

Erl_Interface

Copyright © 1998-2018 Ericsson AB. All Rights Reserved.
Erl_Interface 3.10.1
April 17, 2018

Copyright © 1998-2018 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..

April 17, 2018

1 Erl_Interface User's Guide

1.1 Erl_Interface User's Guide

1.1.1 Introduction

The `Erl_Interface` library contains functions that 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*

> **Note:**
>
> By default, the `Erl_Interface` libraries are only guaranteed to be compatible with other Erlang/OTP components from the same release as the libraries themselves. For information about how to communicate with Erlang/OTP components from earlier releases, see function *`ei:ei_set_compat_rel`* and *`erl_eterm:erl_set_compat_rel`*.

Scope

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 the Erlang Port Mapper Daemon (EPMD)
- Sending and receiving Erlang messages
- Remote procedure calls
- Using global names
- Using the registry

Prerequisites

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

1.1.2 Compiling and Linking Your Code

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. To find this, start Erlang and enter 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, ensure that your C compiler knows where to find `erl_interface.h` by specifying an appropriate `-I` argument on the command line, or add it to the `CFLAGS` definition in your `Makefile`. The correct value for this path is `$OTPROOT/lib/erl_interface-$EIVSN/include`, where:

- `$OTPROOT` is the path reported by `code:root_dir/0` in the example above.
- `$EIVSN` is the version of the `Erl_Interface` application, for example, `erl_interface-3.2.3`.

Compiling the code:

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

When linking:

- Specify the path to `liberl_interface.a` and `libei.a` with `-L$OTPROOT/lib/erl_interface-3.2.3/lib`.
- Specify the name of the libraries with `-lerl_interface -lei`.

Do this on the command line or add the flags to the `LDFLAGS` definition in your `Makefile`.

Linking the code:

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

On some systems it can be necessary to link with some more libraries (for example, `libnsl.a` and `libsocket.a` on Solaris, or `wsock32.lib` on Windows) to use the communication facilities of `Erl_Interface`.

If you use the `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. You must specify extra compiler flags to indicate which of the packages you use. Define `_REENTRANT` and either `STHREADS` or `PTHREADS`. The default is to use POSIX threads if `_REENTRANT` is specified.

1.1.3 Initializing the Erl_Interface Library

Before calling any of the other `Erl_Interface` functions, 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 are therefore to be specified as `erl_init(NULL,0)`.

1.1.4 Encoding, Decoding, and Sending Erlang Terms

Data sent between distributed Erlang nodes is encoded in the Erlang external format. You must therefore 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 several C functions that create and manipulate Erlang data structures. The library also contains an encode and a decode function. The following example 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.

For a complete description, see the following modules:

- *`erl_eterm`* for creating Erlang terms
- *`erl_marshal`* for encoding and decoding routines

1.1.5 Building Terms and Patterns

The previous example can be simplified by using the *`erl_format`* module to create an Erlang term:

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

For a complete description of the different format directives, see the *`erl_format`* module.

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 the 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 remove objects.

If you are unsure 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();
```

For more information, see the *erl_malloc* module.

1.1.6 Pattern Matching

An Erlang pattern is a term that can 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,_}");
```

The *erl_format:erl_match* function performs 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, a pattern is created with a variable Age, which is included at two positions in the tuple. The pattern match is performed as follows:

- erl_match binds the contents of Age to 21 the first time it reaches the variable.
- The second occurrence of Age causes a test for equality between the terms, as Age is already bound to 21. As Age is bound to 21, the equality test succeeds 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);
```

For more information, see the *erl_format:erl_match* function.

1.1.7 Connecting to a Distributed Erlang Node

To connect to a distributed Erlang node, you must first initialize the connection routine with *erl_connect:erl_connect_init*, which stores information, such as the hostname, 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);
```

For more information, see the *erl_connect* module.

After initialization, you set up the connection to the Erlang node. To specify the Erlang node you want to connect to, use erl_connect(). The following example sets up the connection and is to 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. For more information, see the *erl_error* module.

1.1.8 Using EPMD

erts:epmd is the Erlang Port Mapper Daemon. Distributed Erlang nodes register with epmd on the local host 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 to find 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 must 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. If it detects that the connection has been closed, the node becomes unregistered. So, if you explicitly close the descriptor or if your node fails, it becomes unregistered from epmd.

Notice that on some systems (such as VxWorks), a failed node is not detected by this mechanism, as 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 prevents you from registering a new node with the old name, as it thinks that the old name is still in use. In this case, you must unregister the name explicitly:

```
erl_unpublish(node);
```

This causes epmd to close the connection from the far end. Notice 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 can be consumed.

1.1.9 Sending and Receiving Erlang Messages

Use one of the following two functions to send messages:

- *erl_connect:erl_send*
- *erl_connect:erl_reg_send*

As in Erlang, messages can be sent to a pid or to a registered name. It is easier to send a message to a registered name, as it avoids the problem of finding a suitable pid.

Use one of the following two functions to receive messages:

- *erl_connect:erl_receive*
- *erl_connect: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. For more information about the primitives, see the *`erl_connect`* module.

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);
}
```

To provide robustness, a distributed Erlang node occasionally polls all its connected neighbors in an attempt to detect failed nodes or communication links. A node that 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` and `emsg.to` or `emsg.from`, depending on the type of message received.

For more information, see the *`erl_connect`* and *`erl_eterm`* modules.

1.1.10 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`.

For more information about `erl_rpc()` and its companions `erl_rpc_to()` and `erl_rpc_from()`, see the *`erl_connect`* module.

1.1.11 Using Global Names

A C node has access to names registered through the *`global`* module in Kernel. 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. 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:erl_global_names` allocates and returns a buffer containing all the names known to the `global` module in `Kernel`. `count` is initialized to indicate the number of names 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 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 the `global` module in `Kernel`, an Erlang pid is returned that can be used to send messages to the schedule service. Also, `node` is 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 cannot 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, use *`erl_connect:erl_accept`* to wait for incoming connections.

Note:

Remember to free `pid` later with *`erl_malloc:erl_free_term`*.

To unregister a name:

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

1.1.12 Using 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 an `Mnesia` table on an Erlang node. For more detailed information about the individual API functions, see the *`registry`* module.

Keys are strings, that is, `NULL`-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, 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 is important, then you are to 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. The following example shows 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 try to store an object in the registry and there is an existing object with the same key, the new value replaces 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 is 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 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 an object, you can ask:

```
struct ei_reg_stat buf;

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

Buf is 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 must 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);
```

This example back up the contents of the registry to the specified `Mnesia` table `"mtab"`. Once a registry has been backed up to `Mnesia` like this, more backups only affect objects that have been modified since the most recent backup, that is, objects that have been created, changed, or deleted. The backup operation is done as a single atomic transaction, so that either the entire backup is performed or none of it.

Likewise, a registry can be restored from a `Mnesia` table:

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

This reads the entire contents of `"mtab"` into the specified registry. After the restore, all the objects in the registry are marked as unmodified, so a later backup only affects objects that you have modified since the restore.

Notice that if you restore to a non-empty registry, objects in the table 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 some 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 are 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.

Notice that if you store binary objects that are context-dependent (for example, containing pointers or open file descriptors), they lose their meaning if they are backed up to a `Mnesia` table and later 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 occurs that likely causes 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 is not aware of your changes, possibly causing it to be missed the next time you make an `Mnesia` backup of the registry contents. This can be avoided if you mark the object as dirty after any such changes with *`registry:ei_reg_markdirty`*, or pass appropriate flags to *`registry:ei_reg_dump`*.

2 Reference Manual

ei

C Library

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

`ei` allows you to 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, no dynamic memory (except `ei_decode_fun()`) and is often quite fast.

`ei` 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 does not allow for multiple C-nodes. In general, `ei` is the preferred way of doing C-nodes.

The decode and encode functions use a buffer and 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 can crash.

All functions take two parameters:

- `buf` is a pointer to the buffer where the binary data is or will be.
- `index` is a pointer to an index into the buffer. This parameter is incremented with the size of the term decoded/encoded.

The data is thus at `buf[*index]` when an `ei` function is called.

All encode functions assume that the `buf` and `index` parameters point to a buffer large enough for the data. To get the size of an encoded term, without encoding it, pass `NULL` instead of a buffer pointer. Parameter `index` is incremented, but nothing will be encoded. This is the way in `ei` to "preflight" term encoding.

There are also encode functions that use 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` use a dynamic buffer.

All functions return `0` if successful, otherwise `-1` (for example, if a term is not of the expected type, or the data to decode is an invalid Erlang term).

Some of the decode functions need a pre-allocated buffer. This buffer must be allocated large enough, and for non-compound types the `ei_get_type()` function returns the size required (notice that for strings an extra byte is needed for the `NULL`-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. Latin-1 and UTF-8 are different extensions of 7-bit ASCII. All 7-bit ASCII characters are valid Latin-1 and UTF-8 characters. ASCII and Latin-1 both represent each character by one byte. An UTF-8 character can consist of 1-4 bytes. Notice that these constants are bit-flags and can be combined with bitwise OR.

Exports

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

Decodes an atom from the binary format. The NULL-terminated name of the atom is placed at p. At most MAXATOMLEN bytes can be 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)

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 (Latin-1 or UTF-8) can be obtained from *was. The encoding of the resulting string (7-bit ASCII, Latin-1, or UTF-8) can be obtained from *result. Both was and result can be NULL. *result can differ from want if want is a bitwise OR'd combination like ERLANG_LATIN1|ERLANG_UTF8 or if *result turns out to be pure 7-bit ASCII (compatible with both Latin-1 and UTF-8).

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

This function was introduced in Erlang/OTP R16 as part of a first step to support UTF-8 atoms.

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

Decodes an integer in the binary format to a GMP mpz_t integer. To use this function, the ei library must be configured and compiled to use the GMP library.

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

Decodes a binary from the binary format. Parameter len is set to the actual size of the binary. Notice that ei_decode_binary() assumes that there is enough room for the binary. The size required can be fetched by ei_get_type().

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

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)

Decodes a char (8-bit) integer between 0-255 from the binary format. For historical reasons the returned integer is of type char. Your C code is to consider the returned value to be of type unsigned char even if the C compilers and system can define char to be signed.

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

Decodes a double-precision (64-bit) floating point number from the binary format.

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

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 1 is returned, the index is incremented, and term contains the decoded term.

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

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

Decodes a fun from the binary format. Parameter `p` is to 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 is to be freed with `free_fun`. (This has to do with the arbitrary size of the environment for a fun.)

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

Decodes a list header from the binary format. The number of elements is returned in `arity`. The `arity+1` elements follow (the last one is the tail of the list, normally an empty list). If `arity` is `0`, it is an empty list.

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

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

Decodes a long integer from the binary format. If the code is 64 bits, the function `ei_decode_long()` is the same as `ei_decode_longlong()`.

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

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

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

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

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

Decodes a process identifier (pid) from the binary format.

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

Decodes a port identifier from the binary format.

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

Decodes a reference from the binary format.

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

Decodes a string from the binary format. A string in Erlang is a list of integers between 0 and 255. Notice that as 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 must add an extra byte to the memory requirement.

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

Decodes a term from the binary format. The term is return in `t` as a `ETERM*`, so `t` is actually an `ETERM**` (see *`erl_eterm`*). The term is later to be deallocated.

Notice that this function is located in the `Erl_Interface` library.

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)

Decodes a tuple header, the number of elements is returned in `arity`. The tuple elements follow in order in the buffer.

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

Decodes an unsigned long integer from the binary format. If the code is 64 bits, the function `ei_decode_ulong()` is the same as `ei_decode_ulonglong()`.

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

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

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

Decodes the version magic number for the Erlang binary term format. It must be the first token in a binary term.

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. Parameter `p` is the name of the atom in Latin-1 encoding. Only up to `MAXATOMLEN-1` bytes are encoded. The name is to be `NULL`-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. Parameter `p` is the name of the atom with character encoding *`from_enc`* (ASCII, Latin-1, or UTF-8). The name must either be `NULL`-terminated or a function variant with a `len` parameter must be used.

The encoding fails if `p` is not a valid string in encoding `from_enc`.

Argument `to_enc` is ignored. As from Erlang/OTP 20 the encoding is always done in UTF-8 which is readable by nodes as old as Erlang/OTP R16.

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 must be configured and compiled to use the GMP library.

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_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. For historical reasons the integer argument is of type `char`. Your C code is to consider the specified argument to be of type `unsigned char` even if the C compilers and system may define `char` to be signed.

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.

Returns `-1` if the floating point number is not finite.

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

Encodes an empty list. It is often used at the tail of a list.

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. Parameter `p` points to an `erlang_fun` structure. The `erlang_fun` is not freed automatically, the `free_fun` is to be called if the fun is not needed after encoding.

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

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 can contain another list or tuple.

For example, 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. Notice 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_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. If the code is 64 bits, the function `ei_encode_long()` is the same as `ei_encode_longlong()`.

`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. This function is missing in the VxWorks port.

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

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`.

For example, 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 cannot have duplicate keys.

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. Parameter `p` points to an `erlang_pid` structure (which should have been obtained earlier with `ei_decode_pid()`).

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. Parameter `p` 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. Parameter `p` points to a `erlang_ref` structure (which should have been obtained earlier with `ei_decode_ref()`).

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 is to be `NULL`-terminated, except for the `ei_x_encode_string_len()` function.

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

Encodes an `ETERM`, as obtained from `erl_interface`. Parameter `t` is actually an `ETERM` pointer. This function does not 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)

Encodes an Erlang trace token in the binary format. Parameter `p` 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)

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 can contain another tuple or list.

For example, 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_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. If the code is 64 bits, the function `ei_encode_ulong()` is the same as `ei_encode_ulonglong()`.

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. This function is missing in the VxWorks port.

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_get_type(const char *buf, const int *index, int *type, int *size)

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 `NULL`. 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_print_term(FILE* fp, const char* buf, int* index)
int ei_s_print_term(char s, const char* buf, int* index)**

Prints a term, in clear text, to the file specified 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()`, parameter `s` is to point to a dynamically (malloc) allocated string of `BUFSIZ` bytes or a `NULL` pointer. The string can be reallocated (and `*s` can be updated) by this function if the result is more than `BUFSIZ` characters. The string returned is `NULL`-terminated.

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

Argument `index` is updated, that is, this function can be viewed as a decode function that decodes a term into a human-readable format.

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 Erlang/OTP R10 is not compatible with an Erlang emulator from Erlang/OTP R9 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 can only be called once and must be called before any other functions in the `ei` library are called.

> **Warning:**
>
> You can run into trouble if this feature is used carelessly. Always ensure 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_skip_term(const char* buf, int* index)`

Skips a term in the specified buffer; 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: skip them and copy the binary term data to some buffer.

Returns `0` on success, otherwise `-1`.

`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)`

Appends data at the end of buffer `x`.

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

Formats a term, given as a string, to a buffer. Works like a sprintf for Erlang terms. `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 example, to encode a tuple with some stuff:

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

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

`int ei_x_free(ei_x_buff* x)`

Frees an `ei_x_buff` buffer. The memory used by the buffer is returned to the OS.

`int ei_x_new(ei_x_buff* x)`

`int ei_x_new_with_version(ei_x_buff* x)`

Allocates a new `ei_x_buff` buffer. The fields of the structure pointed to by parameter `x` is filled in, and a default buffer is allocated. `ei_x_new_with_version()` also puts an initial version byte, which is used in the binary format (so that `ei_x_encode_version()` will not be needed.)

Debug Information

Some tips on what to check when the emulator does not 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_eterm`

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 can communicate with it in a normal manner, but the node name is not shown in the listing provided by *erlang:nodes/0* in ERTS.

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

Time-Out Functions

Most functions appear in a version with the suffix _tmo appended to the function name. Those functions take an extra argument, a time-out 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 returns an error and erl_errno is set to ETIMEDOUT. With communication primitive is meant an operation on the socket, like connect, accept, recv, or send.

Clearly the time-outs are for implementing fault tolerance, not to keep hard real-time promises. The _tmo functions are for detecting non-responsive peers and to avoid blocking on socket operations.

A time-out value of 0 (zero) means that time-outs are disabled. Calling a _tmo function with the last argument as 0 is therefore the same thing as calling the function without the _tmo suffix.

As with all other functions starting with ei_, 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 time-out 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 handles 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

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

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

struct hostent *ei_gethostbyname(const char *name)

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

Convenience functions for some common name lookup functions.

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

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)

Equivalent to `ei_accept` with an optional time-out argument, see the description at the beginning of this manual page.

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

Sets 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 the latter case they set `erl_errno` to one of the following:

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

`ENOMEM`
: No more memory is available.

`EIO`
: I/O error.

Also, `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_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)

Initializes the `ec` structure, to identify the node name and cookie of the server. One of them must be called before other functions that works on the `ei_cnode` type or a file descriptor associated with a connection to another node is 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 are running on. If long names are to be used, they are to be fully qualified (that is, `durin.erix.ericsson.se` instead of `durin`).
- `thisalivename` is the registered name of the process.
- `thisnodename` is the full name of the node, that is, `einode@durin`.
- `thispaddr` if the IP address of the host.

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

A connection is closed by simply closing the socket. For information about how to close the socket gracefully (when there are outgoing packets before close), see the relevant system documentation.

These functions 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_tmo(ei_cnode* ec, char *nodename, unsigned timeout_ms)

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

Equivalent to `ei_connect` and `ei_xconnect` with an optional time-out argument, see the description at the beginning of this manual page.

int ei_get_tracelevel(void)
void ei_set_tracelevel(int level)

Used to set tracing on the distribution. The levels are different verbosity levels. A higher level means more information. See also section *Debug Information*.

These functions are not thread safe.

int ei_publish(ei_cnode *ec, int port)

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 is to 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 function returns a descriptor connecting the calling process to EPMD. On failure, `-1` is returned and `erl_errno` is set to `EIO`.

Also, `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)

Equivalent to `ei_publish` with an optional time-out argument, see the description at the beginning of this manual page.

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

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, that is, the Erlang node on the other end of the connection has polled this node to see if it is still alive, the function returns `ERL_TICK` and no message is placed in the buffer. Also, `erl_errno` is 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 sets `erl_errno` to one of the following:

`EAGAIN`
: Temporary error: Try again.

`EMSGSIZE`
: Buffer is too small.

`EIO`
: I/O error.

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 is to point to a memory area allocated by `malloc`. Argument `bufsz` is to 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`.

Returns either `ERL_TICK` or the `msgtype` field of the `erlang_msg *msg`. The length of the message is put in `*msglen`. On error a value `< 0` is returned.

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

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

Equivalent to `ei_receive_encoded` with an optional time-out argument, see the description at the beginning of this manual page.

`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)`

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 larger than the pre-allocated 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 functions return `ERL_MSG` and the `msg` struct is 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 the following:

`ERL_SEND`

Indicates that an ordinary send operation has occurred. `msg->to` contains the pid of the recipient (the C-node).

`ERL_REG_SEND`

A registered send operation occurred. `msg->from` contains the pid of the sender.

`ERL_LINK` or `ERL_UNLINK`

`msg->to` and `msg->from` contain the pids of the sender and recipient of the link or unlink.

`ERL_EXIT`

Indicates a broken link. `msg->to` and `msg->from` contain the pids of the linked processes.

The return value is the same as for *`ei_receive`*.

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)

Equivalent to `ei_receive_msg` and `ei_xreceive_msg` with an optional time-out argument, see the description at the beginning of this manual page.

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

Equivalent to `ei_receive` with an optional time-out argument, see the description at the beginning of this manual page.

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

Sends an Erlang term to a registered 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.

Returns `0` if successful, otherwise `-1`. In the latter case it sets `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)

Equivalent to `ei_reg_send` with an optional time-out argument, see the description at the beginning of this manual page.

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)

Supports 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`* in Kernel.

- `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 milliseconds) to wait for results. Specify `ERL_NO_TIMEOUT` to wait forever. `ei_rpc()` waits infinitely for the answer, that is, 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` is structure of type `erlang_msg` and contains information on the message received. For a description of the `erlang_msg` format, see *`ei_receive_msg`*.
- `x` points to the dynamic buffer that receives the result. For `ei_rpc()` this is the result without the version magic number. For `ei_rpc_from()` the result returns 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 the number of bytes, otherwise one of `ERL_TICK`, `ERL_TIMEOUT`, and `ERL_ERROR`. When failing, all three functions set `erl_errno` to one of the following:

`EIO`
: I/O error.

`ETIMEDOUT`
: Time-out 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();
```

`erlang_pid *ei_self(ei_cnode *ec)`

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.

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

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.

Returns `0` if successful, otherwise `-1`. In the latter case it sets `erl_errno` to `EIO`.

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

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

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

Equivalent to `ei_send_encoded` with an optional time-out argument, see the description at the beginning of this manual page.

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 `ei_cnode` as first argument, it takes a second argument, an `erlang_pid`, which is to 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)

Equivalent to `ei_send_reg_encoded` with an optional time-out argument, see the description at the beginning of this manual page.

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

Equivalent to `ei_send` with an optional time-out argument, see the description at the beginning of this manual page.

const char *ei_thisnodename(ei_cnode *ec)

const char *ei_thishostname(ei_cnode *ec)

const char *ei_thisalivename(ei_cnode *ec)

Can be used to retrieve information about the C-node. These values are initially set with `ei_connect_init()` or `ei_connect_xinit()`.

These function simply fetch 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.

int ei_unpublish(ei_cnode *ec)

Can be called by a process to unregister a specified node from EPMD on the local host. This is, however, usually not allowed, unless EPMD was started with flag `-relaxed_command_check`, which it normally is not.

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, `-1` is returned and `erl_errno` is set to `EIO`.

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

Equivalent to `ei_unpublish` with an optional time-out argument, see the description at the beginning of this manual page.

Debug Information

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

- `erl_errno`.
- That the correct cookie was used
- That EPMD is running
- That the remote Erlang node on the other side is running the same version of Erlang as the `ei` library
- That environment variable `ERL_EPMD_PORT` is set correctly

The connection attempt can be traced by setting a trace level by either using `ei_set_tracelevel` or by setting environment variable `EI_TRACELEVEL`. The trace levels have 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

`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.

Returns `0`.

`int ei_reg_delete(reg,key)`

Types:

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

Deletes an object from the registry. The object is not removed from the registry, it is only marked for later removal so that on later 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 is 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.

Returns `0` on success, otherwise `-1`.

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

Types:

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

Dumps the contents of a registry to a `Mnesia` table in an atomic manner, that is, either all data or no data is updated. 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 is to be placed. If the table does not exist, it is created automatically using configurable defaults. For information about configuring this behavior, see *`Mnesia`*.

If `flags` is `0`, the backup includes only those objects that have been created, modified, or deleted since the last backup or restore (that is, 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` causes a full backup to be done, and `EI_NOPURGE` causes the deleted objects to be left in the registry afterwards. These can be bitwise OR'ed together if both behaviors are desired. If `EI_NOPURGE` was specified, `ei_reg_purge()` can be used to explicitly remove the deleted items from the registry later.

Returns `0` on success, otherwise `-1`.

`double ei_reg_getfval(reg,key)`

Types:

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

Gets 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 is not found or if it is not a floating point object, `-1.0` is returned. To avoid problems with in-band error reporting (that is, if you cannot distinguish between `-1.0` and a valid result), use the more general function `ei_reg_getval()` instead.

`int ei_reg_getival(reg,key)`

Types:

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

Gets 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 is not found or if it is not an integer object, `-1` is returned. To avoid problems with in-band error reporting (that is, if you cannot distinguish between `-1` 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;
```

Gets 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` is 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 is not found or if it is not a binary object, `NULL` is returned. To avoid problems with in-band error reporting (that is, if you cannot distinguish between `NULL` 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;
```

Gets 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 is not found or if it is not a string, `NULL` is returned. To avoid problems with in-band error reporting (that is, 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)
```

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`, any kind of object is returned. If `flags` is `EI_INT`, `EI_FLT`, `EI_STR`, or `EI_BIN`, then only values of that kind are returned.

  The buffer pointed to by `v` must be large enough to hold the return data, that is, it must be a pointer to one of `int`, `double`, `char*`, or `void*`, respectively.

  If `flags` is `EI_BIN`, a fifth argument `int *size` is required, so that the size of the object can be returned.

On success, `v` (and `size` if the object is binary) is initialized with the value associated with `key`, and the function returns `EI_INT`, `EI_FLT`, `EI_STR`, or `EI_BIN`, indicating the type of object. On failure, `-1` is returned and the arguments are not updated.

int ei_reg_markdirty(reg,key)

Types:

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

Marks a registry object as dirty. This ensures that it is included in the next backup to `Mnesia`. Normally this operation is not necessary, as 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 is invisible to the registry and the object is 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.

Returns `0` on success, otherwise `-1`.

ei_reg *ei_reg_open(size)

Types:

```
int size;
```

Opens (creates) a registry, which initially is empty. To close the registry later, use `ei_reg_close()`.

`size` is the approximate number of objects you intend to store in the registry. As the registry uses a hash table with collision chaining, no absolute upper limit exists 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. To change the size later, use `ei_reg_resize()`. Notice that the number you provide is increased to the nearest larger prime number.

Returns an empty registry on success, otherwise `NULL`.

int ei_reg_purge(reg)

Types:

```
ei_reg *reg;
```

Removes all objects marked for deletion. When objects are deleted with `ei_reg_delete()` they are not removed from the registry, only marked for later removal. On a later backup to `Mnesia`, the objects can also be removed from the `Mnesia` table. If you are not backing up to `Mnesia`, you may wish to remove the objects manually with this function.

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

Returns `0` on success, otherwise `-1`.

int ei_reg_resize(reg,newsize)

Types:

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

Changes the size of a registry.

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

On success, the registry is resized, all contents rehashed, and `0` is returned. On failure, the registry is left unchanged and `-1` is returned.

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

Notice that only tables of a certain format can be restored, that is, those that have been created and backed up to with `ei_reg_dump()`. If the registry was not empty before the operation, 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 are overwritten with the new values. If the registry contains objects that were not in the table, they are unchanged by this operation.

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

Returns `0` on success, otherwise `-1`.

`int ei_reg_setfval(reg,key,f)`

Types:

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

Creates a key-value pair with the specified `key` and floating point value `f`. If an object already exists 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 is to be placed.
- `key` is the object name.
- `f` is the floating point value to assign.

Returns `0` on success, otherwise `-1`.

`int ei_reg_setival(reg,key,i)`

Types:

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

Creates a key-value pair with the specified `key` and integer value `i`. If an object already exists 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 is to be placed.
- `key` is the object name.
- `i` is the integer value to assign.

Returns `0` on success, otherwise `-1`.

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

Types:

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

Creates a key-value pair with the specified `key` whose "value" is the binary object pointed to by `p`. If an object already exists 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 is to be placed.
- `key` is the object name.
- `p` is a pointer to the binary object. The object itself must have been created through a single call to `malloc()` or a 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.

Returns `0` on success, otherwise `-1`.

int ei_reg_setsval(reg,key,s)

Types:

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

Creates a key-value pair with the specified `key` whose "value" is the specified string `s`. If an object already exists 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 is to be placed.
- `key` is the object name.
- `s` is the string to assign. The string itself must have been created through a single call to `malloc()` or similar a function, so that the registry can later delete it if necessary by calling `free()`.

Returns `0` on success, otherwise `-1`.

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

Types:

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

Creates a key-value pair with the specified `key` whose value is specified by `v`. If an object already exists 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 is to be placed.
- `key` is the object name.
- `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`, 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 is not transferred by an `ei_reg_dump()` operation, only the pointer value.

Returns `0` on success, otherwise `-1`.

int ei_reg_stat(reg,key,obuf)

Types:

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

Returns information about an object.

- `reg` is the registry containing the object.
- `key` is the object name.
- `obuf` is a pointer to an `ei_reg_stat` structure, defined as follows:

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

In `attr` the attributes of the object 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`).

Field `size` 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`.

Returns `0` and initializes `obuf` on success, otherwise `-1`.

`int ei_reg_tabstat(reg,obuf)`

Types:

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

Returns 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 as follows:

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

Field `size` 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 number.

- `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, `0` is returned and `obuf` is initialized to contain table statistics, otherwise `-1` is returned.

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 can communicate with it in a normal manner, but the node name does not appear in the listing provided by *erlang:nodes/0* in ERTS.

Exports

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.

int erl_close_connection(fd)

Types:

```
int fd;
```

Closes an open connection to an Erlang node.

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

Returns 0 on success. 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_connect(node)
int erl_xconnect(addr, alive)

Types:

```
char *node, *alive;
struct in_addr *addr;
```

Sets up a connection to an Erlang node.

erl_xconnect() requires the IP address of the remote host and the alivename 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.

Returns an open file descriptor on success, otherwise a negative value. In the latter case `erl_errno` is set to one of:

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

`ENOMEM`
: No more memory is available.

`EIO`
: I/O error.

Also, `errno` values from `socket(2)` and `connect(2)` system calls can 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 ***/
erl_connect( NODE );

/*** Variant 2 ***/
struct in_addr addr;
addr = inet_addr(IP_ADDR);
erl_xconnect( &addr , ALIVE );
```

`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;
```

Initializes the `erl_connect` module. In particular, these functions 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:

- Hostname of the node, `host`
- Alivename, `alive`
- Node name, `node`
- IP address, `addr`
- Cookie, `cookie`
- Creation number, `creation`

`erl_connect_init()` provides an alternative interface that 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, that is, it will not be fully qualified. If you need to use fully qualified (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 node name. It is to 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 environment variable `HOME` on Unix and from the `HOMEDRIVE` and `HOMEPATH` variables on Windows. For more details, see the *`auth`* module in Kernel.
- `creation` helps identifying 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 is assigned a creation number when it calls `erl_publish()`.

`number` is used by `erl_connect_init()` to construct the actual node name. In Example 2 below, "**c17@a.DNS.name**" is 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_publish(port)`

Types:

```
int port;
```

This function is 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 this function, the process should have called `bind()` and `listen()` on an open socket.

`port` is the local name to register, and is to 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, a descriptor connecting the calling process to EPMD is returned. On failure, `-1` is returned and `erl_errno` is set to:

`EIO`
: I/O error.

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

`int erl_receive(fd, bufp, bufsize)`

Types:

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

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, that is, the Erlang node on the other end of the connection has polled this node to see if it is still alive, the function returns `ERL_TICK` and no message is placed in the buffer. Also, `erl_errno` is 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 sets `erl_errno` to one of:

`EAGAIN`
: Temporary error: Try again.

`EMSGSIZE`
: Buffer is too small.

`EIO`
: I/O error.

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

Types:

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

Receives the message into the specified buffer and decodes into `(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 the following:

`ERL_SEND`
: An ordinary send operation has occurred and `emsg->to` contains the pid of the recipient. The message is in `emsg->msg`.

`ERL_REG_SEND`
: A registered send operation has occurred and `emsg->from` contains the pid of the sender. The message is in `emsg->msg`.

ERL_LINK or ERL_UNLINK

emsg->to and emsg->from contain the pids of the sender and recipient of the link or unlink. emsg->msg is not used.

ERL_EXIT

A link is broken. 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, that is, the Erlang node on the other end of the connection has polled this node to see if it is still alive, the function returns ERL_TICK indicating that the tick has been received and responded to, but no message is placed in the buffer. In this case you are to call erl_receive_msg() again.

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

EMSGSIZE
Buffer is too small.

ENOMEM
No more memory is available.

EIO
I/O error.

int erl_reg_send(fd, to, msg)

Types:

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

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.

Returns 1 on success, otherwise 0. In the latter case erl_errno is set to one of:

ENOMEM
No more memory is available.

EIO
I/O error.

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

Types:

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

```
ErlMessage *emsg;
```

Supports 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`* in `Kernel`.

- `fd` is an open descriptor to an Erlang connection.
- `timeout` is the maximum time (in milliseconds) to wait for results. To wait forever, specify `ERL_NO_TIMEOUT`. 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 use `erl_rpc_from()` in your code, this is the message you will need to parse. If you use `erl_rpc()`, 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 and the memory pointed to by `emsg->msg` and `emsg->to`.

`erl_rpc()` returns the remote function's return value on success, otherwise `NULL`.

`erl_rpc_to()` returns `0` on success, otherwise a negative number.

`erl_rcp_from()` returns `ERL_MSG` on success (with `Emsg` now containing the reply tuple), otherwise one of `ERL_TICK`, `ERL_TIMEOUT`, or `ERL_ERROR`.

When failing, all three functions set `erl_errno` to one of:

`ENOMEM`
: No more memory is available.

`EIO`
: I/O error.

`ETIMEDOUT`
: Timeout has expired.

`EAGAIN`
: Temporary error: Try again.

`int erl_send(fd, to, msg)`

Types:

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

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.

Returns `1` on success, otherwise `0`. In the latter case `erl_errno` is set to one of:

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

ENOMEM
: No more memory is available.

EIO
: I/O error.

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

Retrieves 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 local host. This is, however, usually not allowed, unless EPMD was started with flag `-relaxed_command_check`, which it normally is not.

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, that is, the first component of the node name, without `@hostname`.

If the node was successfully unregistered from EPMD, `0` is returned, otherwise `-1` is returned and `erl_errno` is set to `EIO`.

int erl_xreceive_msg(fd, bufpp, bufsizep, emsg)

Types:

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

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. Therefore both buffer and buffer length are given as pointers; their values can change by the call.

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

EMSGSIZE
: Buffer is too small.

ENOMEM
: No more memory is available.

`EIO`
I/O error.

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

Convenience functions for some common name lookup functions.

Debug Information

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

- `erl_errno`
- That the correct cookie was used
- That EPMD is running
- That 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()`, that is, 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 because of a system call. The message provided by the caller is printed and the process terminates with exit value `1`. This 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 exit value `1`. This 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, 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 can change `erl_errno`

(by calling some other function that fails), but no function does never 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:**
>
> `erl_errno` is a "modifiable lvalue" (just like ISO C defines `errno` to be) rather than a variable. This means it can be implemented as a macro (expanding to, for example, `*_erl_errno()`). For reasons of thread safety (or task safety), this is exactly what we do on most platforms.

erl_eterm

C Library

This module provides 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, as it can be changed in future releases to provide faster and more compact term storage. Instead, applications should use the macros and functions provided.

Each of the following macros takes a single `ETERM` pointer as an argument. The macros return a non-zero value if the test is true, otherwise `0`.

`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()` likely results 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;
```

Concatenates two Erlang terms, prepending `head` onto `tail` and thereby creating a `cons` cell. To make a proper list, `tail` is always to be a list or an empty list. Notice that `NULL` is not a valid list.

- `head` is the new term to be added.
- `tail` is the existing list to which `head` is 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)` do the same thing, but check that the argument really is a list.

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;
```

Creates and returns a copy of the Erlang term `term`.

ETERM *erl_element(position, tuple)

Types:

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

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.

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

ETERM *erl_hd(list)

Types:

```
ETERM *list;
```

Extracts the first element from a list.

`list` is an Erlang term containing a list.

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

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 to initialize the library functions. The arguments must be specified as `erl_init(NULL,0)`.

int erl_iolist_length(list)

Types:

```
ETERM *list;
```

Returns the length of an I/O list.

`list` is an Erlang term containing an I/O list.

Returns the length of `list`, or `-1` if `list` is not an I/O list.

For the definition of an I/O list, see *erl_iolist_to_binary*.

ETERM *erl_iolist_to_binary(term)

Types:

```
ETERM *list;
```

Converts an I/O list to a binary term.

`list` is an Erlang term containing a list.

Returns an Erlang binary term, or `NULL` if `list` was not an I/O list.

Informally, an I/O list is a deep list of characters and binaries that can be sent to an Erlang port. In BNF, an I/O 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;
```

Converts an I/O list to a `NULL`-terminated C string.

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

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

For the definition of an I/O list, see *erl_iolist_to_binary*.

int erl_length(list)

Types:

```
ETERM *list;
```

Determines the length of a proper list.

`list` is an Erlang term containing a 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. Notice that it is the caller's responsibility to ensure 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)` return the length of the atom name.

> **Note:**
>
> The UTF-8 variants were introduced in Erlang/OTP 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;
```

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`.

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()`

Creates and returns an empty Erlang list. Notice 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;
```

Creates a list from a sequence of bytes.

- `string` is a buffer containing a sequence of bytes. The buffer does not need to be `NULL`-terminated.
- `len` is the length of `string`.

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.

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.

Returns an Erlang integer object with the value specified in `n`.

`ERL_INT_VALUE(t)` can be used to retrieve the 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_long_ref(node, n1, n2, n3, creation)`

Types:

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

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 is to be chosen uniquely for each reference created for a given C-node.
- `creation` is an arbitrary number.

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

Returns an Erlang reference object.

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

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

Types:

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

Creates an Erlang process identifier (pid). 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. Notice that these are limited in precision, so only the low 15, 3, and 2 bits of these numbers are used.

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;
```

Creates an Erlang port identifier.

- `node` is the name of the C-node.
- `number` and `creation` are arbitrary numbers. Notice that these are limited in precision, so only the low 18 and 2 bits of these numbers are used.

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;
```

Creates an old Erlang reference, with only 18 bits - use `erl_mk_long_ref` instead.

- `node` is the name of the C-node.
- `number` is to be chosen uniquely for each reference created for a given C-node.
- `creation` is an arbitrary number.

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

Returns an Erlang reference object.

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

ETERM *erl_mk_string(string)

Types:

```
char *string;
```

Creates a list from a `NULL`-terminated string.

`string` is a `NULL`-terminated sequence of characters (that is, a C string) from which the list will be created.

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 function `erl_size` (which checks the type of the checked term and works for a binary as well as for a tuple) or `ERL_TUPLE_SIZE(tuple)` returns the arity of a tuple. `erl_size()` does the same thing, but it checks that the argument 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.

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;
```

Creates an unbound Erlang variable. The variable can later be bound through pattern matching or assignment.

`name` specifies a name for the variable.

Returns an Erlang variable object with the name `name`.

`int erl_print_term(stream, term)`

Types:

```
FILE *stream;
ETERM *term;
```

Prints the specified Erlang term to the specified output stream.

- `stream` indicates where the function is to send its output.
- `term` is the Erlang term to print.

Returns the number of characters written on success, otherwise a negative value.

`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 Erlang/OTP R10 is not compatible with an Erlang emulator from Erlang/OTP R9 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 function *erl_init()*.

Warning:

You may run into trouble if this feature is used carelessly. Always ensure 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 either 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.

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.

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;
```

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 either 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` is 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;
```

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 valid format specifiers are as follows:

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

For each format specifier included in `FormatStr`, there must be a corresponding argument following `FormatStr`. An Erlang term is built according to `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 creates 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. For matching rules and more examples, see section *Pattern Matching* in the Erlang Reference Manual.

- `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` and `Pattern` do not match, `0` is returned. 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 `global` module. For more information, see *`kernel:global`*.

Notice 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, as the function can then receive unexpected data and fail.

Exports

`char **erl_global_names(fd,count)`

Types:

```
int fd;
int *count;
```

Retrieves 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 is 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;
```

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 is to be associated with `name`. This value is returned by `global` when processes request the location of `name`.

Returns `0` on success, otherwise `-1`.

`int erl_global_unregister(fd,name)`

Types:

```
int fd;
const char *name;
```

Unregisters a name from `global`.

- `fd` is an open descriptor to an Erlang connection.
- `name` is the name to unregister from `global`.

Returns `0` on success, otherwise `-1`.

`ETERM *erl_global_whereis(fd,name,node)`

Types:

```
int fd;
const char *name;
char *node;
```

Looks up a name in `global`.

- `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 specified name, and the node is initialized to the node name where `name` is found. On failure, `NULL` is returned and `node` is not modified.

erl_malloc

C Library

This module provides functions for allocating and deallocating memory.

Exports

ETERM *erl_alloc_eterm(etype)

Types:

```
unsigned char etype;
```

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 any size. The size of an integer in Erlang is machine-dependent, but any integer > 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;
```

Reports term allocation statistics.

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(ptr)`

Types:

```
void *ptr;
```

Calls the standard `free()` function.

`void erl_free_array(array, size)`

Types:

```
ETERM **array;
int size;
```

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_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, as many of the functions that build new Erlang terms in fact share objects with other existing terms, it can be difficult for the programmer to maintain pointers to all such terms to free them individually.

`erl_free_compound()` recursively frees all of the subterms associated with a specified Erlang term, regardless of whether we are still holding pointers to the subterms.

For an example, see section *Building Terms and Patterns* in the User's Guide.

`void erl_free_term(t)`

Types:

```
ETERM *t;
```

Frees an Erlang term.

`void erl_malloc(size)`

Types:

```
long size;
```

Calls the standard `malloc()` 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;
```

Compares two encoded terms.

- `bufp1` is a buffer containing an encoded Erlang term term1.
- `bufp2` is a buffer containing an encoded Erlang term term2.

Returns `0` if the terms are equal, `-1` if `term1` < `term2`, or `1` if `term2` < `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` is updated so that it points to the next encoded term.

`erl_decode()` returns an Erlang term corresponding to the contents of `bufp` on success, otherwise `NULL`. `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 return `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` is updated so that it points to the position for the next encoded term.

These functions return the number of bytes written to buffer on success, otherwise `0`.

Notice that no bounds checking is done on the buffer. It is the caller's responsibility to ensure 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. Notice that this information is implementation-specific, and can 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 more 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 is the result calculated from the information above, plus 1 more byte for a version identifier.

`int erl_ext_size(bufp)`

Types:

```
unsigned char *bufp;
```

Returns the number of elements in an encoded term.

`unsigned char erl_ext_type(bufp)`

Types:

```
unsigned char *bufp;
```

Identifies and returns the type of Erlang term encoded in a buffer. It skips a trailing **magic** identifier.

Returns `0` if the type cannot 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, to directly access 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.

Returns a pointer to a subterm that can be used in a later call to erl_decode() 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;
```

Determines the buffer space that would be needed by t if it were encoded into Erlang external format by erl_encode().

Returns the size in bytes.

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 a 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>`

Starts/calls Erlang.

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` is to be in the same format as for *`erlang:apply/3`* in `ERTS`.

Notice that this flag takes exactly one argument, so quoting can be necessary 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 is used as cookie. The Erlang node we want to communicate with must have the same cookie.

`-d`

(**Optional.**) Debug mode. This causes all I/O to be output to the `~/.erl_call.out.Nodename` file, where `Nodename` is the node name of the Erlang node in question.

`-e`

(**Optional.**) Reads a sequence of Erlang expressions, separated by comma (,) and ended with a full stop (.), from `stdin` until EOF (Control-D). Evaluates the expressions and returns the result from the last expression. Returns `{ok,Result}` on success.

`-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 backward 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 option `-s` is specified, an Erlang node will (if necessary) be started with `erl -name`.

`-q`

(**Optional.**) Halts the Erlang node specified with switch `-n`. This switch overrides switch `-s`.

`-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 '`-s`' and '`-n Node`' are constant, only the first call starts the Erlang node. This makes the rest of the communication very fast. This flag is currently only available on Unix-like platforms (Linux, Mac OS X, Solaris, and so on).

`-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 option `-s` is specified, an Erlang node is 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 startup script to be used. If not specified, the standard `erl` startup script is used.

Examples

To start an Erlang node and call `erlang:time/0`:

```
erl_call -s -a 'erlang time' -n madonna
{18,27,34}
```

To terminate an Erlang node by calling `erlang:halt/0`:

```
erl_call -s -a 'erlang halt' -n madonna
```

To apply with many arguments:

```
erl_call -s -a 'lists seq [1,10]' -n madonna
```

To evaluate some 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}}
```

To compile a module and run it (**again, the input ends with EOF (Control-D)**):

(In the example, the output has been formatted afterwards.)

```
erl_call -s -m -a procnames -n madonna
-module(procnames).
-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>,
                  []}]
```