performed asynchronously. That is, cancel_timer() sends an asynchronous request for cancellation to the timer service that manages the timer, and then returns ok.

{info, Info}

Requests information about the Result of the cancellation. Info defaults to true, which means the Result is given. When Info is set to false, no information about the result of the cancellation is given.

- When Async is false: if Info is true, the Result is returned by erlang:cancel_timer(). otherwise ok is returned.
- When Async is true: if Info is true, a message on the form {cancel_timer, TimerRef, Result} is sent to the caller of erlang:cancel_timer() when the cancellation operation has been performed, otherwise no message is sent.

More Options may be added in the future.

172 | Ericsson AB. All Rights Reserved.: Erlang Run-Time System Application (ERTS)
If Result is an integer, it represents the time in milliseconds left until the canceled timer would have expired.

If Result is false, a timer corresponding to TimerRef could not be found. This can be either because the timer had expired, already had been canceled, or because TimerRef never corresponded to a timer. Even if the timer had expired, it does not tell you if the time-out message has arrived at its destination yet.

Note:
The timer service that manages the timer can be co-located with another scheduler than the scheduler that the calling process is executing on. If so, communication with the timer service takes much longer time than if it is located locally. If the calling process is in critical path, and can do other things while waiting for the result of this operation, or is not interested in the result of the operation, you want to use option \( \{ \text{async}, \text{true} \} \). If using option \( \{ \text{async}, \text{false} \} \), the calling process blocks until the operation has been performed.

See also erlang:send_after/4, erlang:start_timer/4, and erlang:read_timer/2.

\[\text{ceil(Number)} \rightarrow \text{integer()}\]

Types:
\[
\text{Number} = \text{number()}\]

Returns the smallest integer not less than Number. For example:

\[\text{> ceil(5.5).} \]
\[6\]

Allowed in guard tests.

\[\text{check_old_code(Module)} \rightarrow \text{boolean()}\]

Types:
\[
\text{Module} = \text{module()}\]

Returns true if Module has old code, otherwise false.

See also code(3).

\[\text{check_process_code(Pid, Module)} \rightarrow \text{CheckResult}\]

Types:
\[
\text{Pid} = \text{pid()}\]
\[
\text{Module} = \text{module()}\]
\[
\text{CheckResult} = \text{boolean()}\]

The same as check_process_code(Pid, Module, []).

\[\text{check_process_code(Pid, Module, OptionList)} \rightarrow \text{CheckResult} \mid \text{async}\]

Types:
Pid = pid()
Module = module()
RequestId = term()
Option = {async, RequestId} | {allow_gc, boolean()}
OptionList = [Option]
CheckResult = boolean() | aborted

Checks if the node local process identified by Pid executes old code for Module.

Options:
{allow_gc, boolean()}
Determined if garbage collection is allowed when performing the operation. If {allow_gc, false} is passed, and a garbage collection is needed to determine the result of the operation, the operation is aborted (see information on CheckResult below). The default is to allow garbage collection, that is, {allow_gc, true}.

{async, RequestId}
The function check_process_code/3 returns the value async immediately after the request has been sent. When the request has been processed, the process that called this function is passed a message on the form {check_process_code, RequestId, CheckResult}.

If Pid equals self(), and no async option has been passed, the operation is performed at once. Otherwise a request for the operation is sent to the process identified by Pid, and is handled when appropriate. If no async option has been passed, the caller blocks until CheckResult is available and can be returned.

CheckResult informs about the result of the request as follows:

true
The process identified by Pid executes old code for Module. That is, the current call of the process executes old code for this module, or the process has references to old code for this module, or the process contains funs that references old code for this module.

false
The process identified by Pid does not execute old code for Module.

aborted
The operation was aborted, as the process needed to be garbage collected to determine the operation result, and the operation was requested by passing option {allow_gc, false}.

Change:
Up until ERTS version 8.*, the check process code operation checks for all types of references to the old code. That is, direct references (e.g. return addresses on the process stack), indirect references (funs in process context), and references to literals in the code.

As of ERTS version 9.0, the check process code operation only checks for direct references to the code. Indirect references via funs will be ignored. If such funs exist and are used after a purge of the old code, an exception will be raised upon usage (same as the case when the fun is received by the process after the purge). Literals will be taken care of (copied) at a later stage. This behavior can as of ERTS version 8.1 be enabled when building OTP, and will automatically be enabled if dirty scheduler support is enabled.

See also code(3).

Failures:
badarg
  If Pid is not a node local process identifier.
badarg
  If Module is not an atom.
badarg
  If OptionList is an invalid list of options.

erlang:convert_time_unit(Time, FromUnit, ToUnit) -> ConvertedTime
Types:
  Time = ConvertedTime = integer()
  FromUnit = ToUnit = time_unit()

Converts the Time value of time unit FromUnit to the corresponding ConvertedTime value of time unit ToUnit. The result is rounded using the floor function.

Warning:
You can lose accuracy and precision when converting between time units. To minimize such loss, collect all data at native time unit and do the conversion on the end result.

erlang:crc32(Data) -> integer() >= 0
Types:
  Data = iodata()
Computes and returns the crc32 (IEEE 802.3 style) checksum for Data.

erlang:crc32(OldCrc, Data) -> integer() >= 0
Types:
  OldCrc = integer() >= 0
  Data = iodata()
Continues computing the crc32 checksum by combining the previous checksum, OldCrc, with the checksum of Data.
The following code:

\[
\begin{align*}
X &= \text{erlang:crc32(Data1)}, \\
Y &= \text{erlang:crc32}(X, \text{Data2}).
\end{align*}
\]

assigns the same value to Y as this:

\[
Y = \text{erlang:crc32}([\text{Data1}, \text{Data2}]).
\]

erlang:crc32_combine(FirstCrc, SecondCrc, SecondSize) ->
  integer() >= 0
Types:
  FirstCrc = SecondCrc = SecondSize = integer() >= 0
Combines two previously computed crc32 checksums. This computation requires the size of the data object for the second checksum to be known.
The following code:
Y = erlang:crc32(Data1),
Z = erlang:crc32(Y,Data2).

assigns the same value to Z as this:

X = erlang:crc32(Data1),
Y = erlang:crc32(Data2),
Z = erlang:crc32_combine(X,Y,iolist_size(Data2)).

date() -> Date
Types:
    Date = calendar:date()

Returns the current date as {Year, Month, Day}.
The time zone and Daylight Saving Time correction depend on the underlying OS. The return value is based on the OS System Time. Example:

> date().
{1995,2,19}

erlang:decode_packet(Type, Bin, Options) ->
    {ok, Packet, Rest} |
    {more, Length} |
    {error, Reason}

Types:
    Type = raw | 0 | 1 | 2 | 4 | asn1 | cdr | sunrm | fcgi | tpkt | line | http | http_bin | httph | httph_bin
    Bin = binary()
    Options = [Opt]
    Opt =
        {packet_size, integer() >= 0} |
        {line_length, integer() >= 0}
    Packet = binary() | HttpPacket
    Rest = binary()
    Length = integer() >= 0 | undefined
    Reason = term()
    HttpPacket =
        HttpRequest | HttpResponse | HttpHeaders | http_eoh | HttpError
    HttpRequest = {http_request, HttpMethod, HttpUri, HttpVersion}
    HttpHeaders =
        {http_response, HttpVersion, integer(), HttpString}
    HttpHeaders =
        {http_header, integer()},
        HttpField,
        UnmodifiedField :: HttpString,
Value :: HttpString}
HttpError = {http_error, HttpString}
HttpMethod =
  'OPTIONS' | 'GET' | 'HEAD' | 'POST' | 'PUT' | 'DELETE' |
  'TRACE' | HttpString
HttpUri =
  '*' |
  {absoluteURI, 
    http | https, 
    Host :: HttpString, 
    Port :: inet:port_number() | undefined, 
    Path :: HttpString} |
  {scheme, Scheme :: HttpString, HttpString} |
  {abs_path, HttpString} |
  HttpString
HttpVersion =
  {Major :: integer() >= 0, Minor :: integer() >= 0}
HttpField =
  'Cache-Control' | 'Connection' | 'Date' | 'Pragma' |
  'Transfer-Encoding' | 'Upgrade' | 'Via' | 'Accept' |
  'Accept-Charset' | 'Accept-Encoding' | 'Accept-Language' |
  'Authorization' | 'From' | 'Host' | 'If-Modified-Since' |
  'If-Match' | 'If-None-Match' | 'If-Range' |
  'If-Unmodified-Since' | 'Max-Forwards' |
  'Proxy-Authorization' | 'Range' | 'Referer' | 'User-Agent' |
  'Age' | 'Location' | 'Proxy-Authenticate' | 'Public' |
  'Retry-After' | 'Server' | 'Vary' | 'Warning' |
  'Www-Authenticate' | 'Allow' | 'Content-Base' |
  'Content-Encoding' | 'Content-Language' | 'Content-Length' |
  'Content-Location' | 'Content-Md5' | 'Content-Range' |
  'Content-Type' | 'Etag' | 'Expires' | 'Last-Modified' |
  'Accept-Ranges' | 'Set-Cookie' | 'Set-Cookie2' |
  'X-Forwarded-For' | 'Cookie' | 'Keep-Alive' |
  'Proxy-Connection' | HttpString

HttpString = string() | binary()

Decodes the binary Bin according to the packet protocol specified by Type. Similar to the packet handling done by sockets with option {packet,Type}.

If an entire packet is contained in Bin, it is returned together with the remainder of the binary as {ok,Packet,Rest}.

If Bin does not contain the entire packet, {more,Length} is returned. Length is either the expected total size of the packet, or undefined if the expected packet size is unknown. decode_packet can then be called again with more data added.

If the packet does not conform to the protocol format, {error,Reason} is returned.

Types:
  raw | 0

No packet handling is done. The entire binary is returned unless it is empty.
Packets consist of a header specifying the number of bytes in the packet, followed by that number of bytes. The length of the header can be one, two, or four bytes; the order of the bytes is big-endian. The header is stripped off when the packet is returned.

**line**

A packet is a line-terminated by a delimiter byte, default is the latin-1 newline character. The delimiter byte is included in the returned packet unless the line was truncated according to option line_length.

<table>
<thead>
<tr>
<th>asn1</th>
<th>cdr</th>
<th>sunrm</th>
<th>fcgi</th>
<th>tpkt</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

The meanings of the packet types are as follows:

- **asn1** - ASN.1 BER
- **sunrm** - Sun's RPC encoding
- **cdr** - CORBA (GIOP 1.1)
- **fcgi** - Fast CGI
- **tpkt** - TPKT format [RFC1006]

<table>
<thead>
<tr>
<th>http</th>
<th>http</th>
<th>http_bin</th>
<th>http_bin</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

The Hypertext Transfer Protocol. The packets are returned with the format according to HttpPacket described earlier. A packet is either a request, a response, a header, or an end of header mark. Invalid lines are returned as HttpError.

Recognized request methods and header fields are returned as atoms. Others are returned as strings. Strings of unrecognized header fields are formatted with only capital letters first and after hyphen characters, for example, "Sec-Websocket-Key". Header field names are also returned in UnmodifiedField as strings, without any conversion or formatting.

The protocol type http is only to be used for the first line when an HttpRequest or an HttpResponse is expected. The following calls are to use http to get HttpHeaders until http_eoh is returned, which marks the end of the headers and the beginning of any following message body.

The variants http_bin and httph_bin return strings (HttpString) as binaries instead of lists.

Since OTP 26.0, Host may be an IPv6 address enclosed in [], as defined in RFC2732.

Options:

- **{packet_size, integer() >= 0}**
  
  Sets the maximum allowed size of the packet body. If the packet header indicates that the length of the packet is longer than the maximum allowed length, the packet is considered invalid. Defaults to 0, which means no size limit.

- **{line_length, integer() >= 0}**
  
  For packet type line, lines longer than the indicated length are truncated.

  Option line_length also applies to http* packet types as an alias for option packet_size if packet_size itself is not set. This use is only intended for backward compatibility.

- **{line_delimiter, 0 =< byte() =< 255}**

  For packet type line, sets the delimiting byte. Default is the latin-1 character $\n$.
erlang:decode_packet(1,<<3,"abcd">>,[]).
{ok,<"abc">>,<<"d">>}
> erlang:decode_packet(1,<<5,"abcd">>,[]).
{more,6}

erlang:delete_element(Index, Tuple1) -> Tuple2

Types:
  Index = integer() >= 1
  1..tuple_size(Tuple1)
  Tuple1 = Tuple2 = tuple()

Returns a new tuple with element at Index removed from tuple Tuple1, for example:

> erlang:delete_element(2, {one, two, three}).
{one,three}

delete_module(Module) -> true | undefined

Types:
  Module = module()

Makes the current code for Module become old code and deletes all references for this module from the export table. Returns undefined if the module does not exist, otherwise true.

Warning:

This BIF is intended for the code server (see code(3)) and is not to be used elsewhere.

Failure: badarg if there already is an old version of Module.

demonitor(MonitorRef) -> true

Types:
  MonitorRef = reference()

If MonitorRef is a reference that the calling process obtained by calling monitor/2, this monitoring is turned off. If the monitoring is already turned off, nothing happens.

Once demonitor(MonitorRef) has returned, it is guaranteed that no ('DOWN', MonitorRef, _, _, _) message, because of the monitor, will be placed in the caller message queue in the future. However, a ('DOWN', MonitorRef, _, _, _) message can have been placed in the caller message queue before the call. It is therefore usually advisable to remove such a 'DOWN' message from the message queue after monitoring has been stopped. demonitor(MonitorRef, [flush]) can be used instead of demonitor(MonitorRef) if this cleanup is wanted.

Note:

For some important information about distributed signals, see the Blocking Signaling Over Distribution section in the Processes chapter of the Erlang Reference Manual.
Change:

Before Erlang/OTP R11B (ERTS 5.5) `demonitor/1` behaved completely asynchronously, that is, the monitor was active until the "demonitor signal" reached the monitored entity. This had one undesirable effect. You could never know when you were guaranteed not to receive a \texttt{DOWN} message because of the monitor.

The current behavior can be viewed as two combined operations: asynchronously send a "demonitor signal" to the monitored entity and ignore any future results of the monitor.

Failure: It is an error if \texttt{MonitorRef} refers to a monitoring started by another process. Not all such cases are cheap to check. If checking is cheap, the call fails with \texttt{badarg}, for example if \texttt{MonitorRef} is a remote reference.

\begin{verbatim}
demonitor(MonitorRef, OptionList) -> boolean()
Types:
  MonitorRef = reference()
  OptionList = [Option]
  Option = flush | info

The returned value is \texttt{true} unless \texttt{info} is part of \texttt{OptionList}.

demonitor (MonitorRef, []) is equivalent to \texttt{demonitor (MonitorRef)}.

Options:

flush
  Removes (one) \{_, MonitorRef, _, _, _\} message, if there is one, from the caller message queue after monitoring has been stopped.

Calling \texttt{demonitor (MonitorRef, [flush])} is equivalent to the following, but more efficient:

\begin{verbatim}
demonitor(MonitorRef),
receive
  \{_, MonitorRef, _, _, _\} ->
  true
after 0 ->
  true
end
\end{verbatim}

info
  The returned value is one of the following:

\texttt{true}
  The monitor was found and removed. In this case, no \texttt{DOWN} message corresponding to this monitor has been delivered and will not be delivered.

\texttt{false}
  The monitor was not found and could not be removed. This probably because someone already has placed a \texttt{DOWN} message corresponding to this monitor in the caller message queue.

If option \texttt{info} is combined with option \texttt{flush}, \texttt{false} is returned if a flush was needed, otherwise \texttt{true}.

Change:

More options can be added in a future release.

Failures:
badarg
    If OptionList is not a list.
badarg
    If Option is an invalid option.
badarg
    The same failure as for demonitor/1.

disconnect_node(Node) -> boolean() | ignored
Types:
    Node = node()
Forces the disconnection of a node. This appears to the node Node as if the local node has crashed. This BIF is mainly used in the Erlang network authentication protocols.
Returns true if disconnection succeeds, otherwise false. If the local node is not alive, ignored is returned.

Note:
This function may return before nodedown messages have been delivered.

erlang:display(Term) -> true
Types:
    Term = term()
Prints a text representation of Term on the standard output.

Warning:
This BIF is intended for debugging only. The printed representation may contain internal details that do not match the high-level representation of the term in Erlang.

erlang:dist_ctrl_get_data(DHandle) -> {Size, Data} | Data | none
Types:
    Size = integer() >= 0
    DHandle = dist_handle()
    Data = iovec()
Get distribution channel data from the local node that is to be passed to the remote node. The distribution channel is identified by DHandle. If no data is available, the atom none is returned. One can request to be informed by a message when more data is available by calling erlang:dist_ctrl_get_data_notification(DHandle).
The returned value when there are data available depends on the value of the get_size option configured on the distribution channel identified by DHandle. For more information see the documentation of the get_size option for the erlang:dist_ctrl_set_opt/3 function.

Note:
Only the process registered as distribution controller for the distribution channel identified by DHandle is allowed to call this function.
This function is used when implementing an alternative distribution carrier using processes as distribution controllers. DHandle is retrieved via the callback f_handshake_complete. More information can be found in the documentation of ERTS User's Guide # How to implement an Alternative Carrier for the Erlang Distribution # Distribution Module.

**erlang:dist_ctrl_get_opt(DHandle, Opt :: get_size) -> Value**

Types:

- **DHandle** = dist_handle()
- **Value** = boolean()

Returns the value of the get_size option on the distribution channel identified by DHandle. For more information see the documentation of the get_size option for the erlang:dist_ctrl_set_opt/3 function.

**Note:**

Only the process registered as distribution controller for the distribution channel identified by DHandle is allowed to call this function.

This function is used when implementing an alternative distribution carrier using processes as distribution controllers. DHandle is retrieved via the callback f_handshake_complete. More information can be found in the documentation of ERTS User's Guide # How to implement an Alternative Carrier for the Erlang Distribution # Distribution Module.

**erlang:dist_ctrl_get_data_notification(DHandle) -> ok**

Types:

- **DHandle** = dist_handle()

Request notification when more data is available to fetch using erlang:dist_ctrl_get_data(DHandle) for the distribution channel identified by DHandle. When more data is present, the caller will be sent the message dist_data. Once a dist_data messages has been sent, no more dist_data messages will be sent until the dist_ctrl_get_data_notification/1 function has been called again.

**Note:**

Only the process registered as distribution controller for the distribution channel identified by DHandle is allowed to call this function.

This function is used when implementing an alternative distribution carrier using processes as distribution controllers. DHandle is retrieved via the callback f_handshake_complete. More information can be found in the documentation of ERTS User's Guide # How to implement an Alternative Carrier for the Erlang Distribution # Distribution Module.

**erlang:dist_ctrl_input_handler(DHandle, InputHandler) -> ok**

Types:

- **DHandle** = dist_handle()
- **InputHandler** = pid()

Register an alternate input handler process for the distribution channel identified by DHandle. Once this function has been called, InputHandler is the only process allowed to call erlang:dist_ctrl_put_data(DHandle, Data) with the DHandle identifying this distribution channel.
**Note:**
Only the process registered as distribution controller for the distribution channel identified by \( D\text{Handle} \) is allowed to call this function.

This function is used when implementing an alternative distribution carrier using processes as distribution controllers. \( D\text{Handle} \) is retrieved via the callback `f_handshake_complete`. More information can be found in the documentation of ERTS User's Guide # How to implement an Alternative Carrier for the Erlang Distribution # Distribution Module.

```
erlang:dist_ctrl_put_data(DHandle, Data) -> ok
```

Types:
- \( D\text{Handle} \) = dist_handle()
- \( Data \) = iodata()

Deliver distribution channel data from a remote node to the local node.

**Note:**
Only the process registered as distribution controller for the distribution channel identified by \( D\text{Handle} \) is allowed to call this function unless an alternate input handler process has been registered using `erlang:dist_ctrl_input_handler(DHandle, InputHandler)`. If an alternate input handler has been registered, only the registered input handler process is allowed to call this function.

This function is used when implementing an alternative distribution carrier using processes as distribution controllers. \( D\text{Handle} \) is retrieved via the callback `f_handshake_complete`. More information can be found in the documentation of ERTS User's Guide # How to implement an Alternative Carrier for the Erlang Distribution # Distribution Module.

```
erlang:dist_ctrl_set_opt(DHandle, Opt :: get_size, Value) -> OldValue
```

Types:
- \( D\text{Handle} \) = dist_handle()
- \( Value \) = OldValue = boolean()

Sets the value of the `get_size` option on the distribution channel identified by \( D\text{Handle} \). This option controls the return value of calls to `erlang:dist_ctrl_get_data(DHandle)` where \( D\text{Handle} \) equals \( D\text{Handle} \) used when setting this option. When the `get_size` option is:

- `false`
  - and there are distribution data available, a call to `erlang:dist_ctrl_get_data(DHandle)` will just return \( Data \) to pass over the channel. This is the default value of the `get_size` option.
- `true`
  - and there are distribution data available, a call to `erlang:dist_ctrl_get_data(DHandle)` will return \( Data \) to pass over the channel as well as the `Size` of \( Data \) in bytes. This is returned as a tuple on the form \( \{Size, Data\} \).

All options are set to default when a channel is closed.
**Note:**

Only the process registered as distribution controller for the distribution channel identified by `DHandle` is allowed to call this function.

This function is used when implementing an alternative distribution carrier using processes as distribution controllers. `DHandle` is retrieved via the callback `f_handshake_complete`. More information can be found in the documentation of ERTS User's Guide # How to implement an Alternative Carrier for the Erlang Distribution # Distribution Module.

```
element(N, Tuple) -> term()
Types:
  N = integer() >= 1
  1..tuple_size(Tuple)
  Tuple = tuple()
Returns the \textit{N}th element (numbering from 1) of \textit{Tuple}, for example:

\begin{verbatim}
> element(2, \{a, b, c\}).
\end{verbatim}

Allowed in guard tests.
```

```
erase() -> [{Key, Val}]
Types:
  Key = Val = term()
Returns the process dictionary and deletes it, for example:

\begin{verbatim}
> put(key1, \{1, 2, 3\}),
put(key2, \[a, b, c\]),
erase().
\{\{key1,\{1,2,3\}\},\{key2,\[a,b,c\]\}\}
\end{verbatim}
```

```
erase(Key) -> Val | undefined
Types:
  Key = Val = term()
Returns the value \textit{Val} associated with \textit{Key} and deletes it from the process dictionary. Returns \texttt{undefined} if no value is associated with \textit{Key}. The average time complexity for the current implementation of this function is \(O(1)\) and the worst case time complexity is \(O(N)\), where \(N\) is the number of items in the process dictionary. Example:

\begin{verbatim}
> put(key1, \{merry, lambs, are, playing\}),
X = erase(key1),
\{X, erase(key1)\},
\{\{merry,lambs,are,playing\},\texttt{undefined}\}
\end{verbatim}
```

```
error(Reason) -> no_return()
Types:
```

---

184 | Ericsson AB. All Rights Reserved.: Erlang Run-Time System Application (ERTS)
Reason = term()

Raises an exception of class error with the reason Reason. As evaluating this function causes an exception to be thrown, it has no return value.

The intent of the exception class error is to signal that an unexpected error has happened (for example, a function is called with a parameter that has an incorrect type). See the guide about errors and error handling for additional information. Example:

```erlang
> catch error(foobar).
{'EXIT',{foobar,[[shell,apply_fun,3,
  [{file,"shell.erl"},{line,906}],
  {erl_eval,do_apply,6,[[file,"erl_eval.erl"],[line,677]]},
  {erl_eval,expr,5,[[file,"erl_eval.erl"],[line,430]]},
  {shell,exprs,7,[[file,"shell.erl"],[line,687]]},
  {shell,eval_exprs,7,[[file,"shell.erl"],[line,642]]},
  {shell,eval_loop,3,[[file,"shell.erl"],[line,627]]]]}]}{ok,0}
```

error(Reason, Args) -> no_return()  

Types:

Reason = term()  

Args = [term()] | none

Raises an exception of class error with the reason Reason. Args is expected to be the list of arguments for the current function or the atom none. If it is a list, it is used to provide the arguments for the current function in the stack back-trace. If it is none, the arity of the calling function is used in the stacktrace. As evaluating this function causes an exception to be raised, it has no return value.

The intent of the exception class error is to signal that an unexpected error has happened (for example, a function is called with a parameter that has an incorrect type). See the guide about errors and error handling for additional information. Example:

test.erl:

```erlang
-module(test).
-export([example_fun/2]).

extample_fun(A1, A2) ->
  erlang:error(my_error, [A1, A2]).
```

Erlang shell:

```erlang
6> c(test).
{ok,test}
7> test:example_fun(arg1,"this is the second argument").
** exception error: my_error
  in function  test:example_fun/2
called as test:example_fun(arg1,"this is the second argument")
```

error(Reason, Args, Options) -> no_return()  

Types:
Reason = term()
Args = [term()] | none
Options = [Option]
Option = {error_info, ErrorInfoMap}
ErrorInfoMap =
    #{cause => term(), module => module(), function => atom()}

Raises an exception of class error with the reason Reason. Args is expected to be the list of arguments for the current function or the atom none. If it is a list, it is used to provide the arguments for the current function in the stack back-trace. If it is none, the arity of the calling function is used in the stacktrace. As evaluating this function causes an exception to be raised, it has no return value.

If the error_info option is given, the ErrorInfoMap will be inserted into the stacktrace. The information given in the ErrorInfoMap is to be used by error formatters such as erl_error to provide more context around an error.

The default module of the ErrorInfoMap is the module that the call to error/3 is made. The default function is format_error. See format_error/2 for more details on how this Module:Function/2 is to be used.

The intent of the exception class error is to signal that an unexpected error has happened (for example, a function is called with a parameter that has an incorrect type). See the guide about errors and error handling for additional information.

exit(Reason) -> no_return()
Types:
  Reason = term()

Raises an exception of class exit with exit reason Reason. As evaluating this function causes an exception to be raised, it has no return value.

The intent of the exception class exit is that the current process should be stopped (for example when a message telling a process to stop is received).

This function differ from error/1,2,3 by causing an exception of a different class and by having a reason that does not include the list of functions from the call stack.

See the guide about error/1,2,3 by causing an exception of a different class and by having a reason that does not include the list of functions from the call stack.

Note:
If a process calls exit (kill) and does not catch the exception, it will terminate with exit reason kill and also emit exit signals with exit reason kill (not killed) to all linked processes. Such exit signals with exit reason kill can be trapped by the linked processes. Note that this means that signals with exit reason kill behave differently depending on how they are sent because the signal will be untrappable if a process sends such a signal to another process with erlang:exit/2.

exit(Pid, Reason) -> true
Types:
Pid = pid() | port()
Reason = term()

Sends an exit signal with exit reason Reason to the process or port identified by Pid.

The following behavior applies if Reason is any term, except normal or kill, and P is the process or port identified by Pid:

- If P is not trapping exits, P exits with exit reason Reason.
- If P is trapping exits, the exit signal is transformed into a message {'EXIT', From, Reason}, where From is the process identifier of the process that sent the exit signal, and delivered to the message queue of P.

The following behavior applies if Reason is the term normal and P is the identifier of a process P which is not the same as the process that invoked erlang:exit(Pid, normal) (the behavior when a process sends a signal with the normal reason to itself is described in the warning):

- If P is trapping exits, the exit signal is transformed into a message {'EXIT', From, normal}, where From is the process identifier of the process that sent the exit signal, and delivered to P’s message queue.
- The signal has no effect if P is not trapping exits.

If Reason is the atom kill, that is, if exit(Pid, kill) is called, an untrappable exit signal is sent to the process that is identified by Pid, which unconditionally exits with exit reason killed. The exit reason is changed from kill to killed to hint to linked processes that the killed process got killed by a call to exit(Pid, kill).

**Note:**

The functions erlang:exit/1 and erlang:exit/2 are named similarly but provide very different functionalities. The erlang:exit/1 function should be used when the intent is to stop the current process while erlang:exit/2 should be used when the intent is to send an exit signal to another process. Note also that erlang:exit/1 raises an exception that can be caught while erlang:exit/2 does not cause any exception to be raised.

**Warning:**

The only scenario that has not been covered by the description above is when a process P sends an exit signal with reason normal to itself, that is erlang:exit(self(), normal). The behavior in this scenario is as follows:

- If P is trapping exits, the exit signal is transformed into a message {'EXIT', From, normal}, where From is P's process identifier, and delivered to P’s message queue.
- P exits with reason normal if P is not trapping exits.

Note that the behavior described above is different from when a process sends an exit signal with reason normal to another process. This is arguably strange but this behavior is kept for backward compatibility reasons.

**Note:**

For some important information about distributed signals, see the Blocking Signaling Over Distribution section in the Processes chapter of the Erlang Reference Manual.

erlang:external_size(Term) -> integer() >= 0

Types:
Term = term()
Calculates, without doing the encoding, the maximum byte size for a term encoded in the Erlang external term format. The following condition applies always:

```
> Size1 = byte_size(term_to_binary()),
> Size2 = erlang:external_size(),
> true = Size1 =< Size2.
true
```
This is equivalent to a call to:

```
erlang:external_size(Term, [])
```

erlang:external_size(Term, Options) -> integer() >= 0
Types:

```
Term = term()
```
```
Options =
    [compressed | {compressed, Level :: 0..9} | deterministic | {minor_version, Version :: 0..2} | local]
```
Calculates, without doing the encoding, the maximum byte size for a term encoded in the Erlang external term format. The following condition applies always:

```
> Size1 = byte_size(term_to_binary(), ),
> Size2 = erlang:external_size(, ),
> true = Size1 =< Size2.
true
```
Option {minor_version, Version} specifies how floats are encoded. For a detailed description, see term_to_binary/2.

float(Number) -> float()
Types:

```
Number = number()
```
Returns a float by converting Number to a float, for example:

```
> float(55).
55.0
```
Allowed in guard tests.
Note:
If used on the top level in a guard, it tests whether the argument is a floating point number; for clarity, use `is_float/1` instead.
When `float/1` is used in an expression in a guard, such as `float(A) == 4.0`, it converts a number as described earlier.

float_to_binary(Float) -> binary()
Types:
    Float = float()
The same as `float_to_binary(Float, [{scientific, 20}]).`

float_to_binary(Float, Options) -> binary()
Types:
    Float = float()
    Options = [Option]
    Option =
        {decimals, Decimals :: 0..253} |
        {scientific, Decimals :: 0..249} |
        compact | short

Returns a binary corresponding to the text representation of `Float` using fixed decimal point formatting. Options behaves in the same way as `float_to_list/2`. Examples:

```
> float_to_binary(7.12, [{decimals, 4}]).
<<"7.1200">>
> float_to_binary(7.12, [{decimals, 4}, compact]).
<<"7.12">>
> float_to_binary(7.12, [{scientific, 3}]).
<<"7.120e+00">>
> float_to_binary(7.12, [short]).
<<"7.12">>
> float_to_binary(0.1+0.2, [short]).
<<"0.30000000000000004",0.1+0.2>>
> float_to_binary(0.1+0.2).
<<"3.680000000000000044409e-01">>
```

float_to_list(Float) -> string()
Types:
    Float = float()
The same as `float_to_list(Float, [{scientific, 20}]).`

float_to_list(Float, Options) -> string()
Types:
    Float = float()
    Options = [Option]
    Option =
        {decimals, Decimals :: 0..253} |
Returns a string corresponding to the text representation of \texttt{Float} using fixed decimal point formatting.

Available options:

- If option \texttt{decimals} is specified, the returned value contains at most \texttt{Decimals} number of digits past the decimal point. If the number does not fit in the internal static buffer of 256 bytes, the function throws \texttt{badarg}.
- If option \texttt{compact} is specified, the trailing zeros at the end of the list are truncated. This option is only meaningful together with option \texttt{decimals}.
- If option \texttt{scientific} is specified, the float is formatted using scientific notation with \texttt{Decimals} digits of precision.
- If option \texttt{short} is specified, the float is formatted with the smallest number of digits that still guarantees that \( F = \text{list_to_float(float_to_list(F, [short]))} \). When the float is inside the range \((-2^3, 2^3)\), the notation that yields the smallest number of characters is used (scientific notation or normal decimal notation). Floats outside the range \((-2^3, 2^3)\) are always formatted using scientific notation to avoid confusing results when doing arithmetic operations.
- If \texttt{Options} is [], the function behaves as \texttt{float_to_list/1}.

Examples:

\begin{verbatim}
> float_to_list(7.12, [{decimals, 4}]).
"7.1200"
> float_to_list(7.12, [{decimals, 4}, compact]).
"7.12"
> float_to_list(7.12, [{scientific, 3}]).
"7.128e+00"
> float_to_list(7.12, [short]).
"7.12"
> float_to_list(0.1+0.2, [short]).
"0.3000000000000004"  
> float_to_list(0.1+0.2)
"3.00000000000000044409e-01"
\end{verbatim}

In the last example, \texttt{float_to_list(0.1+0.2)} evaluates to "3.00000000000000044409e-01". The reason for this is explained in Representation of Floating Point Numbers.

\texttt{floor(Number) -> integer()}

Types:

- \texttt{Number = number()}

Returns the largest integer not greater than \texttt{Number}. For example:

\begin{verbatim}
> floor(-10.5).
-11
\end{verbatim}

Allowed in guard tests.
 pid | type | uniq
Info = term()

Returns a list with information about the fun Fun. Each list element is a tuple. The order of the tuples is undefined, and more tuples can be added in a future release.

Warning:

This BIF is mainly intended for debugging, but it can sometimes be useful in library functions that need to verify, for example, the arity of a fun.

Two types of funs have slightly different semantics:

• A fun created by fun M:F/A is called an external fun. Calling it will always call the function F with arity A in the latest code for module M. Notice that module M does not even need to be loaded when the fun fun M:F/A is created.

• All other funs are called local. When a local fun is called, the same version of the code that created the fun is called (even if a newer version of the module has been loaded).

The following elements are always present in the list for both local and external funs:

{type, Type}
Type is local or external.

{module, Module}
Module (an atom) is the module name.

If Fun is a local fun, Module is the module in which the fun is defined.

If Fun is an external fun, Module is the module that the fun refers to.

{name, Name}
Name (an atom) is a function name.

If Fun is a local fun, Name is the name of the local function that implements the fun. (This name was generated by the compiler, and is only of informational use. As it is a local function, it cannot be called directly.) If no code is currently loaded for the fun, [] is returned instead of an atom.

If Fun is an external fun, Name is the name of the exported function that the fun refers to.

{arity, Arity}
Arity is the number of arguments that the fun is to be called with.

{env, Env}
Env (a list) is the environment or free variables for the fun. For external funs, the returned list is always empty.

The following elements are only present in the list if Fun is local:

{pid, Pid}
Pid is the process identifier of the process that originally created the fun.

It might point to the init process if the Fun was statically allocated when module was loaded (this optimisation is performed for local functions that do not capture the environment).
Change:

In Erlang/OTP 27, we plan to change the return value so that it always points to the local init process, regardless of which process or node the fun was originally created on. See Upcoming Potential Incompatibilities.

{index, Index}
Index (an integer) is an index into the module fun table.

{new_index, Index}
Index (an integer) is an index into the module fun table.

{new_uniq, Uniq}
Uniq (a binary) is a unique value for this fun. It is calculated from the compiled code for the entire module.

{uniq, Uniq}
Uniq (an integer) is a unique value for this fun. As from Erlang/OTP R15, this integer is calculated from the compiled code for the entire module. Before Erlang/OTP R15, this integer was based on only the body of the fun.

erlang:fun_info(Fun, Item) -> {Item, Info}
Types:
Fun = function()
Item = fun_info_item()
Info = term()
fun_info_item() =
arity | env | index | name | module | new_index | new_uniq |
pid | type | uniq
Returns information about Fun as specified by Item, in the form {Item, Info}.
For any fun, Item can be any of the atoms module, name, arity, env, or type.
For a local fun, Item can also be any of the atoms index, new_index, new_uniq, uniq, and pid. For an external fun, the value of any of these items is always the atom undefined.
See erlang:fun_info/1.

erlang:fun_to_list(Fun) -> String :: string()
Types:
Fun = function()
Returns String that represents the code that created Fun.
String has the following form, if Fun was created by a fun expression of the form fun ModuleName:FuncName/Arity:
"fun ModuleName:FuncName/Arity"
The form of String when Fun is created from other types of fun expressions differs depending on if the fun expression was executed while executing compiled code or if the fun expression was executed while executing uncompiled code (uncompiled escripts, the Erlang shell, and other code executed by the erl_eval module):
compiled code

"#Fun<M.I.U>" , where M, I and U correspond to the values named module, index and uniq in the result of erlang:fun_info(Fun).

uncompiled code

All funs created from fun expressions in uncompiled code with the same arity are mapped to the same list by fun_to_list/1.

Note:

Generally, one can not use fun_to_list/1 to check if two funs are equal as fun_to_list/1 does not take the fun's environment into account. See erlang:fun_info/1 for how to get the environment of a fun.

Change:

The output of fun_to_list/1 can differ between Erlang implementations and may change in future versions.

Examples:

-module(test).
-export([add/1, add2/0, fun_tuple/0]).
add(A) -> fun(B) -> A + B end.
add2() -> fun add/1.
fun_tuple() -> {fun() -> 1 end, fun() -> 1 end}.

> {fun test:add/1, test:add2()}.  
{fun test:add/1,#Fun<test.1.107738983>}

Explanation: fun test:add/1 is upgradable but test:add2() is not upgradable.

> {test:add(1), test:add(42)}.  
(#Fun<test.0.107738983>,#Fun<test.0.107738983>)}

Explanation: test:add(1) and test:add(42) has the same string representation as the environment is not taken into account.

> test:fun_tuple().  
(#Fun<test.2.107738983>,#Fun<test.3.107738983>)}

Explanation: The string representations differ because the funs come from different fun expressions.

> {fun() -> 1 end, fun() -> 1 end}. > 
(#Fun<erl_eval.45.97283095>,#Fun<erl_eval.45.97283095>)

Explanation: All funs created from fun expressions of this form in uncompiled code with the same arity are mapped to the same list by fun_to_list/1.

erlang:function_exported(Module, Function, Arity) -> boolean()  
Types:
Module = module()
Function = atom()
Arity = arity()

Returns true if the module Module is current and contains an exported function Function/Arity, or if there is a BIF (a built-in function implemented in C) with the specified name, otherwise returns false.

garbage_collect() -> true

Forces an immediate garbage collection of the executing process. The function is not to be used unless it has been noticed (or there are good reasons to suspect) that the spontaneous garbage collection will occur too late or not at all.

Warning:

Improper use can seriously degrade system performance.

garbage_collect(Pid) -> GResult

Types:

Pid = pid()
GResult = boolean()

The same as garbage_collect(Pid, []).

garbage_collect(Pid, OptionList) -> GResult | async

Types:

Pid = pid()
RequestId = term()
Option = {async, RequestId} | {type, major | minor}
OptionList = [Option]
GResult = boolean()

Garbage collects the node local process identified by Pid.

Option:

{async, RequestId}

The function garbage_collect/2 returns the value async immediately after the request has been sent. When the request has been processed, the process that called this function is passed a message on the form {garbage_collect, RequestId, GResult}.

{type, 'major' | 'minor'}

Triggers garbage collection of requested type. Default value is 'major', which would trigger a fullsweep GC. The option 'minor' is considered a hint and may lead to either minor or major GC run.

If Pid equals self(), and no async option has been passed, the garbage collection is performed at once, that is, the same as calling garbage_collect/0. Otherwise a request for garbage collection is sent to the process identified by Pid, and will be handled when appropriate. If no async option has been passed, the caller blocks until GResult is available and can be returned.

GResult informs about the result of the garbage collection request as follows:

ture

The process identified by Pid has been garbage collected.
false

No garbage collection was performed, as the process identified by Pid terminated before the request could be satisfied.

Notice that the same caveats apply as for garbage_collect/0.

Failures:

badarg
  If Pid is not a node local process identifier.

badarg
  If OptionList is an invalid list of options.

get() -> [{Key, Val}]
Types:
  Key = Val = term()

Returns the process dictionary as a list of {Key, Val} tuples. The items in the returned list can be in any order. Example:

> put(key1, merry),
  put(key2, lambs),
  put(key3, {are, playing}),
  get().
  [{key1,merry},{key2,lambs},{key3,{are,playing}}]

get(Key) -> Val | undefined
Types:
  Key = Val = term()

Returns the value Val associated with Key in the process dictionary, or undefined if Key does not exist. The expected time complexity for the current implementation of this function is O(1) and the worst case time complexity is O(N), where N is the number of items in the process dictionary. Example:

> put(key1, merry),
  put(key2, lambs),
  put({any, [valid, term]}, {are, playing}),
  get{{any, [valid, term]}}.
  {are,playing}

erlang:get_cookie() -> Cookie | nocookie
Types:
  Cookie = atom()

Returns the magic cookie of the local node if the node is alive, otherwise the atom nocookie. This value is set by set_cookie/1.

erlang:get_cookie(Node) -> Cookie | nocookie
Types:
Node = node()
Cookie = atom()

Returns the magic cookie for node Node if the local node is alive, otherwise the atom nocookie. This value is set by set_cookie/2.

get_keys() -> [Key]
Types:
    Key = term()

Returns a list of all keys present in the process dictionary. The items in the returned list can be in any order. Example:

```
> put(dog, {animal,1}),
put(cow, {animal,2}),
put(lamb, {animal,3}),
get_keys().
[dog,cow,lamb]
```

get_keys(Val) -> [Key]
Types:
    Val = Key = term()

Returns a list of keys that are associated with the value Val in the process dictionary. The items in the returned list can be in any order. Example:

```
> put(mary, {1, 2}),
put(had, {1, 2}),
put(a, {1, 2}),
put(little, {1, 2}),
put(dog, {1, 3}),
put(lamb, {1, 2}),
get_keys([{1, 2}]).
[mary,had,a,little,lamb]
```

group_leader() -> pid()

Returns the process identifier of the group leader for the process evaluating the function.

Every process is a member of some process group and all groups have a group leader. All I/O from the group is channeled to the group leader. When a new process is spawned, it gets the same group leader as the spawning process.

Initially, at system startup, init is both its own group leader and the group leader of all processes. During the boot of a system the group leader for processes will be changed depending on the need of the system. Some examples where this is done are:

- When an application is started, the top supervisor of that application will have its group leader set to the application master. See application:start/2 for more details.
- When running tests, both common_test and eunit set the group leader in order to capture any I/O from the testcase.
- The interactive shell sets the group leader to intercept I/O.

group_leader(GroupLeader, Pid) -> true
Types:
GroupLeader = Pid = pid()

Sets the group leader of Pid to GroupLeader. Typically, this is used when a process started from a certain shell is to have another group leader than init.

The group leader should be rarely changed in applications with a supervision tree, because OTP assumes the group leader of their processes is their application master.

Setting the group leader follows the signal ordering guarantees described in the Processes Chapter in the Erlang Reference Manual.

See also group_leader/0 and OTP design principles related to starting and stopping applications.

Note:

For some important information about distributed signals, see the Blocking Signaling Over Distribution section in the Processes chapter of the Erlang Reference Manual.

halt() -> no_return()
The same as calling halt(0, []). Example:

```erlang
> halt().
os_prompt%
```

halt(Status :: integer() >= 0) -> no_return()
The same as calling halt(Status, []). Example:

```erlang
> halt(17).
os_prompt% echo $?
17
os_prompt%
```

halt(Abort :: abort) -> no_return()
The same as calling halt(abort, []).

halt(CrashDumpSlogan :: string()) -> no_return()
The same as calling halt(CrashDumpSlogan, []).

halt(Status :: integer() >= 0, Options :: halt_options()) ->
  no_return()

Types:
  halt_options() = [{flush, boolean()}]

Halt the runtime system with status code Status.

Note:

On many platforms, the OS supports only status codes 0-255. A too large status code is truncated by clearing the high bits.
Currently the following options are valid:

{flush, EnableFlushing}

If EnableFlushing equals true, which also is the default behavior, the runtime system will perform the following operations before terminating:

- Flush all outstanding output.
- Send all Erlang ports exit signals and wait for them to exit.
- Wait for all async threads to complete all outstanding async jobs.
- Call all installed NIF on halt callbacks.
- Wait for all ongoing NIF calls with the delay halt setting enabled to return.
- Call all installed atexit/on_exit callbacks.

If EnableFlushing equals false, the runtime system will terminate immediately without performing any of the above listed operations.

### Change:

Runtime systems prior to OTP 26.0 called all installed atexit/on_exit callbacks also when flush was disabled, but as of OTP 26.0 this is no longer the case.

#### halt(Abort :: abort, Options :: halt_options()) -> no_return()

Types:

halt_options() = [{flush, boolean()}]

Halt the Erlang runtime system by aborting and produce a core dump if core dumping has been enabled in the environment that the runtime system is executing in.

### Note:

The {flush, boolean()} option will be ignored, and flushing will be disabled.

#### halt(CrashDumpSlogan :: string(), Options :: halt_options()) -> no_return()

Types:

halt_options() = [{flush, boolean()}]

Halt the Erlang runtime system and generate an Erlang crash dump. The string CrashDumpSlogan will be used as slogan in the Erlang crash dump created. The slogan will be truncated if CrashDumpSlogan is longer than 1023 characters.

### Note:

The {flush, boolean()} option will be ignored, and flushing will be disabled.

Behavior changes compared to earlier versions:

- Before OTP 24.2, the slogan was truncated if CrashDumpSlogan was longer than 200 characters. Now it will be truncated if longer than 1023 characters.
- Before OTP 20.1, only code points in the range 0-255 were accepted in the slogan. Now any Unicode string is valid.
hd(List) -> Head

Types:
  List = nonempty_maybe_improper_list()
  Head = term()

Returns the head of List, that is, the first element.
It works with improper lists.

Examples:

```erlang
hd([1,2,3,4,5]).
1
hd([first, second, third, so_on | improper_end]).
first
```

Allowed in guard tests.

Failure: badarg if List is an empty list [].

erlang:hibernate(Module, Function, Args) -> no_return()

Types:
  Module = module()
  Function = atom()
  Args = [term()]

Puts the calling process into a wait state where its memory allocation has been reduced as much as possible. This is useful if the process does not expect to receive any messages soon.

The process is awakened when a message is sent to it, and control resumes in Module:Function with the arguments specified by Args with the call stack emptied, meaning that the process terminates when that function returns. Thus erlang:hibernate/3 never returns to its caller. The resume function Module:Function/Arity must be exported (Arity =:= length(Args)).

If the process has any message in its message queue, the process is awakened immediately in the same way as described earlier.

In more technical terms, erlang:hibernate/3 discards the call stack for the process, and then garbage collects the process. After this, all live data is in one continuous heap. The heap is then shrunk to the exact same size as the live data that it holds (even if that size is less than the minimum heap size for the process).

If the size of the live data in the process is less than the minimum heap size, the first garbage collection occurring after the process is awakened ensures that the heap size is changed to a size not smaller than the minimum heap size.

Notice that emptying the call stack means that any surrounding catch is removed and must be re-inserted after hibernation. One effect of this is that processes started using proc_lib (also indirectly, such as gen_server processes), are to use proc_lib:hibernate/3 instead, to ensure that the exception handler continues to work when the process wakes up.

erlang:insert_element(Index, Tuple1, Term) -> Tuple2

Types:
  Index = integer() >= 1
  1..tuple_size(Tuple1) + 1
erlang

Tuple1 = Tuple2 = tuple()
Term = term()

Returns a new tuple with element Term inserted at position Index in tuple Tuple1. All elements from position Index and upwards are pushed one step higher in the new tuple Tuple2. Example:

```erl
> erlang:insert_element(2, {one, two, three}, new).
{one,new,two,three}
```

integer_to_binary(Integer) -> binary()

Types:
Integer = integer()

Returns a binary corresponding to the text representation of Integer, for example:

```erl
> integer_to_binary(77).
<<"77">>
```

integer_to_binary(Integer, Base) -> binary()

Types:
Integer = integer()
Base = 2..36

Returns a binary corresponding to the text representation of Integer in base Base, for example:

```erl
> integer_to_binary(1023, 16).
<<"3FF">>
```

integer_to_list(Integer) -> string()

Types:
Integer = integer()

Returns a string corresponding to the text representation of Integer, for example:

```erl
> integer_to_list(77).
"77"
```

integer_to_list(Integer, Base) -> string()

Types:
Integer = integer()
Base = 2..36

Returns a string corresponding to the text representation of Integer in base Base, for example:

```erl
> integer_to_list(1023, 16).
"3FF"
```
iolist_size(Item) -> integer() \( \geq 0 \)

Types:
- Item = iolist() | binary()

Returns an integer, that is the size in bytes, of the binary that would be the result of iolist_to_binary(Item), for example:

```erlang
> iolist_size([1,2|<<3,4>>]).
4
```

iolist_to_binary(IoListOrBinary) -> binary()

Types:
- IoListOrBinary = iolist() | binary()

Returns a binary that is made from the integers and binaries in IoListOrBinary, for example:

```erlang
> Bin1 = <<1,2,3>>.
<<1,2,3>>
> Bin2 = <<4,5>>.
<<4,5>>
> Bin3 = <<6>>.
<<6>>
> iolist_to_binary([Bin1,1,[2,3,Bin2],4|Bin3]).
<<1,2,3,1,2,3,4,5,4,6>>
```

erlang:iolist_to_iovec(IoListOrBinary) -> iovec()

Types:
- IoListOrBinary = iolist() | binary()

Returns an iovec that is made from the integers and binaries in IoListOrBinary. This function is useful when you want to flatten an iolist but you do not need a single binary. This can be useful for passing the data to nif functions such as enif_inspect_iovec or do more efficient message passing. The advantage of using this function over iolist_to_binary/1 is that it does not have to copy off-heap binaries. Example:

```erlang
> Bin1 = <<1,2,3>>.
<<1,2,3>>
> Bin2 = <<4,5>>.
<<4,5>>
> Bin3 = <<6>>.
<<6>>
%% If you pass small binaries and integers it works as iolist_to_binary
> erlang:iolist_to_iovec([Bin1,1,[2,3,Bin2],4|Bin3]).
[<<1,2,3,1,2,3,4,5,4,6>>]
%% If you pass larger binaries, they are split and returned in a form
%% optimized for calling the C function writev.
> erlang:iolist_to_iovec([<<1>>,<<2:8096>>,<<3:8096>>]).
[<<0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,...>>,
  <<0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,...>>,
  <<0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,...>>>]
```
is_alive() -> boolean()
Returns true if the local node is alive (that is, if the node can be part of a distributed system), otherwise false. A node is alive if it is started with:
• "erl -name LONGNAME" or,
• "erl -sname SHORTNAME".
A node can also be alive if it has got a name from a call to net_kernel:start/2 and has not been stopped by a call to net_kernel:stop/0.

is_atom(Term) -> boolean()
Types:
    Term = term()
Returns true if Term is an atom, otherwise false. Allowed in guard tests.

is_binary(Term) -> boolean()
Types:
    Term = term()
Returns true if Term is a binary, otherwise false. A binary always contains a complete number of bytes. Allowed in guard tests.

is_bitstring(Term) -> boolean()
Types:
    Term = term()
Returns true if Term is a bitstring (including a binary), otherwise false. Allowed in guard tests.

is_boolean(Term) -> boolean()
Types:
    Term = term()
Returns true if Term is the atom true or the atom false (that is, a boolean). Otherwise returns false. Allowed in guard tests.

erlang:is_builtin(Module, Function, Arity) -> boolean()
Types:
    Module = module()
    Function = atom()
    Arity = arity()
This BIF is useful for builders of cross-reference tools. Returns true if Module:Function/Arity is a BIF implemented in C, otherwise false.
is_float(Term) -> boolean()
Types:
    Term = term()
Returns true if Term is a floating point number, otherwise false.
Allowed in guard tests.

is_function(Term) -> boolean()
Types:
    Term = term()
Returns true if Term is a fun, otherwise false.
Allowed in guard tests.

is_function(Term, Arity) -> boolean()
Types:
    Term = term()
    Arity = arity()
Returns true if Term is a fun that can be applied with Arity number of arguments, otherwise false.
Allowed in guard tests.

is_integer(Term) -> boolean()
Types:
    Term = term()
Returns true if Term is an integer, otherwise false.
Allowed in guard tests.

is_list(Term) -> boolean()
Types:
    Term = term()
Returns true if Term is a list with zero or more elements, otherwise false.
Allowed in guard tests.

is_map(Term) -> boolean()
Types:
    Term = term()
Returns true if Term is a map, otherwise false.
Allowed in guard tests.

is_map_key(Key, Map) -> boolean()
Types:
    Key = term()
    Map = map()
Returns true if map Map contains Key and returns false if it does not contain the Key.
The call fails with a \{badmap, Map\} exception if Map is not a map.

**Example:**

```erlang
> Map = 
{"42" => value}.
{"42" => value}
> is_map_key("42",Map).
true
> is_map_key(value,Map).
false
```

Allowed in guard tests.

**is_number(Term) -> boolean()**

Types:

```
Term = term()
```

Returns true if Term is an integer or a floating point number. Otherwise returns false.

Allowed in guard tests.

**is_pid(Term) -> boolean()**

Types:

```
Term = term()
```

Returns true if Term is a process identifier, otherwise false.

Allowed in guard tests.

**is_port(Term) -> boolean()**

Types:

```
Term = term()
```

Returns true if Term is a port identifier, otherwise false.

Allowed in guard tests.

**is_process_alive(Pid) -> boolean()**

Types:

```
Pid = pid()
```

Pid must refer to a process at the local node.

Returns true if the process exists and is alive, that is, is not exiting and has not exited. Otherwise returns false.

If process P1 calls \is_process_alive(P2Pid)\ it is guaranteed that all signals, sent from P1 to P2 (P2 is the process with identifier P2Pid) before the call, will be delivered to P2 before the aliveness of P2 is checked. This guarantee means that one can use \is_process_alive/1\ to let a process P1 wait until a process P2, which has got an exit signal with reason \kill\ from P1, is killed. Example:

```erlang
exit(P2Pid, kill),
% P2 might not be killed
is_process_alive(P2Pid),
% P2 is not alive (the call above always return false)
```

See the documentation about signals and erlang:exit/2 for more information about signals and exit signals.
is_record(Term, RecordTag) -> boolean()
Types:
   Term = term()
   RecordTag = atom()
Returns true if Term is a tuple and its first element is RecordTag. Otherwise returns false.

Note:
Normally the compiler treats calls to is_record/2 especially. It emits code to verify that Term is a tuple, that its first element is RecordTag, and that the size is correct. However, if RecordTag is not a literal atom, the BIF is_record/2 is called instead and the size of the tuple is not verified.

Allowed in guard tests, if RecordTag is a literal atom.

is_record(Term, RecordTag, Size) -> boolean()
Types:
   Term = term()
   RecordTag = atom()
   Size = integer() >= 0
RecordTag must be an atom.
Returns true if Term is a tuple, its first element is RecordTag, and its size is Size. Otherwise returns false.
Allowed in guard tests if RecordTag is a literal atom and Size is a literal integer.

Note:
This BIF is documented for completeness. Usually is_record/2 is to be used.

is_reference(Term) -> boolean()
Types:
   Term = term()
Returns true if Term is a reference, otherwise false.
Allowed in guard tests.

is_tuple(Term) -> boolean()
Types:
   Term = term()
Returns true if Term is a tuple, otherwise false.
Allowed in guard tests.

length(List) -> integer() >= 0
Types:
   List = [term()]
Returns the length of List, for example:
> length([1,2,3,4,5,6,7,8,9]).
9

Allowed in guard tests.

link(PidOrPort) -> true
Types:
    PidOrPort = pid() | port()
Sets up and activates a link between the calling process and another process or a port identified by PidOrPort. We will from here on call the identified process or port linkee. If the linkee is a port, it must reside on the same node as the caller.

If one of the participants of a link terminates, it will send an exit signal to the other participant. The exit signal will contain the exit reason of the terminated participant. Other cases when exit signals are triggered due to a link are when no linkee exist (noproc exit reason) and when the connection between linked processes on different nodes is lost or cannot be established (noconnection exit reason).

An existing link can be removed by calling unlink/1. For more information on links and exit signals due to links, see the Processes chapter in the Erlang Reference Manual:

* Links
* Sending Exit Signals
* Receiving Exit Signals

For historical reasons, link/1 has a strange semi-synchronous behavior when it is "cheap" to check if the linkee exists or not, and the caller does not trap exits. If the above is true and the linkee does not exist, link/1 will raise a noproc error exception. The expected behavior would instead have been that link/1 returned true, and the caller later was sent an exit signal with noproc exit reason, but this is unfortunately not the case. The noproc exception is not to be confused with an exit signal with exit reason noproc. Currently it is "cheap" to check if the linkee exists when it is supposed to reside on the same node as the calling process.

The link setup and activation is performed asynchronously. If the link already exists, or if the caller attempts to create a link to itself, nothing is done. A detailed description of the link protocol can be found in the Distribution Protocol chapter of the ERTS User's Guide.

Note:

For some important information about distributed signals, see the Blocking Signaling Over Distribution section in the Processes chapter of the Erlang Reference Manual.

Failure:

* badarg if PidOrPort does not identify a process or a node local port.
* noproc linkee does not exist and it is "cheap" to check if it exists as described above.

list_to_atom(String) -> atom()
Types:
    String = string()
Returns the atom whose text representation is String.

As from Erlang/OTP 20, String may contain any Unicode character. Earlier versions allowed only ISO-latin-1 characters as the implementation did not allow Unicode characters above 255.
Note:

The number of characters that are permitted in an atom name is limited. The default limits can be found in the efficiency guide (section Advanced).

Note:

There is configurable limit on how many atoms that can exist and atoms are not garbage collected. Therefore, it is recommended to consider if list_to_existing_atom/1 is a better option than list_to_atom/1. The default limits can be found in the efficiency guide (section Advanced).

Example:

```erlang
> list_to_atom("Erlang").
'Erlang'
```

**list_to_binary(IoList) -> binary()**

Types:

```
IoList = iolist()
```

Returns a binary that is made from the integers and binaries in IoList, for example:

```erlang
> Bin1 = <<1,2,3>>.
<<1,2,3>>
> Bin2 = <<4,5>>.
<<4,5>>
> Bin3 = <<6>>.
<<6>>
> list_to_binary([Bin1,1,[2,3,Bin2],4|Bin3]).
<<1,2,3,1,2,3,4,5,4,6>>
```

**list_to_bitstring(BitstringList) -> bitstring()**

Types:

```
BitstringList = bitstring_list()
```

```
bitstring_list() =
maybe_improper_list(byte() | bitstring() | bitstring_list()),
bitstring() | []
```

Returns a bitstring that is made from the integers and bitstrings in BitstringList. (The last tail in BitstringList is allowed to be a bitstring.) Example:

```erlang
> Bin1 = <<1,2,3>>.
<<1,2,3>>
> Bin2 = <<4,5>>.
<<4,5>>
> Bin3 = <<6,7:4>>.
<<6,7:4>>
> list_to_bitstring([Bin1,1,[2,3,Bin2],4|Bin3]).
<<1,2,3,1,2,3,4,5,4,6,7:4>>
```
list_to_existing_atom(String) -> atom()

Types:
   String = string()

Returns the atom whose text representation is String, but only if there already exists such atom. An atom exists if it has been created by the run-time system by either loading code or creating a term in which the atom is part.

Failure: badarg if there does not already exist an atom whose text representation is String.

Note:
Note that the compiler may optimize away atoms. For example, the compiler will rewrite atom_to_list(some_atom) to "some_atom". If that expression is the only mention of the atom some_atom in the containing module, the atom will not be created when the module is loaded, and a subsequent call to list_to_existing_atom("some_atom") will fail.

list_to_float(String) -> float()

Types:
   String = string()

Returns the float whose text representation is String, for example:

> list_to_float("2.2017764e+0").
2.2017764

The float string format is the same as the format for Erlang float literals except for that underscores are not permitted.

Failure: badarg if String contains a bad representation of a float.

list_to_integer(String) -> integer()

Types:
   String = string()

Returns an integer whose text representation is String, for example:

> list_to_integer("123").
123

> list_to_integer("-123").
-123

> list_to_integer("+123234982304982309482093833234234").
123234982304982309482093833234234

String must contain at least one digit character and can have an optional prefix consisting of a single "+" or "-" character (that is, String must match the regular expression "^[+-]?[0-9]+$").

Failure: badarg if String contains a bad representation of an integer.

list_to_integer(String, Base) -> integer()

Types:
String = string()
Base = 2..36

Returns an integer whose text representation in base Base is String, for example:

```erlang
> list_to_integer("3FF", 16).
1023

> list_to_integer("+3FF", 16).
1023

> list_to_integer("3ff", 16).
1023

> list_to_integer("3Ff", 16).
1023

> list_to_integer("-3FF", 16).
-1023
```

For example, when Base is 16, String must match the regular expression "^[+-]?(\[0-9\]|\[A-F\]|\[a-f\])+\$".

Failure: badarg if String contains a bad representation of an integer.

### list_to_pid(String) -> pid()

Types:

String = string()

Returns a process identifier whose text representation is a String, for example:

```erlang
> list_to_pid(<0.4.1>).
<0.4.1>
```

Failure: badarg if String contains a bad representation of a process identifier.

**Warning:**

This BIF is intended for debugging and is not to be used in application programs.

### list_to_port(String) -> port()

Types:

String = string()

Returns a port identifier whose text representation is a String, for example:

```erlang
> list_to_port("#Port<0.4>").
#Port<0.4>
```

Failure: badarg if String contains a bad representation of a port identifier.
Warning:
This BIF is intended for debugging and is not to be used in application programs.

list_to_ref(String) -> reference()
Types:
  String = string()
Returns a reference whose text representation is a String, for example:

> list_to_ref("#Ref<0.4192537678.4073193475.71181>").
#Ref<0.4192537678.4073193475.71181>

Failure: badarg if String contains a bad representation of a reference.

Warning:
This BIF is intended for debugging and is not to be used in application programs.

list_to_tuple(List) -> tuple()
Types:
  List = [term()]
Returns a tuple corresponding to List, for example

> list_to_tuple([share, ['Ericsson_B', 163]]).
{share, ['Ericsson_B', 163]}

List can contain any Erlang terms.

load_module(Module, Binary) -> {module, Module} | {error, Reason}
Types:
  Module = module()
  Binary = binary()
  Reason =
    badfile | not_purged | on_load |
    {features_not_allowed, [atom()]}  
If Binary contains the object code for module Module, this BIF loads that object code. If the code for module Module already exists, all export references are replaced so they point to the newly loaded code. The previously loaded code is kept in the system as old code, as there can still be processes executing that code.

Returns either {module, Module}, or {error, Reason} if loading fails. Reason is one of the following:

badfile
  The object code in Binary has an incorrect format or the object code contains code for another module than Module.
not_purged
  Binary contains a module that cannot be loaded because old code for this module already exists.
The code in Binary contains an on_load declaration that must be executed before Binary can become the current code. Any previous current code for Module will remain until the on_load call has finished.

The code in Binary has been compiled with features that are currently not enabled in the runtime system.

**Warning:**

This BIF is intended for the code server (see code(3)) and is not to be used elsewhere.

---

### erlang:load_nif(Path, LoadInfo) -> ok | Error

**Types:**

- **Path** = string()
- **LoadInfo** = term()
- **Error** = {error, {Reason, Text :: string()}}
- **Reason** =
  - load_failed
  - bad_lib
  - load
  - reload
  - upgrade
  - old_code

Loads and links a dynamic library containing native implemented functions (NIFs) for a module. Path is a file path to the shareable object/dynamic library file minus the OS-dependent file extension (.so for Unix and .dll for Windows). Notice that on most OSs the library has to have a different name on disc when an upgrade of the nif is done. If the name is the same, but the contents differ, the old library may be loaded instead. For information on how to implement a NIF library, see erl_nif(3).

LoadInfo can be any term. It is passed on to the library as part of the initialization. A good practice is to include a module version number to support future code upgrade scenarios.

The call to load_nif/2 must be made directly from the Erlang code of the module that the NIF library belongs to. It returns either ok, or {error, {Reason, Text}} if loading fails. Reason is one of the following atoms while Text is a human readable string that can give more information about the failure:

- **load_failed**
  - The OS failed to load the NIF library.
- **bad_lib**
  - The library did not fulfill the requirements as a NIF library of the calling module.
- **load | upgrade**
  - The corresponding library callback was unsuccessful.
- **reload**
  - A NIF library is already loaded for this module instance. The previously deprecated reload feature was removed in OTP 20.
- **old_code**
  - The call to load_nif/2 was made from the old code of a module that has been upgraded; this is not allowed.

If the -nifs() attribute is used (which is recommended), all NIFs in the dynamic library much be declared as such for load_nif/2 to succeed. On the other hand, all functions declared with the -nifs() attribute do not have to be implemented by the dynamic library. This allows a target independent Erlang file to contain fallback implementations for functions that may lack NIF support depending on target OS/hardware platform.

### erlang:loaded() -> [Module]

**Types:**

- **Module** = module()

Returns a list of all loaded Erlang modules (current and old code), including preloaded modules.
erlang:localtime() -> DateTime
Types:
    DateTime = calendar:datetime()
Returns the current local date and time, {{Year, Month, Day}, {Hour, Minute, Second}}, for example:

    > erlang:localtime().
    {{1996,11,6},{14,45,17}}

The time zone and Daylight Saving Time correction depend on the underlying OS. The return value is based on the OS System Time.

erlang:localtime_to_universaltime(Localtime) -> Universaltime
Types:
    Localtime = Universaltime = calendar:datetime()
Converts local date and time to Universal Time Coordinated (UTC), if supported by the underlying OS. Otherwise no conversion is done and Localtime is returned. Example:

    > erlang:localtime_to_universaltime({{1996,11,6},{14,45,17}}).
    {{1996,11,6},{13,45,17}}

Failure: badarg if Localtime denotes an invalid date and time.

erlang:localtime_to_universaltime(Localtime, IsDst) -> Universaltime
Types:
    Localtime = Universaltime = calendar:datetime()
    IsDst = true | false | undefined
Converts local date and time to Universal Time Coordinated (UTC) as erlang:localtime_to_universaltime/1, but the caller decides if Daylight Saving Time is active. If IsDst == true, Localtime is during Daylight Saving Time, if IsDst == false it is not. If IsDst == undefined, the underlying OS can guess, which is the same as calling erlang:localtime_to_universaltime(Localtime).

Examples:

    > erlang:localtime_to_universaltime({{1996,11,6},{14,45,17}}, true).
    {{1996,11,6},{12,45,17}}
    > erlang:localtime_to_universaltime({{1996,11,6},{14,45,17}}, false).
    {{1996,11,6},{13,45,17}}
    > erlang:localtime_to_universaltime({{1996,11,6},{14,45,17}}, undefined).
    {{1996,11,6},{13,45,17}}

Failure: badarg if Localtime denotes an invalid date and time.

make_ref() -> reference()
Returns a unique reference. The reference is unique among connected nodes.
Warning:
Before OTP-23 when a node is restarted multiple times with the same node name, references created on a newer node can be mistaken for a reference created on an older node with the same node name.

erlang:make_tuple(Arity, InitialValue) -> tuple()
Types:
  Arity = arity()
  InitialValue = term()
Creates a new tuple of the specified Arity, where all elements are InitialValue, for example:

```
> erlang:make_tuple(4, []).
{{[],[],[],[]}}
```

erlang:make_tuple(Arity, DefaultValue, InitList) -> tuple()
Types:
  Arity = arity()
  DefaultValue = term()
  InitList = [{Position :: integer() >= 1, term()}]
Creates a tuple of size Arity, where each element has value DefaultValue, and then fills in values from InitList. Each list element in InitList must be a two-tuple, where the first element is a position in the newly created tuple and the second element is any term. If a position occurs more than once in the list, the term corresponding to the last occurrence is used. Example:

```
> erlang:make_tuple(5, [], [{2, ignored}, {5, zz}, {2, aa}]).
{[],aa,[],[],zz}
```

map_get(Key, Map) -> Value
Types:
  Map = map()
  Key = Value = any()
Returns value Value associated with Key if Map contains Key.
The call fails with a {badmap, Map} exception if Map is not a map, or with a {badkey, Key} exception if no value is associated with Key.
Example:

```
> Key = 1337,
  Map = #{42 => value_two, 1337 => "value one", "a" => 1},
  map_get(Key, Map).
"value one"
```
Allowed in guard tests.

map_size(Map) -> integer() >= 0
Types:
Map = map()

Returns an integer, which is the number of key-value pairs in Map, for example:

```erlang
> map_size(#{a=>1, b=>2, c=>3}).
3
```

Allowed in guard tests.

erlang:match_spec_test(MatchAgainst, MatchSpec, Type) ->
TestResult

Types:
- MatchAgainst = [term()] | tuple()
- MatchSpec = term()
- Type = table | trace
- TestResult =
  {ok, term(), [return_trace], [{error | warning, string()}]}
  | {error, [{error | warning, string()}]}

Tests a match specification used in calls to ets:select/2 and erlang:trace_pattern/3. The function tests both a match specification for "syntactic" correctness and runs the match specification against the object. If the match specification contains errors, the tuple `{error, Errors}` is returned, where Errors is a list of natural language descriptions of what was wrong with the match specification.

If Type is table, the object to match against is to be a tuple. The function then returns `{ok, Result, [], Warnings}`, where Result is what would have been the result in a real ets:select/2 call, or false if the match specification does not match the object tuple.

If Type is trace, the object to match against is to be a list. The function returns `{ok, Result, Flags, Warnings}`, where Result is one of the following:

- `true` if a trace message is to be emitted
- `false` if a trace message is not to be emitted
- The message term to be appended to the trace message

Flags is a list containing all the trace flags to be enabled, currently this is only `return_trace`.

This is a useful debugging and test tool, especially when writing complicated match specifications.

See also ets:test_ms/2.

max(Term1, Term2) -> Maximum

Types:
- Term1 = Term2 = Maximum = term()

Returns the largest of Term1 and Term2. If the terms compare equal with the `==` operator, Term1 is returned.

The Expressions section contains descriptions of the `==` operator and how terms are ordered.

Examples:

```erlang
> max(1, 2).
2
```
<table>
<thead>
<tr>
<th>Expression</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>max(1.0, 1).</code></td>
<td><code>1.0</code></td>
</tr>
<tr>
<td><code>max(1, 1.0).</code></td>
<td><code>1</code></td>
</tr>
<tr>
<td><code>max(&quot;abc&quot;, &quot;b&quot;).</code></td>
<td>&quot;b&quot;</td>
</tr>
</tbody>
</table>

Allowed in guard tests.

**Change:**

Allowed in guards tests from Erlang/OTP 26.

---

```erlang
erlang:md5(Data) -> Digest
```

**Types:**

- `Data = iodata()`
- `Digest = binary()`

Computes an MD5 message digest from `Data`, where the length of the digest is 128 bits (16 bytes). `Data` is a binary or a list of small integers and binaries.

For more information about MD5, see **RFC 1321 - The MD5 Message-Digest Algorithm.**

**Warning:**

The MD5 Message-Digest Algorithm is **not** considered safe for code-signing or software-integrity purposes.

```erlang
erlang:md5_final(Context) -> Digest
```

**Types:**

- `Context = Digest = binary()`

Finishes the update of an MD5 `Context` and returns the computed MD5 message digest.

```erlang
erlang:md5_init() -> Context
```

**Types:**

- `Context = binary()`

Creates an MD5 context, to be used in the following calls to `md5_update/2`.

```erlang
erlang:md5_update(Context, Data) -> NewContext
```

**Types:**

- `Context = binary()`
- `Data = iodata()`
- `NewContext = binary()`

Update an MD5 `Context` with `Data` and returns a `NewContext`. 
erlang:memory() -> [{Type, Size}]

Types:

Type = memory_type()
Size = integer() >= 0

memory_type() =
  total | processes | processes_used | system | atom |
  atom_used | binary | code | ets

Returns a list with information about memory dynamically allocated by the Erlang emulator. Each list element is a tuple {Type, Size}. The first element Type is an atom describing memory type. The second element Size is the memory size in bytes.

Memory types:

total
  The total amount of memory currently allocated. This is the same as the sum of the memory size for processes and system.

processes
  The total amount of memory currently allocated for the Erlang processes.

processes_used
  The total amount of memory currently used by the Erlang processes. This is part of the memory presented as processes memory.

system
  The total amount of memory currently allocated for the emulator that is not directly related to any Erlang process. Memory presented as processes is not included in this memory. instrument(3) can be used to get a more detailed breakdown of what memory is part of this type.

atom
  The total amount of memory currently allocated for atoms. This memory is part of the memory presented as system memory.

atom_used
  The total amount of memory currently used for atoms. This memory is part of the memory presented as atom memory.

binary
  The total amount of memory currently allocated for binaries. This memory is part of the memory presented as system memory.

code
  The total amount of memory currently allocated for Erlang code. This memory is part of the memory presented as system memory.

ets
  The total amount of memory currently allocated for ETS tables. This memory is part of the memory presented as system memory.

maximum
  The maximum total amount of memory allocated since the emulator was started. This tuple is only present when the emulator is run with instrumentation.

  For information on how to run the emulator with instrumentation, see instrument(3) and/or erl(1).
The system value is not complete. Some allocated memory that is to be part of this value is not.

When the emulator is run with instrumentation, the system value is more accurate, but memory directly allocated for malloc (and friends) is still not part of the system value. Direct calls to malloc are only done from OS-specific runtime libraries and perhaps from user-implemented Erlang drivers that do not use the memory allocation functions in the driver interface.

As the total value is the sum of processes and system, the error in system propagates to the total value. The different amounts of memory that are summed are not gathered atomically, which introduces an error in the result.

The different values have the following relation to each other. Values beginning with an uppercase letter is not part of the result.

- `total = processes + system`
- `processes = processes_used + ProcessesNotUsed`
- `system = atom + binary + code + ets + OtherSystem`
- `atom = atom_used + AtomNotUsed`
- `RealTotal = processes + RealSystem`
- `RealSystem = system + MissedSystem`

More tuples in the returned list can be added in a future release.

Note:

The total value is supposed to be the total amount of memory dynamically allocated by the emulator. Shared libraries, the code of the emulator itself, and the emulator stacks are not supposed to be included. That is, the total value is not supposed to be equal to the total size of all pages mapped to the emulator.

Also, because of fragmentation and prerreservation of memory areas, the size of the memory segments containing the dynamically allocated memory blocks can be much larger than the total size of the dynamically allocated memory blocks.

Change:

As from ERTS 5.6.4, `erlang:memory/0` requires that all erts Alloc(3) allocators are enabled (default behavior).

Failure: `notsup` if an erts Alloc(3) allocator has been disabled.

```
erlang:memory(Type :: memory_type()) -> integer() >= 0
erlang:memory(TypeList :: [memory_type()]) -> [{memory_type(), integer() >= 0}]
```

Types:

- `memory_type() = [total | processes | processes_used | system | atom | atom_used | binary | code | ets]`

Returns the memory size in bytes allocated for memory of type `Type`. The argument can also be specified as a list of `memory_type()` atoms, in which case a corresponding list of `{memory_type(), Size :: integer >= 0}` tuples is returned.
Change:

As from ERTS 5.6.4, erlang:memory/1 requires that all erts_alloc(3) allocators are enabled (default behavior).

Failures:

badarg
  If Type is not one of the memory types listed in the description of erlang:memory/0.
badarg
  If maximum is passed as Type and the emulator is not run in instrumented mode.
notsup
  If an erts_alloc(3) allocator has been disabled.

See also erlang:memory/0.

min(Term1, Term2) -> Minimum

Types:
  Term1 = Term2 = Minimum = term()

Returns the smallest of Term1 and Term2. If the terms compare equal with the == operator, Term1 is returned.

The Expressions section contains descriptions of the == operator and how terms are ordered.

Examples:

> min(1, 2).
1

> min(1.0, 1).
1.0

> min(1, 1.0).
1

> min("abc", "b").
"abc"

Allowed in guard tests.

Change:

Allowed in guards tests from Erlang/OTP 26.

module_loaded(Module) -> boolean()

Types:
  Module = module()

Returns true if the module Module is loaded as current code; otherwise, false. It does not attempt to load the module.
monitor(Type :: process, Item :: monitor_process_identifier()) -> MonitorRef
monitor(Type :: port, Item :: monitor_port_identifier()) -> MonitorRef
monitor(Type :: time_offset, Item :: clock_service) -> MonitorRef

Types:
MonitorRef = reference()
registered_name() = atom()
registered_process_identifier() = registered_name() | {registered_name(), node()}
monitor_process_identifier() = pid() | registered_process_identifier()
monitor_port_identifier() = port() | registered_name()

Sends a monitor request of type Type to the entity identified by Item. If the monitored entity does not exist or it changes monitored state, the caller of monitor/2 is notified by a message on the following format:

{Tag, MonitorRef, Type, Object, Info}

**Note:**
The monitor request is an asynchronous signal. That is, it takes time before the signal reaches its destination.

Type can be one of the following atoms: process, port or time_offset.

A process or port monitor is triggered only once, after that it is removed from both monitoring process and the monitored entity. Monitors are fired when the monitored process or port terminates, does not exist at the moment of creation, or if the connection to it is lost. If the connection to it is lost, we do not know if it still exists. The monitoring is also turned off when demonitor/1 is called.

A process or port monitor by name resolves the RegisteredName to pid() or port() only once at the moment of monitor instantiation, later changes to the name registration will not affect the existing monitor.

When a process or port monitor is triggered, a 'DOWN' message is sent that has the following pattern:

{'DOWN', MonitorRef, Type, Object, Info}

In the monitor message MonitorRef and Type are the same as described earlier, and:

Object

The monitored entity, which triggered the event. When monitoring a process or a local port, Object will be equal to the pid() or port() that was being monitored. When monitoring process or port by name, Object will have format {RegisteredName, Node} where RegisteredName is the name which has been used with monitor/2 call and Node is local or remote node name (for ports monitored by name, Node is always local node name).

Info

Either the exit reason of the process, noproc (process or port did not exist at the time of monitor creation), or noconnection (no connection to the node where the monitored process resides).
Monitoring a process

Creates monitor between the current process and another process identified by Item, which can be a pid() (local or remote), an atom RegisteredName or a tuple {RegisteredName, Node} for a registered process, located elsewhere.

<table>
<thead>
<tr>
<th>Change:</th>
</tr>
</thead>
<tbody>
<tr>
<td>Before ERTS 10.0 (OTP 21.0), monitoring a process could fail with badarg if the monitored process resided on a primitive node (such as erl_interface or jinterface), where remote process monitoring is not implemented. Now, such a call to monitor will instead succeed and a monitor is created. But the monitor will only supervise the connection. That is, a ('DOWN', _, process, _, noconnection) is the only message that may be received, as the primitive node have no way of reporting the status of the monitored process.</td>
</tr>
</tbody>
</table>

Monitoring a port

Creates monitor between the current process and a port identified by Item, which can be a port() (only local), an atom RegisteredName or a tuple {RegisteredName, Node} for a registered port, located on this node. Note, that attempt to monitor a remote port will result in badarg.

Monitoring a time_offset

Monitors changes in time offset between Erlang monotonic time and Erlang system time. One valid Item exists in combination with the time_offset Type, namely the atom clock_service. Notice that the atom clock_service is not the registered name of a process. In this case it serves as an identifier of the runtime system internal clock service at current runtime system instance.

The monitor is triggered when the time offset is changed. This either if the time offset value is changed, or if the offset is changed from preliminary to final during finalization of the time offset when the single time warp mode is used. When a change from preliminary to final time offset is made, the monitor is triggered once regardless of whether the time offset value was changed or not.

If the runtime system is in multi time warp mode, the time offset is changed when the runtime system detects that the OS system time has changed. The runtime system does, however, not detect this immediately when it occurs. A task checking the time offset is scheduled to execute at least once a minute, so under normal operation this is to be detected within a minute, but during heavy load it can take longer time.

The monitor is not automatically removed after it has been triggered. That is, repeated changes of the time offset trigger the monitor repeatedly.

When the monitor is triggered a 'CHANGE' message is sent to the monitoring process. A 'CHANGE' message has the following pattern:

```
{'CHANGE', MonitorRef, Type, Item, NewTimeOffset}
```

where MonitorRef, Type, and Item are the same as described above, and NewTimeOffset is the new time offset.

When the 'CHANGE' message has been received you are guaranteed not to retrieve the old time offset when calling erlang:time_offset(). Notice that you can observe the change of the time offset when calling erlang:time_offset() before you get the 'CHANGE' message.

Making several calls to monitor/2 for the same Item and/or Type is not an error; it results in as many independent monitoring instances.

The monitor functionality is expected to be extended. That is, other Types and Items are expected to be supported in a future release.
Note:
If or when `monitor/2` is extended, other possible values for `Tag`, `Object`, and `Info` in the monitor message will be introduced.

Note:
For some important information about distributed signals, see the `Blocking Signaling Over Distribution` section in the `Processes` chapter of the `Erlang Reference Manual`.

```erlang
monitor(Type :: process,
        Item :: monitor_process_identifier(),
        Opts :: [monitor_option()]) ->
    MonitorRef

monitor(Type :: port,
        Item :: monitor_port_identifier(),
        Opts :: [monitor_option()]) ->
    MonitorRef

monitor(Type :: time_offset,
        Item :: clock_service,
        Opts :: [monitor_option()]) ->
    MonitorRef
```

Types:
- `MonitorRef = reference()`
- `registered_name() = atom()`
- `registered_process_identifier() =
    registered_name() | {registered_name(), node()}`
- `monitor_process_identifier() =
    pid() | registered_process_identifier()`
- `monitor_port_identifier() = port() | registered_name()`

Provides an option list for modification of monitoring functionality provided by `monitor/2`. The `Type` and `Item` arguments have the same meaning as when passed to `monitor/2`. Currently available options:

```erlang
{alias, UnaliasOpt}
```

- `The returned monitor reference will also become an alias for the calling process. That is, the returned reference can be used for sending messages to the calling process. See also `alias/0`. The `UnaliasOpt` determines how the alias should be deactivated.
  - `explicit_unalias`

    Only an explicit call to `unalias/1` will deactivate the alias.

- `demonitor`

  The alias will be automatically deactivated when the monitor is removed. This either via an explicit call to `demonitor/1` or when it is automatically removed at the same time as a `'DOWN'` message is delivered due to the monitor. The alias can also still be deactivated via a call to `unalias/1`.

- `reply_demonitor`

  The alias will be automatically deactivated when the monitor is removed (see `demonitor` option above) or a reply message sent via the alias is received. When a reply message is received via the alias the monitor will
also be automatically removed. This is useful in client/server scenarios when a client monitors the server and will get the reply via the alias. Once the response is received both the alias and the monitor will be automatically removed regardless of whether the response is a reply or a 'DOWN' message. The alias can also still be deactivated via a call to unalias/1. Note that if the alias is removed using the unalias/1 BIF, the monitor will still be left active.

Example:

```erlang
server() ->
    receive
        {request, AliasReqId, Request} ->
            Result = perform_request(Request),
            AliasReqId ! {reply, AliasReqId, Result}
    end,
send().

client(ServerPid, Request) ->
    AliasMonReqId = monitor(process, ServerPid, [{alias, reply_demonitor}]),
    ServerPid ! {request, AliasMonReqId, Request},
    %% Alias as well as monitor will be automatically deactivated if we
    %% receive a reply or a 'DOWN' message since we used 'reply_demonitor'
    %% as unalias option...
    receive
        {reply, AliasMonReqId, Result} ->
            Result;
        {'DOWN', AliasMonReqId, process, ServerPid, ExitReason} ->
            error(ExitReason)
    end.
```

Note that both the server and the client in this example must be executing on at least OTP 24 systems in order for this to work.

For more information on process aliases see the Process Aliases section of the Erlang Reference Manual.

{tag, UserDefinedTag}

Replace the default Tag with UserDefinedTag in the monitor message delivered when the monitor is triggered. For example, when monitoring a process, the 'DOWN' tag in the down message will be replaced by UserDefinedTag.

An example of how the {tag, UserDefinedTag} option can be used in order to enable the new selective receive optimization, introduced in OTP 24, when making multiple requests to different servers:
server() ->
    receive
        {request, From, ReqId, Request} ->
            Result = perform_request(Request),
            From ! {reply, self(), ReqId, Result}
    end,
    server().

client(ServerPids, Request) when is_list(ServerPids) ->
    ReqId = make_ref(),
    lists:foreach(fun (ServerPid) ->
        _ = monitor(process, ServerPid,
            [{tag, {'DOWN', ReqId}}]),
        ServerPid ! {request, self(), ReqId, Request}
    end,
    ServerPids),
    receive_replies(ReqId, length(ServerPids), []).

receive_replies(_ReqId, 0, Acc) ->
    Acc;
receive_replies(ReqId, N, Acc) ->
    %% The compiler will detect that we match on the 'ReqId'
    %% reference in all clauses, and will enable the selective
    %% receive optimization which makes the receive able to
    %% skip past all messages present in the message queue at
    %% the time when the 'ReqId' reference was created...
    Res = receive
        {reply, ServerPid, ReqId, Result} ->
            %% Here we typically would have deactivated the
            %% monitor by a call to demonitor(Mon, [flush]) but
            %% we ignore this in this example for simplicity...
            {ok, ServerPid, Result};
        {{'DOWN', ReqId}, Mon, process, ServerPid, ExitReason} ->
            {error, ServerPid, ExitReason}
    end,
    receive_replies(ReqId, N-1, [Res | Acc]).

In order for this example to work as intended, the client must be executing on at least an OTP 24 system, but
the servers may execute on older systems.

monitor_node(Node, Flag) -> true
Types:
    Node = node()
    Flag = boolean()

Monitor the status of the node Node. If Flag is true, monitoring is turned on. If Flag is false, monitoring is
turned off.

Making several calls to monitor_node(Node, true) for the same Node is not an error; it results in as many
independent monitoring instances.

If Node fails or does not exist, the message {nodedown, Node} is delivered to the process. If a process has made
two calls to monitor_node(Node, true) and Node terminates, two nodedown messages are delivered to the
process. If there is no connection to Node, an attempt is made to create one. If this fails, a nodedown message is
delivered.

The delivery of the nodedown signal is not ordered with respect to other link or monitor signals from the node that
goes down. If you need a guarantee that all signals from the remote node has been delivered before the nodedown
signal is sent, you should use net_kernel:monitor_nodes/1.

Nodes connected through hidden connections can be monitored as any other nodes.
erlang

Failure: notalive if the local node is not alive.

erlang:monitor_node(Node, Flag, Options) -> true
Types:
  Node = node()
  Flag = boolean()
  Options = [Option]
  Option = allow_passive_connect
Behaves as monitor_node/2 except that it allows an extra option to be specified, namely allow_passive_connect. This option allows the BIF to wait the normal network connection time-out for the monitored node to connect itself, even if it cannot be actively connected from this node (that is, it is blocked). The state where this can be useful can only be achieved by using the Kernel option dist_auto_connect once. If that option is not used, option allow_passive_connect has no effect.

Note:
Option allow_passive_connect is used internally and is seldom needed in applications where the network topology and the Kernel options in effect are known in advance.

Failure: badarg if the local node is not alive or the option list is malformed.

erlang:monotonic_time() -> integer()
Returns the current Erlang monotonic time in native time unit. This is a monotonically increasing time since some unspecified point in time.

Note:
This is a monotonically increasing time, but not a strictly monotonically increasing time. That is, consecutive calls to erlang:monotonic_time/0 can produce the same result.
Different runtime system instances will use different unspecified points in time as base for their Erlang monotonic clocks. That is, it is pointless comparing monotonic times from different runtime system instances. Different runtime system instances can also place this unspecified point in time different relative runtime system start. It can be placed in the future (time at start is a negative value), the past (time at start is a positive value), or the runtime system start (time at start is zero). The monotonic time at runtime system start can be retrieved by calling erlang:system_info(start_time).

erlang:monotonic_time(Unit) -> integer()
Types:
  Unit = time_unit()
Returns the current Erlang monotonic time converted into the Unit passed as argument.
Same as calling erlang:convert_time_unit( erlang:monotonic_time(), native, Unit), however optimized for commonly used Units.

erlang:nif_error(Reason) -> no_return()
Types:
Reason = term()
Works exactly like error/1, but Dialyzer thinks that this BIF will return an arbitrary term. When used in a stub function for a NIF to generate an exception when the NIF library is not loaded, Dialyzer does not generate false warnings.

erlang:nif_error(Reason, Args) -> no_return()
Types:
   Reason = term()
   Args = [term()]
Works exactly like error/2, but Dialyzer thinks that this BIF will return an arbitrary term. When used in a stub function for a NIF to generate an exception when the NIF library is not loaded, Dialyzer does not generate false warnings.

node() -> Node
Types:
   Node = node()
Returns the name of the local node. If the node is not alive, nonode@nohost is returned instead.
Allowed in guard tests.

node(Arg) -> Node
Types:
   Arg = pid() | port() | reference()
   Node = node()
Returns the node where Arg originates. Arg can be a process identifier, a reference, or a port. If Arg originates from the local node and the local node is not alive, nonode@nohost is returned.
Allowed in guard tests.

nodes() -> Nodes
Types:
   Nodes = [node()]
Returns a list of all nodes connected to this node through normal connections (that is, hidden nodes are not listed). Same as nodes(visible).

nodes(Arg) -> Nodes
Types:
   Arg = NodeType | [NodeType]
   Node = node()
NodeType = visible | hidden | connected | this | known
   Nodes = [node()]
Returns a list of nodes according to the argument specified. The returned result, when the argument is a list, is the list of nodes satisfying the disjunction(s) of the list elements.
NodeType:
visible
   Nodes connected to this node through normal connections.
hidden
   Nodes connected to this node through hidden connections.

connected
   All nodes connected to this node.

this
   This node.

known
   Nodes that are known to this node. That is, connected nodes and nodes referred to by process
   identifiers, port identifiers, and references located on this node. The set of known nodes is
   garbage collected. Notice that this garbage collection can be delayed. For more information, see
   erlang:system_info(delayed_node_table_gc).

Some equalities: [node()] = nodes(this), nodes(connected) = nodes([visible, hidden]),
and nodes() = nodes(visible).

nodes(Arg, InfoOpts) -> [NodeInfo]

Types:
   NodeType = visible | hidden | connected | this | known
   Arg = NodeType | [NodeType]
   InfoOpts =
      #{connection_id => boolean(), node_type => boolean()}
   NodeTypeInfo = visible | hidden | this | known
   ConnectionId = undefined | integer()
   Info =
      #{connection_id => ConnectionId, node_type => NodeTypeInfo}
   NodeInfo = {node(), Info}

Returns a list of NodeInfo tuples. The first element is the node name. Nodes to be included in the list are determined
by the first argument Arg in the same way as for nodes(Arg). The second element of NodeInfo tuples is a map
containing further information about the node identified by the first element. The information present in this map is
determined by the InfoOpts map passed as the second argument. Currently the following associations are allowed
in the InfoOpts map:

connection_id => boolean()
   If the value of the association equals true, the Info map in the returned result will contain the key
   connection_id associated with the value ConnectionId. If ConnectionId equals undefined, the
   node is not connected to the node which the caller is executing on, or is the node which the caller is executing on.
   If ConnectionId is an integer, the node is currently connected to the node which the caller is executing on.

   The integer connection identifier value together with a node name identifies a specific connection instance to the
   node with that node name. The connection identifier value is node local. That is, on the other node the connection
   identifier will not be the same value. If a connection is taken down and then taken up again, the connection
   identifier value will change for the connection to that node. The amount of values for connection identifiers are
   limited, so it is possible to see the same value for different instances, but quite unlikely. It is undefined how the
   value change between two consecutive connection instances.

node_type => boolean()
   If the value of the association equals true, the Info map in the returned result will contain the key node_type
   associated with the value NodeTypeInfo. Currently the following node types exist:
The node is connected to the node of the calling process through an ordinary visible connection. That is, the node name would appear in the result returned by \texttt{nodes/0}.

The node is connected to the node of the calling process through a hidden connection. That is, the node name would not appear in the result returned by \texttt{nodes/0}.

This is the node of the calling process.

The node is not connected but known to the node of the calling process.

Example:

\begin{verbatim}
(a@localhost)1> nodes([this, connected], #{connection_id=>true, node_type=>true}).
[{{c@localhost,#{connection_id => 13892108,node_type => hidden}},
 {b@localhost,#{connection_id => 3607553,node_type => visible}},
 {a@localhost,#{connection_id => undefined,node_type => this}}]
(a@localhost)2>
\end{verbatim}

\texttt{now()} \rightarrow \texttt{Timestamp}

\textbf{Types:}

\begin{verbatim}
    Timestamp = timestamp()
    timestamp() =
{\{MegaSecs :: integer() >= 0,
    Secs :: integer() >= 0,
    MicroSecs :: integer() >= 0\}}
\end{verbatim}

\textbf{Warning:}

\textit{This function is deprecated. Do not use it.}

For more information, see section Time and Time Correction in the User’s Guide. Specifically, section Dos and Don’t’s describes what to use instead of \texttt{erlang:now/0}.

Returns the tuple \{\texttt{MegaSecs}, \texttt{Secs}, \texttt{MicroSecs}\}, which is the elapsed time since 00:00 GMT, January 1, 1970 (zero hour), if provided by the underlying OS. Otherwise some other point in time is chosen. It is also guaranteed that the following calls to this BIF return continuously increasing values. Hence, the return value from \texttt{erlang:now/0} can be used to generate unique time stamps. If it is called in a tight loop on a fast machine, the time of the node can become skewed.

Can only be used to check the local time of day if the time-zone information of the underlying OS is properly configured.

\texttt{open_port(PortName, PortSettings)} \rightarrow \texttt{port()}

\textbf{Types:}

\begin{verbatim}
    PortName =
{\{spawn, Command :: string() \| binary()} \| 
{spawn_driver, Command :: string() \| binary()} \| 
{spawn_executable, FileName :: file:name_all()} \|
\end{verbatim}
PortSettings = [Opt]
Opt =
   {packet, N :: 1 | 2 | 4} |
   stream |
   {line, L :: integer() >= 0} |
   {cd, Dir :: string() | binary()} |
   {env, Env ::
      [{Name :: os:env_var_name(),
        Val :: os:env_var_value() | false}]} |
   {args, [string() | binary()]} |
   {arg0, string() | binary()} |
   exit_status | use_stdio | mouse_stdio | stderr_to_stdout |
   in | out | binary | eof |
   {parallelism, Boolean :: boolean()} |
   hide |
   {busy_limits_port, 
    {integer() >= 0, integer() >= 0} | disabled} |
   {busy_limits_msgq, 
    {integer() >= 0, integer() >= 0} | disabled}

Returns a port identifier as the result of opening a new Erlang port. A port can be seen as an external Erlang process.

The name of the executable as well as the arguments specified in cd, env, args, and arg0 are subject to Unicode filename translation if the system is running in Unicode filename mode. To avoid translation or to force, for example UTF-8, supply the executable and/or arguments as a binary in the correct encoding. For details, see the module file(3), the function file:native_name_encoding/0 in Kernel, and the Using Unicode in Erlang User's Guide.

**Note:**
The characters in the name (if specified as a list) can only be > 255 if the Erlang virtual machine is started in Unicode filename translation mode. Otherwise the name of the executable is limited to the ISO Latin-1 character set.

PortNames:
{spawn, Command}

Starts an external program. Command is the name of the external program to be run. Command runs outside the Erlang work space unless an Erlang driver with the name Command is found. If found, that driver is started. A driver runs in the Erlang work space, which means that it is linked with the Erlang runtime system.

For external programs, PATH is searched (or an equivalent method is used to find programs, depending on the OS). This is done by invoking the shell on certain platforms. The first space-separated token of the command is considered as the name of the executable (or driver). This (among other things) makes this option unsuitable for running programs with spaces in filenames or directory names. If spaces in executable filenames are desired, use {spawn_executable, Command} instead.
Warning:

On Unix systems, arguments are passed to a new operating system process as an array of strings but on Windows it is up to the child process to parse them and some Windows programs may apply their own rules, which are inconsistent with the standard C runtime argv parsing.

This is particularly troublesome when invoking .bat, .cmd, or .com files as these run implicitly through cmd.exe, whose argument parsing is vulnerable to malicious input and can be used to run arbitrary shell commands.

Therefore, if you are running on Windows and you execute batch files or .com applications, you must not pass untrusted input as arguments to the program.

This affects both spawn and spawn_executable.

{spawn_executable, FileName}

Works like {spawn, FileName}, but only runs external executables. FileName in its whole is used as the name of the executable, including any spaces. If arguments are to be passed, the PortSettings args and arg0 can be used.

The shell is usually not invoked to start the program, it is executed directly. PATH (or equivalent) is not searched. To find a program in PATH to execute, use os:find_executable/1.

Only if a shell script or .bat file is executed, the appropriate command interpreter is invoked implicitly, but there is still no command-argument expansion or implicit PATH search.

If FileName cannot be run, an error exception is raised, with the POSIX error code as the reason. The error reason can differ between OSs. Typically the error enoent is raised when an attempt is made to run a program that is not found and eacces is raised when the specified file is not executable.

{spawn_driver, Command}

Works like {spawn, Command}, but demands the first (space-separated) token of the command to be the name of a loaded driver. If no driver with that name is loaded, a badarg error is raised.

{fd, In, Out}

Allows an Erlang process to access any currently opened file descriptors used by Erlang. The file descriptor In can be used for standard input, and the file descriptor Out for standard output. It is only used for various servers in the Erlang OS (shell and user). Hence, its use is limited.

PortSettings is a list of settings for the port. The valid settings are as follows:

{packet, N}

Messages are preceded by their length, sent in N bytes, with the most significant byte first. The valid values for N are 1, 2, and 4.

stream

Output messages are sent without packet lengths. A user-defined protocol must be used between the Erlang process and the external object.

{line, L}

Messages are delivered on a per line basis. Each line (delimited by the OS-dependent newline sequence) is delivered in a single message. The message data format is (Flag, Line), where Flag is eol or noeol, and Line is the data delivered (without the newline sequence).

L specifies the maximum line length in bytes. Lines longer than this are delivered in more than one message, with Flag set to noeol for all but the last message. If end of file is encountered anywhere else than immediately...
following a newline sequence, the last line is also delivered with Flag set to noeol. Otherwise lines are
delivered with Flag set to eol.

The {packet, N} and {line, L} settings are mutually exclusive.

{cd, Dir}

Only valid for {spawn, Command} and {spawn_executable, FileName}. The external program
starts using Dir as its working directory. Dir must be a string.

{env, Env}

Types:
Name = os:env_var_name()
Val = os:env_var_value() | false
Env = [{Name, Val}]

Only valid for {spawn, Command}, and {spawn_executable, FileName}. The environment of the
started process is extended using the environment specifications in Env.

Env is to be a list of tuples {Name, Val}, where Name is the name of an environment variable, and Val is
the value it is to have in the spawned port process. Both Name and Val must be strings. The one exception is
Val being the atom false (in analogy with os:getenv/1), which removes the environment variable.

For information about encoding requirements, see documentation of the types for Name and Val.

{args, [ string() | binary() ] }

Only valid for {spawn_executable, FileName} and specifies arguments to the executable. Each
argument is specified as a separate string and (on Unix) eventually ends up as one element each in the argument
vector. On other platforms, a similar behavior is mimicked.

The arguments are not expanded by the shell before they are supplied to the executable. Most notably
this means that file wildcard expansion does not occur. To expand wildcards for the arguments, use
filelib: wildcard/1. Notice that even if the program is a Unix shell script, meaning that the shell ultimately
is invoked, wildcard expansion does not occur, and the script is provided with the untouched arguments. On
Windows, wildcard expansion is always up to the program itself, therefore this is not an issue.

The executable name (also known as argv[0]) is not to be specified in this list. The proper executable name
is automatically used as argv[0], where applicable.

If you explicitly want to set the program name in the argument vector, option arg0 can be used.

{arg0, string() | binary() } 

Only valid for {spawn_executable, FileName} and explicitly specifies the program name argument
when running an executable. This can in some circumstances, on some OSs, be desirable. How the program
responds to this is highly system-dependent and no specific effect is guaranteed.

exit_status

Only valid for {spawn, Command}, where Command refers to an external program, and for
{spawn_executable, FileName}.

When the external process connected to the port exits, a message of the form {Port,
{exit_status, Status}} is sent to the connected process, where Status is the exit status of the external
process. If the program aborts on Unix, the same convention is used as the shells do (that is, 128+signal).

If option eof is specified also, the messages eof and exit_status appear in an unspecified order.

If the port program closes its stdout without exiting, option exit_status does not work.
use_stdio

Only valid for \{spawn, Command\} and \{spawn_executable, FileName\}. It allows the standard input and output (file descriptors 0 and 1) of the spawned (Unix) process for communication with Erlang.

nouse_stdio

The opposite of use_stdio. It uses file descriptors 3 and 4 for communication with Erlang.

stderr_to_stdout

Affects ports to external programs. The executed program gets its standard error file redirected to its standard output file. stderr_to_stdout and nouse_stdio are mutually exclusive.

overlapped_io

Affects ports to external programs on Windows only. The standard input and standard output handles of the port program are, if this option is supplied, opened with flag FILE_FLAG_OVERLAPPED, so that the port program can (and must) do overlapped I/O on its standard handles. This is not normally the case for simple port programs, but an option of value for the experienced Windows programmer. On all other platforms, this option is silently discarded.

in

The port can only be used for input.

out

The port can only be used for output.

binary

All I/O from the port is binary data objects as opposed to lists of bytes.

eof

The port is not closed at the end of the file and does not produce an exit signal. Instead, it remains open and a \{Port, eof\} message is sent to the process holding the port.

hide

When running on Windows, suppresses creation of a new console window when spawning the port program. (This option has no effect on other platforms.)

\{parallelism, Boolean\}

Sets scheduler hint for port parallelism. If set to true, the virtual machine schedules port tasks; when doing so, it improves parallelism in the system. If set to false, the virtual machine tries to perform port tasks immediately, improving latency at the expense of parallelism. The default can be set at system startup by passing command-line argument +spp to erl(1).

\{busy_limits_port, \{Low, High\} | disabled\}

Sets limits that will be used for controlling the busy state of the port.

When the ports internal output queue size becomes larger than or equal to High bytes, it enters the busy state. When the port is in the busy state, processes sending commands to it will be suspended until the port leaves the busy state. Commands are in this context either \{Port, command, Data\} or port_command/[2,3].

The Low limit is automatically adjusted to the same as High if it is set larger than High. Valid range of values for Low and High is [1, (1 bsl (8*erlang:system_info(wordsize)))-2]. If the atom disabled is passed, the port will never enter the busy state.

The defaults are Low = 4096 and High = 8192.
Note that this option is only valid when spawning an executable (port program) by opening the spawn driver and when opening the fd driver. This option will cause a failure with a badarg exception when opening other drivers.

\[
\text{busy_limits_msgq, \{Low, High\} | disabled}
\]

Sets limits that will be used for controlling the busy state of the port message queue.

- When the ports message queue size becomes larger than or equal to High bytes it enters the busy state. When it becomes less than Low bytes it leaves the busy state. When the port message queue is in the busy state, processes sending commands to it will be suspended until the port message queue leaves the busy state. Commands are in this context either \(\text{Port ! \{Owner, \{command, Data\}\}}\) or \(\text{port_command/\{2,3\}}\).
- The Low limit is automatically adjusted to the same as High if it is set larger than High. Valid range of values for Low and High is \([1, (1 \text{bsl (8*erlang:system_info(wordsize))})-2]\). If the atom disabled is passed, the port message queue will never enter the busy state.

Note that if the driver statically has disabled the use of this feature, a failure with a badarg exception will be raised unless this option also is set to disabled or not passed at all.

The defaults are Low = 4096 and High = 8192 unless the driver itself does modifications of these values.

Note that the driver might fail if it also adjust these limits by itself and you have disabled this feature.

The spawn driver (used when spawning an executable) and the fd driver do not disable this feature and do not adjust these limits by themselves.

For more information see the documentation \texttt{erl_drv_busy_msgq_limits()}.

Default is stream for all port types and use_stdio for spawned ports.

Failure: if the port cannot be opened, the exit reason is badarg, system_limit, or the POSIX error code that most closely describes the error, or einval if no POSIX code is appropriate:

- **badarg**
  
  Bad input arguments to open_port.

- **system_limit**
  
  All available ports in the Erlang emulator are in use.

- **enomem**
  
  Not enough memory to create the port.

- **eagain**
  
  No more available OS processes.

- **enametoolong**
  
  Too long external command.

- **emfile**
  
  No more available file descriptors (for the OS process that the Erlang emulator runs in).

- **enfile**
  
  Full file table (for the entire OS).

- **eaccess**
  
  Command specified in \{\texttt{spawn_executable}, Command\} does not point out an executable file.

- **enoent**
  
  FileName specified in \{\texttt{spawn_executable}, FileName\} does not point out an existing file.

During use of a port opened using \{\texttt{spawn}, Name\}, \{\texttt{spawn_driver}, Name\}, or \{\texttt{spawn_executable}, Name\}, errors arising when sending messages to it are reported to the owning process using signals of the form \{'EXIT', Port, PosixCode\}. For the possible values of PosixCode, see \texttt{file(3)}.

The maximum number of ports that can be open at the same time can be configured by passing command-line flag \(+Q\) to \texttt{erl(1)}.
erlang:phash(Term, Range) -> Hash
Types:
   Term = term()
   Range = Hash = integer() >= 1
   Range = 1..2^32, Hash = 1..Range

Warning:
This function is deprecated as erlang:phash2/2 should be used for new code. Note that erlang:phash(X,N) is not necessarily equal to erlang:phash2(X,N)

Portable hash function that gives the same hash for the same Erlang term regardless of machine architecture and ERTS version (the BIF was introduced in ERTS 4.9.1.1). The function returns a hash value for Term within the range 1..Range. The maximum value for Range is 2^32.

erlang:phash2(Term) -> Hash
erlang:phash2(Term, Range) -> Hash
Types:
   Term = term()
   Range = integer() >= 1
   1..2^32
   Hash = integer() >= 0
   0..Range-1

Portable hash function that gives the same hash for the same Erlang term regardless of machine architecture and ERTS version (the BIF was introduced in ERTS 5.2). The function returns a hash value for Term within the range 0..Range-1. The maximum value for Range is 2^32. When without argument Range, a value in the range 0..2^27-1 is returned.

This BIF is always to be used for hashing terms. It distributes small integers better than phash/2, and it is faster for bignums and binaries.

Notice that the range 0..Range-1 is different from the range of phash/2, which is 1..Range.

pid_to_list(Pid) -> string()
Types:
   Pid = pid()
Returns a string corresponding to the text representation of Pid. Example:

> erlang:pid_to_list(self()).
"<0.85.0>"
erlang:port_call(Port, Operation, Data) -> term()

Types:
  Port = port() | atom()
  Operation = integer()
  Data = term()

Performs a synchronous call to a port. The meaning of Operation and Data depends on the port, that is, on the port driver. Not all port drivers support this feature.

Port is a port identifier, referring to a driver.

Operation is an integer, which is passed on to the driver.

Data is any Erlang term. This data is converted to binary term format and sent to the port.

Returns a term from the driver. The meaning of the returned data also depends on the port driver.

Failures:

badarg
  If Port is not an identifier of an open port, or the registered name of an open port. If the calling process was previously linked to the closed port, identified by Port, the exit signal from the port is guaranteed to be delivered before this badarg exception occurs.

Warning:

Do not call port_call with an unknown Port identifier and expect badarg exception. Any undefined behavior is possible (including node crash) depending on how the port driver interprets the supplied arguments.

port_close(Port) -> true

Types:
  Port = port() | atom()

Closes an open port. Roughly the same as Port ! {self(), close} except for the error behavior (see below), being synchronous, and that the port does not reply with {Port, closed}. Any process can close a port with port_close/1, not only the port owner (the connected process). If the calling process is linked to the port identified by Port, the exit signal from the port is guaranteed to be delivered before port_close/1 returns.

For comparison: Port ! {self(), close} only fails with badarg if Port does not refer to a port or a process. If Port is a closed port, nothing happens. If Port is an open port and the calling process is the port owner, the port replies with {Port, closed} when all buffers have been flushed and the port really closes. If the calling process is not the port owner, the port owner fails with badsig.

Notice that any process can close a port using Port ! {PortOwner, close} as if it itself was the port owner, but the reply always goes to the port owner.

As from Erlang/OTP R16, Port ! {PortOwner, close} is truly asynchronous. Notice that this operation has always been documented as an asynchronous operation, while the underlying implementation has been synchronous. port_close/1 is however still fully synchronous because of its error behavior.
Failure: badarg if Port is not an identifier of an open port, or the registered name of an open port. If the calling process was previously linked to the closed port, identified by Port, the exit signal from the port is guaranteed to be delivered before this badarg exception occurs.

\[
\text{port\_command}(\text{Port}, \text{Data}) \rightarrow \text{true}
\]
Types:
- Port = port() | atom()
- Data = iodata()
Sends data to a port. Same as \(\text{Port} \rightarrow \{	ext{PortOwner}, \{	ext{command}, \text{Data}\}\}\) except for the error behavior and being synchronous (see below). Any process can send data to a port with \text{port\_command}/2, not only the port owner (the connected process).

For comparison: \(\text{Port} \rightarrow \{	ext{PortOwner}, \{	ext{command}, \text{Data}\}\}\) only fails with badarg if Port does not refer to a port or a process. If Port is a closed port, the data message disappears without a sound. If Port is open and the calling process is not the port owner, the \text{port owner} fails with \text{badsig}. The port owner fails with \text{badsig} also if Data is an invalid I/O list.

Notice that any process can send to a port using \(\text{Port} \rightarrow \{	ext{PortOwner}, \{	ext{command}, \text{Data}\}\}\) as if it itself was the port owner.

If the port is busy, the calling process is suspended until the port is not busy any more.

As from Erlang/OTP R16, \(\text{Port} \rightarrow \{	ext{PortOwner}, \{	ext{command}, \text{Data}\}\}\) is truly asynchronous. Notice that this operation has always been documented as an asynchronous operation, while the underlying implementation has been synchronous. \text{port\_command}/2 is however still fully synchronous because of its error behavior.

Failures:
- badarg
  - If Port is not an identifier of an open port, or the registered name of an open port. If the calling process was previously linked to the closed port, identified by Port, the exit signal from the port is guaranteed to be delivered before this badarg exception occurs.

- badarg
  - If Data is an invalid I/O list.

\[\text{Warning:}\]
Do not send data to an unknown port. Any undefined behavior is possible (including node crash) depending on how the port driver interprets the data.

\[
\text{port\_command}(\text{Port}, \text{Data}, \text{OptionList}) \rightarrow \text{boolean()}
\]
Types:
- Port = port() | atom()
- Data = iodata()
- Option = force | nosuspend
- OptionList = [Option]
Sends data to a port. \text{port\_command}(\text{Port}, \text{Data}, []) equals \text{port\_command}(\text{Port}, \text{Data}).
If the port command is aborted, \text{false} is returned, otherwise \text{true}.
If the port is busy, the calling process is suspended until the port is not busy anymore.
Options:

force
The calling process is not suspended if the port is busy, instead the port command is forced through. The call fails with a notsup exception if the driver of the port does not support this. For more information, see driver flag ERL_DRV_FLAG_SOFT_BUSY.
	nosuspend
The calling process is not suspended if the port is busy, instead the port command is aborted and false is returned.

Change:

More options can be added in a future release.

Failures:

badarg
If Port is not an identifier of an open port, or the registered name of an open port. If the calling process was previously linked to the closed port, identified by Port, the exit signal from the port is guaranteed to be delivered before this badarg exception occurs.

badarg
If Data is an invalid I/O list.

badarg
If OptionList is an invalid option list.

notsup
If option force has been passed, but the driver of the port does not allow forcing through a busy port.

Warning:

Do not send data to an unknown port. Any undefined behavior is possible (including node crash) depending on how the port driver interprets the data.

port_connect(Port, Pid) -> true
Types:

    Port = port() | atom()
    Pid = pid()

Sets the port owner (the connected port) to Pid. Roughly the same as Port ! {Owner, {connect, Pid}} except for the following:

• The error behavior differs, see below.
• The port does not reply with {Port, connected}.
• port_connect/1 is synchronous, see below.
• The new port owner gets linked to the port.

The old port owner stays linked to the port and must call unlink(Port) if this is not desired. Any process can set the port owner to be any process with port_connect/2.

For comparison: Port ! {self(), (connect, Pid)} only fails with badarg if Port does not refer to a port or a process. If Port is a closed port, nothing happens. If Port is an open port and the calling process is the port owner, the port replies with {Port, connected} to the old port owner. Notice that the old port owner is still linked to the port, while the new is not. If Port is an open port and the calling process is not the port owner, the port owner fails with badsig. The port owner fails with badsig also if Pid is not an existing local process identifier.
Notice that any process can set the port owner using `Port ! {PortOwner, {connect, Pid}}` as if it itself was the port owner, but the reply always goes to the port owner.

As from Erlang/OTP R16, `Port ! {PortOwner, {connect, Pid}}` is truly asynchronous. Notice that this operation has always been documented as an asynchronous operation, while the underlying implementation has been synchronous. `port_connect/2` is however still fully synchronous because of its error behavior.

Failures:

badarg

- If `Port` is not an identifier of an open port, or the registered name of an open port. If the calling process was previously linked to the closed port, identified by `Port`, the exit signal from the port is guaranteed to be delivered before this `badarg` exception occurs.

badarg

- If the process identified by `Pid` is not an existing local process.

**port_control(Port, Operation, Data) -> iodata() | binary()**

Types:

- `Port = port() | atom()`
- `Operation = integer()`
- `Data = iodata()`

Performs a synchronous control operation on a port. The meaning of `Operation` and `Data` depends on the port, that is, on the port driver. Not all port drivers support this control feature.

Returns a list of integers in the range 0..255, or a binary, depending on the port driver. The meaning of the returned data also depends on the port driver.

Failures:

badarg

- If `Port` is not an open port or the registered name of an open port.

badarg

- If `Operation` cannot fit in a 32-bit integer.

badarg

- If the port driver does not support synchronous control operations.

badarg

- If the port driver so decides for any reason (probably something wrong with `Operation` or `Data`).

**Warning:**

Do not call `port_control/3` with an unknown `Port` identifier and expect `badarg` exception. Any undefined behavior is possible (including node crash) depending on how the port driver interprets the supplied arguments.

**erlang:port_info(Port) -> Result**

Types:

- `Port = port() | atom()`
- `ResultItem = {registered_name, RegisteredName :: atom()} | {id, Index :: integer() >= 0} | {connected, Pid :: pid()} | {links, Pids :: [pid()]} |`
erlang

Result = [{name, String :: string()} | {input, Bytes :: integer() >= 0} | {output, Bytes :: integer() >= 0} | {os_pid, OsPid :: integer() >= 0 | undefined}

Returns a list containing tuples with information about Port, or undefined if the port is not open. The order of the tuples is undefined, and all the tuples are not mandatory. If the port is closed and the calling process was previously linked to the port, the exit signal from the port is guaranteed to be delivered before port_info/1 returns undefined.

The result contains information about the following Items:

• registered_name (if the port has a registered name)
• id
• connected
• links
• name
• input
• output

For more information about the different Items, see port_info/2.

Failure: badarg if Port is not a local port identifier, or an atom.

erlang:port_info(Port, Item :: connected) ->
{connected, Pid} | undefined

Types:
  Port = port() | atom()
  Pid = pid()

Pid is the process identifier of the process connected to the port.

If the port identified by Port is not open, undefined is returned. If the port is closed and the calling process was previously linked to the port, the exit signal from the port is guaranteed to be delivered before port_info/2 returns undefined.

Failure: badarg if Port is not a local port identifier, or an atom.

erlang:port_info(Port, Item :: id) -> {id, Index} | undefined

Types:
  Port = port() | atom()
  Index = integer() >= 0

Index is the internal index of the port. This index can be used to separate ports.

If the port identified by Port is not open, undefined is returned. If the port is closed and the calling process was previously linked to the port, the exit signal from the port is guaranteed to be delivered before port_info/2 returns undefined.

Failure: badarg if Port is not a local port identifier, or an atom.

erlang:port_info(Port, Item :: input) ->
{input, Bytes} | undefined

Types:
Port = port() | atom()

Bytes = integer() >= 0

Bytes is the total number of bytes read from the port.

If the port identified by Port is not open, undefined is returned. If the port is closed and the calling process was previously linked to the port, the exit signal from the port is guaranteed to be delivered before port_info/2 returns undefined.

Failure: badarg if Port is not a local port identifier, or an atom.

erlang:port_info(Port, Item :: links) -> {links, Pids} | undefined

Types:

    Port = port() | atom()
    Pids = [pid()]

Pids is a list of the process identifiers of the processes that the port is linked to.

If the port identified by Port is not open, undefined is returned. If the port is closed and the calling process was previously linked to the port, the exit signal from the port is guaranteed to be delivered before port_info/2 returns undefined.

Failure: badarg if Port is not a local port identifier, or an atom.

erlang:port_info(Port, Item :: locking) -> {locking, Locking} | undefined

Types:

    Port = port() | atom()
    Locking = false | port_level | driver_level

Locking is one of the following:

- port_level (port-specific locking)
- driver_level (driver-specific locking)

Notice that these results are highly implementation-specific and can change in a future release.

If the port identified by Port is not open, undefined is returned. If the port is closed and the calling process was previously linked to the port, the exit signal from the port is guaranteed to be delivered before port_info/2 returns undefined.

Failure: badarg if Port is not a local port identifier, or an atom.

erlang:port_info(Port, Item :: memory) -> {memory, Bytes} | undefined

Types:

    Port = port() | atom()
    Bytes = integer() >= 0

Bytes is the total number of bytes allocated for this port by the runtime system. The port itself can have allocated memory that is not included in Bytes.

If the port identified by Port is not open, undefined is returned. If the port is closed and the calling process was previously linked to the port, the exit signal from the port is guaranteed to be delivered before port_info/2 returns undefined.

Failure: badarg if Port is not a local port identifier, or an atom.
erlang:port_info(Port, Item :: monitors) ->
    {monitors, Monitors} | undefined

Types:
  Port = port() | atom()
  Monitors = [{process, pid()}]

Monitors represent processes monitored by this port.
If the port identified by Port is not open, undefined is returned. If the port is closed and the calling process was previously linked to the port, the exit signal from the port is guaranteed to be delivered before port_info/2 returns undefined.
Failure: badarg if Port is not a local port identifier, or an atom.

erlang:port_info(Port, Item :: monitored_by) ->
    {monitored_by, MonitoredBy} | undefined

Types:
  Port = port() | atom()
  MonitoredBy = [pid()]

Returns list of pids that are monitoring given port at the moment.
If the port identified by Port is not open, undefined is returned. If the port is closed and the calling process was previously linked to the port, the exit signal from the port is guaranteed to be delivered before port_info/2 returns undefined.
Failure: badarg if Port is not a local port identifier, or an atom.

erlang:port_info(Port, Item :: name) -> {name, Name} | undefined

Types:
  Port = port() | atom()
  Name = string()

Name is the command name set by open_port/2.
If the port identified by Port is not open, undefined is returned. If the port is closed and the calling process was previously linked to the port, the exit signal from the port is guaranteed to be delivered before port_info/2 returns undefined.
Failure: badarg if Port is not a local port identifier, or an atom.

erlang:port_info(Port, Item :: os_pid) ->
    {os_pid, OsPid} | undefined

Types:
  Port = port() | atom()
  OsPid = integer() >= 0 | undefined

OsPid is the process identifier (or equivalent) of an OS process created with open_port({spawn | spawn_executable, Command}, Options). If the port is not the result of spawning an OS process, the value is undefined.
If the port identified by Port is not open, undefined is returned. If the port is closed and the calling process was previously linked to the port, the exit signal from the port is guaranteed to be delivered before port_info/2 returns undefined.
Failure: badarg if Port is not a local port identifier, or an atom.
erlang:port_info(Port, Item :: output) ->
    {output, Bytes} | undefined

Types:
    Port = port() | atom()
    Bytes = integer() >= 0

Bytes is the total number of bytes written to the port from Erlang processes using port_command/2, port_command/3, or Port ! {Owner, {command, Data}. If the port identified by Port is not open, undefined is returned. If the port is closed and the calling process was previously linked to the port, the exit signal from the port is guaranteed to be delivered before port_info/2 returns undefined.

Failure: badarg if Port is not a local port identifier, or an atom.

erlang:port_info(Port, Item :: parallelism) ->
    {parallelism, Boolean} | undefined

Types:
    Port = port() | atom()
    Boolean = boolean()

Boolean corresponds to the port parallelism hint used by this port. For more information, see option parallelism of open_port/2.

erlang:port_info(Port, Item :: queue_size) ->
    {queue_size, Bytes} | undefined

Types:
    Port = port() | atom()
    Bytes = integer() >= 0

Bytes is the total number of bytes queued by the port using the ERTS driver queue implementation. If the port identified by Port is not open, undefined is returned. If the port is closed and the calling process was previously linked to the port, the exit signal from the port is guaranteed to be delivered before port_info/2 returns undefined.

Failure: badarg if Port is not a local port identifier, or an atom.

erlang:port_info(Port, Item :: registered_name) ->
    {registered_name, RegisteredName} |
    [] | undefined

Types:
    Port = port() | atom()
    RegisteredName = atom()

RegisteredName is the registered name of the port. If the port has no registered name, [] is returned. If the port identified by Port is not open, undefined is returned. If the port is closed and the calling process was previously linked to the port, the exit signal from the port is guaranteed to be delivered before port_info/2 returns undefined.

Failure: badarg if Port is not a local port identifier, or an atom.
port_to_list(Port) -> string()
Types:
   Port = port()
Returns a string corresponding to the text representation of the port identifier Port.

erlang:ports() -> [port()]
Returns a list of port identifiers corresponding to all the ports existing on the local node.
Notice that an exiting port exists, but is not open.

pre_loaded() -> [module()]
Returns a list of Erlang modules that are preloaded in the run-time system. Pre-loaded modules are Erlang modules that are needed to bootstrap the system to load the first Erlang modules from either disk or by using erl_boot_server.

erlang:process_display(Pid, Type) -> true
Types:
   Pid = pid()
   Type = backtrace
Writes information about the local process Pid on standard error. The only allowed value for the atom Type is backtrace, which shows the contents of the call stack, including information about the call chain, with the current function printed first. The format of the output is not further defined.

process_flag(Flag :: async_dist, Boolean) -> OldBoolean
Types:
   Boolean = OldBoolean = boolean()
Enable or disable fully asynchronous distributed signaling for the calling process. When disabled, which is the default, the process sending a distributed signal will block in the send operation if the buffer for the distribution channel reach the distribution buffer busy limit. The process will remain blocked until the buffer shrinks enough. This might in some cases take a substantial amount of time. When async_dist is enabled, send operations of distributed signals will always buffer the signal on the outgoing distribution channel and then immediately return. That is, these send operations will never block the sending process.

Note:
Since no flow control is enforced by the runtime system when async_dist process flag is enabled, you need to make sure that flow control for such data is implemented, or that the amount of such data is known to always be limited. Unlimited signaling with async_dist enabled in the absence of flow control will typically cause the sending runtime system to crash on an out of memory condition.

Blocking due to disabled async_dist can be monitored by erlang:system_monitor() using the busy_dist_port option. Only data buffered by processes which (at the time of sending a signal) have disabled async_dist will be counted when determining whether or not an operation should block the caller.

The async_dist flag can also be set on a new process when spawning it using the spawn_opt() BIF with the option {async_dist, Enable}. The default async_dist flag to use on newly spawned processes can be set by passing the command line argument +pad <boolean> when starting the runtime system. If the +pad <boolean> command line argument is not passed, the default value of the async_dist flag will be false.
You can inspect the state of the async_dist process flag of a process by calling process_info(Pid, async_dist).

Returns the old value of the async_dist flag.

```erlang
process_flag(Flag :: trap_exit, Boolean) -> OldBoolean
Types:
    Boolean = OldBoolean = boolean()
When trap_exit is set to true, exit signals arriving to a process are converted to {'EXIT', From, Reason} messages, which can be received as ordinary messages. If trap_exit is set to false, the process exits if it receives an exit signal other than normal and the exit signal is propagated to its linked processes. Application processes are normally not to trap exits.
Returns the old value of the flag.
See also exit/2.
```

```erlang
process_flag(Flag :: error_handler, Module) -> OldModule
Types:
    Module = OldModule = atom()
Used by a process to redefine the error handler for undefined function calls and undefined registered processes. Use this flag with substantial caution, as code auto-loading depends on the correct operation of the error handling module.
Returns the old value of the flag.
```

```erlang
process_flag(Flag :: fullsweep_after, FullsweepAfter) -> OldFullsweepAfter
Types:
    FullsweepAfter = OldFullsweepAfter = integer() >= 0
Changes the maximum number of generational collections before forcing a fullsweep for the calling process.
Returns the old value of the flag.
```

```erlang
process_flag(Flag :: min_heap_size, MinHeapSize) -> OldMinHeapSize
Types:
    MinHeapSize = OldMinHeapSize = integer() >= 0
Changes the minimum heap size for the calling process.
Returns the old value of the flag.
```

```erlang
process_flag(Flag :: min_bin_vheap_size, MinBinVHeapSize) -> OldMinBinVHeapSize
Types:
    MinBinVHeapSize = OldMinBinVHeapSize = integer() >= 0
Changes the minimum binary virtual heap size for the calling process.
Returns the old value of the flag.
```
process_flag(Flag :: max_heap_size, MaxHeapSize) -> OldMaxHeapSize

Types:

MaxHeapSize = OldMaxHeapSize = max_heap_size()

max_heap_size() =
    integer() >= 0 |
    #{size => integer() >= 0,
        kill => boolean(),
        error_logger => boolean(),
        include_shared_binaries => boolean()}

This flag sets the maximum heap size for the calling process. If MaxHeapSize is an integer, the system default values for kill and error_logger are used.

For details on how the heap grows, see Sizing the heap in the ERTS internal documentation.

size

The maximum size in words of the process. If set to zero, the heap size limit is disabled. badarg is be thrown if the value is smaller than min_heap_size. The size check is only done when a garbage collection is triggered.

size is the entire heap of the process when garbage collection is triggered. This includes all generational heaps, the process stack, any messages that are considered to be part of the heap, and any extra memory that the garbage collector needs during collection.

size is the same as can be retrieved using erlang:process_info(Pid, total_heap_size), or by adding heap_block_size, old_heap_block_size and mbuf_size from erlang:process_info(Pid, garbage_collection_info).

kill

When set to true, the runtime system sends an untrappable exit signal with reason kill to the process if the maximum heap size is reached. The garbage collection that triggered the kill is not completed, instead the process exits as soon as possible. When set to false, no exit signal is sent to the process, instead it continues executing.

If kill is not defined in the map, the system default will be used. The default system default is true. It can be changed by either option +hmaxk in erl(1), or erlang:system_flag(max_heap_size, MaxHeapSize).

error_logger

When set to true, the runtime system logs an error event via logger, containing details about the process when the maximum heap size is reached. One log event is sent each time the limit is reached.

If error_logger is not defined in the map, the system default is used. The default system default is true. It can be changed by either the option +hmaxel int erl(1), or erlang:system_flag(max_heap_size, MaxHeapSize).

include_shared_binaries

When set to true, off-heap binaries are included in the total sum compared against the size limit. Off-heap binaries are typically larger binaries that may be shared between processes. The size of a shared binary is included by all processes that are referring it. Also, the entire size of a large binary may be included even if only a smaller part of it is referred by the process.

If include_shared_binaries is not defined in the map, the system default is used. The default system default is false. It can be changed by either the option +hmaxib in erl(1), or erlang:system_flag(max_heap_size, MaxHeapSize).

The heap size of a process is quite hard to predict, especially the amount of memory that is used during the garbage collection. When contemplating using this option, it is recommended to first run it in production with kill set to
false and inspect the log events to see what the normal peak sizes of the processes in the system is and then tune
the value accordingly.

process_flag(Flag :: message_queue_data, MQD) -> OldMQD
Types:
MQD = OldMQD = message_queue_data()
message_queue_data() = off_heap | on_heap
Determines how messages in the message queue are stored, as follows:
off_heap
    All messages in the message queue will be stored outside the process heap. This implies that no messages in the
    message queue will be part of a garbage collection of the process.
on_heap
    All messages in the message queue will eventually be placed on the process heap. They can, however, be
temporarily stored off the heap. This is how messages have always been stored up until ERTS 8.0.
The default value of the message_queue_data process flag is determined by the command-line argument +hmqd
in erl(1).
If the process may potentially accumulate a large number of messages in its queue it is recommended to set the flag
value to off_heap. This is due to the fact that the garbage collection of a process that has a large number of messages
stored on the heap can become extremely expensive and the process can consume large amounts of memory. The
performance of the actual message passing is, however, generally better when the flag value is on_heap.
Changing the flag value causes any existing messages to be moved. The move operation is initiated, but not necessarily
completed, by the time the function returns.
Returns the old value of the flag.

process_flag(Flag :: priority, Level) -> OldLevel
Types:
Level = OldLevel = priority_level()
priority_level() = low | normal | high | max
Sets the process priority. Level is an atom. Four priority levels exist: low, normal, high, and max. Default is
normal.

Note:
Priority level max is reserved for internal use in the Erlang runtime system, and is not to be used by others.
Internally in each priority level, processes are scheduled in a round robin fashion.
Execution of processes on priority normal and low are interleaved. Processes on priority low are selected for
execution less frequently than processes on priority normal.
When runnable processes on priority high exist, no processes on priority low or normal are selected for execution.
Notice however that this does not mean that no processes on priority low or normal can run when processes are
running on priority high. When using multiple schedulers, more processes can be running in parallel than processes
on priority high. That is, a low and a high priority process can execute at the same time.
When runnable processes on priority max exist, no processes on priority low, normal, or high are selected for
execution. As with priority high, processes on lower priorities can execute in parallel with processes on priority max.
Scheduling is pre-emptive. Regardless of priority, a process is pre-empted when it has consumed more than a certain number of reductions since the last time it was selected for execution.

**Note:**

Do not depend on the scheduling to remain exactly as it is today. Scheduling is likely to be changed in a future release to use available processor cores better.

There is no automatic mechanism for avoiding priority inversion, such as priority inheritance or priority ceilings. When using priorities, take this into account and handle such scenarios by yourself.

Making calls from a high priority process into code that you has no control over can cause the high priority process to wait for a process with lower priority. That is, effectively decreasing the priority of the high priority process during the call. Even if this is not the case with one version of the code that you have no control over, it can be the case in a future version of it. This can, for example, occur if a high priority process triggers code loading, as the code server runs on priority normal.

Other priorities than normal are normally not needed. When other priorities are used, use them with care, especially priority high. A process on priority high is only to perform work for short periods. Busy looping for long periods in a high priority process causes most likely problems, as important OTP servers run on priority normal.

Returns the old value of the flag.

```
process_flag(Flag :: save_calls, N) -> OldN
```

Types:

```
N = OldN = 0..10000
```

N must be an integer in the interval 0..10000. If N > 0, call saving is made active for the process. This means that information about the N most recent global function calls, BIF calls, sends, and receives made by the process are saved in a list, which can be retrieved with `process_info(Pid, last_calls)`. A global function call is one in which the module of the function is explicitly mentioned. Only a fixed amount of information is saved, as follows:

- A tuple `{Module, Function, Arity}` for function calls
- The atoms `send`, `receive`, and `timeout` for sends and receives (`receive` when a message is received and `timeout` when a receive times out)

If N = 0, call saving is disabled for the process, which is the default. Whenever the size of the call saving list is set, its contents are reset.

Returns the old value of the flag.

```
process_flag(Flag :: sensitive, Boolean) -> OldBoolean
```

Types:

```
Boolean = OldBoolean = boolean()
```

Sets or clears flag sensitive for the current process. When a process has been marked as sensitive by calling `process_flag(sensitive, true)`, features in the runtime system that can be used for examining the data or inner working of the process are silently disabled.

Features that are disabled include (but are not limited to) the following:

- Tracing. Trace flags can still be set for the process, but no trace messages of any kind are generated. (If flag sensitive is turned off, trace messages are again generated if any trace flags are set.)
- Sequential tracing. The sequential trace token is propagated as usual, but no sequential trace messages are generated.
process_info/1,2 cannot be used to read out the message queue or the process dictionary (both are returned as empty lists).

Stack back-traces cannot be displayed for the process.

In crash dumps, the stack, messages, and the process dictionary are omitted.

If \(\text{save\_calls}, N\) has been set for the process, no function calls are saved to the call saving list. (The call saving list is not cleared. Also, send, receive, and time-out events are still added to the list.)

Returns the old value of the flag.

**process_flag(Pid, Flag, Value) -> OldValue**

Types:

- **Pid** = pid()
- **Flag** = save_calls
- **Value** = OldValue = integer() >= 0

Sets certain flags for the process \(\text{Pid}\), in the same manner as \(\text{process\_flag/2}\). Returns the old value of the flag.

The valid values for **Flag** are only a subset of those allowed in \(\text{process\_flag/2}\), namely save_calls.

Failure: badarg if \(\text{Pid}\) is not a local process.

**process_info(Pid) -> Info**

Types:

- **Pid** = pid()
- **Info** = [InfoTuple] | undefined

**InfoTuple** = process_info_result_item()

**process_info_result_item()** =

- \{async\_dist, Enabled :: boolean()\} |
- \{backtrace, Bin :: binary()\} |
- \{binary, BinInfo :: |
  - \{integer() >= 0, integer() >= 0, integer() >= 0\}\} |
- \{catchlevel, CatchLevel :: integer() >= 0\} |
- \{current\_function, |
  - \{Module :: module(), Function :: atom(), Arity :: arity()\} |
  - undefined\} |
- \{current\_location, |
  - \{Module :: module(), |
    - Function :: atom(), |
    - Arity :: arity(), |
    - Location :: |
      - \{{file, Filename :: string()} | |
      - \{line, Line :: integer() >= 1\}\} | |
      - \{current\_stacktrace, Stack :: [stack\_item()]\} | |
      - \{dictionary, Dictionary :: [{Key :: term(), Value :: term()}]\} | |
      - \{{dictionary, Key :: term()}, Value :: term()\} | |
      - \{error\_handler, Module :: module()\} | |
      - \{garbage\_collection, GCInfo :: [{atom(), integer() >= 0}]\} | |
      - \{garbage\_collection\_info, |
GCInfo :: [{atom(), integer() >= 0}] | 
{group_leader, GroupLeader :: pid()} | 
{heap_size, Size :: integer() >= 0} | 
{initial_call, mfa()} | 
{links, PidsAndPorts :: [pid() | port()]} | 
{last_calls, false | (Calls :: [mfa()])} | 
{memory, Size :: integer() >= 0} | 
{message_queue_len, MessageQueueLen :: integer() >= 0} | 
{messages, MessageQueue :: [term()]} | 
{min_heap_size, MinHeapSize :: integer() >= 0} | 
{min_bin_vheap_size, MinBinVHeapSize :: integer() >= 0} | 
{max_heap_size, MaxHeapSize :: max_heap_size()} | 
{monitored_by, MonitoredBy :: [pid() | port() | nif_resource()]} | 
{monitors, Monitors :: 

{{process | port, 
   Pid :: pid() | port() | 
   {RegName :: atom(), Node :: node()}}} | 
{message_queue_data, MQD :: message_queue_data()} | 
{parent, pid() | undefined} | 
{priority, Level :: priority_level()} | 
{reductions, Number :: integer() >= 0} | 
{registered_name, [] | (Atom :: atom())} | 
{sequential_trace_token, [] | (SequentialTraceToken :: term())} | 
{stack_size, Size :: integer() >= 0} | 
{status, Status :: 
   exiting | garbage_collecting | waiting | running | 
   runnable | suspended} | 
{suspending, SuspendeeList :: 

{{Suspendee :: pid(), 
   ActiveSuspendCount :: integer() >= 0, 
   OutstandingSuspendCount :: integer() >= 0}}} | 
{total_heap_size, Size :: integer() >= 0} | 
{trace, InternalTraceFlags :: integer() >= 0} | 
{trap_exit, Boolean :: boolean()}

priority_level() = low | normal | high | max
stack_item() = 
{Module :: module(), 
 Function :: atom(), 
 Arity :: arity() | (Args :: [term()]), 
 Location :: 
  [{file, Filename :: string()} | 
   {line, Line :: integer() >= 1}]}

max_heap_size() = 
    integer() >= 0 |
Returns a list containing InfoTuples with miscellaneous information about the process identified by Pid, or undefined if the process is not alive.

The order of the InfoTuples is undefined and all InfoTuples are not mandatory. The InfoTuples part of the result can be changed without prior notice.

The InfoTuples with the following items are part of the result:

- current_function
- initial_call
- status
- message_queue_len
- links
- dictionary
- trap_exit
- error_handler
- priority
- group_leader
- total_heap_size
- heap_size
- stack_size
- reductions
- garbage_collection

If the process identified by Pid has a registered name, also an InfoTuple with item registered_name is included.

For information about specific InfoTuples, see process_info/2.

**Warning:**

This BIF is intended for debugging only. For all other purposes, use process_info/2.

**Failure:** badarg if Pid is not a local process.

```
process_info(Pid, Item) -> InfoTuple | [] | undefined
process_info(Pid, ItemList) -> InfoTupleList | [] | undefined
```
async_dist | backtrace | binary | catchlevel |
 current_function | current_location | current_stacktrace |
dictionary |
 {dictionary, Key :: term()} |
 error_handler | garbage_collection | garbage_collection_info |
 group_leader | heap_size | initial_call | links | last_calls |
 memory | message_queue_len | messages | min_heap_size |
 min_bin_vheap_size | monitored_by | monitors |
 message_queue_data | parent | priority | reductions |
 registered_name | sequential_trace_token | stack_size |
 status | suspending | total_heap_size | trace | trap_exit

process_info_result_item() =
 {async_dist, Enabled :: boolean()} |
 {backtrace, Bin :: binary()} |
 {binary, 
 BinInfo :: 
 [{integer() >= 0, integer() >= 0, integer() >= 0}]} |
 {catchlevel, CatchLevel :: integer() >= 0} |
 {current_function, 
 {Module :: module(), Function :: atom(), Arity :: arity()} |
 undefined} |
 {current_location, 
 {Module :: module(), 
 Function :: atom(), 
 Arity :: arity(), 
 Location :: 
 [{file, Filename :: string()} |
 {line, Line :: integer() >= 1}]} | 
 {current_stacktrace, Stack :: [stack_item()]} | 
 {dictionary, Dictionary :: [{Key :: term(), Value :: term()}]} |
 {dictionary, Key :: term()}, Value :: term() |
 {error_handler, Module :: module()} |
 {garbage_collection, GCInfo :: [{atom(), integer() >= 0}]} |
 {garbage_collection_info, 
 GCInfo :: [{atom(), integer() >= 0}]} |
 {group_leader, GroupLeader :: pid()} |
 {heap_size, Size :: integer() >= 0} |
 {initial_call, mfa()} |
 {links, PidsAndPorts :: [pid() | port()]} |
 {last_calls, false | (Calls :: [mfa()])} |
 {memory, Size :: integer() >= 0} |
 {message_queue_len, MessageQueueLen :: integer() >= 0} |
 {messages, MessageQueue :: [term()]} |
 {min_heap_size, MinHeapSize :: integer() >= 0} |
 {min_bin_vheap_size, MinBinVHeapSize :: integer() >= 0} |
 {max_heap_size, MaxHeapSize :: max_heap_size()} |
 {monitored_by, 
 MonitoredBy :: [pid() | port() | nif_resource()]} |
 {monitors, 
 Monitors :: 

Returns information about the process identified by Pid, as specified by Item or ItemList. Returns undefined if the process is not alive.

If the process is alive and a single Item is specified, the returned value is the corresponding InfoTuple, unless Item =:= registered_name and the process has no registered name. In this case, [] is returned. This strange behavior is because of historical reasons, and is kept for backward compatibility.

If ItemList is specified, the result is InfoTupleList. The InfoTuples in InfoTupleList are included with the corresponding Items in the same order as the Items were included in ItemList. Valid Items can be included multiple times in ItemList.
Getting process information follows the signal ordering guarantees described in the Processes Chapter in the *Erlang Reference Manual*.

**Note:**

If `registered_name` is part of `ItemList` and the process has no name registered, a `{registered_name, []}, InfoTuple` will be included in the resulting `InfoTupleList`. This behavior is different when a single `Item =:= registered_name` is specified, and when `process_info/1` is used.

Valid `InfoTuple`s with corresponding `Items`:

- `{async_dist, Enabled}`
  - Since: OTP 25.3
  - Current value of the `async_dist` process flag.

- `{backtrace, Bin}`
  - Binary `Bin` contains the same information as the output from `erlang:process_display(Pid, backtrace)`. Use `binary_to_list/1` to obtain the string of characters from the binary.

- `{binary, BinInfo}`
  - `BinInfo` is a list containing miscellaneous information about binaries on the heap of this process. This `InfoTuple` can be changed or removed without prior notice. In the current implementation, `BinInfo` is a list of tuples. The tuples contain; `BinaryId`, `BinarySize`, `BinaryRefCount`.
  - Depending on the value of the `message_queue_data` process flag the message queue may be stored on the heap.

- `{catchlevel, CatchLevel}`
  - `CatchLevel` is the number of currently active catches in this process. This `InfoTuple` can be changed or removed without prior notice.

- `{current_function, {Module, Function, Arity} | undefined}`
  - `Module`, `Function`, `Arity` is the current function call of the process. The value `undefined` can be returned if the process is currently executing native compiled code.

- `{current_location, {Module, Function, Arity, Location}}`
  - `Module`, `Function`, `Arity` is the current function call of the process. `Location` is a list of two-tuples describing the location in the source code.

- `{current_stacktrace, Stack}`
  - Returns the current call stack back-trace (stacktrace) of the process. The stack has the same format as in the `catch` part of a `try`. See The call-stack back trace (stacktrace). The depth of the stacktrace is truncated according to the `backtrace_depth` system flag setting.

- `{dictionary, Dictionary}`
  - `Dictionary` is the process dictionary.

- `{{dictionary, Key}, Value}`
  - `Value` associated with `Key` in the process dictionary.

- `{error_handler, Module}`
  - `Module` is the error handler module used by the process (for undefined function calls, for example).
{garbage_collection, GCInfo}

GCInfo is a list containing miscellaneous information about garbage collection for this process. The content of GCInfo can be changed without prior notice.

{garbage_collection_info, GCInfo}

GCInfo is a list containing miscellaneous detailed information about garbage collection for this process. The content of GCInfo can be changed without prior notice. For details about the meaning of each item, see gc_minor_start in erlang:trace/3.

{group_leader, GroupLeader}

GroupLeader is the group leader for the I/O of the process.

{heap_size, Size}

Size is the size in words of the youngest heap generation of the process. This generation includes the process stack. This information is highly implementation-dependent, and can change if the implementation changes.

{initial_call, {Module, Function, Arity}}

Module, Function, Arity is the initial function call with which the process was spawned.

{links, PidsAndPorts}

PidsAndPorts is a list of process identifiers and port identifiers, with processes or ports to which the process has a link.

{last_calls, false|Calls}

The value is false if call saving is not active for the process (see process_flag/3). If call saving is active, a list is returned, in which the last element is the most recent called.

{memory, Size}

Size is the size in bytes of the process. This includes call stack, heap, and internal structures.

{message_queue_len, MessageQueueLen}

MessageQueueLen is the number of messages currently in the message queue of the process. This is the length of the list MessageQueue returned as the information item messages (see below).

{messages, MessageQueue}

MessageQueue is a list of the messages to the process, which have not yet been processed.

{min_heap_size, MinHeapSize}

MinHeapSize is the minimum heap size for the process.

{min_bin_vheap_size, MinBinVHeapSize}

MinBinVHeapSize is the minimum binary virtual heap size for the process.

{monitored_by, MonitoredBy}

A list of identifiers for all the processes, ports and NIF resources, that are monitoring the process.

{monitors, Monitors}

A list of monitors (started by monitor/2) that are active for the process. For a local process monitor or a remote process monitor by a process identifier, the list consists of:

{process, Pid}

Process is monitored by pid.

{process, {RegName, Node}}

Local or remote process is monitored by name.
{port, PortId}
   Local port is monitored by port id.
{port, (RegName, Node)}
   Local port is monitored by name. Please note, that remote port monitors are not supported, so Node will always be the local node name.

{message_queue_data, MQD}
   MQD is the current value of the message_queue_data process flag, which can be either off_heap or on_heap. For more information, see the documentation of process_flag(message_queue_data, MQD).

{parent, Pid}
   Pid is the identifier of the parent process, the one that spawned current process. When the process does not have a parent undefined is returned. Only the initial process (init) on a node lacks a parent, though.

{priority, Level}
   Level is the current priority level for the process. For more information on priorities, see process_flag(priority, Level).

{reductions, Number}
   Number is the number of reductions executed by the process.

{registered_name, Atom}
   Atom is the registered process name. If the process has no registered name, this tuple is not present in the list.

{sequential_trace_token, [] | SequentialTraceToken}
   SequentialTraceToken is the sequential trace token for the process. This InfoTuple can be changed or removed without prior notice.

{stack_size, Size}
   Size is the stack size, in words, of the process.

{status, Status}
   Status is the status of the process and is one of the following:
   • exiting
   • garbage_collecting
   • waiting (for a message)
   • running
   • runnable (ready to run, but another process is running)
   • suspended (suspended on a "busy" port or by the BIF erlang:suspend_process/1,2)

{suspending, SuspendingList}
   SuspendingList is a list of {Suspending, ActiveSuspendCount, OutstandingSuspendCount} tuples. Suspending is the process identifier of a process that has been, or is to be, suspended by the process identified by Pid through the BIF erlang:suspend_process/2 or erlang:suspend_process/1.

ActiveSuspendCount is the number of times Suspending has been suspended by Pid. OutstandingSuspendCount is the number of not yet completed suspend requests sent by Pid, that is:
   • If ActiveSuspendCount /= 0, Suspending is currently in the suspended state.
   • If OutstandingSuspendCount /= 0, option asynchronous of erlang:suspend_process/2 has been used and the suspending has not yet been suspended by Pid.
Notice that `ActiveSuspendCount` and `OutstandingSuspendCount` are not the total suspend count on `Suspendee`, only the parts contributed by `Pid`.

\[
\text{total_heap_size, Size}
\]

Size is the total size, in words, of all heap fragments of the process. This includes the process stack and any unreceived messages that are considered to be part of the heap.

\[
\text{trace, InternalTraceFlags}
\]

`InternalTraceFlags` is an integer representing the internal trace flag for this process. This `InfoTuple` can be changed or removed without prior notice.

\[
\text{trap_exit, Boolean}
\]

Boolean is `true` if the process is trapping exits, otherwise `false`.

Notice that not all implementations support all these `Items`.

**Failures:**

- **badarg**
  - If `Pid` is not a local process.
  - If `Item` is an invalid item.

**processes() -> [pid()]**

Returns a list of process identifiers corresponding to all the processes currently existing on the local node.

Notice that an exiting process exists, but is not alive. That is, `is_process_alive/1` returns `false` for an exiting process, but its process identifier is part of the result returned from `processes/0`.

**Example:**

```erlang
> processes().
[<0.0.0>,<0.2.0>,<0.4.0>,<0.5.0>,<0.7.0>,<0.8.0>]
```

**purge_module(Module) -> true**

Types:

- `Module = atom()`

Removes old code for `Module`. Before this BIF is used, `check_process_code/2` is to be called to check that no processes execute old code in the module.

**Warning:**

This BIF is intended for the code server (see `code(3)`) and is not to be used elsewhere.

**Change:**

As from ERTS 8.0 (Erlang/OTP 19), any lingering processes that still execute the old code is killed by this function. In earlier versions, such incorrect use could cause much more fatal failures, like emulator crash.

**Failure:** **badarg** if there is no old code for `Module`. 

Ericsson AB. All Rights Reserved.: Erlang Run-Time System Application (ERTS) | 255
put(Key, Val) -> term()
Types:
  Key = Val = term()

Adds a new Key to the process dictionary, associated with the value Val, and returns undefined. If Key exists, the old value is deleted and replaced by Val, and the function returns the old value. The average time complexity for the current implementation of this function is $O(1)$ and the worst case time complexity is $O(N)$, where $N$ is the number of items in the process dictionary. Example:

```erlang
> X = put(name, walrus), Y = put(name, carpenter),
Z = get(name),
{X, Y, Z}.
{undefined,walrus,carpenter}
```

Note:
The values stored when put is evaluated within the scope of a catch are not retracted if a throw is evaluated, or if an error occurs.

erlang:raise(Class, Reason, Stacktrace) -> badarg
Types:
  Class = error | exit | throw
  Reason = term()
  Stacktrace = raise_stacktrace() | stacktrace()
  raise_stacktrace() =
    [{module(), atom(), arity() | [term()]} |
      {function(), arity() | [term()]}]

Raises an exception of the specified class, reason, and call stack backtrace (stacktrace).

Class is error, exit, or throw. So, if it were not for the stacktrace, erlang:raise(Class, Reason, Stacktrace) is equivalent to erlang:Class(Reason) (given that Class is a valid class).

Reason can be any term.

Stacktrace is a list as provided in a try-catch clause.

```erlang
try ...
  catch Class:Reason:Stacktrace ->
  ... end
```

That is, a list of four-tuples (Module, Function, Arity | Args, ExtraInfo), where Module and Function are atoms, and the third element is an integer arity or an argument list. The stacktrace can also contain {Fun, Args, ExtraInfo} tuples, where Fun is a local fun and Args is an argument list.

Element ExtraInfo at the end is optional. Omitting it is equivalent to specifying an empty list.

The stacktrace is used as the exception stacktrace for the calling process; it is truncated to the current maximum stacktrace depth.

As evaluating this function causes the process to terminate, it has no return value unless the arguments are invalid, in which case the function returns the error reason badarg. If you want to be sure not to return, you can call error(erlang:raise(Class, Reason, Stacktrace)) and hope to distinguish exceptions later.
See the reference manual about errors and error handling for more information about exception classes and how to catch exceptions.

```
erlang:read_timer(TimerRef) -> Result
Types:
   TimerRef = reference()
   Time = integer() >= 0
   Result = Time | false
Reads the state of a timer. The same as calling erlang:read_timer(TimerRef, []).
```

```
erlang:read_timer(TimerRef, Options) -> Result | ok
Types:
   TimerRef = reference()
   Async = boolean()
   Option = {async, Async}
   Options = [Option]
   Time = integer() >= 0
   Result = Time | false
Reads the state of a timer that has been created by either erlang:start_timer or erlang:send_after. TimerRef identifies the timer, and was returned by the BIF that created the timer.
Options:
   {async, Async}
   Asynchronous request for state information. Async defaults to false, which causes the operation to be performed synchronously. In this case, the Result is returned by erlang:read_timer. When Async is true, erlang:read_timer sends an asynchronous request for the state information to the timer service that manages the timer, and then returns ok. A message on the format {read_timer, TimerRef, Result} is sent to the caller of erlang:read_timer when the operation has been processed.
More Options can be added in the future.
If Result is an integer, it represents the time in milliseconds left until the timer expires.
If Result is false, a timer corresponding to TimerRef could not be found. This because the timer had expired, or been canceled, or because TimerRef never has corresponded to a timer. Even if the timer has expired, it does not tell you whether or not the time-out message has arrived at its destination yet.

```

**Note:**
The timer service that manages the timer can be co-located with another scheduler than the scheduler that the calling process is executing on. If so, communication with the timer service takes much longer time than if it is located locally. If the calling process is in a critical path, and can do other things while waiting for the result of this operation, you want to use option {async, true}. If using option {async, false}, the calling process is blocked until the operation has been performed.

See also erlang:send_after/4, erlang:start_timer/4, and erlang:cancel_timer/2.
```

```
ref_to_list(Ref) -> string()
Types:
```

Ericsson AB. All Rights Reserved.: Erlang Run-Time System Application (ERTS) | 257
Ref = reference()

Returns a string corresponding to the text representation of Ref.

**Warning:**

This BIF is intended for debugging and is not to be used in application programs.

register(RegName, PidOrPort) -> true

Types:

- RegName = atom()
- PidOrPort = port() | pid()

Registers the name RegName with a process identifier (pid) or a port identifier in the name registry. RegName, which must be an atom, can be used instead of the pid or port identifier in send operator (RegName ! Message) and most other BIFs that take a pid or port identifies as an argument. Example:

```
> register(db, Pid).
true
```

The registered name is considered a Directly Visible Erlang Resource and is automatically unregistered when the process terminates.

Failures:

- badarg
  - If PidOrPort is not an existing local process or port.
- badarg
  - If RegName is already in use.
- badarg
  - If the process or port is already registered (already has a name).
- badarg
  - If RegName is the atom undefined.

registered() -> [RegName]

Types:

- RegName = atom()

Returns a list of names that have been registered using register/2, for example:

```
> registered().
[code_server, file_server, init, user, my_db]
```

erlang:resume_process(Suspendee) -> true

Types:

- Suspendee = pid()

Decreases the suspend count on the process identified by Suspendee. Suspendee is previously to have been suspended through erlang:suspend_process/2 or erlang:suspend_process/1 by the process calling erlang:resume_process(Suspendee). When the suspend count on Suspendee reaches zero, Suspendee is resumed, that is, its state is changed from suspended into the state it had before it was suspended.
**Warning:**

This BIF is intended for debugging only.

**Failures:**

- **badarg**
  - If `Suspendee` is not a process identifier.
- **badarg**
  - If the process calling `erlang:resume_process/1` had not previously increased the suspend count on the process identified by `Suspendee`.
- **badarg**
  - If the process identified by `Suspendee` is not alive.

**round(Number) -> integer()**

**Types:**

- `Number = number()`

Returns an integer by rounding `Number`, for example:

<table>
<thead>
<tr>
<th>Expression</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>round(42.1)</code></td>
<td><code>42</code></td>
</tr>
<tr>
<td><code>round(5.5)</code></td>
<td><code>6</code></td>
</tr>
<tr>
<td><code>round(-5.5)</code></td>
<td><code>-6</code></td>
</tr>
<tr>
<td><code>round(36028797018963969.0)</code></td>
<td><code>36028797018963968</code></td>
</tr>
</tbody>
</table>

In the last example, `round(36028797018963969.0)` evaluates to `36028797018963968`. The reason for this is that the number `36028797018963969.0` cannot be represented exactly as a float value. Instead, the float literal is represented as `36028797018963968.0`, which is the closest number that can be represented exactly as a float value. See Representation of Floating Point Numbers for additional information.

Allowed in guard tests.

**self() -> pid()**

Returns the process identifier of the calling process, for example:

```
> self().
<0.26.0>
```

Allowed in guard tests.

**erlang:send(Dest, Msg) -> Msg**

**Types:**


erlang

```erlang
Dest = send_destination()
Msg = term()
```

Sends a message and returns `Msg`. This is the same as using the send operator: `Dest ! Msg`.

`Dest` can be a remote or local process identifier, an alias, a (local) port, a locally registered name, or a tuple `{RegName, Node}` for a registered name at another node.

The function fails with a `badarg` run-time error if `Dest` is an atom name, but this name is not registered. This is the only case when `send` fails for an unreachable destination `Dest` (of correct type).

**Note:**

For some important information about distributed signals, see the *Blocking Signaling Over Distribution* section in the *Processes* chapter of the *Erlang Reference Manual*.

### erlang:send/3

```erlang
erlang:send(Dest, Msg, Options) -> Res
```

**Types:**

- `Dest` = `send_destination()`
- `Msg` = `term()`
- `Options` = `[nosuspend | noconnect]`
- `Res` = `ok | nosuspend | noconnect`

Either sends a message and returns `ok`, or does not send the message but returns something else (see below). Otherwise the same as `erlang:send/2`. For more detailed explanation and warnings, see `erlang:send_nosuspend/2,3`.

**Options:**

- **nosuspend**
  
  If the sender would have to be suspended to do the send, `nosuspend` is returned instead.

- **noconnect**
  
  If the destination node would have to be auto-connected to do the send, `noconnect` is returned instead.

**Note:**

For some important information about distributed signals, see the *Blocking Signaling Over Distribution* section in the *Processes* chapter of the *Erlang Reference Manual*.

**Warning:**

As with `erlang:send_nosuspend/2,3`: use with extreme care.

### erlang:send_after/3

```erlang
erlang:send_after(Time, Dest, Msg) -> TimerRef
```

**Types:**

- `Time` = `integer() >= 0`
- `Dest` = `pid() | atom()`
- `Msg` = `term()`
- `TimerRef` = `reference()`

Starts a timer. The same as calling `erlang:send_after(Time, Dest, Msg, [])`.

---

260 | Ericsson AB. All Rights Reserved.: Erlang Run-Time System Application (ERTS)
erlang:send_after(Time, Dest, Msg, Options) -> TimerRef

Types:
  Time = integer()
  Dest = pid() | atom()
  Msg = term()
  Options = [Option]
  Abs = boolean()
  Option = {abs, Abs}
  TimerRef = reference()

Starts a timer. When the timer expires, the message Msg is sent to the process identified by Dest. Apart from the format of the time-out message, this function works exactly as erlang:start_timer/4.

erlang:send_nosuspend(Dest, Msg) -> boolean()

Types:
  Dest = send_destination()
  Msg = term()

The same as erlang:send(Dest, Msg, [nosuspend]), but returns true if the message was sent and false if the message was not sent because the sender would have had to be suspended.

This function is intended for send operations to an unreliable remote node without ever blocking the sending (Erlang) process. If the connection to the remote node (usually not a real Erlang node, but a node written in C or Java) is overloaded, this function does not send the message and returns false.

This function is intended for send operations to an unreliable remote node without ever blocking the sending (Erlang) process. If the connection to the remote node (usually not a real Erlang node, but a node written in C or Java) is overloaded, this function does not send the message and returns false.

The same occurs if Dest refers to a local port that is busy. For all other destinations (allowed for the ordinary send operator '!' ), this function sends the message and returns true.

This function is only to be used in rare circumstances where a process communicates with Erlang nodes that can disappear without any trace, causing the TCP buffers and the drivers queue to be over-full before the node is shut down (because of tick time-outs) by net_kernel. The normal reaction to take when this occurs is some kind of premature shutdown of the other node.

Notice that ignoring the return value from this function would result in an unreliable message passing, which is contradictory to the Erlang programming model. The message is not sent if this function returns false.

In many systems, transient states of overloaded queues are normal. Although this function returns false does not mean that the other node is guaranteed to be non-responsive, it could be a temporary overload. Also, a return value of true does only mean that the message can be sent on the (TCP) channel without blocking; the message is not guaranteed to arrive at the remote node. For a disconnected non-responsive node, the return value is true (mimics the behavior of operator !). The expected behavior and the actions to take when the function returns false are application- and hardware-specific.

**Warning:**

Use with extreme care.

erlang:send_nosuspend(Dest, Msg, Options) -> boolean()

Types:
Dest = send_destination()
Msg = term()
Options = [noconnect]

The same as erlang:send(Dest, Msg, [nosuspend | Options]), but with a Boolean return value.

This function behaves like erlang:send_nosuspend/2, but takes a third parameter, a list of options. The only option is noconnect, which makes the function return false if the remote node is not currently reachable by the local node. The normal behavior is to try to connect to the node, which can stall the process during a short period. The use of option noconnect makes it possible to be sure not to get the slightest delay when sending to a remote process. This is especially useful when communicating with nodes that expect to always be the connecting part (that is, nodes written in C or Java).

Whenever the function returns false (either when a suspend would occur or when noconnect was specified and the node was not already connected), the message is guaranteed not to have been sent.

### Warning:

Use with extreme care.

erlang:set_cookie(Cookie) -> true
Types:

Cookie = atom()

Sets the magic cookie of the local node to the atom Cookie, which is also the cookie for all nodes that have no explicit cookie set with set_cookie/2 Cookie (see section Distributed Erlang in the Erlang Reference Manual in System Documentation).

You can get this value using get_cookie/0.

Failure: function_clause if the local node is not alive.

erlang:set_cookie(Node, Cookie) -> true
Types:

Node = node()
Cookie = atom()

Sets the magic cookie for Node to the atom Cookie. If Node is the local node, the function sets the cookie of all other nodes (that have no explicit cookie set with this function) to Cookie (see section Distributed Erlang in the Erlang Reference Manual in System Documentation).

You can get this value using get_cookie/1.

Failure: function_clause if the local node is not alive.

setelement(Index, Tuple1, Value) -> Tuple2
Types:

Index = integer() >= 1
1..tuple_size(Tuple1)
Tuple1 = Tuple2 = tuple()
Value = term()

Returns a tuple that is a copy of argument Tuple1 with the element specified by integer argument Index (the first element is the element with index 1) replaced by argument Value, for example:
setelement(2, {10, green, bottles}, red).
{10, red, bottles}

size(Item) -> integer() \= 0
Types:
  Item = tuple() | binary()
Returns the number of elements in a tuple or the number of bytes in a binary or bitstring, for example:

> size({morni, mulle, bwange}).
3
> size(<<11, 22, 33>>).
3

For bitstrings, the number of whole bytes is returned. That is, if the number of bits in the bitstring is not divisible by 8, the resulting number of bytes is rounded down.

Allowed in guard tests.
See also tuple_size/1, byte_size/1, and bit_size/1.

spawn(Fun) -> pid()
Types:
  Fun = function()
Returns the process identifier of a new process started by the application of Fun to the empty list []. Otherwise works like spawn/3.

spawn(Node, Fun) -> pid()
Types:
  Node = node()
  Fun = function()
Returns the process identifier of a new process started by the application of Fun to the empty list [] on Node. If Node does not exist, a useless pid is returned. Otherwise works like spawn/3.

spawn(Module, Function, Args) -> pid()
Types:
  Module = module()
  Function = atom()
  Args = [term()]
Returns the process identifier of a new process started by the application of Module:Function to Args.

error_handler:undefined_function(Module, Function, Args) is evaluated by the new process if Module:Function/Arity does not exist (where Arity is the length of Args). The error handler can be redefined (see process_flag/2). If error_handler is undefined, or the user has redefined the default error_handler and its replacement is undefined, a failure with reason undef occurs.

Example:
> spawn(speed, regulator, [high_speed, thin_cut]).
<0.13.1>

spawn(Node, Module, Function, Args) -> pid()
Types:
  Node = node()
  Module = module()
  Function = atom()
  Args = [term()]
Returns the process identifier (pid) of a new process started by the application of Module:Function to Args on Node. If Node does not exist, a useless pid is returned. Otherwise works like spawn/3.

spawn_link(Fun) -> pid()
Types:
  Fun = function()
Returns the process identifier of a new process started by the application of Fun to the empty list []. A link is created between the calling process and the new process, atomically. Otherwise works like spawn/3.

spawn_link(Node, Fun) -> pid()
Types:
  Node = node()
  Fun = function()
Returns the process identifier (pid) of a new process started by the application of Fun to the empty list [] on Node. A link is created between the calling process and the new process, atomically. If Node does not exist, a useless pid is returned and an exit signal with reason noconnection is sent to the calling process. Otherwise works like spawn/3.

spawn_link(Module, Function, Args) -> pid()
Types:
  Module = module()
  Function = atom()
  Args = [term()]
Returns the process identifier of a new process started by the application of Module:Function to Args. A link is created between the calling process and the new process, atomically. Otherwise works like spawn/3.

spawn_link(Node, Module, Function, Args) -> pid()
Types:
  Node = node()
  Module = module()
  Function = atom()
  Args = [term()]
Returns the process identifier (pid) of a new process started by the application of Module:Function to Args on Node. A link is created between the calling process and the new process, atomically. If Node does not exist, a useless
pid is returned and an exit signal with reason noconnection is sent to the calling process. Otherwise works like spawn/3.

spawn_monitor(Fun) -> {pid(), reference()}
Types:
  Fun = function()

Returns the process identifier of a new process, started by the application of Fun to the empty list [], and a reference for a monitor created to the new process. Otherwise works like spawn/3.

spawn_monitor(Node, Fun) -> {pid(), reference()}
Types:
  Node = node()
  Fun = function()

Returns the process identifier of a new process, started by the application of Fun to the empty list [] on the node Node, and a reference for a monitor created to the new process. Otherwise works like spawn/3. If the node identified by Node does not support distributed spawn_monitor(), the call will fail with a notsup exception.

spawn_monitor(Module, Function, Args) -> {pid(), reference()}
Types:
  Module = module()
  Function = atom()
  Args = [term()]

A new process is started by the application of Module:Function to Args. The process is monitored at the same time. Returns the process identifier and a reference for the monitor. Otherwise works like spawn/3.

spawn_monitor(Node, Module, Function, Args) ->
  {pid(), reference()}
Types:
  Node = node()
  Module = module()
  Function = atom()
  Args = [term()]

A new process is started by the application of Module:Function to Args on the node Node. The process is monitored at the same time. Returns the process identifier and a reference for the monitor. Otherwise works like spawn/3. If the node identified by Node does not support distributed spawn_monitor(), the call will fail with a notsup exception.

spawn_opt(Fun, Options) -> pid() | {pid(), reference()}
Types:
Fun = function()
Options = [spawn_opt_option()]
priority_level() = low | normal | high | max
max_heap_size() =
   integer() >= 0 |
   #{size => integer() >= 0, 
   kill => boolean(), 
   error_logger => boolean(), 
   include_shared_binaries => boolean()}
message_queue_data() = off_heap | on_heap
spawn_opt_option() =
   link | monitor | 
   {monitor, MonitorOpts :: [monitor_option()]} | 
   {priority, Level :: priority_level()} | 
   {fullsweep_after, Number :: integer() >= 0} | 
   {min_heap_size, Size :: integer() >= 0} | 
   {min_bin_vheap_size, VSize :: integer() >= 0} | 
   {max_heap_size, Size :: max_heap_size()} | 
   {message_queue_data, MQD :: message_queue_data()} | 
   {async_dist, Enabled :: boolean()}

Returns the process identifier (pid) of a new process started by the application of Fun to the empty list []. Otherwise works like spawn_opt/4.

If option monitor is specified, the newly created process is monitored, and both the pid and reference for the monitor are returned.

spawn_opt(Node, Fun, Options) -> pid() | {pid(), reference()}
Types:
   Node = node()
   Fun = function()
   Options =
      [monitor | {monitor, [monitor_option()]} | link | OtherOption]
   OtherOption = term()

Returns the process identifier (pid) of a new process started by the application of Fun to the empty list [] on Node. If Node does not exist, a useless pid is returned. Otherwise works like spawn_opt/4.

Valid options depends on what options are supported by the node identified by Node. A description of valid Options for the local node of current OTP version can be found in the documentation of spawn_opt/4.

spawn_opt(Module, Function, Args, Options) ->
   Pid | {Pid, MonitorRef}
Types:
Module = module()
Function = atom()
Args = [term()]
Options = [spawn_opt_option()]
Pid = pid()
MonitorRef = reference()
priority_level() = low | normal | high | max
max_heap_size() =
  integer() >= 0 |
  #{size => integer() >= 0, 
  kill => boolean(), 
  error_logger => boolean(), 
  include_shared_binaries => boolean()}
message_queue_data() = off_heap | on_heap
spawn_opt_option() =
  link | monitor |
  {monitor, MonitorOpts :: [monitor_option()]} | 
  {priority, Level :: priority_level()} | 
  {fullsweep_after, Number :: integer() >= 0} | 
  {min_heap_size, Size :: integer() >= 0} | 
  {min_bin_vheap_size, VSize :: integer() >= 0} | 
  {max_heap_size, Size :: max_heap_size()} | 
  {message_queue_data, MQD :: message_queue_data()} | 
  {async_dist, Enabled :: boolean()}

Works as spawn/3, except that an extra option list is specified when creating the process.

If option monitor is specified, the newly created process is monitored, and both the pid and reference for the monitor are returned.

Options:
link

Sets a link to the parent process (like spawn_link/3 does).

monitor

Monitors the new process (like monitor(process, Pid) does). A {Pid, MonitorRef} tuple will be returned instead of just a Pid.

{monitor, MonitorOpts}

Monitors the new process with options (like monitor(process, Pid, MonitorOpts) does). A {Pid, MonitorRef} tuple will be returned instead of just a Pid.

{priority, Level}

Sets the priority of the new process. Equivalent to executing process_flag(priority, Level) in the start function of the new process, except that the priority is set before the process is selected for execution for the first time. For more information on priorities, see process_flag(priority, Level).

{fullsweep_after, Number}

Useful only for performance tuning. Do not use this option unless you know that there is problem with execution times or memory consumption, and ensure that the option improves matters.
The Erlang runtime system uses a generational garbage collection scheme, using an "old heap" for data that has survived at least one garbage collection. When there is no more room on the old heap, a fullsweep garbage collection is done.

Option \texttt{fullsweep\_after} makes it possible to specify the maximum number of generational collections before forcing a fullsweep, even if there is room on the old heap. Setting the number to zero disables the general collection algorithm, that is, all live data is copied at every garbage collection.

A few cases when it can be useful to change \texttt{fullsweep\_after}:

- If binaries that are no longer used are to be thrown away as soon as possible. (Set \texttt{Number} to zero.)
- A process that mostly have short-lived data is fullsweepeped seldom or never, that is, the old heap contains mostly garbage. To ensure a fullsweep occasionally, set \texttt{Number} to a suitable value, such as 10 or 20.
- In embedded systems with a limited amount of RAM and no virtual memory, you might want to preserve memory by setting \texttt{Number} to zero. (The value can be set globally, see \texttt{erlang:system\_flag/2}.)

\texttt{\{min\_heap\_size, Size\}}

Useful only for performance tuning. Do not use this option unless you know that there is problem with execution times or memory consumption, and ensure that the option improves matters.

Gives a minimum heap size, in words. Setting this value higher than the system default can speed up some processes because less garbage collection is done. However, setting a too high value can waste memory and slow down the system because of worse data locality. Therefore, use this option only for fine-tuning an application and to measure the execution time with various \texttt{Size} values.

\texttt{\{min\_bin\_vheap\_size, VSize\}}

Useful only for performance tuning. Do not use this option unless you know that there is problem with execution times or memory consumption, and ensure that the option improves matters.

Gives a minimum binary virtual heap size, in words. Setting this value higher than the system default can speed up some processes because less garbage collection is done. However, setting a too high value can waste memory. Therefore, use this option only for fine-tuning an application and to measure the execution time with various \texttt{VSize} values.

\texttt{\{max\_heap\_size, Size\}}

Sets the \texttt{max\_heap\_size} process flag. The default \texttt{max\_heap\_size} is determined by command-line argument +hmax in \texttt{erl(1)}. For more information, see the documentation of \texttt{process\_flag(max\_heap\_size, Size)}.

\texttt{\{message\_queue\_data, MQD\}}

Sets the value of the \texttt{message\_queue\_data} process flag. \texttt{MQD} can be either \texttt{off\_heap} or \texttt{on\_heap}. The default value of the \texttt{message\_queue\_data} process flag is determined by the command-line argument +hmqd in \texttt{erl(1)}. For more information, see the documentation of \texttt{process\_flag(message\_queue\_data, MQD)}.

\texttt{\{async\_dist, Enabled\}}

Since: OTP 25.3

Set the \texttt{async\_dist} process flag of the spawned process. This option will override the default value set by the command line argument +pad <boolean>.

\texttt{spawn\_opt(Node, Module, Function, Args, Options) -> pid() | \{pid(), reference()}\}}

Types:
Node = node()
Module = module()
Function = atom()
Args = [term()]
Options =
    [monitor | {monitor, [monitor_option()]} | link | OtherOption]
OtherOption = term()

Returns the process identifier (pid) of a new process started by the application of Module:Function to Args on Node. If Node does not exist, a useless pid is returned. Otherwise works like spawn_opt/4.

Valid options depends on what options are supported by the node identified by Node. A description of valid Options for the local node of current OTP version can be found in the documentation of spawn_opt/4.

spawn_request(Fun) -> ReqId
Types:
    Fun = function()
    ReqId = reference()

The same as the call spawn_request(node(),Fun,[]). That is, a spawn request on the local node with no options.

spawn_request(Fun, Options) -> ReqId
Types:
    Fun = function()
    Option =
        {reply_tag, ReplyTag} | {reply, Reply} | spawn_opt_option()
    ReplyTag = term()
    Reply = yes | no | error_only | success_only
    Options = [Option]
    ReqId = reference()

The same as the call spawn_request(node(),Fun,Options). That is, a spawn request on the local node.

spawn_request(Node, Fun) -> ReqId
Types:
    Node = node()
    Fun = function()
    ReqId = reference()

The same as the call spawn_request(Node,Fun,[]). That is, a spawn request with no options.

spawn_request(Node, Fun, Options) -> ReqId
Types:
    Node = node()
    Fun = function()
    Options = [Option]
    Option =
        monitor |
\{monitor, \{monitor_option()\}\} | link | \{reply_tag, ReplyTag\} | \{reply, Reply\} | OtherOption

ReplyTag = term()
Reply = yes | no | error_only | success_only
OtherOption = term()
ReqId = reference()

The same as spawn_request(Node, erlang, apply, \[Fun, []\], Options). That is, a spawn request using the fun Fun of arity zero as entry point.

This function will fail with a badarg exception if:
- Node is not an atom.
- Fun is not a fun of arity zero.
- Options is not a proper list of terms.

spawn_request(Module, Function, Args) -> ReqId
Types:
Module = module()
Function = atom()
Args = [term()]
ReqId = reference()

The same as the call spawn_request(node(), Module, Function, Args, []). That is, a spawn request on the local node with no options.

spawn_request(Node, Module, Function, Args) -> ReqId
Types:
Node = node()
Module = module()
Function = atom()
Args = [term()]
ReqId = reference()

The same as the call spawn_request(Node, Module, Function, Args, []). That is, a spawn request with no options.

spawn_request(Module, Function, Args, Options) -> ReqId
Types:
Module = module()
Function = atom()
Args = [term()]
Option =
{reply_tag, ReplyTag} | {reply, Reply} | spawn_opt_option()

ReplyTag = term()
Reply = yes | no | error_only | success_only
Options = [Option]
ReqId = reference()

The same as the call spawn_request(node(), Module, Function, Args, Options). That is, a spawn request on the local node.

spawn_request(Node, Module, Function, Args, Options) -> ReqId
Types:
  Node = node()
  Module = module()
  Function = atom()
  Args = [term()]
  Options = [Option]
  Option =
    monitor |
    {monitor, [monitor_option()]} |
    link |
    {reply_tag, ReplyTag} |
    {reply, Reply} |
    OtherOption
  ReplyTag = term()
  Reply = yes | no | error_only | success_only
  OtherOption = term()
  ReqId = reference()

Asynchronously send a spawn request. Returns a request identifier ReqId.

If the spawn operation succeeds, a new process is created on the node identified by Node. When a spawn operation succeeds, the caller will by default be sent a message on the form \{ReplyTag, ReqId, ok, Pid\} where Pid is the process identifier of the newly created process. Such a message is referred to as a success message below in the text. ReplyTag is by default the atom spawn_reply unless modified by the \{reply_tag, ReplyTag\} option. The new process is started by the application of Module:Function to Args.

The spawn operation fails either if creation of a new process failed or if the spawn operation was interrupted by a connection failure. When a spawn operation fails, the caller will by default be sent a message on the form \{ReplyTag, ReqId, error, Reason\} where Reason is the error reason. Such a message is referred to as an error message below in the text. Currently the following spawn error Reasons are defined, but other reasons can appear at any time without prior notice:

badopt

An invalid Option was passed as argument. Note that different runtime systems may support different options.

notsup

The node identified by Node does not support spawn operations issued by spawn_request().

noconnection

Failure to set up a connection to the node identified by Node or the connection to that node was lost during the spawn operation. In the case the connection was lost, a process may or may not have been created.
system_limit

Could not create a new process due to that some system limit was reached. Typically the process table was full.

Valid Options:

monitor

In the absence of spawn operation failures, atomically sets up a monitor to the newly created process. That is, as if the calling process had called \texttt{monitor(process, Pid)} where \texttt{Pid} is the process identifier of the newly created process. The \texttt{ReqId} returned by \texttt{spawn_request()} is also used as monitor reference as if it was returned from \texttt{monitor(process, Pid)}.

The monitor will not be activated for the calling process until the spawn operation has succeeded. The monitor can not be demonitored before the operation has succeeded. A 'DOWN' message for the corresponding monitor is guaranteed not to be delivered before a \textit{success message} that corresponds to the spawn operation. If the spawn operation fails, no 'DOWN' message will be delivered.

If the connection between the nodes involved in the spawn operation is lost during the spawn operation, the spawn operation will fail with an error reason of \textit{noconnection}. A new process may or may not have been created.

\{monitor, MonitorOpts\}

In the absence of spawn operation failures, atomically sets up a monitor to the newly created process. That is, as if the calling process had called \texttt{monitor(process, Pid, MonitorOpts)} where \texttt{Pid} is the process identifier of the newly created process. See the \texttt{monitor} option above for more information.

Note that the monitor will not be activated for the calling process until the spawn operation has succeeded. For example, in the case that an alias is created using the monitor option, the alias will not be active until the monitor is activated.

link

In absence of spawn operation failures, atomically sets up a link between the calling process and the newly created process. That is, as if the calling process had called \texttt{link(Pid)} where \texttt{Pid} is the process identifier of the newly created process.

The link will not be activated for the calling process until the spawn operation has succeeded. The link can not be removed before the operation has succeeded. An exit signal due to the link is guaranteed not to be delivered before a \textit{success message} that corresponds to the spawn operation. If the spawn operation fails, no exit signal due to the link will be delivered to the caller of \texttt{spawn_request()}.

If the connection between the nodes involved in the spawn operation is lost during the spawn operation, the spawn operation will fail with an error reason of \textit{noconnection}. A new process may or may not have been created. If it has been created, it will be delivered an exit signal with an exit reason of \textit{noconnection}.

\{reply, Reply\}

Valid Reply values:

\texttt{yes}\n
A spawn reply message will be sent to the caller regardless of whether the operation succeeds or not. If the call to \texttt{spawn_request()} returns without raising an exception and the reply option is set to \texttt{yes}, the caller is guaranteed to be delivered either a \textit{success message} or an \textit{error message}. The reply option is by default set to \texttt{yes}.

\texttt{no}\n
No spawn reply message will be sent to the caller when the spawn operation completes. This regardless of whether the operation succeeds or not.
error_only

No spawn reply message will be sent to the caller if the spawn operation succeeds, but an error message will be sent to the caller if the operation fails.

success_only

No spawn reply message will be sent to the caller if the spawn operation fails, but a success message will be sent to the caller if the operation succeeds.

\[\{\text{reply\_tag}, \text{ReplyTag}\}\]

Sets the reply tag to ReplyTag in the reply message. That is, in the success or error message that is sent to the caller due to the spawn operation. The default reply tag is the atom spawn_reply.

OtherOption

Other valid options depends on what options are supported by the node identified by Node. A description of other valid Options for the local node of current OTP version can be found in the documentation of spawn_opt/4.

If a spawn reply message is delivered, it is guaranteed to be delivered before any other signals from the newly spawned process are delivered to the process issuing the spawn request.

This function will fail with a badarg exception if:

- Node is not an atom.
- Module is not an atom.
- Function is not an atom.
- Args is not a proper list of terms.
- Options is not a proper list of terms.

Note that not all individual Options are checked when the spawn request is sent. Some Options can only be checked on reception of the request. Therefore an invalid option does not cause a badarg exception, but will cause the spawn operation to fail with an error reason of badopt.

A spawn request can be abandoned by calling spawn_request_abandon/1.

Note:

For some important information about distributed signals, see the Blocking Signaling Over Distribution section in the Processes chapter of the Erlang Reference Manual.

\[
\text{spawn\_request\_abandon}(\text{ReqId :: reference()} \rightarrow \text{boolean()})
\]

Abandon a previously issued spawn request. ReqId corresponds to a request identifier previously returned by spawn_request() in a call from current process. That is, only the process that has made the request can abandon the request.

A spawn request can only be successfully abandoned until the spawn request has completed. When a spawn request has been successfully abandoned, the caller will not be affected by future direct effects of the spawn request itself. For example, it will not receive a spawn reply message. The request is however not withdrawn, so a new process may or may not be created due to the request. If a new process is created after the spawn request was abandoned, no monitors nor links will be set up to the caller of spawn_request_abandon/1 due to the spawn request. If the spawn request included the link option, the process created due to this request will be sent an exit signal from its parent with the exit reason abandoned when it is detected that the spawn operation has succeeded.
Note:
A process created due to a spawn request that has been abandoned may communicate with its parent as any other process. It is only the direct effects on the parent of the actual spawn request, that will be canceled by abandoning a spawn request.

Return values:
true
The spawn request was successfully abandoned.
false
No spawn request was abandoned. The ReqId request identifier did not correspond to an outstanding spawn request issued by the calling process. The reason for this is either:
• ReqId corresponds to a spawn request previously made by the calling process. The spawn operation has completed and a spawn reply has already been delivered to the calling process unless the spawn reply was disabled in the request.
• ReqId does not correspond to a spawn request that has been made by the calling process.

This function fail with a badarg exception if ReqId is not a reference.

split_binary(Bin, Pos) -> {binary(), binary()}
Types:
Bin = binary()
Pos = integer() >= 0
0..byte_size(Bin)

Returns a tuple containing the binaries that are the result of splitting Bin into two parts at position Pos. This is not a destructive operation. After the operation, there are three binaries altogether. Example:

```
> B = list_to_binary("0123456789").
<<"0123456789">>
> byte_size(B).
10
> {B1, B2} = split_binary(B,3).
{<<"012">>,<<"3456789">>}
> byte_size(B1).
3
> byte_size(B2).
7
```

erlang:start_timer(Time, Dest, Msg) -> TimerRef
Types:
Time = integer() >= 0
Dest = pid() | atom()
Msg = term()
TimerRef = reference()

Starts a timer. The same as calling erlang:start_timer(Time, Dest, Msg, []).
erlang:start_timer(Time, Dest, Msg, Options) -> TimerRef

Types:
  Time = integer()
  Dest = pid() | atom()
  Msg = term()
  Options = [Option]
  Abs = boolean()
  Option = {abs, Abs}
  TimerRef = reference()

Starts a timer. When the timer expires, the message {timeout, TimerRef, Msg} is sent to the process identified by Dest.

Options:
  {abs, false}
  This is the default. It means the Time value is interpreted as a time in milliseconds relative current Erlang monotonic time.

  {abs, true}
  Absolute Time value. The Time value is interpreted as an absolute Erlang monotonic time in milliseconds.

More Options can be added in the future.

The absolute point in time, the timer is set to expire on, must be in the interval [ \( \text{erlang:convert_time_unit}(\text{erlang:system_info(start_time)}, \text{native}, \text{millisecond}), \text{erlang:convert_time_unit}(\text{erlang:system_info(end_time)}, \text{native}, \text{millisecond}) \) ]. If a relative time is specified, the Time value is not allowed to be negative.

If Dest is a pid(), it must be a pid() of a process created on the current runtime system instance. This process has either terminated or not. If Dest is an atom(), it is interpreted as the name of a locally registered process. The process referred to by the name is looked up at the time of timer expiration. No error is returned if the name does not refer to a process.

If Dest is a pid(), the timer is automatically canceled if the process referred to by the pid() is not alive, or if the process exits. This feature was introduced in ERTS 5.4.11. Notice that timers are not automatically canceled when Dest is an atom().

See also erlang:send_after/4, erlang:cancel_timer/2, and erlang:read_timer/2.

Failure: badarg if the arguments do not satisfy the requirements specified here.

statistics(Item :: active_tasks) -> [ActiveTasks]

Types:
  ActiveTasks = integer() >= 0

Returns the same as statistics(active_tasks_all) with the exception that no information about the dirty IO run queue and its associated schedulers is part of the result. That is, only tasks that are expected to be CPU bound are part of the result.

statistics(Item :: active_tasks_all) -> [ActiveTasks]

Types:
ActiveTasks = integer() \(\geq 0\)

Returns a list where each element represents the amount of active processes and ports on each run queue and its associated schedulers. That is, the number of processes and ports that are ready to run, or are currently running. Values for normal run queues and their associated schedulers are located first in the resulting list. The first element corresponds to scheduler number 1 and so on. If support for dirty schedulers exist, an element with the value for the dirty CPU run queue and its associated dirty CPU schedulers follow and then as last element the value for the dirty IO run queue and its associated dirty IO schedulers follow. The information is not gathered atomically. That is, the result is not necessarily a consistent snapshot of the state, but instead quite efficiently gathered.

**Note:**

Each normal scheduler has one run queue that it manages. If dirty schedulers are supported, all dirty CPU schedulers share one run queue, and all dirty IO schedulers share one run queue. That is, we have multiple normal run queues, one dirty CPU run queue and one dirty IO run queue. Work can not migrate between the different types of run queues. Only work in normal run queues can migrate to other normal run queues. This has to be taken into account when evaluating the result.

See also statistics(total_active_tasks), statistics(run_queue_lengths), statistics(run_queue_lengths_all), statistics(total_run_queue_lengths), and statistics(total_run_queue_lengths_all).

```erlang
statistics(Item :: context_switches) -> {ContextSwitches, 0}
Types:
   ContextSwitches = integer() \(\geq 0\)
Returns the total number of context switches since the system started.

statistics(Item :: exact_reductions) ->
   {Total_Exact_Reductions,
    Exact_Reductions_Since_Last_Call}
Types:
   Total_Exact_Reductions = Exact_Reductions_Since_Last_Call = integer() \(\geq 0\)
Returns the number of exact reductions.

**Note:**

statistics(exact_reductions) is a more expensive operation than statistics(reductions).

statistics(Item :: garbage_collection) ->
   {Number_of_GCs, Words_Reclaimed, 0}
Types:
   Number_of_GCs = Words_Reclaimed = integer() \(\geq 0\)
Returns information about garbage collection, for example:

```erlang
> statistics(garbage_collection).
{85,23961,0}
```

This information can be invalid for some implementations.
erlang

statistics(Item :: io) -> {{input, Input}, {output, Output}}
Types:
  Input = Output = integer() >= 0
Returns Input, which is the total number of bytes received through ports, and Output, which is the total number of bytes output to ports.

statistics(Item :: microstate_accounting) -> [MSAcc_Thread] | undefined
Types:
  MSAcc_Thread =
    #{type := MSAcc_Thread_Type,
       id := MSAcc_Thread_Id,
       counters := MSAcc_Counters}
  MSAcc_Thread_Type =
    async | aux | dirty_io_scheduler | dirty_cpu_scheduler |
    poll | scheduler
  MSAcc_Thread_Id = integer() >= 0
  MSAcc_Counters = #{MSAcc_Thread_State => integer() >= 0}
  MSAcc_Thread_State =
    alloc | aux | bif | busy_wait | check_io | emulator | ets |
    gc | gc_fullsweep | nif | other | port | send | sleep | timers

Microstate accounting can be used to measure how much time the Erlang runtime system spends doing various tasks. It is designed to be as lightweight as possible, but some overhead exists when this is enabled. Microstate accounting is meant to be a profiling tool to help finding performance bottlenecks. To start/stop/reset microstate accounting, use system flag microstate_accounting.

statistics(microstate_accounting) returns a list of maps representing some of the OS threads within ERTS. Each map contains type and id fields that can be used to identify what thread it is, and also a counters field that contains data about how much time has been spent in the various states.

Example:

> erlang:statistics(microstate_accounting).
[#{counters => #{aux => 1899182914,
                check_io => 2605863602,
                emulator => 45731880463,
                gc => 1512206910,
                other => 5421338456,
                port => 221631,
                sleep => 5150294100},
   id => 1,
   type => scheduler}...]

The time unit is the same as returned by os:perf_counter/0. So, to convert it to milliseconds, you can do something like this:
erlang

```
lists:map(
    fun(#{ counters := Cnt } = M) ->
        MsCnt = maps:map(fun(_K, PerfCount) ->
            erlang:convert_time_unit(PerfCount, perf_counter, 1000)
        end, Cnt),
        M#{ counters := MsCnt }
    end, erlang:statistics(microstate_accounting)).
```

Notice that these values are not guaranteed to be the exact time spent in each state. This is because of various optimisation done to keep the overhead as small as possible.

**MSAcc_Thread_Types:**

- **scheduler**
  - The main execution threads that do most of the work. See erl +S for more details.
- **dirty_cpu_scheduler**
  - The threads for long running cpu intensive work. See erl +SDcpu for more details.
- **dirty_io_scheduler**
  - The threads for long running I/O work. See erl +SDio for more details.
- **async**
  - Async threads are used by various linked-in drivers (mainly the file drivers) do offload non-CPU intensive work. See erl +A for more details.
- **aux**
  - Takes care of any work that is not specifically assigned to a scheduler.
- **poll**
  - Does the IO polling for the emulator. See erl +IOt for more details.

The following **MSAcc_Thread_States** are available. All states are exclusive, meaning that a thread cannot be in two states at once. So, if you add the numbers of all counters in a thread, you get the total runtime for that thread.

- **aux**
  - Time spent handling auxiliary jobs.
- **check_io**
  - Time spent checking for new I/O events.
- **emulator**
  - Time spent executing Erlang processes.
- **gc**
  - Time spent doing garbage collection. When extra states are enabled this is the time spent doing non-fullsweep garbage collections.
- **other**
  - Time spent doing unaccounted things.
- **port**
  - Time spent executing ports.
- **sleep**
  - Time spent sleeping.

More fine-grained **MSAcc_Thread_States** can be added through configure (such as ./configure --with-microstate-accounting=extra). Enabling these states causes performance degradation when microstate accounting is turned off and increases the overhead when it is turned on.

- **alloc**
  - Time spent managing memory. Without extra states this time is spread out over all other states.
- **bif**
  - Time spent in BIFs. Without extra states this time is part of the emulator state.
busy_wait
Time spent busy waiting. This is also the state where a scheduler no longer reports that it is active when using
statistics(scheduler_wall_time). So, if you add all other states but this and sleep, and then
divide that by all time in the thread, you should get something very similar to the scheduler_wall_time
fraction. Without extra states this time is part of the other state.

ets
Time spent executing ETS BIFs. Without extra states this time is part of the emulator state.
gc_full
Time spent doing fullsweep garbage collection. Without extra states this time is part of the gc state.
nif
Time spent in NIFs. Without extra states this time is part of the emulator state.
send
Time spent sending messages (processes only). Without extra states this time is part of the emulator state.
timers
Time spent managing timers. Without extra states this time is part of the other state.

The utility module msacc(3) can be used to more easily analyse these statistics.
Returns undefined if system flag microstate_accounting is turned off.
The list of thread information is unsorted and can appear in different order between calls.

Note:
The threads and states are subject to change without any prior notice.

statistics(Item :: reductions) ->
  {Total_Reductions, Reductions_Since_Last_Call}
Types:
  Total_Reductions = Reductions_Since_Last_Call = integer() >= 0
Returns information about reductions, for example:

> statistics(reductions).
\{2046,11\}

Change:
As from ERTS 5.5 (Erlang/OTP R11B), this value does not include reductions performed in current time slices of
currently scheduled processes. If an exact value is wanted, use statistics(exact_reductions).

statistics(Item :: run_queue) -> integer() >= 0
Returns the total length of all normal and dirty CPU run queues. That is, queued work that is expected to be CPU
bound. The information is gathered atomically. That is, the result is a consistent snapshot of the state, but this operation
is much more expensive compared to statistics(total_run_queue_lengths), especially when a large
amount of schedulers is used.

statistics(Item :: run_queue_lengths) -> [RunQueueLength]
Types:
RunQueueLength = integer() >= 0

Returns the same as statistics(run_queue_lengths_all) with the exception that no information about the dirty IO run queue is part of the result. That is, only run queues with work that is expected to be CPU bound is part of the result.

statistics(Item :: run_queue_lengths_all) -&gt; [RunQueueLength]

Types:
  RunQueueLength = integer() >= 0

Returns a list where each element represents the amount of processes and ports ready to run for each run queue. Values for normal run queues are located first in the resulting list. The first element corresponds to the normal run queue of scheduler number 1 and so on. If support for dirty schedulers exist, values for the dirty CPU run queue and the dirty IO run queue follow (in that order) at the end. The information is not gathered atomically. That is, the result is not necessarily a consistent snapshot of the state, but instead quite efficiently gathered.

Note:

Each normal scheduler has one run queue that it manages. If dirty schedulers are supported, all dirty CPU schedulers share one run queue, and all dirty IO schedulers share one run queue. That is, we have multiple normal run queues, one dirty CPU run queue and one dirty IO run queue. Work can not migrate between the different types of run queues. Only work in normal run queues can migrate to other normal run queues. This has to be taken into account when evaluating the result.

See also statistics(run_queue_lengths), statistics(total_run_queue_lengths_all), statistics(total_run_queue_lengths), statistics(active_tasks), statistics(active_tasks_all), and statistics(total_active_tasks), statistics(total_active_tasks_all).

statistics(Item :: runtime) -&gt; {Total_Run_Time, Time_Since_Last_Call}

Types:
  Total_Run_Time = Time_Since_Last_Call = integer() >= 0

Returns information about runtime, in milliseconds. This is the sum of the runtime for all threads in the Erlang runtime system and can therefore be greater than the wall clock time.

Warning:

This value might wrap due to limitations in the underlying functionality provided by the operating system that is used.

Example:

> statistics(runtime).
{1690,1620}

statistics(Item :: scheduler_wall_time) -&gt; [{SchedulerId, ActiveTime, TotalTime}] | undefined

Types:

280 | Ericsson AB. All Rights Reserved.: Erlang Run-Time System Application (ERTS)
SchedulerId = integer() >= 1  
ActiveTime = TotalTime = integer() >= 0

Returns information describing how much time normal and dirty CPU schedulers in the system have been busy. This value is normally a better indicator of how much load an Erlang node is under instead of looking at the CPU utilization provided by tools such as top or sysstat. This is because scheduler_wall_time also includes time where the scheduler is waiting for some other resource (such as an internal mutex) to be available but does not use the CPU.

In order to better understand what a scheduler is busy doing you can use microstate accounting.

The definition of a busy scheduler is when it is not idle and not busy waiting for new work, that is:

- Executing process code
- Executing linked-in driver or NIF code
- Executing BIFs, or any other runtime handling
- Garbage collecting
- Handling any other memory management

Notice that a scheduler can also be busy even if the OS has scheduled out the scheduler thread.

**Note:**

It is recommended to use the module scheduler instead of this function directly as it provides an easier way to get the information that you usually want.

If enabled this function returns a list of tuples with {SchedulerId, ActiveTime, TotalTime}, where SchedulerId is an integer ID of the scheduler, ActiveTime is the duration the scheduler has been busy, and TotalTime is the total time duration since scheduler_wall_time activation for the specific scheduler. The time unit returned is undefined and can be subject to change between releases, OSs, and system restarts.

scheduler_wall_time is only to be used to calculate relative values for scheduler utilization. The ActiveTime can never exceed TotalTime. The list of scheduler information is unsorted and can appear in different order between calls.

The disabled this function returns undefined.

The activation time can differ significantly between schedulers. Currently dirty schedulers are activated at system start while normal schedulers are activated some time after the scheduler_wall_time functionality is enabled.

Only information about schedulers that are expected to handle CPU bound work is included in the return values from this function. If you also want information about dirty I/O schedulers, use statistics(scheduler_wall_time_all) instead.

Normal schedulers will have scheduler identifiers in the range 1 <= SchedulerId =< erlang:system_info(schedulers). Dirty CPU schedulers will have scheduler identifiers in the range erlang:system_info(schedulers) < SchedulerId =< erlang:system_info(schedulers) + erlang:system_info(dirty_cpu_schedulers).

**Note:**

The different types of schedulers handle specific types of jobs. Every job is assigned to a specific scheduler type. Jobs can migrate between different schedulers of the same type, but never between schedulers of different types. This fact has to be taken under consideration when evaluating the result returned.

You can use scheduler_wall_time to calculate scheduler utilization. First you take a sample of the values returned by erlang:statistics(scheduler_wall_time).
erlang

```erlang
> erlang:system_flag(scheduler_wall_time, true).
false
> Ts0 = lists:sort(erlang:statistics(scheduler_wall_time)), ok.
ok

Some time later the user takes another snapshot and calculates scheduler utilization per scheduler, for example:

```erlang
> Ts1 = lists:sort(erlang:statistics(scheduler_wall_time)), ok.
ok
> lists:map(fun({{I, A0, T0}, {I, A1, T1}}) ->
    {I, (A1 - A0)/(T1 - T0)} end, lists:zip(Ts0, Ts1)).
[{1,0.9743474730177548},
 {2,0.9744843782751444},
 {3,0.9995902361669045},
 {4,0.9738012596572161},
 {5,0.971795667018103},
 {6,0.973923584620741},
 {7,0.973237833077876},
 {8,0.974297293248656}]
```

Using the same snapshots to calculate a total scheduler utilization:

```erlang
> {A, T} = lists:foldl(fun({{_, A0, T0}, {_, A1, T1}}) ->
    {A1 + (A1 - A0), T1 + (T1 - T0)} end, {0, 0}, lists:zip(Ts0, Ts1)),

TotalSchedulerUtilization = A/T.
0.9769136803764825
```

Total scheduler utilization will equal 1.0 when all schedulers have been active all the time between the two measurements.

Another (probably more) useful value is to calculate total scheduler utilization weighted against maximum amount of available CPU time:

```erlang
> WeightedSchedulerUtilization = (TotalSchedulerUtilization * (erlang:system_info(schedulers) + erlang:system_info(dirty_cpu_schedulers))) / erlang:system_info(logical_processors_available).
0.9769136803764825
```

This weighted scheduler utilization will reach 1.0 when schedulers are active the same amount of time as maximum available CPU time. If more schedulers exist than available logical processors, this value may be greater than 1.0.

As of ERTS version 9.0, the Erlang runtime system will as default have more schedulers than logical processors. This due to the dirty schedulers.

**Note:**

scheduler_wall_time is by default disabled. To enable it, use
erlang:system_flag(scheduler_wall_time, true).

```erlang
statistics(Item :: scheduler_wall_time_all) ->
    [{SchedulerId, ActiveTime, TotalTime}] | undefined
```

Types:
SchedulerId = integer() >= 1
ActiveTime = TotalTime = integer() >= 0

The same as statistics(scheduler_wall_time), except that it also include information about all dirty I/O schedulers.

Dirty IO schedulers will have scheduler identifiers in the range erlang:system_info(schedulers) + erlang:system_info(dirty_cpu_schedulers) < SchedulerId =< erlang:system_info(schedulers) + erlang:system_info(dirty_cpu_schedulers) + erlang:system_info(dirty_io_schedulers).

Note:
Note that work executing on dirty I/O schedulers are expected to mainly wait for I/O. That is, when you get high scheduler utilization on dirty I/O schedulers, CPU utilization is not expected to be high due to this work.

statistics(Item :: total_active_tasks) -> ActiveTasks
Types:
   ActiveTasks = integer() >= 0
The same as calling lists:sum(statistics(active_tasks)), but more efficient.

statistics(Item :: total_active_tasks_all) -> ActiveTasks
Types:
   ActiveTasks = integer() >= 0
The same as calling lists:sum(statistics(active_tasks_all)), but more efficient.

statistics(Item :: total_run_queue_lengths) ->
   TotalRunQueueLengths
Types:
   TotalRunQueueLengths = integer() >= 0
The same as calling lists:sum(statistics(run_queue_lengths)), but more efficient.

statistics(Item :: total_run_queue_lengths_all) ->
   TotalRunQueueLengths
Types:
   TotalRunQueueLengths = integer() >= 0
The same as calling lists:sum(statistics(run_queue_lengths_all)), but more efficient.

statistics(Item :: wall_clock) ->
   {Total_Wallclock_Time,
    Wallclock_Time_Since_Last_Call}
Types:
   Total_Wallclock_Time = Wallclock_Time_Since_Last_Call = integer() >= 0
Returns information about wall clock. wall_clock can be used in the same manner as runtime, except that real time is measured as opposed to runtime or CPU time.
erlang:suspend_process(Suspendee) -> true
Types:
  Suspendee = pid()
Suspends the process identified by Suspendee. The same as calling
erlang:suspend_process(Suspendee, []).

Warning:
This BIF is intended for debugging only.

erlang:suspend_process(Suspendee, OptList) -> boolean()
Types:
  Suspendee = pid()
  OptList = [Opt]
  Opt = unless_suspending | asynchronous | {asynchronous, term()}
Increases the suspend count on the process identified by Suspendee and puts it in the suspended state if it is not already in that state. A suspended process is not scheduled for execution until the process has been resumed.
A process can be suspended by multiple processes and can be suspended multiple times by a single process. A suspended process does not leave the suspended state until its suspend count reaches zero. The suspend count of Suspendee is decreased when erlang:resume_process(Suspendee) is called by the same process that called erlang:suspend_process(Suspendee). All increased suspend counts on other processes acquired by a process are automatically decreased when the process terminates.
Options (Opts):
asynchronous
  A suspend request is sent to the process identified by Suspendee. Suspendee eventually suspends unless it is resumed before it could suspend. The caller of erlang:suspend_process/2 returns immediately, regardless of whether Suspendee has suspended yet or not. The point in time when Suspendee suspends cannot be deduced from other events in the system. It is only guaranteed that Suspendee eventually suspends (unless it is resumed). If no asynchronous options has been passed, the caller of erlang:suspend_process/2 is blocked until Suspendee has suspended.
{asynchronous, ReplyTag}
  A suspend request is sent to the process identified by Suspendee. When the suspend request has been processed, a reply message is sent to the caller of this function. The reply is on the form {ReplyTag, State} where State is either:
exited
  Suspendee has exited.
suspended
  Suspendee is now suspended.
not_suspended
  Suspendee is not suspended. This can only happen when the process that issued this request, have called resume_process(Suspendee) before getting the reply.
Apart from the reply message, the {asynchronous, ReplyTag} option behaves exactly the same as the asynchronous option without reply tag.
The process identified by Suspendee is suspended unless the calling process already is suspending Suspendee. If unless_suspending is combined with option asynchronous, a suspend request is sent unless the calling process already is suspending Suspendee or if a suspend request already has been sent and is in transit. If the calling process already is suspending Suspendee, or if combined with option synchronous and a send request already is in transit, false is returned and the suspend count on Suspendee remains unchanged.

If the suspend count on the process identified by Suspendee is increased, true is returned, otherwise false.

**Warning:**
This BIF is intended for debugging only.

**Warning:**
You can easily create deadlocks if processes suspends each other (directly or in circles). In ERTS versions prior to ERTS version 10.0, the runtime system prevented such deadlocks, but this prevention has now been removed due to performance reasons.

Failures:
badarg
  If Suspendee is not a process identifier.
badarg
  If the process identified by Suspendee is the same process as the process calling erlang:suspend_process/2.
badarg
  If the process identified by Suspendee is not alive.
badarg
  If the process identified by Suspendee resides on another node.
badarg
  If OptList is not a proper list of valid Opt.
system_limit
  If the process identified by Suspendee has been suspended more times by the calling process than can be represented by the currently used internal data structures. The system limit is greater than 2,000,000,000 suspends and will never be lower.

erlang:system_flag(Flag :: backtrace_depth, Depth) -> OldDepth
Types:
  Depth = OldDepth = integer() >= 0
Sets the maximum depth of call stack back-traces in the exit reason element of 'EXIT' tuples. The flag also limits the stacktrace depth returned by process_info item current_stacktrace.
Returns the old value of the flag.

erlang:system_flag(Flag :: cpu_topology, CpuTopology) -> OldCpuTopology
Types:
CpuTopology = OldCpuTopology = cpu_topology()
cpu_topology() = [LevelEntry :: level_entry()] | undefined
level_entry() = {LevelTag :: level_tag(), SubLevel :: sub_level()}
    | {LevelTag :: level_tag(),
      InfoList :: info_list(),
      SubLevel :: sub_level()}
level_tag() = core | node | processor | thread
sub_level() = [LevelEntry :: level_entry()] | (LogicalCpuId :: {logical, integer() >= 0})
info_list() = []

**Warning:**

**This argument is deprecated.** Instead of using this argument, use command-line argument +sct in erl(1). When this argument is removed, a final CPU topology to use is determined at emulator boot time.

Sets the user-defined CpuTopology. The user-defined CPU topology overrides any automatically detected CPU topology. By passing undefined as CpuTopology, the system reverts to the CPU topology automatically detected. The returned value equals the value returned from erlang:system_info(cpu_topology) before the change was made.

Returns the old value of the flag.

The CPU topology is used when binding schedulers to logical processors. If schedulers are already bound when the CPU topology is changed, the schedulers are sent a request to rebind according to the new CPU topology.

The user-defined CPU topology can also be set by passing command-line argument +sct to erl(1).

For information on type CpuTopology and more, see erlang:system_info(cpu_topology) as well as command-line flags +sct and +sbt in erl(1).

erlang:system_flag(Flag :: dirty_cpuSchedulers_online, DirtyCPUSchedulersOnline) ->
    OldDirtyCPUSchedulersOnline
Types:
    DirtyCPUSchedulersOnline = OldDirtyCPUSchedulersOnline = integer() >= 1

Sets the number of dirty CPU schedulers online. Range is 1 <= DirtyCPUSchedulersOnline <= N, where N is the smallest of the return values of erlang:system_info(dirty_cpu_schedulers) and erlang:system_info(schedulers_online).

Returns the old value of the flag.

The number of dirty CPU schedulers online can change if the number of schedulers online changes. For example, if 12 schedulers and 6 dirty CPU schedulers are online, and system_flag/2 is used to set the number of schedulers online to 6, then the number of dirty CPU schedulers online is automatically decreased by half as well, down to 3. Similarly, the number of dirty CPU schedulers online increases proportionally to increases in the number of schedulers online.

For more information, see erlang:system_info(dirty_cpu_schedulers) and erlang:system_info(dirty_cpu_schedulers_online).

erlang:system_flag(Flag :: erts_alloc, Value :: {Alloc, F, V}) ->
ok | notsup

Types:

Alloc = F = atom()
V = integer()

Sets system flags for erts_alloc(3). Alloc is the allocator to affect, for example binary_alloc. F is the flag to change and V is the new value.

Only a subset of all erts_alloc flags can be changed at run time. This subset is currently only the flag sbct.

Returns ok if the flag was set or notsup if not supported by erts_alloc.

erlang:system_flag(Flag :: fullsweep_after, Number) -> OldNumber

Types:

Number = OldNumber = integer() >= 0

Sets system flag fullsweep_after. Number is a non-negative integer indicating how many times generational garbage collections can be done without forcing a fullsweep collection. The value applies to new processes, while processes already running are not affected.

Returns the old value of the flag.

In low-memory systems (especially without virtual memory), setting the value to 0 can help to conserve memory.

This value can also be set through (OS) environment variable ERL_FULLSWEEP_AFTER.

erlang:system_flag(Flag :: microstate_accounting, Action) -> OldState

Types:

Action = true | false | reset
OldState = true | false

Turns on/off microstate accounting measurements. When passing reset, all counters are reset to 0.

For more information see statistics(microstate_accounting).

erlang:system_flag(Flag :: min_heap_size, MinHeapSize) -> OldMinHeapSize

Types:

MinHeapSize = OldMinHeapSize = integer() >= 0

Sets the default minimum heap size for processes. The size is specified in words. The new min_heap_size effects only processes spawned after the change of min_heap_size has been made. min_heap_size can be set for individual processes by using spawn_opt/4 or process_flag/2.

Returns the old value of the flag.

erlang:system_flag(Flag :: min_bin_vheap_size, MinBinVHeapSize) -> OldMinBinVHeapSize

Types:

MinBinVHeapSize = OldMinBinVHeapSize = integer() >= 0

Sets the default minimum binary virtual heap size for processes. The size is specified in words. The new min_bin_vheap_size effects only processes spawned after the change of min_bin_vheap_size has
been made. \texttt{min_bin_vheap_size} can be set for individual processes by using \texttt{spawn_opt/2,3,4} or \texttt{process_flag/2}.

Returns the old value of the flag.

\begin{verbatim}
erlang:system_flag(Flag :: max_heap_size, MaxHeapSize) -> OldMaxHeapSize

Types:
MaxHeapSize = OldMaxHeapSize = max_heap_size()
max_heap_size() =
    integer() >= 0 |
    #{size => integer() >= 0,
        kill => boolean(),
        error_logger => boolean(),
        include_shared_binaries => boolean()}

Sets the default maximum heap size settings for processes. The size is specified in words. The new \texttt{max_heap_size} effects only processes spawned after the change has been made. \texttt{max_heap_size} can be set for individual processes using \texttt{spawn_opt/2,3,4} or \texttt{process_flag/2}.

Returns the old value of the flag.

For details on how the heap grows, see \textit{Sizing the heap in the ERTS internal documentation}.

\end{verbatim}

\begin{verbatim}
erlang:system_flag(Flag :: multi_scheduling, BlockState) -> OldBlockState

Types:
BlockState = block | unblock | block_normal | unblock_normal
OldBlockState = blocked | disabled | enabled

If multi-scheduling is enabled, more than one scheduler thread is used by the emulator. Multi-scheduling can be blocked in two different ways. Either all schedulers but one is blocked, or all \texttt{normal} schedulers but one is blocked. When only normal schedulers are blocked, dirty schedulers are free to continue to schedule processes.

If \texttt{BlockState} = := \texttt{block}, multi-scheduling is blocked. That is, one and only one scheduler thread will execute. If \texttt{BlockState} = := \texttt{unblock} and no one else blocks multi-scheduling, and this process has blocked only once, multi-scheduling is unblocked.

If \texttt{BlockState} = := \texttt{block_normal}, normal multi-scheduling is blocked. That is, only one normal scheduler thread will execute, but multiple dirty schedulers can execute. If \texttt{BlockState} = := \texttt{unblock_normal} and no one else blocks normal multi-scheduling, and this process has blocked only once, normal multi-scheduling is unblocked.

One process can block multi-scheduling and normal multi-scheduling multiple times. If a process has blocked multiple times, it must unblock exactly as many times as it has blocked before it has released its multi-scheduling block. If a process that has blocked multi-scheduling or normal multi-scheduling exits, it automatically releases its blocking of multi-scheduling and normal multi-scheduling.

The return values are \texttt{disabled}, \texttt{blocked}, \texttt{blocked_normal}, or \texttt{enabled}. The returned value describes the state just after the call to \texttt{erlang:system_flag(multi_scheduling, BlockState)} has been made. For information about the return values, see \texttt{erlang:system_info(multi_scheduling)}.

\end{verbatim}
**Note:**

Blocking of multi-scheduling and normal multi-scheduling is normally not needed. If you feel that you need to use these features, consider it a few more times again. Blocking multi-scheduling is only to be used as a last resort, as it is most likely a very inefficient way to solve the problem.

### erlang:system_flag(Flag :: outstanding_system_requests_limit, NewLimit) -> OldLimit

**Types:**

\[
\text{NewLimit} = \text{OldLimit} = 1..134217727
\]

Sets a limit on the amount of outstanding requests made by a system process orchestrating system wide changes. Currently there are two such processes:

**The Code Purger**

The code purger orchestrates checking of references to old code before old code is removed from the system.

**The Literal Area Collector**

The literal area collector orchestrates copying of references from old literal areas before removal of such areas from the system.

Each of these processes are allowed to have as many outstanding requests as this limit is set to. By default this limit is set to twice the amount of schedulers on the system. This will ensure that schedulers will have enough work scheduled to perform these operations as quickly as possible at the same time as other work will be interleaved with this work. Currently used limit can be checked by calling `erlang:system_info(outstanding_system_requests_limit)`.

This limit can also be set by passing the command line argument `+zosrl <Limit>` to `erl`.

### erlang:system_flag(Flag :: scheduler_bind_type, How) -> OldBindType

**Types:**

\[
\text{How} = \text{scheduler_bind_type()} | \text{default_bind}
\]

\[
\text{OldBindType} = \text{scheduler_bind_type()}
\]

\[
\text{scheduler_bind_type()} =
\text{no_node_processor_spread} | \text{no_node_thread_spread} | \text{no_spread} | \text{processor_spread} | \text{spread} | \text{thread_spread} | \text{thread_no_node_processor_spread} | \text{unbound}
\]

**Warning:**

**This argument is deprecated.** Instead of using this argument, use command-line argument `+sbt` in `erl(1)`. When this argument is removed, a final scheduler bind type to use is determined at emulator boot time.

Controls if and how schedulers are bound to logical processors.
When `erlang:system_flag(scheduler_bind_type, How)` is called, an asynchronous signal is sent to all schedulers online, causing them to try to bind or unbind as requested.

**Note:**

If a scheduler fails to bind, this is often silently ignored, as it is not always possible to verify valid logical processor identifiers. If an error is reported, an error event is logged. To verify that the schedulers have bound as requested, call `erlang:system_info(scheduler_bindings)`.

Schedulers can be bound on newer Linux, Solaris, FreeBSD, and Windows systems, but more systems will be supported in future releases.

In order for the runtime system to be able to bind schedulers, the CPU topology must be known. If the runtime system fails to detect the CPU topology automatically, it can be defined. For more information on how to define the CPU topology, see command-line flag `+sct` in `erl(1)`.

The runtime system does by default not bind schedulers to logical processors.

**Note:**

If the Erlang runtime system is the only OS process binding threads to logical processors, this improves the performance of the runtime system. However, if other OS processes (for example, another Erlang runtime system) also bind threads to logical processors, there can be a performance penalty instead. Sometimes this performance penalty can be severe. If so, it is recommended to not bind the schedulers.

Schedulers can be bound in different ways. Argument `How` determines how schedulers are bound and can be any of the following:

- **unbound**
  - Same as command-line argument `+sbt u` in `erl(1)`.
- **no_spread**
  - Same as command-line argument `+sbt ns` in `erl(1)`.
- **thread_spread**
  - Same as command-line argument `+sbt ts` in `erl(1)`.
- **processor_spread**
  - Same as command-line argument `+sbt ps` in `erl(1)`.
- **spread**
  - Same as command-line argument `+sbt s` in `erl(1)`.
- **no_node_thread_spread**
  - Same as command-line argument `+sbt nnts` in `erl(1)`.
- **no_node_processor_spread**
  - Same as command-line argument `+sbt nnps` in `erl(1)`.
- **thread_no_node_processor_spread**
  - Same as command-line argument `+sbt tnnps` in `erl(1)`.
- **default_bind**
  - Same as command-line argument `+sbt db` in `erl(1)`.

The returned value equals `How` before flag `scheduler_bind_type` was changed.

**Failures:**

- **notsup**
  - If binding of schedulers is not supported.
- **badarg**
  - If `How` is not one of the documented alternatives.
badarg

If CPU topology information is unavailable.

The scheduler bind type can also be set by passing command-line argument +sbt to erl(1).

For more information, see erlang:system_info(scheduler_bind_type),
erlang:system_info(scheduler_bindings), as well as command-line flags +sbt and +sct in erl(1).

erlang:system_flag(Flag :: scheduler_wall_time, Boolean) ->
   OldBoolean

Types:
   Boolean = OldBoolean = boolean()

Try enable or disable scheduler wall time measurements by passing Boolean as either true or false.

For more information about how to use scheduler wall time measurements, see
statistics(scheduler_wall_time).

Scheduler wall time measurements has a node global state. It is either enabled for all processes on the node or
disabled for all processes. Each process has a logical counter initialized as zero. A call with Boolean as true will
increase that counter one step for the calling process. A call with false will decrease it one step unless it already
is zero. The node global state for scheduler_wall_time will be enabled as long as there is at least one process
alive with a counter value larger than zero. When a process terminates, its counter will also disappear. To ensure
scheduler_wall_time is kept enabled, the process that enabled it must therefore be kept alive.

Returns the old value of the node global state, true if scheduler wall time measurements were enabled, false if
it were disabled.

Scheduler wall time measurements do consume some cpu overhead and should not be left turned on unless used.

erlang:system_flag(Flag :: schedulers_online, SchedulersOnline) ->
   OldSchedulersOnline

Types:
   SchedulersOnline = OldSchedulersOnline = integer() >= 1

Sets the number of schedulers online. Range is 1 <= SchedulersOnline <=
erlang:system_info(schedulers).

Returns the old value of the flag.

If the emulator was built with support for dirty schedulers, changing the number of schedulers online can also change
the number of dirty CPU schedulers online. For example, if 12 schedulers and 6 dirty CPU schedulers are online, and
system_flag/2 is used to set the number of schedulers online to 6, then the number of dirty CPU schedulers online
is automatically decreased by half as well, down to 3. Similarly, the number of dirty CPU schedulers online increases
proportionally to increases in the number of schedulers online.

For more information, see erlang:system_info(schedulers) and
erlang:system_info(schedulers_online).

erlang:system_flag(Flag :: system_logger, Logger) -> PrevLogger

Types:
   Logger = PrevLogger = logger | undefined | pid()

Sets the process that will receive the logging messages generated by ERTS. If set to undefined, all logging messages
generated by ERTS will be dropped. The messages will be in the format:
erlang

{log,Level,Format,ArgList,Metadata} where
Level = atom(),
Format = string(),
ArgList = list(term()),
Metadata = #{ pid => pid(),
group_leader => pid(),
time := logger:timestamp(),
error_logger := #{ emulator := true, tag := atom() }

If the system_logger process dies, this flag will be reset to logger.
The default is the process named logger.
Returns the old value of the flag.

Note:
This function is designed to be used by the KERNEL logger. Be careful if you change it to something else as log messages may be lost. If you want to intercept emulator log messages, do it by adding a specialized handler to the KERNEL logger.

erlang:system_flag(Flag :: trace_control_word, TCW) -> OldTCW
Types:
TCW = OldTCW = integer() >= 0
Sets the value of the node trace control word to TCW, which is to be an unsigned integer. For more information, see function set_tcw in section "Match Specifications in Erlang" in the User's Guide.
Returns the old value of the flag.

erlang:system_flag(Flag :: time_offset, Value :: finalize) -> OldState
Types:
OldState = preliminary | final | volatile
Finalizes the time offset when single time warp mode is used. If another time warp mode is used, the time offset state is left unchanged.
Returns the old state identifier, that is:
• If preliminary is returned, finalization was performed and the time offset is now final.
• If final is returned, the time offset was already in the final state. This either because another erlang:system_flag(time_offset, finalize) call or because no time warp mode is used.
• If volatile is returned, the time offset cannot be finalized because multi-time warp mode is used.

erlang:system_info(Item :: overview) -> boolean()
Returns information about the current system. The documentation of this function is broken into the following sections in order to make it easier to navigate.
Memory Allocation
allocated_areas,allocator,alloc_util_allocators,allocator_sizes
CPU Topology
cpu_topology,logical_processors,cpu_quota,update_cpu_info
Process Information
fullsweep_after, garbage_collection, heap_sizes, heap_type, max_heap_size, message_queue_data, min_heap_size, min_bin_vheap_size, procs

System Limits
atom_count, atom_limit, ets_count, ets_limit, port_count, port_limit, process_count, process_limit

System Time
derr_follow_after, garbage_collection, message_queue_data, min_heap_size, min_bin_vheap_size, procs

Scheduler Information
dirty_cpuSchedulers, dirty_cpuSchedulersOnline, dirty_ioSchedulers, multi_scheduling, multi_schedulingBlockers, normal_multi_schedulingBlockers, scheduler_bind_type, scheduler_bindings, scheduler_id, schedulers, smp_support, threads, thread_pool_size

Distribution Information
async_dist.creation, delayed_node_table_gc, dist, dist_buf_busy_limit, dist_ctrl

System Information
c_compiler_used, check_io, compat_rel, debug_compiled, driver_version, dynamic_trace, dynamic_trace_probes, emu_flavor, emu_type, info, kernel_poll, loaded, machine, modified_timing_level, nif_version, otp_release, outstanding_system_requests_limit, port_parallelism, system_architecture, system_logger, system_version, trace_control_word, version, wordsize

erlang:system_info(Item :: allocated_areas) -> [tuple()]
erlang:system_info(Item :: allocator) ->
    {Allocator, Version, Features, Settings}
erlang:system_info(Item :: {allocator, Alloc}) -> [term()]
erlang:system_info(Item :: alloc_util_allocators) -> [Alloc]
erlang:system_info(Item :: {allocator_sizes, Alloc}) -> [term()]

Types:
  Allocator = undefined | glibc
  Version = [integer() >= 0]
  Features = [atom()]
  Settings =
    [{Subsystem :: atom(),
      [{Parameter :: atom(), Value :: term()}]}]
  Alloc = atom()

Returns various information about the memory allocators of the current system (emulator) as specified by Item:
allocated_areas

Returns a list of tuples with information about miscellaneous allocated memory areas.
Each tuple contains an atom describing the type of memory as first element and the amount of allocated memory in bytes as second element. When information about allocated and used memory is present, also a third element is present, containing the amount of used memory in bytes.
erlang:system_info(allocated_areas) is intended for debugging, and the content is highly implementation-dependent. The content of the results therefore changes when needed without prior notice.

Notice that the sum of these values is not the total amount of memory allocated by the emulator. Some values are part of other values, and some memory areas are not part of the result. For information about the total amount of memory allocated by the emulator, see erlang:memory/0,1.

allocator

Returns {Allocator, Version, Features, Settings, where:
- Allocator corresponds to the malloc() implementation used. If Allocator equals undefined, the malloc() implementation used cannot be identified. glibc can be identified.
- Version is a list of integers (but not a string) representing the version of the malloc() implementation used.
- Features is a list of atoms representing the allocation features used.
- Settings is a list of subsystems, their configurable parameters, and used values. Settings can differ between different combinations of platforms, allocators, and allocation features. Memory sizes are given in bytes.

See also “System Flags Effecting erts_alloc” in erts_alloc(3).

{allocator, Alloc}

Returns information about the specified allocator. As from ERTS 5.6.1, the return value is a list of {instance, InstanceNo, InstanceInfo} tuples, where InstanceInfo contains information about a specific instance of the allocator. If Alloc is not a recognized allocator, undefined is returned. If Alloc is disabled, false is returned.

Notice that the information returned is highly implementation-dependent and can be changed or removed at any time without prior notice. It was initially intended as a tool when developing new allocators, but as it can be of interest for others it has been briefly documented.

The recognized allocators are listed in erts_alloc(3). Information about super carriers can be obtained from ERTS 8.0 with {allocator, erts_mmap} or from ERTS 5.10.4; the returned list when calling with {allocator, mseg_alloc} also includes an {erts_mmap, _} tuple as one element in the list.

After reading the erts_alloc(3) documentation, the returned information more or less speaks for itself, but it can be worth explaining some things. Call counts are presented by two values, the first value is giga calls, and the second value is calls. mbcs and sbcs denote multi-block carriers, and single-block carriers, respectively. Sizes are presented in bytes. When a size is not presented, it is the amount of something. Sizes and amounts are often presented by three values:
- The first is the current value.
- The second is the maximum value since the last call to erlang:system_info({allocator, Alloc}).
- The third is the maximum value since the emulator was started.

If only one value is present, it is the current value. fix_alloc memory block types are presented by two values. The first value is the memory pool size and the second value is the used memory size.

alloc_util_allocators

Returns a list of the names of all allocators using the ERTS internal alloc_util framework as atoms. For more information, see section The alloc_util framework in erts_alloc(3).

{allocator_sizes, Alloc}

Returns various size information for the specified allocator. The information returned is a subset of the information returned by erlang:system_info({allocator, Alloc}).
erlang:system_info(Item :: cpu_topology) -> CpuTopology
erlang:system_info(Item ::
    {cpu_topology, defined | detected | used}) ->
    CpuTopology
erlang:system_info(Item :: cpu_quota) -> integer() >= 1 | unknown
erlang:system_info(Item ::
    logical_processors |
    logical_processors_available |
    logical_processors_online) ->
    unknown | integer() >= 1
erlang:system_info(Item :: update_cpu_info) -> changed | unchanged

Types:

cpu_topology() = [LevelEntry :: level_entry()] | undefined
All LevelEntries of a list must contain the same LevelTag, except on the top level where both node and processor LevelTags can coexist.

level_entry() =
    {LevelTag :: level_tag(), SubLevel :: sub_level()} |
    {LevelTag :: level_tag(),
        InfoList :: info_list(),
        SubLevel :: sub_level()}
(LevelTag, SubLevel) == (LevelTag, [], SubLevel)

level_tag() = core | node | processor | thread
More LevelTags can be introduced in a future release.

sub_level() =
    [LevelEntry :: level_entry()] |
    (LogicalCpuId :: {logical, integer() >= 0})

info_list() = []
The info_list() can be extended in a future release.

Returns various information about the CPU topology of the current system (emulator) as specified by Item:

cpu_topology
Returns the CpuTopology currently used by the emulator. The CPU topology is used when binding schedulers to logical processors. The CPU topology used is the user-defined CPU topology, if such exists, otherwise the automatically detected CPU topology, if such exists. If no CPU topology exists, undefined is returned.

node refers to Non-Uniform Memory Access (NUMA) nodes. thread refers to hardware threads (for example, Intel hyper-threads).
A level in term CpuTopology can be omitted if only one entry exists and InfoList is empty.

thread can only be a sublevel to core. core can be a sublevel to processor or node. processor can be on the top level or a sublevel to node. node can be on the top level or a sublevel to processor. That is, NUMA nodes can be processor internal or processor external. A CPU topology can consist of a mix of processor internal and external NUMA nodes, as long as each logical CPU belongs to one NUMA node. Cache hierarchy is not part of the CpuTopology type, but will be in a future release. Other things can also make it into the CPU topology in a future release. So, expect the CpuTopology type to change.

{cpu_topology, defined}
Returns the user-defined CpuTopology. For more information, see command-line flag +sct in erl(1) and argument cpu_topology.
erlang

{cpu_topology, detected}

Returns the automatically detected CpuTopology. The emulator detects the CPU topology on some newer Linux, Solaris, FreeBSD, and Windows systems. On Windows system with more than 32 logical processors, the CPU topology is not detected.

For more information, see argument cpu_topology.

{cpu_topology, used}

Returns CpuTopology used by the emulator. For more information, see argument cpu_topology.

logical_processors

Returns the detected number of logical processors configured in the system. The return value is either an integer, or the atom unknown if the emulator cannot detect the configured logical processors.

logical_processors_available

Returns the detected number of logical processors available to the Erlang runtime system. The return value is either an integer, or the atom unknown if the emulator cannot detect the available logical processors. The number of available logical processors is less than or equal to the number of logical processors online.

logical_processors_online

Returns the detected number of logical processors online on the system. The return value is either an integer, or the atom unknown if the emulator cannot detect logical processors online. The number of logical processors online is less than or equal to the number of logical processors configured.

cpu_quota

Returns the detected CPU quota the emulator is limited by. The return value is an integer saying how many processors' worth of runtime we get (between 1 and the number of logical processors), or the atom unknown if the emulator cannot detect a quota.

update_cpu_info

The runtime system rereads the CPU information available and updates its internally stored information about the detected CPU topology and the number of logical processors configured, online, available, and cpu quota.

If the CPU information has changed since the last time it was read, the atom changed is returned, otherwise the atom unchanged. If the CPU information has changed, you probably want to adjust the number of schedulers online. You typically want to have as many schedulers online as logical processors available.

erlang:system_info(Item :: fullsweep_after) ->
    {fullsweep_after, integer() >= 0}
erlang:system_info(Item :: garbage_collection) ->
    [[atom(), integer()]]
erlang:system_info(Item :: heap_sizes) -> [integer() >= 0]
erlang:system_info(Item :: heap_type) -> private
erlang:system_info(Item :: max_heap_size) ->
    {max_heap_size,
     MaxHeapSize :: max_heap_size()}
erlang:system_info(Item :: message_queue_data) ->
    message_queue_data()
erlang:system_info(Item :: min_heap_size) ->
    {min_heap_size,
     MinHeapSize :: integer() >= 1}
erlang:system_info(Item :: min_bin_vheap_size) ->
erlang

{min_bin_vheap_size, MinBinVHeapSize :: integer() >= 1}

erlang:system_info(Item :: procs) -> binary()

Types:
message_queue_data() = off_heap | on_heap
max_heap_size() =
    integer() >= 0 |
    #{size => integer() >= 0, kill => boolean(),
        error_logger => boolean(),
        include_shared_binaries => boolean()}

Returns information about the default process heap settings:

fullsweep_after
Returns {fullsweep_after, integer() >= 0}, which is the fullsweep_after garbage collection setting used by default. For more information, see garbage_collection described below.

garbage_collection
Returns a list describing the default garbage collection settings. A process spawned on the local node by a spawn or spawn_link uses these garbage collection settings. The default settings can be changed by using erlang:system_flag/2. spawn_opt/2,3,4 can spawn a process that does not use the default settings.

heap_sizes
Returns a list of integers representing valid heap sizes in words. All Erlang heaps are sized from sizes in this list.

heap_type
Returns the heap type used by the current emulator. One heap type exists:
private
    Each process has a heap reserved for its use and no references between heaps of different processes are allowed. Messages passed between processes are copied between heaps.

max_heap_size
Returns {max_heap_size, MaxHeapSize}, where MaxHeapSize is the current system-wide maximum heap size settings for spawned processes. This setting can be set using the command-line flags +hmax, +hmaxk, +hmaxel and +hmaxibl in erl(1). It can also be changed at runtime using erlang:system_flag(max_heap_size, MaxHeapSize). For more details about the max_heap_size process flag, see process_flag(max_heap_size, MaxHeapSize).

message_queue_data
Returns the default value of the message_queue_data process flag, which can be either off_heap or on_heap. The default value is set by the command-line argument +hmqd in erl(1). For more information, see the documentation of process_flag(message_queue_data, MQD).

min_heap_size
Returns {min_heap_size, MinHeapSize}, where MinHeapSize is the current system-wide minimum heap size for spawned processes.

min_bin_vheap_size
Returns {min_bin_vheap_size, MinBinVHeapSize}, where MinBinVHeapSize is the current system-wide minimum binary virtual heap size for spawned processes.
erlang:procs

Returns a binary containing a string of process and port information formatted as in Erlang crash dumps. For more information, see section How to interpret the Erlang crash dumps in the User's Guide.

erlang:system_info(Item :: atom_count) -> integer() >= 1
erlang:system_info(Item :: atom_limit) -> integer() >= 1
erlang:system_info(Item :: ets_count) -> integer() >= 1
erlang:system_info(Item :: ets_limit) -> integer() >= 1
erlang:system_info(Item :: port_parallelism) -> boolean()
erlang:system_info(Item :: port_count) -> integer() >= 0
erlang:system_info(Item :: port_limit) -> integer() >= 1
erlang:system_info(Item :: process_count) -> integer() >= 1

Returns information about the current system (emulator) limits as specified by Item:

atom_count

Returns the number of atoms currently existing at the local node. The value is given as an integer.

atom_limit

Returns the maximum number of atoms allowed. This limit can be increased at startup by passing command-line flag +t to erl(1).

ets_count

Returns the number of ETS tables currently existing at the local node.

ets_limit

Returns the limit for number of ETS tables. This limit is partially obsolete and number of tables are only limited by available memory.

port_count

Returns the number of ports currently existing at the local node. The value is given as an integer. This is the same value as returned by length(erlang:ports()), but more efficient.

port_limit

Returns the maximum number of simultaneously existing ports at the local node as an integer. This limit can be configured at startup by using command-line flag +Q in erl(1).

process_count

Returns the number of processes currently existing at the local node. The value is given as an integer. This is the same value as returned by length(processes()), but more efficient.

process_limit

Returns the maximum number of simultaneously existing processes at the local node. The value is given as an integer. This limit can be configured at startup by using command-line flag +P in erl(1).

erlang:system_info(Item :: end_time) -> integer() >= 0
erlang:system_info(Item :: os_monotonic_time_source) -> [{atom(), term()}]
erlang:system_info(Item :: os_system_time_source) ->
erlang:system_info(Item :: start_time) -> integer()
erlang:system_info(Item :: time_correction) -> true | false
erlang:system_info(Item :: time_offset) ->
  preliminary | final | volatile
erlang:system_info(Item :: time_warp_mode) ->
  no_time_warp | single_time_warp | multi_time_warp
erlang:system_info(Item :: tolerant_timeofday) ->
  enabled | disabled

Returns information about the current system (emulator) time as specified by Item:

end_time

The last Erlang monotonic time in native time unit that can be represented internally in the current Erlang runtime system instance. The time between the start time and the end time is at least a quarter of a millennium.

os_monotonic_time_source

Returns a list containing information about the source of OS monotonic time that is used by the runtime system.

If [] is returned, no OS monotonic time is available. The list contains two-tuples with Keys as first element, and Values as second element. The order of these tuples is undefined. The following tuples can be part of the list, but more tuples can be introduced in the future:

{function, Function}

Function is the name of the function used. This tuple always exists if OS monotonic time is available to the runtime system.

{clock_id, ClockId}

This tuple only exists if Function can be used with different clocks. ClockId corresponds to the clock identifier used when calling Function.

{resolution, OsMonotonicTimeResolution}

Highest possible resolution of current OS monotonic time source as parts per second. If no resolution information can be retrieved from the OS, OsMonotonicTimeResolution is set to the resolution of the time unit of Functions return value. That is, the actual resolution can be lower than OsMonotonicTimeResolution. Notice that the resolution does not say anything about the accuracy or whether the precision aligns with the resolution. You do, however, know that the precision is not better than OsMonotonicTimeResolution.

{used_resolution, UsedOsMonotonicTimeResolution}

The OS monotonic time resolution used by the runtime system. This is very often the same as OsMonotonicTimeResolution. However, on some systems the resolution has to be reduced in order to reliably produce monotonic timestamps. An example of this is when QueryPerformanceCounter() is used as OS monotonic time source on Windows. If such a reduction of the resolution has been done, UsedOsMonotonicTimeResolution will be smaller than OsMonotonicTimeResolution.

{extended, Extended}

Extended equals yes if the range of time values has been extended; otherwise Extended equals no. The range must be extended if Function returns values that wrap fast. This typically is the case when the return value is a 32-bit value.
{parallel, Parallel}
Parallel equals yes if Function is called in parallel from multiple threads. If it is not called in parallel, because calls must be serialized, Parallel equals no.

{time, OsMonotonicTime}
OsMonotonicTime equals current OS monotonic time in native time unit.

os_system_time_source
Returns a list containing information about the source of OS system time that is used by the runtime system. The list contains two-tuples with Keys as first element, and Values as second element. The order of these tuples is undefined. The following tuples can be part of the list, but more tuples can be introduced in the future:

{function, Function}
Function is the name of the function used.

{clock_id, ClockId}
Exists only if Function can be used with different clocks. ClockId corresponds to the clock identifier used when calling Function.

{resolution, OsSystemTimeResolution}
Highest possible resolution of current OS system time source as parts per second. If no resolution information can be retrieved from the OS, OsSystemTimeResolution is set to the resolution of the time unit of Functions return value. That is, the actual resolution can be lower than OsSystemTimeResolution. Notice that the resolution does not say anything about the accuracy or whether the precision do align with the resolution. You do, however, know that the precision is not better than OsSystemTimeResolution.

{parallel, Parallel}
Parallel equals yes if Function is called in parallel from multiple threads. If it is not called in parallel, because calls needs to be serialized, Parallel equals no.

{time, OsSystemTime}
OsSystemTime equals current OS system time in native time unit.

start_time
The Erlang monotonic time in native time unit at the time when current Erlang runtime system instance started. See also erlang:system_info(end_time).

time_correction
Returns a boolean value indicating whether time correction is enabled or not.

time_offset
Returns the state of the time offset:

preliminary
The time offset is preliminary, and will be changed and finalized later. The preliminary time offset is used during the preliminary phase of the single time warp mode.

final
The time offset is final. This either because no time warp mode is used, or because the time offset have been finalized when single time warp mode is used.

volatile
The time offset is volatile. That is, it can change at any time. This is because multi-time warp mode is used.
time_warp_mode

Returns a value identifying the time warp mode that is used:

no_time_warp
The no time warp mode is used.
single_time_warp
The single time warp mode is used.
multi_time_warp
The multi-time warp mode is used.

tolerant_timeofday

Returns whether a pre ERTS 7.0 backwards compatible compensation for sudden changes of system time is enabled or disabled. Such compensation is enabled when the time offset is final, and time correction is enabled.

erlang:system_info(Item :: dirty_cpu_schedulers) ->
integer() >= 0
erlang:system_info(Item :: dirty_cpu_schedulers_online) ->
integer() >= 0
erlang:system_info(Item :: dirty_io_schedulers) ->
integer() >= 0
erlang:system_info(Item :: multi_scheduling) ->
disabled | blocked | blocked_normal | enabled
erlang:system_info(Item :: multi_scheduling_blockers) ->
[Pid :: pid()]
erlang:system_info(Item :: normal_multi_scheduling_blockers) ->
[Pid :: pid()]
erlang:system_info(Item :: scheduler_bind_type) ->
spread | processor_spread | thread_spread |
thread_no_node_processor_spread |
no_node_processor_spread |
no_node_thread_spread | no_spread | unbound
erlang:system_info(Item :: scheduler_bindings) -> tuple()
erlang:system_info(Item :: scheduler_id) ->
SchedulerId :: integer() >= 1
erlang:system_info(Item :: schedulers | schedulers_online) ->
integer() >= 1
erlang:system_info(Item :: smp_support) -> boolean()
erlang:system_info(Item :: threads) -> boolean()
erlang:system_info(Item :: thread_pool_size) -> integer() >= 0

Returns information about schedulers, scheduling and threads in the current system as specified by Item:

dirty_cpu_schedulers

Returns the number of dirty CPU scheduler threads used by the emulator. Dirty CPU schedulers execute CPU-bound native functions, such as NIFs, linked-in driver code, and BIFs that cannot be managed cleanly by the normal emulator schedulers.
The number of dirty CPU scheduler threads is determined at emulator boot time and cannot be changed after that. However, the number of dirty CPU scheduler threads online can be changed at any time. The number of dirty CPU schedulers can be set at startup by passing command-line flag `+SDcpu` or `+SDPcpu` in `erl(1)`. See also `erlang:system_flag(dirty_cpu_schedulers_online, DirtyCPUSchedulersOnline)`, `erlang:system_info(dirty_cpu_schedulers_online)`, `erlang:system_info(dirty_cpu_schedulers)`, `erlang:system_info(schedulers)`, `erlang:system_info(schedulers_online)`, and `erlang:system_flag(schedulers_online, SchedulersOnline)`.

`dirty_cpu_schedulers_online` Returns the number of dirty CPU schedulers online. The return value satisfies $1 \leq \text{DirtyCPUSchedulersOnline} \leq N$, where $N$ is the smallest of the return values of `erlang:system_info(dirty_cpu_schedulers)` and `erlang:system_info(schedulers_online)`.

The number of dirty CPU schedulers online can be set at startup by passing command-line flag `+SDcpu` in `erl(1)`.

For more information, see `erlang:system_info(dirty_cpu_schedulers)`, `erlang:system_info(dirty_cpu_schedulers_online)`, `erlang:system_info(schedulers)`, and `erlang:system_flag(dirty_cpu_schedulers_online, DirtyCPUSchedulersOnline)`.

`dirty_io_schedulers` Returns the number of dirty I/O schedulers as an integer. Dirty I/O schedulers execute I/O-bound native functions, such as NIFs and linked-in driver code, which cannot be managed cleanly by the normal emulator schedulers.

This value can be set at startup by passing command-line argument `+SDio` in `erl(1)`.

For more information, see `erlang:system_info(dirty_cpu_schedulers)`, `erlang:system_info(dirty_cpu_schedulers_online)`, `erlang:system_flag(dirty_cpu_schedulers_online, DirtyCPUSchedulersOnline)`, and `erlang:system_info(schedulers)`.

`multi_scheduling` Returns one of the following:

- `disabled`
  The emulator has been started with only one scheduler thread.

- `blocked`
  The emulator has more than one scheduler thread, but all scheduler threads except one are blocked. That is, only one scheduler thread schedules Erlang processes and executes Erlang code.

- `blocked_normal`
  The emulator has more than one scheduler thread, but all normal scheduler threads except one are blocked. Notice that dirty schedulers are not blocked, and can schedule Erlang processes and execute native code.

- `enabled`
  The emulator has more than one scheduler thread, and no scheduler threads are blocked. That is, all available scheduler threads schedule Erlang processes and execute Erlang code.

See also `erlang:system_flag(multi_scheduling, BlockState)`, `erlang:system_info(multi_scheduling_blockers)`, `erlang:system_info(normal_multi_scheduling_blockers)`, and `erlang:system_info(schedulers)`.
multi_scheduling_blockers

Returns a list of Pids when multi-scheduling is blocked, otherwise the empty list is returned. The Pids in the list represent all the processes currently blocking multi-scheduling. A Pid occurs only once in the list, even if the corresponding process has blocked multiple times.

See also erlang:system_flag(multi_scheduling, BlockState), erlang:system_info(multi_scheduling), erlang:system_info(normal_multi_scheduling_blockers), and erlang:system_info(schedulers).

normal_multi_scheduling_blockers

Returns a list of Pids when normal multi-scheduling is blocked (that is, all normal schedulers but one is blocked), otherwise the empty list is returned. The Pids in the list represent all the processes currently blocking normal multi-scheduling. A Pid occurs only once in the list, even if the corresponding process has blocked multiple times.

See also erlang:system_flag(multi_scheduling, BlockState), erlang:system_info(multi_scheduling), erlang:system_info(normal_multi_scheduling_blockers), and erlang:system_info(schedulers).

scheduler_bind_type

Returns information about how the user has requested schedulers to be bound or not bound.

Notice that although a user has requested schedulers to be bound, they can silently have failed to bind. To inspect the scheduler bindings, call erlang:system_info(scheduler_bindings).

For more information, see command-line argument +sbt in erl(1) and erlang:system_info(scheduler_bindings).

scheduler_bindings

Returns information about the currently used scheduler bindings.

A tuple of a size equal to erlang:system_info(schedulers) is returned. The tuple elements are integers or the atom unbound. Logical processor identifiers are represented as integers. The $n$th element of the tuple equals the current binding for the scheduler with the scheduler identifier equal to $n$. For example, if the schedulers are bound, element(erlang:system_info(scheduler_id), erlang:system_info(scheduler_bindings)) returns the identifier of the logical processor that the calling process is executing on.

Notice that only schedulers online can be bound to logical processors.

For more information, see command-line argument +sbt in erl(1) and erlang:system_info(scheduler_bindings).

scheduler_id

Returns the scheduler ID (SchedulerId) of the scheduler thread that the calling process is executing on. SchedulerId is a positive integer, where 1 <= SchedulerId <= erlang:system_info(schedulers).

See also erlang:system_info(schedulers).

.schedulers

Returns the number of scheduler threads used by the emulator. Scheduler threads online schedules Erlang processes and Erlang ports, and execute Erlang code and Erlang linked-in driver code.

The number of scheduler threads is determined at emulator boot time and cannot be changed later. However, the number of schedulers online can be changed at any time.
See also `erlang:system_flag(schedulers_online, SchedulersOnline)`, `erlang:system_info(schedulers_online)`, `erlang:system_info(scheduler_id)`, `erlang:system_flag(multi_scheduling, BlockState)`, `erlang:system_info(multi_scheduling)`, `erlang:system_info(normal_multi_scheduling_blockers)` and `erlang:system_info(multi_scheduling_blockers)`.

**schedulers_online**

Returns the number of schedulers online. The scheduler identifiers of schedulers online satisfy the relationship $1 \leq \text{SchedulerId} \leq \text{erlang:system_info(schedulers_online)}$.

For more information, see `erlang:system_info(schedulers)` and `erlang:system_flag(schedulers_online, SchedulersOnline)`.

**smp_support**

Returns `true`.

**threads**

Returns `true`.

**thread_pool_size**

Returns the number of async threads in the async thread pool used for asynchronous driver calls (`erl_driver:driver_async()`). The value is given as an integer.

```
erlang:system_info(Item :: async_dist) -> boolean()
erlang:system_info(Item :: creation) -> integer()
erlang:system_info(Item :: delayed_node_table_gc) ->
  infinity | integer() >= 0
erlang:system_info(Item :: dist) -> binary()
erlang:system_info(Item :: dist_buf_busy_limit) ->
  integer() >= 0
erlang:system_info(Item :: dist_ctrl) ->
  [{Node :: node(),
    ControllingEntity :: port() | pid()}]
```

Returns information about Erlang Distribution in the current system as specified by `Item`:

**async_dist**

Since: OTP 25.3

Returns the value of the command line argument `-pad <boolean>` which the runtime system use. This value determines the default `async_dist` value for newly spawned processes.

**creation**

Returns the “creation” value of the local node as an integer. The creation is changed when a node is restarted. The creation of a node is stored in process identifiers, port identifiers, and references. This makes it possible to distinguish between identifiers from different incarnations of a node. Creation values are currently 32-bit positive integers, but this may change in future releases. If the node is not alive, 0 is returned.

**delayed_node_table_gc**

Returns the amount of time in seconds garbage collection of an entry in a node table is delayed. This limit can be set on startup by passing command-line flag `-zdntgc` to `erl(1)`. For more information, see the documentation of the command-line flag.
dist
Returns a binary containing a string of distribution information formatted as in Erlang crash dumps. For more information, see section How to interpret the Erlang crash dumps in the User’s Guide.

dist_buf_busy_limit
Returns the value of the distribution buffer busy limit in bytes. This limit can be set at startup by passing command-line flag +zdbbl to erl(1).

dist_ctrl
Returns a list of tuples \{Node, ControllingEntity\}. one entry for each connected remote node. Node is the node name and ControllingEntity is the port or process identifier responsible for the communication to that node. More specifically, ControllingEntity for nodes connected through TCP/IP (the normal case) is the socket used in communication with the specific node.

erlang:system_info(Item :: c_compiler_used) -> {atom(), term()}
erlang:system_info(Item :: check_io) -> [term()]
erlang:system_info(Item :: compat_rel) -> integer()
erlang:system_info(Item :: debug_compiled) -> boolean()
erlang:system_info(Item :: driver_version) -> string()
erlang:system_info(Item :: dynamic_trace) ->
none | dtrace | systemtap
erlang:system_info(Item :: dynamic_trace_probes) -> boolean()
erlang:system_info(Item :: emu_flavor) -> emu | jit
erlang:system_info(Item :: emu_type) ->
opt | debug | gcov | valgrind | gprof |
lcnt | frmptr
erlang:system_info(Item :: info) -> binary()
erlang:system_info(Item :: kernel_poll) -> boolean()
erlang:system_info(Item :: loaded) -> binary()
erlang:system_info(Item :: machine) -> string()
erlang:system_info(Item :: modified_timing_level) ->
integer() | undefined
erlang:system_info(Item :: nif_version) -> string()
erlang:system_info(Item :: otp_release) -> string()
erlang:system_info(Item :: outstanding_system_requests_limit) ->
1..134217727
erlang:system_info(Item :: port_parallelism) -> boolean()
erlang:system_info(Item :: system_architecture) -> string()
erlang:system_info(Item :: system_logger) ->
logger | undefined | pid()
erlang:system_info(Item :: system_version) -> string()
erlang:system_info(Item :: trace_control_word) ->
integer() >= 0
erlang:system_info(Item :: version) -> string()
erlang:system_info(Item ::
wordsize |
{wordsize, internal} |
{wordsize, external}) ->
4 | 8
Returns various information about the current system (emulator) as specified by Item:

build_type
   Deprecated, use emu_type instead

c_compiler_used
   Returns a two-tuple describing the C compiler used when compiling the runtime system. The first element is an atom describing the name of the compiler, or undefined if unknown. The second element is a term describing the version of the compiler, or undefined if unknown.

check io
   Returns a list containing miscellaneous information about the emulators internal I/O checking. Notice that the content of the returned list can vary between platforms and over time. It is only guaranteed that a list is returned.

compat rel
   Returns the compatibility mode of the local node as an integer. The integer returned represents the Erlang/OTP release that the current emulator has been set to be backward compatible with. The compatibility mode can be configured at startup by using command-line flag +R in erl(1).

debug compiled
   Returns true if the emulator has been debug-compiled, otherwise false.

driver version
   Returns a string containing the Erlang driver version used by the runtime system. It has the form "<major ver>.<minor ver>".

dynamic trace
   Returns an atom describing the dynamic trace framework compiled into the virtual machine. It can be dtrace, systemtap, or none. For a commercial or standard build, it is always none. The other return values indicate a custom configuration (for example, ./configure --with-dynamic-trace=dtrace). For more information about dynamic tracing, see dyntrace(3) manual page and the README.dtrace/README.systemtap files in the Erlang source code top directory.

dynamic trace probes
   Returns a boolean() indicating if dynamic trace probes (dtrace or systemtap) are built into the emulator. This can only be true if the virtual machine was built for dynamic tracing (that is, system_info(dynamic_trace) returns dtrace or systemtap).

emu flavor
   Returns an atom describing the flavor of the runtime system. This will be either emu or jit. Possible return values can be added or removed at any time without prior notice.

emu type
   Returns an atom describing the build type of the runtime system. This is normally the atom opt for optimized. Other possible return values are debug, gcov, valgrind, gprof, and lcnt. Possible return values can be added or removed at any time without prior notice.

info
   Returns a binary containing a string of miscellaneous system information formatted as in Erlang crash dumps. For more information, see section How to interpret the Erlang crash dumps in the User’s Guide.
**kernel_poll**

Returns true if the emulator uses some kind of kernel-poll implementation, otherwise false.

**loaded**

Returns a binary containing a string of loaded module information formatted as in Erlang crash dumps. For more information, see section How to interpret the Erlang crash dumps in the User’s Guide.

**machine**

Returns a string containing the Erlang machine name.

**modified_timing_level**

Returns the modified timing-level (an integer) if modified timing is enabled, otherwise undefined. For more information about modified timing, see command-line flag +T in erl(1).

**nif_version**

Returns a string containing the version of the Erlang NIF interface used by the runtime system. It is on the form “<major ver>.<minor ver>”.

**otp_release**

Returns a string containing the OTP release number of the OTP release that the currently executing ERTS application is part of.

As from Erlang/OTP 17, the OTP release number corresponds to the major OTP version number. No `erlang:system_info()` argument gives the exact OTP version. This is because the exact OTP version in the general case is difficult to determine. For more information, see the description of versions in System principles in System Documentation.

**outstanding_system_requests_limit**

Returns the limit on the amount of outstanding requests made by a system process orchestrating system wide changes. See `erlang:system_flag(outstanding_system_requests_limit, Limit)` for more information.

**port_parallelism**

Returns the default port parallelism scheduling hint used. For more information, see command-line argument +spp in erl(1).

**system_architecture**

Returns a string containing the processor and OS architecture the emulator is built for.

**system_logger**

Returns the current `system_logger` as set by `erlang:system_flag(system_logger, _)`.

**system_version**

Returns a string containing version number and some important properties, such as the number of schedulers.

**trace_control_word**

Returns the value of the node trace control word. For more information, see function `get_tcw` in section Match Specifications in Erlang in the User's Guide.

**version**

Returns a string containing the version number of the emulator.

**wordsize**

Same as `{wordsize, internal}`.
Returns the size of Erlang term words in bytes as an integer, that is, 4 is returned on a 32-bit architecture, and 8 is returned on a 64-bit architecture.

Returns the true word size of the emulator, that is, the size of a pointer. The value is given in bytes as an integer. On a pure 32-bit architecture, 4 is returned. On a 64-bit architecture, 8 is returned.

erlang:system_monitor() -> MonSettings
Types:
  MonSettings = undefined | {MonitorPid, Options}
  MonitorPid = pid()
  Options = [system_monitor_option()]
  system_monitor_option() =
    busy_port | busy_dist_port |
    {long_gc, integer() >= 0} |
    {long_schedule, integer() >= 0} |
    {large_heap, integer() >= 0}

Returns the current system monitoring settings set by erlang:system_monitor/2 as {MonitorPid, Options}, or undefined if no settings exist. The order of the options can be different from the one that was set.

erlang:system_monitor(Arg) -> MonSettings
Types:
  Arg = MonSettings = undefined | {MonitorPid, Options}
  MonitorPid = pid()
  Options = [system_monitor_option()]
  system_monitor_option() =
    busy_port | busy_dist_port |
    {long_gc, integer() >= 0} |
    {long_schedule, integer() >= 0} |
    {large_heap, integer() >= 0}

When called with argument undefined, all system performance monitoring settings are cleared.

Calling the function with {MonitorPid, Options} as argument is the same as calling erlang:system_monitor(MonitorPid, Options).

Returns the previous system monitor settings just like erlang:system_monitor/0.

erlang:system_monitor(MonitorPid, Options) -> MonSettings
Types:
  MonitorPid = pid()
  Options = [system_monitor_option()]
  MonSettings = undefined | {OldMonitorPid, OldOptions}
  OldMonitorPid = pid()
  OldOptions = [system_monitor_option()]
  system_monitor_option() =
    busy_port | busy_dist_port |
    {long_gc, integer() >= 0} |
Sets the system performance monitoring options. MonitorPid is a local process identifier (pid) receiving system monitor messages. The second argument is a list of monitoring options:

{long_schedule, integer() >= 0} | {large_heap, integer() >= 0}

If a garbage collection in the system takes at least Time wall clock milliseconds, a message {monitor, GcPid, long_gc, Info} is sent to MonitorPid. GcPid is the pid that was garbage collected. Info is a list of two-element tuples describing the result of the garbage collection.

One of the tuples is {timeout, GcTime}, where GcTime is the time for the garbage collection in milliseconds. The other tuples are tagged with heap_size, heap_block_size, stack_size, mbuf_size, old_heap_size, and old_heap_block_size. These tuples are explained in the description of trace message gc_minor_start (see erlang:trace/3). New tuples can be added, and the order of the tuples in the Info list can be changed at any time without prior notice.

{long_schedule, Time}

If a process or port in the system runs uninterrupted for at least Time wall clock milliseconds, a message {monitor, PidOrPort, long_schedule, Info} is sent to MonitorPid. PidOrPort is the process or port that was running. Info is a list of two-element tuples describing the event.

If a pid(), the tuples {timeout, Millis}, {in, Location}, and {out, Location} are present, where Location is either an MFA ([Module, Function, Arity]) describing the function where the process was scheduled in/out, or the atom undefined.

If a port(), the tuples {timeout, Millis} and {port_op, Op} are present. Op is one of proc_sig, timeout, input, output, event, or dist_cmd, depending on which driver callback was executing.

proc_sig is an internal operation and is never to appear, while the others represent the corresponding driver callbacks. timeout, ready_input, ready_output, event, and outputv (when the port is used by distribution). Value Millis in tuple timeout informs about the uninterrupted execution time of the process or port, which always is equal to or higher than the Time value supplied when starting the trace. New tuples can be added to the Info list in a future release. The order of the tuples in the list can be changed at any time without prior notice.

This can be used to detect problems with NIFs or drivers that take too long to execute. 1 ms is considered a good maximum time for a driver callback or a NIF. However, a time-sharing system is usually to consider everything < 100 ms as “possible” and fairly “normal”. However, longer schedule times can indicate swapping or a misbehaving NIF/driver. Misbehaving NIFs and drivers can cause bad resource utilization and bad overall system performance.

{l large_heap, Size}

If a garbage collection in the system results in the allocated size of a heap being at least Size words, a message {monitor, GcPid, large_heap, Info} is sent to MonitorPid. GcPid and Info are the same as for long_gc earlier, except that the tuple tagged with timeout is not present.

The monitor message is sent if the sum of the sizes of all memory blocks allocated for all heap generations after a garbage collection is equal to or higher than Size.

When a process is killed by max_heap_size, it is killed before the garbage collection is complete and thus no large heap message is sent.

busy_port

If a process in the system gets suspended because it sends to a busy port, a message {monitor, SusPid, busy_port, Port} is sent to MonitorPid. SusPid is the pid that got suspended when sending to Port.
busy_dist_port
If a process in the system gets suspended because it sends to a process on a remote node whose inter-node communication was handled by a busy port, a message \{monitor, SusPid, busy_dist_port, Port\} is sent to MonitorPid. SusPid is the pid that got suspended when sending through the inter-node communication port Port.

Returns the previous system monitor settings just like erlang:system_monitor/0.

The arguments to system_monitor/2 specifies how all system monitoring on the node should be done, not how it should be changed. This means only one process at a time (MonitorPid) can be the receiver of system monitor messages. Also, the way to clear a specific monitor option is to not include it in the list Options. All system monitoring will, however, be cleared if the process identified by MonitorPid terminates.

There are no special option values (like zero) to clear an option. Some of the options have a unspecified minimum value. Lower values will be adjusted to the minimum value. For example, it is currently not possible to monitor all garbage collections with \{long_gc, 0\}.

Note:
If a monitoring process gets so large that it itself starts to cause system monitor messages when garbage collecting, the messages enlarge the process message queue and probably make the problem worse.
Keep the monitoring process neat and do not set the system monitor limits too tight.

Failures:
badarg
  If MonitorPid does not exist.
badarg
  If MonitorPid is not a local process.

erlang:system_profile() -> ProfilerSettings
Types:
  ProfilerSettings = undefined | {ProfilerPid, Options}
  ProfilerPid = pid() | port()
  Options = [system_profile_option()]
  system_profile_option() =
    exclusive | runnable_ports | runnable_procs | scheduler |
    timestamp | monotonic_timestamp | strict_monotonic_timestamp

Returns the current system profiling settings set by erlang:system_profile/2 as \{ProfilerPid, Options\}, or undefined if there are no settings. The order of the options can be different from the one that was set.

erlang:system_profile(ProfilerPid, Options) -> ProfilerSettings
Types:
  ProfilerPid = pid() | port() | undefined
  Options = [system_profile_option()]
  ProfilerSettings =
    undefined | \{pid() | port(), [system_profile_option()]\}
    system_profile_option() =
      exclusive | runnable_ports | runnable_procs | scheduler |
timestamp | monotonic_timestamp | strict_monotonic_timestamp

Sets system profiler options. ProfilerPid is a local process identifier (pid) or port receiving profiling messages. The receiver is excluded from all profiling. The second argument is a list of profiling options:

exclusive

If a synchronous call to a port from a process is done, the calling process is considered not runnable during the call runtime to the port. The calling process is notified as inactive, and later active when the port callback returns.

monotonic_timestamp

Time stamps in profile messages use Erlang monotonic time. The time stamp (Ts) has the same format and value as produced by \texttt{erlang:monotonic\_time(nanosecond)}.

runnable_procs

If a process is put into or removed from the run queue, a message, \{profile, Pid, State, Mfa, Ts\}, is sent to \texttt{ProfilerPid}. Running processes that are reinserted into the run queue after having been pre-empted do not trigger this message.

runnable_ports

If a port is put into or removed from the run queue, a message, \{profile, Port, State, 0, Ts\}, is sent to \texttt{ProfilerPid}.

scheduler

If a scheduler is put to sleep or awoken, a message, \{profile, scheduler, Id, State, NoScheds, Ts\}, is sent to \texttt{ProfilerPid}.

strict_monotonic_timestamp

Time stamps in profile messages consist of Erlang monotonic time and a monotonically increasing integer. The time stamp (Ts) has the same format and value as produced by \{erlang:monotonic\_time(nanosecond), erlang:unique\_integer([monotonic])\}.

timestamp

Time stamps in profile messages include a time stamp (Ts) that has the same form as returned by \texttt{erlang:now()}. This is also the default if no time stamp flag is specified. If \texttt{cpu\_timestamp} has been enabled through \texttt{erlang:trace/3}, this also effects the time stamp produced in profiling messages when flag \texttt{timestamp} is enabled.

\textbf{Note:}

\texttt{erlang:system\_profile} behavior can change in a future release.

\texttt{erlang:system\_time()} \texttt{->} \texttt{integer()}

Returns current Erlang system time in native time unit.

Calling \texttt{erlang:system\_time()} is equivalent to \texttt{erlang:monotonic\_time() \texttt{+ erlang:time\_offset()}}.

\textbf{Note:}

This time is \textbf{not} a monotonically increasing time in the general case. For more information, see the documentation of time warp modes in the User's Guide.
erlang:system_time(Unit) -> integer()

Types:
   Unit = time_unit()

Returns current Erlang system time converted into the Unit passed as argument.

Calling erlang:system_time(Unit) is equivalent to
erlang:convert_time_unit(erlang:system_time(), native, Unit).

Note:
This time is not a monotonically increasing time in the general case. For more information, see the documentation of time warp modes in the User's Guide.

term_to_binary(Term) -> ext_binary()

Types:
   Term = term()

Returns a binary data object that is the result of encoding Term according to the Erlang external term format.

This can be used for various purposes, for example, writing a term to a file in an efficient way, or sending an Erlang term to some type of communications channel not supported by distributed Erlang.

> Bin = term_to_binary(hello).
<<131,100,0,5,104,101,108,108,111>>
> hello = binary_to_term(Bin).
hello

See also binary_to_term/1.

Note:
There is no guarantee that this function will return the same encoded representation for the same term.

term_to_binary(Term, Options) -> ext_binary()

Types:
   Term = term()
   Options =
      [compressed |
       {compressed, Level :: 0..9} |
       deterministic |
       {minor_version, Version :: 0..2} |
       local]

Returns a binary data object that is the result of encoding Term according to the Erlang external term format.

Currently supported options:

   compressed

   Compress the external term format. The compressed format is automatically recognized by
   binary_to_term/1 as from Erlang/OTP R7B.
{compressed, Level}
Compress the external term format to a given level. The compression level is specified by Level which is an integer in the range 0..9, where:

0
No compression is done (it is the same as giving no compressed option).

1
Takes least time but may not compress as well as the higher levels.

6
Default level when option compressed is provided.

9
Takes most time and tries to produce a smaller result. Notice "tries" in the preceding sentence; depending on the input term, level 9 compression either does or does not produce a smaller result than level 1 compression.

{minor_version, Version}
The option can be used to control some encoding details. Valid values for Version are:

0
Floats are encoded using a textual representation.
Atoms that can be represented by a latin1 string are encoded using latin1 while only atoms that cannot be represented by latin1 are encoded using utf8.

1
Floats are encoded in a more space-efficient and exact way (namely in the 64-bit IEEE format, rather than converted to a textual representation). As from Erlang/OTP R11B-4, binary_to_term/1 can decode this representation.
Atoms that can be represented by a latin1 string are encoded using latin1 while only atoms that cannot be represented by latin1 are encoded using utf8.

2
This is as of Erlang/OTP 26.0 the default. Atoms are unconditionally encoded using utf8. Erlang/OTP systems as of R16B can decode this representation.

deterministic
This option can be used to ensure that, within the same major release of Erlang/OTP, the same encoded representation is returned for the same term. There is still no guarantee that the encoded representation remains the same between major releases of Erlang/OTP.
This option cannot be combined with the local option.

local
This option will cause encoding of Term to an alternative local version of the external term format which when decoded by the same runtime system instance will produce a term identical to the encoded term even when the node name and/or creation of the current runtime system instance have changed between encoding and decoding. When encoding without the local option, local identifiers such as pids, ports and references will not be the same if node name and/or creation of the current runtime system instance changed between encoding and decoding. This since such identifiers refer to a specific node by node name and creation.
Node name and creation of a runtime system instance change when the distribution is started or stopped. The distribution is started when the runtime system is started using the -name or -sname command line arguments. Note that the actual start of the distribution happens after other code in the startup phase has begun.
executing. The distribution can also be started by calling `net_kernel:start/2` and stopped by calling `net_kernel:stop/1` if it has not been started via the command line.

The decoding of a term encoded with the `local` option, using for example `binary_to_term()`, will try to verify that the term actually was encoded by the same runtime system instance, and will in the vast majority of cases fail if the encoding was performed by another runtime system instance. You should however **not** trust that this verification will work in all cases. You **should** make sure to **only** decode terms encoded with the `local` option on the same Erlang runtime system instance as the one that encoded the terms.

Since it is only the runtime system that encoded a term using the `local` option that can decode it, the local encoding is typically pieced together with something else to produce a reply to where the local encoding originates from. If a term encoded using the `local` option is stripped of its leading version number, it can be added as part of a larger term (for example as an element in a tuple) when encoding on the external term format using, for example, `ei`. In the `ei` case, you would strip it of the version number using `ei_decode_version()` and then add the remaining local encoding to what you are encoding using for example `ei_x_append_buf()`.

A good example of when you want to use the `local` option, is when you want to make a request from a process to a port driver and utilize the selective receive optimization when receiving the reply. In this scenario you want to create a reference, serialize the reference on the external term format using the `local` option, pass this to the driver in the request, and then wait for the reply message in a selective receive matching on the reference. The driver should send the reply using either `erl_drv_output_term()` or `erl_drv_send_term()` using the term type `ERL_DRV_EXT2TERM` for the, in the request, previously received reference on the external term format. Note that you should not strip the leading version number from the local encoding when using the term type `ERL_DRV_EXT2TERM` of this functionality. If you in this example do not encode the reference using the `local` option, and the distribution is started or stopped while the request is ongoing, the process that made the request will hang indefinitely since the reference in the reply message will never match.

This option cannot be combined with the `deterministic` option.

For more information see the `LOCAL_EXT` tag in the documentation of the external term format.

See also `binary_to_term/1`.

```erlang
term_to_iocvec(Term) -> ext_iocvec()
```

Types:

```erlang
Term = term()
```

Returns the encoding of `Term` according to the Erlang external term format as `ext_iocvec()`.

This function produce the same encoding as `term_to_binary/1`, but with another return type. The call `iolist_to_binary(term_to_iocvec(Term))` will produce exactly the same result as the call `term_to_binary(Term)`.

`term_to_iocvec()` is a pure optimization of the functionality `term_to_binary()` provide. `term_to_iocvec()` can for example refer directly to off heap binaries instead of copying the binary data into the result.

See also `term_to_binary/1`.

```erlang
term_to_iocvec(Term, Options) -> ext_iocvec()
```

Types:

```erlang
Term = term()
Options =
    [compressed |
     {compressed, Level :: 0..9} |
      deterministic |
```
Returns the encoding of Term according to the Erlang external term format as ext_iovec().

This function produce the same encoding as term_to_binary/2, but with another return type. The call iolist_to_binary(term_to_iovec(Term, Opts)) will produce exactly the same result as term_to_binary(Term, Opts).

Currently recognised options are all options recognised by term_to_binary/2.

term_to_iovec() is a pure optimization of the functionality term_to_binary() provide. term_to_iovec() can for example refer directly to off heap binaries instead of copying the binary data into the result.

See also term_to_binary/2.

throw(Any) -> no_return()

Types:
  Any = term()

Raises an exception of class throw. Intended to be used to do non-local returns from functions.

If evaluated within a catch expression, the catch expression returns value Any. Example:

```erlang
> catch throw({hello, there}).
{hello,there}
```

If evaluated within a try-block of a try expression, the value Any can be caught within the catch block. Example:

```erlang
try
  throw({my_exception, "Something happened"})
catch
  throw:{my_exception, Desc} ->
    io:format(standard_error, "Error: ~s~n", [Desc])
end
```

Failure: nocatch if not caught by an exception handler.

See the guide about errors and error handling for additional information.

time() -> Time

Types:
  Time = calendar:time()

Returns the current time as {Hour, Minute, Second}.

The time zone and Daylight Saving Time correction depend on the underlying OS. The return value is based on the OS System Time. Example:

```erlang
> time().
{9,42,44}
```

erlang:time_offset() -> integer()

Returns the current time offset between Erlang monotonic time and Erlang system time in native time unit. Current time offset added to an Erlang monotonic time gives corresponding Erlang system time.
The time offset may or may not change during operation depending on the time warp mode used.

**Note:**
A change in time offset can be observed at slightly different points in time by different processes.

If the runtime system is in multi-time warp mode, the time offset is changed when the runtime system detects that the OS system time has changed. The runtime system will, however, not detect this immediately when it occurs. A task checking the time offset is scheduled to execute at least once a minute; so, under normal operation this is to be detected within a minute, but during heavy load it can take longer time.

```erlang
erlang:time_offset(Unit) -> integer()
```

Types:

- `Unit = time_unit()`

Returns the current time offset between Erlang monotonic time and Erlang system time converted into the `Unit` passed as argument.

Same as calling `erlang:convert_time_unit(erlang:time_offset(), native, Unit)` however optimized for commonly used `Units`.

```erlang
erlang:timestamp() -> Timestamp
```

Types:

- `Timestamp = timestamp()`

- `timestamp() = {MegaSecs :: integer() >= 0, Secs :: integer() >= 0, MicroSecs :: integer() >= 0}`

Returns current Erlang system time on the format `{MegaSecs, Secs, MicroSecs}`. This format is the same as `os:timestamp/0` and the deprecated `erlang:now/0` use. The reason for the existence of `erlang:timestamp()` is purely to simplify use for existing code that assumes this time stamp format. Current Erlang system time can more efficiently be retrieved in the time unit of your choice using `erlang:system_time/1`.

The `erlang:timestamp()` BIF is equivalent to:

```erlang
timestamp() ->
    ErlangSystemTime = erlang:system_time(microsecond),
    MegaSecs = ErlangSystemTime div 1000_000_000_000,
    Secs = ErlangSystemTime div 1000_000 - MegaSecs*1000_000,
    MicroSecs = ErlangSystemTime rem 1000_000,
    {MegaSecs, Secs, MicroSecs}.
```

It, however, uses a native implementation that does not build garbage on the heap and with slightly better performance.

**Note:**
This time is **not** a monotonically increasing time in the general case. For more information, see the documentation of time warp modes in the User's Guide.

```erlang
tl(List) -> Tail
```

Types:
List = nonempty_maybe_improper_list()
Tail = term()

Returns the tail of List, that is, the list minus the first element.
It works with improper lists.

Examples:

> tl([geesties, guilies, beasties]).
[guilies, beasties]

> tl([geesties]).
[]

> tl([geesties, guilies, beasties | improper_end]).
[guilies, beasties | improper_end]

> tl([geesties | improper_end]).
improper_end

Allowed in guard tests.
Failure: badarg if List is an empty list [].

erlang:trace(PidPortSpec, How, FlagList) -> integer()

Types:

PidPortSpec =
  pid() |
  port() |
  all | processes | ports | existing | existing_processes |
  existing_ports | new | new_processes | new_ports

How = boolean()

FlagList = [trace_flag()]

trace_flag() =
  all | send | 'receive' | proc | ports | call | arity |
  return_to | silent | running | exiting | running_procs |
  running_ports | garbage_collection | timestamp |
  cpu_timestamp | monotonic_timestamp |
  strict_monotonic_timestamp | set_on_spawn |
  set_on_first_spawn | set_on_link | set_on_first_link |
  {tracer, pid() | port()} |
  {tracer, module(), term()}

Turns on (if How == true) or off (if How == false) the trace flags in FlagList for the process or processes represented by PidPortSpec.

PidPortSpec is either a process identifier (pid) for a local process, a port identifier, or one of the following atoms:
all
  All currently existing processes and ports and all that will be created in the future.
processes
  All currently existing processes and all that will be created in the future.
ports
    All currently existing ports and all that will be created in the future.
existing
    All currently existing processes and ports.
existing_processes
    All currently existing processes.
existing_ports
    All currently existing ports.
new
    All processes and ports that will be created in the future.
new_processes
    All processes that will be created in the future.
new_ports
    All ports that will be created in the future.

FlagList can contain any number of the following flags (the "message tags" refers to the list of trace messages):

all
    Sets all trace flags except tracer and cpu_timestamp, which are in their nature different than the others.
send
    Traces sending of messages.
      Message tags: send and send_to_non_existing_process.
'receive'
    Traces receiving of messages.
      Message tags: 'receive'.
call
    Traces certain function calls. Specify which function calls to trace by calling erlang:trace_pattern/3.
      Message tags: call and return_from.
silent
    Used with the call trace flag. The call, return_from, and return_to trace messages are inhibited if this flag is set, but they are executed as normal if there are match specifications.
    Silent mode is inhibited by executing erlang:trace(_, false, [silent|_]), or by a match specification executing the function {silent, false}.
    The silent trace flag facilitates setting up a trace on many or even all processes in the system. The trace can then be activated and deactivated using the match specification function {silent,Bool}, giving a high degree of control of which functions with which arguments that trigger the trace.
    Message tags: call, return_from, and return_to. Or rather, the absence of.
return_to
    Used with the call trace flag. Traces the return from a traced function back to its caller. Only works for functions traced with option local to erlang:trace_pattern/3.
    The semantics is that a trace message is sent when a call traced function returns, that is, when a chain of tail recursive calls ends. Only one trace message is sent per chain of tail recursive calls, so the properties of tail recursiveness for function calls are kept while tracing with this flag. Using call and return_to trace together makes it possible to know exactly in which function a process executes at any time.
To get trace messages containing return values from functions, use the \{return_trace\} match specification action instead.

Message tags: return_to.

**procs**

Traces process-related events.

Message tags: spawn, spawned, exit, register, unregister, link, unlink, getting_linked, and getting_unlinked.

**ports**

Traces port-related events.

Message tags: open, closed, register, unregister, getting_linked, and getting_unlinked.

**running**

Traces scheduling of processes.

Message tags: in and out.

**exiting**

Traces scheduling of exiting processes.

Message tags: in_exiting, out_exiting, and out_exited.

**running_procs**

Traces scheduling of processes just like running. However, this option also includes schedule events when the process executes within the context of a port without being scheduled out itself.

Message tags: in and out.

**running_ports**

Traces scheduling of ports.

Message tags: in and out.

**garbage_collection**

Traces garbage collections of processes.

Message tags: gc_minor_start, gc_max_heap_size, and gc_minor_end.

**timestamp**

Includes a time stamp in all trace messages. The time stamp (Ts) has the same form as returned by erlang:now().

**cpu_timestamp**

A global trace flag for the Erlang node that makes all trace time stamps using flag timestamp to be in CPU time, not wall clock time. That is, cpu_timestamp is not be used if monotonic_timestamp or strict_monotonic_timestamp is enabled. Only allowed with PidPortSpec==all. If the host machine OS does not support high-resolution CPU time measurements, trace/3 exits with badarg. Notice that most OS do not synchronize this value across cores, so be prepared that time can seem to go backwards when using this option.

**monotonic_timestamp**

Includes an Erlang monotonic time time stamp in all trace messages. The time stamp (Ts) has the same format and value as produced by erlang:monotonic_time(nanosecond). This flag overrides flag cpu_timestamp.
strict_monotonic_timestamp

Includes an time stamp consisting of Erlang monotonic time and a monotonically increasing integer in all trace messages. The time stamp (Ts) has the same format and value as produced by {erlang:monotonic_time(nanosecond), erlang:unique_integer([monotonic])}. This flag overrides flag cpu_timestamp.

arity

Used with the call trace flag. {M, F, Arity} is specified instead of {M, F, Args} in call trace messages.

set_on_spawn

Makes any process created by a traced process inherit its trace flags, including flag set_on_spawn.

set_on_first_spawn

Makes the first process created by a traced process inherit its trace flags, excluding flag set_on_first_spawn.

set_on_link

Makes any process linked by a traced process inherit its trace flags, including flag set_on_link.

set_on_first_link

Makes the first process linked to by a traced process inherit its trace flags, excluding flag set_on_first_link.

{tracer, Tracer}

Specifies where to send the trace messages. Tracer must be the process identifier of a local process or the port identifier of a local port.

{tracer, TracerModule, TracerState}

Specifies that a tracer module is to be called instead of sending a trace message. The tracer module can then ignore or change the trace message. For more details on how to write a tracer module, see erl_tracer(3).

If no tracer is specified, the calling process receives all the trace messages.

The effect of combining set_on_first_link with set_on_link is the same as set_on_first_link alone. Likewise for set_on_spawn and set_on_first_spawn.

The tracing process receives the trace messages described in the following list. Pid is the process identifier of the traced process in which the traced event has occurred. The third tuple element is the message tag.

If flag timestamp, strict_monotonic_timestamp, or monotonic_timestamp is specified, the first tuple element is trace_ts instead, and the time stamp is added as an extra element last in the message tuple. If multiple time stamp flags are passed, timestamp has precedence over strict_monotonic_timestamp, which in turn has precedence over monotonic_timestamp. All time stamp flags are remembered, so if two are passed and the one with highest precedence later is disabled, the other one becomes active.

If a match specification (applicable only for call, send and 'receive' tracing) contains a (message) action function with a non-boolean value, that value is added as an extra element to the message tuple either in the last position or before the timestamp (if it is present).

Trace messages:

{trace, PidPort, send, Msg, To}

When PidPort sends message Msg to process To.

{trace, PidPort, send_to_non_existing_process, Msg, To}

When PidPort sends message Msg to the non-existing process To.
{trace, PidPort, 'receive', Msg}
When PidPort receives message Msg. If Msg is set to time-out, a receive statement can have timed out, or the process received a message with the payload timeout.

{trace, Pid, call, {M, F, {Args}}}
When Pid calls a traced function. The return values of calls are never supplied, only the call and its arguments.
Trace flag arity can be used to change the contents of this message, so that Arity is specified instead of Args.

{trace, Pid, return_to, {M, F, Arity}}
When Pid returns to the specified function. This trace message is sent if both the flags call and return_to are set, and the function is set to be traced on local function calls. The message is only sent when returning from a chain of tail recursive function calls, where at least one call generated a call trace message (that is, the functions match specification matched, and (message, false) was not an action).

{trace, Pid, return_from, {M, F, Arity}, ReturnValue}
When Pid returns from the specified function. This trace message is sent if flag call is set, and the function has a match specification with a return_trace or exception_trace action.

{trace, Pid, exception_from, {M, F, Arity}, {Class, Value}}
When Pid exits from the specified function because of an exception. This trace message is sent if flag call is set, and the function has a match specification with an exception_trace action.

{trace, Pid, spawn, Pid2, {M, F, Args}}
When Pid spawns a new process Pid2 with the specified function call as entry point.
Args is supposed to be the argument list, but can be any term if the spawn is erroneous.

{trace, Pid, spawned, Pid2, {M, F, Args}}
When Pid is spawned by process Pid2 with the specified function call as entry point.
Args is supposed to be the argument list, but can be any term if the spawn is erroneous.

{trace, Pid, exit, Reason}
When Pid exits with reason Reason.

{trace, PidPort, register, RegName}
When PidPort gets the name RegName registered.

{trace, PidPort, unregister, RegName}
When PidPort gets the name RegName unregistered. This is done automatically when a registered process or port exits.

{trace, Pid, link, Pid2}
When Pid links to a process Pid2.

{trace, Pid, unlink, Pid2}
When Pid removes the link from a process Pid2.

{trace, PidPort, getting_linked, Pid2}
When PidPort gets linked to a process Pid2.

{trace, PidPort, getting_unlinked, Pid2}
When PidPort gets unlinked from a process Pid2.
{trace, Port, open, Pid, Driver}

When Pid opens a new port Port with the running Driver.

Driver is the name of the driver as an atom.

{trace, Port, closed, Reason}

When Port closes with Reason.

{trace, Pid, in | in_exiting, {M, F, Arity} | 0}

When Pid is scheduled to run. The process runs in function {M, F, Arity}. On some rare occasions, the current function cannot be determined, then the last element is 0.

{trace, Pid, out | out_exiting | out_exited, {M, F, Arity} | 0}

When Pid is scheduled out. The process was running in function {M, F, Arity}. On some rare occasions, the current function cannot be determined, then the last element is 0.

{trace, Pid, in, Command | 0}

When Port is scheduled to run. Command is the first thing the port will execute, it can however run several commands before being scheduled out. On some rare occasions, the current function cannot be determined, then the last element is 0.

The possible commands are call, close, command, connect, control, flush, info, link, open, and unlink.

{trace, Pid, out, Command | 0}

When Port is scheduled out. The last command run was Command. On some rare occasions, the current function cannot be determined, then the last element is 0. Command can contain the same commands as in

{trace, Pid, gc_minor_start, Info}

Sent when a young garbage collection is about to be started. Info is a list of two-element tuples, where the first element is a key, and the second is the value. Do not depend on any order of the tuples. The following keys are defined:

heap_size
  The size of the used part of the heap.
heap_block_size
  The size of the memory block used for storing the heap and the stack.
old_heap_size
  The size of the used part of the old heap.
old_heap_block_size
  The size of the memory block used for storing the old heap.
stack_size
  The size of the stack.
recent_size
  The size of the data that survived the previous garbage collection.
mbuf_size
  The combined size of message buffers associated with the process.
bin_vheap_size
  The total size of unique off-heap binaries referenced from the process heap.
bin_vheap_block_size
  The total size of binaries allowed in the virtual heap in the process before doing a garbage collection.
bin_old_vheap_size
  The total size of unique off-heap binaries referenced from the process old heap.
bin_old_vheap_block_size
The total size of binaries allowed in the virtual old heap in the process before doing a garbage collection.

wordsize
For the gc_minor_start event it is the size of the need that triggered the GC. For the corresponding gc_minor_end event it is the size of reclaimed memory = start heap_size - end heap_size.

All sizes are in words.

{trace, Pid, gc_max_heap_size, Info}
Sent when the max_heap_size is reached during garbage collection. Info contains the same kind of list as in message gc_start, but the sizes reflect the sizes that triggered max_heap_size to be reached.

{trace, Pid, gc_minor_end, Info}
Sent when young garbage collection is finished. Info contains the same kind of list as in message gc_minor_start, but the sizes reflect the new sizes after garbage collection.

{trace, Pid, gc_major_start, Info}
Sent when fullsweep garbage collection is about to be started. Info contains the same kind of list as in message gc_minor_start.

{trace, Pid, gc_major_end, Info}
Sent when fullsweep garbage collection is finished. Info contains the same kind of list as in message gc_minor_start, but the sizes reflect the new sizes after a fullsweep garbage collection.

If the tracing process/port dies or the tracer module returns remove, the flags are silently removed.

Each process can only be traced by one tracer. Therefore, attempts to trace an already traced process fail.

Returns a number indicating the number of processes that matched PidPortSpec. If PidPortSpec is a process identifier, the return value is 1. If PidPortSpec is all or existing, the return value is the number of processes running. If PidPortSpec is new, the return value is 0.

Failure: badarg if the specified arguments are not supported. For example, cpu_timestamp is not supported on all platforms.

erlang:trace_delivered(Tracee) -> Ref
Types:
    Tracee = pid() | all
    Ref = reference()

The delivery of trace messages (generated by erlang:trace/3, seq_trace(3), or erlang:system_profile/2) is dislocated on the time-line compared to other events in the system. If you know that Tracee has passed some specific point in its execution, and you want to know when at least all trace messages corresponding to events up to this point have reached the tracer, use erlang:trace_delivered(Tracee).

When it is guaranteed that all trace messages are delivered to the tracer up to the point that Tracee reached at the time of the call to erlang:trace_delivered(Tracee), then a {trace_delivered, Tracee, Ref} message is sent to the caller of erlang:trace_delivered(Tracee).

Notice that message trace_delivered does not imply that trace messages have been delivered. Instead it implies that all trace messages that are to be delivered have been delivered. It is not an error if Tracee is not, and has not been traced by someone, but if this is the case, no trace messages have been delivered when the trace_delivered message arrives.

Notice that Tracee must refer to a process currently or previously existing on the same node as the caller of erlang:trace_delivered(Tracee) resides on. The special Tracee atom all denotes all processes that currently are traced in the node.
When used together with a Tracer Module, any message sent in the trace callback is guaranteed to have reached its recipient before the `trace_delivered` message is sent.

Example: Process A is Tracee, port B is tracer, and process C is the port owner of B. C wants to close B when A exits. To ensure that the trace is not truncated, C can call `erlang:trace_delivered(A)` when A exits, and wait for message `{trace_delivered, A, Ref}` before closing B.

Failure: badarg if Tracee does not refer to a process (dead or alive) on the same node as the caller of `erlang:trace_delivered(Tracee)` resides on.

```
erlang:trace_info(PidPortFuncEvent, Item) -> Res
```

Types:

- `PidPortFuncEvent` =
  - `pid()` |
  - `port()` |
  - `new` | `new_processes` | `new_ports` |
  - `{Module, Function, Arity}` |
  - `on_load` | `send` | `'receive'`

- `Module` = `module()`
- `Function` = `atom()`
- `Arity` = `arity()`

- `Item` =
  - `flags` | `tracer` | `traced` | `match_spec` | `meta` |
  - `meta_match_spec` | `call_count` | `call_time` | `call_memory` | `all`

- `Res = trace_info_return()`

```
trace_info_return() =
  undefined |
  `{flags, [trace_info_flag()]} |` |
  `{tracer, pid() | port() | []} |` |
  `{tracer, module(), term()} |` |
  `trace_info_item_result()} |`
  `{all, [trace_info_item_result()]} | false | undefined}
```

```
trace_info_item_result() =
  `{traced, global | local | false | undefined} |` |
  `{match_spec, trace_match_spec() | false | undefined} |` |
  `{meta, pid() | port() | false | undefined | []} |` |
  `{meta, module(), term()} |` |
  `{meta_match_spec, trace_match_spec() | false | undefined} |` |
  `{call_count, integer() >= 0 | boolean() | undefined} |` |
  `{call_time | call_memory, |
  [{pid(), |
  integer() >= 0, |
  integer() >= 0, |
  integer() >= 0} |` |
  boolean() | undefined |
  undefined}
```

```
trace_info_flag() =
  send | `'receive' | set_on_spawn | call | return_to | procs | set_on_first_spawn | set_on_link | running | garbage_collection | timestamp | monotonic_timestamp |
strict_monotonic_timestamp | arity
trace_match_spec() =
  [[{term()} | '_' | match_variable(), [term()],[term()]]]
match_variable() = atom()
Approximation of '$1' | '$2' | '$3' | ...

Returns trace information about a port, process, function, or event.

**To get information about a port or process**, *PidPortFuncEvent* is to be a process identifier (pid), port identifier, or one of the atoms new, new_processes, or new_ports. The atom new or new_processes means that the default trace state for processes to be created is returned. The atom new_ports means that the default trace state for ports to be created is returned.

Valid Items for ports and processes:

flags

- Returns a list of atoms indicating what kind of traces is enabled for the process. The list is empty if no traces are enabled, and one or more of the following atoms if traces are enabled: send, 'receive', set_on_spawn, call, return_to, procfs, ports, set_on_first_spawn, set_on_link, running, running_procs, running_ports, silent, exiting, monotonic_timestamp, strict_monotonic_timestamp, garbage_collection, timestamp, and arity. The order is arbitrary.

tracer

- Returns the identifier for process, port, or a tuple containing the tracer module and tracer state tracing this process. If this process is not traced, the return value is [].

**To get information about a function**, *PidPortFuncEvent* is to be the three-element tuple {Module, Function, Arity} or the atom on_load. No wildcards are allowed. Returns undefined if the function does not exist, or false if the function is not traced. If *PidPortFuncEvent* is on_load, the information returned refers to the default value for code that will be loaded.

Valid Items for functions:

traced

- Returns global if this function is traced on global function calls, local if this function is traced on local function calls (that is, local and global function calls), and false if local or global function calls are not traced.

match_spec

- Returns the match specification for this function, if it has one. If the function is locally or globally traced but has no match specification defined, the returned value is [].

meta

- Returns the meta-trace tracer process, port, or trace module for this function, if it has one. If the function is not meta-traced, the returned value is false. If the function is meta-traced but has once detected that the tracer process is invalid, the returned value is [].

meta_match_spec

- Returns the meta-trace match specification for this function, if it has one. If the function is meta-traced but has no match specification defined, the returned value is [].

call_count

- Returns the call count value for this function or true for the pseudo function on_load if call count tracing is active. Otherwise false is returned.

See also erlang:trace_pattern/3.
call_time
Returns the call time values for this function or true for the pseudo function on_load if call time tracing is active. Otherwise false is returned. The call time values returned, [{Pid, Count, S, Us}], is a list of each process that executed the function and its specific counters.

See also erlang:trace_pattern/3.

call_memory
Returns the accumulated number of words allocated by this function. Accumulation stops at the next memory traced function: if there are outer, middle and inner functions each allocating 3 words, but only outer is traced, it will report 9 allocated words. If outer and inner are traced, 6 words are reported for outer and 3 for inner. When function is not traced, false is returned. Returned tuple is [{Pid, Count, Words}], for each process that executed the function.

See also erlang:trace_pattern/3.

all
Returns a list containing the {Item, Value} tuples for all other items, or returns false if no tracing is active for this function.

To get information about an event, PidPortFuncEvent is to be one of the atoms send or 'receive'.

One valid Item for events exists:

match_spec
Returns the match specification for this event, if it has one, or true if no match specification has been set.

The return value is {Item, Value}, where Value is the requested information as described earlier. If a pid for a dead process was specified, or the name of a non-existing function, Value is undefined.

erlang:trace_pattern(MFA, MatchSpec) -> integer() >= 0
Types:
MFA = trace_pattern_mfa() | send | 'receive'
MatchSpec =
  (MatchSpecList :: trace_match_spec()) | boolean() | restart | pause
trace_pattern_mfa() = {atom(), atom(), arity() | '_'} | on_load
trace_match_spec() =
  [{[term() | '_' | match_variable(), [term()], [term()]]
match_variable() = atom()
Approximation of '$1' | '$2' | '$3' |

The same as erlang:trace_pattern(Event, MatchSpec, []), retained for backward compatibility.

erlang:trace_pattern(MFA :: send, MatchSpec, FlagList :: []) ->
  integer() >= 0
Types:
MatchSpec = (MatchSpecList :: trace_match_spec()) | boolean()
trace_match_spec() =
  [{[term() | '_' | match_variable(), [term()], [term()]]
match_variable() = atom()
Approximation of '$1' | '$2' | '$3' |
Sets trace pattern for **message sending**. Must be combined with **erlang:trace/3** to set the send trace flag for one or more processes. By default all messages sent from send traced processes are traced. To limit traced send events based on the message content, the sender and/or the receiver, use **erlang:trace_pattern/3**.

Argument **MatchSpec** can take the following forms:

**MatchSpecList**

A list of match specifications. The matching is done on the list [Receiver, Msg]. Receiver is the process or port identity of the receiver and Msg is the message term. The pid of the sending process can be accessed with the guard function **self/0**. An empty list is the same as **true**. For more information, see section **Match Specifications in Erlang** in the User's Guide.

**true**

Enables tracing for all sent messages (from send traced processes). Any match specification is removed. **This is the default.**

**false**

Disables tracing for all sent messages. Any match specification is removed.

Argument **FlagList** must be [] for send tracing.

The return value is always 1.

Examples:

Only trace messages to a specific process Pid:

```erlang
> erlang:trace_pattern(send, [[Pid, '_'],[],[]], []).
1
```

Only trace messages matching {reply, _}:

```erlang
> erlang:trace_pattern(send, [['_'], {reply,'_'}],[],[]), []).
1
```

Only trace messages sent to the sender itself:

```erlang
> erlang:trace_pattern(send, [['$1', '_']],[{'=/=',node,'$1',node}],[],[]), []).
1
```

Only trace messages sent to other nodes:

```erlang
> erlang:trace_pattern(send, [['$1', '_']],[{'=:=','$1',{self}}],[],[]), []).
1
```

**Note:**

A match specification for send trace can use all guard and body functions except **caller**.

Fails by raising an error exception with an error reason of:

**badarg**

If an argument is invalid.
system_limit

If a match specification passed as argument has excessive nesting which causes scheduler stack exhaustion for the scheduler that the calling process is executing on. Scheduler stack size can be configured when starting the runtime system.

erlang:trace_pattern(MFA :: 'receive', MatchSpec, FlagList :: []) ->
    integer() >= 0

Types:
    MatchSpec = (MatchSpecList :: trace_match_spec()) | boolean()
    trace_match_spec() =
        [{[term()] | '_' | match_variable(), [term()], [term()]}]
    match_variable() = atom()
        Approximation of '$1' | '$2' | '$3' | ...

Sets trace pattern for message receiving. Must be combined with erlang:trace/3 to set the 'receive' trace flag for one or more processes. By default all messages received by 'receive' traced processes are traced. To limit traced receive events based on the message content, the sender and/or the receiver, use erlang:trace_pattern/3.

Argument MatchSpec can take the following forms:

MatchSpecList

A list of match specifications. The matching is done on the list [Node, Sender, Msg]. Node is the node name of the sender. Sender is the process or port identity of the sender, or the atom undefined if the sender is not known (which can be the case for remote senders). Msg is the message term. The pid of the receiving process can be accessed with the guard function self/0. An empty list is the same as true. For more information, see section Match Specifications in Erlang in the User's Guide.

true

Enables tracing for all received messages (to 'receive' traced processes). Any match specification is removed.

This is the default.

false

Disables tracing for all received messages. Any match specification is removed.

Argument FlagList must be [] for receive tracing.

The return value is always 1.

Examples:

Only trace messages from a specific process Pid:

> erlang:trace_pattern('receive', [[{'_','Pid','_'},[],[]]], []).
1

Only trace messages matching {reply, _}:

> erlang:trace_pattern('receive', [[{'_','_'}, {reply,'_'}],[],[]]], []).
1

Only trace messages from other nodes:
> erlang:trace_pattern('receive', [[{'$1', '_', '_'}], [{'==/=' , '$1', {node}}], []], []).
1

**Note:**

A match specification for 'receive' trace can use all guard and body functions except caller, is_seq_trace, get_seq_token, set_seq_token, enable_trace, disable_trace, trace, silent, and process_dump.

Fails by raising an error exception with an error reason of:

- **badarg**
  - If an argument is invalid.

- **system_limit**
  - If a match specification passed as argument has excessive nesting which causes scheduler stack exhaustion for the scheduler that the calling process is executing on. Scheduler stack size can be configured when starting the runtime system.

**erlang:trace_pattern(MFA, MatchSpec, FlagList) ->
integer() >= 0**

Types:

- **MFA** = trace_pattern_mfa()
- **MatchSpec** =
  - (MatchSpecList :: trace_match_spec()) | boolean() | restart | pause
- **FlagList** = [trace_pattern_flag()]
- **trace_pattern_mfa()** = {atom(), atom(), arity() | '_'} | on_load
- **trace_match_spec()** =
  - [[term()] | '_' | match_variable(), [term()], [term()]]
- **trace_pattern_flag()** =
  - global | local | meta |
  - {meta, Pid :: pid()}
  - {meta, TracerModule :: module(), TracerState :: term()}
  - call_count | call_time | call_memory
- **match_variable()** = atom()

Approximation of '$1' | '$2' | '$3'...

Enables or disables call tracing for one or more functions. Must be combined with erlang:trace/3 to set the call trace flag for one or more processes.

Conceptually, call tracing works as follows. Inside the Erlang virtual machine, a set of processes and a set of functions are to be traced. If a traced process calls a traced function, the trace action is taken. Otherwise, nothing happens.

To add or remove one or more processes to the set of traced processes, use erlang:trace/3.

To add or remove functions to the set of traced functions, use erlang:trace_pattern/3.

The BIF erlang:trace_pattern/3 can also add match specifications to a function. A match specification comprises a pattern that the function arguments must match, a guard expression that must evaluate to true, and an
action to be performed. The default action is to send a trace message. If the pattern does not match or the guard fails, the action is not executed.

Argument MFA is to be a tuple, such as \{Module, Function, Arity\}, or the atom on_load (described below). It can be the module, function, and arity for a function (or a BIF in any module). The atom '(_) can be used as a wildcard in any of the following ways:

\{Module,Function,'_'\}

All functions of any arity named Function in module Module.

\{Module,'_','_\}

All functions in module Module.

\{'_','_','_\}

All functions in all loaded modules.

Other combinations, such as \{Module,'_',Arity\}, are not allowed. Local functions match wildcards only if option local is in FlagList.

If argument MFA is the atom on_load, the match specification and flag list are used on all modules that are newly loaded.

Argument MatchSpec can take the following forms:

false

Disables tracing for the matching functions. Any match specification is removed.

ture

Enables tracing for the matching functions. Any match specification is removed.

MatchSpecList

A list of match specifications. An empty list is equivalent to true. For a description of match specifications, see section Match Specifications in Erlang in the User's Guide.

restart

For the FlagList options call_count, call_time and call_memory: restarts the existing counters. The behavior is undefined for other FlagList options.

pause

For the FlagList options call_count, call_time and call_memory: pauses the existing counters. The behavior is undefined for other FlagList options.

Parameter FlagList is a list of options. The following are the valid options:

global

Turns on or off call tracing for global function calls (that is, calls specifying the module explicitly). Only exported functions match and only global calls generate trace messages. This is the default.

local

Turns on or off call tracing for all types of function calls. Trace messages are sent whenever any of the specified functions are called, regardless of how they are called. If flag return_to is set for the process, a return_to message is also sent when this function returns to its caller.

meta | {meta, Pid} | {meta, TracerModule, TracerState}

Turns on or off meta-tracing for all types of function calls. Trace messages are sent to the tracer whenever any of the specified functions are called. If no tracer is specified, self() is used as a default tracer process.
Meta-tracing traces all processes and does not care about the process trace flags set by `erlang:trace/3`, the trace flags are instead fixed to `[call, timestamp]`.

The match specification function `{return_trace}` works with meta-trace and sends its trace message to the same tracer.

**call_count**

Starts `(MatchSpec == true)` or stops `(MatchSpec == false)` call count tracing for all types of function calls. For every function, a counter is incremented when the function is called, in any process. No process trace flags need to be activated.

If call count tracing is started while already running, the count is restarted from zero. To pause running counters, use `MatchSpec == pause`. Paused and running counters can be restarted from zero with `MatchSpec == restart`.

To read the counter value, use `erlang:trace_info/2`.

**call_time**

Starts `(MatchSpec == true)` or stops `(MatchSpec == false)` call time tracing for all types of function calls. For every function, a counter is incremented when the function is called. Time spent in the function is accumulated in two other counters, seconds and microseconds. The counters are stored for each call traced process.

If call time tracing is started while already running, the count and time restart from zero. To pause running counters, use `MatchSpec == pause`. Paused and running counters can be restarted from zero with `MatchSpec == restart`.

To read the counter value, use `erlang:trace_info/2`.

**call_memory**

Starts `(MatchSpec == true)` or stops `(MatchSpec == false)` call memory tracing for all types of function calls.

If call memory tracing is started while already running, counters and allocations restart from zero. To pause running counters, use `MatchSpec == pause`. Paused and running counters can be restarted from zero with `MatchSpec == restart`.

To read the counter value, use `erlang:trace_info/2`.

The options `global` and `local` are mutually exclusive, and `global` is the default (if no options are specified). The options `call_count` and `meta` perform a kind of local tracing, and cannot be combined with `global`. A function can be globally or locally traced. If global tracing is specified for a set of functions, then local, meta, call time, and call count tracing for the matching set of local functions is disabled, and conversely.

When disabling trace, the option must match the type of trace set on the function. That is, local tracing must be disabled with option `local` and global tracing with option `global` (or no option), and so on.

Part of a match specification list cannot be changed directly. If a function has a match specification, it can be replaced with a new one. To change an existing match specification, use the BIF `erlang:trace_info/2` to retrieve the existing match specification.

Returns the number of functions matching argument `MFA`. This is zero if none matched.

Fails by raising an error exception with an error reason of:

**badarg**

If an argument is invalid.
system_limit

If a match specification passed as argument has excessive nesting which causes scheduler stack exhaustion for
the scheduler that the calling process is executing on. Scheduler stack size can be configured when starting the
runtime system.

trunc(Number) -> integer()
Types:
    Number = number()
Truncates the decimals of Number, for example:

> trunc(5.7).
  5

> trunc(-5.7).
  -5

> trunc(5).
  5

> trunc(36028797018963969.0).
  36028797018963968

In the last example, trunc(36028797018963969.0) evaluates to 36028797018963968. The reason for this
is that the number 36028797018963969.0 cannot be represented exactly as a float value. Instead, the float literal
is represented as 36028797018963968.0, which is the closest number that can be represented exactly as a float
value. See Representation of Floating Point Numbers for additional information.

Allowed in guard tests.

tuple_size(Tuple) -> integer() >= 0
Types:
    Tuple = tuple()
Returns an integer that is the number of elements in Tuple, for example:

> tuple_size({morni, mulle, bwange}).
  3

Allowed in guard tests.

tuple_to_list(Tuple) -> [term()]
Types:
    Tuple = tuple()
Returns a list corresponding to Tuple. Tuple can contain any Erlang terms. Example:

> tuple_to_list([share, {'Ericsson_B', 163}]).
  [share, {'Ericsson_B', 163}]
unalias(Alias) -> boolean()
Types:
  Alias = reference()
Deactivate the alias Alias previously created by the calling process. An alias can for example be created via alias/0, or monitor/3. unalias/1 will always deactivate the alias regardless of options used when creating the alias.
Returns true if Alias was a currently active alias for current processes; otherwise, false.
For more information on process aliases see the Process Aliases section of the Erlang Reference Manual.

erlang:unique_integer() -> integer()
Generates and returns an integer unique on current runtime system instance. The same as calling erlang:unique_integer([]).

erlang:unique_integer(ModifierList) -> integer()
Types:
  ModifierList = [Modifier]
  Modifier = positive | monotonic
Generates and returns an integer unique on current runtime system instance. The integer is unique in the sense that this BIF, using the same set of modifiers, does not return the same integer more than once on the current runtime system instance. Each integer value can of course be constructed by other means.
By default, when [] is passed as ModifierList, both negative and positive integers can be returned. This to use the range of integers that do not need heap memory allocation as much as possible. By default the returned integers are also only guaranteed to be unique, that is, any returned integer can be smaller or larger than previously returned integers.
Modifiers:
  positive
    Returns only positive integers.
    Notice that by passing the positive modifier you will get heap allocated integers (bignums) quicker.
  monotonic
    Returns strictly monotonically increasing integers corresponding to creation time. That is, the integer returned is always larger than previously returned integers on the current runtime system instance.
    These values can be used to determine order between events on the runtime system instance. That is, if both X = erlang:unique_integer([monotonic]) and Y = erlang:unique_integer([monotonic]) are executed by different processes (or the same process) on the same runtime system instance and X < Y, we know that X was created before Y.

**Warning:**

Strictly monotonically increasing values are inherently quite expensive to generate and scales poorly. This is because the values need to be synchronized between CPU cores. That is, do not pass the monotonic modifier unless you really need strictly monotonically increasing values.

All valid Modifiers can be combined. Repeated (valid) Modifiers in the ModifierList are ignored.
Note:
The set of integers returned by `erlang:unique_integer/1` using different sets of `Modifiers` will overlap. For example, by calling `unique_integer([monotonic])`, and `unique_integer([positive, monotonic])` repeatedly, you will eventually see some integers that are returned by both calls.

Failures:

**badarg**
- if `ModifierList` is not a proper list.
- if `Modifier` is not a valid modifier.

```
erlang:universaltime() -> DateTime
Types:
   DateTime = calendar:datetime()
Returns the current date and time according to Universal Time Coordinated (UTC) in the form `{{Year, Month, Day}, {Hour, Minute, Second}}` if supported by the underlying OS. Otherwise `erlang:universaltime()` is equivalent to `erlang:localtime()`. The return value is based on the OS System Time. Example:

   > erlang:universaltime().
   {{1996,11,6},{14,18,43}}
```

```
erlang:universaltime_to_localtime(Universaltime) -> Localtime
Types:
   Localtime = Universaltime = calendar:datetime()
Converts Universal Time Coordinated (UTC) date and time to local date and time in the form `{{Year, Month, Day}, {Hour, Minute, Second}}` if supported by the underlying OS. Otherwise no conversion is done, and `Universaltime` is returned. Example:

   > erlang:universaltime_to_localtime({{1996,11,6},{14,18,43}}).
   {{1996,11,7},{15,18,43}}
```

**badarg** if `Universaltime` denotes an invalid date and time.

```
unlink(Id) -> true
Types:
   Id = pid() | port()
Removes a link between the calling process and another process or a port identified by `Id`. We will from here on call the identified process or port unlinkee.

A link can be set up using the `link/1` BIF. For more information on links and exit signals due to links, see the Processes chapter in the Erlang Reference Manual:
- Links
- Sending Exit Signals
- Receiving Exit Signals
```
Once `unlink(Id)` has returned, it is guaranteed that the link between the caller and the unlinkee has no effect on the caller in the future (unless the link is setup again). Note that if the caller is trapping exits, an `{'EXIT', Id, ExitReason}` message due to the link may have been placed in the message queue of the caller before the `unlink(Id)` call completed. Also note that the `{'EXIT', Id, ExitReason}` message may be the result of the link, but may also be the result of the unlinkee sending the caller an exit signal by calling the `exit/2` BIF. Therefore, it may or may not be appropriate to clean up the message queue after a call to `unlink(Id)` as follows, when trapping exits:

```erlang
unlink(Id),
receive
  {'EXIT', Id, _} ->
  true
  after 0 ->
  true
end
```

The link removal is performed asynchronously. If such a link does not exist, nothing is done. A detailed description of the link protocol can be found in the Distribution Protocol chapter of the ERTS User's Guide.

**Note:**

For some important information about distributed signals, see the Blocking Signaling Over Distribution section in the Processes chapter of the Erlang Reference Manual.

Failure: `badarg` if `Id` does not identify a process or a node local port.

```erlang
unregister(RegName) -> true
```

Types:

- `RegName = atom()`

Removes the registered name `RegName` associated with a process identifier or a port identifier from the name registry. For example:

```erlang
> unregister(db).
true
```

Keep in mind that you can still receive signals associated with the registered name after it has been unregistered as the sender may have looked up the name before sending to it.

Users are advised not to unregister system processes.

Failure: `badarg` if `RegName` is not a registered name.

```erlang
whereis(RegName) -> pid() | port() | undefined
```

Types:

- `RegName = atom()`

Returns the process identifier or port identifier with the registered name `RegName` from the name registry. Returns `undefined` if the name is not registered. Example:

```erlang
> whereis(db).
<0.43.0>
```
erlang:yield() -> true

Tries to give other processes with the same or higher priority (if any) a chance to execute before returning. There is no guarantee that any other process runs between the invocation and return of erlang:yield/0.

See the documentation for receive-after expressions for how to make the current process sleep for a specific number of milliseconds.

**Warning:**

There is seldom or never any need to use this BIF. Using this BIF without a thorough grasp of how the scheduler works can cause performance degradation. The current implementation of this function puts the current process last in the current scheduler's queue for processes of the same priority as the current process.
erlc

Command

The erlc program provides a common way to run all compilers in the Erlang system. Depending on the extension of each input file, erlc invokes the appropriate compiler. Regardless of which compiler is used, the same flags are used to provide parameters, such as include paths and output directory.

The current working directory, ".", is not included in the code path when running the compiler. This is to avoid loading Beam files from the current working directory that could potentially be in conflict with the compiler or the Erlang/OTP system used by the compiler.

Exports

erlc flags file1.ext file2.ext...

Compiles one or more files. The files must include the extension, for example, .erl for Erlang source code, or .yrl for Yecc source code. Erlc uses the extension to invoke the correct compiler.

Generally Useful Flags

The following flags are supported:

-\( \text{-I <Directory}> \)
  Instructs the compiler to search for include files in the Directory. When encountering an \text{--include} or \text{--include_lib} directive, the compiler searches for header files in the following directories:
  • "".", the current working directory of the file server
  • The base name of the compiled file
  • The directories specified using option \text{-I}; the directory specified last is searched first

-\( \text{-o <Directory>} \)
  The directory where the compiler is to place the output files. Defaults to the current working directory.

-\( \text{-D<Name>} \)
  Defines a macro.

-\( \text{-D<Name>=<Value>} \)
  Defines a macro with the specified value. The value can be any Erlang term. Depending on the platform, the value may need to be quoted if the shell itself interprets certain characters. On Unix, terms containing tuples and lists must be quoted. Terms containing spaces must be quoted on all platforms.

-\( \text{-WError} \)
  Makes all warnings into errors.

-\( \text{-W<Number>} \)
  Sets warning level to Number. Defaults to 1. To turn off warnings, use \text{-W0}.

-\( \text{-W} \)
  Same as \text{-W1}. Default.

-\( \text{-v} \)
  Enables verbose output.
-b <Output_type>
   Specifies the type of output file. Output_type is the same as the file extension of the output file, but without
   the period. This option is ignored by compilers that have a single output format.

-no-server
   Do not use the compile server.

-server
   Use the compile server.

-enable-feature <Feature>
   Enables the feature feature during compilation. The special feature all can be used to enable all features.

-disable-feature <feature>
   Disables the feature feature during compilation. The special feature all can be used to disable all non
   permanent features.

-list-features
   List short descriptions of the current configurable features. Non configurable features, i.e., those with a status of
   rejected or permanent will not be shown.

-describe-feature <feature>
   Show long description and history of feature feature.

-M
   Produces a Makefile rule to track header dependencies. The rule is sent to stdout. No object file is produced.

-MMD
   Generate dependencies as a side-effect. The object file will be produced as normal. This option overrides the
   option -M.

-MF <Makefile>
   As option -M, except that the Makefile is written to Makefile. No object file is produced.

-MD
   Same as -M -MF <File>.Pbeam.

-MT <Target>
   In conjunction with option -M or -MF, changes the name of the rule emitted to Target.

-MQ <Target>
   As option -MT, except that characters special to make/1 are quoted.

-MP
   In conjunction with option -M or -MF, adds a phony target for each dependency.

-MG
   In conjunction with option -M or -MF, considers missing headers as generated files and adds them to the
   dependencies.

--
   Signals that no more options will follow. The rest of the arguments is treated as filenames, even if they start
   with hyphens.
A flag starting with a plus (+) rather than a hyphen is converted to an Erlang term and passed unchanged to the compiler. For example, option `export_all` for the Erlang compiler can be specified as follows:

```
erlc +export_all file.erl
```

Depending on the platform, the value may need to be quoted if the shell itself interprets certain characters. On Unix, terms containing tuples and lists must be quoted. Terms containing spaces must be quoted on all platforms.

### Special Flags

The following flags are useful in special situations, such as rebuilding the OTP system:

- `-pa <Directory>`
  
  Appends `Directory` to the front of the code path in the invoked Erlang emulator. This can be used to invoke another compiler than the default one.

- `-pz <Directory>`
  
  Appends `Directory` to the code path in the invoked Erlang emulator.

### Supported Compilers

The following compilers are supported:

- `.erl`
  
  Erlang source code. It generates a `.beam` file.

  Options `-P`, `-E`, and `-S` are equivalent to `+'P'`, `+'E'`, and `+'S'`, except that it is not necessary to include the single quotes to protect them from the shell.

  Supported options: `~I`, `~o`, `~D`, `~v`, `~W`, `~b`.

- `.S`
  
  Erlang assembler source code. It generates a `.beam` file.

  Supported options: same as for `.erl`.

- `.core`
  
  Erlang core source code. It generates a `.beam` file.

  Supported options: same as for `.erl`.

- `.yrl`
  
  Yecc source code. It generates an `.erl` file.

  Use option `~I` with the name of a file to use that file as a customized prologue file (option `includefile`).

  Supported options: `~o`, `~v`, `~I`, `~W`.

- `.mib`
  
  MIB for SNMP. It generates a `.bin` file.

  Supported options: `~I`, `~o`, `~W`.

- `.bin`
  
  A compiled MIB for SNMP. It generates a `.hrl` file.

  Supported options: `~o`, `~v`.
.rel
Script file. It generates a boot file.
Use option -I to name directories to be searched for application files (equivalent to the path in the option list for systools:make_script/2).
Supported option: -o.

.asn1
ASN1 file. It creates an .erl, .hrl, and .asn1db file from an .asn1 file. Also compiles the .erl using the Erlang compiler unless option +noobj is specified.
Supported options: -I, -o, -b, -W.

.idl
IC file. It runs the IDL compiler.
Supported options: -I, -o.

Compile Server

The compile server can be used to potentially speed up the build of multi-file projects by avoiding to start an Erlang system for each file to compile. Whether it will speed up the build depends on the nature of the project and the build machine.

By default, the compile server is not used. It can be enabled by giving erlc the option -server or by setting the environment variable ERLC_USE_SERVER to yes or true.

When the compile server is enabled, erlc will automatically use the server if it is started and start the server if has not already started. The server will terminate itself when it has been idle for some number of seconds.

erlc and the compile server communicate using the Erlang distribution. The compile server is started as a hidden node, with a name that includes the current user. Thus, each user on a computer has their own compile server.

Using the compile server does not always speed up the build, as the compile server sometimes must be restarted to ensure correctness. Here are some examples of situations that force a restart:
  * erlc wants to use a different version of Erlang than the compile server is using.
  * erlc wants to use different options for erl than the compile server was started with. (A change to code path using the option -pa could cause different parse transforms to be loaded. To be safe, the compile server will be restarted when any erl option is changed.)
  * If the current working directory for erlc is different from the working directory active when the compile server was started, and if the compile server has active jobs, it will be restarted as soon as those jobs have finished. (Build systems that build files randomly across multiple directories in parallel will probably not benefit from the compile server.)

Environment Variables

ERLC_EMULATOR
The command for starting the emulator. Defaults to erl in the same directory as the erlc program itself, or, if it does not exist, erl in any of the directories specified in environment variable PATH.

ERLC_USE_SERVER
Allowed values are yes or true to use the compile server, and no or false to not use the compile server. If other values are given, erlc will print a warning message and continue.

ERLC_SERVER_ID
Tells erlc to identify the compile server by the given name, allowing a single user to run multiple unrelated builds in parallel without them affecting each other, which can be useful for shared build machines and the like. The name must be alphanumeric, and it defaults to being empty.
See Also

erl(1), compile(3), yecc(3), snmp(3)
An Erlang driver is a library containing a set of native driver callback functions that the Erlang Virtual Machine calls when certain events occur. There can be multiple instances of a driver, each instance is associated with an Erlang port.

**Warning:**

Use this functionality with extreme care.

A driver callback is executed as a direct extension of the native code of the VM. Execution is not made in a safe environment. The VM cannot provide the same services as provided when executing Erlang code, such as pre-emptive scheduling or memory protection. If the driver callback function does not behave well, the whole VM will misbehave.

- A driver callback that crash will crash the whole VM.
- An erroneously implemented driver callback can cause a VM internal state inconsistency, which can cause a crash of the VM, or miscellaneous misbehaviors of the VM at any point after the call to the driver callback.
- A driver callback doing lengthy work before returning degrades responsiveness of the VM and can cause miscellaneous strange behaviors. Such strange behaviors include, but are not limited to, extreme memory usage and bad load balancing between schedulers. Strange behaviors that can occur because of lengthy work can also vary between Erlang/OTP releases.

As from ERTS 5.5.3 the driver interface has been extended (see extended marker). The extended interface introduces version management, the possibility to pass capability flags (see driver_flags) to the runtime system at driver initialization, and some new driver API functions.

**Note:**

As from ERTS 5.9 old drivers must be recompiled and use the extended interface. They must also be adjusted to the 64-bit capable driver interface.

The driver calls back to the emulator, using the API functions declared in erl_driver.h. They are used for outputting data from the driver, using timers, and so on.

Each driver instance is associated with a port. Every port has a port owner process. Communication with the port is normally done through the port owner process. Most of the functions take the port handle as an argument. This identifies the driver instance. Notice that this port handle must be stored by the driver, it is not given when the driver is called from the emulator (see driver_entry).

Some of the functions take a parameter of type ErlDrvBinary, a driver binary. It is to be both allocated and freed by the caller. Using a binary directly avoids one extra copying of data.

Many of the output functions have a "header buffer", with hbuf and hlen parameters. This buffer is sent as a list before the binary (or list, depending on port mode) that is sent. This is convenient when matching on messages received from the port. (Although in the latest Erlang versions there is the binary syntax, which enables you to match on the beginning of a binary.)

Drivers are locked either on driver level or port level (driver instance level). By default driver level locking will be used, that is, only one emulator thread will execute code in the driver at a time. If port level locking is used, multiple emulator threads can execute code in the driver at the same time. Only one thread at a time will call driver callbacks corresponding to the same port, though. To enable port level locking, set the ERL_DRV_FLAG_USE_PORT_LOCKING driver flag in the driver_entry used by the driver. When port level
locking is used, the driver writer is responsible for synchronizing all accesses to data shared by the ports (driver instances).

Most drivers written before the runtime system with SMP support existed can run in the runtime system with SMP support, without being rewritten, if driver level locking is used.

**Note:**

It is assumed that drivers do not access other drivers. If drivers access each other, they must provide their own mechanism for thread-safe synchronization. Such “inter-driver communication” is strongly discouraged.

**Note:**

Regardless of locking scheme used, calls to driver callbacks can be made from different threads.

Most functions in this API are not thread-safe, that is, they cannot be called from arbitrary threads. Functions that are not documented as thread-safe can only be called from driver callbacks or function calls descending from a driver callback call. Notice that driver callbacks can be called from different threads. This, however, is not a problem for any function in this API, as the emulator has control over these threads.

**Warning:**

Functions not explicitly documented as thread-safe are not thread safe.

A function not explicitly documented as thread-safe can, at some point in time, have a thread-safe implementation in the runtime system. Such an implementation can however change to a thread unsafe implementation at any time without any notice.

Only use functions explicitly documented as thread-safe from arbitrary threads.

As mentioned in the warning text at the beginning of this section, it is of vital importance that a driver callback returns relatively fast. It is difficult to give an exact maximum amount of time that a driver callback is allowed to work, but usually a well-behaving driver callback is to return within 1 millisecond. This can be achieved using different approaches. If you have full control over the code to execute in the driver callback, the best approach is to divide the work into multiple chunks of work, and trigger multiple calls to the time-out callback using zero time-outs. Function `erl_drv_consume_timeslice` can be useful to determine when to trigger such time-out callback calls. However, sometimes it cannot be implemented this way, for example when calling third-party libraries. In this case, you typically want to dispatch the work to another thread. Information about thread primitives is provided below.

**Functionality**

All functions that a driver needs to do with Erlang are performed through driver API functions. Functions exist for the following functionality:

**Timer functions**

Control the timer that a driver can use. The timer has the emulator call the `timeout` entry function after a specified time. Only one timer is available for each driver instance.

**Queue handling**

Every driver instance has an associated queue. This queue is a `SysIOVec`, which works as a buffer. It is mostly used for the driver to buffer data that is to be written to a device, it is a byte stream. If the port owner process closes the driver, and the queue is not empty, the driver is not closed. This enables the driver to flush its buffers before closing.
The queue can be manipulated from any threads if a port data lock is used. For more information, see ErlDrvPDL.

Output functions

With these functions, the driver sends data back to the emulator. The data is received as messages by the port owner process, see erlang:open_port/2. The vector function and the function taking a driver binary are faster, as they avoid copying the data buffer. There is also a fast way of sending terms from the driver, without going through the binary term format.

Failure

The driver can exit and signal errors up to Erlang. This is only for severe errors, when the driver cannot possibly keep open.

Asynchronous calls

Erlang/OTP R7B and later versions have provision for asynchronous function calls, using a thread pool provided by Erlang. There is also a select call, which can be used for asynchronous drivers.

Multi-threading

A POSIX thread like API for multi-threading is provided. The Erlang driver thread API only provides a subset of the functionality provided by the POSIX thread API. The subset provided is more or less the basic functionality needed for multi-threaded programming:

- Threads
- Mutexes
- Condition variables
- Read/write locks
- Thread-specific data

The Erlang driver thread API can be used in conjunction with the POSIX thread API on UNIX-ices and with the Windows native thread API on Windows. The Erlang driver thread API has the advantage of being portable, but there can exist situations where you want to use functionality from the POSIX thread API or the Windows native thread API.

The Erlang driver thread API only returns error codes when it is reasonable to recover from an error condition. If it is not reasonable to recover from an error condition, the whole runtime system is terminated. For example, if a create mutex operation fails, an error code is returned, but if a lock operation on a mutex fails, the whole runtime system is terminated.

Notice that there is no "condition variable wait with time-out" in the Erlang driver thread API. This because of issues with pthread_cond_timedwait. When the system clock suddenly is changed, it is not always guaranteed that you will wake up from the call as expected. An Erlang runtime system must be able to cope with sudden changes of the system clock. Therefore, we have omitted it from the Erlang driver thread API. In the Erlang driver case, time-outs can and are to be handled with the timer functionality of the Erlang driver API.

Notice that many functions in the Erlang driver API are not thread-safe. If a function is not documented as thread-safe, it is not thread-safe.

Note:

When executing in an emulator thread, it is very important that you unlock all locks you have locked before letting the thread out of your control; otherwise you are very likely to deadlock the whole emulator.

If you need to use thread-specific data in an emulator thread, only have the thread-specific data set while the thread is under your control, and clear the thread-specific data before you let the thread out of your control.
In the future, debug functionality will probably be integrated with the Erlang driver thread API. All functions that create entities take a name argument. Currently the name argument is unused, but it will be used when the debug functionality is implemented. If you name all entities created well, the debug functionality will be able to give you better error reports.

Adding/removing drivers

A driver can add and later remove drivers.

Monitoring processes

A driver can monitor a process that does not own a port.

Version management

Version management is enabled for drivers that have set the extended_marker field of their driver_entry to ERL_DRV_EXTENDED_MARKER. erl_driver.h defines:

- ERL_DRV_EXTENDED_MARKER
- ERL_DRV_EXTENDED_MAJOR_VERSION, which is incremented when driver incompatible changes are made to the Erlang runtime system. Normally it suffices to recompile drivers when ERL_DRV_EXTENDED_MAJOR_VERSION has changed, but it can, under rare circumstances, mean that drivers must be slightly modified. If so, this will of course be documented.
- ERL_DRV_EXTENDED_MINOR_VERSION, which is incremented when new features are added. The runtime system uses the minor version of the driver to determine what features to use.

The runtime system normally refuses to load a driver if the major versions differ, or if the major versions are equal and the minor version used by the driver is greater than the one used by the runtime system. Old drivers with lower major versions are however allowed after a bump of the major version during a transition period of two major releases. Such old drivers can, however, fail if deprecated features are used.

The emulator refuses to load a driver that does not use the extended driver interface, to allow for 64-bit capable drivers, as incompatible type changes for the callbacks output, control, and call were introduced in Erlang/OTP R15B. A driver written with the old types would compile with warnings and when called return garbage sizes to the emulator, causing it to read random memory and create huge incorrect result blobs.

Therefore it is not enough to only recompile drivers written with version management for pre R15B types; the types must be changed in the driver suggesting other rewrites, especially regarding size variables. Investigate all warnings when recompiling.

Also, the API driver functions driver_output*, driver_vec_to_buf, driver_alloc/realloc*, and the driver_* queue functions were changed to have larger length arguments and return values. This is a lesser problem, as code that passes smaller types gets them auto-converted in the calls, and as long as the driver does not handle sizes that overflow an int, all will work as before.

Time measurement

Support for time measurement in drivers:

- ErlDrvTime
- ErlDrvTimeUnit
- erl_drv_monotonic_time
- erl_drv_time_offset
- erl_drv_convert_time_unit

Rewrites for 64-Bit Driver Interface

ERTS 5.9 introduced two new integer types, ErlDrvSizeT and ErlDrvSSizeT, which can hold 64-bit sizes if necessary.
To not update a driver and only recompile, it probably works when building for a 32-bit machine creating a false sense of security. Hopefully that will generate many important warnings. But when recompiling the same driver later on for a 64-bit machine, there will be warnings and almost certainly crashes. So it is a bad idea to postpone updating the driver and not fixing the warnings.

When recompiling with gcc, use flag `-Wstrict-prototypes` to get better warnings. Try to find a similar flag if you use another compiler.

The following is a checklist for rewriting a pre ERTS 5.9 driver, most important first:

**Return types for driver callbacks**

- Rewrite driver callback `control` to use return type `ErlDrvSSizeT` instead of `int`.
- Rewrite driver callback `call` to use return type `ErlDrvSSizeT` instead of `int`.

<table>
<thead>
<tr>
<th>Note:</th>
</tr>
</thead>
<tbody>
<tr>
<td>These changes are essential not to crash the emulator or worse cause malfunction. Without them a driver can return garbage in the high 32 bits to the emulator, causing it to build a huge result from random bytes, either crashing on memory allocation or succeeding with a random result from the driver call.</td>
</tr>
</tbody>
</table>

**Arguments to driver callbacks**

- Driver callback `output` now gets `ErlDrvSizeT` as 3rd argument instead of previously `int`.
- Driver callback `control` now gets `ErlDrvSizeT` as 4th and 6th arguments instead of previously `int`.
- Driver callback `call` now gets `ErlDrvSizeT` as 4th and 6th arguments instead of previously `int`.

Sane compiler's calling conventions probably make these changes necessary only for a driver to handle data chunks that require 64-bit size fields (mostly larger than 2 GB, as that is what an `int` of 32 bits can hold). But it is possible to think of non-sane calling conventions that would make the driver callbacks mix up the arguments causing malfunction.

<table>
<thead>
<tr>
<th>Note:</th>
</tr>
</thead>
<tbody>
<tr>
<td>The argument type change is from signed to unsigned. This can cause problems for, for example, loop termination conditions or error conditions if you only change the types all over the place.</td>
</tr>
</tbody>
</table>

**Larger size field in ErlIOVec**

- The `size` field in `ErlIOVec` has been changed to `ErlDrvSizeT` from `int`. Check all code that use that field.
- Automatic type-casting probably makes these changes necessary only for a driver that encounters sizes > 32 bits.

<table>
<thead>
<tr>
<th>Note:</th>
</tr>
</thead>
<tbody>
<tr>
<td>The <code>size</code> field changed from signed to unsigned. This can cause problems for, for example, loop termination conditions or error conditions if you only change the types all over the place.</td>
</tr>
</tbody>
</table>

**Arguments and return values in the driver API**

- Many driver API functions have changed argument type and/or return value to `ErlDrvSizeT` from mostly `int`.
- Automatic type-casting probably makes these changes necessary only for a driver that encounters sizes > 32 bits.

**driver_output**

- 3rd argument
driver_output2
  3rd and 5th arguments
driver_output_binary
  3rd, 5th, and 6th arguments
driver_outputv
  3rd and 5th arguments
driver_vec_to_buf
  3rd argument and return value
driver_alloc
  1st argument
driver_realloc
  2nd argument
driver_alloc_binary
  1st argument
driver_realloc_binary
  2nd argument
driver_enq
  3rd argument
driver_pushq
  3rd argument
driver_deq
  2nd argument and return value
driver_sizeq
  Return value
driver_enq_bin
  3rd and 4th arguments
driver_pushq_bin
  3rd and 4th arguments
driver_enqv
  3rd argument
driver_pushqv
  3rd argument
driver_peekqv
  Return value

Note:
This is a change from signed to unsigned. This can cause problems for, for example, loop termination conditions and error conditions if you only change the types all over the place.

Data Types

ErlDrvSizeT
  An unsigned integer type to be used as size_t.
ErlDrvSSizeT
  A signed integer type, the size of ErlDrvSizeT.
ErlDrvSysInfo

typedef struct ErlDrvSysInfo {
    int driver_major_version;
    int driver_minor_version;
    char *erts_version;
    char *otp_release;
    int thread_support;
    int smp_support;
    int async_threads;
    int scheduler_threads;
    int nif_major_version;
    int nif_minor_version;
    int dirty_scheduler_support;
} ErlDrvSysInfo;

The ErlDrvSysInfo structure is used for storage of information about the Erlang runtime system. driver_system_info writes the system information when passed a reference to an ErlDrvSysInfo structure. The fields in the structure are as follows:

driver_major_version

The value of ERL_DRV_EXTENDED_MAJOR_VERSION when the runtime system was compiled. This value is the same as the value of ERL_DRV_EXTENDED_MAJOR_VERSION used when compiling the driver; otherwise the runtime system would have refused to load the driver.

driver_minor_version

The value of ERL_DRV_EXTENDED_MINOR_VERSION when the runtime system was compiled. This value can differ from the value of ERL_DRV_EXTENDED_MINOR_VERSION used when compiling the driver.

erts_version

A string containing the version number of the runtime system (the same as returned by erlang:system_info(version)).

otp_release

A string containing the OTP release number (the same as returned by erlang:system_info(otp_release)).

thread_support

A value != 0 if the runtime system has thread support; otherwise 0.

smp_support

A value != 0 if the runtime system has SMP support; otherwise 0.

async_threads

The number of async threads in the async thread pool used by driver_async (the same as returned by erlang:system_info(thread_pool_size)).

scheduler_threads

The number of scheduler threads used by the runtime system (the same as returned by erlang:system_info(schedulers)).

nif_major_version

The value of ERL_NIF_MAJOR_VERSION when the runtime system was compiled.

nif_minor_version

The value of ERL_NIF_MINOR_VERSION when the runtime system was compiled.
dirty_scheduler_support

A value != 0 if the runtime system has support for dirty scheduler threads; otherwise 0.

ErlDrvBinary

typedef struct ErlDrvBinary {
    ErlDrvSint orig_size;
    char orig_bytes[];
} ErlDrvBinary;

The ErlDrvBinary structure is a binary, as sent between the emulator and the driver. All binaries are reference counted; when driver_binary_free is called, the reference count is decremented, when it reaches zero, the binary is deallocated. orig_size is the binary size and orig_bytes is the buffer. ErlDrvBinary has not a fixed size, its size is orig_size + 2 * sizeof(int).

Note:

The refc field has been removed. The reference count of an ErlDrvBinary is now stored elsewhere. The reference count of an ErlDrvBinary can be accessed through driver_binary_get_refc, driver_binary_inc_refc, and driver_binary_dec_refc.

Some driver calls, such as driver_enq_binary, increment the driver reference count, and others, such as driver_deq decrement it.

Using a driver binary instead of a normal buffer is often faster, as the emulator needs not to copy the data, only the pointer is used.

A driver binary allocated in the driver, with driver_alloc_binary, is to be freed in the driver (unless otherwise stated) with driver_free_binary. (Notice that this does not necessarily deallocate it, if the driver is still referred in the emulator, the ref-count will not go to zero.)

Driver binaries are used in the driver_output2 and driver_outputv calls, and in the queue. Also the driver callback outputv uses driver binaries.

If the driver for some reason wants to keep a driver binary around, for example in a static variable, the reference count is to be incremented, and the binary can later be freed in the stop callback, with driver_free_binary.

Notice that as a driver binary is shared by the driver and the emulator. A binary received from the emulator or sent to the emulator must not be changed by the driver.

Since ERTS 5.5 (Erlang/OTP R11B), orig_bytes is guaranteed to be properly aligned for storage of an array of doubles (usually 8-byte aligned).

ErlDrvData

A handle to driver-specific data, passed to the driver callbacks. It is a pointer, and is most often type cast to a specific pointer in the driver.

SysIOVec

A system I/O vector, as used by writev on Unix and WSASend on Win32. It is used in ErlIOVec.

ErlIOVec

typedef struct ErlIOVec {
    int vsize;
    ErlDrvSizeT size;
    SysIOVec* iov;
    ErlDrvBinary** binv;
} ErlIOVec;
The I/O vector used by the emulator and drivers is a list of binaries, with a SysIOVec pointing to the buffers of the binaries. It is used in driver_outputv and the outputv driver callback. Also, the driver queue is an ErlIOVec.

**ErlDrvMonitor**

When a driver creates a monitor for a process, a ErlDrvMonitor is filled in. This is an opaque data type that can be assigned to, but not compared without using the supplied compare function (that is, it behaves like a struct).

The driver writer is to provide the memory for storing the monitor when calling driver_monitor_process. The address of the data is not stored outside of the driver, so ErlDrvMonitor can be used as any other data, it can be copied, moved in memory, forgotten, and so on.

**ErlDrvNowData**

The ErlDrvNowData structure holds a time stamp consisting of three values measured from some arbitrary point in the past. The three structure members are:

- **megasecs**
  - The number of whole megaseconds elapsed since the arbitrary point in time
- **secs**
  - The number of whole seconds elapsed since the arbitrary point in time
- **microsecs**
  - The number of whole microseconds elapsed since the arbitrary point in time

**ErlDrvPDL**

If certain port-specific data must be accessed from other threads than those calling the driver callbacks, a port data lock can be used to synchronize the operations on the data. Currently, the only port-specific data that the emulator associates with the port data lock is the driver queue.

Normally a driver instance has no port data lock. If the driver instance wants to use a port data lock, it must create the port data lock by calling driver_pdl_create.

*Note:*

Once the port data lock has been created, every access to data associated with the port data lock must be done while the port data lock is locked. The port data lock is locked and unlocked by driver_pdl_lock, and driver_pdl_unlock, respectively.

A port data lock is reference counted, and when the reference count reaches zero, it is destroyed. The emulator at least increments the reference count once when the lock is created and decrements it once the port associated with the lock terminates. The emulator also increments the reference count when an async job is enqueued and decrements it when an async job has been invoked. Also, the driver is responsible for ensuring that the reference count does not reach zero before the last use of the lock by the driver has been made. The reference count can be read, incremented, and decremented by driver_pdl_get_refc, driver_pdl_inc_refc, and driver_pdl_dec_refc, respectively.

**ErlDrvTid**

Thread identifier.

See also erl_drv_thread_create, erl_drv_thread_exit, erl_drv_thread_join, erl_drv_thread_self, and erl_drv_equal_tids.

**ErlDrvThreadOpts**

```c
int suggested_stack_size;
```

Thread options structure passed to erl_drv_thread_create. The following field exists:
suggested_stack_size

A suggestion, in kilowords, on how large a stack to use. A value < 0 means default size.

See also erl_drv_thread_opts_create, erl_drv_thread_opts_destroy, and erl_drv_thread_create.

ErlDrvMutex

Mutual exclusion lock. Used for synchronizing access to shared data. Only one thread at a time can lock a mutex.

See also erl_drv_mutex_create, erl_drv_mutex_destroy, erl_drv_mutex_lock, erl_drv_mutex_trylock, and erl_drv_mutex_unlock.

ErlDrvCond

Condition variable. Used when threads must wait for a specific condition to appear before continuing execution. Condition variables must be used with associated mutexes.

See also erl_drv_cond_create, erl_drv_cond_destroy, erl_drv_cond_signal, erl_drv_cond_broadcast, and erl_drv_cond_wait.

ErlDrvRWLock

Read/write lock. Used to allow multiple threads to read shared data while only allowing one thread to write the same data. Multiple threads can read lock an rwlock at the same time, while only one thread can read/write lock an rwlock at a time.

See also erlDrv_rwlock_create, erl_drv_rwlock_destroy, erl_drv_rwlock_rwlock, erl_drv_rwlock_tryrlock, erl_drv_rwlock_runlock, erl_drv_rwlock_rwlock, erl_drv_rwlock_tryrwlock, and erl_drv_rwlock_rwunlock.

ErlDrvTSDKey

Key that thread-specific data can be associated with.

See also erl_drv_tsd_key_create, erl_drv_tsd_key_destroy, erl_drv_tsd_set, and erl_drv_tsd_get.

ErlDrvTime

A signed 64-bit integer type for time representation.

ErlDrvTimeUnit

An enumeration of time units supported by the driver API:

- ERL_DRV_SEC  Seconds
- ERL_DRV_MSEC  Milliseconds
- ERL_DRV_USEC  Microseconds
- ERL_DRV_NSEC  Nanoseconds

Exports

void add_driver_entry(ErlDrvEntry *de)

Adds a driver entry to the list of drivers known by Erlang. The init function of parameter de is called.
**Note:**

To use this function for adding drivers residing in dynamically loaded code is dangerous. If the driver code for the added driver resides in the same dynamically loaded module (that is, .so file) as a normal dynamically loaded driver (loaded with the erl_ddll interface), the caller is to call `driver_lock_driver` before adding driver entries.

Use of this function is generally deprecated.

```c
void *driver_alloc(ErlDrvSizeT size)
```

Allocates a memory block of the size specified in `size`, and returns it. This fails only on out of memory, in which case `NULL` is returned. (This is most often a wrapper for `malloc`).

Memory allocated must be explicitly freed with a corresponding call to `driver_free` (unless otherwise stated).

This function is thread-safe.

```c
ErlDrvBinary *driver_alloc_binary(ErlDrvSizeT size)
```

Allocates a driver binary with a memory block of at least `size` bytes, and returns a pointer to it, or `NULL` on failure (out of memory). When a driver binary has been sent to the emulator, it must not be changed. Every allocated binary is to be freed by a corresponding call to `driver_free_binary` (unless otherwise stated).

Notice that a driver binary has an internal reference counter. This means that calling `driver_free_binary`, it may not actually dispose of it. If it is sent to the emulator, it can be referenced there.

The driver binary has a field, `orig_bytes`, which marks the start of the data in the binary.

This function is thread-safe.

```c
long driver_async(ErlDrvPort port, unsigned int* key, void (*async_invoke)(void*), void* async_data, void (*async_free)(void*))
```

Performs an asynchronous call. The function `async_invoke` is invoked in a thread separate from the emulator thread. This enables the driver to perform time-consuming, blocking operations without blocking the emulator.

The async thread pool size can be set with command-line argument `+A` in `erl(1)`. If an async thread pool is unavailable, the call is made synchronously in the thread calling `driver_async`. The current number of async threads in the async thread pool can be retrieved through `driver_system_info`.

If a thread pool is available, a thread is used. If argument `key` is `NULL`, the threads from the pool are used in a round-robin way, each call to `driver_async` uses the next thread in the pool. With argument `key` set, this behavior is changed. The two same values of `*key` always get the same thread.

To ensure that a driver instance always uses the same thread, the following call can be used:

```c
unsigned int myKey = driver_async_port_key(myPort);

r = driver_async(myPort, &myKey, myData, myFunc);
```

It is enough to initialize `myKey` once for each driver instance.

If a thread is already working, the calls are queued up and executed in order. Using the same thread for each driver instance ensures that the calls are made in sequence.

The `async_data` is the argument to the functions `async_invoke` and `async_free`. It is typically a pointer to a structure containing a pipe or event that can be used to signal that the async operation completed. The data is to be freed in `async_free`. 

352 | Ericsson AB. All Rights Reserved.: Erlang Run-Time System Application (ERTS)
When the async operation is done, ready_async driver entry function is called. If ready_async is NULL in the driver entry, the async_free function is called instead.

The return value is -1 if the driver_async call fails.

**Note:**

As from ERTS 5.5.4.3 the default stack size for threads in the async-thread pool is 16 kilowords, that is, 64 kilobyte on 32-bit architectures. This small default size has been chosen because the amount of async-threads can be quite large. The default stack size is enough for drivers delivered with Erlang/OTP, but is possibly not sufficiently large for other dynamically linked-in drivers that use the driver_async functionality. A suggested stack size for threads in the async-thread pool can be configured through command-line argument +a in erl(1).

```c
unsigned int driver_async_port_key(ErlDrvPort port)
```

Calculates a key for later use in driver_async. The keys are evenly distributed so that a fair mapping between port IDs and async thread IDs is achieved.

**Note:**

Before Erlang/OTP R16, the port ID could be used as a key with proper casting, but after the rewrite of the port subsystem, this is no longer the case. With this function, you can achieve the same distribution based on port IDs as before Erlang/OTP R16.

```c
long driver_binary_dec_refc(ErlDrvBinary *bin)
```

Decrements the reference count on bin and returns the reference count reached after the decrement.

This function is thread-safe.

**Note:**

The reference count of driver binary is normally to be decremented by calling driver_free_binary. driver_binary_dec_refc does not free the binary if the reference count reaches zero. Only use driver_binary_dec_refc when you are sure not to reach a reference count of zero.

```c
long driver_binary_get_refc(ErlDrvBinary *bin)
```

Returns the current reference count on bin. This function is thread-safe.

```c
long driver_binary_inc_refc(ErlDrvBinary *bin)
```

Increments the reference count on bin and returns the reference count reached after the increment. This function is thread-safe.

```
ErlDrvTermData driver_caller(ErlDrvPort port)
```

Returns the process ID of the process that made the current call to the driver. The process ID can be used with driver_send_term to send back data to the caller. driver_caller only returns valid data when currently executing in one of the following driver callbacks:
start
   Called from erlang:open_port/2.
output
   Called from erlang:send/2 and erlang:port_command/2.
outputv
   Called from erlang:send/2 and erlang:port_command/2.
control
   Called from erlang:port_control/3.
call
   Called from erlang:port_call/3.

Notice that this function is not thread-safe.

int driver_cancel_timer(ErlDrvPort port)
Cancels a timer set with driver_set_timer.
The return value is 0.

int driver_compare_monitors(const ErlDrvMonitor *monitor1, const
ErlDrvMonitor *monitor2)
Compares two ErlDrvMonitors. Can also be used to imply some artificial order on monitors, for whatever reason.
Returns 0 if monitor1 and monitor2 are equal, < 0 if monitor1 < monitor2, and > 0 if monitor1 > monitor2.

ErlDrvTermData driver_connected(ErlDrvPort port)
Returns the port owner process.
Notice that this function is not thread-safe.

ErlDrvPort driver_create_port(ErlDrvPort port, ErlDrvTermData owner_pid,
char* name, ErlDrvData drv_data)
Creates a new port executing the same driver code as the port creating the new port.

port
   The port handle of the port (driver instance) creating the new port.
owner_pid
   The process ID of the Erlang process to become owner of the new port. This process will be linked to the new
   port. You usually want to use driver_caller(port) as owner_pid.
name
   The port name of the new port. You usually want to use the same port name as the driver name (driver_name
   field of the driver_entry).
drv_data
   The driver-defined handle that is passed in later calls to driver callbacks. Notice that the driver start callback is
   not called for this new driver instance. The driver-defined handle is normally created in the driver start callback
   when a port is created through erlang:open_port/2.

The caller of driver_create_port is allowed to manipulate the newly created port when
driver_create_port has returned. When port level locking is used, the creating port is only allowed to
manipulate the newly created port until the current driver callback, which was called by the emulator, returns.
int driver_demonitor_process(ErlDrvPort port, const ErlDrvMonitor *monitor)
Cancels a monitor created earlier.
Returns 0 if a monitor was removed and > 0 if the monitor no longer exists.

ErlDrvSizeT driver_deq(ErlDrvPort port, ErlDrvSizeT size)
Dequeues data by moving the head pointer forward in the driver queue by size bytes. The data in the queue is deallocated.
Returns the number of bytes remaining in the queue on success, otherwise -1.
This function can be called from any thread if a port data lock associated with the port is locked by the calling thread during the call.

int driver_enq(ErlDrvPort port, char* buf, ErlDrvSizeT len)
Enqueues data in the driver queue. The data in buf is copied (len bytes) and placed at the end of the driver queue. The driver queue is normally used in a FIFO way.
The driver queue is available to queue output from the emulator to the driver (data from the driver to the emulator is queued by the emulator in normal Erlang message queues). This can be useful if the driver must wait for slow devices, and so on, and wants to yield back to the emulator. The driver queue is implemented as an ErlIOVec.
When the queue contains data, the driver does not close until the queue is empty.
The return value is 0.
This function can be called from any thread if a port data lock associated with the port is locked by the calling thread during the call.

int driver_enq_bin(ErlDrvPort port, ErlDrvBinary *bin, ErlDrvSizeT offset, ErlDrvSizeT len)
Enqueues a driver binary in the driver queue. The data in bin at offset with length len is placed at the end of the queue. This function is most often faster than driver_enq, because no data must be copied.
This function can be called from any thread if a port data lock associated with the port is locked by the calling thread during the call.
The return value is 0.

int driver_enqv(ErlDrvPort port, ErlIOVec *ev, ErlDrvSizeT skip)
Enqueues the data in ev, skipping the first skip bytes of it, at the end of the driver queue. It is faster than driver_enq, because no data must be copied.
The return value is 0.
This function can be called from any thread if a port data lock associated with the port is locked by the calling thread during the call.

int driver_failure(ErlDrvPort port, int error)
int driver_failure_atom(ErlDrvPort port, char *string)
int driver_failure_posix(ErlDrvPort port, int error)
Signals to Erlang that the driver has encountered an error and is to be closed. The port is closed and the tuple {'EXIT', error, Err} is sent to the port owner process, where error is an error atom (driver_failure_atom and driver_failure_posix) or an integer (driver_failure).
The driver is to fail only when in severe error situations, when the driver cannot possibly keep open, for example, buffer allocation gets out of memory. For normal errors it is more appropriate to send error codes with driver_output.

The return value is 0.

```c
int driver_failure_eof(ErlDrvPort port)
```
Signals to Erlang that the driver has encountered an EOF and is to be closed, unless the port was opened with option `eof`, in which case `eof` is sent to the port. Otherwise the port is closed and an 'EXIT' message is sent to the port owner process.

The return value is 0.

```c
void driver_free(void *ptr)
```
Frees the memory pointed to by `ptr`. The memory is to have been allocated with `driver_alloc`. All allocated memory is to be deallocated, only once. There is no garbage collection in drivers.

This function is thread-safe.

```c
void driver_free_binary(ErlDrvBinary *bin)
```
Frees a driver binary `bin`, allocated previously with `driver_alloc_binary`. As binaries in Erlang are reference counted, the binary can still be around.

This function is thread-safe.

```c
ErlDrvTermData driver_get_monitored_process(ErlDrvPort port, const ErlDrvMonitor *monitor)
```
Returns the process ID associated with a living monitor. It can be used in the `process_exit` callback to get the process identification for the exiting process.

Returns `driver_term_nil` if the monitor no longer exists.

```c
int driver_get_now(ErlDrvNowData *now)
```

**Warning:**

This function is deprecated. Do not use it. Use `erl_drv_monotonic_time` (perhaps in combination with `erl_drv_time_offset`) instead.

Reads a time stamp into the memory pointed to by parameter `now`. For information about specific fields, see `ErlDrvNowData`.

The return value is 0, unless the `now` pointer is invalid, in which case it is < 0.

```c
int driver_lock_driver(ErlDrvPort port)
```
Locks the driver used by the port `port` in memory for the rest of the emulator process' lifetime. After this call, the driver behaves as one of Erlang's statically linked-in drivers.

```c
ErlDrvTermData driver_mk_atom(char* string)
```
Returns an atom given a name `string`. The atom is created and does not change, so the return value can be saved and reused, which is faster than looking up the atom several times.

Notice that this function is not thread-safe.
ErlDrvTermData driver_mk_port(ErlDrvPort port)
Converts a port handle to the Erlang term format, usable in erl_drv_output_term and erl_drv_send_term.
Notice that this function is not thread-safe.

int driver_monitor_process(ErlDrvPort port, ErlDrvTermData process, ErlDrvMonitor *monitor)
Starts monitoring a process from a driver. When a process is monitored, a process exit results in a call to the provided process_exit callback in the ErlDrvEntry structure. The ErlDrvMonitor structure is filled in, for later removal or compare.
Parameter process is to be the return value of an earlier call to driver_caller or driver_connected call.
Returns 0 on success, < 0 if no callback is provided, and > 0 if the process is no longer alive.

int driver_output(ErlDrvPort port, char *buf, ErlDrvSizeT len)
Sends data from the driver up to the emulator. The data is received as terms or binary data, depending on how the driver port was opened.
The data is queued in the port owner process’ message queue. Notice that this does not yield to the emulator (as the driver and the emulator run in the same thread).
Parameter buf points to the data to send, and len is the number of bytes.
The return value for all output functions is 0 for normal use. If the driver is used for distribution, it can fail and return -1.

int driver_output_binary(ErlDrvPort port, char *hbuf, ErlDrvSizeT hlen, ErlDrvBinary* bin, ErlDrvSizeT offset, ErlDrvSizeT len)
Sends data to a port owner process from a driver binary. It has a header buffer (hbuf and hlen) just like driver_output2. Parameter hbuf can be NULL.
Parameter offset is an offset into the binary and len is the number of bytes to send.
Driver binaries are created with driver_alloc_binary.
The data in the header is sent as a list and the binary as an Erlang binary in the tail of the list.
For example, if hlen is 2, the port owner process receives [H1, H2 | <<T>>].
The return value is 0 for normal use.
Notice that, using the binary syntax in Erlang, the driver application can match the header directly from the binary, so the header can be put in the binary, and hlen can be set to 0.

int driver_output_term(ErlDrvPort port, ErlDrvTermData* term, int n)

**Warning:**
This function is deprecated. Use erl_drv_output_term instead.

Parameters term and n work as in erl_drv_output_term.
Notice that this function is not thread-safe.
int driver_output2(ErlDrvPort port, char *hbuf, ErlDrvSizeT hlen, char *buf, ErlDrvSizeT len)

First sends hbuf (length in hlen) data as a list, regardless of port settings. Then sends buf as a binary or list. For example, if hlen is 3, the port owner process receives \([\text{H1}, \text{H2}, \text{H3} | T]\).

The point of sending data as a list header, is to facilitate matching on the data received.

The return value is 0 for normal use.

int driver_outputv(ErlDrvPort port, char* hbuf, ErlDrvSizeT hlen, ErlIOVec *ev, ErlDrvSizeT skip)

Sends data from an I/O vector, ev, to the port owner process. It has a header buffer (hbuf and hlen), just like driver_output2.

Parameter skip is a number of bytes to skip of the ev vector from the head.

You get vectors of ErlIOVec type from the driver queue (see below), and the outputv driver entry function. You can also make them yourself, if you want to send several ErlDrvBinary buffers at once. Often it is faster to use driver_output or.

For example, if hlen is 2 and ev points to an array of three binaries, the port owner process receives \([\text{H1}, \text{H2}, \text{<<B1>>}, \text{<<B2>>} | \text{<<B3>>}]\).

The return value is 0 for normal use.

The comment for driver_output_binary also applies for driver_outputv.

ErlDrvPDL driver_pdl_create(ErlDrvPort port)

Creates a port data lock associated with the port.

Note:

Once a port data lock has been created, it must be locked during all operations on the driver queue of the port.

Returns a newly created port data lock on success, otherwise NULL. The function fails if port is invalid or if a port data lock already has been associated with the port.

long driver_pdl_dec_refc(ErlDrvPDL pdl)

Decrements the reference count of the port data lock passed as argument (pdl).

The current reference count after the decrement has been performed is returned.

This function is thread-safe.

long driver_pdl_get_refc(ErlDrvPDL pdl)

Returns the current reference count of the port data lock passed as argument (pdl).

This function is thread-safe.

long driver_pdl_inc_refc(ErlDrvPDL pdl)

Increments the reference count of the port data lock passed as argument (pdl).

The current reference count after the increment has been performed is returned.

This function is thread-safe.
void driver_pdl_lock(ErlDrvPDL pdl)
Locks the port data lock passed as argument (pdl).
This function is thread-safe.

void driver_pdl_unlock(ErlDrvPDL pdl)
Unlocks the port data lock passed as argument (pdl).
This function is thread-safe.

SysIOVec *driver.peekq(ErlDrvPort port, int *vlen)
Retrieves the driver queue as a pointer to an array of SysIOVecs. It also returns the number of elements in vlen. This is one of two ways to get data out of the queue.
Nothing is removed from the queue by this function, that must be done with driver_deq.
The returned array is suitable to use with the Unix system call writev.
This function can be called from any thread if a port data lock associated with the port is locked by the calling thread during the call.

ErlDrvSizeT driver_peekqv(ErlDrvPort port, ErlIOVec *ev)
Retrieves the driver queue into a supplied ErlIOVec ev. It also returns the queue size. This is one of two ways to get data out of the queue.
If ev is NULL, all ones that is -1 type cast to ErlDrvSizeT are returned.
Nothing is removed from the queue by this function, that must be done with driver_deq.
This function can be called from any thread if a port data lock associated with the port is locked by the calling thread during the call.

int driver_pushq(ErlDrvPort port, char* buf, ErlDrvSizeT len)
Puts data at the head of the driver queue. The data in buf is copied (len bytes) and placed at the beginning of the queue.
The return value is 0.
This function can be called from any thread if a port data lock associated with the port is locked by the calling thread during the call.

int driver_pushq_bin(ErlDrvPort port, ErlDrvBinary *bin, ErlDrvSizeT offset, ErlDrvSizeT len)
Puts data in the binary bin, at offset with length len at the head of the driver queue. It is most often faster than driver_pushq, because no data must be copied.
This function can be called from any thread if a port data lock associated with the port is locked by the calling thread during the call.
The return value is 0.

int driver_pushqv(ErlDrvPort port, ErlIOVec *ev, ErlDrvSizeT skip)
Puts the data in ev, skipping the first skip bytes of it, at the head of the driver queue. It is faster than driver_pushq, because no data must be copied.
The return value is 0.

This function can be called from any thread if a port data lock associated with the port is locked by the calling thread during the call.

```c
int driver_read_timer(ErlDrvPort port, unsigned long *time_left)
```

Reads the current time of a timer, and places the result in `time_left`. This is the time in milliseconds, before the time-out occurs.

The return value is 0.

```c
void *driver_realloc(void *ptr, ErlDrvSizeT size)
```

Resizes a memory block, either in place, or by allocating a new block, copying the data, and freeing the old block. A pointer is returned to the reallocated memory. On failure (out of memory), NULL is returned. (This is most often a wrapper for `realloc`.)

This function is thread-safe.

```c
ErlDrvBinary *driver_realloc_binary(ErlDrvBinary *bin, ErlDrvSizeT size)
```

Resizes a driver binary, while keeping the data.

Returns the resized driver binary on success. Returns NULL on failure (out of memory).

This function is thread-safe.

```c
int driver_select(ErlDrvPort port, ErlDrvEvent event, int mode, int on)
```

This function is used by drivers to provide the emulator with events to check for. This enables the emulator to call the driver when something has occurred asynchronously.

Parameter `event` identifies an OS-specific event object. On Unix systems, the functions `select/poll` are used. The event object must be a socket or pipe (or other object that `select/poll` can use). On Windows, the Win32 API function `WaitForMultipleObjects` is used. This places other restrictions on the event object; see the Win32 SDK documentation.

Parameter `on` is to be 1 for setting events and 0 for clearing them.

Parameter `mode` is a bitwise OR combination of `ERL_DRV_READ`, `ERL_DRV_WRITE`, and `ERL_DRV_USE`. The first two specify whether to wait for read events and/or write events. A fired read event calls `ready_input` and a fired write event calls `ready_output`.

**Note:**

Some OS (Windows) do not differentiate between read and write events. The callback for a fired event then only depends on the value of `mode`.

`ERL_DRV_USE` specifies if we are using the event object or if we want to close it. It is not safe to clear all events and then close the event object after `driver_select` has returned. Another thread can still be using the event object internally. To safely close an event object, call `driver_select` with `ERL_DRV_USE` and `on==0`, which clears all events and then either calls `stop_select` or schedules it to be called when it is safe to close the event object. `ERL_DRV_USE` is to be set together with the first event for an event object. It is harmless to set `ERL_DRV_USE` even if it already has been done. Clearing all events but keeping `ERL_DRV_USE` set indicates that we are using the event object and probably will set events for it again.
Note:

ERL_DRV_USE was added in Erlang/OTP R13. Old drivers still work as before, but it is recommended to update them to use ERL_DRV_USE and stop_select to ensure that event objects are closed in a safe way.

The return value is 0, unless ready_input/ready_output is NULL, in which case it is -1.

```c
int driver_send_term(ErlDrvPort port, ErlDrvTermData receiver, ErlDrvTermData* term, int n)
```

Warning:

This function is deprecated. Use erl_drv_send_term instead.

Note:

The parameters of this function cannot be properly checked by the runtime system when executed by arbitrary threads. This can cause the function not to fail when it should.

Parameters term and n work as in erl_drv_output_term.

This function is thread-safe.

```c
int driver_set_timer(ErlDrvPort port, unsigned long time)
```

Sets a timer on the driver, which will count down and call the driver when it is timed out. Parameter time is the time in milliseconds before the timer expires.

When the timer reaches 0 and expires, the driver entry function timeout is called.

Notice that only one timer exists on each driver instance; setting a new timer replaces an older one.

Return value is 0, unless the timeout driver function is NULL, in which case it is -1.

```c
ErlDrvSizeT driver_sizeq(ErlDrvPort port)
```

Returns the number of bytes currently in the driver queue.

This function can be called from any thread if a port data lock associated with the port is locked by the calling thread during the call.

```c
void driver_system_info(ErlDrvSysInfo *sys_info_ptr, size_t size)
```

Writes information about the Erlang runtime system into the ErlDrvSysInfo structure referred to by the first argument. The second argument is to be the size of the ErlDrvSysInfo structure, that is, sizeof(ErlDrvSysInfo).

For information about specific fields, see ErlDrvSysInfo.

```c
ErlDrvSizeT driver_vec_to_buf(ErlIOVec *ev, char *buf, ErlDrvSizeT len)
```

Collects several segments of data, referenced by ev, by copying them in order to the buffer buf, of the size len.

If the data is to be sent from the driver to the port owner process, it is faster to use driver_outputv.
The return value is the space left in the buffer, that is, if `ev` contains less than `len` bytes it is the difference, and if `ev` contains `len` bytes or more, it is 0. This is faster if there is more than one header byte, as the binary syntax can construct integers directly from the binary.

```c
void erl_drv_busy_msgq_limits(ErlDrvPort port, ErlDrvSizeT *low, ErlDrvSizeT *high)
```

Sets and gets limits that will be used for controlling the busy state of the port message queue.

The port message queue is set into a busy state when the amount of command data queued on the message queue reaches the high limit. The port message queue is set into a not busy state when the amount of command data queued on the message queue falls below the low limit. Command data is in this context data passed to the port using either `Port ! {Owner, {command, Data}}` or `port_command/[2,3]`. Notice that these limits only concerns command data that have not yet reached the port. The busy port feature can be used for data that has reached the port.

Valid limits are values in the range `[ERL_DRV_BUSY_MSGQ_LIM_MIN, ERL_DRV_BUSY_MSGQ_LIM_MAX]`. Limits are automatically adjusted to be sane. That is, the system adjusts values so that the low limit used is lower than or equal to the high limit used. By default the high limit is 8 kB and the low limit is 4 kB.

By passing a pointer to an integer variable containing the value `ERL_DRV_BUSY_MSGQ_READ_ONLY`, the currently used limit is read and written back to the integer variable. A new limit can be set by passing a pointer to an integer variable containing a valid limit. The passed value is written to the internal limit. The internal limit is then adjusted. After this the adjusted limit is written back to the integer variable from which the new value was read. Values are in bytes.

The busy message queue feature can be disabled either by setting the `ERL_DRV_FLAG_NO_BUSY_MSGQ` driver flag in the `driver_entry` used by the driver, or by calling this function with `ERL_DRV_BUSY_MSGQ_DISABLED` as a limit (either low or high). When this feature has been disabled, it cannot be enabled again. When reading the limits, both are `ERL_DRV_BUSY_MSGQ_DISABLED` if this feature has been disabled.

Processes sending command data to the port are suspended if either the port is busy or if the port message queue is busy. Suspended processes are resumed when neither the port or the port message queue is busy.

For information about busy port functionality, see `set_busy_port`.

```c
void erl_drv_cond_broadcast(ErlDrvCond *cnd)
```

Broadcasts on a condition variable. That is, if other threads are waiting on the condition variable being broadcast on, all of them are woken.

`cnd` is a pointer to a condition variable to broadcast on.

This function is thread-safe.

```c
ErlDrvCond *erl_drv_cond_create(char *name)
```

Creates a condition variable and returns a pointer to it.

`name` is a string identifying the created condition variable. It is used to identify the condition variable in planned future debug functionality.

Returns NULL on failure. The driver creating the condition variable is responsible for destroying it before the driver is unloaded.

This function is thread-safe.

```c
void erl_drv_cond_destroy(ErlDrvCond *cnd)
```

Destroys a condition variable previously created by `erl_drv_cond_create`. 
erl_driver

cnd is a pointer to a condition variable to destroy.
This function is thread-safe.

char *erl_drv_cond_name(ErlDrvCond *cnd)
Returns a pointer to the name of the condition.
cnd is a pointer to an initialized condition.

Note:
This function is intended for debugging purposes only.

void erl_drv_cond_signal(ErlDrvCond *cnd)
Signals on a condition variable. That is, if other threads are waiting on the condition variable being signaled, one of them is woken.
cnd is a pointer to a condition variable to signal on.
This function is thread-safe.

void erl_drv_cond_wait(ErlDrvCond *cnd, ErlDrvMutex *mtx)
Waits on a condition variable. The calling thread is blocked until another thread wakes it by signaling or broadcasting on the condition variable. Before the calling thread is blocked, it unlocks the mutex passed as argument. When the calling thread is woken, it locks the same mutex before returning. That is, the mutex currently must be locked by the calling thread when calling this function.
cnd is a pointer to a condition variable to wait on. mtx is a pointer to a mutex to unlock while waiting.

Note:
erl_drv_cond_wait can return even if no one has signaled or broadcast on the condition variable. Code calling erl_drv_cond_wait is always to be prepared for erl_drv_cond_wait returning even if the condition that the thread was waiting for has not occurred. That is, when returning from erl_drv_cond_wait, always check if the condition has occurred, and if not call erl_drv_cond_wait again.
This function is thread-safe.

int erl_drv_consume_timeslice(ErlDrvPort port, int percent)
Gives the runtime system a hint about how much CPU time the current driver callback call has consumed since the last hint, or since the the start of the callback if no previous hint has been given.

  port          Port handle of the executing port.
  percent       Approximate consumed fraction of a full time-slice in percent.

The time is specified as a fraction, in percent, of a full time-slice that a port is allowed to execute before it is to surrender the CPU to other runnable ports or processes. Valid range is [1, 100]. The scheduling time-slice is not an exact entity, but can usually be approximated to about 1 millisecond.
Notice that it is up to the runtime system to determine if and how to use this information. Implementations on some
platforms can use other means to determine the consumed fraction of the time-slice. Lengthy driver callbacks should,
regardless of this, frequently call this function to determine if it is allowed to continue execution or not.
This function returns a non-zero value if the time-slice has been exhausted, and zero if the callback is allowed to
continue execution. If a non-zero value is returned, the driver callback is to return as soon as possible in order for
the port to be able to yield.
This function is provided to better support co-operative scheduling, improve system responsiveness, and to make it
easier to prevent misbehaviors of the VM because of a port monopolizing a scheduler thread. It can be used when
dividing lengthy work into some repeated driver callback calls, without the need to use threads.
See also the important warning text at the beginning of this manual page.

ErlDrvTime erl_drv_convert_time_unit(ErlDrvTime val, ErlDrvTimeUnit from,
ErlDrvTimeUnit to)
Converts the val value of time unit from to the corresponding value of time unit to. The result is rounded using
the floor function.

val
Value to convert time unit for.
from
Time unit of val.
to
Time unit of returned value.
Returns ERL_DRV_TIME_ERROR if called with an invalid time unit argument.
See also ErlDrvTime and ErlDrvTimeUnit.

int erl_drv_equal_tids(ErlDrvTid tid1, ErlDrvTid tid2)
Compares two thread identifiers, tid1 and tid2, for equality.
Returns 0 if they are not equal, and a value not equal to 0 if they are equal.

Note:
A thread identifier can be reused very quickly after a thread has terminated. Therefore, if a thread corresponding
to one of the involved thread identifiers has terminated since the thread identifier was saved, the result of
erl_drv_equal_tids does possibly not give the expected result.

This function is thread-safe.

int erl_drv_getenv(const char *key, char *value, size_t *value_size)
Retrieves the value of an environment variable.

key
A NULL-terminated string containing the name of the environment variable.
value
A pointer to an output buffer.
value_size
A pointer to an integer. The integer is used both for passing input and output sizes (see below).
When this function is called, *value_size is to contain the size of the value buffer.
On success, 0 is returned, the value of the environment variable has been written to the value buffer, and *value_size contains the string length (excluding the terminating NULL character) of the value written to the value buffer.

On failure, that is, no such environment variable was found, a value < 0 is returned. When the size of the value buffer is too small, a value > 0 is returned and *value_size has been set to the buffer size needed.

Warning:

This function reads the emulated environment used by os:getenv/1 and not the environment used by libc's getenv(3) or similar. Drivers that require that these are in sync will need to do so themselves, but keep in mind that they are segregated for a reason; getenv(3) and its friends are not thread-safe and may cause unrelated code to misbehave or crash the emulator.

This function is thread-safe.

void erl_drv_init_ack(ErlDrvPort port, ErlDrvData res)
Acknowledges the start of the port.

port
The port handle of the port (driver instance) doing the acknowledgment.

res
The result of the port initialization. Can be the same values as the return value of start, that is, any of the error codes or the ErlDrvData that is to be used for this port.

When this function is called the initiating erlang:open_port call is returned as if the start function had just been called. It can only be used when flag ERL_DRV_FLAG_USE_INIT_ACK has been set on the linked-in driver.

ErlDrvTime erl_drv_monotonic_time(ErlDrvTimeUnit time_unit)
Returns Erlang monotonic time. Notice that negative values are not uncommon.

time_unit is time unit of returned value.

Returns ERL_DRV_TIME_ERROR if called with an invalid time unit argument, or if called from a thread that is not a scheduler thread.

See also ErlDrvTime and ErlDrvTimeUnit.

ErlDrvMutex *erl_drv_mutex_create(char *name)
Creates a mutex and returns a pointer to it.

name is a string identifying the created mutex. It is used to identify the mutex in debug functionality (see note).

Returns NULL on failure. The driver creating the mutex is responsible for destroying it before the driver is unloaded.

This function is thread-safe.
**Note:**

One such debug functionality is the lock checker, which can detect locking order violations and thereby potential deadlock bugs. For the lock checker to work the name should be on the format "App.Type" or "App.Type[Instance]", where App is the name of the application, Type is the name of the lock type and Instance is optional information about each lock instance. "App.Type" should be a unique name for the lock checker to detect lock order violations between locks of different types. The Instance information is currently ignored.

For example, if we have mutexes of types "myapp.xtable" and "myapp.xitem" then the lock checker will make sure either "myapp.xtable" locks are never locked after "myapp.xitem" locks or vice versa.

```c
void erl_drv_mutex_destroy(ErlDrvMutex *mtx)
```

Destroys a mutex previously created by `erl_drv_mutex_create`. The mutex must be in an unlocked state before it is destroyed.

*mtx* is a pointer to a mutex to destroy.

This function is thread-safe.

```c
void erl_drv_mutex_lock(ErlDrvMutex *mtx)
```

Locks a mutex. The calling thread is blocked until the mutex has been locked. A thread that has currently locked the mutex **cannot** lock the same mutex again.

*mtx* is a pointer to a mutex to lock.

**Warning:**

If you leave a mutex locked in an emulator thread when you let the thread out of your control, you will **very likely** deadlock the whole emulator.

This function is thread-safe.

```c
char *erl_drv_mutex_name(ErlDrvMutex *mtx)
```

Returns a pointer to the mutex name.

*mtx* is a pointer to an initialized mutex.

**Note:**

This function is intended for debugging purposes only.

```c
int erl_drv_mutex_trylock(ErlDrvMutex *mtx)
```

Tries to lock a mutex. A thread that has currently locked the mutex **cannot** try to lock the same mutex again.

*mtx* is a pointer to a mutex to try to lock.

Returns 0 on success, otherwise EBUSY.
Warning:
If you leave a mutex locked in an emulator thread when you let the thread out of your control, you will very likely deadlock the whole emulator.

This function is thread-safe.

void erl_drv_mutex_unlock(ErlDrvMutex *mtx)
Unlocks a mutex. The mutex currently must be locked by the calling thread.
mtx is a pointer to a mutex to unlock.
This function is thread-safe.

int erl_drv_output_term(ErlDrvTermData port, ErlDrvTermData* term, int n)
Sends data in the special driver term format to the port owner process. This is a fast way to deliver term data from a driver. It needs no binary conversion, so the port owner process receives data as normal Erlang terms. The erl_drv_send_term functions can be used for sending to any process on the local node.

Note:
Parameter port is not an ordinary port handle, but a port handle converted using driver_mk_port.

Parameter term points to an array of ErlDrvTermData with n elements. This array contains terms described in the driver term format. Every term consists of 1-4 elements in the array. The first term has a term type and then arguments. Parameter port specifies the sending port.

Tuples, maps, and lists (except strings, see below) are built in reverse polish notation, so that to build a tuple, the elements are specified first, and then the tuple term, with a count. Likewise for lists and maps.

- A tuple must be specified with the number of elements. (The elements precede the ERL_DRV_TUPLE term.)
- A map must be specified with the number of key-value pairs N. The key-value pairs must precede the ERL_DRV_MAP in this order: key1,value1,key2,value2,...,keyN,valueN. Duplicate keys are not allowed.
- A list must be specified with the number of elements, including the tail, which is the last term preceding ERL_DRV_LIST.

The special term ERL_DRV_STRING_CONS is used to "splice" in a string in a list, a string specified this way is not a list in itself, but the elements are elements of the surrounding list.
<table>
<thead>
<tr>
<th>Term type</th>
<th>Arguments</th>
</tr>
</thead>
<tbody>
<tr>
<td>ERL_DRV_NIL</td>
<td></td>
</tr>
<tr>
<td>ERL_DRV_ATOM</td>
<td>ErlDrvTermData atom (from driver_mk_atom(char *string))</td>
</tr>
<tr>
<td>ERL_DRV_INT</td>
<td>ErlDrvSInt integer</td>
</tr>
<tr>
<td>ERL_DRV_UINT</td>
<td>ErlDrvUInt integer</td>
</tr>
<tr>
<td>ERL_DRV_INT64</td>
<td>ErlDrvSInt64 *integer_ptr</td>
</tr>
<tr>
<td>ERL_DRV_UINT64</td>
<td>ErlDrvUInt64 *integer_ptr</td>
</tr>
<tr>
<td>ERL_DRV_PORT</td>
<td>ErlDrvTermData port (from driver_mk_port(ErlDrvPort port))</td>
</tr>
<tr>
<td>ERL_DRV_BINARY</td>
<td>ErDrvBinary *bin, ErlDrvUInt len, ErlDrvUInt offset</td>
</tr>
<tr>
<td>ERL_DRV_BUF2BINARY</td>
<td>char *buf, ErlDrvUInt len</td>
</tr>
<tr>
<td>ERL_DRV_STRING</td>
<td>char *str, int len</td>
</tr>
<tr>
<td>ERL_DRV_TUPLE</td>
<td>int sz</td>
</tr>
<tr>
<td>ERL_DRV_LIST</td>
<td>int sz</td>
</tr>
<tr>
<td>ERL_DRV_PID</td>
<td>ErlDrvTermData pid (from driver_connected(ErlDrvPort port))</td>
</tr>
<tr>
<td>ERL_DRV_STRING_CONS</td>
<td>char *str, int len</td>
</tr>
<tr>
<td>ERL_DRV_FLOAT</td>
<td>double *dbl</td>
</tr>
<tr>
<td>ERL_DRV_EXT2TERM</td>
<td>char *buf, ErlDrvUInt len</td>
</tr>
<tr>
<td>ERL_DRV_MAP</td>
<td>int sz</td>
</tr>
</tbody>
</table>

The unsigned integer data type ErlDrvUInt and the signed integer data type ErlDrvSInt are 64 bits wide on a 64-bit runtime system and 32 bits wide on a 32-bit runtime system. They were introduced in ERTS 5.6 and replaced some of the int arguments in the list above.

The unsigned integer data type ErlDrvUInt64 and the signed integer data type ErlDrvSInt64 are always 64 bits wide. They were introduced in ERTS 5.7.4.

To build the tuple {tcp, Port, [100 | Binary]}, the following call can be made.

```erlang
ErlDrvBinary* bin = ...;
ErlDrvPort port = ...;
ErlDrvTermData spec[] = {
  ERL_DRV_ATOM, driver_mk_atom("tcp"),
  ERL_DRV_PORT, driver_mk_port(drvport),
  ERL_DRV_INT, 100,
  ERL_DRV_BINARY, bin, 50, 0,
  ERL_DRV_LIST, 2,
  ERL_DRV_TUPLE, 3,
};
```

Here bin is a driver binary of length at least 50 and drvport is a port handle. Notice that ERL_DRV_LIST comes after the elements of the list, likewise ERL_DRV_TUPLE.

The ERL_DRV_STRING_CONS term is a way to construct strings. It works differently from how ERL_DRV_STRING works. ERL_DRV_STRING_CONS builds a string list in reverse order (as opposed to how ERL_DRV_LIST works), concatenating the strings added to a list. The tail must be specified before ERL_DRV_STRING_CONS.

ERL_DRV_STRING constructs a string, and ends it. (So it is the same as ERL_DRV_NIL followed by ERL_DRV_STRING_CONS.)

```erlang
/* to send [x, "abc", y] to the port: */
ErlDrvTermData spec[] = {
  ERL_DRV_ATOM, driver_mk_atom("x"),
  ERL_DRV_STRING, (ErlDrvTermData)"abc", 3,
  ERL_DRV_ATOM, driver_mk_atom("y"),
  ERL_DRV_NIL,
  ERL_DRV_LIST, 4
};
```

368 | Ericsson AB. All Rights Reserved.: Erlang Run-Time System Application (ERTS)
/* to send "abc123" to the port: */
ErlDrvTermData spec[] = {
    ERL_DRV_NIL,        /* with STRING CONS, the tail comes first */
    ERL_DRV_STRING_CONS, (ErlDrvTermData)"123", 3,
    ERL_DRV_STRING_CONS, (ErlDrvTermData)"abc", 3,
};
erl_drv_output_term(driver_mk_port(drvport), spec, sizeof(spec) / sizeof(spec[0]));

The ERL_DRV_EXT2TERM term type is used for passing a term encoded with the external format, that is, a term that has been encoded by erlang:term_to_binary(), erl_interface:ei(3), and so on. For example, if binp is a pointer to an ErlDrvBinary that contains term {17, 4711} encoded with the external format, and you want to wrap it in a two-tuple with the tag my_tag, that is, {my_tag, {17, 4711}}, you can do as follows:

ErlDrvTermData spec[] = {
    ERL_DRV_ATOM, driver_mk_atom("my_tag"),
    ERL_DRV_EXT2TERM, (ErlDrvTermData) binp->orig_bytes, binp->orig_size
    ERL_DRV_TUPLE, 2,
};
erl_drv_output_term(driver_mk_port(drvport), spec, sizeof(spec) / sizeof(spec[0]));

To build the map #{key1 => 100, key2 => {200, 300}}, the following call can be made.

ErlDrvPort port = ...
ErlDrvTermData spec[] = {
    ERL_DRV_ATOM, driver_mk_atom("key1"),
    ERL_DRV_INT, 100,
    ERL_DRV_ATOM, driver_mk_atom("key2"),
    ERL_DRV_INT, 200,
    ERL_DRV_INT, 300,
    ERL_DRV_TUPLE, 2,
    ERL_DRV_MAP, 2
};
erl_drv_output_term(driver_mk_port(drvport), spec, sizeof(spec) / sizeof(spec[0]));

If you want to pass a binary and do not already have the content of the binary in an ErlDrvBinary, you can benefit from using ERL_DRV_BUF2BINARY instead of creating an ErlDrvBinary through driver_alloc_binary and then pass the binary through ERL_DRV_BINARY. The runtime system often allocates binaries smarter if ERL_DRV_BUF2BINARY is used. However, if the content of the binary to pass already resides in an ErlDrvBinary, it is normally better to pass the binary using ERL_DRV_BINARY and the ErlDrvBinary in question.

The ERL_DRV_UINT, ERL_DRV_BUF2BINARY, and ERL_DRV_EXT2TERM term types were introduced in ERTS 5.6.

This function is thread-safe.

int erl_drdrv_putenv(const char *key, char *value)
Sets the value of an environment variable.
key is a NULL-terminated string containing the name of the environment variable.
value is a NULL-terminated string containing the new value of the environment variable.
Returns 0 on success, otherwise a value != 0.

Note:
The result of passing the empty string ("") as a value is platform-dependent. On some platforms the variable value is set to the empty string, on others the environment variable is removed.

Ericsson AB. All Rights Reserved.: Erlang Run-Time System Application (ERTS) | 369
**Warning:**

This function modifies the emulated environment used by `os:putenv/2` and not the environment used by libc's `putenv(3)` or similar. Drivers that require that these are in sync will need to do so themselves, but keep in mind that they are segregated for a reason; `putenv(3)` and its friends are **not thread-safe** and may cause unrelated code to misbehave or crash the emulator.

This function is thread-safe.

```c
ErlDrvRWLock *erl_drv_rwlock_create(char *name)
```

Creates an rwlock and returns a pointer to it.

- `name` is a string identifying the created rwlock. It is used to identify the rwlock in debug functionality (see note about the lock checker).

Returns `NULL` on failure. The driver creating the rwlock is responsible for destroying it before the driver is unloaded.

This function is thread-safe.

```c
void erl_drv_rwlock_destroy(ErlDrvRWLock *rwlck)
```

Destroys an rwlock previously created by `erl_drv_rwlock_create`. The rwlock must be in an unlocked state before it is destroyed.

- `rwlck` is a pointer to an rwlock to destroy.

This function is thread-safe.

```c
char *erl_drv_rwlock_name(ErlDrvRWLock *rwlck)
```

Returns a pointer to the name of the rwlock.

- `rwlck` is a pointer to an initialized rwlock.

**Note:**

This function is intended for debugging purposes only.

```c
void erl_drv_rwlock_rlock(ErlDrvRWLock *rwlck)
```

Read locks an rwlock. The calling thread is blocked until the rwlock has been read locked. A thread that currently has read or read/write locked the rwlock **cannot** lock the same rwlock again.

- `rwlck` is a pointer to the rwlock to read lock.

**Warning:**

If you leave an rwlock locked in an emulator thread when you let the thread out of your control, you will **very likely** deadlock the whole emulator.

This function is thread-safe.

```c
void erl_drv_rwlock_runlock(ErlDrvRWLock *rwlck)
```

Read unlocks an rwlock. The rwlock currently must be read locked by the calling thread.
**erl_driver**

rwlc

is a pointer to an rwlock to read unlock.

This function is thread-safe.

```c
void erl_drv_rwlock_rwlock(ErlDrvRWLock *rwlc)
```

Read/write locks an rwlock. The calling thread is blocked until the rwlock has been read/write locked. A thread that currently has read or read/write locked the rwlock cannot lock the same rwlock again.

**Warning:**

If you leave an rwlock locked in an emulator thread when you let the thread out of your control, you will very likely deadlock the whole emulator.

This function is thread-safe.

```c
void erl_drv_rwlock_rwunlock(ErlDrvRWLock *rwlc)
```

Read/write unlocks an rwlock. The rwlock currently must be read/write locked by the calling thread.

**Warning:**

If you leave an rwlock locked in an emulator thread when you let the thread out of your control, you will very likely deadlock the whole emulator.

This function is thread-safe.

```c
int erl_drv_rwlock_tryrlock(ErlDrvRWLock *rwlc)
```

Tries to read lock an rwlock.

**Warning:**

If you leave an rwlock locked in an emulator thread when you let the thread out of your control, you will very likely deadlock the whole emulator.

This function is thread-safe.

```c
int erl_drv_rwlock_tryrwlock(ErlDrvRWLock *rwlc)
```

Tries to read/write lock an rwlock. A thread that currently has read or read/write locked the rwlock cannot try to lock the same rwlock again.

**Warning:**

If you leave an rwlock locked in an emulator thread when you let the thread out of your control, you will very likely deadlock the whole emulator.

This function is thread-safe.
erl_driver

int erl_drv_send_term(ErlDrvTermData port, ErlDrvTermData receiver, 
ErlDrvTermData* term, int n)

This function is the only way for a driver to send data to other processes than the port owner process. Parameter receiver specifies the process to receive the data.

**Note:**

Parameter port is **not** an ordinary port handle, but a port handle converted using driver_mk_port.

Parameters port, term, and n work as in erl_drv_output_term.

This function is thread-safe.

void erl_drv_set_os_pid(ErlDrvPort port, ErlDrvSInt pid)

Sets the os_pid seen when doing erlang:port_info/2 on this port.

port is the port handle of the port (driver instance) to set the pid on. pid is the pid to set.

int erl_drv_thread_create(char *name, ErlDrvTid *tid, void * (*func)(void *), 
void *arg, ErlDrvThreadOpts *opts)

Creates a new thread.

- **name**
  A string identifying the created thread. It is used to identify the thread in planned future debug functionality.

- **tid**
  A pointer to a thread identifier variable.

- **func**
  A pointer to a function to execute in the created thread.

- **arg**
  A pointer to argument to the func function.

- **opts**
  A pointer to thread options to use or NULL.

Returns 0 on success, otherwise an errno value is returned to indicate the error. The newly created thread begins executing in the function pointed to by func, and func is passed arg as argument. When erl_drv_thread_create returns, the thread identifier of the newly created thread is available in *tid. opts can be either a NULL pointer, or a pointer to an ErlDrvThreadOpts structure. If opts is a NULL pointer, default options are used, otherwise the passed options are used.

**Warning:**

You are not allowed to allocate the ErlDrvThreadOpts structure by yourself. It must be allocated and initialized by erl_drv_thread_opts_create.

The created thread terminates either when func returns or if erl_drv_thread_exit is called by the thread. The exit value of the thread is either returned from func or passed as argument to erl_drv_thread_exit. The driver creating the thread is responsible for joining the thread, through erl_drv_thread_join, before the driver is unloaded. "Detached" threads cannot be created, that is, threads that do not need to be joined.
### Warning:

All created threads must be joined by the driver before it is unloaded. If the driver fails to join all threads created before it is unloaded, the runtime system most likely crashes when the driver code is unloaded.

This function is thread-safe.

```c
void erl_drv_thread_exit(void *exit_value)
```

Terminates the calling thread with the exit value passed as argument. `exit_value` is a pointer to an exit value or NULL.

You are only allowed to terminate threads created with `erl_drv_thread_create`.

The exit value can later be retrieved by another thread through `erl_drv_thread_join`.

This function is thread-safe.

```c
int erl_drv_thread_join(ErlDrvTid tid, void **exit_value)
```

Joins the calling thread with another thread, that is, the calling thread is blocked until the thread identified by `tid` has terminated.

`tid` is the thread identifier of the thread to join. `exit_value` is a pointer to a pointer to an exit value, or NULL.

Returns 0 on success, otherwise an `errno` value is returned to indicate the error.

A thread can only be joined once. The behavior of joining more than once is undefined, an emulator crash is likely. If `exit_value == NULL`, the exit value of the terminated thread is ignored, otherwise the exit value of the terminated thread is stored at `*exit_value`.

This function is thread-safe.

```c
char *erl_drv_thread_name(ErlDrvTid tid)
```

Returns a pointer to the name of the thread.

`tid` is a thread identifier.

---

### Note:

This function is intended for debugging purposes only.

```c
ErlDrvThreadOpts *erl_drv_thread_opts_create(char *name)
```

Allocates and initializes a thread option structure.

`name` is a string identifying the created thread options. It is used to identify the thread options in planned future debug functionality.

Returns `NULL` on failure. A thread option structure is used for passing options to `erl_drv_thread_create`. If the structure is not modified before it is passed to `erl_drv_thread_create`, the default values are used.

---

### Warning:

You are not allowed to allocate the `ErlDrvThreadOpts` structure by yourself. It must be allocated and initialized by `erl_drv_thread_opts_create`.

---
This function is thread-safe.

```c
void erl_drv_thread_opts_destroy(ErlDrvThreadOpts *opts)
```

Destroys thread options previously created by `erl_drv_thread_opts_create`.

`opts` is a pointer to thread options to destroy.

This function is thread-safe.

```c
ErlDrvTid erl_drv_thread_self(void)
```

Returns the thread identifier of the calling thread.

This function is thread-safe.

```c
ErlDrvTime erl_drv_time_offset(ErlDrvTimeUnit time_unit)
```

Returns the current time offset between Erlang monotonic time and Erlang system time converted into the `time_unit` passed as argument.

`time_unit` is time unit of returned value.

Returns `ERL_DRV_TIME_ERROR` if called with an invalid time unit argument, or if called from a thread that is not a scheduler thread.

See also `ErlDrvTime` and `ErlDrvTimeUnit`.

```c
void *erl_drv_tsd_get(ErlDrvTSDKey key)
```

Returns the thread-specific data associated with `key` for the calling thread.

`key` is a thread-specific data key.

Returns `NULL` if no data has been associated with `key` for the calling thread.

This function is thread-safe.

```c
int erl_drv_tsd_key_create(char *name, ErlDrvTSDKey *key)
```

Creates a thread-specific data key.

`name` is a string identifying the created key. It is used to identify the key in planned future debug functionality.

`key` is a pointer to a thread-specific data key variable.

Returns `0` on success, otherwise an `errno` value is returned to indicate the error. The driver creating the key is responsible for destroying it before the driver is unloaded.

This function is thread-safe.

```c
void erl_drv_tsd_key_destroy(ErlDrvTSDKey key)
```

Destroys a thread-specific data key previously created by `erl_drv_tsd_key_create`. All thread-specific data using this key in all threads must be cleared (see `erl_drv_tsd_set`) before the call to `erl_drv_tsd_key_destroy`.

`key` is a thread-specific data key to destroy.
Warning:
A destroyed key is very likely to be reused soon. Therefore, if you fail to clear the thread-specific data using this key in a thread before destroying the key, you will very likely get unexpected errors in other parts of the system.

This function is thread-safe.

```c
void erl_drv_tsd_set(ErlDrvTSDKey key, void *data)
```
Sets thread-specific data associated with `key` for the calling thread. You are only allowed to set thread-specific data for threads while they are fully under your control. For example, if you set thread-specific data in a thread calling a driver callback function, it must be cleared, that is, set to NULL, before returning from the driver callback function.

`key` is a thread-specific data key.

`data` is a pointer to data to associate with `key` in the calling thread.

Warning:
If you fail to clear thread-specific data in an emulator thread before letting it out of your control, you might never be able to clear this data with later unexpected errors in other parts of the system as a result.

This function is thread-safe.

```c
char *erl_errno_id(int error)
```
Returns the atom name of the Erlang error, given the error number in `error`. The error atoms are `einval`, `enoent`, and so on. It can be used to make error terms from the driver.

```c
int remove_driver_entry(ErlDrvEntry *de)
```
Removes a driver entry `de` previously added with `add_driver_entry`. Driver entries added by the `erl_ddll` Erlang interface cannot be removed by using this interface.

```c
void set_busy_port(ErlDrvPort port, int on)
```
Sets and unsets the busy state of the port. If `on` is non-zero, the port is set to busy. If it is zero, the port is set to not busy. You typically want to combine this feature with the busy port message queue functionality.

Processes sending command data to the port are suspended if either the port or the port message queue is busy. Suspended processes are resumed when neither the port or the port message queue is busy. Command data is in this context data passed to the port using `erlang:port_command(Port, Data, [force])` even if the driver has signaled that it is busy.

If the `ERL_DRV_FLAG_SOFT_BUSY` has been set in the `driver_entry`, data can be forced into the driver through `erlang:port_command(Port, Data, [force])` even if the driver has signaled that it is busy.

For information about busy port message queue functionality, see `erl_drv_busy_msgq_limits`.

```c
void set_port_control_flags(ErlDrvPort port, int flags)
```
Sets flags for how the control driver entry function will return data to the port owner process. (The control function is called from `erlang:port_control/3`.)

Currently there are only two meaningful values for `flags`: 0 means that data is returned in a list, and `PORT_CONTROL_FLAG_BINARY` means data is returned as a binary from control.
See Also

driver_entry(3), erlang(3), erl_ddll(3), section How to Implement an Alternative Carrier for the
Erlang Distribution in the User's Guide
A NIF library contains native implementation of some functions of an Erlang module. The native implemented functions (NIFs) are called like any other functions without any difference to the caller. A NIF library is built as a dynamically linked library file and loaded in runtime by calling `erlang:load_nif/2`.

**Warning:**

*Use this functionality with extreme care.*

A native function is executed as a direct extension of the native code of the VM. Execution is not made in a safe environment. The VM **cannot** provide the same services as provided when executing Erlang code, such as pre-emptive scheduling or memory protection. If the native function does not behave well, the whole VM will misbehave.

- A native function that crashes will crash the whole VM.
- An erroneously implemented native function can cause a VM internal state inconsistency, which can cause a crash of the VM, or miscellaneous misbehaviors of the VM at any point after the call to the native function.
- A native function doing lengthy work before returning degrades responsiveness of the VM, and can cause miscellaneous strange behaviors. Such strange behaviors include, but are not limited to, extreme memory usage, and bad load balancing between schedulers. Strange behaviors that can occur because of lengthy work can also vary between Erlang/OTP releases.

**Example**

A minimal example of a NIF library can look as follows:

```c
/* niftest.c */
#include <erl_nif.h>

static ERL_NIF_TERM hello(ErlNifEnv* env, int argc, const ERL_NIF_TERM argv[])
{
    return enif_make_string(env, "Hello world!", ERL_NIF_LATIN1);
}

static ErlNifFunc nif_funcs[] =
{
    {"hello", 0, hello}
};

ERL_NIF_INIT(niftest, nif_funcs, NULL, NULL, NULL, NULL)
```

The Erlang module can look as follows:
erl_nif

-module(niftest).
-export([init/0, hello/0]).
-nifs([hello/0]).
-on_load(init/0).
init() ->
erlang:load_nif("./niftest", 0).
hello() ->
erlang:nif_error("NIF library not loaded").

Compile and test can look as follows (on Linux):

```bash
$> gcc -fPIC -shared -o niftest.so niftest.c -I $ERL_ROOT/usr/include/
$> erl
1> c(niftest).
{ok,niftest}
2> niftest:hello().
"Hello world!"
```

In the example above the `on_load` directive is used get function `init` called automatically when the module is loaded. Function `init` in turn calls `erlang:load_nif/2` which loads the NIF library and replaces the `hello` function with its native implementation in C. Once loaded, a NIF library is persistent. It will not be unloaded until the module code version that it belongs to is purged.

The `-nifs()` attribute specifies which functions in the module that are to be replaced by NIFs.

Each NIF must have an implementation in Erlang to be invoked if the function is called before the NIF library is successfully loaded. A typical such stub implementation is to call `erlang:nif_error` which will raise an exception. The Erlang function can also be used as a fallback implementation if the NIF library lacks implementation for some OS or hardware architecture for example.

**Note:**

A NIF does not have to be exported, it can be local to the module. However, unused local stub functions will be optimized away by the compiler, causing loading of the NIF library to fail.

**Functionality**

All interaction between NIF code and the Erlang runtime system is performed by calling NIF API functions. Functions exist for the following functionality:

Read and write Erlang terms

Any Erlang terms can be passed to a NIF as function arguments and be returned as function return values. The terms are of C-type `ERL_NIF_TERM` and can only be read or written using API functions. Most functions to read the content of a term are prefixed `enif_get_` and usually return `true` (or `false`) if the term is of the expected type (or not). The functions to write terms are all prefixed `enif_make_` and usually return the created `ERL_NIF_TERM`. There are also some functions to query terms, like `enif_is_atom`, `enif_is_identical`, and `enif_compare`.

All terms of type `ERL_NIF_TERM` belong to an environment of type `ErlNifEnv`, except atoms created during loading (by callbacks `load` or `upgrade`). The lifetime of a term is controlled by the lifetime of its environment object. All API functions that read or write terms have the environment that the term belongs to as the first function argument.
argument. However, the atoms created during loading can be referred as a term in any ErlNifEnv. That is, the best practice is to create all your atoms during loading and store them in static/global variables, for example:

```c
#include <erl_nif.h>

ERL_NIF_TERM world_atom;

static int load(ErlNifEnv* env, void** priv_data, ERL_NIF_TERM load_info)
{
    world_atom = enif_make_atom(env, "world");
    return 0;
}

static ERL_NIF_TERM hello(ErlNifEnv* env, int argc, const ERL_NIF_TERM argv[])
{
    ERL_NIF_TERM hello_string = enif_make_string(env, "Hello", ERL_NIF_LATIN1);
    return enif_make_tuple2(env, hello_string, world_atom);
}

static ErlNifFunc nif_funcs[] = { { "hello", 0, hello } }; 

ERL_NIF_INIT(niftest, nif_funcs, load, NULL, NULL, NULL)
```

**Binaries**

Terms of type binary are accessed with the help of struct type ErlNifBinary, which contains a pointer (data) to the raw binary data and the length (size) of the data in bytes. Both data and size are read-only and are only to be written using calls to API functions. Instances of ErlNifBinary are, however, always allocated by the user (usually as local variables).

The raw data pointed to by data is only mutable after a call to enif_alloc_binary or enif_realloc_binary. All other functions that operate on a binary leave the data as read-only. A mutable binary must in the end either be freed with enif_release_binary or made read-only by transferring it to an Erlang term with enif_make_binary. However, it does not have to occur in the same NIF call. Read-only binaries do not have to be released.

enif_make_new_binary can be used as a shortcut to allocate and return a binary in the same NIF call.

Binaries are sequences of whole bytes. Bitstrings with an arbitrary bit length have no support yet.

**Resource objects**

The use of resource objects is a safe way to return pointers to native data structures from a NIF. A resource object is only a block of memory allocated with enif_alloc_resource. A handle ("safe pointer") to this memory block can then be returned to Erlang by the use of enif_make_resource. The term returned by enif_make_resource is opaque in nature. It can be stored and passed between processes, but the only real end usage is to pass it back as an argument to a NIF. The NIF can then call enif_get_resource and get back a pointer to the memory block, which is guaranteed to still be valid. A resource object is not deallocated until the last handle term is garbage collected by the VM and the resource is released with enif_release_resource (not necessarily in that order).

All resource objects are created as instances of some resource type. This makes resources from different modules to be distinguishable. A resource type is created by calling enif_open_resource_type when a library is loaded. Objects of that resource type can then later be allocated and enif_get_resource verifies that the resource is of the expected type. A resource type can have a user-supplied destructor function, which is automatically called when resources of that type are released (by either the garbage collector or enif_release_resource). Resource types are uniquely identified by a supplied name string and the name of the implementing module.

The following is a template example of how to create and return a resource object.
ERL_NIF_TERM term;
MyStruct* obj = enif_alloc_resource(my_resource_type, sizeof(MyStruct));

/* initialize struct ... */
term = enif_make_resource(env, obj);

if (keep_a_reference_of_our_own) {
    /* store 'obj' in static variable, private data or other resource object */
} else {
    enif_release_resource(obj);
    /* resource now only owned by "Erlang" */
}
return term;

Notice that once enif_make_resource creates the term to return to Erlang, the code can choose to either keep its own native pointer to the allocated struct and release it later, or release it immediately and rely only on the garbage collector to deallocate the resource object eventually when it collects the term.

Another use of resource objects is to create binary terms with user-defined memory management. enif_make_resource_binary creates a binary term that is connected to a resource object. The destructor of the resource is called when the binary is garbage collected, at which time the binary data can be released. An example of this can be a binary term consisting of data from a mmap'ed file. The destructor can then do munmap to release the memory region.

Resource types support upgrade in runtime by allowing a loaded NIF library to take over an already existing resource type and by that "inherit" all existing objects of that type. The destructor of the new library is called for the inherited objects and the library with the old destructor function can be safely unloaded. Existing resource objects, of a module that is upgraded, must either be deleted or taken over by the new NIF library. The unloading of a library is postponed as long as there exist resource objects with a destructor function in the library.

Module upgrade and static data
A loaded NIF library is tied to the Erlang module instance that loaded it. If the module is upgraded, the new module instance needs to load its own NIF library (or maybe choose not to). The new module instance can, however, choose to load the exact same NIF library as the old code if it wants to. Sharing the dynamic library means that static data defined by the library is shared as well. To avoid unintentionally shared static data between module instances, each Erlang module version can keep its own private data. This private data can be set when the NIF library is loaded and later retrieved by calling enif_priv_data.

Threads and concurrency
A NIF is thread-safe without any explicit synchronization as long as it acts as a pure function and only reads the supplied arguments. When you write to a shared state either through static variables or enif_priv_data, you need to supply your own explicit synchronization. This includes terms in process independent environments that are shared between threads. Resource objects also require synchronization if you treat them as mutable.

The library initialization callbacks load and upgrade are thread-safe even for shared state data.

Version Management
When a NIF library is built, information about the NIF API version is compiled into the library. When a NIF library is loaded, the runtime system verifies that the library is of a compatible version. erl_nif.h defines the following:

ERL_NIF_MAJOR_VERSION

Incremented when NIF library incompatible changes are made to the Erlang runtime system. Normally it suffices to recompile the NIF library when the ERL_NIF_MAJOR_VERSION has changed, but it can,
under rare circumstances, mean that NIF libraries must be slightly modified. If so, this will of course be documented.

ERL_NIF_MINOR_VERSION

Incremented when new features are added. The runtime system uses the minor version to determine what features to use.

The runtime system normally refuses to load a NIF library if the major versions differ, or if the major versions are equal and the minor version used by the NIF library is greater than the one used by the runtime system. Old NIF libraries with lower major versions are, however, allowed after a bump of the major version during a transition period of two major releases. Such old NIF libraries can however fail if deprecated features are used.

Time Measurement

Support for time measurement in NIF libraries:

• ErlNifTime
• ErlNifTimeUnit
• enif_monotonic_time()
• enif_time_offset()
• enif_convert_time_unit()

I/O Queues

The Erlang nif library contains function for easily working with I/O vectors as used by the unix system call writev. The I/O Queue is not thread safe, so some other synchronization mechanism has to be used.

• SysIOVec
• ErlNifIOVec
• enif_ioq_create()
• enif_ioq_destroy()
• enif_ioq_enq_binary()
• enif_ioq_enqv()
• enif_ioq_deq()
• enif_ioq_peek()
• enif_ioq_peek_head()
• enif_inspect_iovec()
• enif_free_iovec()

Typical usage when writing to a file descriptor looks like this:
```c
int writeiovec(ErlNifEnv *env, ERL_NIF_TERM term, ERL_NIF_TERM *tail, ErlNifIOQueue *q, int fd) {
    ErlNifIOVec vec, *iovec = &vec;
    SysIOVec *sysiovec;
    int saved_errno;
    int iovcnt, n;
    if (!enif_inspect_iovec(env, 64, term, tail, &iovec))
        return -2;
    if (enif_ioq_size(q) > 0) {
        /* If the I/O queue contains data we enqueue the iovec and
           then peek the data to write out of the queue. */
        if (!enif_ioq_enqv(q, iovec, 0))
            return -3;
        sysiovec = enif_ioq_peek(q, &iovcnt);
    } else {
        /* If the I/O queue is empty we skip the trip through it. */
        iovcnt = iovec->iovcnt;
        sysiovec = iovec->iov;
    }
    /* Attempt to write the data */
    n = writev(fd, sysiovec, iovcnt);
    saved_errno = errno;
    if (enif_ioq_size(q) == 0) {
        /* If the I/O queue was initially empty we enqueue any
           remaining data into the queue for writing later. */
        if (n >= 0 && !enif_ioq_enqv(q, iovec, n))
            return -3;
    } else {
        /* Dequeue any data that was written from the queue. */
        if (n > 0 && !enif_ioq_deq(q, n, NULL))
            return -4;
    }
    /* return n, which is either number of bytes written or -1 if
       some error happened */
    errno = saved_errno;
    return n;
}
```

Long-running NIFs

As mentioned in the warning text at the beginning of this manual page, it is of vital importance that a native function returns relatively fast. It is difficult to give an exact maximum amount of time that a native function is allowed to work, but usually a well-behaving native function is to return to its caller within 1 millisecond. This can be achieved using different approaches. If you have full control over the code to execute in the native function, the best approach is to divide the work into multiple chunks of work and call the native function multiple times. This is, however, not always possible, for example when calling third-party libraries.

The enif_consume_timeslice() function can be used to inform the runtime system about the length of the NIF call. It is typically always to be used unless the NIF executes very fast.

If the NIF call is too lengthy, this must be handled in one of the following ways to avoid degraded responsiveness, scheduler load balancing problems, and other strange behaviors:

Yielding NIF

If the functionality of a long-running NIF can be split so that its work can be achieved through a series of shorter NIF calls, the application has two options:
• Make that series of NIF calls from the Erlang level.
• Call a NIF that first performs a chunk of the work, then invokes the enif_schedule_nif function to schedule another NIF call to perform the next chunk. The final call scheduled in this manner can then return the overall result.

Breaking up a long-running function in this manner enables the VM to regain control between calls to the NIFs.

This approach is always preferred over the other alternatives described below. This both from a performance perspective and a system characteristics perspective.

Threaded NIF
This is accomplished by dispatching the work to another thread managed by the NIF library, return from the NIF, and wait for the result. The thread can send the result back to the Erlang process using enif_send. Information about thread primitives is provided below.

Dirty NIF
A NIF that cannot be split and cannot execute in a millisecond or less is called a "dirty NIF", as it performs work that the ordinary schedulers of the Erlang runtime system cannot handle cleanly. Applications that make use of such functions must indicate to the runtime that the functions are dirty so they can be handled specially. This is handled by executing dirty jobs on a separate set of schedulers called dirty schedulers. A dirty NIF executing on a dirty scheduler does not have the same duration restriction as a normal NIF.

It is important to classify the dirty job correct. An I/O bound job should be classified as such, and a CPU bound job should be classified as such. If you should classify CPU bound jobs as I/O bound jobs, dirty I/O schedulers might starve ordinary schedulers. I/O bound jobs are expected to either block waiting for I/O, and/or spend a limited amount of time moving data.

To schedule a dirty NIF for execution, the application has two options:
• Set the appropriate flags value for the dirty NIF in its ErlNifFunc entry.
• Call enif_schedule_nif, pass to it a pointer to the dirty NIF to be executed, and indicate with argument flags whether it expects the operation to be CPU-bound or I/O-bound.

A job that alternates between I/O bound and CPU bound can be reclassified and rescheduled using enif_schedule_nif so that it executes on the correct type of dirty scheduler at all times. For more information see the documentation of the erl(1) command line arguments +SDcpu, and +SDio.

While a process executes a dirty NIF, some operations that communicate with it can take a very long time to complete. Suspend or garbage collection of a process executing a dirty NIF cannot be done until the dirty NIF has returned. Thus, other processes waiting for such operations to complete might have to wait for a very long time. Blocking multi-scheduling, that is, calling erlang:system_flag(multi_scheduling, block), can also take a very long time to complete. This is because all ongoing dirty operations on all dirty schedulers must complete before the block operation can complete.

Many operations communicating with a process executing a dirty NIF can, however, complete while it executes the dirty NIF. For example, retrieving information about it through process_info, setting its group leader, register/unregister its name, and so on.

Termination of a process executing a dirty NIF can only be completed up to a certain point while it executes the dirty NIF. All Erlang resources, such as its registered name and its ETS tables, are released. All links and monitors are triggered. The execution of the NIF is, however, not stopped. The NIF can safely continue execution, allocate heap memory, and so on, but it is of course better to stop executing as soon as possible. The NIF can check whether a current process is alive using enif_is_current_process_alive. Communication using enif_send and enif_port_command is also dropped when the sending process is not alive. Deallocating certain internal resources, such as process heap and process control block, is delayed until the dirty NIF has completed.
Initialization

ERL_NIF_INIT(MODULE, ErlNifFunc funcs[], load, NULL, upgrade, unload)

This is the magic macro to initialize a NIF library. It is to be evaluated in global file scope.

MODULE is the name of the Erlang module as an identifier without string quotations. It is stringified by the macro.

funcs is a static array of function descriptors for all the implemented NIFs in this library.

load, upgrade and unload are pointers to functions. One of load or upgrade is called to initialize the library. unload is called to release the library. All are described individually below.

The fourth argument NULL is ignored. It was earlier used for the deprecated reload callback which is no longer supported since OTP 20.

If compiling a NIF lib for static inclusion through --enable-static-nifs, then the macro STATIC_ERLANG_NIF_LIBNAME must be defined as the name of the archive file (excluding file extension .a) without string quotations. It must only contain characters allowed in a C identifier. The macro must be defined before erl_nif.h is included. If the older macro STATIC_ERLANG_NIF is instead used, then the name of the archive file must match the name of the module.

int (*load)(ErlNifEnv* caller_env, void** priv_data, ERL_NIF_TERM load_info)

load is called when the NIF library is loaded and no previously loaded library exists for this module.

*priv_data can be set to point to some private data if the library needs to keep a state between NIF calls. enif_priv_data returns this pointer. *priv_data is initialized to NULL when load is called.

load_info is the second argument to erlang:load_nif/2.

The library fails to load if load returns anything other than 0. load can be NULL if initialization is not needed.

int (*upgrade)(ErlNifEnv* caller_env, void** priv_data, void** old_priv_data, ERL_NIF_TERM load_info)

upgrade is called when the NIF library is loaded and there is old code of this module with a loaded NIF library.

Works as load, except that *old_priv_data already contains the value set by the last call to load or upgrade for the old module code. *priv_data is initialized to NULL when upgrade is called. It is allowed to write to both *priv_data and *old_priv_data.

The library fails to load if upgrade returns anything other than 0 or if upgrade is NULL.

void (*unload)(ErlNifEnv* caller_env, void* priv_data)

unload is called when the module code that the NIF library belongs to is purged as old. New code of the same module may or may not exist.

Data Types

ERL_NIF_TERM

Variables of type ERL_NIF_TERM can refer to any Erlang term. This is an opaque type and values of it can only by used either as arguments to API functions or as return values from NIFs. All ERL_NIF_TERMs belong to an environment (ErlNifEnv). A term cannot be destructed individually, it is valid until its environment is destructed.

ErlNifEnv

ErlNifEnv represents an environment that can host Erlang terms. All terms in an environment are valid as long as the environment is valid. ErlNifEnv is an opaque type; pointers to it can only be passed on to API functions. Three types of environments exist:
Process bound environment

Passed as the first argument to all NIFs. All function arguments passed to a NIF belong to that environment. The return value from a NIF must also be a term belonging to the same environment.

A process bound environment contains transient information about the calling Erlang process. The environment is only valid in the thread where it was supplied as argument until the NIF returns. It is thus useless and dangerous to store pointers to process bound environments between NIF calls.

Callback environment

Passed as the first argument to all the non-NIF callback functions (load, upgrade, unload, dtor, down, stop and dyncall). Works like a process bound environment but with a temporary pseudo process that "terminates" when the callback has returned. Terms may be created in this environment but they will only be accessible during the callback.

Process independent environment

Created by calling enif_alloc_env. This environment can be used to store terms between NIF calls and to send terms with enif_send. A process independent environment with all its terms is valid until you explicitly invalidate it with enif_free_env or enif_send.

All contained terms of a list/tuple/map must belong to the same environment as the list/tuple/map itself. Terms can be copied between environments with enif_make_copy.

ErlNifFunc

typedef struct {
    const char* name;
    unsigned arity;
    ERL_NIF_TERM (*fptr)(ErlNifEnv* env, int argc, const ERL_NIF_TERM argv[]);
    unsigned flags;
} ErlNifFunc;

Describes a NIF by its name, arity, and implementation.

fptr

A pointer to the function that implements the NIF.

argv

Contains the function arguments passed to the NIF.

argc

The array length, that is, the function arity. argv[N-1] thus denotes the Nth argument to the NIF. Notice that the argument argc allows for the same C function to implement several Erlang functions with different arity (but probably with the same name).

flags

Is 0 for a regular NIF (and so its value can be omitted for statically initialized ErlNifFunc instances). flags can be used to indicate that the NIF is a dirty NIF that is to be executed on a dirty scheduler thread. If the dirty NIF is expected to be CPU-bound, its flags field is to be set to ERL_NIF_DIRTY_JOB_CPU_BOUND or ERL_NIF_DIRTY_JOB_IO_BOUND.

Note:

If one of the ERL_NIF_DIRTY_JOB_*_BOUND flags is set, and the runtime system has no support for dirty schedulers, the runtime system refuses to load the NIF library.
ErlNifBinary

typedef struct {
    size_t size;
    unsigned char* data;
} ErlNifBinary;

ErlNifBinary contains transient information about an inspected binary term. data is a pointer to a buffer of size bytes with the raw content of the binary.

Notice that ErlNifBinary is a semi-opaque type and you are only allowed to read fields size and data.

ErlNifBinaryToTerm

An enumeration of the options that can be specified to enif_binary_to_term. For default behavior, use value 0.

When receiving data from untrusted sources, use option ERL_NIF_BIN2TERM_SAFE.

ErlNifMonitor

This is an opaque data type that identifies a monitor.

The nif writer is to provide the memory for storing the monitor when calling enif_monitor_process. The address of the data is not stored by the runtime system, so ErlNifMonitor can be used as any other data, it can be copied, moved in memory, forgotten, and so on. To compare two monitors, enif_compare_monitors must be used.

ErlNifOnHaltCallback

typedef void ErlNifOnHaltCallback(void *priv_data);

The function prototype of an on halt callback function.

An on halt callback can be installed using enif_set_option(). Such an installed callback will be called when the runtime system is halting.

ErlNifOption

An enumeration of the options that can be set using enif_set_option().

Currently valid options:

ERL_NIF_OPT_DELAY_HALT

Enable delay of runtime system halt with flushing enabled until all calls to NIFs in the NIF library have returned.

ERL_NIF_OPT_ON_HALT

Install a callback that will be called when the runtime system halts with flushing enabled.

ErlNifPid

A process identifier (pid). In contrast to pid terms (instances of ERL_NIF_TERM), ErlNifPids are self-contained and not bound to any environment. ErlNifPid is an opaque type. It can be copied, moved in memory, forgotten, and so on.

ErlNifPort

A port identifier. In contrast to port ID terms (instances of ERL_NIF_TERM), ErlNifPorts are self-contained and not bound to any environment. ErlNifPort is an opaque type. It can be copied, moved in memory, forgotten, and so on.
ErlNifResourceType

Each instance of ErlNifResourceType represents a class of memory-managed resource objects that can be garbage collected. Each resource type has a unique name and a destructor function that is called when objects of its type are released.

ErlNifResourceTypeInit

typedef struct {
    ErlNifResourceDtor* dtor;       // #1 Destructor
    ErlNifResourceStop* stop;       // #2 Select stop
    ErlNifResourceDown* down;       // #3 Monitor down
    int members;
    ErlNifResourceDynCall* dyncall; // #4 Dynamic call
} ErlNifResourceTypeInit;

Initialization structure read by enif_open_resource_type_x enif_init_resource_type.

ErlNifResourceDtor

typedef void ErlNifResourceDtor(ErlNifEnv* caller_env, void* obj);

The function prototype of a resource destructor function.
The obj argument is a pointer to the resource. The only allowed use for the resource in the destructor is to access its user data one final time. The destructor is guaranteed to be the last callback before the resource is deallocated.

ErlNifResourceDown

typedef void ErlNifResourceDown(ErlNifEnv* caller_env, void* obj, ErlNifPid* pid, ErlNifMonitor* mon);

The function prototype of a resource down function, called on the behalf of enif_monitor_process. obj is the resource, pid is the identity of the monitored process that is exiting, and mon is the identity of the monitor.

ErlNifResourceStop

typedef void ErlNifResourceStop(ErlNifEnv* caller_env, void* obj, ErlNifEvent event, int is_direct_call);

The function prototype of a resource stop function, called on the behalf of enif_select. obj is the resource, event is OS event, is_direct_call is true if the call is made directly from enif_select or false if it is a scheduled call (potentially from another thread).

ErlNifResourceDynCall

typedef void ErlNifResourceDynCall(ErlNifEnv* caller_env, void* obj, void* call_data);

The function prototype of a dynamic resource call function, called by enif_dynamic_resource_call. Argument obj is the resource object and call_data is the last argument to enif_dynamic_resource_call passed through.

ErlNifCharEncoding

typedef enum {
    ERL_NIF_LATIN1,
    ERL_NIF_UTF8,
}ErlNifCharEncoding;

The character encoding used in strings and atoms. The only supported encodings are ERL_NIF_LATIN1 for ISO Latin-1 (8-bit ASCII) and ERL_NIF_UTF8 for UTF-8.

ErlNifSysInfo

Used by enif_system_info to return information about the runtime system. Contains the same content as ErlDrvSysInfo.
ErlNifSInt64
   A native signed 64-bit integer type.
ErlNifUInt64
   A native unsigned 64-bit integer type.
ErlNifTime
   A signed 64-bit integer type for representation of time.
ErlNifTimeUnit
   An enumeration of time units supported by the NIF API:
      ERL_NIF_SEC
         Seconds
      ERL_NIF_MSEC
         Milliseconds
      ERL_NIF_USEC
         Microseconds
      ERL_NIF_NSEC
         Nanoseconds
ErlNifUniqueInteger
   An enumeration of the properties that can be requested from enif_make_unique_integer. For default properties, use value 0.
      ERL_NIF_UNIQUE_POSITIVE
         Return only positive integers.
      ERL_NIF_UNIQUE_MONOTONIC
         Return only strictly monotonically increasing integer corresponding to creation time.
ErlNifHash
   An enumeration of the supported hash types that can be generated using enif_hash.
      ERL_NIF_INTERNAL_HASH
         Non-portable hash function that only guarantees the same hash for the same term within one Erlang VM instance.
         It takes 32-bit salt values and generates hashes within 0..2^32-1.
      ERL_NIF_PHASH2
         Portable hash function that gives the same hash for the same Erlang term regardless of machine architecture and ERTS version.
            It ignores salt values and generates hashes within 0..2^27-1.
            Slower than ERL_NIF_INTERNAL_HASH. It corresponds to erlang:phash2/1.
SysIOVec
   A system I/O vector, as used by writev on Unix and WSASend on Win32. It is used in ErlNifIOVec and by enif_ioq_peek.
ErlNifIOVec

typedef struct {
    int iovcnt;
    size_t size;
    SysIOVec* iov;
} ErlNifIOVec;

An I/O vector containing iovcnt SysIOVecs pointing to the data. It is used by enif_inspect_iovec and enif_ioq_enqv.

ErlNifIOQueueOpts

Options to configure a ErlNifIOQueue.

ERL_NIF_IOQ_NORMAL

Create a normal I/O Queue

Exports

void *enif_alloc(size_t size)

Allocates memory of size bytes.

Returns NULL if the allocation fails.

The returned pointer is suitably aligned for any built-in type that fit in the allocated memory.

int enif_alloc_binary(size_t size, ErlNifBinary* bin)

Allocates a new binary of size bytes. Initializes the structure pointed to by bin to refer to the allocated binary. The binary must either be released by enif_release_binary or ownership transferred to an Erlang term with enif_make_binary. An allocated (and owned) ErlNifBinary can be kept between NIF calls.

If you do not need to reallocate or keep the data alive across NIF calls, consider using enif_make_new_binary instead as it will allocate small binaries on the process heap when possible.

Returns true on success, or false if allocation fails.

ErlNifEnv *enif_alloc_env()

Allocates a new process independent environment. The environment can be used to hold terms that are not bound to any process. Such terms can later be copied to a process environment with enif_make_copy or be sent to a process as a message with enif_send.

Returns pointer to the new environment.

void *enif_alloc_resource(ErlNifResourceType* type, unsigned size)

Allocates a memory-managed resource object of type type and size bytes.

size_t enif_binary_to_term(ErlNifEnv *env, const unsigned char* data, size_t size, ERL_NIF_TERM *term, unsigned int opts)

Creates a term that is the result of decoding the binary data at data, which must be encoded according to the Erlang external term format. No more than size bytes are read from data. Argument opts corresponds to the second argument to erlang:binary_to_term/2 and must be either 0 or ERL_NIF_BIN2TERM_SAFE.

On success, stores the resulting term at *term and returns the number of bytes read. Returns 0 if decoding fails or if opts is invalid.
See also ErlNifBinaryToTerm, erlang:binary_to_term/2, and enif_term_to_binary.

void enif_clear_env(ErlNifEnv* env)
Frees all terms in an environment and clears it for reuse. The environment must have been allocated with enif_alloc_env.

int enif_compare(ERL_NIF_TERM lhs, ERL_NIF_TERM rhs)
Returns an integer < 0 if lhs < rhs, 0 if lhs = rhs, and > 0 if lhs > rhs. Corresponds to the Erlang operators ==, /=, =<, <, =>, and > (but not =:= or /=).

int enif_compare_monitors(const ErlNifMonitor *monitor1, const ErlNifMonitor *monitor2)
Compares two ErlNifMonitors. Can also be used to imply some artificial order on monitors, for whatever reason.
Returns 0 if monitor1 and monitor2 are equal, < 0 if monitor1 < monitor2, and > 0 if monitor1 > monitor2.

int enif_compare_pids(const ErlNifPid *pid1, const ErlNifPid *pid2)
Compares two ErlNifPids according to term order.
Returns 0 if pid1 and pid2 are equal, < 0 if pid1 < pid2, and > 0 if pid1 > pid2.

void enif_cond_broadcast(ErlNifCond *cnd)
Same as erl_drv_cond_broadcast.

ErlNifCond *enif_cond_create(char *name)
Same as erl_drv_cond_create.

void enif_cond_destroy(ErlNifCond *cnd)
Same as erl_drv_cond_destroy.

char*enif_cond_name(ErlNifCond* cnd)
Same as erl_drv_cond_name.

void enif_cond_signal(ErlNifCond *cnd)
Same as erl_drv_cond_signal.

void enif_cond_wait(ErlNifCond *cnd, ErlNifMutex *mtx)
Same as erl_drv_cond_wait.

int enif_consume_timeslice(ErlNifEnv *env, int percent)
Gives the runtime system a hint about how much CPU time the current NIF call has consumed since the last hint, or since the start of the NIF if no previous hint has been specified. The time is specified as a percent of the timeslice that a process is allowed to execute Erlang code until it can be suspended to give time for other runnable processes. The scheduling timeslice is not an exact entity, but can usually be approximated to about 1 millisecond.
Notice that it is up to the runtime system to determine if and how to use this information. Implementations on some platforms can use other means to determine consumed CPU time. Lengthy NIFs should regardless of this frequently call enif_consume_timeslice to determine if it is allowed to continue execution.

Argument percent must be an integer between 1 and 100. This function must only be called from a NIF-calling thread, and argument env must be the environment of the calling process.

Returns 1 if the timeslice is exhausted, otherwise 0. If 1 is returned, the NIF is to return as soon as possible in order for the process to yield.

This function is provided to better support co-operative scheduling, improve system responsiveness, and make it easier to prevent misbehaviors of the VM because of a NIF monopolizing a scheduler thread. It can be used to divide length work into a number of repeated NIF calls without the need to create threads.

See also the warning text at the beginning of this manual page.

ErlNifTime enif_convert_time_unit(ErlNifTime val, ErlNifTimeUnit from, ErlNifTimeUnit to)

Converts the val value of time unit from to the corresponding value of time unit to. The result is rounded using the floor function.

val
  Value to convert time unit for.
from
  Time unit of val.
to
  Time unit of returned value.

Returns ERL_NIF_TIME_ERROR if called with an invalid time unit argument.

See also ErlNifTime and ErlNifTimeUnit.

ERL_NIF_TERM enif_cpu_time(ErlNifEnv *)

Returns the CPU time in the same format as erlang:timestamp(). The CPU time is the time the current logical CPU has spent executing since some arbitrary point in the past. If the OS does not support fetching this value, enif_cpu_time invokes enif_make_badarg.

int enif_demonitor_process(ErlNifEnv* caller_env, void* obj, const ErlNifMonitor* mon)

Cancels a monitor created earlier with enif_monitor_process. Argument obj is a pointer to the resource holding the monitor and *mon identifies the monitor.

Argument caller_env is the environment of the calling thread (process bound or callback environment) or NULL if calling from a custom thread not spawned by ERTS.

Returns 0 if the monitor was successfully identified and removed. Returns a non-zero value if the monitor could not be identified, which means it was either

- never created for this resource
- already cancelled
- already triggered
- just about to be triggered by a concurrent thread

This function is thread-safe.
int enif_dynamic_resource_call(ErlNifEnv* caller_env, ERL_NIF_TERM rt_module, ERL_NIF_TERM rt_name, ERL_NIF_TERM resource, void* call_data)

Call code of a resource type implemented by another NIF module. The atoms rt_module and rt_name identifies the resource type to be called. Argument resource identifies a resource object of that type.

The callback dyncall of the identified resource type will be called with a pointer to the resource objects obj and the argument call_data passed through. The call_data argument is typically a pointer to a struct used to passed both arguments to the dyncall function as well as results back to the caller.

Returns 0 if the dyncall callback function was called. Returns a non-zero value if no call was made, which happens if rt_module and rt_name did not identify a resource type with a dyncall callback or if resource was not a resource object of that type.

int enif_equal_tids(ErlNifTid tid1, ErlNifTid tid2)

Same as erl_drv_equal_tids.

int enif_fprintf(FILE *stream, const char *format, ...)

Similar to fprintf but this format string also accepts "%T", which formats Erlang terms of type ERL_NIF_TERM. This function is primarily intended for debugging purpose. It is not recommended to print very large terms with %T. The function may change errno, even if successful.

void enif_free(void* ptr)

Frees memory allocated by enif_alloc.

void enif_free_env(ErlNifEnv* env)

Frees an environment allocated with enif_alloc_env. All terms created in the environment are freed as well.

void enif_free_iovec(ErlNifIOVec* iov)

Frees an io vector returned from enif_inspect_iovec. This is needed only if a NULL environment is passed to enif_inspect_iovec.

ErlNifIOVec *iovec = NULL;
size_t max_elements = 128;
ERL_NIF_TERM tail;
if (!enif_inspect_iovec(NULL, max_elements, term, &tail, &iovec))
    return 0;

// Do things with the iovec
/* Free the iovector, possibly in another thread or nif function call */
enif_free_iovec(iovec);

int enif_get_atom(ErlNifEnv *env, ERL_NIF_TERM term, char *buf, unsigned size, ERL_NifCharEncoding encoding)

Writes a NULL-terminated string in the buffer pointed to by buf of size size bytes, consisting of the string representation of the atom term with encoding.

Returns the number of bytes written (including terminating NULL character) or 0 if term is not an atom with maximum length of size-1 bytes in encoding.
int enif_get_atom_length(ErlNifEnv *env, ERL_NIF_TERM term, unsigned *len, ErlNifCharEncoding encoding)
Sets *len to the length (number of bytes excluding terminating NULL character) of the atom term with encoding. Returns true on success, or false if term is not an atom or if the atom cannot be encoded using encoding.

int enif_get_double(ErlNifEnv* env, ERL_NIF_TERM term, double* dp)
Sets *dp to the floating-point value of term. Returns true on success, or false if term is not a float.

int enif_get_int(ErlNifEnv* env, ERL_NIF_TERM term, int* ip)
Sets *ip to the integer value of term. Returns true on success, or false if term is not an integer or is outside the bounds of type int.

int enif_get_int64(ErlNifEnv* env, ERL_NIF_TERM term, ErlNifSInt64* ip)
Sets *ip to the integer value of term. Returns true on success, or false if term is not an integer or is outside the bounds of a signed 64-bit integer.

int enif_get_local_pid(ErlNifEnv* env, ERL_NIF_TERM term, ErlNifPid* pid)
If term is the pid of a node local process, this function initializes the pid variable *pid from it and returns true. Otherwise returns false. No check is done to see if the process is alive.

Note:
enif_get_local_pid will return false if argument term is the atom undefined.

int enif_get_local_port(ErlNifEnv* env, ERL_NIF_TERM term, ErlNifPort* port_id)
If term identifies a node local port, this function initializes the port variable *port_id from it and returns true. Otherwise returns false. No check is done to see if the port is alive.

int enif_get_list_cell(ErlNifEnv* env, ERL_NIF_TERM list, ERL_NIF_TERM* head, ERL_NIF_TERM* tail)
Sets *head and *tail from list list. Returns true on success, or false if it is not a list or the list is empty.

int enif_get_list_length(ErlNifEnv* env, ERL_NIF_TERM term, unsigned* len)
Sets *len to the length of list term. Returns true on success, or false if term is not a proper list.

int enif_get_long(ErlNifEnv* env, ERL_NIF_TERM term, long int* ip)
Sets *ip to the long integer value of term. Returns true on success, or false if term is not an integer or is outside the bounds of type long int.
int enif_get_map_size(ErlNifEnv* env, ERL_NIF_TERM term, size_t *size)
Sets *size to the number of key-value pairs in the map term.
Returns true on success, or false if term is not a map.

int enif_get_map_value(ErlNifEnv* env, ERL_NIF_TERM map, ERL_NIF_TERM key, ERL_NIF_TERM* value)
Sets *value to the value associated with key in the map map.
Returns true on success, or false if map is not a map or if map does not contain key.

int enif_get_resource(ErlNifEnv* env, ERL_NIF_TERM term, ErlNifResourceType* type, void** objp)
Sets *objp to point to the resource object referred to by term.
Returns true on success, or false if term is not a handle to a resource object of type type.
enif_get_resource does not add a reference to the resource object. However, the pointer received in *objp is guaranteed to be valid at least as long as the resource handle term is valid.

int enif_get_string(ErlNifEnv* env, ERL_NIF_TERM list, char* buf, unsigned size, ErlNifCharEncoding encoding)
Writes a NULL-terminated string in the buffer pointed to by buf with size size, consisting of the characters in the string list. The characters are written using encoding.
Returns one of the following:
• The number of bytes written (including terminating NULL character)
• -size if the string was truncated because of buffer space
• 0 if list is not a string that can be encoded with encoding or if size was <1.
The written string is always NULL-terminated, unless buffer size is <1.

int enif_get_string_length(ErlNifEnv *env, ERL_NIF_TERM list, unsigned *len, ErlNifCharEncoding encoding)
Sets *len to the length (number of bytes excluding terminating NULL character) of the string list with encoding.
Returns true on success, or false if list is not a string that can be encoded with encoding.

int enif_get_tuple(ErlNifEnv* env, ERL_NIF_TERM term, int* arity, const ERL_NIF_TERM** array)
If term is a tuple, this function sets *array to point to an array containing the elements of the tuple, and sets *arity to the number of elements. Notice that the array is read-only and (*array)[N-1] is the Nth element of the tuple. *array is undefined if the arity of the tuple is zero.
Returns true on success, or false if term is not a tuple.

int enif_get_uint(ErlNifEnv* env, ERL_NIF_TERM term, unsigned int* ip)
Sets *ip to the unsigned integer value of term.
Returns true on success, or false if term is not an unsigned integer or is outside the bounds of type unsigned int.
int enif_get_uint64(ErlNifEnv* env, ERL_NIF_TERM term, ErlNifUInt64* ip)
Sets *ip to the unsigned integer value of term.
Returns true on success, or false if term is not an unsigned integer or is outside the bounds of an unsigned 64-bit integer.

int enif_get_ulong(ErlNifEnv* env, ERL_NIF_TERM term, unsigned long* ip)
Sets *ip to the unsigned long integer value of term.
Returns true on success, or false if term is not an unsigned integer or is outside the bounds of type unsigned long.

int enif_getenv(const char* key, char* value, size_t *value_size)
Same as erl_drv_getenv.

int enif_has_pending_exception(ErlNifEnv* env, ERL_NIF_TERM* reason)
Returns true if a pending exception is associated with the environment env. If reason is a NULL pointer, ignore it. Otherwise, if a pending exception associated with env exists, set *reason to the value of the exception term. For example, if enif_make_badarg is called to set a pending badarg exception, a later call to enif_has_pending_exception(env, &reason) sets *reason to the atom badarg, then return true.
See also enif_make_badarg and enif_raise_exception.

ErlNifUint64 enif_hash(ErlNifHash type, ERL_NIF_TERM term, ErlNifUInt64 salt)
Hashes term according to the specified ErlNifHash type.
Ranges of taken salt (if any) and returned value depend on the hash type.

int enif_inspect_binary(ErlNifEnv* env, ERL_NIF_TERM bin_term, ErlNifBinary* bin)
Initializes the structure pointed to by bin with information about binary term bin_term.
Returns true on success, or false if bin_term is not a binary.

int enif_inspect_iolist_as_binary(ErlNifEnv* env, ERL_NIF_TERM term, ErlNifBinary* bin)
Initializes the structure pointed to by bin with a continuous buffer with the same byte content as iolist. As with inspect_binary, the data pointed to by bin is transient and does not need to be released.
Returns true on success, or false if iolist is not an iolist.

int enif_inspect_iovec(ErlNifEnv* env, size_t max_elements, ERL_NIF_TERM iovec_term, ERL_NIF_TERM* tail, ErlNifIOVec** iovec)
Fills iovec with the list of binaries provided in iovec_term. The number of elements handled in the call is limited to max_elements, and tail is set to the remainder of the list. Note that the output may be longer than max_elements on some platforms.
To create a list of binaries from an arbitrary iolist, use erlang:iolist_to_iovec/1.
When calling this function, iovec should contain a pointer to NULL or a ErlNifIOVec structure that should be used if possible. e.g.
/* Don't use a pre-allocated structure */
ErlnifIOVec *iovec = NULL;
enif_inspect_iovec(env, max_elements, term, &tail, &iovec);

/* Use a stack-allocated vector as an optimization for vectors with few elements */
ErlnifIOVec vec, *iovec = &vec;
enif_inspect_iovec(env, max_elements, term, &tail, &iovec);

The contents of the iovec is valid until the called nif function returns. If the iovec should be valid after the nif call returns, it is possible to call this function with a NULL environment. If no environment is given the iovec owns the data in the vector and it has to be explicitly freed using enif_free_iovec.

Returns true on success, or false if iovec_term not an iovec.

ErlnifIOQueue *enif_ioq_create(ErlNifIOQueueOpts opts)
Create a new I/O Queue that can be used to store data. opts has to be set to ERL_NIF_IOQ_NORMAL.

void enif_ioq_destroy(ErlNifIOQueue *q)
Destroy the I/O queue and free all of its contents.

int enif_ioq_deq(ErlNifIOQueue *q, size_t count, size_t *size)
Dequeue count bytes from the I/O queue. If size is not NULL, the new size of the queue is placed there.
Returns true on success, or false if the I/O does not contain count bytes. On failure the queue is left un-altered.

int enif_ioq_enq_binary(ErlNifIOQueue *q, ErlNifBinary *bin, size_t skip)
Enqueue the bin into q skipping the first skip bytes.
Returns true on success, or false if skip is greater than the size of bin. Any ownership of the binary data is transferred to the queue and bin is to be considered read-only for the rest of the NIF call and then as released.

int enif_ioq_enqv(ErlNifIOQueue *q, ErlNifIOVec *iovec, size_t skip)
Enqueue the iovec into q skipping the first skip bytes.
Returns true on success, or false if skip is greater than the size of iovec.

SysIOVec *enif_ioq_peek(ErlNifIOQueue *q, int *iovlen)
Get the I/O queue as a pointer to an array of SysIOVecs. It also returns the number of elements in iovlen.
Nothing is removed from the queue by this function, that must be done with enif_ioq_deq.
The returned array is suitable to use with the Unix system call writev.

int enif_ioq_peek_head(ErlNifEnv *env, ErlNifIOQueue *q, size_t *size, ERL_NIF_TERM *bin_term)
Get the head of the IO Queue as a binary term.
If size is not NULL, the size of the head is placed there.
Nothing is removed from the queue by this function, that must be done with enif_ioq_deq.
Returns true on success, or false if the queue is empty.
size_t enif_ioq_size(ErlNifIOQueue *q)
Get the size of q.

int enif_is_atom(ErlNifEnv* env, ERL_NIF_TERM term)
Returns true if term is an atom.

int enif_is_binary(ErlNifEnv* env, ERL_NIF_TERM term)
Returns true if term is a binary.

int enif_is_current_process_alive(ErlNifEnv* env)
Returns true if the currently executing process is currently alive, otherwise false.
This function can only be used from a NIF-calling thread, and with an environment corresponding to currently executing processes.

int enif_is_empty_list(ErlNifEnv* env, ERL_NIF_TERM term)
Returns true if term is an empty list.

int enif_is_exception(ErlNifEnv* env, ERL_NIF_TERM term)
Return true if term is an exception.

int enif_is_fun(ErlNifEnv* env, ERL_NIF_TERM term)
Returns true if term is a fun.

int enif_is_identical(ERL_NIF_TERM lhs, ERL_NIF_TERM rhs)
Returns true if the two terms are identical. Corresponds to the Erlang operators =:= and /=.

int enif_is_list(ErlNifEnv* env, ERL_NIF_TERM term)
Returns true if term is a list.

int enif_is_map(ErlNifEnv* env, ERL_NIF_TERM term)
Returns true if term is a map, otherwise false.

int enif_is_number(ErlNifEnv* env, ERL_NIF_TERM term)
Returns true if term is a number.

int enif_is_pid(ErlNifEnv* env, ERL_NIF_TERM term)
Returns true if term is a pid.

int enif_is_pid_undefined(const ErlNifPid* pid)
Returns true if pid has been set as undefined by enif_set_pid_undefined.
int enif_is_port(ErlNifEnv* env, ERL_NIF_TERM term)
Returns true if term is a port.

int enif_is_port_alive(ErlNifEnv* env, ErlNifPort *port_id)
Returns true if port_id is alive.
This function is thread-safe.

int enif_is_process_alive(ErlNifEnv* env, ErlNifPid *pid)
Returns true if pid is alive.
This function is thread-safe.

int enif_is_ref(ErlNifEnv* env, ERL_NIF_TERM term)
Returns true if term is a reference.

int enif_is_tuple(ErlNifEnv* env, ERL_NIF_TERM term)
Returns true if term is a tuple.

int enif_keep_resource(void* obj)
Adds a reference to resource object obj obtained from enif_alloc_resource. Each call to
enif_keep_resource for an object must be balanced by a call to enif_release_resource before the object
is destructed.

ERL_NIF_TERM enif_make_atom(ErlNifEnv *env, const char *name)
Creates an atom term from the NULL-terminated C-string name with ISO Latin-1 encoding. If the length
of name exceeds the maximum length allowed for an atom (255 characters), enif_make_atom invokes
enif_make_badarg.

ERL_NIF_TERM enif_make_atom_len(ErlNifEnv *env, const char *name, size_t len)
Create an atom term from the string name with length len and ISO Latin-1 encoding. NULL characters are treated as
any other characters. If len exceeds the maximum length allowed for an atom (255 characters), enif_make_atom
invokes enif_make_badarg.

ERL_NIF_TERM enif_make_badarg(ErlNifEnv* env)
Makes a badarg exception to be returned from a NIF, and associates it with environment env. Once a NIF or any
function it calls invokes enif_make_badarg, the runtime ensures that a badarg exception is raised when the NIF
returns, even if the NIF attempts to return a non-exception term instead.
The return value from enif_make_badarg can be used only as the return value from the NIF that invoked it
(directly or indirectly) or be passed to enif_is_exception, but not to any other NIF API function.
See also enif_has_pending_exception and enifRaise_exception.
Note:

Before ERTS 7.0 (Erlang/OTP 18), the return value from enif_make_badarg had to be returned from the NIF. This requirement is now lifted as the return value from the NIF is ignored if enif_make_badarg has been invoked.

ERL_NIF_TERM enif_make_binary(ErlNifEnv* env, ErlNifBinary* bin)
Makes a binary term from bin. Any ownership of the binary data is transferred to the created term and bin is to be considered read-only for the rest of the NIF call and then as released.

ERL_NIF_TERM enif_make_copy(ErlNifEnv* dst_env, ERL_NIF_TERM src_term)
Makes a copy of term src_term. The copy is created in environment dst_env. The source term can be located in any environment.

ERL_NIF_TERM enif_make_double(ErlNifEnv* env, double d)
Creates a floating-point term from a double. If argument double is not finite or is NaN, enif_make_double invokes enif_make_badarg.

int enif_make_existing_atom(ErlNifEnv *env, const char *name, ERL_NIF_TERM *atom, ErlNifCharEncoding encoding)
Tries to create the term of an already existing atom from the NULL-terminated C-string name with encoding. If the atom already exists, this function stores the term in *atom and returns true, otherwise returns false. It also returns false if the string name exceeds the maximum length allowed for an atom (255 characters) or if name is not correctly encoded.

int enif_make_existing_atom_len(ErlNifEnv *env, const char *name, size_t len, ERL_NIF_TERM *atom, ErlNifCharEncoding encoding)
Tries to create the term of an already existing atom from the string name with length len bytes and encoding. NULL characters are treated as any other characters. If the atom already exists, this function stores the term in *atom and returns true, otherwise returns false. It also returns false if the string name exceeds the maximum length allowed for an atom (255 characters) or if name is not correctly encoded.

ERL_NIF_TERM enif_make_int(ErlNifEnv* env, int i)
Creates an integer term.

ERL_NIF_TERM enif_make_int64(ErlNifEnv* env, ErlNifSInt64 i)
Creates an integer term from a signed 64-bit integer.

ERL_NIF_TERM enif_make_list(ErlNifEnv* env, unsigned cnt, ...)
Creates an ordinary list term of length cnt. Expects cnt number of arguments (after cnt) of type ERL_NIF_TERM as the elements of the list.
Returns an empty list if cnt is 0.
ERL_NIF_TERM enif_make_list1(ErlNifEnv* env, ERL_NIF_TERM e1)
ERL_NIF_TERM enif_make_list2(ErlNifEnv* env, ERL_NIF_TERM e1, ERL_NIF_TERM e2)
ERL_NIF_TERM enif_make_list3(ErlNifEnv* env, ERL_NIF_TERM e1, ERL_NIF_TERM e2, ERL_NIF_TERM e3)
ERL_NIF_TERM enif_make_list4(ErlNifEnv* env, ERL_NIF_TERM e1, ..., ERL_NIF_TERM e4)
ERL_NIF_TERM enif_make_list5(ErlNifEnv* env, ERL_NIF_TERM e1, ..., ERL_NIF_TERM e5)
ERL_NIF_TERM enif_make_list6(ErlNifEnv* env, ERL_NIF_TERM e1, ..., ERL_NIF_TERM e6)
ERL_NIF_TERM enif_make_list7(ErlNifEnv* env, ERL_NIF_TERM e1, ..., ERL_NIF_TERM e7)
ERL_NIF_TERM enif_make_list8(ErlNifEnv* env, ERL_NIF_TERM e1, ..., ERL_NIF_TERM e8)
ERL_NIF_TERM enif_make_list9(ErlNifEnv* env, ERL_NIF_TERM e1, ..., ERL_NIF_TERM e9)

Creates an ordinary list term with length indicated by the function name. Prefer these functions (macros) over the variadic enif_make_list to get a compile-time error if the number of arguments does not match.

ERL_NIF_TERM enif_make_list_cell(ErlNifEnv* env, ERL_NIF_TERM head, ERL_NIF_TERM tail)

Creates a list cell [head | tail].

ERL_NIF_TERM enif_make_list_from_array(ErlNifEnv* env, const ERL_NIF_TERM arr[], unsigned cnt)

Creates an ordinary list containing the elements of array arr of length cnt.

Returns an empty list if cnt is 0.

ERL_NIF_TERM enif_make_long(ErlNifEnv* env, long int i)

Creates an integer term from a long int.

int enif_make_map_put(ErlNifEnv* env, ERL_NIF_TERM map_in, ERL_NIF_TERM key, ERL_NIF_TERM value, ERL_NIF_TERM* map_out)

Makes a copy of map map_in and inserts key with value. If key already exists in map_in, the old associated value is replaced by value.

If successful, this function sets *map_out to the new map and returns true. Returns false if map_in is not a map.

The map_in term must belong to environment env.

int enif_make_map_remove(ErlNifEnv* env, ERL_NIF_TERM map_in, ERL_NIF_TERM key, ERL_NIF_TERM* map_out)

If map map_in contains key, this function makes a copy of map_in in *map_out, and removes key and the associated value. If map map_in does not contain key, *map_out is set to map_in.

Returns true on success, or false if map_in is not a map.
The map_in term must belong to environment env.

int enif_make_map_update(ErlNifEnv* env, ERL_NIF_TERM map_in, ERL_NIF_TERM key, ERL_NIF_TERM new_value, ERL_NIF_TERM* map_out)

Makes a copy of map map_in and replace the old associated value for key with new_value.
If successful, this function sets *map_out to the new map and returns true. Returns false if map_in is not a map or if it does not contain key.
The map_in term must belong to environment env.

int enif_make_map_from_arrays(ErlNifEnv* env, ERL_NIF_TERM keys[], ERL_NIF_TERM values[], size_t cnt, ERL_NIF_TERM *map_out)

Makes a map term from the given keys and values.
If successful, this function sets *map_out to the new map and returns true. Returns false there are any duplicate keys.
All keys and values must belong to env.

ERL_NIF_TERM enif_make_monitor_term(ErlNifEnv* env, const ErlNifMonitor* mon)
Creates a term identifying the given monitor received from enif_monitor_process.
This function is primarily intended for debugging purpose.

int enif_make_new_atom(ErlNifEnv *env, const char *name, ERL_NIF_TERM *atom, ErlNifCharEncoding encoding)

Creates an atom term from the NULL-terminated C-string name with encoding.
If successful, true is returned and the atom term is stored in *atom.
Otherwise, false is returned if the length of name exceeds the maximum length allowed for an atom (255 characters) or if name is not correctly encoded.

int enif_make_new_atom_len(ErlNifEnv *env, const char *name, size_t len, ERL_NIF_TERM *atom, ErlNifCharEncoding encoding)

Create an atom term from string name with length len bytes and encoding.
If successful, true is returned and atom term is stored in *atom.
Otherwise, false is returned if the string exceeds the maximum length allowed for an atom (255 characters) or if the string is not correctly encoded.

unsigned char *enif_make_new_binary(ErlNifEnv* env, size_t size, ERL_NIF_TERM* termp)

Allocates a binary of size size bytes and creates an owning term. The binary data is mutable until the calling NIF returns. This is a quick way to create a new binary without having to use ErlNifBinary. The drawbacks are that the binary cannot be kept between NIF calls and it cannot be reallocated.
Returns a pointer to the raw binary data and sets *termp to the binary term.

ERL_NIF_TERM enif_make_new_map(ErlNifEnv* env)

Makes an empty map term.
ERL_NIF_TERM enif_make_pid(ErlNifEnv* env, const ErlNifPid* pid)
Makes a pid term or the atom undefined from *pid.

ERL_NIF_TERM enif_make_ref(ErlNifEnv* env)
Creates a reference like erlang:make_ref/0.

ERL_NIF_TERM enif_make_resource(ErlNifEnv* env, void* obj)
Creates an opaque handle to a memory-managed resource object obtained by enif_alloc_resource. No ownership transfer is done, as the resource object still needs to be released by enif_release_resource. However, notice that the call to enif_release_resource can occur immediately after obtaining the term from enif_make_resource, in which case the resource object is deallocated when the term is garbage collected. For more details, see the example of creating and returning a resource object in the User's Guide.

Note:

Since ERTS 9.0 (OTP-20.0), resource terms have a defined behavior when compared and serialized through term_to_binary or passed between nodes.

• Two resource terms will compare equal if and only if they would yield the same resource object pointer when passed to enif_get_resource.
• A resource term can be serialized with term_to_binary and later be fully recreated if the resource object is still alive when binary_to_term is called. A stale resource term will be returned from binary_to_term if the resource object has been deallocated. enif_get_resource will return false for stale resource terms.

The same principles of serialization apply when passing resource terms in messages to remote nodes and back again. A resource term will act stale on all nodes except the node where its resource object is still alive in memory.

Before ERTS 9.0 (OTP-20.0), all resource terms did compare equal to each other and to empty binaries (<<>>). If serialized, they would be recreated as plain empty binaries.

ERL_NIF_TERM enif_make_resource_binary(ErlNifEnv* env, void* obj, const void* data, size_t size)
Creates a binary term that is memory-managed by a resource object obj obtained by enif_alloc_resource. The returned binary term consists of size bytes pointed to by data. This raw binary data must be kept readable and unchanged until the destructor of the resource is called. The binary data can be stored external to the resource object, in which case the destructor is responsible for releasing the data.

Several binary terms can be managed by the same resource object. The destructor is not called until the last binary is garbage collected. This can be useful to return different parts of a larger binary buffer.

As with enif_make_resource, no ownership transfer is done. The resource still needs to be released with enif_release_resource.

int enif_make_reverse_list(ErlNifEnv* env, ERL_NIF_TERM list_in, ERL_NIF_TERM *list_out)
Sets *list_out to the reverse list of the list list_in and returns true, or returns false if list_in is not a list.
This function is only to be used on short lists, as a copy is created of the list, which is not released until after the NIF returns.
The list_in term must belong to environment env.

ERL_NIF_TERM enif_make_string(ErlNifEnv *env, const char *string, ErlNifCharEncoding encoding)

Creates a list containing the characters of the NULL-terminated string string with encoding.

ERL_NIF_TERM enif_make_string_len(ErlNifEnv *env, const char *string, size_t len, ErlNifCharEncoding encoding)

Creates a list containing the characters of the string string with length len and encoding. NULL characters are treated as any other characters.

ERL_NIF_TERM enif_make_sub_binary(ErlNifEnv* env, ERL_NIF_TERM bin_term, size_t pos, size_t size)

Makes a subbinary of binary bin_term, starting at zero-based position pos with a length of size bytes. bin_term must be a binary or bitstring. pos+size must be less or equal to the number of whole bytes in bin_term.

ERL_NIF_TERM enif_make_tuple(ErlNifEnv* env, unsigned cnt, ...)

Creates a tuple term of arity cnt. Expects cnt number of arguments (after cnt) of type ERL_NIF_TERM as the elements of the tuple.

ERL_NIF_TERM enif_make_tuple1(ErlNifEnv* env, ERL_NIF_TERM e1)

ERL_NIF_TERM enif_make_tuple2(ErlNifEnv* env, ERL_NIF_TERM e1, ERL_NIF_TERM e2)

ERL_NIF_TERM enif_make_tuple3(ErlNifEnv* env, ERL_NIF_TERM e1, ERL_NIF_TERM e2, ERL_NIF_TERM e3)

ERL_NIF_TERM enif_make_tuple4(ErlNifEnv* env, ERL_NIF_TERM e1, ERL_NIF_TERM e2, ERL_NIF_TERM e3, ERL_NIF_TERM e4)

ERL_NIF_TERM enif_make_tuple5(ErlNifEnv* env, ERL_NIF_TERM e1, ERL_NIF_TERM e2, ERL_NIF_TERM e3, ERL_NIF_TERM e4, ERL_NIF_TERM e5)

ERL_NIF_TERM enif_make_tuple6(ErlNifEnv* env, ERL_NIF_TERM e1, ERL_NIF_TERM e2, ERL_NIF_TERM e3, ERL_NIF_TERM e4, ERL_NIF_TERM e5, ERL_NIF_TERM e6)

ERL_NIF_TERM enif_make_tuple7(ErlNifEnv* env, ERL_NIF_TERM e1, ERL_NIF_TERM e2, ERL_NIF_TERM e3, ERL_NIF_TERM e4, ERL_NIF_TERM e5, ERL_NIF_TERM e6, ERL_NIF_TERM e7)

ERL_NIF_TERM enif_make_tuple8(ErlNifEnv* env, ERL_NIF_TERM e1, ERL_NIF_TERM e2, ERL_NIF_TERM e3, ERL_NIF_TERM e4, ERL_NIF_TERM e5, ERL_NIF_TERM e6, ERL_NIF_TERM e7, ERL_NIF_TERM e8)

ERL_NIF_TERM enif_make_tuple9(ErlNifEnv* env, ERL_NIF_TERM e1, ERL_NIF_TERM e2, ERL_NIF_TERM e3, ERL_NIF_TERM e4, ERL_NIF_TERM e5, ERL_NIF_TERM e6, ERL_NIF_TERM e7, ERL_NIF_TERM e8, ERL_NIF_TERM e9)

Creates a tuple term with length indicated by the function name. Prefer these functions (macros) over the variadic enif_make_tuple to get a compile-time error if the number of arguments does not match.

ERL_NIF_TERM enif_make_tuple_from_array(ErlNifEnv* env, const ERL_NIF_TERM arr[], unsigned cnt)

Creates a tuple containing the elements of array arr of length cnt.

ERL_NIF_TERM enif_make_uint(ErlNifEnv* env, unsigned int i)

Creates an integer term from an unsigned int.
ERL_NIF_TERM enif_make_uint64(ErlNifEnv* env, ErlNifUInt64 i)
Creates an integer term from an unsigned 64-bit integer.

ERL_NIF_TERM enif_make_ulong(ErlNifEnv* env, unsigned long i)
Creates an integer term from an unsigned long int.

ERL_NIF_TERM enif_make_unique_integer(ErlNifEnv *env, ErlNifUniqueInteger properties)
Returns a unique integer with the same properties as specified by erlang:unique_integer/1. env is the environment to create the integer in.
ERL_NIF_UNIQUE_POSITIVE and ERL_NIF_UNIQUE_MONOTONIC can be passed as the second argument to change the properties of the integer returned. They can be combined by OR:ing the two values together.
See also ErlNifUniqueInteger.

int enif_map_iterator_create(ErlNifEnv *env, ERL_NIF_TERM map, ErlNifMapIterator *iter, ErlNifMapIteratorEntry entry)
Creates an iterator for the map map by initializing the structure pointed to by iter. Argument entry determines the start position of the iterator: ERL_NIF_MAP_ITERATOR_FIRST or ERL_NIF_MAP_ITERATOR_LAST.
Returns true on success, or false if map is not a map.
A map iterator is only useful during the lifetime of environment env that the map belongs to. The iterator must be destroyed by calling enif_map_iterator_destroy:

```c
ERL_NIF_TERM key, value;
ErlNifMapIterator iter;
enif_map_iterator_create(env, my_map, &iter, ERL_NIF_MAP_ITERATOR_FIRST);
while (enif_map_iterator_get_pair(env, &iter, &key, &value)) {
    do_something(key, value);
    enif_map_iterator_next(env, &iter);
}
enif_map_iterator_destroy(env, &iter);
```

Note:
The key-value pairs of a map have no defined iteration order. The only guarantee is that the iteration order of a single map instance is preserved during the lifetime of the environment that the map belongs to.

void enif_map_iterator_destroy(ErlNifEnv *env, ErlNifMapIterator *iter)
Destroys a map iterator created by enif_map_iterator_create.

int enif_map_iterator_get_pair(ErlNifEnv *env, ErlNifMapIterator *iter, ERL_NIF_TERM *key, ERL_NIF_TERM *value)
Gets key and value terms at the current map iterator position.
On success, sets *key and *value and returns true. Returns false if the iterator is positioned at head (before first entry) or tail (beyond last entry).
int enif_map_iterator_is_head(ErlNifEnv *env, ErlNifMapIterator *iter)
Returns true if map iterator iter is positioned before the first entry.

int enif_map_iterator_is_tail(ErlNifEnv *env, ErlNifMapIterator *iter)
Returns true if map iterator iter is positioned after the last entry.

int enif_map_iterator_next(ErlNifEnv *env, ErlNifMapIterator *iter)
Increments map iterator to point to the next key-value entry.
Returns true if the iterator is now positioned at a valid key-value entry, or false if the iterator is positioned at the tail (beyond the last entry).

int enif_map_iterator_prev(ErlNifEnv *env, ErlNifMapIterator *iter)
Decrements map iterator to point to the previous key-value entry.
Returns true if the iterator is now positioned at a valid key-value entry, or false if the iterator is positioned at the head (before the first entry).

int enif_monitor_process(ErlNifEnv* caller_env, void* obj, const ErlNifPid* target_pid, ErlNifMonitor* mon)
Starts monitoring a process from a resource. When a process is monitored, a process exit results in a call to the provided down callback associated with the resource type.
Argument obj is pointer to the resource to hold the monitor and *target_pid identifies the local process to be monitored.
If mon is not NULL, a successful call stores the identity of the monitor in the ErlNifMonitor struct pointed to by mon. This identifier is used to refer to the monitor for later removal with enif_demonitor_process or compare with enif_compare_monitors. A monitor is automatically removed when it triggers or when the resource is deallocated.
Argument caller_env is the environment of the calling thread (process bound or callback environment) or NULL if calling from a custom thread not spawned by ERTS.
Returns 0 on success, < 0 if no down callback is provided, and > 0 if the process is no longer alive or if target_pid is undefined.
This function is thread-safe.

ErlNifTime enif_monotonic_time(ErlNifTimeUnit time_unit)
Returns the current Erlang monotonic time. Notice that it is not uncommon with negative values.
time_unit is the time unit of the returned value.
Returns ERL_NIF_TIME_ERROR if called with an invalid time unit argument, or if called from a thread that is not a scheduler thread.
See also ErlNifTime and ErlNifTimeUnit.

ErlNifMutex *enif_mutex_create(char *name)
Same as erl_drv_mutex_create.
**erl_nif**

```c
void enif_mutex_destroy(ErlNifMutex *mtx)
Same as erl_drv_mutex_destroy.

void enif_mutex_lock(ErlNifMutex *mtx)
Same as erl_drv_mutex_lock.

char* enif_mutex_name(ErlNifMutex* mtx)
Same as erl_drv_mutex_name.

int enif_mutex_trylock(ErlNifMutex *mtx)
Same as erl_drv_mutex_trylock.

void enif_mutex_unlock(ErlNifMutex *mtx)
Same as erl_drv_mutex_unlock.

ERL_NIF_TERM enif_now_time(ErlNifEnv *env)
Returns an `erlang:now()` time stamp.
This function is deprecated.

ErlNifResourceType *enif_open_resource_type(ErlNifEnv* env, const char* module_str, const char* name, ErlNifResourceDtor* dtor, ErlNifResourceFlags flags, ErlNifResourceFlags* tried)
Creates or takes over a resource type identified by the string name and gives it the destructor function pointed to by dtor. Argument flags can have the following values:

ERL_NIF_RT_CREATE
Creates a new resource type that does not already exist.

ERL_NIF_RT_TAKEOVER
Opens an existing resource type and takes over ownership of all its instances. The supplied destructor dtor is called both for existing instances and new instances not yet created by the calling NIF library.

The two flag values can be combined with bitwise OR. The resource type name is local to the calling module. Argument module_str is not (yet) used and must be NULL. dtor can be NULL if no destructor is needed.

On success, the function returns a pointer to the resource type and *tried is set to either ERL_NIF_RT_CREATE or ERL_NIF_RT_TAKEOVER to indicate what was done. On failure, returns NULL and sets *tried to flags. It is allowed to set tried to NULL.

Notice that enif_open_resource_type is only allowed to be called in the two callbacks load and upgrade. The resource type is only created or taken over if the calling load/upgrade function returns successfully.

See also enif_open_resource_type_x.

ErlNifResourceType *enif_open_resource_type_x(ErlNifEnv* env, const char* name, const ErlNifResourceTypeInit* init, ErlNifResourceFlags flags, ErlNifResourceFlags* tried)
Same as enif_open_resource_type except it accepts additional callback functions for resource types that are used together with enif_select and enif_monitor_process.
```
Argument `init` is a pointer to an `ErlNifResourceTypeInit` structure that contains the function pointers for destructor, down and stop callbacks for the resource type.

**Note:**

Only members `dtor`, `down` and `stop` in `ErlNifResourceTypeInit` are read by `enif_open_resource_type_x`. To implement the new dyncall callback use `enif_init_resource_type`.

```c
ErlNifResourceType *enif_init_resource_type(ErlNifEnv* env, const char* name, const ErlNifResourceTypeInit* init, ErlNifResourceFlags flags, ErlNifResourceFlags* tried)
```

Same as `enif_open_resource_type_x` except it accepts an additional callback function for resource types that are used together with `enif_dynamic_resource_call`. Argument `init` is a pointer to an `ErlNifResourceTypeInit` structure that contains the callback function pointers `dtor`, `down`, `stop` and the new dyncall. The struct also contains the field members that must be set to the number of initialized callbacks counted from the top of the struct. For example, to initialize all callbacks including `dyncall`, members should be set to 4. All callbacks are optional and may be set to `NULL`.

```c
int enif_port_command(ErlNifEnv* env, const ErlNifPort* to_port, ErlNifEnv *msg_env, ERL_NIF_TERM msg)
```

Works as `erlang:port_command/2`, except that it is always completely asynchronous.

- `env`
  The environment of the calling process. Must not be `NULL`.
- `*to_port`
  The port ID of the receiving port. The port ID is to refer to a port on the local node.
- `msg_env`
  The environment of the message term. Can be a process independent environment allocated with `enif_alloc_env` or `NULL`.
- `msg`
  The message term to send. The same limitations apply as on the payload to `erlang:port_command/2`.

Using a `msg_env` of `NULL` is an optimization, which groups together calls to `enif_alloc_env`, `enif_make_copy`, `enif_port_command` and `enif_free_env` into one call. This optimization is only useful when a majority of the terms are to be copied from `env` to `msg_env`.

Returns `true` if the command is successfully sent. Returns `false` if the command fails, for example:

- `*to_port` does not refer to a local port.
- The currently executing process (that is, the sender) is not alive.
- `msg` is invalid.

See also `enif_get_local_port`.

```c
void *enif_priv_data(ErlNifEnv* env)
```

Returns the pointer to the private data that was set by `load` or `upgrade`. 
ERL_NIF_TERM enif_raise_exception(ErlNifEnv* env, ERL_NIF_TERM reason)
Creates an error exception with the term reason to be returned from a NIF, and associates it with environment env.
Once a NIF or any function it calls invokes enif_raise_exception, the runtime ensures that the exception it creates is raised when the NIF returns, even if the NIF attempts to return a non-exception term instead.
The return value from enif_raise_exception can only be used as the return value from the NIF that invoked it (directly or indirectly) or be passed to enif_is_exception, but not to any other NIF API function.
See also enif_has_pending_exception and enif_make_badarg.

void *enif_realloc(void* ptr, size_t size)
Reallocates memory allocated by enif_alloc to size bytes.
Returns NULL if the reallocation fails.
The returned pointer is suitably aligned for any built-in type that fit in the allocated memory.

int enif_realloc_binary(ErlNifBinary* bin, size_t size)
Changes the size of a binary bin. The source binary can be read-only, in which case it is left untouched and a mutable copy is allocated and assigned to *bin.
Returns true on success, or false if memory allocation failed.

void enif_release_binary(ErlNifBinary* bin)
Releases a binary obtained from enif_alloc_binary.

void enif_release_resource(void* obj)
Removes a reference to resource object obj obtained from enif_alloc_resource. The resource object is destructed when the last reference is removed. Each call to enif_release_resource must correspond to a previous call to enif_alloc_resource or enif_keep_resource. References made by enif_make_resource can only be removed by the garbage collector.
There are no guarantees exactly when the destructor of an unreferenced resource is called. It could be called directly by enif_release_resource but it could also be scheduled to be called at a later time possibly by another thread.

ErlNifRWLock *enif_rwlock_create(char *name)
Same as erl_drv_rwlock_create.

void enif_rwlock_destroy(ErlNifRWLock *rwlck)
Same as erl_drv_rwlock_destroy.

char*enif_rwlock_name(ErlNifRWLock* rwlck)
Same as erl_drv_rwlock_name.

void enif_rwlock_rlock(ErlNifRWLock *rwlck)
Same as erl_drv_rwlock_rlock.

void enif_rwlock_runlock(ErlNifRWLock *rwlck)
Same as erl_drv_rwlock_runlock.

408 | Ericsson AB. All Rights Reserved.: Erlang Run-Time System Application (ERTS)
void enif_rwlock_rwlock(ErlNifRWLock *rwlck)
Same as erl_drv_rwlock_rwlock.

void enif_rwlock_rwlock(ErlNifRWLock *rwlck)
Same as erl_drv_rwlock_rwlock.

int enif_rwlock_tryrlock(ErlNifRWLock *rwlck)
Same as erl_drv_rwlock_tryrlock.

int enif_rwlock_tryrwlock(ErlNifRWLock *rwlck)
Same as erl_drv_rwlock_tryrwlock.

ERL_NIF_TERM enif_schedule_nif( ErlNifEnv* caller_env, const char* fun_name,
int flags, ERL_NIF_TERM (*fp)(ErlNifEnv* env, int argc, const ERL_NIF_TERM
argv[]), int argc, const ERL_NIF_TERM argv[])
Schedules NIF fp to execute. This function allows an application to break up long-running work into multiple regular
NIF calls or to schedule a dirty NIF to execute on a dirty scheduler thread.

caller_env
Must be process bound environment of the calling NIF.

fun_name
Provides a name for the NIF that is scheduled for execution. If it cannot be converted to an atom,
enif_schedule_nif returns a badarg exception.

flags
Must be set to 0 for a regular NIF. If the emulator was built with dirty scheduler support enabled, flags
can be set to either ERL_NIF_DIRTY_JOB_CPU_BOUND if the job is expected to be CPU-bound, or
ERL_NIF_DIRTY_JOB_IO_BOUND for jobs that will be I/O-bound. If dirty scheduler threads are not available
in the emulator, an attempt to schedule such a job results in a notsup exception.

argc and argv
Can either be the originals passed into the calling NIF, or can be values created by the calling NIF.
The calling NIF must use the return value of enif_schedule_nif as its own return value.
Be aware that enif_schedule_nif, as its name implies, only schedules the NIF for future execution. The calling
NIF does not block waiting for the scheduled NIF to execute and return. This means that the calling NIF cannot expect
to receive the scheduled NIF return value and use it for further operations.

int enif_select(ErlNifEnv* env, ErlNifEvent event, enum ErlNifSelectFlags
mode, void* obj, const ErlNifPid* pid, ERL_NIF_TERM ref)
This function can be used to receive asynchronous notifications when OS-specific event objects become ready for
either read or write operations.
Argument event identifies the event object. On Unix systems, the functions select/poll are used. The event
object must be a socket, pipe or other file descriptor object that select/poll can use.
Argument mode describes the type of events to wait for. It can be ERL_NIF_SELECT_READ,
ERL_NIF_SELECT_WRITE or a bitwise OR combination to wait for both. It can also be ERL_NIF_SELECT_STOP
or ERL_NIF_SELECT_CANCEL which are described further below. When a read or write event is triggered, a notification message like this is sent to the process identified by pid:

\[
\text{\{select, Obj, Ref, ready_input | ready_output\}}
\]

ready_input or ready_output indicates if the event object is ready for reading or writing.

### Note:

For complete control over the message format use the newer functions enif_select_read or enif_select_write introduced in erts-11.0 (OTP-22.0).

Argument \text{pid} may be NULL to indicate the calling process. It must not be set as undefined.

Argument \text{obj} is a resource object obtained from \text{enif_alloc_resource}. The purpose of the resource objects is as a container of the event object to manage its state and lifetime. A handle to the resource is received in the notification message as \text{Obj}.

Argument \text{ref} must be either a reference obtained from \text{erlang:make_ref/0} or the atom undefined. It will be passed as \text{Ref} in the notifications. If a selective receive statement is used to wait for the notification then a reference created just before the receive will exploit a runtime optimization that bypasses all earlier received messages in the queue.

The notifications are one-shot only. To receive further notifications of the same type (read or write), repeated calls to \text{enif_select} must be made after receiving each notification.

ERL_NIF_SELECT_CANCEL can be used to cancel previously selected events. It must be used in a bitwise OR combination with ERL_NIF_SELECT_READ and/or ERL_NIF_SELECT_WRITE to indicate which type of event to cancel. Arguments \text{pid} and \text{ref} are ignored when ERL_NIF_SELECT_CANCEL is specified. The return value will tell if the event was actually cancelled or if a notification may already have been sent.

Use ERL_NIF_SELECT_STOP as mode in order to safely close an event object that has been passed to \text{enif_select}. The stop callback of the resource \text{obj} will be called when it is safe to close the event object. This safe way of closing event objects must be used even if all notifications have been received (or cancelled) and no further calls to \text{enif_select} have been made. ERL_NIF_SELECT_STOP will first cancel any selected events before it calls or schedules the stop callback. Arguments \text{pid} and \text{ref} are ignored when ERL_NIF_SELECT_STOP is specified.

The first call to \text{enif_select} for a specific OS event will establish a relation between the event object and the containing resource. All subsequent calls for an event must pass its containing resource as argument \text{obj}. The relation is dissolved when \text{enif_select} has been called with mode as ERL_NIF_SELECT_STOP and the corresponding stop callback has returned. A resource can contain several event objects but one event object can only be contained within one resource. A resource will not be destructed until all its contained relations have been dissolved.

### Note:

Use \text{enif_monitor_process} together with \text{enif_select} to detect failing Erlang processes and prevent them from causing permanent leakage of resources and their contained OS event objects.

Returns a non-negative value on success where the following bits can be set:

- ERL_NIF_SELECT_STOP_CALLED
  - The stop callback was called directly by \text{enif_select}.
- ERL_NIF_SELECT_STOP_SCHEDULED
  - The stop callback was scheduled to run on some other thread or later by this thread.
ERL_NIF_SELECT_READ_CANCELLED
A read event was cancelled by ERL_NIF_SELECT_CANCEL or ERL_NIF_SELECT_STOP and is guaranteed not to generate a ready_input notification message.

ERL_NIF_SELECT_WRITE_CANCELLED
A write event was cancelled by ERL_NIF_SELECT_CANCEL or ERL_NIF_SELECT_STOP and is guaranteed not to generate a ready_output notification message.

Returns a negative value if the call failed where the following bits can be set:

ERL_NIF_SELECT_INVALID_EVENT
Argument event is not a valid OS event object.

ERL_NIF_SELECT_FAILED
The system call failed to add the event object to the poll set.

Note:
Use bitwise AND to test for specific bits in the return value. New significant bits may be added in future releases to give more detailed information for both failed and successful calls. Do NOT use equality tests like ==, as that may cause your application to stop working.

Example:

```c
retval = enif_select(env, fd, ERL_NIF_SELECT_STOP, resource, ref);
if (retval < 0) {
    /* handle error */
}
/* Success! */
if (retval & ERL_NIF_SELECT_STOP_CALLED) {
    /* ... */
}
```

Note:
The mode flag ERL_NIF_SELECT_CANCEL and the return flags ERL_NIF_SELECT_READ_CANCELLED and ERL_NIF_SELECT_WRITE_CANCELLED were introduced in erts-11.0 (OTP-22.0).

int enif_select_read(ErlNifEnv* env, ErlNifEvent event, void* obj, const ErlNifPid* pid, ERL_NIF_TERM msg, ErlNifEnv* msg_env)
int enif_select_write(ErlNifEnv* env, ErlNifEvent event, void* obj, const ErlNifPid* pid, ERL_NIF_TERM msg, ErlNifEnv* msg_env)

These are variants of enif_select where you can supply your own message term msg that will be sent to the process instead of the predefined tuple {select,_,_,_,_}.

Argument msg_env must either be NULL or the environment of msg allocated with enif_alloc_env. If argument msg_env is NULL the term msg will be copied, otherwise both msg and msg_env will be invalidated by a successful call to enif_select_read or enif_select_write. The environment is then to either be freed with enif_free_env or cleared for reuse with enif_clear_env. An unsuccessful call will leave msg and msg_env still valid.

Apart from the message format enif_select_read and enif_select_write behaves exactly the same as enif_select with argument mode as either ERL_NIF_SELECT_READ or ERL_NIF_SELECT_WRITE. To cancel or close events use enif_select.
ErlNifPid *enif_self(ErlNifEnv* caller_env, ErlNifPid* pid)
Initializes the ErlNifPid variable at *pid to represent the calling process.
Returns pid if successful, or NULL if caller_env is not a process bound environment.

int enif_send(ErlNifEnv* caller_env, ErlNifPid* to_pid, ErlNifEnv* msg_env, ERL_NIF_TERM msg)
Sends a message to a process.
caller_env
    The environment of the calling thread (process bound or callback environment) or NULL if calling from a
custom thread not spawned by ERTS.
to_pid
    The pid of the receiving process. The pid is to refer to a process on the local node.
msg_env
    The environment of the message term. Must be a process independent environment allocated with
    enif_alloc_env or NULL.
msg
    The message term to send.
Returns true if the message is successfully sent. Returns false if the send operation fails, that is:
• *to_pid does not refer to an alive local process.
• The currently executing process (that is, the sender) is not alive.
The message environment msg_env with all its terms (including msg) is invalidated by a successful call
to enif_send. The environment is to either be freed with enif_free_env or cleared for reuse with
enif_clear_env. An unsuccessful call will leave msg and msg_env still valid.
If msg_env is set to NULL, the msg term is copied and the original term and its environment is still valid after the call.
This function is thread-safe.

Note:
Passing msg_env as NULL is only supported as from ERTS 8.0 (Erlang/OTP 19).

int enif_set_option(ErlNifEnv *env, ErlNifOption opt, ...)
Set an option. On success, zero will be returned. On failure, a non zero value will be returned. Currently the following
options can be set:

int enif_set_option(ErlNifEnv *env, ERL_NIF_OPT_DELAY_HALT)
Enable delay of runtime system halt with flushing enabled until all calls to NIFs in the NIF library have returned.
If the delay halt feature has not been enabled, a halt with flushing enabled may complete even though processes
are still executing inside NIFs in the NIF library. Note that by returning we here mean the first point where the
NIF returns control back to the runtime system, and not the point where a call to a NIF return a value back to the
Erlang code that called the NIF. That is, if you schedule execution of a NIF, using enif_schedule_nif(),
from within a NIF while the system is halting, the scheduled NIF call will not be executed even though delay
halt has been enabled for the NIF library.

The runtime system halts when one of the erlang:halt() BIFs are called. By default flushing is enabled,
but can be disabled using the erlang:halt/2 BIF. When flushing has been disabled, the delay halt setting
will have no effect. That is, the runtime system will halt without waiting for NIFs to return even if the delay halt
setting has been enabled. See the {flush, boolean()} option of erlang:halt/2 for more information.
The `ERL_NIF_OPT_DELAY_HALT` option can only be set during loading of a NIF library in a call to `enif_set_option()` inside a NIF library `load()` or `upgrade()` call, and will fail if set somewhere else. The `env` argument `must` be the callback environment passed to the `load()` or the `upgrade()` call. This option can also only be set once. That is, the `delay halt` setting cannot be changed once it has been enabled. The `delay halt` setting is tied to the module instance with which the NIF library instance has been loaded. That is, in case both a new and old version of a module using the NIF library are loaded, they can have the same or different `delay halt` settings.

The `delay halt` feature can be used in combination with an `on halt` callback. The `on halt` callback is in this case typically used to notify processes blocked in NIFs in the library that it is time to return in order to let the runtime system complete the halting. Such NIFs should be dirty NIFs, since ordinary NIFs should never block for a long time.

```c
int enif_set_option(ErlNifEnv *env, ERL_NIF_OPT_ON_HALT, ErlNifOnHaltCallback *on_halt)
```

Install a callback that will be called when the runtime system halts with flushing enabled.

The runtime system halts when one of the `erlang:halt()` BIFs are called. By default flushing is enabled, but can be disabled using the `erlang:halt/2` BIF. When flushing has been disabled, the runtime system will halt without calling any `on halt` callbacks even if such are installed. See the `{flush, boolean()}` option of `erlang:halt/2` for more information.

The `ERL_NIF_OPT_ON_HALT` option can only be set during loading of a NIF library in a call to `enif_set_option()` inside a NIF library `load()` or `upgrade()` call, and will fail if called somewhere else. The `env` argument `must` be the callback environment passed to the `load()` or the `upgrade()` call. The `on_halt` argument should be a function pointer to the callback to install. The `on halt` callback will be tied to the module instance with which the NIF library instance has been loaded. That is, in case both a new and old version of a module using the NIF library are loaded, they can both have different, none, or the same `on halt` callbacks installed. When unloading the NIF library during a code purge, an installed `on halt` callback will be uninstalled. The `ERL_NIF_OPT_ON_HALT` option can also only be set once. That is, the `on halt` callback cannot be changed or removed once it has been installed by any other means than purging the module instance that loaded the NIF library.

When the installed `on halt` callback is called, it will be passed a pointer to `priv_data` as argument. The `priv_data` pointer can be set when loading the NIF library.

The `on halt` callback can be used in combination with `delay of halt` until all calls into the library have returned. The `on halt` callback is in this case typically used to notify processes blocked in NIFs in the library that it is time to return in order to let the runtime system complete the halting. Such NIFs should be dirty NIFs, since ordinary NIFs should never block for a long time.

```c
void enif_set_pid_undefined(ErlNifPid* pid)
```

Sets an `ErlNifPid` variable as undefined. See `enif_is_pid_undefined`.

```c
unsigned enif_sizeof_resource(void* obj)
```

Gets the byte size of resource object `obj` obtained by `enif_alloc_resource`.

```c
int enif_snprintf(char *str, size_t size, const char *format, ...)
```

Similar to `snprintf` but this format string also accepts `%T`, which formats Erlang terms of type `ERL_NIF_TERM`. This function is primarily intended for debugging purpose. It is not recommended to print very large terms with `%T`. The function may change `errno`, even if successful.
void enif_system_info(ErlNifSysInfo *sys_info_ptr, size_t size)
Same as driver_system_info.

int enif_term_to_binary(ErlNifEnv *env, ERL_NIF_TERM term, ErlNifBinary *bin)
Allocates a new binary with enif_alloc_binary and stores the result of encoding term according to the Erlang external term format.
Returns true on success, or false if the allocation fails.
See also erlang:term_to_binary/1 and enif_binary_to_term.

ErlNifTermType enif_term_type(ErlNifEnv *env, ERL_NIF_TERM term)
Determines the type of the given term. The term must be an ordinary Erlang term and not one of the special terms returned by enif_raise_exception, enif_schedule_nif, or similar.
The following types are defined at the moment:
ERL_NIF_TERM_TYPE_ATOM
ERL_NIF_TERM_TYPE_BITSTRING
   A bitstring or binary
ERL_NIF_TERM_TYPE_FLOAT
ERL_NIF_TERM_TYPE_FUN
ERL_NIF_TERM_TYPE_INTEGER
ERL_NIF_TERM_TYPE_LIST
   A list, empty or not
ERL_NIF_TERM_TYPE_MAP
ERL_NIF_TERM_TYPE_PID
ERL_NIF_TERM_TYPE_PORT
ERL_NIF_TERM_TYPE_REFERENCE
ERL_NIF_TERM_TYPE_TUPLE
Note that new types may be added in the future, so the caller must be prepared to handle unknown types.

int enif_thread_create(char *name,ErlNifTid *tid,void * (*func)(void *),void *args,ErlNifThreadOpts *opts)
Same as erl_drv_thread_create.

void enif_thread_exit(void *resp)
Same as erl_drv_thread_exit.

int enif_thread_join(ErlNifTid, void **respp)
Same as erl_drv_thread_join.

char*enif_thread_name(ErlNifTid tid)
Same as erl_drv_thread_name.

ErlNifThreadOpts *enif_thread_opts_create(char *name)
Same as erl_drv_thread_opts_create.
void enif_thread_opts_destroy(ErlNifThreadOpts *opts)
Same as erl_drv_thread_opts_destroy.

ErlNifTid enif_thread_self(void)
Same as erl_drv_thread_self.

int enif_thread_type(void)
Determine the type of currently executing thread. A positive value indicates a scheduler thread while a negative value
or zero indicates another type of thread. Currently the following specific types exist (which may be extended in the
future):

ERL_NIF_THR_UNDEFINED
   Undefined thread that is not a scheduler thread.
ERL_NIF_THR_NORMAL_SCHEDULER
   A normal scheduler thread.
ERL_NIF_THR_DIRTY_CPU_SCHEDULER
   A dirty CPU scheduler thread.
ERL_NIF_THR_DIRTY_IO_SCHEDULER
   A dirty I/O scheduler thread.

ErlNifTime enif_time_offset(ErlNifTimeUnit time_unit)
Returns the current time offset between Erlang monotonic time and Erlang system time converted into the time_unit
passed as argument.
time_unit is the time unit of the returned value.
Returns ERL_NIF_TIME_ERROR if called with an invalid time unit argument or if called from a thread that is not
a scheduler thread.
See also ErlNifTime and ErlNifTimeUnit.

void *enif_tsd_get(ErlNifTSDKey key)
Same as erl_drv_tsd_get.

int enif_tsd_key_create(char *name, ErlNifTSDKey *key)
Same as erl_drv_tsd_key_create.

void enif_tsd_key_destroy(ErlNifTSDKey key)
Same as erl_drv_tsd_key_destroy.

void enif_tsd_set(ErlNifTSDKey key, void *data)
Same as erl_drv_tsd_set.

int enif_vfprintf(FILE *stream, const char *format, va_list ap)
Equivalent to enif_fprintf except that its called with a va_list instead of a variable number of arguments.
int enif_vsnprintf(char *str, size_t size, const char *format, va_list ap)
Equivalent to enif_snprintf except that its called with a va_list instead of a variable number of arguments.

int enif_whereis_pid(ErlNifEnv *caller_env, ERL_NIF_TERM name, ErlNifPid *pid)
Looks up a process by its registered name.

caller_env
  The environment of the calling thread (process bound or callback environment) or NULL if calling from a custom thread not spawned by ERTS.
name
  The name of a registered process, as an atom.
*pid
  The ErlNifPid in which the resolved process id is stored.
On success, sets *pid to the local process registered with name and returns true. If name is not a registered process, or is not an atom, false is returned and *pid is unchanged.
Works as erlang:whereis/1, but restricted to processes. See enif_whereis_port to resolve registered ports.

int enif_whereis_port(ErlNifEnv *caller_env, ERL_NIF_TERM name, ErlNifPort *port)
Looks up a port by its registered name.

caller_env
  The environment of the calling thread (process bound or callback environment) or NULL if calling from a custom thread not spawned by ERTS.
name
  The name of a registered port, as an atom.
*port
  The ErlNifPort in which the resolved port id is stored.
On success, sets *port to the port registered with name and returns true. If name is not a registered port, or is not an atom, false is returned and *port is unchanged.
Works as erlang:whereis/1, but restricted to ports. See enif_whereis_pid to resolve registered processes.

See Also
erlang:load_nif/2
NIFs (tutorial)
Debugging NIFs and Port Drivers
erl_prim_loader

Erlang module

This module is used to load all Erlang modules into the system. The start script is also fetched with this low-level loader. erl_prim_loader knows about the environment and how to fetch modules.

Command-line flag -loader Loader can be used to choose the method used by erl_prim_loader. Two Loader methods are supported by the Erlang runtime system: efile and inet.

Exports

get_file(Filename) -> {ok, Bin, FullName} | error
Types:
   Filename = atom() | string()
   Bin = binary()
   FullName = string()

Fetches a file using the low-level loader. Filename is either an absolute filename or only the name of the file, for example, "lists.beam". If an internal path is set to the loader, this path is used to find the file. FullName is the complete name of the fetched file. Bin is the contents of the file as a binary.

Filename can also be a file in an archive, for example, $OTPROOT/lib/mnesia-4.4.7.ez/mnesia-4.4.7/ebin/mnesia.beam. For information about archive files, see code(3).

get_path() -> {ok, Path}
Types:
   Path = [Dir :: string()]

Gets the path set in the loader. The path is set by the init(3) process according to information found in the start script.

list_dir(Dir) -> {ok, Filenames} | error
Types:
   Dir = string()
   Filenames = [Filename :: string()]

Lists all the files in a directory. Returns {ok, Filenames} if successful, otherwise error. Filenames is a list of the names of all the files in the directory. The names are not sorted.

Dir can also be a directory in an archive, for example, $OTPROOT/lib/mnesia-4.4.7.ez/mnesia-4.4.7/ebin. For information about archive files, see code(3).

read_file_info(Filename) -> {ok, FileInfo} | error
Types:
   Filename = string()
   FileInfo = file:file_info()

Retrieves information about a file. Returns {ok, FileInfo} if successful, otherwise error. FileInfo is a record file_info, defined in the Kernel include file file.hrl. Include the following directive in the module from which the function is called:
-include_lib("kernel/include/file.hrl").

For more information about the record file_info, see file(3).
Filename can also be a file in an archive, for example, $OTPROOT/lib/mnesia-4.4.7.ez/mnesia-4.4.7/ebin/mnesia. For information about archive files, see code(3).

read_link_info(Filename) -> {ok, FileInfo} | error
Types:
  Filename = string()
  FileInfo = file:file_info()
Works like read_file_info/1 except that if Filename is a symbolic link, information about the link is returned in the file_info record and the type field of the record is set to symlink.
If Filename is not a symbolic link, this function returns exactly the same result as read_file_info/1. On platforms that do not support symbolic links, this function is always equivalent to read_file_info/1.

set_path(Path) -> ok
Types:
  Path = [Dir :: string()]
Sets the path of the loader if init(3) interprets a path command in the start script.

Command-Line Flags
The erl_prim_loader module interprets the following command-line flags:

-loader Loader
  Specifies the name of the loader used by erl_prim_loader. Loader can be efile (use the local file system) or inet (load using the boot_server on another Erlang node).
  If flag -loader is omitted, it defaults to efile.

-loader_debug
  Makes the efile loader write some debug information, such as the reason for failures, while it handles files.

-hosts Hosts
  Specifies which other Erlang nodes the inet loader can use. This flag is mandatory if flag -loader inet is present. On each host, there must be on Erlang node with the erl_boot_server(3), which handles the load requests. Hosts is a list of IP addresses (hostnames are not acceptable).

-setcookie Cookie
  Specifies the cookie of the Erlang runtime system. This flag is mandatory if flag -loader inet is present.

See Also
init(3), erl_boot_server(3)
erlsrv

Command

This utility is specific to Windows NT/2000/XP (and later versions of Windows). It allows Erlang emulators to run as services on the Windows system, allowing embedded systems to start without any user needing to log on. The emulator started in this way can be manipulated through the Windows services applet in a manner similar to other services.

Notice that erlsrv is not a general service utility for Windows, but designed for embedded Erlang systems.

erlsrv also provides a command-line interface for registering, changing, starting, and stopping services.

To manipulate services, the logged on user is to have administrator privileges on the machine. The Erlang machine itself is (default) run as the local administrator. This can be changed with the Services applet in Windows.

The processes created by the service can, as opposed to normal services, be "killed" with the task manager. Killing an emulator that is started by a service triggers the "OnFail" action specified for that service, which can be a reboot.

The following parameters can be specified for each Erlang service:

**StopAction**

Tells erlsrv how to stop the Erlang emulator. Default is to kill it (Win32 TerminateProcess), but this action can specify any Erlang shell command that will be executed in the emulator to make it stop. The emulator is expected to stop within 30 seconds after the command is issued in the shell. If the emulator is not stopped, it reports a running state to the service manager.

**OnFail**

Can be one of the following:

- **reboot**
  
  The Windows system is rebooted whenever the emulator stops (a more simple form of watchdog). This can be useful for less critical systems, otherwise use the heart functionality to accomplish this.

- **restart**
  
  Makes the Erlang emulator be restarted (with whatever parameters are registered for the service at the occasion) when it stops. If the emulator stops again within 10 seconds, it is not restarted to avoid an infinite loop, which could hang the Windows system.

- **restart_always**
  
  Similar to restart, but does not try to detect cyclic restarts; it is expected that some other mechanism is present to avoid the problem.

- **ignore** (the default)
  
  Reports the service as stopped to the service manager whenever it fails; it must be manually restarted.

On a system where release handling is used, this is always to be set to ignore. Use heart to restart the service on failure instead.

**Machine**

The location of the Erlang emulator. The default is the erl.exe located in the same directory as erlsrv.exe. If the system uses release handling, this is to be set to a program similar to start_erl.exe.

**Env**

Specifies an extra environment for the emulator. The environment variables specified here are added to the system-wide environment block that is normally present when a service starts up. Variables present in both the
system-wide environment and in the service environment specification will be set to the value specified in the service.

**WorkDir**

The working directory for the Erlang emulator. Must be on a local drive (no network drives are mounted when a service starts). Default working directory for services is `%SystemDrive%\%SystemPath`. Debug log files will be placed in this directory.

**Priority**

The process priority of the emulator. Can be one of the following:

- **realtime**
  Not recommended, as the machine will possibly be inaccessible to interactive users.

- **high**
  Can be used if two Erlang nodes are to reside on one dedicated system and one is to have precedence over the other.

- **low**
  Can be used if interactive performance is not to be affected by the emulator process.

  default (the default)

**SName or Name**

Specifies the short or long node name of the Erlang emulator. The Erlang services are always distributed. Default is to use the service name as (short) nodename.

**DebugType**

Specifies that output from the Erlang shell is to be sent to a "debug log". The log file is named `<servicename>.debug` or `<servicename>.debug.<N>`, where `<N>` is an integer from 1 through 99. The log file is placed in the working directory of the service (as specified in WorkDir).

Can be one of the following:

- **new**
  Uses a separate log file for every invocation of the service (`<servicename>.debug.<N>`).

- **reuse**
  Reuses the same log file (`<servicename>.debug`).

- **console**
  Opens an interactive Windows console window for the Erlang shell of the service. Automatically disables the StopAction. A service started with an interactive console window does not survive logouts. OnFail actions do not work with debug consoles either.

  **none** (the default)

  The output of the Erlang shell is discarded.
Note:

The `console` option is **not** intended for production. It is **only** a convenient way to debug Erlang services during development.

The `new` and `reuse` options might seem convenient in a production system, but consider that the logs grow indefinitely during the system lifetime and cannot be truncated, except if the service is restarted.

In short, the `DebugType` is intended for debugging only. Logs during production are better produced with the standard Erlang logging facilities.

**Args**

PASSES extra arguments to the emulator startup program `erl.exe` (or `start_erl.exe`). Arguments that cannot be specified here are `-noinput` (`StopActions` would not work), `-name`, and `-sname` (they are specified in any way). The most common use is for specifying cookies and flags to be passed to `init:boot()` (-s).

**InternalServiceName**

Specifies the Windows-internal service name (not the display name, which is the one `erlsrv` uses to identify the service).

This internal name cannot be changed, it is fixed even if the service is renamed. `erlsrv` generates a unique internal name when a service is created. It is recommended to keep to the default if release handling is to be used for the application.

The internal service name can be seen in the Windows service manager if viewing `Properties` for an Erlang service.

**Comment**

A textual comment describing the service. Not mandatory, but shows up as the service description in the Windows service manager.

The naming of the service in a system that uses release handling must follow the convention `NodeName_Release`, where `NodeName` is the first part of the Erlang node name (up to, but not including the "@") and `Release` is the current release of the application.

**Exports**

`erlsrv {set | add} <service-name> [service options]`

The `set` and `add` commands modifies or adds an Erlang service, respectively. The simplest form of an `add` command is without any options in which case all default values (described above) apply. The service name is mandatory.

Every option can be specified without parameters, the default value is then applied. Values to the options are supplied only when the default is not to be used. For example, `erlsrv set myservice -prio -arg` sets the default priority and removes all arguments.

**Service options:**

- `st{opaction} [{<erlang shell command>}]`
  
  Defines the `StopAction`, the command given to the Erlang shell when the service is stopped. Default is none.

- `on{fail} [{reboot | restart | restart_always}]`
  
  The action to take when the Erlang emulator stops unexpectedly. Default is to ignore.
erlsrv

-m[achine] [<erl-command>]

The complete path to the Erlang emulator. Never use the werl program for this. Defaults to the erl.exe in the same directory as erlsrv.exe. When release handling is used, this is to be set to a program similar to start_erl.exe.

-e[nv] [<variable>=<value>] ...

Edits the environment block for the service. Every environment variable specified is added to the system environment block. If a variable specified here has the same name as a system-wide environment variable, the specified value overrides the system-wide. Environment variables are added to this list by specifying <variable>=<value> and deleted from the list by specifying <variable> alone. The environment block is automatically sorted. Any number of -env options can be specified in one command. Default is to use the system environment block unmodified (except for two additions, see section Environment below).

-w[orkdir] [<directory>]

The initial working directory of the Erlang emulator. Defaults to the system directory.

-p[riority] [{low|high|realtime}]

The priority of the Erlang emulator. Default to the Windows default priority.

{-s[n]ame | -n[ame]} [<node-name>]

The node name of the Erlang machine. Distribution is mandatory. Defaults to -sname <service name>.

-d[ebugtype] [{new|reuse|console}]

Specifies where shell output is to be sent. Default is that shell output is discarded. To be used only for debugging.

-ar[gs] [<limited erl arguments>]

Extra arguments to the Erlang emulator. Avoid -noinput, -noshell, and -sname/-name. Default is no extra arguments. Remember that the services cookie file is not necessarily the same as the interactive users. The service runs as the local administrator. Specify all arguments together in one string, use double quotes (") to specify an argument string containing spaces, and use quoted quotes (\") to specify a quote within the argument string if necessary.

-i[nternalservicename] [<internal name>]

Only allowed for add. Specifies a Windows-internal service name for the service, which by default is set to something unique (prefixed with the original service name) by erlsrv when adding a new service. Specifying this is a purely cosmetic action and is not recommended if release handling is to be performed. The internal service name cannot be changed once the service is created. The internal name is not to be confused with the ordinary service name, which is the name used to identify a service to erlsrv.

-c[omment] [<short description>]

Specifies a textual comment describing the service. This comment shows up as the service description in the Windows service manager.

erlsrv {start | start Disabled | stop | disable | enable} <service-name>

These commands are only added for convenience, the normal way to manipulate the state of a service is through the control panels services applet.

The start and stop commands communicates with the service manager for starting and stopping a service. The commands wait until the service is started or stopped. When disabling a service, it is not stopped, the disabled state does not take effect until the service is stopped. Enabling a service sets it in automatic mode, which is started at boot. This command cannot set the service to manual.
The `start_disabled` command operates on a service regardless of if it is enabled/disabled or started/stopped. It does this by first enabling it (regardless of if it is enabled or not), then starting it (if not already started), and then disabling it. The result is a disabled but started service, regardless of its earlier state. This is useful for starting services temporarily during a release upgrade. The difference between using `start_disabled` and the sequence `enable`, `start`, and `disable` is that all other `erlsrv` commands are locked out during the sequence of operations in `start_disabled`, making the operation atomic from an `erlsrv` user's point of view.

`erlsrv remove <service-name>`
Removes the service completely with all its registered options. It is stopped before it is removed.

`erlsrv list [<service-name>]`
If no service name is specified, a brief listing of all Erlang services is presented. If a service name is supplied, all options for that service are presented.

`erlsrv help`
Displays a brief help text.

**Environment**
The environment of an Erlang machine started as a service contains two special variables:

- `ERLSRV_SERVICE_NAME`
The name of the service that started the machine.
- `ERLSRV_EXECUTABLE`
The full path to the `erlsrv.exe`, which can be used to manipulate the service. This comes in handy when defining a heart command for your service.

A command file for restarting a service looks as follows:

```bash
@echo off
%ERLSRV_EXECUTABLE% stop %ERLSRV_SERVICE_NAME%
%ERLSRV_EXECUTABLE% start %ERLSRV_SERVICE_NAME%
```

This command file is then set as heart command.

The environment variables can also be used to detect that we are running as a service and make port programs react correctly to the control events generated on logout (see the next section).

**Port Programs**
When a program runs in the service context, it must handle the control events that are sent to every program in the system when the interactive user logs off. This is done in different ways for programs running in the console subsystem and programs running as window applications. An application running in the console subsystem (normal for port programs) uses the win32 function `SetConsoleCtrlHandler` to register a control handler that returns true in answer to the `CTRL_LOGOFF_EVENT` and `CTRL_SHUTDOWN_EVENT` events. Other applications only forward `WM_ENDSESSION` and `WM_QUERYENDSESSION` to the default window procedure.

A brief example in C of how to set the console control handler:
#include <windows.h>

/*
** A Console control handler that ignores the log off events,
** and lets the default handler take care of other events.
*/
BOOL WINAPI service_aware_handler(DWORD ctrl)
{
    if(ctrl == CTRL_LOGOFF_EVENT)
        return TRUE;
    if(ctrl == CTRL_SHUTDOWN_EVENT)
        return TRUE;
    return FALSE;
}

void initialize_handler(void)
{
    char buffer[2];
    /* We assume we are running as a service if this
    * environment variable is defined.
    */
    if(GetEnvironmentVariable("ERLSRV_SERVICE_NAME", buffer,
                               (DWORD) 2))
    {
        /* Actually set the control handler
        */
        SetConsoleCtrlHandler(&service_aware_handler, TRUE);
    }
}

Notes

Although the options are described in a Unix-like format, the case of the options or commands is not relevant, and
both character "/" and "-" can be used for options.

Notice that the program resides in the emulator's bin directory, not in the bin directory directly under the Erlang
root. The reasons for this are the subtle problem of upgrading the emulator on a running system, where a new version
of the runtime system should not need to overwrite existing (and probably used) executables.

To manipulate the Erlang services easily, put the <erlang_root>\erts-<version>\bin directory in the
path instead of <erlang_root>\bin. The erlsrv program can be found from inside Erlang by using the
os:find_executable/1 Erlang function.

For release handling to work, use start_erl as the Erlang machine. As stated above, the service name is significant.

See Also

start_erl(1), release_handler(3)
erl_tracer

Erlang module

This behavior module implements the back end of the Erlang tracing system. The functions in this module are called whenever a trace probe is triggered. Both the `enabled` and `trace` functions are called in the context of the entity that triggered the trace probe. This means that the overhead by having the tracing enabled is greatly effected by how much time is spent in these functions. So, do as little work as possible in these functions.

**Note:**

All functions in this behavior must be implemented as NIFs. This limitation can be removed in a future releases. An example tracer module NIF implementation is provided at the end of this page.

**Warning:**

Do not send messages or issue port commands to the `Tracee` in any of the callbacks. This is not allowed and can cause all sorts of strange behavior, including, but not limited to, infinite recursions.

**Data Types**

```erlang
trace_tag_call() =
    call | return_to | return_from | exception_from
trace_tag_gc() =
    gc_minor_start | gc_minor_end | gc_major_start | gc_major_end
trace_tag_ports() =
    open | closed | link | unlink | getting_linked |
                        | getting_unlinked
trace_tag_procs() =
    spawn | spawned | exit | link | unlink | getting_linked |
                        | getting_unlinked | register | unregister
trace_tag_receive() = 'receive'
trace_tag_running_ports() =
    in | out | in_exiting | out_exiting | out_exited
trace_tag_running_procs() =
    in | out | in_exiting | out_exiting | out_exited
trace_tag_send() = send | send_to_non_existing_process
trace_tag() =
    trace_tag_send() |
    trace_tag_receive() |
    trace_tag_call() |
    trace_tag_procs() |
    trace_tag_ports() |
    trace_tag_running_procs() |
    trace_tag_running_ports() |
    trace_tag_gc()
```

The different trace tags that the tracer is called with. Each trace tag is described in detail in `Module:trace/5`. 

Ericsson AB. All Rights Reserved.: Erlang Run-Time System Application (ERTS) | 425
tracee() = port() | pid() | undefined
The process or port that the trace belongs to.

trace_opts() =
    #{extra => term(),
        match_spec_result => term(),
        scheduler_id => integer() >= 0,
        timestamp =>
            timestamp | cpu_timestamp | monotonic | strict_monotonic}

The options for the tracee:

timestamp
    If set the tracer has been requested to include a time stamp.
extra
    If set the tracepoint has included additional data about the trace event. What the additional data is depends on which TraceTag has been triggered. The extra trace data corresponds to the fifth element in the trace tuples described in erlang:trace/3.
match_spec_result
    If set the tracer has been requested to include the output of a match specification that was run.
scheduler_id
    If set the scheduler id is to be included by the tracer.

tracer_state() = term()
The state specified when calling erlang:trace(PidPortSpec, true, [{tracer, Module, TracerState}]). The tracer state is an immutable value that is passed to erl_tracer callbacks and is to contain all the data that is needed to generate the trace event.

The following functions are to be exported from an erl_tracer callback module:

Module:enabled/3
    Mandatory
Module:trace/5
    Mandatory
Module:enabled_call/3
    Optional
Module:trace_call/5
    Optional
Module:enabled_garbage_collection/3
    Optional
Module:trace_garbage_collection/5
    Optional
Module:enabled_ports/3
    Optional
Module:trace_ports/5
    Optional
Module:enabled_procs/3
    Optional
Module:trace_procs/5
    Optional
Module:enabled_receive/3
    Optional
Module:trace_receive/5
    Optional
Module:enabled_running_ports/3
  Optional
Module:trace_running_ports/5
  Optional
Module:enabled_running_procs/3
  Optional
Module:trace_running_procs/5
  Optional
Module:enabled_send/3
  Optional
Module:trace_send/5
  Optional

Exports

Module:enabled(TraceTag, TracerState, Tracee) -> Result
Types:
  TraceTag = trace_tag() | trace_status
  TracerState = term()
  Tracee = tracee()
  Result = trace | discard | remove

This callback is called whenever a tracepoint is triggered. It allows the tracer to decide whether a trace is to be generated or not. This check is made as early as possible to limit the amount of overhead associated with tracing. If trace is returned, the necessary trace data is created and the trace callback of the tracer is called. If discard is returned, this trace call is discarded and no call to trace is done.

trace_status is a special type of TraceTag, which is used to check if the tracer is still to be active. It is called in multiple scenarios, but most significantly it is used when tracing is started using this tracer. If remove is returned when the trace_status is checked, the tracer is removed from the tracee.

This function can be called multiple times per tracepoint, so it is important that it is both fast and without side effects.

Module:enabled_call(TraceTag, TracerState, Tracee) -> Result
Types:
  TraceTag = trace_tag_call()
  TracerState = term()
  Tracee = tracee()
  Result = trace | discard | remove

This callback is called whenever a tracepoint with trace flag call | return_to is triggered. If enabled_call/3 is undefined, Module:enabled/3 is called instead.

Module:enabled_garbage_collection(TraceTag, TracerState, Tracee) -> Result
Types:
  TraceTag = trace_tag_gc()
  TracerState = term()
  Tracee = tracee()
  Result = trace | discard | remove
This callback is called whenever a tracepoint with trace flag \texttt{garbage\_collection} is triggered. If \texttt{enabled\_garbage\_collection/3} is undefined, \texttt{Module:enabled/3} is called instead.

\begin{verbatim}
Module:enabled\_ports(TraceTag, TracerState, Tracee) -> Result

Types:
\begin{verbatim}
TraceTag = trace\_tag\_ports()
TracerState = term()
Tracee = tracee()
Result = trace | discard | remove
\end{verbatim}
\end{verbatim}

This callback is called whenever a tracepoint with trace flag \texttt{ports} is triggered. If \texttt{enabled\_ports/3} is undefined, \texttt{Module:enabled/3} is called instead.

\begin{verbatim}
Module:enabled\_procs(TraceTag, TracerState, Tracee) -> Result

Types:
\begin{verbatim}
TraceTag = trace\_tag\_procs()
TracerState = term()
Tracee = tracee()
Result = trace | discard | remove
\end{verbatim}
\end{verbatim}

This callback is called whenever a tracepoint with trace flag \texttt{procs} is triggered. If \texttt{enabled\_procs/3} is undefined, \texttt{Module:enabled/3} is called instead.

\begin{verbatim}
Module:enabled\_receive(TraceTag, TracerState, Tracee) -> Result

Types:
\begin{verbatim}
TraceTag = trace\_tag\_receive()
TracerState = term()
Tracee = tracee()
Result = trace | discard | remove
\end{verbatim}
\end{verbatim}

This callback is called whenever a tracepoint with trace flag \texttt{receive} is triggered. If \texttt{enabled\_receive/3} is undefined, \texttt{Module:enabled/3} is called instead.

\begin{verbatim}
Module:enabled\_running\_ports(TraceTag, TracerState, Tracee) -> Result

Types:
\begin{verbatim}
TraceTag = trace\_tag\_running\_ports()
TracerState = term()
Tracee = tracee()
Result = trace | discard | remove
\end{verbatim}
\end{verbatim}

This callback is called whenever a tracepoint with trace flag \texttt{running\_ports} is triggered. If \texttt{enabled\_running\_ports/3} is undefined, \texttt{Module:enabled/3} is called instead.

\begin{verbatim}
Module:enabled\_running\_procs(TraceTag, TracerState, Tracee) -> Result

Types:
\begin{verbatim}
TraceTag = trace\_tag\_running\_procs()
\end{verbatim}
\end{verbatim}
TracerState = term()
Tracee = tracee()
Result = trace | discard | remove

This callback is called whenever a tracepoint with trace flag running_procs | running is triggered.
If enabled_running_procs/3 is undefined, Module:enabled/3 is called instead.

Module:enabled_send(TraceTag, TracerState, Tracee) -> Result
Types:
   TraceTag = trace_tag_send()
   TracerState = term()
   Tracee = tracee()
   Result = trace | discard | remove

This callback is called whenever a tracepoint with trace flag send is triggered.
If enabled_send/3 is undefined, Module:enabled/3 is called instead.

Module:trace(TraceTag, TracerState, Tracee, TraceTerm, Opts) -> Result
Types:
   TraceTag = trace_tag()
   TracerState = term()
   Tracee = tracee()
   TraceTerm = term()
   Opts = trace_opts()
   Result = ok

This callback is called when a tracepoint is triggered and the Module:enabled/3 callback returned trace. In it any side effects needed by the tracer are to be done. The tracepoint payload is located in the TraceTerm. The content of the TraceTerm depends on which TraceTag is triggered. TraceTerm corresponds to the fourth element in the trace tuples described in erlang:trace/3.
If the trace tuple has five elements, the fifth element will be sent as the extra value in the Opts maps.

Module:trace(seq_trace, TracerState, Label, SeqTraceInfo, Opts) -> Result
Types:
   TracerState = term()
   Label = term()
   SeqTraceInfo = term()
   Opts = trace_opts()
   Result = ok

The TraceTag seq_trace is handled slightly differently. There is no Tracee for seq_trace, instead the Label associated with the seq_trace event is specified.
For more information on what Label and SeqTraceInfo can be, see seq_trace(3).

Module:trace_call(TraceTag, TracerState, Tracee, TraceTerm, Opts) -> Result
Types:
   TraceTag = trace_tag_call()
TracerState = term()
Tracee = tracee()
TraceTerm = term()
Opts = trace_opts()
Result = ok

This callback is called when a tracepoint is triggered and the Module:enabled_call/3 callback returned trace.
If trace_call/5 is undefined, Module:trace/5 is called instead.

Module:trace_garbage_collection(TraceTag, TracerState, Tracee, TraceTerm, Opts) -> Result
Types:
  TraceTag = trace_tag_gc()
  TracerState = term()
  Tracee = tracee()
  TraceTerm = term()
  Opts = trace_opts()
  Result = ok

This callback is called when a tracepoint is triggered and the Module:enabled_garbage_collection/3 callback returned trace.
If trace_garbage_collection/5 is undefined, Module:trace/5 is called instead.

Module:trace_ports(TraceTag, TracerState, Tracee, TraceTerm, Opts) -> Result
Types:
  TraceTag = trace_tag()
  TracerState = term()
  Tracee = tracee()
  TraceTerm = term()
  Opts = trace_opts()
  Result = ok

This callback is called when a tracepoint is triggered and the Module:enabled_ports/3 callback returned trace.
If trace_ports/5 is undefined, Module:trace/5 is called instead.

Module:trace_procs(TraceTag, TracerState, Tracee, TraceTerm, Opts) -> Result
Types:
  TraceTag = trace_tag()
  TracerState = term()
  Tracee = tracee()
  TraceTerm = term()
  Opts = trace_opts()
  Result = ok

This callback is called when a tracepoint is triggered and the Module:enabled_procs/3 callback returned trace.
If `trace_procs/5` is undefined, `Module:trace/5` is called instead.

**Module:trace_receive(TraceTag, TracerState, Tracee, TraceTerm, Opts) -> Result**

Types:
- TraceTag = `trace_tag_receive()`
- TracerState = `term()`
- Tracee = `tracee()`
- TraceTerm = `term()`
- Opts = `trace_opts()`
- Result = `ok`

This callback is called when a tracepoint is triggered and the `Module:enabled_receive/3` callback returned `trace`.

If `trace_receive/5` is undefined, `Module:trace/5` is called instead.

**Module:trace_running_ports(TraceTag, TracerState, Tracee, TraceTerm, Opts) -> Result**

Types:
- TraceTag = `trace_tag_running_ports()`
- TracerState = `term()`
- Tracee = `tracee()`
- TraceTerm = `term()`
- Opts = `trace_opts()`
- Result = `ok`

This callback is called when a tracepoint is triggered and the `Module:enabled_running_ports/3` callback returned `trace`.

If `trace_running_ports/5` is undefined, `Module:trace/5` is called instead.

**Module:trace_running_procs(TraceTag, TracerState, Tracee, TraceTerm, Opts) -> Result**

Types:
- TraceTag = `trace_tag_running_procs()`
- TracerState = `term()`
- Tracee = `tracee()`
- TraceTerm = `term()`
- Opts = `trace_opts()`
- Result = `ok`

This callback is called when a tracepoint is triggered and the `Module:enabled_running_procs/3` callback returned `trace`.

If `trace_running_procs/5` is undefined, `Module:trace/5` is called instead.

**Module:trace_send(TraceTag, TracerState, Tracee, TraceTerm, Opts) -> Result**

Types:
- TraceTag = `trace_tag_send()`
ErL Tracer Example

```erl
TracerState = term()
Tracee = tracee()
TraceTerm = term()
Opts = trace_opts()
Result = ok
```

This callback is called when a tracepoint is triggered and the `Module:enabled_send/3` callback returned `trace`. If `trace_send/5` is undefined, `Module:trace/5` is called instead.

Erl Tracer Module Example

In this example, a tracer module with a NIF back end sends a message for each `send` trace tag containing only the sender and receiver. Using this tracer module, a much more lightweight message tracer is used, which only records who sent messages to who.

The following is an example session using it on Linux:

```bash
$ gcc -I ./erts-8.0/include/ -fPIC -shared -o erl_msg_tracer.so erl_msg_tracer.c
$ erl
Erlang/OTP 19 [DEVELOPMENT] [erts-8.0] [source-ed2b56b] [64-bit] [smp:8:8] [async-threads:10] [hipe] [kernel-poll:false]
Eshell V8.0 (abort with ^G)
1> c(erl_msg_tracer), erl_msg_tracer:load().
ok
2> Tracer = spawn(fun F() -> receive M -> io:format("~p~n", [M]), F() end end).
   <0.37.0>
3> erlang:trace(new, true, [send, {tracer, erl_msg_tracer, Tracer}]).
0
{trace,<0.39.0>,<0.27.0>}
4> {ok, D} = file:open("/tmp/tmp.data", [write]).
   {trace,#Port<0.486>,<0.40.0>}
   {trace,<0.40.0>,<0.21.0>}
   {trace,#Port<0.487>,<0.4.0>}
   {trace,#Port<0.488>,<0.4.0>}
   {trace,#Port<0.489>,<0.4.0>}
   {trace,#Port<0.490>,<0.4.0>}
   {ok,<0.40.0>}
   {trace,<0.41.0>,<0.27.0>}
5>
```

```
erl_msg_tracer.erl:

-module(erl_msg_tracer).
-export([enabled/3, trace/5, load/0])

load() ->
    erlang:load_nif("erl_msg_tracer").

enabled(_, _, _) ->
    error.

trace(_, _, _, _, _) ->
    error.
```

```
erl_msg_tracer.c:

432 | Ericsson AB. All Rights Reserved.: Erlang Run-Time System Application (ERTS)```
#include <erl_nif.h>

/* NIF interface declarations */
static int load(ErlNifEnv* env, void** priv_data, ERL_NIF_TERM load_info);
static int upgrade(ErlNifEnv* env, void** priv_data, void** old_priv_data, ERL_NIF_TERM load_info);
static void unload(ErlNifEnv* env, void* priv_data);

/* The NIFs: */
static ERL_NIF_TERM enabled(ErlNifEnv* env, int argc, const ERL_NIF_TERM argv[]);
static ERL_NIF_TERM trace(ErlNifEnv* env, int argc, const ERL_NIF_TERM argv[]);

static ErlNifFunc nif_funcs[] = {
   {"enabled", 3, enabled},
   {"trace", 5, trace}
};

ERL_NIF_INIT(erl_msg_tracer, nif_funcs, load, NULL, upgrade, unload)

static int load(ErlNifEnv* env, void** priv_data, ERL_NIF_TERM load_info)
{
    *priv_data = NULL;
    return 0;
}

static void unload(ErlNifEnv* env, void* priv_data)
{
}

static int upgrade(ErlNifEnv* env, void** priv_data, void** old_priv_data,
                   ERL_NIF_TERM load_info)
{
    if (*old_priv_data != NULL || *priv_data != NULL) {
        return -1; /* Don't know how to do that */
    }
    if (load(env, priv_data, load_info)) {
        return -1;
    }
    return 0;
}

/* argv[0]: TraceTag
   argv[1]: TracerState
   argv[2]: Tracee */
static ERL_NIF_TERM enabled(ErlNifEnv* env, int argc, const ERL_NIF_TERM argv[])
{
    ErlNifPid to_pid;
    if (enif_get_local_pid(env, argv[1], &to_pid))
        if (enif_is_process_alive(env, &to_pid))
            if (enif_is_identical(enif_make_atom(env, "trace_status"), argv[0]))
                /* tracer is dead so we should remove this tracepoint */
                return enif_make_atom(env, "remove");
            else
                return enif_make_atom(env, "discard");
        /* Only generate trace for when tracer != tracee */
        if (enif_is_identical(argv[1], argv[2]))
            return enif_make_atom(env, "discard");
    /* Only trigger trace messages on 'send' */
    if (enif_is_identical(enif_make_atom(env, "send"), argv[0]))
        return enif_make_atom(env, "trace");
/* Have to answer trace_status */
if (enif_is_identical(enif_make_atom(env, "trace_status"), argv[0]))
    return enif_make_atom(env, "trace");

    return enif_make_atom(env, "discard");
}

/*
 * argv[0]: TraceTag, should only be 'send'
 * argv[1]: TracerState, process to send {Tracee, Recipient} to
 * argv[2]: Tracee
 * argv[3]: Message
 * argv[4]: Options, map containing Recipient
 */
static ERL_NIF_TERM trace(ErlNifEnv* env, int argc, const ERL_NIF_TERM argv[])
{
    ErlNifPid to_pid;
    ERL_NIF_TERM recipient, msg;

    if (enif_get_local_pid(env, argv[1], &to_pid)) {
        if (enif_get_map_value(env, argv[4], enif_make_atom(env, "extra"), &recipient)) {
            msg = enif_make_tuple3(env, enif_make_atom(env, "trace"), argv[2], recipient);
            enif_send(env, &to_pid, NULL, msg);
        }
    }

    return enif_make_atom(env, "ok");
}
erts_alloc is an Erlang runtime system internal memory allocator library. erts_alloc provides the Erlang runtime system with a number of memory allocators.

Allocators

The following allocators are present:

- **temp_alloc**: Allocator used for temporary allocations.
- **eheap_alloc**: Allocator used for Erlang heap data, such as Erlang process heaps.
- **binary_alloc**: Allocator used for Erlang binary data.
- **ets_alloc**: Allocator used for ets data.
- **driver_alloc**: Allocator used for driver data.
- **literal_alloc**: Allocator used for constant terms in Erlang code.
- **si_alloc**: Allocator used for memory blocks that are expected to be short-lived.
- **ll_alloc**: Allocator used for memory blocks that are expected to be long-lived, for example, Erlang code.
- **fix_alloc**: A fast allocator used for some frequently used fixed size data types.
- **std_alloc**: Allocator used for most memory blocks not allocated through any of the other allocators described above.
- **sys_alloc**: This is normally the default malloc implementation used on the specific OS.
- **mseg_alloc**: A memory segment allocator. It is used by other allocators for allocating memory segments and is only available on systems that have the mmap system call. Memory segments that are deallocated are kept for a while in a segment cache before they are destroyed. When segments are allocated, cached segments are used if possible instead of creating new segments. This to reduce the number of system calls made.

sys_alloc, literal_alloc and temp_alloc are always enabled and cannot be disabled. mseg_alloc is always enabled if it is available and an allocator that uses it is enabled. All other allocators can be enabled or disabled. By default all allocators are enabled. When an allocator is disabled, sys_alloc is used instead of the disabled allocator.

The main idea with the erts_alloc library is to separate memory blocks that are used differently into different memory areas, to achieve less memory fragmentation. By putting less effort in finding a good fit for memory blocks that are frequently allocated than for those less frequently allocated, a performance gain can be achieved.

The alloc_util Framework

Internally a framework called alloc_util is used for implementing allocators. sys_alloc and mseg_alloc do not use this framework, so the following does not apply to them.
An allocator manages multiple areas, called carriers, in which memory blocks are placed. A carrier is either placed in a separate memory segment (allocated through \texttt{mseg\_alloc}), or in the heap segment (allocated through \texttt{sys\_alloc}).

- Multiblock carriers are used for storage of several blocks.
- Singleblock carriers are used for storage of one block.
- Blocks that are larger than the value of the singleblock carrier threshold (\texttt{sbct}) parameter are placed in singleblock carriers.
- Blocks that are smaller than the value of parameter \texttt{sbct} are placed in multiblock carriers.

Normally an allocator creates a "main multiblock carrier". Main multiblock carriers are never deallocated. The size of the main multiblock carrier is determined by the value of parameter \texttt{mmbcs}.

Sizes of multiblock carriers allocated through \texttt{mseg\_alloc} are decided based on the following parameters:

- The values of the largest multiblock carrier size (\texttt{lmbcs})
- The smallest multiblock carrier size (\texttt{smbcs})
- The multiblock carrier growth stages (\texttt{mbcgs})

If \texttt{nc} is the current number of multiblock carriers (the main multiblock carrier excluded) managed by an allocator, the size of the next \texttt{mseg\_alloc} multiblock carrier allocated by this allocator is roughly \texttt{smbcs+nc*(lmbcs-smbcs)/mbcgs} when \texttt{nc <= mbcgs}, and \texttt{lmbcs} when \texttt{nc > mbcgs}. If the value of parameter \texttt{sbct} is larger than the value of parameter \texttt{lmbcs}, the allocator may have to create multiblock carriers that are larger than the value of parameter \texttt{lmbcs}, though. Singleblock carriers allocated through \texttt{mseg\_alloc} are sized to whole pages.

Sizes of carriers allocated through \texttt{sys\_alloc} are decided based on the value of the \texttt{sys\_alloc} carrier size (\texttt{ycs}) parameter. The size of a carrier is the least number of multiples of the value of parameter \texttt{ycs} satisfying the request.

Coalescing of free blocks are always performed immediately. Boundary tags (headers and footers) in free blocks are used, which makes the time complexity for coalescing constant.

The memory allocation strategy used for multiblock carriers by an allocator can be configured using parameter \texttt{as}. The following strategies are available:

**Best fit**

Strategy: Find the smallest block satisfying the requested block size.

Implementation: A balanced binary search tree is used. The time complexity is proportional to \texttt{log N}, where \texttt{N} is the number of sizes of free blocks.

**Address order best fit**

Strategy: Find the smallest block satisfying the requested block size. If multiple blocks are found, choose the one with the lowest address.

Implementation: A balanced binary search tree is used. The time complexity is proportional to \texttt{log N}, where \texttt{N} is the number of free blocks.

**Address order first fit**

Strategy: Find the block with the lowest address satisfying the requested block size.

Implementation: A balanced binary search tree is used. The time complexity is proportional to \texttt{log N}, where \texttt{N} is the number of free blocks.

**Address order first fit carrier best fit**

Strategy: Find the carrier with the lowest address that can satisfy the requested block size, then find a block within that carrier using the "best fit" strategy.

Implementation: Balanced binary search trees are used. The time complexity is proportional to \texttt{log N}, where \texttt{N} is the number of free blocks.
Address order first fit carrier address order best fit  
Strategy: Find the carrier with the lowest address that can satisfy the requested block size, then find a block within that carrier using the "address order best fit" strategy.  
Implementation: Balanced binary search trees are used. The time complexity is proportional to log N, where N is the number of free blocks.

Age order first fit carrier address order first fit  
Strategy: Find the oldest carrier that can satisfy the requested block size, then find a block within that carrier using the "address order first fit" strategy.  
Implementation: A balanced binary search tree is used. The time complexity is proportional to log N, where N is the number of free blocks.

Age order first fit carrier best fit  
Strategy: Find the oldest carrier that can satisfy the requested block size, then find a block within that carrier using the "best fit" strategy.  
Implementation: Balanced binary search trees are used. The time complexity is proportional to log N, where N is the number of free blocks.

Age order first fit carrier address order best fit  
Strategy: Find the oldest carrier that can satisfy the requested block size, then find a block within that carrier using the "address order best fit" strategy.  
Implementation: Balanced binary search trees are used. The time complexity is proportional to log N, where N is the number of free blocks.

Good fit  
Strategy: Try to find the best fit, but settle for the best fit found during a limited search.  
Implementation: The implementation uses segregated free lists with a maximum block search depth (in each list) to find a good fit fast. When the maximum block search depth is small (by default 3), this implementation has a time complexity that is constant. The maximum block search depth can be configured using parameter mbsd.

A fit  
Strategy: Do not search for a fit, inspect only one free block to see if it satisfies the request. This strategy is only intended to be used for temporary allocations.  
Implementation: Inspect the first block in a free-list. If it satisfies the request, it is used, otherwise a new carrier is created. The implementation has a time complexity that is constant.  
As from ERTS 5.6.1 the emulator refuses to use this strategy on other allocators than temp_alloc. This because it only causes problems for other allocators.

Apart from the ordinary allocators described above, some pre-allocators are used for some specific data types. These pre-allocators pre-allocate a fixed amount of memory for certain data types when the runtime system starts. As long as pre-allocated memory is available, it is used. When no pre-allocated memory is available, memory is allocated in ordinary allocators. These pre-allocators are typically much faster than the ordinary allocators, but can only satisfy a limited number of requests.
System Flags Effecting erts_alloc

**Warning:**

Only use these flags if you are sure what you are doing. Unsuitable settings can cause serious performance degradation and even a system crash at any time during operation.

Memory allocator system flags have the following syntax: `+M<S><P> <V>`, where `<S>` is a letter identifying a subsystem, `<P>` is a parameter, and `<V>` is the value to use. The flags can be passed to the Erlang emulator (`erl(1)`) as command-line arguments.

System flags effecting specific allocators have an uppercase letter as `<S>`. The following letters are used for the allocators:

- **B**: binary_alloc
- **D**: std_alloc
- **E**: ets_alloc
- **F**: fix_alloc
- **H**: eheap_alloc
- **I**: literal_alloc
- **L**: ll_alloc
- **M**: mseg_alloc
- **R**: driver_alloc
- **S**: sl_alloc
- **T**: temp_alloc
- **Y**: sys_alloc

**Flags for Configuration of mseg_alloc**

- **+MMamcbf <size>**
  
  Absolute maximum cache bad fit (in kilobytes). A segment in the memory segment cache is not reused if its size exceeds the requested size with more than the value of this parameter. Defaults to 4096.

- **+MMrmbcf <ratio>**
  
  Relative maximum cache bad fit (in percent). A segment in the memory segment cache is not reused if its size exceeds the requested size with more than relative maximum cache bad fit percent of the requested size. Defaults to 20.

- **+MMsco true|false**

  Sets super carrier only flag. Defaults to `true`. When a super carrier is used and this flag is `true`, mseg_alloc only creates carriers in the super carrier. Notice that the alloc_util framework can create sys_alloc carriers, so if you want all carriers to be created in the super carrier, you therefore want to disable use of sys_alloc carriers by also passing `+Musac false`. When the flag is `false`, mseg_alloc tries to create carriers outside of the super carrier when the super carrier is full.

**Note:**

Setting this flag to `false` is not supported on all systems. The flag is then ignored.
+MMscrfsd <amount>
Sets super carrier reserved free segment descriptors. Defaults to 65536. This parameter determines the amount of memory to reserve for free segment descriptors used by the super carrier. If the system runs out of reserved memory for free segment descriptors, other memory is used. This can however cause fragmentation issues, so you want to ensure that this never happens. The maximum amount of free segment descriptors used can be retrieved from the erts_mmap tuple part of the result from calling erlang:system_info({allocator, mseg_alloc}).

+MMscrpm true|false
Sets super carrier reserve physical memory flag. Defaults to true. When this flag is true, physical memory is reserved for the whole super carrier at once when it is created. The reservation is after that left unchanged. When this flag is set to false, only virtual address space is reserved for the super carrier upon creation. The system attempts to reserve physical memory upon carrier creations in the super carrier, and attempt to unreserve physical memory upon carrier destructions in the super carrier.

**Note:**
What reservation of physical memory means, highly depends on the operating system, and how it is configured. For example, different memory overcommit settings on Linux drastically change the behavior.
Setting this flag to false is possibly not supported on all systems. The flag is then ignored.

+MMscs <size in MB>
Sets super carrier size (in MB). Defaults to 0, that is, the super carrier is by default disabled. The super carrier is a large continuous area in the virtual address space. mseg_alloc always tries to create new carriers in the super carrier if it exists. Notice that the alloc_util framework can create sys_alloc carriers. For more information, see +MMsco.

+MMmcs <amount>
Maximum cached segments. The maximum number of memory segments stored in the memory segment cache. Valid range is \[0, 30\]. Defaults to 10.

Flags for Configuration of sys_alloc

+MYe true
Enables sys_alloc.

**Note:**
sys_alloc cannot be disabled.

+MYtt <size>
Trim threshold size (in kilobytes). This is the maximum amount of free memory at the top of the heap (allocated by sbrk) that is kept by malloc (not released to the operating system). When the amount of free memory at the top of the heap exceeds the trim threshold, malloc releases it (by calling sbrk). Trim threshold is specified in kilobytes. Defaults to 128.

**Note:**
This flag has effect only when the emulator is linked with the GNU C library, and uses its malloc implementation.
erts_alloc

+MYtp <size>

Top pad size (in kilobytes). This is the amount of extra memory that is allocated by malloc when sbrk is called to get more memory from the operating system. Defaults to 0.

Note:

This flag has effect only when the emulator is linked with the GNU C library, and uses its malloc implementation.

Flags for Configuration of Allocators Based on alloc_util

If u is used as subsystem identifier (that is, <S> = u), all allocators based on alloc_util are effected. If B, D, E, F, H, I, L, R, S, T, X is used as subsystem identifier, only the specific allocator identifier is effected.

+M<S>acul <utilization>|de

Abandon carrier utilization limit. A valid <utilization> is an integer in the range [0, 100] representing utilization in percent. When a utilization value > 0 is used, allocator instances are allowed to abandon multiblock carriers. If de (default enabled) is passed instead of a <utilization>, a recommended non-zero utilization value is used. The value chosen depends on the allocator type and can be changed between ERTS versions. Defaults to de, but this can be changed in the future.

Carriers are abandoned when memory utilization in the allocator instance falls below the utilization value used. Once a carrier is abandoned, no new allocations are made in it. When an allocator instance gets an increased multiblock carrier need, it first tries to fetch an abandoned carrier from another allocator instance. If no abandoned carrier can be fetched, it creates a new empty carrier. When an abandoned carrier has been fetched, it will function as an ordinary carrier. This feature has special requirements on the allocation strategy used. Only the strategies aoff, aoffcbf, aoffcaobf, ageffcaoff, ageffcbf and ageffcaobf support abandoned carriers.

This feature also requires multiple thread specific instances to be enabled. When enabling this feature, multiple thread-specific instances are enabled if not already enabled, and the aoffcbf strategy is enabled if the current strategy does not support abandoned carriers. This feature can be enabled on all allocators based on the alloc_util framework, except temp_alloc (which would be pointless).

+M<S>acnl <amount>

Abandon carrier number limit. A valid <amount> is a positive integer representing max number of abandoned carriers per allocator instance. Defaults to 1000 which will practically disable the limit, but this can be changed in the future.

See also acul.

+M<S>acful <utilization>|de

Abandon carrier free utilization limit. When the utilization of a carrier falls belows this limit erts_alloc instructs the OS that unused memory in the carrier can be re-used for allocation by other OS processes. On Unix this is done by calling madvise(..., ..., MADV_FREE) on the unused memory region, on Windows it is done by calling VirtualAlloc(..., ..., MEM_RESET, PAGE_READWRITE). Defaults to 0 which means that no memory will be marked as re-usable by the OS.
A valid \texttt{<utilization>} is an integer in the range \([0, 100]\) representing utilization in percent. If this value is larger than the \texttt{acul} limit it will be lowered to the current \texttt{acul} limit. If \texttt{de} (default enabled) is passed instead of a \texttt{<utilization>}, a recommended non-zero utilization value is used. The value chosen depends on the allocator type and can be changed between ERTS versions.

See also \texttt{acul}.

\texttt{+M<S>as \textbackslash bf|aobf|aoff|aoffcbf|aoffcaobf|ageffcaoff|ageffcbf|ageffcaobf|gf|af}

Allocation strategy. The following strategies are valid:

- \texttt{bf} (best fit)
- \texttt{aobf} (address order best fit)
- \texttt{aoff} (address order first fit)
- \texttt{aoffcbf} (address order first fit carrier best fit)
- \texttt{aoffcaobf} (address order first fit carrier address order best fit)
- \texttt{ageffcaoff} (age order first fit carrier address order first fit)
- \texttt{ageffcbf} (age order first fit carrier best fit)
- \texttt{ageffcaobf} (age order first fit carrier address order best fit)
- \texttt{gf} (good fit)
- \texttt{af} (a fit)

See the description of allocation strategies in section The alloc_util Framework.

\texttt{+M<S>asbcst <size>}

Absolute singleblock carrier shrink threshold (in kilobytes). When a block located in an \texttt{mseg_alloc} singleblock carrier is shrunk, the carrier is left unchanged if the amount of unused memory is less than this threshold, otherwise the carrier is shrunk. See also \texttt{rsbcst}.

\texttt{+M<S>atags true|false}

Adds a small tag to each allocated block that contains basic information about what it is and who allocated it. Use the \texttt{instrument} module to inspect this information.

The runtime overhead is one word per allocation when enabled. This may change at any time in the future.

The default is \texttt{true} for \texttt{binary_alloc} and \texttt{driver_alloc}, and \texttt{false} for the other allocator types.

\texttt{+M<S>cp B|D|E|F|H||L|R|S|\@|:}

Set carrier pool to use for the allocator. Memory carriers will only migrate between allocator instances that use the same carrier pool. The following carrier pool names exist:

- \texttt{B} Carrier pool associated with \texttt{binary_alloc}.
- \texttt{D} Carrier pool associated with \texttt{std_alloc}.
- \texttt{E} Carrier pool associated with \texttt{ets_alloc}.
- \texttt{F} Carrier pool associated with \texttt{fix_alloc}.
- \texttt{H} Carrier pool associated with \texttt{eheap_alloc}.
- \texttt{L} Carrier pool associated with \texttt{ll_alloc}.
- \texttt{R} Carrier pool associated with \texttt{driver_alloc}.
Carrier pool associated with `sl_alloc`.

Carrier pool associated with the system as a whole.

Besides passing carrier pool name as value to the parameter, you can also pass :. By passing : instead of carrier pool name, the allocator will use the carrier pool associated with itself. By passing the command line argument "+Mucg :", all allocators that have an associated carrier pool will use the carrier pool associated with themselves.

The association between carrier pool and allocator is very loose. The associations are more or less only there to get names for the amount of carrier pools needed and names of carrier pools that can be easily identified by the : value.

This flag is only valid for allocators that have an associated carrier pool. Besides that, there are no restrictions on carrier pools to use for an allocator.

Currently each allocator with an associated carrier pool defaults to using its own associated carrier pool.

`+M<s>e true|false`

Enables allocator <s>.

`+M<s>lmbcs <size>`

Largest (mseg_alloc) multiblock carrier size (in kilobytes). See the description on how sizes for mseg_alloc multiblock carriers are decided in section The alloc_util Framework. On 32-bit Unix style OS this limit cannot be set > 64 MB.

`+M<s>mbcgs <ratio>`

(mseg_alloc) multiblock carrier growth stages. See the description on how sizes for mseg_alloc multiblock carriers are decided in section The alloc_util Framework.

`+M<s>mbsd <depth>`

Maximum block search depth. This flag has effect only if the good fit strategy is selected for allocator <s>. When the good fit strategy is used, free blocks are placed in segregated free-lists. Each free-list contains blocks of sizes in a specific range. The maximum block search depth sets a limit on the maximum number of blocks to inspect in a free-list during a search for suitable block satisfying the request.

`+M<s>mmbcs <size>`

Main multiblock carrier size. Sets the size of the main multiblock carrier for allocator <s>. The main multiblock carrier is allocated through sys_alloc and is never deallocated.

`+M<s>mmmbc <amount>`

Maximum mseg_alloc multiblock carriers. Maximum number of multiblock carriers allocated through mseg_alloc by allocator <s>. When this limit is reached, new multiblock carriers are allocated through sys_alloc.

`+M<s>mmsbc <amount>`

Maximum mseg_alloc singleblock carriers. Maximum number of singleblock carriers allocated through mseg_alloc by allocator <s>. When this limit is reached, new singleblock carriers are allocated through sys_alloc.

`+M<s>ramv <bool>`

Realloc always moves. When enabled, reallocate operations are more or less translated into an allocate, copy, free sequence. This often reduces memory fragmentation, but costs performance.
+M<rmbcmt <ratio>
Relative multiblock carrier move threshold (in percent). When a block located in a multiblock carrier is shrunk, the block is moved if the ratio of the size of the freed memory compared to the previous size is more than this threshold, otherwise the block is shrunk at the current location.

+M<rscbmt <ratio>
Relative singleblock carrier move threshold (in percent). When a block located in a singleblock carrier is shrunk to a size smaller than the value of parameter sbct, the block is left unchanged in the singleblock carrier if the ratio of unused memory is less than this threshold, otherwise it is moved into a multiblock carrier.

+M<sbcst <ratio>
Relative singleblock carrier shrink threshold (in percent). When a block located in an mseg_alloc singleblock carrier is shrunk, the carrier is left unchanged if the ratio of unused memory is less than this threshold, otherwise the carrier is shrunk. See also asbcst.

+M<sbct <size>
Singleblock carrier threshold (in kilobytes). Blocks larger than this threshold are placed in singleblock carriers. Blocks smaller than this threshold are placed in multiblock carriers. On 32-bit Unix style OS this threshold cannot be set > 8 MB.

+M<smbcs <size>
Smallest (mseg_alloc) multiblock carrier size (in kilobytes). See the description on how sizes for mseg_alloc multiblock carriers are decided in section The alloc_util Framework.

+M<t true|false
Multiple, thread-specific instances of the allocator. Default behavior is NoSchedulers+1 instances. Each scheduler uses a lock-free instance of its own and other threads use a common instance.
Before ERTS 5.9 it was possible to configure a smaller number of thread-specific instances than schedulers. This is, however, not possible anymore.

Flags for Configuration of alloc_util
All allocators based on alloc_util are effected.

+M<syscs <size>
 sys_alloc carrier size. Carriers allocated through sys_alloc are allocated in sizes that are multiples of the sys_alloc carrier size. This is not true for main multiblock carriers and carriers allocated during a memory shortage, though.

+M<mummc <amount>
Maximum mseg_alloc carriers. Maximum number of carriers placed in separate memory segments. When this limit is reached, new carriers are placed in memory retrieved from sys_alloc.

+M<usac <bool>
Allow sys_alloc carriers. Defaults to true. If set to false, sys_alloc carriers are never created by allocators using the alloc_util framework.

Special Flag for literal_alloc

+M<iscs <size in MB>
literal_alloc super carrier size (in MB). The amount of virtual address space reserved for literal terms in Erlang code on 64-bit architectures. Defaults to 1024 (that is, 1 GB), which is usually sufficient. The flag is ignored on 32-bit architectures.
Instrumentation Flags

+M< tags

Adds a small tag to each allocated block that contains basic information about what it is and who allocated it. See +M< tags for a more complete description.

Note:

When instrumentation of the emulator is enabled, the emulator uses more memory and runs slower.

Other Flags

+Mea min|max|r9c|r10b|r11b|config

Options:

min

Disables all allocators that can be disabled.

max

Enables all allocators (default).

r9c|r10b|r11b

Configures all allocators as they were configured in respective Erlang/OTP release. These will eventually be removed.

+Mlpm all|no

Lock physical memory. Defaults to no, that is, no physical memory is locked. If set to all, all memory mappings made by the runtime system are locked into physical memory. If set to all, the runtime system fails to start if this feature is not supported, the user has not got enough privileges, or the user is not allowed to lock enough physical memory. The runtime system also fails with an out of memory condition if the user limit on the amount of locked memory is reached.

+Mdai max|<amount>

Set amount of dirty allocator instances used. Defaults to 0. That is, by default no instances will be used. The maximum amount of instances equals the amount of dirty CPU schedulers on the system.

By default, each normal scheduler thread has its own allocator instance for each allocator. All other threads in the system, including dirty schedulers, share one instance for each allocator. By enabling dirty allocator instances, dirty schedulers will get and use their own set of allocator instances. Note that these instances are not exclusive to each dirty scheduler, but instead shared among dirty schedulers. The more instances used the less risk of lock contention on these allocator instances. Memory consumption do however increase with increased amount of dirty allocator instances.

Notes

Only some default values have been presented here. For information about the currently used settings and the current status of the allocators, see erlang:system_info(allocator) and erlang:system_info({allocator, Alloc}).

Note:

Most of these flags are highly implementation-dependent and can be changed or removed without prior notice. erts_alloc is not obliged to strictly use the settings that have been passed to it (it can even ignore them).
See Also

erl(1), erlang(3), instrument(3)
escript provides support for running short Erlang programs without having to compile them first, and an easy way to retrieve the command-line arguments.

It is possible to bundle escript(s) with an Erlang runtime system to make it self-sufficient and relocatable. In such a standalone system, the escript(s) should be located in the top bin directory of the standalone system and given .escript as file extension. Further the (built-in) escript program should be copied to the same directory and given the script's original name (without the .escript extension). This will enable use of the bundled Erlang runtime system.

The (built-in) escript program first determines which Erlang runtime system to use and then starts it to execute your script. Usually the runtime system is located in the same Erlang installation as the escript program itself. But for standalone systems with one or more escripts it may be the case that the escript program in your path actually starts the runtime system bundled with the escript. This is intentional, and typically happens when the standalone system bin directory is not in the execution path (as it may cause its erl program to override the desired one) and the escript(s) are referred to via symbolic links from a bin directory in the path.

Exports

script-name script-arg1 script-arg2...
escript escript-flags script-name script-arg1 script-arg2...
escript runs a script written in Erlang.

Example:
$ chmod u+x factorial
$ cat factorial
#!/usr/bin/env escript
%% -*- erlang -*-
%%! -sname factorial -mnesia debug verbose
main([String]) ->
    try
        N = list_to_integer(String),
        F = fac(N),
        io:format("factorial ~w = ~w\n", [N,F])
    catch
        _:_ ->
            usage()
    end;
main(_) ->
    usage().
usage() ->
    io:format("usage: factorial integer\n"),
    halt(1).
fac(0) -> 1;
fac(N) -> N * fac(N-1).
$ ./factorial 5
factorial 5 = 120
$ ./factorial
usage: factorial integer
$ ./factorial five
usage: factorial integer

The header of the Erlang script in the example differs from a normal Erlang module. The first line is intended to be the interpreter line, which invokes escript.

However, if you invoke the escript as follows, the contents of the first line do not matter, but it cannot contain Erlang code as it will be ignored:

$ escript factorial 5

The second line in the example contains an optional directive to the Emacs editor, which causes it to enter the major mode for editing Erlang source files. If the directive is present, it must be located on the second line.

If a comment selecting the encoding exists, it can be located on the second line.

Note:

The encoding specified by the above mentioned comment applies to the script itself. The encoding of the I/O-server, however, must be set explicitly as follows:

    io:setopts([{encoding, latin1}])

The default encoding of the I/O-server for standard_io is unicode if its supported. (see section Summary of Options) in the STDLIB User's Guide.

On the third line (or second line depending on the presence of the Emacs directive), arguments can be specified to the emulator, for example:

    %%! -sname factorial -mnesia debug verbose
Such an argument line must start with \&\%\&! and the remaining line is interpreted as arguments to the emulator.

If you know the location of the escript executable, the first line can directly give the path to escript, for example:

```
#!/usr/local/bin/escript
```

As any other type of scripts, Erlang scripts do not work on Unix platforms if the execution bit for the script file is not set. (To turn on the execution bit, use `chmod +x script-name`.)

The remaining Erlang script file can either contain Erlang source code, an inlined beam file, or an inlined archive file.

An Erlang script file must always contain the `main/1` function. When the script is run, the `main/1` function is called with a list of strings representing the arguments specified to the script (not changed or interpreted in any way).

If the `main/1` function in the script returns successfully, the exit status for the script is 0. If an exception is generated during execution, a short message is printed and the script terminates with exit status 127.

To return your own non-zero exit code, call `halt(ExitCode)`, for example:

```
halt(1).
```

To retrieve the pathname of the script, call `escript:script_name()` from your script (the pathname is usually, but not always, absolute).

If the file contains source code (as in the example above), it is processed by the epp preprocessor. This means that you, for example, can use predefined macros (such as `?MODULE`) and include directives like the `-include_lib` directive. For example, use

```
-include_lib("kernel/include/file.hrl").
```

to include the record definitions for the records used by function `file:read_link_info/1`. You can also select encoding by including an encoding comment here, but if a valid encoding comment exists on the second line, it takes precedence.

The script is checked for syntactic and semantic correctness before it is run. If there are warnings (such as unused variables), they are printed and the script will still be run. If there are errors, they are printed and the script will not be run and its exit status is 127.

Both the module declaration and the export declaration of the `main/1` function are optional.

By default, the script will be interpreted. You can force it to be compiled by including the following line somewhere in the script file:

```
-mode(compile).
```

Execution of interpreted code is slower than compiled code. If much of the execution takes place in interpreted code, it can be worthwhile to compile it, although the compilation itself takes a little while.

As mentioned earlier, a script can contains precompiled beam code. In a precompiled script, the interpretation of the script header is the same as in a script containing source code. This means that you can make a beam file executable by prepending the file with the lines starting with `#!` and \&\%\&! mentioned above. In a precompiled script, the `main/1` function must be exported.

Another option is to have an entire Erlang archive in the script. In an archive script, the interpretation of the script header is the same as in a script containing source code. This means that you can make an archive file executable by prepending the file with the lines starting with `#!` and \&\%\&! mentioned above. In an archive script, the `main/1` function must be exported. By default the `main/1` function in the module with the same name as the basename of the escript file is invoked. This behavior can be overridden by setting flag `-escript main Module` as one of the
emulator flags. Module must be the name of a module that has an exported `main/1` function. For more information about archives and code loading, see `code(3)`.

It is often very convenient to have a header in the escript, especially on Unix platforms. However, the header is optional, so you directly can "execute" an Erlang module, Beam file, or archive file without adding any header to them. But then you have to invoke the script as follows:

```
$ escript factorial.erl 5
factorial 5 = 120
$ escript factorial.beam 5
factorial 5 = 120
$ escript factorial.zip 5
factorial 5 = 120
```

escript:create(FileOrBin, Sections) -> ok | {ok, binary()} | {error, term()}

Types:

```
FileOrBin = filename() | 'binary'
Sections = [Header] Body | Body
  Header = shebang | {shebang, Shebang} | comment | {comment, Comment}
            | {emu_args, EmuArgs}
  Shebang = string() | 'default' | 'undefined'
  Comment = string() | 'default' | 'undefined'
  EmuArgs = string() | 'undefined'
  Body = {source, SourceCode} | {beam, BeamCode} | {archive, ZipArchive}
          | {archive, ZipFiles, ZipOptions}
  SourceCode = BeamCode = file:filename() | binary()
  ZipArchive = zip:filename() | binary()
  ZipFiles = [ZipFile]
  ZipFile = file:filename() | {file:filename(), binary()}
           | {file:filename(), binary(), file:file_info()}
  ZipOptions = [ zip:create_option()]
```

Creates an escript from a list of sections. The sections can be specified in any order. An escript begins with an optional `Header` followed by a mandatory `Body`. If the header is present, it does always begin with a shebang, possibly followed by a comment and `emu_args`. The shebang defaults to "/usr/bin/env escript". The comment defaults to "This is an -*- erlang -*- file". The created escript can either be returned as a binary or written to file.

As an example of how the function can be used, we create an interpreted escript that uses `emu_args` to set some emulator flag. In this case, it happens to set number of schedulers with `+S3`. We also extract the different sections from the newly created script:
An escript without header can be created as follows:

```erlang
> file:write_file("demo.erl", ["%% demo.erl
-module(demo).
-export([main/1]).

", Source"]).
ok
> {ok, _, BeamCode} = compile:file("demo.erl", [binary, debug_info]).
{ok,demo,
<<70,79,82,49,0,0,2,208,66,69,65,77,65,116,111,109,0,0,0,
79,0,0,9,4,100,...>>}
> escript:create("demo.beam", [{beam,BeamCode}]).
ok
> escript:extract("demo.beam", []).
{ok,[{shebang,undefined}, {comment,undefined}, {emu_args,undefined},
{beam,<<70,79,82,49,0,0,3,68,66,69,65,77,65,116,
111,109,0,0,0,83,0,0,9,...>>}]}
> os:cmd("escript demo.beam").
"true"
```

Here we create an archive script containing both Erlang code and Beam code, then we iterate over all files in the archive and collect their contents and some information about them:
escript:extract(File, Options) -> {ok, Sections} | {error, term()}

Types:

File = filename()

Options = [] | [compile_source]

Sections = Headers Body

Headers = {shebang, Shebang} {comment, Comment} {emu_args, EmuArgs}

Shebang = string() | 'default' | 'undefined'

Comment = string() | 'default' | 'undefined'

EmuArgs = string() | 'undefined'

Body = {source, SourceCode} | {source, BeamCode} | {beam, BeamCode} | {archive, ZipArchive}

SourceCode = BeamCode = ZipArchive = binary()

Parses an escript and extracts its sections. This is the reverse of create/2.

All sections are returned even if they do not exist in the escript. If a particular section happens to have the same value as the default value, the extracted value is set to the atom default. If a section is missing, the extracted value is set to the atom undefined.

Option compile_source only affects the result if the escript contains source code. In this case the Erlang code is automatically compiled and {source, BeamCode} is returned instead of {source, SourceCode}.

Example:
escript

> escript:create("demo.escript",
    [shebang, {archive, [{"demo.erl", SourceCode},
                         {"demo.beam", BeamCode}], []}]).
ok
> {ok, [{shebang,default}, {comment,undefined}, {emu_args,undefined},
        {archive, ArchiveBin}]} =
    escript:extract("demo.escript", []).
{ok,{{archive,<<80,75,3,4,20,0,0,0,8,0,118,7,98,60,105,
       152,61,93,107,0,0,0,118,0,...>>}
        {emu_args,undefined}}}
init

Erlang module

This module is preloaded and contains the code for the init system process that coordinates the startup of the system. The first function evaluated at startup is `boot(BootArgs)`, where `BootArgs` is a list of command-line arguments supplied to the Erlang runtime system from the local operating system; see `erl(1)`. `init` reads the boot script, which contains instructions on how to initiate the system. For more information about boot scripts, see `script(4)`. `init` also contains functions to restart, reboot, and stop the system.

Data Types

`mode() = embedded | interactive`

Code loading mode.

Exports

```erlang
boot(BootArgs) -> no_return()
```

Types:

`BootArgs = [binary()]`

Starts the Erlang runtime system. This function is called when the emulator is started and coordinates system startup. `BootArgs` are all command-line arguments except the emulator flags, that is, flags and plain arguments; see `erl(1)`. `init` interprets some of the flags, see section Command-Line Flags below. The remaining flags ("user flags") and plain arguments are passed to the `init` loop and can be retrieved by calling `get_arguments/0` and `get_plain_arguments/0`, respectively.

```erlang
get_argument(Flag) -> {ok, Arg} | error
```

Types:

`Flag = atom()
Arg = [Values :: [string()]]`

Returns all values associated with the command-line user flag `Flag`. If `Flag` is provided several times, each `Values` is returned in preserved order. Example:

```erl
% erl -a b c -a d
... 1> init:get_argument(a).
{ok,[{"b","c"},{"d"}]
```

The following flags are defined automatically and can be retrieved using this function:

```erlang
root
```

The installation directory of Erlang/OTP, `$ROOT`:

```erl
2> init:get_argument(root).
{ok,[{"/usr/local/otp/releases/otp_beam_solaris8_r180_b patched"}]
```
The name of the program which started Erlang:

```erlang
3> init:get_argument(progname).
{ok,["erl"]}
```

The home directory (on Unix, the value of $HOME):

```erlang
4> init:get_argument(home).
{ok,["/home/harry"]}
```

Returns error if no value is associated with Flag.

**get_arguments() -> Flags**

Types:
```
Flags = [{Flag :: atom(), Values :: [string()]}
```

Returns all command-line flags and the system-defined flags, see `get_argument/1`.

**get_plain_arguments() -> [Arg]**

Types:
```
Arg = string()
```

Returns any plain command-line arguments as a list of strings (possibly empty).

**get_status() -> {InternalStatus, ProvidedStatus}**

Types:
```
InternalStatus = internal_status()
ProvidedStatus = term()
```

The current status of the `init` process can be inspected. During system startup (initialization), InternalStatus is starting, and ProvidedStatus indicates how far the boot script has been interpreted. Each `{progress, Info}` term interpreted in the boot script affects ProvidedStatus, that is, ProvidedStatus gets the value of Info.

**reboot() -> ok**

All applications are taken down smoothly, all code is unloaded, and all ports are closed before the system terminates. If command-line flag `-heart` was specified, the heart program tries to reboot the system. For more information, see `heart(3)`.

To limit the shutdown time, the time `init` is allowed to spend taking down applications, command-line flag `-shutdown_time` is to be used.

**restart() -> ok**

The same as `restart([])`.
init

restart(Opts :: [{mode, mode()}'}) -> ok

The system is restarted inside the running Erlang node, which means that the emulator is not restarted. All applications are taken down smoothly, all code is unloaded, and all ports are closed before the system is booted again in the same way as initially started.

The same BootArgs are used when restarting the system unless the mode option is given, allowing the code loading mode to be set to either embedded or interactive. All other BootArgs remain the same.

To limit the shutdown time, the time init is allowed to spend taking down applications, command-line flag –shutdown_time is to be used.

script_id() -> Id

Types:

    Id = term()

Gets the identity of the boot script used to boot the system. Id can be any Erlang term. In the delivered boot scripts, Id is {Name, Vsn}. Name and Vsn are strings.

stop() -> ok

The same as stop(0).

stop(Status) -> ok

Types:

    Status = integer() >= 0 | string()

All applications are taken down smoothly, all code is unloaded, and all ports are closed before the system terminates by calling halt(Status). If command-line flag –heart was specified, the heart program is terminated before the Erlang node terminates. For more information, see heart(3).

To limit the shutdown time, the time init is allowed to spend taking down applications, command-line flag –shutdown_time is to be used.

Command-Line Flags

Warning:

The support for loading of code from archive files is experimental. The only purpose of releasing it before it is ready is to obtain early feedback. The file format, semantics, interfaces, and so on, can be changed in a future release. The –code_path_choice flag is also experimental.

The init module interprets the following command-line flags:

--

   Everything following -- up to the next flag is considered plain arguments and can be retrieved using get_plain_arguments/0.

-code_path_choice Choice

   Can be set to strict or relaxed. It controls how each directory in the code path is to be interpreted:

   • Strictly as it appears in the boot script, or
   • init is to be more relaxed and try to find a suitable directory if it can choose from a regular ebin directory and an ebin directory in an archive file.
This flag is particularly useful when you want to elaborate with code loading from archives without editing the boot script. For more information about interpretation of boot scripts, see `script(4)`. The flag has also a similar effect on how the code server works; see `code(3)`.

**-epmd_module Module**

Specifies the module to use for registration and lookup of node names. Defaults to `erl_epmd`.

**-eval Expr**

Scans, parses, and evaluates an arbitrary expression `Expr` during system initialization. If any of these steps fail (syntax error, parse error, or exception during evaluation), Erlang stops with an error message. In the following example Erlang is used as a hexadecimal calculator:

```bash
% erl -noshell -eval 'R = 16#1F+16#A0, io:format("\.16B~n", [R])' \ \
-s erlang halt
BF
```

If multiple `-eval` expressions are specified, they are evaluated sequentially in the order specified. `-eval` expressions are evaluated sequentially with `-s` and `-run` function calls (this also in the order specified). As with `-s` and `-run`, an evaluation that does not terminate blocks the system initialization process.

**-extra**

Everything following `-extra` is considered plain arguments and can be retrieved using `get_plain_arguments/0`.

**-run Mod [Func [Arg1, Arg2, ...]]**

Evaluates the specified function call during system initialization. `Func` defaults to `start`. If no arguments are provided, the function is assumed to be of arity 0. Otherwise it is assumed to be of arity 1, taking the list `[Arg1,Arg2,...]` as argument. All arguments are passed as strings. If an exception is raised, Erlang stops with an error message.

Example:

```bash
% erl -run foo -run foo bar -run foo bar baz 1 2
```

This starts the Erlang runtime system and evaluates the following functions:

```bash
foo:start()
foo:bar()
foo:bar(["baz", "1", "2"]).
```

The functions are executed sequentially in an initialization process, which then terminates normally and passes control to the user. This means that a `-run` call that does not return blocks further processing; to avoid this, use some variant of `spawn` in such cases.

**-s Mod [Func [Arg1, Arg2, ...]]**

Evaluates the specified function call during system initialization. `Func` defaults to `start`. If no arguments are provided, the function is assumed to be of arity 0. Otherwise it is assumed to be of arity 1, taking the list `[Arg1,Arg2,...]` as argument. All arguments are passed as atoms. If an exception is raised, Erlang stops with an error message.

Example:

```bash
% erl -s foo -s foo bar -s foo bar baz 1 2
```

This starts the Erlang runtime system and evaluates the following functions:
foo:start()
foo:bar()
foo:bar([baz, '1', '2']).

The functions are executed sequentially in an initialization process, which then terminates normally and passes control to the user. This means that a `~a` call that does not return blocks further processing; to avoid this, use some variant of `spawn` in such cases.

Because of the limited length of atoms, it is recommended to use `~run` instead.

Example

```
% erl -- a b -children thomas claire -ages 7 3 -- x y
...
1> init:get_plain_arguments().

["a","b","x","y"]
2> init:get_argument(children).

{ok,[["thomas","claire"]]}  
3> init:get_argument(ages).

{ok,[["7","3"]]}  
4> init:get_argument(silly).

error
```

See Also

`erl_prim_loader(3), heart(3)`
**persistent_term**

Erlang module

This module is similar to **ets** in that it provides a storage for Erlang terms that can be accessed in constant time, but with the difference that **persistent_term** has been highly optimized for reading terms at the expense of writing and updating terms. When a persistent term is updated or deleted, a global garbage collection pass is run to scan all processes for the deleted term, and to copy it into each process that still uses it. Therefore, **persistent_term** is suitable for storing Erlang terms that are frequently accessed but never or infrequently updated.

**Warning:**

Persistent terms is an advanced feature and is not a general replacement for ETS tables. Before using persistent terms, make sure to fully understand the consequence to system performance when updating or deleting persistent terms.

Term lookup (using **get/1**) is done in constant time and without taking any locks, and the term is **not** copied to the heap (as is the case with terms stored in ETS tables).

Storing or updating a term (using **put/2**) is proportional to the number of already created persistent terms because the hash table holding the keys will be copied. In addition, the term itself will be copied.

When a (complex) term is deleted (using **erase/1**) or replaced by another (using **put/2**), a global garbage collection is initiated. It works like this:

- All processes in the system will be scheduled to run a scan of their heaps for the term that has been deleted. While such scan is relatively light-weight, if there are many processes, the system can become less responsive until all processes have scanned their heaps.
- If the deleted term (or any part of it) is still used by a process, that process will do a major (fullsweep) garbage collection and copy the term into the process. However, at most two processes at a time will be scheduled to do that kind of garbage collection.

Deletion of atoms and other terms that fit in one machine word is specially optimized to avoid doing a global GC. It is still not recommended to update persistent terms with such values too frequently because the hash table holding the keys is copied every time a persistent term is updated.

Some examples are suitable uses for persistent terms are:

- Storing of configuration data that must be easily accessible by all processes.
- Storing of references for NIF resources.
- Storing of references for efficient counters.
- Storing an atom to indicate a logging level or whether debugging is turned on.

### Storing Huge Persistent Terms

The current implementation of persistent terms uses the literal allocator also used for literals (constant terms) in BEAM code. By default, 1 GB of virtual address space is reserved for literals in BEAM code and persistent terms. The amount of virtual address space reserved for literals can be changed by using the **+MIs cs** option when starting the emulator.

Here is an example how the reserved virtual address space for literals can be raised to 2 GB (2048 MB):

```
erl +MIs cs 2048
```
Best Practices for Using Persistent Terms

It is recommended to use keys like `?MODULE` or `{?MODULE, SubKey}` to avoid name collisions.

Prefer creating a few large persistent terms to creating many small persistent terms. The execution time for storing a persistent term is proportional to the number of already existing terms.

Updating a persistent term with the same value as it already has is specially optimized to do nothing quickly; thus, there is no need to compare the old and new values and avoid calling `put/2` if the values are equal.

When atoms or other terms that fit in one machine word are deleted, no global GC is needed. Therefore, persistent terms that have atoms as their values can be updated more frequently, but note that updating such persistent terms is still much more expensive than reading them.

Updating or deleting a persistent term will trigger a global GC if the term does not fit in one machine word. Processes will be scheduled as usual, but all processes will be made runnable at once, which will make the system less responsive until all processes have run and scanned their heaps for the deleted terms. One way to minimize the effects on responsiveness could be to minimize the number of processes on the node before updating or deleting a persistent term. It would also be wise to avoid updating terms when the system is at peak load.

Avoid storing a retrieved persistent term in a process if that persistent term could be deleted or updated in the future. If a process holds a reference to a persistent term when the term is deleted, the process will be garbage collected and the term copied to the process.

Avoid updating or deleting more than one persistent term at a time. Each deleted term will trigger its own global GC. That means that deleting N terms will make the system less responsive N times longer than deleting a single persistent term. Therefore, terms that are to be updated at the same time should be collected into a larger term, for example, a map or a tuple.

Example

The following example shows how lock contention for ETS tables can be minimized by having one ETS table for each scheduler. The table identifiers for the ETS tables are stored as a single persistent term:

```erlang
%% There is one ETS table for each scheduler.
Sid = erlang:system_info(scheduler_id),
Tid = element(Sid, persistent_term:get(?MODULE)),
ets:update_counter(Tid, Key, 1).
```

Data Types

- `key() = term()`
  Any Erlang term.
- `value() = term()`
  Any Erlang term.

Exports

- `erase(Key) -> Result`
  Types:
  ```erlang
  Key = key()
  Result = boolean()
  ```
  Erase the name for the persistent term with key Key. The return value will be true if there was a persistent term with the key Key, and false if there was no persistent term associated with the key.
If there existed a previous persistent term associated with key \texttt{Key}, a global GC has been initiated when \texttt{erase/1} returns. See Description.

\textbf{get()} -> \textbf{List}

Types:
\begin{verbatim}
List = [{key(), value()}]
\end{verbatim}

Retrieve the keys and values for all persistent terms. The keys will be copied to the heap for the process calling \texttt{get/0}, but the values will not.

\textbf{get(Key)} -> \textbf{Value}

Types:
\begin{verbatim}
  Key = key()
  Value = value()
\end{verbatim}

Retrieve the value for the persistent term associated with the key \texttt{Key}. The lookup will be made in constant time and the value will not be copied to the heap of the calling process.

This function fails with a \texttt{badarg} exception if no term has been stored with the key \texttt{Key}.

If the calling process holds on to the value of the persistent term and the persistent term is deleted in the future, the term will be copied to the process.

\textbf{get(Key, Default)} -> \textbf{Value}

Types:
\begin{verbatim}
  Key = key()
  Default = Value = value()
\end{verbatim}

Retrieve the value for the persistent term associated with the key \texttt{Key}. The lookup will be made in constant time and the value will not be copied to the heap of the calling process.

This function returns \texttt{Default} if no term has been stored with the key \texttt{Key}.

If the calling process holds on to the value of the persistent term and the persistent term is deleted in the future, the term will be copied to the process.

\textbf{info()} -> \textbf{Info}

Types:
\begin{verbatim}
  Info = #{
    count := Count,
    memory := Memory
  }
  Count = Memory = integer() \geq 0
\end{verbatim}

Return information about persistent terms in a map. The map has the following keys:

\begin{itemize}
  \item \texttt{count} The number of persistent terms.
  \item \texttt{memory} The total amount of memory (measured in bytes) used by all persistent terms.
\end{itemize}

\textbf{put(Key, Value)} -> \textbf{ok}

Types:
Key = key()
Value = value()

Store the value Value as a persistent term and associate it with the key Key.

If the value Value is equal to the value previously stored for the key, put/2 will do nothing and return quickly.

If there existed a previous persistent term associated with key Key, a global GC has been initiated when put/2 returns. See Description.
run_erl

Command

The run_erl program is specific to Unix systems. This program redirects the standard input and standard output streams so that all output can be logged. It also lets the program to_erl connect to the Erlang console, making it possible to monitor and debug an embedded system remotely.

For more information about the use, see the Embedded System User's Guide in System Documentation.

Exports

run_erl [-daemon] pipe_dir/ log_dir "exec command arg1 arg2 ...

Arguments:

-daemon

This option is highly recommended. It makes run_erl run in the background completely detached from any controlling terminal and the command returns to the caller immediately. Without this option, run_erl must be started using several tricks in the shell to detach it completely from the terminal in use when starting it. The option must be the first argument to run_erl on the command line.

pipe_dir

The named pipe, usually /tmp/. It must be suffixed by a / (slash), that is, /tmp/epipes/, not /tmp/epipes.

log_dir

The log files, that is:

- One log file, run_erl.log, which logs progress and warnings from the run_erl program itself.
- Up to five log files at maximum 100 KB each with the content of the standard streams from and to the command. (Both the number of logs and sizes can be changed by environment variables, see section Environment Variables below.) When the logs are full, run_erl deletes and reuses the oldest log file.

"exec command arg1 arg2 ...

A space-separated string specifying the program to be executed. The second field is typically a command name such as erl.

Notes concerning the Log Files

While running, run_erl sends all output, uninterpreted, to a log file. The file is named erlang.log.N, where N is an integer. When the log is "full" (default log size is 100 KB), run_erl starts to log in file erlang.log. (N+1), until N reaches a certain number (default 5), whereupon N starts at 1 again and the oldest files start getting overwritten.

If no output comes from the Erlang shell, but the Erlang machine still seems to be alive, an "ALIVE" message is written to the log; it is a time stamp and is written, by default, after 15 minutes of inactivity. Also, if output from Erlang is logged, but more than 5 minutes (default) has passed since last time we got anything from Erlang, a time stamp is written in the log. The "ALIVE" messages look as follows:

===== ALIVE <date-time-string>

The other time stamps look as follows:
date-time-string is the date and time the message is written, default in local time (can be changed to UTC if needed). It is formatted with the ANSI-C function strftime using the format string %a %b %e %T %Z %Y, which produces messages like ===== ALIVE Thu May 15 10:13:36 MEST 2003; this can be changed, see the next section.

Environment Variables

The following environment variables are recognized by run_erl and change the logging behavior. For more information, see the previous section.

RUN_ERL_LOG_ALIVE_MINUTES
How long to wait for output (in minutes) before writing an "ALIVE" message to the log. Defaults to 15, minimum is 1.

RUN_ERL_LOG_ACTIVITY_MINUTES
How long Erlang needs to be inactive before output is preceded with a time stamp. Defaults to RUN_ERL_LOG_ALIVE_MINUTES div 3, minimum is 1.

RUN_ERL_LOG_ALIVE_FORMAT
Specifies another format string to be used in the strftime C library call. That is, specifying this to "%e-%b-%Y, %T %Z" gives log messages with time stamps like 15-May-2003, 10:23:04 MET. For more information, see the documentation for the C library function strftime. Defaults to "%a %b %e %T %Z %Y".

RUN_ERL_LOG_ALIVE_IN_UTC
If set to anything else than 0, it makes all times displayed by run_erl to be in UTC (GMT, CET, MET, without Daylight Saving Time), rather than in local time. This does not affect data coming from Erlang, only the logs output directly by run_erl. Application SASL can be modified accordingly by setting the Erlang application variable utc_log to true.

RUN_ERL_LOG_GENERATIONS
Controls the number of log files written before older files are reused. Defaults to 5, minimum is 2, maximum is 1000.

Note that, as a way to indicate the newest file, run_erl will delete the oldest log file to maintain a "hole" in the file sequences. For example, if log files #1, #2, #4 and #5 exists, that means #2 is the latest and #4 is the oldest. You will therefore at most get one less log file than the value set by RUN_ERL_LOG_GENERATIONS.

RUN_ERL_LOG_MAXSIZE
The size, in bytes, of a log file before switching to a new log file. Defaults to 100000, minimum is 1000, maximum is about 2^30.

RUN_ERL_DISABLE_FLOWCNTRL
If defined, disables input and output flow control for the pty opened by run_erl. Useful if you want to remove any risk of accidentally blocking the flow control by using Ctrl-S (instead of Ctrl-D to detach), which can result in blocking of the entire Beam process, and in the case of running heart as supervisor even the heart process becomes blocked when writing log message to terminal, leaving the heart process unable to do its work.

See Also

start(1), start_erl(1)
**start**

Command

The `start` script is an example script on how to start up the Erlang system in embedded mode on Unix. For more information about the use, see the Embedded System User's Guide in System Documentation.

**Exports**

```
start [ data_file ]
```

Argument:

data_file

Optional. Specifies what `start_erl.data` file to use.

Environment variable RELDIR can be set before calling this example, which sets the directory where to find the release files.

**See Also**

`run_erl(1), start_erl(1)`
The `start_erl` program is specific to Windows NT/2000/XP (and later versions of Windows). Although there are programs with the same name on other platforms, their functionality is different.

This program is distributed both in compiled form (under `<Erlang root>\erts-<version>\bin`) and in source form (under `<Erlang root>\erts-<version>\src`). The purpose of the source code is to ease customization of the program for local needs, such as cyclic restart detection. There is also a "make"-file, written for the `nmake` program distributed with Microsoft Visual C++. This program can, however, be compiled with any Win32 C compiler (possibly with minor modifications).

This program aids release handling on Windows systems. The program is to be called by the `erlsrv` program, read up the release data file `start_erl.data`, and start Erlang. Some options to `start_erl` are added and removed by the release handler during upgrade with emulator restart (more specifically option `-data`).

### Exports

```
start_erl [<erl options>] ++ [<start_erl options>]
```

The `start_erl` program in its original form recognizes the following options:

++

Mandatory. Delimits `start_erl` options from normal Erlang options. Everything on the command line before ++ is interpreted as options to be sent to the `erl` program. Everything after ++ is interpreted as options to `start_erl` itself.

```
-reldir <release root>
```

Mandatory if environment variable RELDIR is not specified and no `-rootdir` option is specified. Tells `start_erl` where the root of the release tree is located in the file system (typically `<Erlang root>\releases`). The `start_erl.data` file is expected to be located in this directory (unless otherwise specified). If only option `-rootdir` is specified, the directory is assumed to be `<Erlang root>\releases`.

```
-rootdir <Erlang root directory>
```

Mandatory if `-reldir` is not specified and no RELDIR exists in the environment. This specifies the Erlang installation root directory (under which the `lib`, `releases`, and `erts-<Version>` directories are located). If only `-reldir` (or environment variable RELDIR) is specified, the Erlang root is assumed to be the directory exactly one level above the release directory.

```
-data <data file name>
```

Optional. Specifies another data file than `start_erl.data` in the `<release root>`. It is specified relative to the `<release root>` or absolute (including drive letter, and so on). This option is used by the release handler during upgrade and is not to be used during normal operation. Normally the release data file is not to be named differently.

```
-bootflags <boot flags file name>
```

Optional. Specifies a file name relative to the release directory (that is, the subdirectory of `<release root>` where the `.boot` file and others are located). The contents of this file are appended to the command line when Erlang is started. This makes it easy to start the emulator with different options for different releases.
Notes

• As the source code is distributed, it can easily be modified to accept other options. The program must still accept option -data with the semantics described above for the release handler to work correctly.
• The Erlang emulator is found by examining the registry keys for the emulator version specified in the release data file. The new emulator must be properly installed before the upgrade for this to work.
• Although the program is located together with files specific to the emulator version, it is not expected to be specific to the emulator version. The release handler does not change option -machine to erlsrv during emulator restart. Locate the (possibly customized) start_erl program so that it is not overwritten during upgrade.
• The default options of the erlsrv program are not sufficient for release handling. The machine started by erlsrv is be specified as the start_erl program and the arguments are to contain ++ followed by the desired options.

See Also

erlsrv(1), release_handler(3)
On Windows, the preferred way to start the Erlang system for interactive use is as follows:

```
werl <arguments>
```

This starts Erlang in its own window, with fully functioning command-line editing and scrollbars. All flags except `oldshell` work as they do for `erl(1)`.

- To copy text to the clipboard, use `Ctrl-C`.
- To paste text, use `Ctrl-V`.
- To interrupt the runtime system or the shell process (depending on what has been specified with system flag `+B`), use `Ctrl-Break`.

In cases where you want to redirect standard input and/or standard output or use Erlang in a pipeline, `werl` is not suitable, and the `erl` program is to be used instead.

The `werl` window is in many ways modeled after the `xterm` window present on other platforms, as the `xterm` model fits well with line-oriented command-based interaction. This means that selecting text is line-oriented rather than rectangle-oriented.

- To select text in the `werl` window, press and hold the left mouse button and drag the mouse over the text you want to select. If the selection crosses line boundaries, the selected text consists of complete lines where applicable (just like in a word processor).
- To select more text than fits in the window, start by selecting a small part in the beginning of the text you want, then use the scrollbar to view the end of the desired selection, point to it, and press the `right` mouse button. The whole area between your first selection and the point where you right-clicked is included in the selection.
- To copy the selected text to the clipboard, either use `Ctrl-C`, use the menu, or press the copy button in the toolbar. Pasted text is inserted at the current prompt position and is interpreted by Erlang as usual keyboard input.
- To retrieve previous command lines, press the `Up arrow` or use `Ctrl-P`.

A drop-down box in the toolbar contains the command history. Selecting a command in the drop-down box inserts the command at the prompt, as if you used the keyboard to retrieve the command.

- To stop the Erlang emulator, close the `werl` window.
This module provides an API for the zlib library (www.zlib.net). It is used to compress and decompress data. The data format is described by RFC 1950, RFC 1951, and RFC 1952.

A typical (compress) usage is as follows:

```erlang
Z = zlib:open(),
ok = zlib:deflateInit(Z, default),
Compress = fun(end_of_data, Cont) -> [];
    (Data, Cont) ->
        [zlib:deflate(Z, Data)|Cont(Read(), Cont)]
    end,
Compressed = Compress(Read(), Compress),
Last = zlib:deflate(Z, [], finish),
ok = zlib:deflateEnd(Z),
zlib:close(Z),
list_to_binary([Compressed|Last])
```

In all functions errors, {'EXIT', {Reason, Backtrace}}, can be thrown, where Reason describes the error.

Typical Reasons:

- **badarg**
  - Bad argument.
- **not_initialized**
  - The stream hasn't been initialized, e.g. if `inflateInit/1` wasn't called prior to a call to `inflate/2`.
- **not_on_controlling_process**
  - The stream was used by a process that doesn't control it. Use `set_controlling_process/2` if you need to transfer a stream to a different process.
- **data_error**
  - The data contains errors.
- **stream_error**
  - Inconsistent stream state.
- **{need_dictionary, Adler32}**
  - See `inflate/2`.

**Data Types**

- **zstream() = reference()**
  - A zlib stream, see `open/0`.
- **zlevel()**
  - `none` | `default` | `best_compression` | `best_speed` | `0..9`
- **zflush()**
  - `none` | `sync` | `full` | `finish`
- **zmemlevel()**
  - `1..9`
- **zmethod()** = `deflated`
- **zstrategy()** = `default` | `filtered` | `huffman_only` | `rle`
- **zwindowbits()** = `-15..-8 | 8..47`

Normally in the range `-15..-8 | 8..15`. 
Exports

adler32(Z, Data) -> CheckSum

Types:
  Z = zstream()
  Data = iodata()
  CheckSum = integer() >= 0

Calculates the Adler-32 checksum for Data.

Warning:
This function is deprecated and will be removed in a future release. Use erlang:adler32/1 instead.

adler32(Z, PrevAdler, Data) -> CheckSum

Types:
  Z = zstream()
  PrevAdler = integer() >= 0
  Data = iodata()
  CheckSum = integer() >= 0

Updates a running Adler-32 checksum for Data. If Data is the empty binary or the empty iolist, this function returns the required initial value for the checksum.

Example:

Cr = lists:foldl(fun(Data,Cr) ->
      zlib:adler32(Z, Cr0, Data),
      end, zlib:adler32(Z,<< >>), Datas)

Warning:
This function is deprecated and will be removed in a future release. Use erlang:adler32/2 instead.

adler32_combine(Z, Adler1, Adler2, Size2) -> Adler

Types:
  Z = zstream()
  Adler = Adler1 = Adler2 = Size2 = integer() >= 0

Combines two Adler-32 checksums into one. For two binaries or iolists, Data1 and Data2 with sizes of Size1 and Size2, with Adler-32 checksums Adler1 and Adler2.

This function returns the Adler checksum of [Data1,Data2], requiring only Adler1, Adler2, and Size2.

Warning:
This function is deprecated and will be removed in a future release. Use erlang:adler32_combine/3 instead.
close(Z) -> ok
Types:
   Z = zstream()
Closes the stream referenced by Z.

compress(Data) -> Compressed
Types:
   Data = iodata()
   Compressed = binary()
Compresses data with zlib headers and checksum.

crc32(Z) -> CRC
Types:
   Z = zstream()
   CRC = integer() >= 0
Gets the current calculated CRC checksum.

Warning:
This function is deprecated and will be removed in a future release. Use erlang:crc32/1 on the uncompressed data instead.

crc32(Z, Data) -> CRC
Types:
   Z = zstream()
   Data = iodata()
   CRC = integer() >= 0
Calculates the CRC checksum for Data.

Warning:
This function is deprecated and will be removed in a future release. Use erlang:crc32/1 instead.

crc32(Z, PrevCRC, Data) -> CRC
Types:
   Z = zstream()
   PrevCRC = integer() >= 0
   Data = iodata()
   CRC = integer() >= 0
Updates a running CRC checksum for Data. If Data is the empty binary or the empty iolist, this function returns the required initial value for the CRC.
Example:
Crc = lists:foldl(fun(Data, Crc0) ->
    zlib:crc32(Z, Crc0, Data),
   end, zlib:crc32(Z, <<>>, Datas))

Warning:
This function is deprecated and will be removed in a future release. Use erlang:crc32/2 instead.

crc32_combine(Z, CRC1, CRC2, Size2) -> CRC
Types:
    Z = zstream()
    CRC = CRC1 = CRC2 = Size2 = integer() >= 0

Combines two CRC checksums into one. For two binaries or iolists, Data1 and Data2 with sizes of Size1 and Size2, with CRC checksums CRC1 and CRC2.
This function returns the CRC checksum of [Data1,Data2], requiring only CRC1,CRC2, and Size2.

Warning:
This function is deprecated and will be removed in a future release. Use erlang:crc32_combine/3 instead.

deflate(Z, Data) -> Compressed
Types:
    Z = zstream()
    Data = iodata()
    Compressed = iolist()

Same as deflate(Z, Data, none).

deflate(Z, Data, Flush) -> Compressed
Types:
    Z = zstream()
    Data = iodata()
    Flush = zflush()
    Compressed = iolist()

Compresses as much data as possible, and stops when the input buffer becomes empty. It can introduce some output latency (reading input without producing any output) except when forced to flush.
If Flush is set to sync, all pending output is flushed to the output buffer and the output is aligned on a byte boundary, so that the decompressor can get all input data available so far. Flushing can degrade compression for some compression algorithms; thus, use it only when necessary.
If Flush is set to full, all output is flushed as with sync, and the compression state is reset so that decompression can restart from this point if previous compressed data has been damaged or if random access is desired. Using full too often can seriously degrade the compression.
If Flush is set to finish, pending input is processed, pending output is flushed, and deflate/3 returns. Afterwards the only possible operations on the stream are deflateReset/1 or deflateEnd/1.
Flush can be set to finish immediately after deflateInit if all compression is to be done in one step.

Example:

```erlang
zlib:deflateInit(Z),
B1 = zlib:deflate(Z,Data),
B2 = zlib:deflate(Z,<< >>,finish),
zlib:deflateEnd(Z),
list_to_binary([B1,B2])
```

deflateEnd(Z) -> ok

Types:

```
Z = zstream()
```

Ends the deflate session and cleans all data used. Notice that this function throws a data_error exception if the last call to deflate/3 was not called with Flush set to finish.

deflateInit(Z) -> ok

Types:

```
Z = zstream()
```

Same as zlib:deflateInit(Z, default).

deflateInit(Z, Level) -> ok

Types:

```
Z = zstream()
Level = zlevel()
```

Initializes a zlib stream for compression.

Level decides the compression level to be used:

- default gives default compromise between speed and compression
- none (0) gives no compression
- best_speed (1) gives best speed
- best_compression (9) gives best compression

deflateInit(Z, Level, Method, WindowBits, MemLevel, Strategy) -> ok

Types:

```
Z = zstream()
Level = zlevel()
Method = zmethod()
WindowBits = zwindowbits()
MemLevel = zmemlevel()
Strategy = zstrategy()
```

Initiates a zlib stream for compression.

Level

- Compression level to use:
Method

Compression method to use, currently the only supported method is deflated.

WindowBits

The base two logarithm of the window size (the size of the history buffer). It is to be in the range 8 through 15. Larger values result in better compression at the expense of memory usage. Defaults to 15 if `deflateInit/2` is used. A negative `WindowBits` value suppresses the zlib header (and checksum) from the stream. Notice that the zlib source mentions this only as a undocumented feature.

**Warning:**
Due to a known bug in the underlying zlib library, `WindowBits` values 8 and -8 do not work as expected. In zlib versions before 1.2.9 values 8 and -8 are automatically changed to 9 and -9. **From zlib version 1.2.9 value -8 is rejected** causing `zlib:deflateInit/6` to fail (8 is still changed to 9). It also seem possible that future versions of zlib may fix this bug and start accepting 8 and -8 as is.

Conclusion: Avoid values 8 and -8 unless you know your zlib version supports them.

MemLevel

Specifies how much memory is to be allocated for the internal compression state: `MemLevel=1` uses minimum memory but is slow and reduces compression ratio; `MemLevel=9` uses maximum memory for optimal speed. Defaults to 8.

Strategy

Tunes the compression algorithm. Use the following values:

- default for normal data
- filtered for data produced by a filter (or predictor)
- huffman_only to force Huffman encoding only (no string match)
- rle to limit match distances to one (run-length encoding)

Filtered data consists mostly of small values with a somewhat random distribution. In this case, the compression algorithm is tuned to compress them better. The effect of filtered is to force more Huffman coding and less string matching: it is somewhat intermediate between default and huffman_only. rle is designed to be almost as fast as huffman_only, but gives better compression for PNG image data.

Strategy affects only the compression ratio, but not the correctness of the compressed output even if it is not set appropriately.

deflateParams(Z, Level, Strategy) -> ok

Types:

- `Z = zstream()`
- `Level = zlevel()`
- `Strategy = zstrategy()`

Dynamically updates the compression level and compression strategy. The interpretation of `Level` and `Strategy` is as in `deflateInit/6`. This can be used to switch between compression and straight copy of the input data, or to switch to a different kind of input data requiring a different strategy. If the compression level is changed, the input
available so far is compressed with the old level (and can be flushed); the new level takes effect only at the next call of deflate/3.

Before the call of deflateParams, the stream state must be set as for a call of deflate/3, as the currently available input may have to be compressed and flushed.

deflateReset(Z) -> ok
Types:
   Z = zstream()
Equivalent to deflateEnd/1 followed by deflateInit/1,2,6, but does not free and reallocate all the internal compression state. The stream keeps the same compression level and any other attributes.

deflateSetDictionary(Z, Dictionary) -> Adler32
Types:
   Z = zstream()
   Dictionary = iodata()
   Adler32 = integer() >= 0
Initializes the compression dictionary from the specified byte sequence without producing any compressed output.
This function must be called immediately after deflateInit/1,2,6 or deflateReset/1, before any call of deflate/3.
The compressor and decompressor must use the same dictionary (see inflateSetDictionary/2).
The Adler checksum of the dictionary is returned.

deflateReset(Z) -> ok
Types:
   Z = zstream()
Equivalent to deflateEnd/1 followed by deflateInit/1,2,6, but does not free and reallocate all the internal compression state. The stream keeps the same compression level and any other attributes.

deflateSetDictionary(Z, Dictionary) -> Adler32
Types:
   Z = zstream()
   Dictionary = iodata()
   Adler32 = integer() >= 0
Initializes the compression dictionary from the specified byte sequence without producing any compressed output.
This function must be called immediately after deflateInit/1,2,6 or deflateReset/1, before any call of deflate/3.
The compressor and decompressor must use the same dictionary (see inflateSetDictionary/2).
The Adler checksum of the dictionary is returned.

deflateReset(Z) -> ok
Types:
   Z = zstream()
Equivalent to deflateEnd/1 followed by deflateInit/1,2,6, but does not free and reallocate all the internal compression state. The stream keeps the same compression level and any other attributes.

deflateSetDictionary(Z, Dictionary) -> Adler32
Types:
   Z = zstream()
   Dictionary = iodata()
   Adler32 = integer() >= 0
Initializes the compression dictionary from the specified byte sequence without producing any compressed output.
This function must be called immediately after deflateInit/1,2,6 or deflateReset/1, before any call of deflate/3.
The compressor and decompressor must use the same dictionary (see inflateSetDictionary/2).
The Adler checksum of the dictionary is returned.

deflateReset(Z) -> ok
Types:
   Z = zstream()
Equivalent to deflateEnd/1 followed by deflateInit/1,2,6, but does not free and reallocate all the internal compression state. The stream keeps the same compression level and any other attributes.

deflateSetDictionary(Z, Dictionary) -> Adler32
Types:
   Z = zstream()
   Dictionary = iodata()
   Adler32 = integer() >= 0
Initializes the compression dictionary from the specified byte sequence without producing any compressed output.
This function must be called immediately after deflateInit/1,2,6 or deflateReset/1, before any call of deflate/3.
The compressor and decompressor must use the same dictionary (see inflateSetDictionary/2).
The Adler checksum of the dictionary is returned.

**Warning:**

This function is deprecated and will be removed in a future release.

gunzip(Data) -> Decompressed
Types:
   Data = iodata()
   Decompressed = binary()
Uncompresses data with gz headers and checksum.

gzip(Data) -> Compressed
Types:
   Data = iodata()
   Compressed = binary()
Compresses data with gz headers and checksum.
inflate(Z, Data) -> Decompressed
Types:
  Z = zstream()
  Data = iodata()
  Decompressed = iolist()
Equivalent to inflate(Z, Data, [])

inflate(Z, Data, Options) -> Decompressed
Types:
  Z = zstream()
  Data = iodata()
  Options = [{exception_on_need_dict, boolean()}]
  Decompressed = iolist() | {need_dictionary,
    Adler32 :: integer() >= 0,
    Output :: iolist()}
Decompresses as much data as possible. It can introduce some output latency (reading input without producing any output).
Currently the only available option is {exception_on_need_dict, boolean()} which controls whether the function should throw an exception when a preset dictionary is required for decompression. When set to false, a need_dictionary tuple will be returned instead. See inflateSetDictionary/2 for details.

Warning:
This option defaults to true for backwards compatibility but we intend to remove the exception behavior in a future release. New code that needs to handle dictionaries manually should always specify {exception_on_need_dict,false}.

inflateChunk(Z) -> Decompressed | {more, Decompressed}
Types:
  Z = zstream()
  Decompressed = iolist()

Warning:
This function is deprecated and will be removed in a future release. Use safeInflate/2 instead.
Reads the next chunk of uncompressed data, initialized by inflateChunk/2.
This function is to be repeatedly called, while it returns {more, Decompressed}.

inflateChunk(Z, Data) -> Decompressed | {more, Decompressed}
Types:
Z = zstream()
Data = iodata()
Decompressed = iolist()

Warning:
This function is deprecated and will be removed in a future release. Use safeInflate/2 instead.

Like inflate/2, but decompresses no more data than will fit in the buffer configured through setBufSize/2. It is useful when decompressing a stream with a high compression ratio, such that a small amount of compressed input can expand up to 1000 times.

This function returns {more, Decompressed}, when there is more output available, and inflateChunk/1 is to be used to read it.

This function can introduce some output latency (reading input without producing any output).
An exception will be thrown if a preset dictionary is required for further decompression. See inflateSetDictionary/2 for details.

Example:

walk(Compressed, Handler) ->
    Z = zlib:open(),
    zlib:inflateInit(Z),
    % Limit single uncompressed chunk size to 512kb
    zlib:setBufSize(Z, 512 * 1024),
    loop(Z, Handler, zlib:inflateChunk(Z, Compressed)),
    zlib:inflateEnd(Z),
    zlib:close(Z).

loop(Z, Handler, {more, Uncompressed}) ->
    Handler(Uncompressed),
    loop(Z, Handler, zlib:inflateChunk(Z));
loop(Z, Handler, Uncompressed) ->
    Handler(Uncompressed).

inflateEnd(Z) -> ok
Types:
    Z = zstream()

Ends the inflate session and cleans all data used. Notice that this function throws a data_error exception if no end of stream was found (meaning that not all data has been uncompressed).

inflateGetDictionary(Z) -> Dictionary
Types:
    Z = zstream()
    Dictionary = binary()

Returns the decompression dictionary currently in use by the stream. This function must be called between inflateInit/1,2 and inflateEnd.
Only supported if ERTS was compiled with zlib >= 1.2.8.
inflateInit(Z) -> ok
Types:
  Z = zstream()
Initializes a zlib stream for decompression.

inflateInit(Z, WindowBits) -> ok
Types:
  Z = zstream()
  WindowBits = zwindowbits()
Initializes a decompression session on zlib stream.

WindowBits is the base two logarithm of the maximum window size (the size of the history buffer). It is to be in the range 8 through 15. Default to 15 if inflateInit/1 is used.

If a compressed stream with a larger window size is specified as input, inflate/2 throws the data_error exception.

A negative WindowBits value makes zlib ignore the zlib header (and checksum) from the stream. Notice that the zlib source mentions this only as a undocumented feature.

inflateReset(Z) -> ok
Types:
  Z = zstream()
Equivalent to inflateEnd/1 followed by inflateInit/1, but does not free and reallocate all the internal decompression state. The stream will keep attributes that could have been set by inflateInit/1,2.

inflateSetDictionary(Z, Dictionary) -> ok
Types:
  Z = zstream()
  Dictionary = iodata()
Initializes the decompression dictionary from the specified uncompressed byte sequence. This function must be called as a response to an inflate operation (eg. safeInflate/2) returning {need_dictionary,Adler,Output} or in the case of deprecated functions, throwing an {'EXIT', {{need_dictionary,Adler},_StackTrace}} exception.

The dictionary chosen by the compressor can be determined from the Adler value returned or thrown by the call to the inflate function. The compressor and decompressor must use the same dictionary (See deflateSetDictionary/2).

After setting the dictionary the inflate operation should be retried without new input.

Example:
deprecated_unpack(Z, Compressed, Dict) ->
    case catch zlib:inflate(Z, Compressed) of
        {'EXIT',{{need_dictionary, _DictID},_}} ->
            ok = zlib:inflateSetDictionary(Z, Dict),
            Uncompressed = zlib:inflate(Z, []);
        Uncompressed ->
            Uncompressed
    end.

new_unpack(Z, Compressed, Dict) ->
    case zlib:inflate(Z, Compressed, [{exception_on_need_dict, false}]) of
        {need_dictionary, _DictId, Output} ->
            ok = zlib:inflateSetDictionary(Z, Dict),
            [Output | zlib:inflate(Z, [])];
        Uncompressed ->
            Uncompressed
    end.

open() -> zstream()
Opens a zlib stream.

safeInflate(Z, Data) -> Result
Types:
  Z = zstream()
  Data = iodata()
  Result =
    {continue, Output :: iolist()} |
    {finished, Output :: iolist()} |
    {need_dictionary,
      Adler32 :: integer() >= 0,
      Output :: iolist()}
Like inflate/2, but returns once it has expanded beyond a small implementation-defined threshold. It's useful when decompressing untrusted input which could have been maliciously crafted to expand until the system runs out of memory.

This function returns \{continue | finished, Output\}, where Output is the data that was decompressed in this call. New input can be queued up on each call if desired, and the function will return \{finished, Output\} once all queued data has been decompressed.

This function can introduce some output latency (reading input without producing any output).

If a preset dictionary is required for further decompression, this function returns a \{need_dictionary\} tuple. See inflateSetDictionary/2) for details.

Example:
walk(Compressed, Handler) ->
    Z = zlib:open(),
    zlib:inflateInit(Z),
    loop(Z, Handler, zlib:safeInflate(Z, Compressed)),
    zlib:inflateEnd(Z),
    zlib:close(Z).
loop(Z, Handler, {continue, Output}) ->
    Handler(Output),
    loop(Z, Handler, zlib:safeInflate(Z, []));
loop(Z, Handler, {finished, Output}) ->
    Handler(Output).

setBufSize(Z, Size) -> ok
Types:
    Z = zstream()
    Size = integer() >= 0
Sets the intermediate buffer size.

Warning:
This function is deprecated and will be removed in a future release.

set_controlling_process(Z, Pid) -> ok
Types:
    Z = zstream()
    Pid = pid()
Changes the controlling process of Z to Pid, which must be a local process.

uncompress(Data) -> Decompressed
Types:
    Data = iodata()
    Decompressed = binary()
Uncompresses data with zlib headers and checksum.

unzip(Data) -> Decompressed
Types:
    Data = iodata()
    Decompressed = binary()
Uncompresses data without zlib headers and checksum.

zip(Data) -> Compressed
Types:
    Data = iodata()
    Compressed = binary()
Compresses data without zlib headers and checksum.