


# Erlang Interface

Copyright © 1998-2017 Ericsson AB. All Rights Reserved.
Erlang Interface 3.8.2
December 1, 2017

**Copyright © 1998-2017 Ericsson AB. All Rights Reserved.**
Licensed under the Apache License, Version 2.0 (the "License"); you may not use this file except in compliance with the License. You may obtain a copy of the License at http://www.apache.org/licenses/LICENSE-2.0 Unless required by applicable law or agreed to in writing, software distributed under the License is distributed on an "AS IS" BASIS, WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied. See the License for the specific language governing permissions and limitations under the License. Ericsson AB. All Rights Reserved..

**December 1, 2017**

# 1 EI User's Guide

## 1.1 The EI Library User's Guide

The Erl_Interface library contains functions. which help you integrate programs written in C and Erlang. The functions in Erl_Interface support the following:

- manipulation of data represented as Erlang data types
- conversion of data between C and Erlang formats
- encoding and decoding of Erlang data types for transmission or storage
- communication between C nodes and Erlang processes
- backup and restore of C node state to and from Mnesia

In the following sections, these topics are described:

- compiling your code for use with Erl_Interface
- initializing Erl_Interface
- encoding, decoding, and sending Erlang terms
- building terms and patterns
- pattern matching
- connecting to a distributed Erlang node
- using EPMD
- sending and receiving Erlang messages
- remote procedure calls
- global names
- the registry

### 1.1.1 Compiling and Linking Your Code

In order to use any of the Erl_Interface functions, include the following lines in your code:

```
#include "erl_interface.h"
#include "ei.h"
```

Determine where the top directory of your OTP installation is. You can find this out by starting Erlang and entering the following command at the Eshell prompt:

```
Eshell V4.7.4  (abort with ^G)
1> code:root_dir().
/usr/local/otp
```

To compile your code, make sure that your C compiler knows where to find `erl_interface.h` by specifying an appropriate `-I` argument on the command line, or by adding it to the `CFLAGS` definition in your `Makefile`. The correct value for this path is `$OTPROOT/lib/erl_interface`*Vsn*`/include`, where `$OTPROOT` is the path

reported by `code:root_dir/0` in the above example, and *Vsn* is the version of the Erl_interface application, for example `erl_interface-3.2.3`

```
$ cc -c -I/usr/local/otp/lib/erl_interface-3.2.3/include myprog.c
```

When linking, you will need to specify the path to `liberl_interface.a` and `libei.a` with `-L $OTPROOT/lib/erl_interface-3.2.3/lib`, and you will need to specify the name of the libraries with `-lerl_interface -lei`. You can do this on the command line or by adding the flags to the `LDFLAGS` definition in your `Makefile`.

```
$ ld -L/usr/local/otp/lib/erl_interface-3.2.3/
                           lib myprog.o -lerl_interface -lei -o myprog
```

Also, on some systems it may be necessary to link with some additional libraries (e.g. `libnsl.a` and `libsocket.a` on Solaris, or `wsock32.lib` on Windows) in order to use the communication facilities of Erl_Interface.

If you are using Erl_Interface functions in a threaded application based on POSIX threads or Solaris threads, then Erl_Interface needs access to some of the synchronization facilities in your threads package, and you will need to specify additional compiler flags in order to indicate which of the packages you are using. Define `_REENTRANT` and either `STHREADS` or `PTHREADS`. The default is to use POSIX threads if `_REENTRANT` is specified.

### 1.1.2 Initializing the erl_interface Library

Before calling any of the other Erl_Interface functions, you must call `erl_init()` exactly once to initialize the library. `erl_init()` takes two arguments, however the arguments are no longer used by Erl_Interface, and should therefore be specified as `erl_init(NULL,0)`.

### 1.1.3 Encoding, Decoding and Sending Erlang Terms

Data sent between distributed Erlang nodes is encoded in the Erlang external format. Consequently, you have to encode and decode Erlang terms into byte streams if you want to use the distribution protocol to communicate between a C program and Erlang.

The Erl_Interface library supports this activity. It has a number of C functions which create and manipulate Erlang data structures. The library also contains an encode and a decode function. The example below shows how to create and encode an Erlang tuple `{tobbe,3928}`:

```
ETERM *arr[2], *tuple;
char buf[BUFSIZ];
int i;

arr[0] = erl_mk_atom("tobbe");
arr[1] = erl_mk_integer(3928);
tuple  = erl_mk_tuple(arr, 2);
i = erl_encode(tuple, buf);
```

Alternatively, you can use `erl_send()` and `erl_receive_msg`, which handle the encoding and decoding of messages transparently.

Refer to the Reference Manual for a complete description of the following modules:

- the `erl_eterm` module for creating Erlang terms

- the `erl_marshal` module for encoding and decoding routines.

### 1.1.4 Building Terms and Patterns

The previous example can be simplified by using `erl_format()` to create an Erlang term.

```
ETERM *ep;
ep = erl_format("{~a,~i}", "tobbe", 3928);
```

Refer to the Reference Manual, the `erl_format` module, for a full description of the different format directives. The following example is more complex:

```
ETERM *ep;
ep = erl_format("[{name,~a},{age,~i},{data,~w}]",
                "madonna",
                21,
                erl_format("[{adr,~s,~i}]", "E-street", 42));
erl_free_compound(ep);
```

As in previous examples, it is your responsibility to free the memory allocated for Erlang terms. In this example, `erl_free_compound()` ensures that the complete term pointed to by `ep` is released. This is necessary, because the pointer from the second call to `erl_format()` is lost.

The following example shows a slightly different solution:

```
ETERM *ep,*ep2;
ep2 = erl_format("[{adr,~s,~i}]","E-street",42);
ep  = erl_format("[{name,~a},{age,~i},{data,~w}]",
                "madonna", 21, ep2);
erl_free_term(ep);
erl_free_term(ep2);
```

In this case, you free the two terms independently. The order in which you free the terms `ep` and `ep2` is not important, because the Erl_Interface library uses reference counting to determine when it is safe to actually remove objects.

If you are not sure whether you have freed the terms properly, you can use the following function to see the status of the fixed term allocator:

```
long allocated, freed;

erl_eterm_statistics(&allocated,&freed);
printf("currently allocated blocks: %ld\n",allocated);
printf("length of freelist: %ld\n",freed);

/* really free the freelist */
erl_eterm_release();
```

Refer to the Reference Manual, the `erl_malloc` module for more information.

### 1.1.5 Pattern Matching

An Erlang pattern is a term that may contain unbound variables or `"do not care"` symbols. Such a pattern can be matched against a term and, if the match is successful, any unbound variables in the pattern will be bound as a side effect. The content of a bound variable can then be retrieved.

```
ETERM *pattern;
pattern = erl_format("{madonna,Age,_}");
```

`erl_match()` is used to perform pattern matching. It takes a pattern and a term and tries to match them. As a side effect any unbound variables in the pattern will be bound. In the following example, we create a pattern with a variable *Age* which appears at two positions in the tuple. The pattern match is performed as follows:

- `erl_match()` will bind the contents of *Age* to *21* the first time it reaches the variable
- the second occurrence of *Age* will cause a test for equality between the terms since *Age* is already bound to *21*. Since *Age* is bound to 21, the equality test will succeed and the match continues until the end of the pattern.
- if the end of the pattern is reached, the match succeeds and you can retrieve the contents of the variable

```
ETERM *pattern,*term;
pattern = erl_format("{madonna,Age,Age}");
term    = erl_format("{madonna,21,21}");
if (erl_match(pattern, term)) {
  fprintf(stderr, "Yes, they matched: Age = ");
  ep = erl_var_content(pattern, "Age");
  erl_print_term(stderr, ep);
  fprintf(stderr,"\n");
  erl_free_term(ep);
}
erl_free_term(pattern);
erl_free_term(term);
```

Refer to the Reference Manual, the `erl_match()` function for more information.

### 1.1.6 Connecting to a Distributed Erlang Node

In order to connect to a distributed Erlang node you need to first initialize the connection routine with `erl_connect_init()`, which stores information such as the host name, node name, and IP address for later use:

```
int identification_number = 99;
int creation=1;
char *cookie="a secret cookie string"; /* An example */
erl_connect_init(identification_number, cookie, creation);
```

Refer to the Reference Manual, the `erl_connect` module for more information.

After initialization, you set up the connection to the Erlang node. Use `erl_connect()` to specify the Erlang node you want to connect to. The following example sets up the connection and should result in a valid socket file descriptor:

```
int sockfd;
char *nodename="xyz@chivas.du.etx.ericsson.se"; /* An example */
if ((sockfd = erl_connect(nodename)) < 0)
```

```
  erl_err_quit("ERROR: erl_connect failed");
```

`erl_err_quit()` prints the specified string and terminates the program. Refer to the Reference Manual, the `erl_error()` function for more information.

### 1.1.7 Using EPMD

`Epmd` is the Erlang Port Mapper Daemon. Distributed Erlang nodes register with `epmd` on the localhost to indicate to other nodes that they exist and can accept connections. `Epmd` maintains a register of node and port number information, and when a node wishes to connect to another node, it first contacts `epmd` in order to find out the correct port number to connect to.

When you use `erl_connect()` to connect to an Erlang node, a connection is first made to `epmd` and, if the node is known, a connection is then made to the Erlang node.

C nodes can also register themselves with `epmd` if they want other nodes in the system to be able to find and connect to them.

Before registering with `epmd`, you need to first create a listen socket and bind it to a port. Then:

```
int pub;

pub = erl_publish(port);
```

`pub` is a file descriptor now connected to `epmd`. `Epmd` monitors the other end of the connection, and if it detects that the connection has been closed, the node will be unregistered. So, if you explicitly close the descriptor or if your node fails, it will be unregistered from `epmd`.

Be aware that on some systems (such as VxWorks), a failed node will not be detected by this mechanism since the operating system does not automatically close descriptors that were left open when the node failed. If a node has failed in this way, `epmd` will prevent you from registering a new node with the old name, since it thinks that the old name is still in use. In this case, you must unregister the name explicitly:

```
erl_unpublish(node);
```

This will cause `epmd` to close the connection from the far end. Note that if the name was in fact still in use by a node, the results of this operation are unpredictable. Also, doing this does not cause the local end of the connection to close, so resources may be consumed.

### 1.1.8 Sending and Receiving Erlang Messages

Use one of the following two functions to send messages:

- `erl_send()`
- `erl_reg_send()`

As in Erlang, it is possible to send messages to a *Pid* or to a registered name. It is easier to send a message to a registered name because it avoids the problem of finding a suitable *Pid*.

Use one of the following two functions to receive messages:

- `erl_receive()`
- `erl_receive_msg()`

`erl_receive()` receives the message into a buffer, while `erl_receive_msg()` decodes the message into an Erlang term.

## Example of Sending Messages

In the following example, `{Pid, hello_world}` is sent to a registered process `my_server`. The message is encoded by `erl_send()`:

```
extern const char *erl_thisnodename(void);
extern short erl_thiscreation(void);
#define SELF(fd) erl_mk_pid(erl_thisnodename(),fd,0,erl_thiscreation())
ETERM *arr[2], *emsg;
int sockfd, creation=1;

arr[0] = SELF(sockfd);
arr[1] = erl_mk_atom("Hello world");
emsg   = erl_mk_tuple(arr, 2);

erl_reg_send(sockfd, "my_server", emsg);
erl_free_term(emsg);
```

The first element of the tuple that is sent is your own *Pid*. This enables `my_server` to reply. Refer to the Reference Manual, the `erl_connect` module for more information about send primitives.

## Example of Receiving Messages

In this example `{Pid, Something}` is received. The received Pid is then used to return `{goodbye,Pid}`

```
ETERM *arr[2], *answer;
int sockfd,rc;
char buf[BUFSIZE];
ErlMessage emsg;

if ((rc = erl_receive_msg(sockfd , buf, BUFSIZE, &emsg)) == ERL_MSG) {
   arr[0] = erl_mk_atom("goodbye");
   arr[1] = erl_element(1, emsg.msg);
   answer = erl_mk_tuple(arr, 2);
   erl_send(sockfd, arr[1], answer);
   erl_free_term(answer);
   erl_free_term(emsg.msg);
   erl_free_term(emsg.to);
}
```

In order to provide robustness, a distributed Erlang node occasionally polls all its connected neighbours in an attempt to detect failed nodes or communication links. A node which receives such a message is expected to respond immediately with an `ERL_TICK` message. This is done automatically by `erl_receive()`, however when this has occurred `erl_receive` returns `ERL_TICK` to the caller without storing a message into the `ErlMessage` structure.

When a message has been received, it is the caller's responsibility to free the received message `emsg.msg` as well as `emsg.to` or `emsg.from`, depending on the type of message received.

Refer to the Reference Manual for additional information about the following modules:

- `erl_connect`
- `erl_eterm`.

### 1.1.9 Remote Procedure Calls

An Erlang node acting as a client to another Erlang node typically sends a request and waits for a reply. Such a request is included in a function call at a remote node and is called a remote procedure call. The following example shows how the Erl_Interface library supports remote procedure calls:

```
char modname[]=THE_MODNAME;
ETERM *reply,*ep;
ep = erl_format("[~a,[]]", modname);
if (!(reply = erl_rpc(fd, "c", "c", ep)))
  erl_err_msg("<ERROR> when compiling file: %s.erl !\n", modname);
erl_free_term(ep);
ep = erl_format("{ok,_}");
if (!erl_match(ep, reply))
  erl_err_msg("<ERROR> compiler errors !\n");
erl_free_term(ep);
erl_free_term(reply);
```

`c:c/1` is called to compile the specified module on the remote node. `erl_match()` checks that the compilation was successful by testing for the expected `ok`.

Refer to the Reference Manual, the `erl_connect` module for more information about `erl_rpc()`, and its companions `erl_rpc_to()` and `erl_rpc_from()`.

### 1.1.10 Using Global Names

A C node has access to names registered through the Erlang Global module. Names can be looked up, allowing the C node to send messages to named Erlang services. C nodes can also register global names, allowing them to provide named services to Erlang processes or other C nodes.

Erl_Interface does not provide a native implementation of the global service. Instead it uses the global services provided by a "nearby" Erlang node. In order to use the services described in this section, it is necessary to first open a connection to an Erlang node.

To see what names there are:

```
char **names;
int count;
int i;

names = erl_global_names(fd,&count);

if (names)
  for (i=0; i<count; i++)
    printf("%s\n",names[i]);

free(names);
```

`erl_global_names()` allocates and returns a buffer containing all the names known to global. `count` will be initialized to indicate how many names are in the array. The array of strings in names is terminated by a NULL pointer, so it is not necessary to use `count` to determine when the last name is reached.

It is the caller's responsibility to free the array. `erl_global_names()` allocates the array and all of the strings using a single call to `malloc()`, so `free(names)` is all that is necessary.

To look up one of the names:

```
ETERM *pid;
char node[256];

pid = erl_global_whereis(fd,"schedule",node);
```

If `"schedule"` is known to global, an Erlang pid is returned that can be used to send messages to the schedule service. Additionally, `node` will be initialized to contain the name of the node where the service is registered, so that you can make a connection to it by simply passing the variable to `erl_connect()`.

Before registering a name, you should already have registered your port number with `epmd`. This is not strictly necessary, but if you neglect to do so, then other nodes wishing to communicate with your service will be unable to find or connect to your process.

Create a pid that Erlang processes can use to communicate with your service:

```
ETERM *pid;

pid = erl_mk_pid(thisnode,14,0,0);
erl_global_register(fd,servicename,pid);
```

After registering the name, you should use `erl_accept()` to wait for incoming connections.

Do not forget to free `pid` later with `erl_free_term()`!

To unregister a name:

```
erl_global_unregister(fd,servicename);
```

### 1.1.11 The Registry

This section describes the use of the registry, a simple mechanism for storing key-value pairs in a C-node, as well as backing them up or restoring them from a Mnesia table on an Erlang node. More detailed information about the individual API functions can be found in the Reference Manual.

Keys are strings, i.e. 0-terminated arrays of characters, and values are arbitrary objects. Although integers and floating point numbers are treated specially by the registry, you can store strings or binary objects of any type as pointers.

To start, you need to open a registry:

```
ei_reg *reg;

reg = ei_reg_open(45);
```

The number 45 in the example indicates the approximate number of objects that you expect to store in the registry. Internally the registry uses hash tables with collision chaining, so there is no absolute upper limit on the number of objects that the registry can contain, but if performance or memory usage are important, then you should choose a number accordingly. The registry can be resized later.

You can open as many registries as you like (if memory permits).

Objects are stored and retrieved through set and get functions. In the following examples you see how to store integers, floats, strings and arbitrary binary objects:

```
struct bonk *b = malloc(sizeof(*b));
char *name = malloc(7);

ei_reg_setival(reg,"age",29);
ei_reg_setfval(reg,"height",1.85);

strcpy(name,"Martin");
ei_reg_setsval(reg,"name",name);

b->l = 42;
b->m = 12;
ei_reg_setpval(reg,"jox",b,sizeof(*b));
```

If you attempt to store an object in the registry and there is an existing object with the same key, the new value will replace the old one. This is done regardless of whether the new object and the old one have the same type, so you can, for example, replace a string with an integer. If the existing value is a string or binary, it will be freed before the new value is assigned.

Stored values are retrieved from the registry as follows:

```
long i;
double f;
char *s;
struct bonk *b;
int size;

i = ei_reg_getival(reg,"age");
f = ei_reg_getfval(reg,"height");
s = ei_reg_getsval(reg,"name");
b = ei_reg_getpval(reg,"jox",&size);
```

In all of the above examples, the object must exist and it must be of the right type for the specified operation. If you do not know the type of a given object, you can ask:

```
struct ei_reg_stat buf;

ei_reg_stat(reg,"name",&buf);
```

Buf will be initialized to contain object attributes.

Objects can be removed from the registry:

```
ei_reg_delete(reg,"name");
```

When you are finished with a registry, close it to remove all the objects and free the memory back to the system:

```
ei_reg_close(reg);
```

## Backing Up the Registry to Mnesia

The contents of a registry can be backed up to Mnesia on a "nearby" Erlang node. You need to provide an open connection to the Erlang node (see `erl_connect()`). Also, Mnesia 3.0 or later must be running on the Erlang node before the backup is initiated:

```
ei_reg_dump(fd, reg, "mtab", dumpflags);
```

The example above will backup the contents of the registry to the specified Mnesia table `"mtab"`. Once a registry has been backed up to Mnesia in this manner, additional backups will only affect objects that have been modified since the most recent backup, i.e. objects that have been created, changed or deleted. The backup operation is done as a single atomic transaction, so that the entire backup will be performed or none of it will.

In the same manner, a registry can be restored from a Mnesia table:

```
ei_reg_restore(fd, reg, "mtab");
```

This will read the entire contents of `"mtab"` into the specified registry. After the restore, all of the objects in the registry will be marked as unmodified, so a subsequent backup will only affect objects that you have modified since the restore.

Note that if you restore to a non-empty registry, objects in the table will overwrite objects in the registry with the same keys. Also, the *entire* contents of the registry is marked as unmodified after the restore, including any modified objects that were not overwritten by the restore operation. This may not be your intention.

## Storing Strings and Binaries

When string or binary objects are stored in the registry it is important that a number of simple guidelines are followed.

Most importantly, the object must have been created with a single call to `malloc()` (or similar), so that it can later be removed by a single call to `free()`. Objects will be freed by the registry when it is closed, or when you assign a new value to an object that previously contained a string or binary.

You should also be aware that if you store binary objects that are context-dependent (e.g. containing pointers or open file descriptors), they will lose their meaning if they are backed up to a Mnesia table and subsequently restored in a different context.

When you retrieve a stored string or binary value from the registry, the registry maintains a pointer to the object and you are passed a copy of that pointer. You should never free an object retrieved in this manner because when the registry later attempts to free it, a runtime error will occur that will likely cause the C-node to crash.

You are free to modify the contents of an object retrieved this way. However when you do so, the registry will not be aware of the changes you make, possibly causing it to be missed the next time you make a Mnesia backup of the registry contents. This can be avoided if you mark the object as dirty after any such changes with `ei_reg_markdirty()`, or pass appropriate flags to `ei_reg_dump()`.

# 2 Reference Manual

The `ei` and `erl_interface` are `C` interface libraries for communication with `Erlang`.

> Note:
>
> By default, the `ei` and `erl_interface` libraries are only guaranteed to be compatible with other Erlang/OTP components from the same release as the libraries themself. See the documentation of the *ei_set_compat_rel()* and *erl_set_compat_rel()* functions on how to communicate with Erlang/OTP components from earlier releases.

# ei

C Library

The library `ei` contains macros and functions to encode and decode the erlang binary term format.

With `ei`, you can convert atoms, lists, numbers and binaries to and from the binary format. This is useful when writing port programs and drivers. `ei` uses a given buffer, and no dynamic memory (with the exception of `ei_decode_fun()`), and is often quite fast.

It also handles C-nodes, C-programs that talks erlang distribution with erlang nodes (or other C-nodes) using the erlang distribution format. The difference between `ei` and `erl_interface` is that `ei` uses the binary format directly when sending and receiving terms. It is also thread safe, and using threads, one process can handle multiple C-nodes. The `erl_interface` library is built on top of `ei`, but of legacy reasons, it doesn't allow for multiple C-nodes. In general, `ei` is the preferred way of doing C-nodes.

The decode and encode functions use a buffer an index into the buffer, which points at the point where to encode and decode. The index is updated to point right after the term encoded/decoded. No checking is done whether the term fits in the buffer or not. If encoding goes outside the buffer, the program may crash.

All functions takes two parameter, `buf` is a pointer to the buffer where the binary data is / will be, `index` is a pointer to an index into the buffer. This parameter will be incremented with the size of the term decoded / encoded. The data is thus at `buf[*index]` when an `ei` function is called.

The encode functions all assumes that the `buf` and `index` parameters points to a buffer big enough for the data. To get the size of an encoded term, without encoding it, pass `NULL` instead of a buffer pointer. The `index` parameter will be incremented, but nothing will be encoded. This is the way in `ei` to "preflight" term encoding.

There are also encode-functions that uses a dynamic buffer. It is often more convenient to use these to encode data. All encode functions comes in two versions: those starting with `ei_x`, uses a dynamic buffer.

All functions return `0` if successful, and `-1` if not. (For instance, if a term is not of the expected type, or the data to decode is not a valid erlang term.)

Some of the decode-functions needs a preallocated buffer. This buffer must be allocated big enough, and for non compound types the `ei_get_type()` function returns the size required (note that for strings an extra byte is needed for the 0 string terminator).

## DATA TYPES

erlang_char_encoding

```
typedef enum {
    ERLANG_ASCII = 1,
    ERLANG_LATIN1 = 2,
    ERLANG_UTF8 = 4
}erlang_char_encoding;
```

The character encodings used for atoms. `ERLANG_ASCII` represents 7-bit ASCII. Latin1 and UTF8 are different extensions of 7-bit ASCII. All 7-bit ASCII characters are valid Latin1 and UTF8 characters. ASCII and Latin1 both represent each character by one byte. A UTF8 character can consist of one to four bytes. Note that these constants are bit-flags and can be combined with bitwise-or.

## Exports

**`void ei_set_compat_rel(release_number)`**

Types:

```
unsigned release_number;
```

By default, the `ei` library is only guaranteed to be compatible with other Erlang/OTP components from the same release as the `ei` library itself. For example, `ei` from the OTP R10 release is not compatible with an Erlang emulator from the OTP R9 release by default.

A call to `ei_set_compat_rel(release_number)` sets the `ei` library in compatibility mode of release `release_number`. Valid range of `release_number` is [7, current release]. This makes it possible to communicate with Erlang/OTP components from earlier releases.

> ### Note:
>
> If this function is called, it may only be called once and must be called before any other functions in the `ei` library is called.

> ### Warning:
>
> You may run into trouble if this feature is used carelessly. Always make sure that all communicating components are either from the same Erlang/OTP release, or from release X and release Y where all components from release Y are in compatibility mode of release X.

**`int ei_encode_version(char *buf, int *index)`**
**`int ei_x_encode_version(ei_x_buff* x)`**

Encodes a version magic number for the binary format. Must be the first token in a binary term.

**`int ei_encode_long(char *buf, int *index, long p)`**
**`int ei_x_encode_long(ei_x_buff* x, long p)`**

Encodes a long integer in the binary format. Note that if the code is 64 bits the function ei_encode_long() is exactly the same as ei_encode_longlong().

**`int ei_encode_ulong(char *buf, int *index, unsigned long p)`**
**`int ei_x_encode_ulong(ei_x_buff* x, unsigned long p)`**

Encodes an unsigned long integer in the binary format. Note that if the code is 64 bits the function ei_encode_ulong() is exactly the same as ei_encode_ulonglong().

**`int ei_encode_longlong(char *buf, int *index, long long p)`**
**`int ei_x_encode_longlong(ei_x_buff* x, long long p)`**

Encodes a GCC `long long` or Visual C++ `__int64` (64 bit) integer in the binary format. Note that this function is missing in the VxWorks port.

**int ei_encode_ulonglong(char *buf, int *index, unsigned long long p)**
**int ei_x_encode_ulonglong(ei_x_buff* x, unsigned long long p)**

Encodes a GCC `unsigned long long` or Visual C++ `unsigned __int64` (64 bit) integer in the binary format. Note that this function is missing in the VxWorks port.

**int ei_encode_bignum(char *buf, int *index, mpz_t obj)**
**int ei_x_encode_bignum(ei_x_buff *x, mpz_t obj)**

Encodes a GMP `mpz_t` integer to binary format. To use this function the ei library needs to be configured and compiled to use the GMP library.

**int ei_encode_double(char *buf, int *index, double p)**
**int ei_x_encode_double(ei_x_buff* x, double p)**

Encodes a double-precision (64 bit) floating point number in the binary format.

The function returns `-1` if the floating point number is not finite.

**int ei_encode_boolean(char *buf, int *index, int p)**
**int ei_x_encode_boolean(ei_x_buff* x, int p)**

Encodes a boolean value, as the atom `true` if p is not zero or `false` if p is zero.

**int ei_encode_char(char *buf, int *index, char p)**
**int ei_x_encode_char(ei_x_buff* x, char p)**

Encodes a char (8-bit) as an integer between 0-255 in the binary format. Note that for historical reasons the integer argument is of type `char`. Your C code should consider the given argument to be of type `unsigned char` even if the C compilers and system may define `char` to be signed.

**int ei_encode_string(char *buf, int *index, const char *p)**
**int ei_encode_string_len(char *buf, int *index, const char *p, int len)**
**int ei_x_encode_string(ei_x_buff* x, const char *p)**
**int ei_x_encode_string_len(ei_x_buff* x, const char* s, int len)**

Encodes a string in the binary format. (A string in erlang is a list, but is encoded as a character array in the binary format.) The string should be zero-terminated, except for the `ei_x_encode_string_len()` function.

**int ei_encode_atom(char *buf, int *index, const char *p)**
**int ei_encode_atom_len(char *buf, int *index, const char *p, int len)**
**int ei_x_encode_atom(ei_x_buff* x, const char *p)**
**int ei_x_encode_atom_len(ei_x_buff* x, const char *p, int len)**

Encodes an atom in the binary format. The `p` parameter is the name of the atom in latin1 encoding. Only upto `MAXATOMLEN-1` bytes are encoded. The name should be zero-terminated, except for the `ei_x_encode_atom_len()` function.

**int ei_encode_atom_as(char *buf, int *index, const char *p, erlang_char_encoding from_enc, erlang_char_encoding to_enc)**

**int ei_encode_atom_len_as(char *buf, int *index, const char *p, int len, erlang_char_encoding from_enc, erlang_char_encoding to_enc)**

**int ei_x_encode_atom_as(ei_x_buff* x, const char *p, erlang_char_encoding from_enc, erlang_char_encoding to_enc)**

**int ei_x_encode_atom_len_as(ei_x_buff* x, const char *p, int len, erlang_char_encoding from_enc, erlang_char_encoding to_enc)**

Encodes an atom in the binary format with character encoding *to_enc* (latin1 or utf8). The p parameter is the name of the atom with character encoding *from_enc* (ascii, latin1 or utf8). The name must either be zero-terminated or a function variant with a len parameter must be used. If to_enc is set to the bitwise-or'd combination (ERLANG_LATIN1|ERLANG_UTF8), utf8 encoding is only used if the atom string can not be represented in latin1 encoding.

The encoding will fail if p is not a valid string in encoding from_enc, if the string is too long or if it can not be represented with character encoding to_enc.

These functions were introduced in R16 release of Erlang/OTP as part of a first step to support UTF8 atoms. Atoms encoded with ERLANG_UTF8 can not be decoded by earlier releases than R16.

**int ei_encode_binary(char *buf, int *index, const void *p, long len)**

**int ei_x_encode_binary(ei_x_buff* x, const void *p, long len)**

Encodes a binary in the binary format. The data is at p, of len bytes length.

**int ei_encode_pid(char *buf, int *index, const erlang_pid *p)**

**int ei_x_encode_pid(ei_x_buff* x, const erlang_pid *p)**

Encodes an erlang process identifier, pid, in the binary format. The p parameter points to an erlang_pid structure (which should have been obtained earlier with ei_decode_pid()).

**int ei_encode_fun(char *buf, int *index, const erlang_fun *p)**

**int ei_x_encode_fun(ei_x_buff* x, const erlang_fun* fun)**

Encodes a fun in the binary format. The p parameter points to an erlang_fun structure. The erlang_fun is not freed automatically, the free_fun should be called if the fun is not needed after encoding.

**int ei_encode_port(char *buf, int *index, const erlang_port *p)**

**int ei_x_encode_port(ei_x_buff* x, const erlang_port *p)**

Encodes an erlang port in the binary format. The p parameter points to a erlang_port structure (which should have been obtained earlier with ei_decode_port().

**int ei_encode_ref(char *buf, int *index, const erlang_ref *p)**

**int ei_x_encode_ref(ei_x_buff* x, const erlang_ref *p)**

Encodes an erlang reference in the binary format. The p parameter points to a erlang_ref structure (which should have been obtained earlier with ei_decode_ref().

**int ei_encode_term(char *buf, int *index, void *t)**
**int ei_x_encode_term(ei_x_buff* x, void *t)**

This function encodes an ETERM, as obtained from erl_interface. The t parameter is actually an ETERM pointer. This function doesn't free the ETERM.

**int ei_encode_trace(char *buf, int *index, const erlang_trace *p)**
**int ei_x_encode_trace(ei_x_buff* x, const erlang_trace *p)**

This function encodes an erlang trace token in the binary format. The p parameter points to a erlang_trace structure (which should have been obtained earlier with ei_decode_trace().

**int ei_encode_tuple_header(char *buf, int *index, int arity)**
**int ei_x_encode_tuple_header(ei_x_buff* x, int arity)**

This function encodes a tuple header, with a specified arity. The next arity terms encoded will be the elements of the tuple. Tuples and lists are encoded recursively, so that a tuple may contain another tuple or list.

E.g. to encode the tuple {a, {b, {}}}:

```
ei_encode_tuple_header(buf, &i, 2);
ei_encode_atom(buf, &i, "a");
ei_encode_tuple_header(buf, &i, 2);
ei_encode_atom(buf, &i, "b");
ei_encode_tuple_header(buf, &i, 0);
```

**int ei_encode_list_header(char *buf, int *index, int arity)**
**int ei_x_encode_list_header(ei_x_buff* x, int arity)**

This function encodes a list header, with a specified arity. The next arity+1 terms are the elements (actually its arity cons cells) and the tail of the list. Lists and tuples are encoded recursively, so that a list may contain another list or tuple.

E.g. to encode the list [c, d, [e | f]]:

```
ei_encode_list_header(buf, &i, 3);
ei_encode_atom(buf, &i, "c");
ei_encode_atom(buf, &i, "d");
ei_encode_list_header(buf, &i, 1);
ei_encode_atom(buf, &i, "e");
ei_encode_atom(buf, &i, "f");
ei_encode_empty_list(buf, &i);
```

> **Note:**
>
> It may seem that there is no way to create a list without knowing the number of elements in advance. But indeed there is a way. Note that the list [a, b, c] can be written as [a | [b | [c]]]. Using this, a list can be written as conses.

To encode a list, without knowing the arity in advance:

```
while (something()) {
    ei_x_encode_list_header(&x, 1);
    ei_x_encode_ulong(&x, i); /* just an example */
}
ei_x_encode_empty_list(&x);
```

**`int ei_encode_empty_list(char* buf, int* index)`**
**`int ei_x_encode_empty_list(ei_x_buff* x)`**

This function encodes an empty list. It's often used at the tail of a list.

**`int ei_encode_map_header(char *buf, int *index, int arity)`**
**`int ei_x_encode_map_header(ei_x_buff* x, int arity)`**

This function encodes a map header, with a specified arity. The next `arity*2` terms encoded will be the keys and values of the map encoded in the following order: `K1, V1, K2, V2, ..., Kn, Vn`.

E.g. to encode the map `#{a => "Apple", b => "Banana"}`:

```
ei_x_encode_map_header(&x, 2);
ei_x_encode_atom(&x, "a");
ei_x_encode_string(&x, "Apple");
ei_x_encode_atom(&x, "b");
ei_x_encode_string(&x, "Banana");
```

A correctly encoded map can not have duplicate keys.

**`int ei_get_type(const char *buf, const int *index, int *type, int *size)`**

This function returns the type in `type` and size in `size` of the encoded term. For strings and atoms, size is the number of characters *not* including the terminating 0. For binaries, `size` is the number of bytes. For lists and tuples, `size` is the arity of the object. For other types, `size` is 0. In all cases, `index` is left unchanged.

**`int ei_decode_version(const char *buf, int *index, int *version)`**

This function decodes the version magic number for the erlang binary term format. It must be the first token in a binary term.

**`int ei_decode_long(const char *buf, int *index, long *p)`**

This function decodes a long integer from the binary format. Note that if the code is 64 bits the function ei_decode_long() is exactly the same as ei_decode_longlong().

**`int ei_decode_ulong(const char *buf, int *index, unsigned long *p)`**

This function decodes an unsigned long integer from the binary format. Note that if the code is 64 bits the function ei_decode_ulong() is exactly the same as ei_decode_ulonglong().

### int ei_decode_longlong(const char *buf, int *index, long long *p)

This function decodes a GCC `long long` or Visual C++ `__int64` (64 bit) integer from the binary format. Note that this function is missing in the VxWorks port.

### int ei_decode_ulonglong(const char *buf, int *index, unsigned long long *p)

This function decodes a GCC `unsigned long long` or Visual C++ `unsigned __int64` (64 bit) integer from the binary format. Note that this function is missing in the VxWorks port.

### int ei_decode_bignum(const char *buf, int *index, mpz_t obj)

This function decodes an integer in the binary format to a GMP `mpz_t` integer. To use this function the ei library needs to be configured and compiled to use the GMP library.

### int ei_decode_double(const char *buf, int *index, double *p)

This function decodes an double-precision (64 bit) floating point number from the binary format.

### int ei_decode_boolean(const char *buf, int *index, int *p)

This function decodes a boolean value from the binary format. A boolean is actually an atom, `true` decodes 1 and `false` decodes 0.

### int ei_decode_char(const char *buf, int *index, char *p)

This function decodes a char (8-bit) integer between 0-255 from the binary format. Note that for historical reasons the returned integer is of type `char`. Your C code should consider the returned value to be of type `unsigned char` even if the C compilers and system may define `char` to be signed.

### int ei_decode_string(const char *buf, int *index, char *p)

This function decodes a string from the binary format. A string in erlang is a list of integers between 0 and 255. Note that since the string is just a list, sometimes lists are encoded as strings by `term_to_binary/1`, even if it was not intended.

The string is copied to `p`, and enough space must be allocated. The returned string is null terminated so you need to add an extra byte to the memory requirement.

### int ei_decode_atom(const char *buf, int *index, char *p)

This function decodes an atom from the binary format. The null terminated name of the atom is placed at `p`. There can be at most `MAXATOMLEN` bytes placed in the buffer.

### int ei_decode_atom_as(const char *buf, int *index, char *p, int plen, erlang_char_encoding want, erlang_char_encoding* was, erlang_char_encoding* result)

This function decodes an atom from the binary format. The null terminated name of the atom is placed in buffer at `p` of length `plen` bytes.

The wanted string encoding is specified by *`want`*. The original encoding used in the binary format (latin1 or utf8) can be obtained from `*was`. The actual encoding of the resulting string (7-bit ascii, latin1 or utf8) can be obtained from `*result`. Both `was` and `result` can be `NULL`. `*result` may differ from `want` if `want` is a bitwise-or'd combination like `ERLANG_LATIN1|ERLANG_UTF8` or if `*result` turn out to be pure 7-bit ascii (compatible with both latin1 and utf8).

This function fails if the atom is too long for the buffer or if it can not be represented with encoding `want`.

This function was introduced in R16 release of Erlang/OTP as part of a first step to support UTF8 atoms.

### `int ei_decode_binary(const char *buf, int *index, void *p, long *len)`

This function decodes a binary from the binary format. The `len` parameter is set to the actual size of the binary. Note that `ei_decode_binary()` assumes that there are enough room for the binary. The size required can be fetched by `ei_get_type()`.

### `int ei_decode_fun(const char *buf, int *index, erlang_fun *p)`
### `void free_fun(erlang_fun* f)`

This function decodes a fun from the binary format. The `p` parameter should be NULL or point to an `erlang_fun` structure. This is the only decode function that allocates memory; when the `erlang_fun` is no longer needed, it should be freed with `free_fun`. (This has to do with the arbitrary size of the environment for a fun.)

### `int ei_decode_pid(const char *buf, int *index, erlang_pid *p)`

Decodes a pid, process identifier, from the binary format.

### `int ei_decode_port(const char *buf, int *index, erlang_port *p)`

This function decodes a port identifier from the binary format.

### `int ei_decode_ref(const char *buf, int *index, erlang_ref *p)`

This function decodes a reference from the binary format.

### `int ei_decode_trace(const char *buf, int *index, erlang_trace *p)`

Decodes an erlang trace token from the binary format.

### `int ei_decode_tuple_header(const char *buf, int *index, int *arity)`

This function decodes a tuple header, the number of elements is returned in `arity`. The tuple elements follows in order in the buffer.

### `int ei_decode_list_header(const char *buf, int *index, int *arity)`

This function decodes a list header from the binary format. The number of elements is returned in `arity`. The `arity +1` elements follows (the last one is the tail of the list, normally an empty list.) If `arity` is `0`, it's an empty list.

Note that lists are encoded as strings, if they consist entirely of integers in the range 0..255. This function will not decode such strings, use `ei_decode_string()` instead.

### `int ei_decode_map_header(const char *buf, int *index, int *arity)`

This function decodes a map header from the binary format. The number of key-value pairs is returned in `*arity`. Keys and values follow in the following order: `K1, V1, K2, V2, ..., Kn, Vn`. This makes a total of `arity*2` terms. If `arity` is zero, it's an empty map. A correctly encoded map does not have duplicate keys.

### `int ei_decode_ei_term(const char* buf, int* index, ei_term* term)`

This function decodes any term, or at least tries to. If the term pointed at by `*index` in `buf` fits in the `term` union, it is decoded, and the appropriate field in `term->value` is set, and `*index` is incremented by the term size.

The function returns 1 on successful decoding, -1 on error, and 0 if the term seems alright, but does not fit in the `term` structure. If it returns 1, the `index` will be incremented, and the `term` contains the decoded term.

The `term` structure will contain the arity for a tuple or list, size for a binary, string or atom. It will contains a term if it's any of the following: integer, float, atom, pid, port or ref.

### `int ei_decode_term(const char *buf, int *index, void *t)`

This function decodes a term from the binary format. The term is return in `t` as a `ETERM*`, so `t` is actually an `ETERM**` (see `erl_interface(3)`. The term should later be deallocated.

Note that this function is located in the erl_interface library.

### `int ei_print_term(FILE* fp, const char* buf, int* index)`
### `int ei_s_print_term(char** s, const char* buf, int* index)`

This function prints a term, in clear text, to the file given by `fp`, or the buffer pointed to by `s`. It tries to resemble the term printing in the erlang shell.

In `ei_s_print_term()`, the parameter `s` should point to a dynamically (malloc) allocated string of `BUFSIZ` bytes or a NULL pointer. The string may be reallocated (and `*s` may be updated) by this function if the result is more than `BUFSIZ` characters. The string returned is zero-terminated.

The return value is the number of characters written to the file or string, or -1 if `buf[index]` doesn't contain a valid term. Unfortunately, I/O errors on `fp` is not checked.

The argument `index` is updated, i.e. this function can be viewed as en decode function that decodes a term into a human readable format.

### `int ei_x_format(ei_x_buff* x, const char* fmt, ...)`
### `int ei_x_format_wo_ver(ei_x_buff* x, const char *fmt, ... )`

Format a term, given as a string, to a buffer. This functions works like a sprintf for erlang terms. The `fmt` contains a format string, with arguments like `~d`, to insert terms from variables. The following formats are supported (with the C types given):

```
~a - an atom, char*
~c - a character, char
~s - a string, char*
~i - an integer, int
~l - a long integer, long int
~u - a unsigned long integer, unsigned long int
~f - a float, float
~d - a double float, double float
~p - an Erlang PID, erlang_pid*
```

For instance, to encode a tuple with some stuff:

```
ei_x_format("{~a,~i,~d}", "numbers", 12, 3.14159)
encodes the tuple {numbers,12,3.14159}
```

The `ei_x_format_wo_ver()` formats into a buffer, without the initial version byte.

**`int ei_x_new(ei_x_buff* x)`**
**`int ei_x_new_with_version(ei_x_buff* x)`**

This function allocates a new `ei_x_buff` buffer. The fields of the structure pointed to by `x` parameter is filled in, and a default buffer is allocated. The `ei_x_new_with_version()` also puts an initial version byte, that is used in the binary format. (So that `ei_x_encode_version()` won't be needed.)

**`int ei_x_free(ei_x_buff* x)`**

This function frees an `ei_x_buff` buffer. The memory used by the buffer is returned to the OS.

**`int ei_x_append(ei_x_buff* x, const ei_x_buff* x2)`**
**`int ei_x_append_buf(ei_x_buff* x, const char* buf, int len)`**

These functions appends data at the end of the buffer `x`.

**`int ei_skip_term(const char* buf, int* index)`**

This function skips a term in the given buffer, it recursively skips elements of lists and tuples, so that a full term is skipped. This is a way to get the size of an erlang term.

`buf` is the buffer.

`index` is updated to point right after the term in the buffer.

> Note:
>
> This can be useful when you want to hold arbitrary terms: just skip them and copy the binary term data to some buffer.

The function returns `0` on success and `-1` on failure.

## Debug Information

Some tips on what to check when the emulator doesn't seem to receive the terms that you send.

- be careful with the version header, use `ei_x_new_with_version()` when appropriate
- turn on distribution tracing on the erlang node
- check the result codes from ei_decode_-calls

## See Also

erl_interface(3)

# ei_connect

C Library

This module enables C programs to communicate with erlang nodes, using the erlang distribution over TCP/IP.

A C node appears to Erlang as a *hidden node*. That is, Erlang processes that know the name of the C node are able to communicate with it in a normal manner, but the node name will not appear in the listing provided by the Erlang function `nodes/0`.

The environment variable `ERL_EPMD_PORT` can be used to indicate which logical cluster a C node belongs to.

## Timeout functions

Most functions appear in a version with the suffix `_tmo` appended to the function name. Those function take an additional argument, a timeout in *milliseconds*. The semantics is this; for each communication primitive involved in the operation, if the primitive does not complete within the time specified, the function will return an error and `erl_errno` will be set to `ETIMEDOUT`. With communication primitive is meant an operation on the socket, like `connect`, `accept`, `recv` or `send`.

Obviously the timeouts are for implementing fault tolerance, not to keep hard realtime promises. The `_tmo` functions are for detecting non-responsive peers and to avoid blocking on socket operations.

A timeout value of `0` (zero), means that timeouts are disabled. Calling a `_tmo`-function with the last argument as `0` is therefore exactly the same thing as calling the function without the `_tmo` suffix.

As with all other ei functions, you are *not* expected to put the socket in non blocking mode yourself in the program. Every use of non blocking mode is embedded inside the timeout functions. The socket will always be back in blocking mode after the operations are completed (regardless of the result). To avoid problems, leave the socket options alone. Ei will handle any socket options that need modification.

In all other senses, the `_tmo` functions inherit all the return values and the semantics from the functions without the `_tmo` suffix.

## Exports

**`int ei_connect_init(ei_cnode* ec, const char* this_node_name, const char *cookie, short creation)`**

**`int ei_connect_xinit(ei_cnode* ec, const char *thishostname, const char *thisalivename, const char *thisnodename, Erl_IpAddr thisipaddr, const char *cookie, short creation)`**

These function initializes the `ec` structure, to identify the node name and cookie of the server. One of them has to be called before other functions that works on the type `ei_cnode` or a file descriptor associated with a connection to another node are used.

`ec` is a structure containing information about the C-node. It is used in other `ei` functions for connecting and receiving data.

`this_node_name` is the registered name of the process (the name before '@').

`cookie` is the cookie for the node.

`creation` identifies a specific instance of a C node. It can help prevent the node from receiving messages sent to an earlier process with the same registered name.

`thishostname` is the name of the machine we're running on. If long names are to be used, it should be fully qualified (i.e. `durin.erix.ericsson.se` instead of `durin`).

`thisalivename` is the registered name of the process.

`thisnodename` is the full name of the node, i.e. `einode@durin`.

`thispaddr` if the IP address of the host.

A C node acting as a server will be assigned a creation number when it calls `ei_publish()`.

A connection is closed by simply closing the socket. Refer to system documentation to close the socket gracefully (when there are outgoing packets before close).

This function return a negative value indicating that an error occurred.

Example 1:

```
int n = 0;
struct in_addr addr;
ei_cnode ec;
addr.s_addr = inet_addr("150.236.14.75");
if (ei_connect_xinit(&ec,
                     "chivas",
                     "madonna",
                     "madonna@chivas.du.etx.ericsson.se",
                     &addr;
                     "cookie...",
                     n++) < 0) {
    fprintf(stderr,"ERROR when initializing: %d",erl_errno);
    exit(-1);
}
```

Example 2:

```
if (ei_connect_init(&ec, "madonna", "cookie...", n++) < 0) {
    fprintf(stderr,"ERROR when initializing: %d",erl_errno);
    exit(-1);
}
```

**`int ei_connect(ei_cnode* ec, char *nodename)`**

**`int ei_xconnect(ei_cnode* ec, Erl_IpAddr adr, char *alivename)`**

These functions set up a connection to an Erlang node.

`ei_xconnect()` requires the IP address of the remote host and the alive name of the remote node to be specified. `ei_connect()` provides an alternative interface, and determines the information from the node name provided.

`addr` is the 32-bit IP address of the remote host.

`alive` is the alivename of the remote node.

`node` is the name of the remote node.

These functions return an open file descriptor on success, or a negative value indicating that an error occurred --- in which case they will set `erl_errno` to one of:

`EHOSTUNREACH`
: The remote host `node` is unreachable

`ENOMEM`
: No more memory available.

EIO
: I/O error.

Additionally, errno values from socket*(2)* and connect*(2)* system calls may be propagated into erl_errno.

Example:

```
#define NODE   "madonna@chivas.du.etx.ericsson.se"
#define ALIVE  "madonna"
#define IP_ADDR "150.236.14.75"

/*** Variant 1 ***/
int fd = ei_connect(&ec, NODE);

/*** Variant 2 ***/
struct in_addr addr;
addr.s_addr = inet_addr(IP_ADDR);
fd = ei_xconnect(&ec, &addr, ALIVE);
```

### int ei_connect_tmo(ei_cnode* ec, char *nodename, unsigned timeout_ms)
### int ei_xconnect_tmo(ei_cnode* ec, Erl_IpAddr adr, char *alivename, unsigned timeout_ms)

ei_connect and ei_xconnect with an optional timeout argument, see the description at the beginning of this document.

### int ei_receive(int fd, unsigned char* bufp, int bufsize)

This function receives a message consisting of a sequence of bytes in the Erlang external format.

fd is an open descriptor to an Erlang connection. It is obtained from a previous ei_connect or ei_accept.

bufp is a buffer large enough to hold the expected message.

bufsize indicates the size of bufp.

If a *tick* occurs, i.e., the Erlang node on the other end of the connection has polled this node to see if it is still alive, the function will return ERL_TICK and no message will be placed in the buffer. Also, erl_errno will be set to EAGAIN.

On success, the message is placed in the specified buffer and the function returns the number of bytes actually read. On failure, the function returns ERL_ERROR and will set erl_errno to one of:

EAGAIN
: Temporary error: Try again.

EMSGSIZE
: Buffer too small.

EIO
: I/O error.

### int ei_receive_tmo(int fd, unsigned char* bufp, int bufsize, unsigned timeout_ms)

ei_receive with an optional timeout argument, see the description at the beginning of this document.

**int ei_receive_msg(int fd, erlang_msg* msg, ei_x_buff* x)**

**int ei_xreceive_msg(int fd, erlang_msg* msg, ei_x_buff* x)**

These functions receives a message to the buffer in `x`. `ei_xreceive_msg` allows the buffer in `x` to grow, but `ei_receive_msg` fails if the message is bigger than the preallocated buffer in `x`.

`fd` is an open descriptor to an Erlang connection.

`msg` is a pointer to an `erlang_msg` structure and contains information on the message received.

`x` is buffer obtained from `ei_x_new`.

On success, the function returns `ERL_MSG` and the `msg` struct will be initialized. `erlang_msg` is defined as follows:

```
typedef struct {
    long msgtype;
    erlang_pid from;
    erlang_pid to;
    char toname[MAXATOMLEN+1];
    char cookie[MAXATOMLEN+1];
    erlang_trace token;
} erlang_msg;
```

`msgtype` identifies the type of message, and is one of `ERL_SEND`, `ERL_REG_SEND`, `ERL_LINK`, `ERL_UNLINK` and `ERL_EXIT`.

If `msgtype` is `ERL_SEND` this indicates that an ordinary send operation has taken place, and `msg->to` contains the Pid of the recipient (the C-node). If `type` is `ERL_REG_SEND` then a registered send operation took place, and `msg->from` contains the Pid of the sender.

If `msgtype` is `ERL_LINK` or `ERL_UNLINK`, then `msg->to` and `msg->from` contain the pids of the sender and recipient of the link or unlink.

If `msgtype` is `ERL_EXIT`, then this indicates that a link has been broken. In this case, `msg->to` and `msg->from` contain the pids of the linked processes.

The return value is the same as for `ei_receive`, see above.

**int ei_receive_msg_tmo(int fd, erlang_msg* msg, ei_x_buff* x, unsigned imeout_ms)**

**int ei_xreceive_msg_tmo(int fd, erlang_msg* msg, ei_x_buff* x, unsigned timeout_ms)**

ei_receive_msg and ei_xreceive_msg with an optional timeout argument, see the description at the beginning of this document.

**int ei_receive_encoded(int fd, char **mbufp, int *bufsz, erlang_msg *msg, int *msglen)**

This function is retained for compatibility with code generated by the interface compiler and with code following examples in the same application.

In essence the function performs the same operation as `ei_xreceive_msg`, but instead of using an ei_x_buff, the function expects a pointer to a character pointer (`mbufp`), where the character pointer should point to a memory area allocated by `malloc`. The argument `bufsz` should be a pointer to an integer containing the exact size (in bytes) of the memory area. The function may reallocate the memory area and will in such cases put the new size in `*bufsz` and update `*mbufp`.

Furthermore the function returns either ERL_TICK or the `msgtype` field of the `erlang_msg *msg`. The actual length of the message is put in `*msglen`. On error it will return a value `< 0`.

It is recommended to use ei_xreceive_msg instead when possible, for the sake of readability. The function will however be retained in the interface for compatibility and will *not* be removed not be removed in future releases without notice.

### `int ei_receive_encoded_tmo(int fd, char **mbufp, int *bufsz, erlang_msg *msg, int *msglen, unsigned timeout_ms)`

ei_receive_encoded with an optional timeout argument, see the description at the beginning of this document.

### `int ei_send(int fd, erlang_pid* to, char* buf, int len)`

This function sends an Erlang term to a process.

`fd` is an open descriptor to an Erlang connection.

`to` is the Pid of the intended recipient of the message.

`buf` is the buffer containing the term in binary format.

`len` is the length of the message in bytes.

The function returns 0 if successful, otherwise -1, in the latter case it will set `erl_errno` to `EIO`.

### `int ei_send_tmo(int fd, erlang_pid* to, char* buf, int len, unsigned timeout_ms)`

ei_send with an optional timeout argument, see the description at the beginning of this document.

### `int ei_send_encoded(int fd, erlang_pid* to, char* buf, int len)`

Works exactly as ei_send, the alternative name retained for backward compatibility. The function will *not* be removed without notice.

### `int ei_send_encoded_tmo(int fd, erlang_pid* to, char* buf, int len, unsigned timeout_ms)`

ei_send_encoded with an optional timeout argument, see the description at the beginning of this document.

### `int ei_reg_send(ei_cnode* ec, int fd, char* server_name, char* buf, int len)`

This function sends an Erlang term to a registered process.

This function sends an Erlang term to a process.

`fd` is an open descriptor to an Erlang connection.

`server_name` is the registered name of the intended recipient.

`buf` is the buffer containing the term in binary format.

`len` is the length of the message in bytes.

The function returns 0 if successful, otherwise -1, in the latter case it will set `erl_errno` to `EIO`.

Example, send the atom "ok" to the process "worker":

```
ei_x_buff x;
ei_x_new_with_version(&x);
ei_x_encode_atom(&x, "ok");
```

```
if (ei_reg_send(&ec, fd, x.buff, x.index) < 0)
    handle_error();
```

**int ei_reg_send_tmo(ei_cnode* ec, int fd, char* server_name, char* buf, int len, unsigned timeout_ms)**

ei_reg_send with an optional timeout argument, see the description at the beginning of this document.

**int ei_send_reg_encoded(int fd, const erlang_pid *from, const char *to, const char *buf, int len)**

This function is retained for compatibility with code generated by the interface compiler and with code following examples in the same application.

The function works as `ei_reg_send` with one exception. Instead of taking the `ei_cnode` as a first argument, it takes a second argument, an `erlang_pid` which should be the process identifier of the sending process (in the erlang distribution protocol).

A suitable `erlang_pid` can be constructed from the `ei_cnode` structure by the following example code:

```
        ei_cnode ec;
        erlang_pid *self;
        int fd; /* the connection fd */
        ...
        self = ei_self(&ec);
        self->num = fd;
```

**int ei_send_reg_encoded_tmo(int fd, const erlang_pid *from, const char *to, const char *buf, int len)**

ei_send_reg_encoded with an optional timeout argument, see the description at the beginning of this document.

**int ei_rpc(ei_cnode *ec, int fd, char *mod, char *fun, const char *argbuf, int argbuflen, ei_x_buff *x)**

**int ei_rpc_to(ei_cnode *ec, int fd, char *mod, char *fun, const char *argbuf, int argbuflen)**

**int ei_rpc_from(ei_cnode *ec, int fd, int timeout, erlang_msg *msg, ei_x_buff *x)**

These functions support calling Erlang functions on remote nodes. `ei_rpc_to()` sends an rpc request to a remote node and `ei_rpc_from()` receives the results of such a call. `ei_rpc()` combines the functionality of these two functions by sending an rpc request and waiting for the results. See also `rpc:call/4`.

`ec` is the C-node structure previously initiated by a call to `ei_connect_init()` or `ei_connect_xinit()`

`fd` is an open descriptor to an Erlang connection.

`timeout` is the maximum time (in ms) to wait for results. Specify `ERL_NO_TIMEOUT` to wait forever. `ei_rpc()` will wait infinitely for the answer, i.e. the call will never time out.

`mod` is the name of the module containing the function to be run on the remote node.

`fun` is the name of the function to run.

`argbuf` is a pointer to a buffer with an encoded Erlang list, without a version magic number, containing the arguments to be passed to the function.

`argbuflen` is the length of the buffer containing the encoded Erlang list.

`msg` structure of type `erlang_msg` and contains information on the message received. See `ei_receive_msg()` for a description of the `erlang_msg` format.

`x` points to the dynamic buffer that receives the result. For for `ei_rpc()` this will be the result without the version magic number. For `ei_rpc_from()` the result will return a version magic number and a 2-tuple `{rex,Reply}`.

`ei_rpc()` returns the number of bytes in the result on success and -1 on failure. `ei_rpc_from()` returns number of bytes or one of `ERL_TICK`, `ERL_TIMEOUT` and `ERL_ERROR` otherwise. When failing, all three functions set `erl_errno` to one of:

`EIO`
: I/O error.

`ETIMEDOUT`
: Timeout expired.

`EAGAIN`
: Temporary error: Try again.

Example, check to see if an erlang process is alive:

```
int index = 0, is_alive;
ei_x_buff args, result;

ei_x_new(&result);
ei_x_new(&args);
ei_x_encode_list_header(&args, 1);
ei_x_encode_pid(&args, &check_pid);
ei_x_encode_empty_list(&args);

if (ei_rpc(&ec, fd, "erlang", "is_process_alive",
           args.buff, args.index, &result) < 0)
    handle_error();

if (ei_decode_version(result.buff, &index) < 0
    || ei_decode_bool(result.buff, &index, &is_alive) < 0)
    handle_error();
```

### `int ei_publish(ei_cnode *ec, int port)`

These functions are used by a server process to register with the local name server *epmd*, thereby allowing other processes to send messages by using the registered name. Before calling either of these functions, the process should have called `bind()` and `listen()` on an open socket.

`ec` is the C-node structure.

`port` is the local name to register, and should be the same as the port number that was previously bound to the socket.

`addr` is the 32-bit IP address of the local host.

To unregister with epmd, simply close the returned descriptor. Do not use `ei_unpublish()`, which is deprecated anyway.

On success, the functions return a descriptor connecting the calling process to epmd. On failure, they return -1 and set `erl_errno` to `EIO`.

Additionally, `errno` values from `socket`*(2)* and `connect`*(2)* system calls may be propagated into `erl_errno`.

### int ei_publish_tmo(ei_cnode *ec, int port, unsigned timeout_ms)

ei_publish with an optional timeout argument, see the description at the beginning of this document.

### int ei_accept(ei_cnode *ec, int listensock, ErlConnect *conp)

This function is used by a server process to accept a connection from a client process.

`ec` is the C-node structure.

`listensock` is an open socket descriptor on which `listen()` has previously been called.

`conp` is a pointer to an `ErlConnect` struct, described as follows:

```
typedef struct {
  char ipadr[4];
  char nodename[MAXNODELEN];
} ErlConnect;
```

On success, `conp` is filled in with the address and node name of the connecting client and a file descriptor is returned. On failure, `ERL_ERROR` is returned and `erl_errno` is set to `EIO`.

### int ei_accept_tmo(ei_cnode *ec, int listensock, ErlConnect *conp, unsigned timeout_ms)

ei_accept with an optional timeout argument, see the description at the beginning of this document.

### int ei_unpublish(ei_cnode *ec)

This function can be called by a process to unregister a specified node from epmd on the localhost. This is however usually not allowed, unless epmd was started with the -relaxed_command_check flag, which it normally isn't.

To unregister a node you have published, you should close the descriptor that was returned by `ei_publish()`.

> Warning:
>
> This function is deprecated and will be removed in a future release.

`ec` is the node structure of the node to unregister.

If the node was successfully unregistered from epmd, the function returns 0. Otherwise, it returns -1 and sets `erl_errno` is to `EIO`.

### int ei_unpublish_tmo(ei_cnode *ec, unsigned timeout_ms)

ei_unpublish with an optional timeout argument, see the description at the beginning of this document.

### const char *ei_thisnodename(ei_cnode *ec)
### const char *ei_thishostname(ei_cnode *ec)
### const char *ei_thisalivename(ei_cnode *ec)

These functions can be used to retrieve information about the C Node. These values are initially set with `ei_connect_init()` or `ei_connect_xinit()`.

They simply fetches the appropriate field from the `ec` structure. Read the field directly will probably be safe for a long time, so these functions are not really needed.

**`erlang_pid *ei_self(ei_cnode *ec)`**

This function retrieves the Pid of the C-node. Every C-node has a (pseudo) pid used in `ei_send_reg`, `ei_rpc` and others. This is contained in a field in the `ec` structure. It will be safe for a long time to fetch this field directly from the `ei_cnode` structure.

**`struct hostent *ei_gethostbyname(const char *name)`**

**`struct hostent *ei_gethostbyaddr(const char *addr, int len, int type)`**

**`struct hostent *ei_gethostbyname_r(const char *name, struct hostent *hostp, char *buffer, int buflen, int *h_errnop)`**

**`struct hostent *ei_gethostbyaddr_r(const char *addr, int length, int type, struct hostent *hostp, char *buffer, int buflen, int *h_errnop)`**

These are convenience functions for some common name lookup functions.

**`int ei_get_tracelevel(void)`**

**`void ei_set_tracelevel(int level)`**

These functions are used to set tracing on the distribution. The levels are different verbosity levels. A higher level means more information. See also Debug Information and `EI_TRACELEVEL` below.

`ei_set_tracelevel` and `ei_get_tracelevel` are not thread safe.

## Debug Information

If a connection attempt fails, the following can be checked:

- `erl_errno`
- that the right cookie was used
- that *epmd* is running
- the remote Erlang node on the other side is running the same version of Erlang as the `ei` library.
- the environment variable `ERL_EPMD_PORT` is set correctly.

The connection attempt can be traced by setting a tracelevel by either using `ei_set_tracelevel` or by setting the environment variable `EI_TRACELEVEL`. The different tracelevels has the following messages:

- 1: Verbose error messages
- 2: Above messages and verbose warning messages
- 3: Above messages and progress reports for connection handling
- 4: Above messages and progress reports for communication
- 5: Above messages and progress reports for data conversion

# registry

C Library

This module provides support for storing key-value pairs in a table known as a registry, backing up registries to Mnesia in an atomic manner, and later restoring the contents of a registry from Mnesia.

## Exports

**`ei_reg *ei_reg_open(size)`**

Types:

```
int size;
```

Open (create) a registry. The registry will be initially empty. Use `ei_reg_close()` to close the registry later.

`size` is the approximate number of objects you intend to store in the registry. Since the registry uses a hash table with collision chaining, there is no absolute upper limit on the number of objects that can be stored in it. However for reasons of efficiency, it is a good idea to choose a number that is appropriate for your needs. It is possible to use `ei_reg_resize()` to change the size later. Note that the number you provide will be increased to the nearest larger prime number.

On success, an empty registry will be returned. On failure, NULL will be returned.

**`int ei_reg_resize(reg,newsize)`**

Types:

```
ei_reg *reg;
int newsize;
```

Change the size of a registry.

`newsize` is the new size to make the registry. The number will be increased to the nearest larger prime number.

On success, the registry will be resized, all contents rehashed, and the function will return 0. On failure, the registry will be left unchanged and the function will return -1.

**`int ei_reg_close(reg)`**

Types:

```
ei_reg *reg;
```

A registry that has previously been created with `ei_reg_open()` is closed, and all the objects it contains are freed.

`reg` is the registry to close.

The function returns 0.

**`int ei_reg_setival(reg,key,i)`**

Types:

```
ei_reg *reg;
const char *key;
int i;
```

Create a key-value pair with the specified `key` and integer value `i`. If an object already existed with the same `key`, the new value replaces the old one. If the previous value was a binary or string, it is freed with `free()`.

`reg` is the registry where the object should be placed.

`key` is the name of the object.

`i` is the integer value to assign.

The function returns 0 on success, or -1 on failure.

**`int ei_reg_setfval(reg,key,f)`**

Types:

```
ei_reg *reg;
const char *key;
double f;
```

Create a key-value pair with the specified `key` and floating point value `f`. If an object already existed with the same `key`, the new value replaces the old one. If the previous value was a binary or string, it is freed with `free()`.

`reg` is the registry where the object should be placed.

`key` is the name of the object.

`f` is the floating point value to assign.

The function returns 0 on success, or -1 on failure.

**`int ei_reg_setsval(reg,key,s)`**

Types:

```
ei_reg *reg;
const char *key;
const char *s;
```

Create a key-value pair with the specified `key` whose "value" is the specified string `s`. If an object already existed with the same `key`, the new value replaces the old one. If the previous value was a binary or string, it is freed with `free()`.

`reg` is the registry where the object should be placed.

`key` is the name of the object.

`s` is the string to assign. The string itself must have been created through a single call to `malloc()` or similar function, so that the registry can later delete it if necessary by calling `free()`.

The function returns 0 on success, or -1 on failure.

**`int ei_reg_setpval(reg,key,p,size)`**

Types:

```
ei_reg *reg;
const char *key;
const void *p;
int size;
```

Create a key-value pair with the specified `key` whose "value" is the binary object pointed to by `p`. If an object already existed with the same `key`, the new value replaces the old one. If the previous value was a binary or string, it is freed with `free()`.

`reg` is the registry where the object should be placed.

`key` is the name of the object.

`p` is a pointer to the binary object. The object itself must have been created through a single call to `malloc()` or similar function, so that the registry can later delete it if necessary by calling `free()`.

`size` is the length in bytes of the binary object.

The function returns 0 on success, or -1 on failure.

**`int ei_reg_setval(reg,key,flags,v,...)`**

Types:

```
ei_reg *reg;
const char *key;
int flags;
v (see below)
```

Create a key-value pair with the specified `key` whose value is specified by `v`. If an object already existed with the same `key`, the new value replaces the old one. If the previous value was a binary or string, it is freed with `free()`.

`reg` is the registry where the object should be placed.

`key` is the name of the object.

`flags` indicates the type of the object specified by `v`. Flags must be one of EI_INT, EI_FLT, EI_STR and EI_BIN, indicating whether `v` is `int`, `double`, `char*` or `void*`. If `flags` is EI_BIN, then a fifth argument `size` is required, indicating the size in bytes of the object pointed to by `v`.

If you wish to store an arbitrary pointer in the registry, specify a `size` of 0. In this case, the object itself will not be transferred by an `ei_reg_dump()` operation, just the pointer value.

The function returns 0 on success, or -1 on failure.

**`int ei_reg_getival(reg,key)`**

Types:

```
ei_reg *reg;
const char *key;
```

Get the value associated with `key` in the registry. The value must be an integer.

`reg` is the registry where the object will be looked up.

`key` is the name of the object to look up.

On success, the function returns the value associated with `key`. If the object was not found or it was not an integer object, -1 is returned. To avoid problems with in-band error reporting (i.e. if you cannot distinguish between -1 and a valid result) use the more general function `ei_reg_getval()` instead.

**`double ei_reg_getfval(reg,key)`**

Types:

```
ei_reg *reg;
const char *key;
```

Get the value associated with `key` in the registry. The value must be a floating point type.

`reg` is the registry where the object will be looked up.

`key` is the name of the object to look up.

On success, the function returns the value associated with `key`. If the object was not found or it was not a floating point object, -1.0 is returned. To avoid problems with in-band error reporting (i.e. if you cannot distinguish between -1.0 and a valid result) use the more general function `ei_reg_getval()` instead.

**`const char *ei_reg_getsval(reg,key)`**

Types:

```
ei_reg *reg;
const char *key;
```

Get the value associated with `key` in the registry. The value must be a string.

`reg` is the registry where the object will be looked up.

`key` is the name of the object to look up.

On success, the function returns the value associated with `key`. If the object was not found or it was not a string, NULL is returned. To avoid problems with in-band error reporting (i.e. if you cannot distinguish between NULL and a valid result) use the more general function `ei_reg_getval()` instead.

**`const void *ei_reg_getpval(reg,key,size)`**

Types:

```
ei_reg *reg;
const char *key;
int size;
```

Get the value associated with `key` in the registry. The value must be a binary (pointer) type.

`reg` is the registry where the object will be looked up.

`key` is the name of the object to look up.

`size` will be initialized to contain the length in bytes of the object, if it is found.

On success, the function returns the value associated with `key` and indicates its length in `size`. If the object was not found or it was not a binary object, NULL is returned. To avoid problems with in-band error reporting (i.e. if you cannot distinguish between NULL and a valid result) use the more general function `ei_reg_getval()` instead.

**`int ei_reg_getval(reg,key,flags,v,...)`**

Types:

```
ei_reg *reg;
const char *key;
int flags;
void *v (see below)
```

This is a general function for retrieving any kind of object from the registry.

`reg` is the registry where the object will be looked up.

`key` is the name of the object to look up.

`flags` indicates the type of object that you are looking for. If `flags` is 0, then any kind of object will be returned. If `flags` is one of EI_INT, EI_FLT, EI_STR or EI_BIN, then only values of that kind will be returned. The buffer pointed to by `v` must be large enough to hold the return data, i.e. it must be a pointer to one of `int`, `double`, `char*` or `void*`, respectively. Also, if `flags` is EI_BIN, then a fifth argument `int *size` is required, so that the size of the object can be returned.

If the function succeeds, `v` (and `size` if the object is binary) will be initialized with the value associated with `key`, and the function will return one of EI_INT, EI_FLT, EI_STR or EI_BIN, indicating the type of object. On failure the function will return -1 and the arguments will not be updated.

### `int ei_reg_markdirty(reg,key)`

Types:

```
ei_reg *reg;
const char *key;
```

Mark a registry object as dirty. This will ensure that it is included in the next backup to Mnesia. Normally this operation will not be necessary since all of the normal registry 'set' functions do this automatically. However if you have retrieved the value of a string or binary object from the registry and modified the contents, then the change will be invisible to the registry and the object will be assumed to be unmodified. This function allows you to make such modifications and then let the registry know about them.

`reg` is the registry containing the object.

`key` is the name of the object to mark.

The function returns 0 on success, or -1 on failure.

### `int ei_reg_delete(reg,key)`

Types:

```
ei_reg *reg;
const char *key;
```

Delete an object from the registry. The object is not actually removed from the registry, it is only marked for later removal so that on subsequent backups to Mnesia, the corresponding object can be removed from the Mnesia table as well. If another object is later created with the same key, the object will be reused.

The object will be removed from the registry after a call to `ei_reg_dump()` or `ei_reg_purge()`.

`reg` is the registry containing `key`.

`key` is the object to remove.

If the object was found, the function returns 0 indicating success. Otherwise the function returns -1.

### `int ei_reg_stat(reg,key,obuf)`

Types:

```
ei_reg *reg;
const char *key;
struct ei_reg_stat *obuf;
```

Return information about an object.

`reg` is the registry containing the object.

`key` is the name of the object.

`obuf` is a pointer to an `ei_reg_stat` structure, defined below:

```
struct ei_reg_stat {
  int attr;
  int size;
};
```

In `attr` the object's attributes are stored as the logical OR of its type (one of EI_INT, EI_FLT, EI_BIN and EI_STR), whether it is marked for deletion (EI_DELET) and whether it has been modified since the last backup to Mnesia (EI_DIRTY).

The `size` field indicates the size in bytes required to store EI_STR (including the terminating 0) and EI_BIN objects, or 0 for EI_INT and EI_FLT.

The function returns 0 and initializes `obuf` on success, or returns -1 on failure.

### `int ei_reg_tabstat(reg,obuf)`

Types:

```
ei_reg *reg;
struct ei_reg_tabstat *obuf;
```

Return information about a registry. Using information returned by this function, you can see whether the size of the registry is suitable for the amount of data it contains.

`reg` is the registry to return information about.

`obuf` is a pointer to an `ei_reg_tabstat` structure, defined below:

```
struct ei_reg_tabstat {
  int size;
  int nelem;
  int npos;
  int collisions;
};
```

The `size` field indicates the number of hash positions in the registry. This is the number you provided when you created or last resized the registry, rounded up to the nearest prime.

`nelem` indicates the number of elements stored in the registry. It includes objects that are deleted but not purged.

`npos` indicates the number of unique positions that are occupied in the registry.

`collisions` indicates how many elements are sharing positions in the registry.

On success, the function returns 0 and `obuf` is initialized to contain table statistics. On failure, the function returns -1.

### `int ei_reg_dump(fd,reg,mntab,flags)`

Types:

```
int fd;
ei_reg *reg;
const char *mntab;
int flags;
```

Dump the contents of a registry to a Mnesia table in an atomic manner, i.e. either all data will be updated, or none of it will. If any errors are encountered while backing up the data, the entire operation is aborted.

`fd` is an open connection to Erlang. Mnesia 3.0 or later must be running on the Erlang node.

`reg` is the registry to back up.

`mntab` is the name of the Mnesia table where the backed up data should be placed. If the table does not exist, it will be created automatically using configurable defaults. See your Mnesia documentation for information about configuring this behaviour.

If `flags` is 0, the backup will include only those objects which have been created, modified or deleted since the last backup or restore (i.e. an incremental backup). After the backup, any objects that were marked dirty are now clean, and any objects that had been marked for deletion are deleted.

Alternatively, setting flags to EI_FORCE will cause a full backup to be done, and EI_NOPURGE will cause the deleted objects to be left in the registry afterwards. These can be bitwise ORed together if both behaviours are desired. If EI_NOPURGE was specified, you can use `ei_reg_purge()` to explicitly remove the deleted items from the registry later.

The function returns 0 on success, or -1 on failure.

### `int ei_reg_restore(fd,reg,mntab)`

Types:

```
int fd;
ei_reg *reg;
const char *mntab;
```

The contents of a Mnesia table are read into the registry.

`fd` is an open connection to Erlang. Mnesia 3.0 or later must be running on the Erlang node.

`reg` is the registry where the data should be placed.

`mntab` is the name of the Mnesia table to read data from.

Note that only tables of a certain format can be restored, i.e. those that have been created and backed up to with `ei_reg_dump()`. If the registry was not empty before the operation, then the contents of the table are added to the contents of the registry. If the table contains objects with the same keys as those already in the registry, the registry objects will be overwritten with the new values. If the registry contains objects that were not in the table, they will be unchanged by this operation.

After the restore operation, the entire contents of the registry is marked as unmodified. Note that this includes any objects that were modified before the restore and not overwritten by the restore.

The function returns 0 on success, or -1 on failure.

### `int ei_reg_purge(reg)`

Types:

```
ei_reg *reg;
```

Remove all objects marked for deletion. When objects are deleted with `ei_reg_delete()` they are not actually removed from the registry, only marked for later removal. This is so that on a subsequent backup to Mnesia, the objects can also be removed from the Mnesia table. If you are not backing up to Mnesia then you may wish to remove the objects manually with this function.

`reg` is a registry containing objects marked for deletion.

The function returns 0 on success, or -1 on failure.

# erl_connect

C Library

This module provides support for communication between distributed Erlang nodes and C nodes, in a manner that is transparent to Erlang processes.

A C node appears to Erlang as a *hidden node*. That is, Erlang processes that know the name of the C node are able to communicate with it in a normal manner, but the node name will not appear in the listing provided by the Erlang function `nodes/0`.

## Exports

**`int erl_connect_init(number, cookie, creation)`**
**`int erl_connect_xinit(host, alive, node, addr, cookie, creation)`**

Types:

```
int number;
char *cookie;
short creation;
char *host,*alive,*node;
struct in_addr *addr;
```

These functions initialize the `erl_connect` module. In particular, they are used to identify the name of the C-node from which they are called. One of these functions must be called before any of the other functions in the erl_connect module are used.

`erl_connect_xinit()` stores for later use information about the node's host name `host`, alive name `alive`, node name `node`, IP address `addr`, cookie `cookie`, and creation number `creation`. `erl_connect_init()` provides an alternative interface which does not require as much information from the caller. Instead, `erl_connect_init()` uses `gethostbyname()` to obtain default values.

If you use `erl_connect_init()` your node will have a short name, i.e., it will not be fully qualified. If you need to use fully qualified (a.k.a. long) names, use `erl_connect_xinit()` instead.

`host` is the name of the host on which the node is running.

`alive` is the alivename of the node.

`node` is the name of the node. The nodename should be of the form *alivename@hostname*.

`addr` is the 32-bit IP address of `host`.

`cookie` is the authorization string required for access to the remote node. If NULL the user HOME directory is searched for a cookie file `.erlang.cookie`. The path to the home directory is retrieved from the environment variable `HOME` on Unix and from the `HOMEDRIVE` and `HOMEPATH` variables on Windows. Refer to the `auth` module for more details.

`creation` helps identify a particular instance of a C node. In particular, it can help prevent us from receiving messages sent to an earlier process with the same registered name.

A C node acting as a server will be assigned a creation number when it calls `erl_publish()`.

`number` is used by `erl_connect_init()` to construct the actual node name. In the second example shown below, *"c17@a.DNS.name"* will be the resulting node name.

Example 1:

```
struct in_addr addr;
addr = inet_addr("150.236.14.75");
if (!erl_connect_xinit("chivas",
                       "madonna",
                       "madonna@chivas.du.etx.ericsson.se",
                       &addr;
                       "samplecookiestring..."),
                       0)
  erl_err_quit("<ERROR> when initializing !");
```

Example 2:

```
if (!erl_connect_init(17, "samplecookiestring...", 0))
  erl_err_quit("<ERROR> when initializing !");
```

**int erl_connect(node)**

**int erl_xconnect(addr, alive)**

Types:

**char *node, *alive;**

**struct in_addr *addr;**

These functions set up a connection to an Erlang node.

`erl_xconnect()` requires the IP address of the remote host and the alive name of the remote node to be specified. `erl_connect()` provides an alternative interface, and determines the information from the node name provided.

`addr` is the 32-bit IP address of the remote host.

`alive` is the alivename of the remote node.

`node` is the name of the remote node.

These functions return an open file descriptor on success, or a negative value indicating that an error occurred --- in which case they will set `erl_errno` to one of:

`EHOSTUNREACH`
: The remote host `node` is unreachable

`ENOMEM`
: No more memory available.

`EIO`
: I/O error.

Additionally, `errno` values from `socket`*(2)* and `connect`*(2)* system calls may be propagated into `erl_errno`.

```
#define NODE   "madonna@chivas.du.etx.ericsson.se"
#define ALIVE  "madonna"
#define IP_ADDR "150.236.14.75"

/*** Variant 1 ***/
erl_connect( NODE );

/*** Variant 2 ***/
struct in_addr addr;
addr = inet_addr(IP_ADDR);
```

```
erl_xconnect( &addr , ALIVE );
```

### int erl_close_connection(fd)

Types:

```
int fd;
```

This function closes an open connection to an Erlang node.

`Fd` is a file descriptor obtained from `erl_connect()` or `erl_xconnect()`.

On success, 0 is returned. If the call fails, a non-zero value is returned, and the reason for the error can be obtained with the appropriate platform-dependent call.

### int erl_receive(fd, bufp, bufsize)

Types:

```
int fd;
char *bufp;
int bufsize;
```

This function receives a message consisting of a sequence of bytes in the Erlang external format.

`fd` is an open descriptor to an Erlang connection.

`bufp` is a buffer large enough to hold the expected message.

`bufsize` indicates the size of `bufp`.

If a *tick* occurs, i.e., the Erlang node on the other end of the connection has polled this node to see if it is still alive, the function will return `ERL_TICK` and no message will be placed in the buffer. Also, `erl_errno` will be set to `EAGAIN`.

On success, the message is placed in the specified buffer and the function returns the number of bytes actually read. On failure, the function returns a negative value and will set `erl_errno` to one of:

`EAGAIN`
: Temporary error: Try again.

`EMSGSIZE`
: Buffer too small.

`EIO`
: I/O error.

### int erl_receive_msg(fd, bufp, bufsize, emsg)

Types:

```
int fd;
unsigned char *bufp;
int bufsize;
ErlMessage *emsg;
```

This function receives the message into the specified buffer, and decodes into the `(ErlMessage *) emsg`.

`fd` is an open descriptor to an Erlang connection.

`bufp` is a buffer large enough to hold the expected message.

`bufsize` indicates the size of `bufp`.

emsg is a pointer to an ErlMessage structure, into which the message will be decoded. ErlMessage is defined as follows:

```
typedef struct {
  int type;
  ETERM *msg;
  ETERM *to;
  ETERM *from;
  char to_name[MAXREGLEN];
} ErlMessage;
```

> **Note:**
>
> The definition of ErlMessage has changed since earlier versions of Erl_Interface.

type identifies the type of message, one of ERL_SEND, ERL_REG_SEND, ERL_LINK, ERL_UNLINK and ERL_EXIT.

If type contains ERL_SEND this indicates that an ordinary send operation has taken place, and emsg->to contains the Pid of the recipient. If type contains ERL_REG_SEND then a registered send operation took place, and emsg->from contains the Pid of the sender. In both cases, the actual message will be in emsg->msg.

If type contains one of ERL_LINK or ERL_UNLINK, then emsg->to and emsg->from contain the pids of the sender and recipient of the link or unlink. emsg->msg is not used in these cases.

If type contains ERL_EXIT, then this indicates that a link has been broken. In this case, emsg->to and emsg->from contain the pids of the linked processes, and emsg->msg contains the reason for the exit.

> **Note:**
>
> It is the caller's responsibility to release the memory pointed to by emsg->msg, emsg->to and emsg->from.

If a *tick* occurs, i.e., the Erlang node on the other end of the connection has polled this node to see if it is still alive, the function will return ERL_TICK indicating that the tick has been received and responded to, but no message will be placed in the buffer. In this case you should call erl_receive_msg() again.

On success, the function returns ERL_MSG and the Emsg struct will be initialized as described above, or ERL_TICK, in which case no message is returned. On failure, the function returns ERL_ERROR and will set erl_errno to one of:

EMSGSIZE
: Buffer too small.

ENOMEM
: No more memory available.

EIO
: I/O error.

**int erl_xreceive_msg(fd, bufpp, bufsizep, emsg)**

Types:

```
int fd;
```

```
unsigned char **bufpp;
int *bufsizep;
ErlMessage *emsg;
```

This function is similar to `erl_receive_msg`. The difference is that `erl_xreceive_msg` expects the buffer to have been allocated by `malloc`, and reallocates it if the received message does not fit into the original buffer. For that reason, both buffer and buffer length are given as pointers - their values may change by the call.

On success, the function returns `ERL_MSG` and the `Emsg` struct will be initialized as described above, or `ERL_TICK`, in which case no message is returned. On failure, the function returns `ERL_ERROR` and will set `erl_errno` to one of:

`EMSGSIZE`
: Buffer too small.

`ENOMEM`
: No more memory available.

`EIO`
: I/O error.

**`int erl_send(fd, to, msg)`**

Types:

```
int fd;
ETERM *to, *msg;
```

This function sends an Erlang term to a process.

`fd` is an open descriptor to an Erlang connection.

`to` is an Erlang term containing the Pid of the intended recipient of the message.

`msg` is the Erlang term to be sent.

The function returns 1 if successful, otherwise 0 --- in which case it will set `erl_errno` to one of:

`EINVAL`
: Invalid argument: `to` is not a valid Erlang pid.

`ENOMEM`
: No more memory available.

`EIO`
: I/O error.

**`int erl_reg_send(fd, to, msg)`**

Types:

```
int fd;
char *to;
ETERM *msg;
```

This function sends an Erlang term to a registered process.

`fd` is an open descriptor to an Erlang connection.

`to` is a string containing the registered name of the intended recipient of the message.

`msg` is the Erlang term to be sent.

The function returns 1 if successful, otherwise 0 --- in which case it will set `erl_errno` to one of:

`ENOMEM`
: No more memory available.

EIO
I/O error.

```
ETERM *erl_rpc(fd, mod, fun, args)
int erl_rpc_to(fd, mod, fun, args)
int erl_rpc_from(fd, timeout, emsg)
```

Types:

```
int fd, timeout;
char *mod, *fun;
ETERM *args;
ErlMessage *emsg;
```

These functions support calling Erlang functions on remote nodes. `erl_rpc_to()` sends an rpc request to a remote node and `erl_rpc_from()` receives the results of such a call. `erl_rpc()` combines the functionality of these two functions by sending an rpc request and waiting for the results. See also `rpc:call/4`.

`fd` is an open descriptor to an Erlang connection.

`timeout` is the maximum time (in ms) to wait for results. Specify `ERL_NO_TIMEOUT` to wait forever. When erl_rpc() calls erl_rpc_from(), the call will never timeout.

`mod` is the name of the module containing the function to be run on the remote node.

`fun` is the name of the function to run.

`args` is an Erlang list, containing the arguments to be passed to the function.

`emsg` is a message containing the result of the function call.

The actual message returned by the rpc server is a 2-tuple `{rex,Reply}`. If you are using `erl_rpc_from()` in your code then this is the message you will need to parse. If you are using `erl_rpc()` then the tuple itself is parsed for you, and the message returned to your program is the erlang term containing `Reply` only. Replies to rpc requests are always ERL_SEND messages.

> **Note:**
>
> It is the caller's responsibility to free the returned `ETERM` structure as well as the memory pointed to by `emsg->msg` and `emsg->to`.

`erl_rpc()` returns the remote function's return value (or `NULL` if it failed). `erl_rpc_to()` returns 0 on success, and a negative number on failure. `erl_rcp_from()` returns `ERL_MSG` when successful (with `Emsg` now containing the reply tuple), and one of `ERL_TICK`, `ERL_TIMEOUT` and `ERL_ERROR` otherwise. When failing, all three functions set `erl_errno` to one of:

ENOMEM
No more memory available.
EIO
I/O error.
ETIMEDOUT
Timeout expired.
EAGAIN
Temporary error: Try again.

**int erl_publish(port)**

Types:

```
int port;
```

These functions are used by a server process to register with the local name server *epmd*, thereby allowing other processes to send messages by using the registered name. Before calling either of these functions, the process should have called `bind()` and `listen()` on an open socket.

`port` is the local name to register, and should be the same as the port number that was previously bound to the socket.

To unregister with epmd, simply close the returned descriptor.

On success, the functions return a descriptor connecting the calling process to epmd. On failure, they return -1 and set `erl_errno` to:

`EIO`
: I/O error

Additionally, `errno` values from `socket`*(2)* and `connect`*(2)* system calls may be propagated into `erl_errno`.

**int erl_accept(listensock, conp)**

Types:

```
int listensock;
ErlConnect *conp;
```

This function is used by a server process to accept a connection from a client process.

`listensock` is an open socket descriptor on which `listen()` has previously been called.

`conp` is a pointer to an `ErlConnect` struct, described as follows:

```
typedef struct {
  char ipadr[4];
  char nodename[MAXNODELEN];
} ErlConnect;
```

On success, `conp` is filled in with the address and node name of the connecting client and a file descriptor is returned. On failure, `ERL_ERROR` is returned and `erl_errno` is set to `EIO`.

**const char *erl_thiscookie()**
**const char *erl_thisnodename()**
**const char *erl_thishostname()**
**const char *erl_thisalivename()**
**short erl_thiscreation()**

These functions can be used to retrieve information about the C Node. These values are initially set with `erl_connect_init()` or `erl_connect_xinit()`.

**int erl_unpublish(alive)**

Types:

```
char *alive;
```

This function can be called by a process to unregister a specified node from epmd on the localhost. This is however usually not allowed, unless epmd was started with the -relaxed_command_check flag, which it normally isn't.

To unregister a node you have published, you should instead close the descriptor that was returned by `ei_publish()`.

> Warning:
>
> This function is deprecated and will be removed in a future release.

`alive` is the name of the node to unregister, i.e., the first component of the nodename, without the `@hostname`.

If the node was successfully unregistered from epmd, the function returns 0. Otherwise, it returns -1 and sets `erl_errno` is to `EIO`.

```
struct hostent *erl_gethostbyname(name)
struct hostent *erl_gethostbyaddr(addr, length, type)
struct hostent *erl_gethostbyname_r(name, hostp, buffer, buflen, h_errnop)
struct hostent *erl_gethostbyaddr_r(addr, length, type, hostp, buffer,
buflen, h_errnop)
```

Types:

```
const char *name;
const char *addr;
int length;
int type;
struct hostent *hostp;
char *buffer;
int buflen;
int *h_errnop;
```

These are convenience functions for some common name lookup functions.

## Debug Information

If a connection attempt fails, the following can be checked:

- `erl_errno`
- that the right cookie was used
- that *epmd* is running
- the remote Erlang node on the other side is running the same version of Erlang as the `erl_interface` library.

# erl_error

C Library

This module contains some error printing routines taken from *Advanced Programming in the UNIX Environment* by W. Richard Stevens.

These functions are all called in the same manner as `printf()`, i.e. with a string containing format specifiers followed by a list of corresponding arguments. All output from these functions is to `stderr`.

## Exports

**`void erl_err_msg(FormatStr, ... )`**

Types:

```
const char *FormatStr;
```

The message provided by the caller is printed. This function is simply a wrapper for `fprintf()`.

**`void erl_err_quit(FormatStr, ... )`**

Types:

```
const char *FormatStr;
```

Use this function when a fatal error has occurred that is not due to a system call. The message provided by the caller is printed and the process terminates with an exit value of 1. The function does not return.

**`void erl_err_ret(FormatStr, ... )`**

Types:

```
const char *FormatStr;
```

Use this function after a failed system call. The message provided by the caller is printed followed by a string describing the reason for failure.

**`void erl_err_sys(FormatStr, ... )`**

Types:

```
const char *FormatStr;
```

Use this function after a failed system call. The message provided by the caller is printed followed by a string describing the reason for failure, and the process terminates with an exit value of 1. The function does not return.

## Error Reporting

Most functions in erl_interface report failures to the caller by returning some otherwise meaningless value (typically `NULL` or a negative number). As this only tells you that things did not go well, you will have to examine the error code in `erl_errno` if you want to find out more about the failure.

## Exports

**`volatile int erl_errno`**

`erl_errno` is initially (at program startup) zero and is then set by many erl_interface functions on failure to a non-zero error code to indicate what kind of error it encountered. A successful function call might change `erl_errno`

(by calling some other function that fails), but no function will ever set it to zero. This means that you cannot use `erl_errno` to see *if* a function call failed. Instead, each function reports failure in its own way (usually by returning a negative number or `NULL`), in which case you can examine `erl_errno` for details.

`erl_errno` uses the error codes defined in your system's `<errno.h>`.

> ### Note:
>
> Actually, `erl_errno` is a "modifiable lvalue" (just like ISO C defines `errno` to be) rather than a variable. This means it might be implemented as a macro (expanding to, e.g., `*_erl_errno()`). For reasons of thread- (or task-)safety, this is exactly what we do on most platforms.

# erl_eterm

C Library

This module contains functions for creating and manipulating Erlang terms.

An Erlang term is represented by a C structure of type `ETERM`. Applications should not reference any fields in this structure directly, because it may be changed in future releases to provide faster and more compact term storage. Instead, applications should us the macros and functions provided.

The following macros each take a single ETERM pointer as an argument. They return a non-zero value if the test is true, and 0 otherwise:

`ERL_IS_INTEGER(t)`
: True if `t` is an integer.

`ERL_IS_UNSIGNED_INTEGER(t)`
: True if `t` is an integer.

`ERL_IS_FLOAT(t)`
: True if `t` is a floating point number.

`ERL_IS_ATOM(t)`
: True if `t` is an atom.

`ERL_IS_PID(t)`
: True if `t` is a Pid (process identifier).

`ERL_IS_PORT(t)`
: True if `t` is a port.

`ERL_IS_REF(t)`
: True if `t` is a reference.

`ERL_IS_TUPLE(t)`
: True if `t` is a tuple.

`ERL_IS_BINARY(t)`
: True if `t` is a binary.

`ERL_IS_LIST(t)`
: True if `t` is a list with zero or more elements.

`ERL_IS_EMPTY_LIST(t)`
: True if `t` is an empty list.

`ERL_IS_CONS(t)`
: True if `t` is a list with at least one element.

The following macros can be used for retrieving parts of Erlang terms. None of these do any type checking; results are undefined if you pass an ETERM* containing the wrong type. For example, passing a tuple to ERL_ATOM_PTR() will likely result in garbage.

`char *ERL_ATOM_PTR(t)`
`char *ERL_ATOM_PTR_UTF8(t)`
: A string representing atom `t`.

`int ERL_ATOM_SIZE(t)`
`int ERL_ATOM_SIZE_UTF8(t)`
: The length (in bytes) of atom t.

`void *ERL_BIN_PTR(t)`
: A pointer to the contents of `t`

`int ERL_BIN_SIZE(t)`
: The length (in bytes) of binary object `t`.

`int ERL_INT_VALUE(t)`
: The integer of `t`.

`unsigned int ERL_INT_UVALUE(t)`
The unsigned integer value of `t`.
`double ERL_FLOAT_VALUE(t)`
The floating point value of `t`.
`ETERM *ERL_PID_NODE(t)`
`ETERM *ERL_PID_NODE_UTF8(t)`
The Node in pid `t`.
`int ERL_PID_NUMBER(t)`
The sequence number in pid `t`.
`int ERL_PID_SERIAL(t)`
The serial number in pid `t`.
`int ERL_PID_CREATION(t)`
The creation number in pid `t`.
`int ERL_PORT_NUMBER(t)`
The sequence number in port `t`.
`int ERL_PORT_CREATION(t)`
The creation number in port `t`.
`ETERM *ERL_PORT_NODE(t)`
`ETERM *ERL_PORT_NODE_UTF8(t)`
The node in port `t`.
`int ERL_REF_NUMBER(t)`
The first part of the reference number in ref `t`. Use only for compatibility.
`int ERL_REF_NUMBERS(t)`
Pointer to the array of reference numbers in ref `t`.
`int ERL_REF_LEN(t)`
The number of used reference numbers in ref `t`.
`int ERL_REF_CREATION(t)`
The creation number in ref `t`.
`int ERL_TUPLE_SIZE(t)`
The number of elements in tuple `t`.
`ETERM *ERL_CONS_HEAD(t)`
The head element of list `t`.
`ETERM *ERL_CONS_TAIL(t)`
A List representing the tail elements of list `t`.

## Exports

**`ETERM *erl_cons(head, tail)`**

Types:

```
ETERM *head;
ETERM *tail;
```

This function concatenates two Erlang terms, prepending `head` onto `tail` and thereby creating a `cons` cell. To make a proper list, `tail` should always be a list or an empty list. Note that NULL is not a valid list.

`head` is the new term to be added.

`tail` is the existing list to which `head` will be concatenated.

The function returns a new list.

`ERL_CONS_HEAD(list)` and `ERL_CONS_TAIL(list)` can be used to retrieve the head and tail components from the list. `erl_hd(list)` and `erl_tl(list)` will do the same thing, but check that the argument really is a list.

For example:

```
ETERM *list,*anAtom,*anInt;
anAtom = erl_mk_atom("madonna");
anInt  = erl_mk_int(21);
list   = erl_mk_empty_list();
list   = erl_cons(anAtom, list);
list   = erl_cons(anInt, list);
 ... /* do some work */
erl_free_compound(list);
```

**ETERM *erl_copy_term(term)**

Types:

```
ETERM *term;
```

This function creates and returns a copy of the Erlang term `term`.

**ETERM *erl_element(position, tuple)**

Types:

```
int position;
ETERM *tuple;
```

This function extracts a specified element from an Erlang tuple.

`position` specifies which element to retrieve from `tuple`. The elements are numbered starting from 1.

`tuple` is an Erlang term containing at least `position` elements.

The function returns a new Erlang term corresponding to the requested element, or NULL if `position` was greater than the arity of `tuple`.

**void erl_init(NULL, 0)**

Types:

```
void *NULL;
int 0;
```

This function must be called before any of the others in the `erl_interface` library in order to initialize the library functions. The arguments must be specified as `erl_init(NULL,0)`.

**ETERM *erl_hd(list)**

Types:

```
ETERM *list;
```

Extracts the first element from a list.

`list` is an Erlang term containing a list.

The function returns an Erlang term corresponding to the head element in the list, or a NULL pointer if `list` was not a list.

**ETERM *erl_iolist_to_binary(term)**

Types:

**ETERM *list;**

This function converts an IO list to a binary term.

`list` is an Erlang term containing a list.

This function an Erlang binary term, or NULL if `list` was not an IO list.

Informally, an IO list is a deep list of characters and binaries which can be sent to an Erlang port. In BNF, an IO list is formally defined as follows:

```
iolist ::= []
        |   Binary
        |   [iohead | iolist]
        ;
iohead ::= Binary
        |   Byte (integer in the range [0..255])
        |   iolist
        ;
```

### char *erl_iolist_to_string(list)

Types:

**ETERM *list;**

This function converts an IO list to a '\0' terminated C string.

`list` is an Erlang term containing an IO list. The IO list must not contain the integer 0, since C strings may not contain this value except as a terminating marker.

This function returns a pointer to a dynamically allocated buffer containing a string. If `list` is not an IO list, or if `list` contains the integer 0, NULL is returned. It is the caller's responsibility free the allocated buffer with `erl_free()`.

Refer to `erl_iolist_to_binary()` for the definition of an IO list.

### int erl_iolist_length(list)

Types:

**ETERM *list;**

Returns the length of an IO list.

`list` is an Erlang term containing an IO list.

The function returns the length of `list`, or -1 if `list` is not an IO list.

Refer to `erl_iolist_to_binary()` for the definition of an IO list.

### int erl_length(list)

Types:

**ETERM *list;**

Determines the length of a proper list.

`list` is an Erlang term containing proper list. In a proper list, all tails except the last point to another list cell, and the last tail points to an empty list.

Returns -1 if `list` is not a proper list.

**ETERM *erl_mk_atom(string)**

Types:

```
const char *string;
```

Creates an atom.

`string` is the sequence of characters that will be used to create the atom.

Returns an Erlang term containing an atom. Note that it is the callers responsibility to make sure that `string` contains a valid name for an atom.

`ERL_ATOM_PTR(atom)` and `ERL_ATOM_PTR_UTF8(atom)` can be used to retrieve the atom name (as a null terminated string). `ERL_ATOM_SIZE(atom)` and `ERL_ATOM_SIZE_UTF8(atom)` returns the length of the atom name.

> Note:
>
> Note that the UTF8 variants were introduced in Erlang/OTP releases R16 and the string returned by `ERL_ATOM_PTR(atom)` was not null terminated on older releases.

**ETERM *erl_mk_binary(bptr, size)**

Types:

```
char *bptr;
int size;
```

This function produces an Erlang binary object from a buffer containing a sequence of bytes.

`bptr` is a pointer to a buffer containing data to be converted.

`size` indicates the length of `bptr`.

The function returns an Erlang binary object.

`ERL_BIN_PTR(bin)` retrieves a pointer to the binary data. `ERL_BIN_SIZE(bin)` retrieves the size.

**ETERM *erl_mk_empty_list()**

This function creates and returns an empty Erlang list. Note that NULL is not used to represent an empty list; Use this function instead.

**ETERM *erl_mk_estring(string, len)**

Types:

```
char *string;
int len;
```

This function creates a list from a sequence of bytes.

`string` is a buffer containing a sequence of bytes. The buffer does not need to be zero-terminated.

`len` is the length of `string`.

The function returns an Erlang list object corresponding to the character sequence in `string`.

**ETERM *erl_mk_float(f)**

Types:

```
double f;
```

Creates an Erlang float.

`f` is a value to be converted to an Erlang float.

The function returns an Erlang float object with the value specified in `f` or `NULL` if `f` is not finite.

`ERL_FLOAT_VALUE(t)` can be used to retrieve the value from an Erlang float.

**`ETERM *erl_mk_int(n)`**

Types:

```
int n;
```

Creates an Erlang integer.

`n` is a value to be converted to an Erlang integer.

The function returns an Erlang integer object with the value specified in `n`.

`ERL_INT_VALUE(t)` can be used to retrieve the value value from an Erlang integer.

**`ETERM *erl_mk_list(array, arrsize)`**

Types:

```
ETERM **array;
int arrsize;
```

Creates an Erlang list from an array of Erlang terms, such that each element in the list corresponds to one element in the array.

`array` is an array of Erlang terms.

`arrsize` is the number of elements in `array`.

The function creates an Erlang list object, whose length `arrsize` and whose elements are taken from the terms in `array`.

**`ETERM *erl_mk_pid(node, number, serial, creation)`**

Types:

```
const char *node;
unsigned int number;
unsigned int serial;
unsigned int creation;
```

This function creates an Erlang process identifier. The resulting pid can be used by Erlang processes wishing to communicate with the C node.

`node` is the name of the C node.

`number`, `serial` and `creation` are arbitrary numbers. Note though, that these are limited in precision, so only the low 15, 3 and 2 bits of these numbers are actually used.

The function returns an Erlang pid object.

`ERL_PID_NODE(pid)`, `ERL_PID_NUMBER(pid)`, `ERL_PID_SERIAL(pid)` and `ERL_PID_CREATION(pid)` can be used to retrieve the four values used to create the pid.

**ETERM *erl_mk_port(node, number, creation)**

Types:

```
const char *node;
unsigned int number;
unsigned int creation;
```

This function creates an Erlang port identifier.

`node` is the name of the C node.

`number` and `creation` are arbitrary numbers. Note though, that these are limited in precision, so only the low 18 and 2 bits of these numbers are actually used.

The function returns an Erlang port object.

`ERL_PORT_NODE(port)`, `ERL_PORT_NUMBER(port)` and `ERL_PORT_CREATION` can be used to retrieve the three values used to create the port.

**ETERM *erl_mk_ref(node, number, creation)**

Types:

```
const char *node;
unsigned int number;
unsigned int creation;
```

This function creates an old Erlang reference, with only 18 bits - use `erl_mk_long_ref` instead.

`node` is the name of the C node.

`number` should be chosen uniquely for each reference created for a given C node.

`creation` is an arbitrary number.

Note that `number` and `creation` are limited in precision, so only the low 18 and 2 bits of these numbers are actually used.

The function returns an Erlang reference object.

`ERL_REF_NODE(ref)`, `ERL_REF_NUMBER(ref)`, and `ERL_REF_CREATION(ref)` to retrieve the three values used to create the reference.

**ETERM *erl_mk_long_ref(node, n1, n2, n3, creation)**

Types:

```
const char *node;
unsigned int n1, n2, n3;
unsigned int creation;
```

This function creates an Erlang reference, with 82 bits.

`node` is the name of the C node.

`n1`, `n2` and `n3` can be seen as one big number `n1*2^64+n2*2^32+n3` which should be chosen uniquely for each reference created for a given C node.

`creation` is an arbitrary number.

Note that `n3` and `creation` are limited in precision, so only the low 18 and 2 bits of these numbers are actually used.

The function returns an Erlang reference object.

`ERL_REF_NODE(ref)`, `ERL_REF_NUMBERS(ref)`, `ERL_REF_LEN(ref)` and `ERL_REF_CREATION(ref)` to retrieve the values used to create the reference.

**`ETERM *erl_mk_string(string)`**

Types:

```
char *string;
```

This function creates a list from a zero terminated string.

`string` is the zero-terminated sequence of characters (i.e. a C string) from which the list will be created.

The function returns an Erlang list.

**`ETERM *erl_mk_tuple(array, arrsize)`**

Types:

```
ETERM **array;
int arrsize;
```

Creates an Erlang tuple from an array of Erlang terms.

`array` is an array of Erlang terms.

`arrsize` is the number of elements in `array`.

The function creates an Erlang tuple, whose arity is `size` and whose elements are taken from the terms in `array`.

To retrieve the size of a tuple, either use the `erl_size` function (which checks the type of the checked term and works for a binary as well as for a tuple), or the `ERL_TUPLE_SIZE(tuple)` returns the arity of a tuple. `erl_size()` will do the same thing, but it checks that the argument really is a tuple. `erl_element(index,tuple)` returns the element corresponding to a given position in the tuple.

**`ETERM *erl_mk_uint(n)`**

Types:

```
unsigned int n;
```

Creates an Erlang unsigned integer.

`n` is a value to be converted to an Erlang unsigned integer.

The function returns an Erlang unsigned integer object with the value specified in `n`.

`ERL_INT_UVALUE(t)` can be used to retrieve the value from an Erlang unsigned integer.

**`ETERM *erl_mk_var(name)`**

Types:

```
char *name;
```

This function creates an unbound Erlang variable. The variable can later be bound through pattern matching or assignment.

`name` specifies a name for the variable.

The function returns an Erlang variable object with the name `name`.

**`int erl_print_term(stream, term)`**

Types:

```
FILE *stream;
```

```
ETERM *term;
```

This function prints the specified Erlang term to the given output stream.

`stream` indicates where the function should send its output.

`term` is the Erlang term to print.

The function returns the number of characters written, or a negative value if there was an error.

### void erl_set_compat_rel(release_number)

Types:

```
unsigned release_number;
```

By default, the `erl_interface` library is only guaranteed to be compatible with other Erlang/OTP components from the same release as the `erl_interface` library itself. For example, `erl_interface` from the OTP R10 release is not compatible with an Erlang emulator from the OTP R9 release by default.

A call to `erl_set_compat_rel(release_number)` sets the `erl_interface` library in compatibility mode of release `release_number`. Valid range of `release_number` is [7, current release]. This makes it possible to communicate with Erlang/OTP components from earlier releases.

> **Note:**
>
> If this function is called, it may only be called once directly after the call to the *erl_init()* function.

> **Warning:**
>
> You may run into trouble if this feature is used carelessly. Always make sure that all communicating components are either from the same Erlang/OTP release, or from release X and release Y where all components from release Y are in compatibility mode of release X.

### int erl_size(term)

Types:

```
ETERM *term;
```

Returns the arity of an Erlang tuple, or the number of bytes in an Erlang binary object.

`term` is an Erlang tuple or an Erlang binary object.

The function returns the size of `term` as described above, or -1 if `term` is not one of the two supported types.

### ETERM *erl_tl(list)

Types:

```
ETERM *list;
```

Extracts the tail from a list.

`list` is an Erlang term containing a list.

The function returns an Erlang list corresponding to the original list minus the first element, or NULL pointer if `list` was not a list.

**ETERM *erl_var_content(term, name)**

Types:

```
ETERM *term;
char *name;
```

This function returns the contents of the specified variable in an Erlang term.

`term` is an Erlang term. In order for this function to succeed, `term` must be an Erlang variable with the specified name, or it must be an Erlang list or tuple containing a variable with the specified name. Other Erlang types cannot contain variables.

`name` is the name of an Erlang variable.

Returns the Erlang object corresponding to the value of `name` in `term`. If no variable with the name `name` was found in `term`, or if `term` is not a valid Erlang term, NULL is returned.

# erl_format

C Library

This module contains two routines - one general function for creating Erlang terms and one for pattern matching Erlang terms.

## Exports

**`ETERM *erl_format(FormatStr, ... )`**

Types:

```
char *FormatStr;
```

This is a general function for creating Erlang terms using a format specifier and a corresponding set of arguments, much in the way `printf()` works.

`FormatStr` is a format specification string. The set of valid format specifiers is as follows:

- ~i - Integer
- ~f - Floating point
- ~a - Atom
- ~s - String
- ~w - Arbitrary Erlang term

For each format specifier that appears in `FormatStr`, there must be a corresponding argument following `FormatStr`. An Erlang term is built according to the `FormatStr` with values and Erlang terms substituted from the corresponding arguments and according to the individual format specifiers. For example:

```
erl_format("[{name,~a},{age,~i},{data,~w}]",
           "madonna",
           21,
           erl_format("[{adr,~s,~i}]","E-street",42));
```

This will create an `(ETERM *)` structure corresponding to the Erlang term: `[{name,madonna},{age,21},{data,[{adr,"E-street",42}]}]`

The function returns an Erlang term, or NULL if `FormatStr` does not describe a valid Erlang term.

**`int erl_match(Pattern, Term)`**

Types:

```
ETERM *Pattern,*Term;
```

This function is used to perform pattern matching similar to that done in Erlang. Refer to an Erlang manual for matching rules and more examples.

`Pattern` is an Erlang term, possibly containing unbound variables.

`Term` is an Erlang term that we wish to match against `Pattern`.

`Term` and `Pattern` are compared, and any unbound variables in `Pattern` are bound to corresponding values in `Term`.

If `Term` and `Pattern` can be matched, the function returns a non-zero value and binds any unbound variables in `Pattern`. If `Term` `Pattern` do not match, the function returns 0. For example:

```
ETERM *term, *pattern, *pattern2;
term1    = erl_format("{14,21}");
term2    = erl_format("{19,19}");
pattern1 = erl_format("{A,B}");
pattern2 = erl_format("{F,F}");
if (erl_match(pattern1, term1)) {
  /* match succeeds:
   * A gets bound to 14,
   * B gets bound to 21
   */
  ...
}
if (erl_match(pattern2, term1)) {
  /* match fails because F cannot be
   * bound to two separate values, 14 and 21
   */
  ...
}
if (erl_match(pattern2, term2)) {
  /* match succeeds and F gets bound to 19 */
  ...
}
```

`erl_var_content()` can be used to retrieve the content of any variables bound as a result of a call to `erl_match()`.

# erl_global

C Library

This module provides support for registering, looking up and unregistering names in the Erlang Global module. For more information, see the description of Global in the reference manual.

Note that the functions below perform an RPC using an open file descriptor provided by the caller. This file descriptor must not be used for other traffic during the global operation or the function may receive unexpected data and fail.

## Exports

**`char **erl_global_names(fd,count)`**

Types:

```
int fd;
int *count;
```

Retrieve a list of all known global names.

`fd` is an open descriptor to an Erlang connection.

`count` is the address of an integer, or NULL. If `count` is not NULL, it will be set by the function to the number of names found.

On success, the function returns an array of strings, each containing a single registered name, and sets `count` to the number of names found. The array is terminated by a single NULL pointer. On failure, the function returns NULL and `count` is not modified.

> ### Note:
>
> It is the caller's responsibility to free the array afterwards. It has been allocated by the function with a single call to `malloc()`, so a single `free()` is all that is necessary.

**`int erl_global_register(fd,name,pid)`**

Types:

```
int fd;
const char *name;
ETERM *pid;
```

This function registers a name in Global.

`fd` is an open descriptor to an Erlang connection.

`name` is the name to register in Global.

`pid` is the pid that should be associated with `name`. This is the value that Global will return when processes request the location of `name`.

The function returns 0 on success, or -1 on failure.

**`int erl_global_unregister(fd,name)`**

Types:

```
int fd;
const char *name;
```

This function unregisters a name from Global.

`fd` is an open descriptor to an Erlang connection.

`name` is the name to unregister from Global.

The function returns 0 on success, or -1 on failure.

### ETERM *erl_global_whereis(fd,name,node)

Types:

```
int fd;
const char *name;
char *node;
```

`fd` is an open descriptor to an Erlang connection.

`name` is the name that is to be looked up in Global.

If `node` is not NULL, it is a pointer to a buffer where the function can fill in the name of the node where `name` is found. `node` can be passed directly to `erl_connect()` if necessary.

On success, the function returns an Erlang Pid containing the address of the given name, and node will be initialized to the nodename where `name` is found. On failure NULL will be returned and `node` will not be modified.

# erl_malloc

C Library

This module provides functions for allocating and deallocating memory.

## Exports

### ETERM *erl_alloc_eterm(etype)

Types:

```
unsigned char etype;
```

This function allocates an `(ETERM)` structure. Specify `etype` as one of the following constants:

- ERL_INTEGER
- ERL_U_INTEGER `/* unsigned integer */`
- ERL_ATOM
- ERL_PID `/* Erlang process identifier */`
- ERL_PORT
- ERL_REF `/* Erlang reference */`
- ERL_LIST
- ERL_EMPTY_LIST
- ERL_TUPLE
- ERL_BINARY
- ERL_FLOAT
- ERL_VARIABLE
- ERL_SMALL_BIG `/* bignum */`
- ERL_U_SMALL_BIG `/* bignum */`

`ERL_SMALL_BIG` and `ERL_U_SMALL_BIG` are for creating Erlang `bignums`, which can contain integers of arbitrary size. The size of an integer in Erlang is machine dependent, but in general any integer larger than 2^28 requires a bignum.

### void erl_eterm_release(void)

Clears the freelist, where blocks are placed when they are released by `erl_free_term()` and `erl_free_compound()`.

### void erl_eterm_statistics(allocated, freed)

Types:

```
long *allocated;
long *freed;
```

`allocated` and `freed` are initialized to contain information about the fix-allocator used to allocate ETERM components. `allocated` is the number of blocks currently allocated to ETERM objects. `freed` is the length of the freelist, where blocks are placed when they are released by `erl_free_term()` and `erl_free_compound()`.

### void erl_free_array(array, size)

Types:

```
ETERM **array;
int size;
```

This function frees an array of Erlang terms.

`array` is an array of ETERM* objects.

`size` is the number of terms in the array.

### void erl_free_term(t)

Types:

```
ETERM *t;
```

Use this function to free an Erlang term.

### void erl_free_compound(t)

Types:

```
ETERM *t;
```

Normally it is the programmer's responsibility to free each Erlang term that has been returned from any of the `erl_interface` functions. However since many of the functions that build new Erlang terms in fact share objects with other existing terms, it may be difficult for the programmer to maintain pointers to all such terms in order to free them individually.

`erl_free_compound()` will recursively free all of the sub-terms associated with a given Erlang term, regardless of whether we are still holding pointers to the sub-terms.

There is an example in the User Manual under "Building Terms and Patterns"

### void erl_malloc(size)

Types:

```
long size;
```

This function calls the standard `malloc()` function.

### void erl_free(ptr)

Types:

```
void *ptr;
```

This function calls the standard `free()` function.

# erl_marshal

C Library

This module contains functions for encoding Erlang terms into a sequence of bytes, and for decoding Erlang terms from a sequence of bytes.

## Exports

**`int erl_compare_ext(bufp1, bufp2)`**

Types:

```
unsigned char *bufp1,*bufp2;
```

This function compares two encoded terms.

`bufp1` is a buffer containing an encoded Erlang term term1.

`bufp2` is a buffer containing an encoded Erlang term term2.

The function returns 0 if the terms are equal, -1 if term1 is less than term2, or 1 if term2 is less than term1.

**`ETERM *erl_decode(bufp)`**

**`ETERM *erl_decode_buf(bufpp)`**

Types:

```
unsigned char *bufp;
unsigned char **bufpp;
```

`erl_decode()` and `erl_decode_buf()` decode the contents of a buffer and return the corresponding Erlang term. `erl_decode_buf()` provides a simple mechanism for dealing with several encoded terms stored consecutively in the buffer.

`bufp` is a pointer to a buffer containing one or more encoded Erlang terms.

`bufpp` is the address of a buffer pointer. The buffer contains one or more consecutively encoded Erlang terms. Following a successful call to `erl_decode_buf()`, `bufpp` will be updated so that it points to the next encoded term.

`erl_decode()` returns an Erlang term corresponding to the contents of `bufp` on success, or NULL on failure. `erl_decode_buf()` returns an Erlang term corresponding to the first of the consecutive terms in `bufpp` and moves `bufpp` forward to point to the next term in the buffer. On failure, each of the functions returns NULL.

**`int erl_encode(term, bufp)`**

**`int erl_encode_buf(term, bufpp)`**

Types:

```
ETERM *term;
unsigned char *bufp;
unsigned char **bufpp;
```

`erl_encode()` and `erl_encode_buf()` encode Erlang terms into external format for storage or transmission. `erl_encode_buf()` provides a simple mechanism for encoding several terms consecutively in the same buffer.

`term` is an Erlang term to be encoded.

`bufp` is a pointer to a buffer containing one or more encoded Erlang terms.

`bufpp` is a pointer to a pointer to a buffer containing one or more consecutively encoded Erlang terms. Following a successful call to `erl_encode_buf()`, `bufpp` will be updated so that it points to the position for the next encoded term.

These functions returns the number of bytes written to buffer if successful, otherwise returns 0.

Note that no bounds checking is done on the buffer. It is the caller's responsibility to make sure that the buffer is large enough to hold the encoded terms. You can either use a static buffer that is large enough to hold the terms you expect to need in your program, or use `erl_term_len()` to determine the exact requirements for a given term.

The following can help you estimate the buffer requirements for a term. Note that this information is implementation specific, and may change in future versions. If you are unsure, use `erl_term_len()`.

Erlang terms are encoded with a 1 byte tag that identifies the type of object, a 2- or 4-byte length field, and then the data itself. Specifically:

`Tuples`
: need 5 bytes, plus the space for each element.

`Lists`
: need 5 bytes, plus the space for each element, and 1 additional byte for the empty list at the end.

`Strings and atoms`
: need 3 bytes, plus 1 byte for each character (the terminating 0 is not encoded). Really long strings (more than 64k characters) are encoded as lists. Atoms cannot contain more than 256 characters.

`Integers`
: need 5 bytes.

`Characters`
: (integers < 256) need 2 bytes.

`Floating point numbers`
: need 32 bytes.

`Pids`
: need 10 bytes, plus the space for the node name, which is an atom.

`Ports and Refs`
: need 6 bytes, plus the space for the node name, which is an atom.

The total space required will be the result calculated from the information above, plus 1 additional byte for a version identifier.

### `int erl_ext_size(bufp)`

Types:

```
unsigned char *bufp;
```

This function returns the number of elements in an encoded term.

### `unsigned char erl_ext_type(bufp)`

Types:

```
unsigned char *bufp;
```

This function identifies and returns the type of Erlang term encoded in a buffer. It will skip a trailing *magic* identifier. Returns `0` if the type can't be determined or one of

- ERL_INTEGER
- ERL_ATOM
- ERL_PID `/* Erlang process identifier */`
- ERL_PORT
- ERL_REF `/* Erlang reference */`

- ERL_EMPTY_LIST
- ERL_LIST
- ERL_TUPLE
- ERL_FLOAT
- ERL_BINARY
- ERL_FUNCTION

### unsigned char *erl_peek_ext(bufp, pos)

Types:

```
unsigned char *bufp;
int pos;
```

This function is used for stepping over one or more encoded terms in a buffer, in order to directly access a later term.

`bufp` is a pointer to a buffer containing one or more encoded Erlang terms.

`pos` indicates how many terms to step over in the buffer.

The function returns a pointer to a sub-term that can be used in a subsequent call to `erl_decode()` in order to retrieve the term at that position. If there is no term, or `pos` would exceed the size of the terms in the buffer, NULL is returned.

### int erl_term_len(t)

Types:

```
ETERM *t;
```

This function determines the buffer space that would be needed by `t` if it were encoded into Erlang external format by `erl_encode()`.

The size in bytes is returned.

# erl_call

Command

`erl_call` makes it possible to start and/or communicate with a distributed Erlang node. It is built upon the `erl_interface` library as an example application. Its purpose is to use an Unix shell script to interact with a distributed Erlang node. It performs all communication with the Erlang *rex server*, using the standard Erlang RPC facility. It does not require any special software to be run at the Erlang target node.

The main use is to either start a distributed Erlang node or to make an ordinary function call. However, it is also possible to pipe an Erlang module to `erl_call` and have it compiled, or to pipe a sequence of Erlang expressions to be evaluated (similar to the Erlang shell).

Options, which cause `stdin` to be read, can be used with advantage as scripts from within (Unix) shell scripts. Another nice use of `erl_call` could be from (http) CGI-bin scripts.

## Exports

### `erl_call <options>`

Each option flag is described below with its name, type and meaning.

-a [Mod [Fun [Args]]]]

(*optional*): Applies the specified function and returns the result. `Mod` must be specified, however `start` and `[]` are assumed for unspecified `Fun` and `Args`, respectively. `Args` should be in the same format as for `erlang:apply/3`. Note that this flag takes exactly one argument, so quoting may be necessary in order to group `Mod`, `Fun` and `Args`, in a manner dependent on the behavior of your command shell.

-c Cookie

(*optional*): Use this option to specify a certain cookie. If no cookie is specified, the `~/.erlang.cookie` file is read and its content are used as cookie. The Erlang node we want to communicate with must have the same cookie.

-d

(*optional*): Debug mode. This causes all IO to be output to the file `~/.erl_call.out.Nodename`, where `Nodename` is the node name of the Erlang node in question.

-e

(*optional*): Reads a sequence of Erlang expressions, separated by ',' and ended with a '.', from `stdin` until EOF (Control-D). Evaluates the expressions and returns the result from the last expression. Returns `{ok,Result}` if successful.

-h HiddenName

(*optional*): Specifies the name of the hidden node that `erl_call` represents.

-m

(*optional*): Reads an Erlang module from `stdin` and compiles it.

-n Node

(one of `-n, -name, -sname` is required): Has the same meaning as `-name` and can still be used for backwards compatibility reasons.

-name Node

(one of `-n, -name, -sname` is required): `Node` is the name of the node to be started or communicated with. It is assumed that `Node` is started with `erl -name`, which means that fully qualified long node names are used. If the `-s` option is given, an Erlang node will (if necessary) be started with `erl -name`.

-q

(*optional*): Halts the Erlang node specified with the -n switch. This switch overrides the -s switch.

-r

(*optional*): Generates a random name of the hidden node that `erl_call` represents.

-s

(*optional*): Starts a distributed Erlang node if necessary. This means that in a sequence of calls, where the '`-s`' and '`-n Node`' are constant, only the first call will start the Erlang node. This makes the rest of the communication very fast. This flag is currently only available on the Unix platform.

-sname Node

(one of `-n, -name, -sname` is required): `Node` is the name of the node to be started or communicated with. It is assumed that `Node` is started with `erl -sname` which means that short node names are used. If `-s` option is given, an Erlang node will be started (if necessary) with `erl -sname`.

-v

(*optional*): Prints a lot of `verbose` information. This is only useful for the developer and maintainer of `erl_call`.

-x ErlScript

(*optional*): Specifies another name of the Erlang start-up script to be used. If not specified, the standard `erl` start-up script is used.

## Examples

Starts an Erlang node and calls `erlang:time/0`.

```
erl_call -s -a 'erlang time' -n madonna
{18,27,34}
```

Terminates an Erlang node by calling `erlang:halt/0`.

```
erl_call -s -a 'erlang halt' -n madonna
```

An apply with several arguments.

```
erl_call -s -a 'lists map [{math,sqrt},[1,4,9,16,25]]' -n madonna
```

Evaluates a couple of expressions. *The input ends with EOF (Control-D).*

```
erl_call -s -e -n madonna
statistics(runtime),
X=1,
Y=2,
{_,T}=statistics(runtime),
{X+Y,T}.
^D
{ok,{3,0}}
```

Compiles a module and runs it. *Again, the input ends with EOF (Control-D).* (In the example shown, the output has been formatted afterwards).

```
erl_call -s -m -a lolita -n madonna
-module(lolita).
-compile(export_all).
start() ->
        P = processes(),
        F = fun(X) -> {X,process_info(X,registered_name)} end,
        lists:map(F,[],P).
^D
[{<madonna@chivas.du.etx.ericsson.se,0,0>,
                  {registered_name,init}},
 {<madonna@chivas.du.etx.ericsson.se,2,0>,
                  {registered_name,erl_prim_loader}},
 {<madonna@chivas.du.etx.ericsson.se,4,0>,
                  {registered_name,error_logger}},
 {<madonna@chivas.du.etx.ericsson.se,5,0>,
                  {registered_name,application_controller}},
 {<madonna@chivas.du.etx.ericsson.se,6,0>,
                  {registered_name,kernel}},
 {<madonna@chivas.du.etx.ericsson.se,7,0>,
                  []},
 {<madonna@chivas.du.etx.ericsson.se,8,0>,
                  {registered_name,kernel_sup}},
 {<madonna@chivas.du.etx.ericsson.se,9,0>,
                  {registered_name,net_sup}},
 {<madonna@chivas.du.etx.ericsson.se,10,0>,
                  {registered_name,net_kernel}},
 {<madonna@chivas.du.etx.ericsson.se,11,0>,
                  []},
 {<madonna@chivas.du.etx.ericsson.se,12,0>,
                  {registered_name,global_name_server}},
 {<madonna@chivas.du.etx.ericsson.se,13,0>,
                  {registered_name,auth}},
 {<madonna@chivas.du.etx.ericsson.se,14,0>,
                  {registered_name,rex}},
 {<madonna@chivas.du.etx.ericsson.se,15,0>,
                  []},
 {<madonna@chivas.du.etx.ericsson.se,16,0>,
                  {registered_name,file_server}},
 {<madonna@chivas.du.etx.ericsson.se,17,0>,
                  {registered_name,code_server}},
 {<madonna@chivas.du.etx.ericsson.se,20,0>,
                  {registered_name,user}},
 {<madonna@chivas.du.etx.ericsson.se,38,0>,
                  []}]
```