Erl_Interface

Copyright © 1998-2026 Ericsson AB. All Rights Reserved.

Erl_Interface 5.5.1

July 10, 2026

Copyright © 1998-2026 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..

July 10, 2026

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` library is 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_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

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 line in your code:

```
#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 `ei.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 `libei.a` with `-L$OTPROOT/lib/erl_interface-3.2.3/lib`.
- Specify the name of the library with `-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 -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 Library

Before calling any of the other functions in the library, initialize it by calling `ei_init()` exactly once.

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 following example shows how to create and encode an Erlang tuple `{tobbe,3928}`:

```
ei_x_buff buf;
ei_x_new(&buf);
int i = 0;
ei_x_encode_tuple_header(&buf, 2);
ei_x_encode_atom(&buf, "tobbe");
ei_x_encode_long(&buf, 3928);
```

For a complete description, see the `ei` module.

1.1.5 Building Terms

The previous example can be simplified by using the `ei_x_format_wo_ver` function to create an Erlang term:

```
ei_x_buff buf;
ei_x_new(&buf);
ei_x_format_wo_ver(&buf, "{~a,~i}", "tobbe", 3928);
```

For a complete description of the different format directives, see the the `ei_x_format_wo_ver` function.

The following example is more complex:

```
ei_x_buff buf;
int i = 0;
ei_x_new(&buf);
ei_x_format_wo_ver(&buf,
                   "[{name,~a},{age,~i},{data,[{adr,~s,~i}]}]",
                   "madonna",
                   21,
                  "E-street", 42);
ei_print_term(stdout, buf.buff, &i);
ei_x_free(&buf);
```

As in the previous examples, it is your responsibility to free the memory allocated for Erlang terms. In this example, `ei_x_free()` ensures that the data pointed to by `buf` is released.

1.1.6 Connecting to a Distributed Erlang Node

To connect to a distributed Erlang node, you must first initialize the connection routine with one of the `ei_connect_init_*` functions, which stores information, such as the hostname, and node name for later use:

```
int identification_number = 99;
int creation=1;
char *cookie="a secret cookie string"; /* An example */
const char* node_name = "einode@durin";
const char *cookie = NULL;
short creation = time(NULL) + 1;
ei_cnode ec;
ei_connect_init(&ec,
                node_name,
                cookie,
                creation);
```

For more information, see the `ei_connect` module.

After initialization, you set up the connection to the Erlang node. To specify the Erlang node you want to connect to, use the `ei_connect_*()` family of functions. The following example sets up the connection and is to result in a valid socket file descriptor:

```
int sockfd;
const char* node_name = "einode@durin"; /* An example */
if ((sockfd = ei_connect(&ec, nodename)) < 0)
  fprintf(stderr, "ERROR: ei_connect failed");
```

1.1.7 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 `ei_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 = ei_publish(&ec, 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 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 close the port explicitly

1.1.8 Sending and Receiving Erlang Messages

Use one of the following two functions to send messages:

- `ei_send`
- `ei_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:

- `ei_receive`
- `ei_receive_msg`

Example of Sending Messages

In the following example, `{Pid, hello_world}` is sent to a registered process `my_server`:

```
ei_x_buff buf;
ei_x_new_with_version(&buf);

ei_x_encode_tuple_header(&buf, 2);
ei_x_encode_pid(&buf, ei_self(ec));
ei_x_encode_atom(&buf, "Hello world");

ei_reg_send(&ec, fd, "my_server", buf.buff, buf.index);
```

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 `ei_connect` module.

Example of Receiving Messages

In this example, `{Pid, Something}` is received.

```
erlang_msg msg;
int index = 0;
int version;
int arity = 0;
erlang_pid pid;
ei_x_buff buf;
ei_x_new(&buf);
for (;;) {
  int got = ei_xreceive_msg(fd, &msg, &x);
  if (got == ERL_TICK)
    continue;
  if (got == ERL_ERROR) {
    fprintf(stderr, "ei_xreceive_msg, got==%d", got);
    exit(1);
  }
  break;
}
ei_decode_version(buf.buff, &index, &version);
ei_decode_tuple_header(buf.buff, &index, &arity);
if (arity != 2) {
  fprintf(stderr, "got wrong message");
  exit(1);
}
ei_decode_pid(buf.buff, &index, &pid);
```

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 `ei_xreceive_msg()`. However, when this has occurred, `ei_xreceive_msg` returns `ERL_TICK` to the caller without storing a message into the `erlang_msg` structure.

When a message has been received, it is the caller's responsibility to free the received message.

For more information, see the `ei_connect` and `ei` modules.

1.1.9 Remote Procedure Calls

An Erlang node acting as a client to another Erlang node typically sends a request and waits for a reply. Such a request is included in a function call at a remote node and is called a remote procedure call.

The following example checks if a specific 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();
```

For more information about `ei_rpc()` and its companions `ei_rpc_to()` and `ei_rpc_from()`, see the `ei_connect` module.

1.1.10 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 = ei_global_names(&ec,fd,&count);

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

free(names);
```

`ei_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. `ei_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];
erlang_pid the_pid;

if (ei_global_whereis(&ec,fd,"schedule",&the_pid,node) < 0)
   fprintf(stderr, "ei_global_whereis error\n");
```

If `"schedule"` is known to the `global` module in `Kernel`, an Erlang pid is written to the_pid. This pid 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 `ei_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 name that Erlang processes can use to communicate with your service:

```
ei_global_register(fd,servicename,ei_self(ec));
```

After registering the name, use `ei_accept` to wait for incoming connections.

Note:

Remember to free `pid` later with `ei_x_free`.

To unregister a name:

```
ei_global_unregister(&ec,fd,servicename);
```

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 `ei` library is thread safe, and using threads, one process can handle multiple 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 of type `ei_x_buff`.

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

`ei_term`

```
typedef struct {
    char ei_type;
    int arity;
    int size;
    union {
 long i_val;
 double d_val;
 char atom_name[MAXATOMLEN_UTF8];
 erlang_pid pid;
 erlang_port port;
 erlang_ref ref;
    } value;
} ei_term;
```

Structure written by `ei_decode_ei_term()`. The `ei_type` field is the type of the term which equals to what `ei_get_type()` sets `*type` to.

`ei_x_buff`

A dynamically resized buffer. It is a `struct` with two fields of interest for the user:

`char *buff`

Pointer to the dynamically allocated buffer.

`int index`

Offset to the next byte to write which also equals the amount of bytes currently written.

An `ei_x_buff` is initialized by calling either `ei_x_new()` or `ei_x_new_with_version()`. The memory used by an initialized `ei_x_buff` is released by calling `ei_x_free()`.

`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.

`erlang_fun`

Opaque data type representing an Erlang fun.

`erlang_pid`

Opaque data type representing an Erlang process identifier.

`erlang_port`

Opaque data type representing an Erlang port identifier.

`erlang_ref`

Opaque data type representing an Erlang reference.

`erlang_trace`

Opaque data type representing an Erlang sequential trace token.

Exports

`int ei_cmp_pids(erlang_pid *a, erlang_pid *b)`

Types:

`erlang_pid`

Compare two process identifiers. The comparison is done the same way as Erlang does.

Returns `0` if `a` and `b` are equal. Returns a value less than `0` if `a` compares as less than `b`. Returns a value larger than `0` if `a` compares as larger than `b`.

`int ei_cmp_ports(erlang_port *a, erlang_port *b)`

Types:

`erlang_port`

Compare two port identifiers. The comparison is done the same way as Erlang does.

Returns `0` if `a` and `b` are equal. Returns a value less than `0` if `a` compares as less than `b`. Returns a value larger than `0` if `a` compares as larger than `b`.

`int ei_cmp_refs(erlang_ref *a, erlang_ref *b)`

Types:

`erlang_ref`

Compare two references. The comparison is done the same way as Erlang does.

Returns `0` if `a` and `b` are equal. Returns a value less than `0` if `a` compares as less than `b`. Returns a value larger than `0` if `a` compares as larger than `b`.

`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)`

Types:

`erlang_char_encoding`

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_bitstring(const char *buf, int *index, const char **pp, unsigned int *bitoffsp, size_t *nbitsp)`

Decodes a bit string from the binary format.

`pp`

Either `NULL` or `*pp` returns a pointer to the first byte of the bit string. The returned bit string is readable as long as the buffer pointed to by `buf` is readable and not written to.

`bitoffsp`

Either `NULL` or `*bitoffsp` returns the number of unused bits in the first byte pointed to by `*pp`. The value of `*bitoffsp` is between 0 and 7. Unused bits in the first byte are the most significant bits.

`nbitsp`

Either `NULL` or `*nbitsp` returns the length of the bit string in **bits**.

Returns `0` if it was a bit string term.

The number of **bytes** pointed to by `*pp`, which are part of the bit string, is `(*bitoffsp + *nbitsp + 7)/8`. If `(*bitoffsp + *bitsp)%8 > 0` then only `(*bitoffsp + *bitsp)%8` bits of the last byte are used. Unused bits in the last byte are the least significant bits.

The values of unused bits in the first and last byte are undefined and cannot be relied on.

Number of bits may be divisible by 8, which means a binary decodable by `ei_decode_binary` is also decodable by `ei_decode_bitstring`.

`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)`

Types:

`ei_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)`

Types:

`erlang_fun`

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_iodata(const char *buf, int *index, int *size, char *outbuf)

Decodes a term of the type `iodata()`. The `iodata()` term will be flattened an written into the buffer pointed to by the `outbuf` argument. The byte size of the `iodata` is written into the integer variable pointed to by the `size` argument. Both `size` and `outbuf` can be set to `NULL`. The integer pointed to by the `index` argument is updated to refer to the term following after the `iodata()` term regardless of the the state of the `size` and the `outbuf` arguments.

Note that the buffer pointed to by the `outbuf` argument must be large enough if a non `NULL` value is passed as `outbuf`. You typically want to call `ei_decode_iodata()` twice. First with a non `NULL` `size` argument and a `NULL` `outbuf` argument in order to determine the size of the buffer needed, and then once again in order to do the actual decoding. Note that the integer pointed to by `index` will be updated by the call determining the size as well, so you need to reset it before the second call doing the actual decoding.

Returns `0` on success and `-1` on failure. Failure might be either due to invalid encoding of the term or due to the term not being of the type `iodata()`. On failure, the integer pointed to by the `index` argument will be updated to refer to the sub term where the failure was detected.

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.

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)

Types:

erlang_pid

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

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

Types:

erlang_port

Decodes a port identifier from the binary format.

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

Types:

erlang_ref

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_trace(const char *buf, int *index, erlang_trace *p)

Types:

erlang_trace

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.

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)

Types:

ei_x_buff

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)

Types:

ei_x_buff

erlang_char_encoding

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)

Types:

ei_x_buff

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)

Types:

ei_x_buff

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

int ei_encode_bitstring(char *buf, int *index, const char *p, size_t bitoffs, size_t nbits)

int ei_x_encode_bitstring(ei_x_buff* x, const char *p, size_t bitoffs, size_t nbits)

Types:

ei_x_buff

Encodes a bit string in the binary format.

The data is at `p`. The length of the bit string is `nbits` bits. The first `bitoffs` bits of the data at `p` are unused. The first byte which is part of the bit string is `p[bitoffs/8]`. The `bitoffs%8` most significant bits of the first byte `p[bitoffs/8]` are unused.

The number of bytes which is part of the bit string is `(bitoffs + nbits + 7)/8`. If `(bitoffs + nbits)%8 > 0` then only `(bitoffs + nbits)%8` bits of the last byte are used. Unused bits in the last byte are the least significant bits.

The values of unused bits are disregarded and does not need to be cleared.

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

Types:

ei_x_buff

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)

Types:

ei_x_buff

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)

Types:

ei_x_buff

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)

Types:

ei_x_buff

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)

Types:

ei_x_buff
erlang_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)

Types:

ei_x_buff

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)`

Types:

`ei_x_buff`

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)`

Types:

`ei_x_buff`

Encodes a GCC `long long` or Visual C++ `__int64` (64-bit) integer in the binary format.

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

Types:

`ei_x_buff`

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)

Types:

ei_x_buff
erlang_pid

Encodes an Erlang process identifier (pid) in the binary format. Parameter `p` points to an `erlang_pid` structure which should either have been obtained earlier with `ei_decode_pid()`, `ei_self()` or created by `ei_make_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)

Types:

ei_x_buff
erlang_port

Encodes an Erlang port in the binary format. Parameter `p` points to an `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)

Types:

ei_x_buff
erlang_ref

Encodes an Erlang reference in the binary format. Parameter `p` points to an `erlang_ref` structure which either should have been obtained earlier with `ei_decode_ref()`, or created by `ei_make_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)

Types:

ei_x_buff

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

Types:

ei_x_buff

erlang_trace

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)

Types:

ei_x_buff

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)

Types:

ei_x_buff

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)

Types:

ei_x_buff

Encodes a GCC `unsigned long long` or Visual C++ `unsigned __int64` (64-bit) integer in the binary format.

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

Types:

ei_x_buff

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 and bitstrings, `*size` is the number of bytes. For lists, tuples and maps, `*size` is the arity of the object. For bignum integers, `*size` is the number of bytes for the absolute value of the bignum. For other types, `*size` is 0. In all cases, `index` is left unchanged.

Currently `*type` is one of:

ERL_ATOM_EXT

Decode using either `ei_decode_atom()`, `ei_decode_atom_as()`, or `ei_decode_boolean()`.

ERL_BINARY_EXT

Decode using either `ei_decode_binary()`, `ei_decode_bitstring()`, or `ei_decode_iodata()`.

ERL_BIT_BINARY_EXT

Decode using `ei_decode_bitstring()`.

ERL_FLOAT_EXT

Decode using `ei_decode_double()`.

ERL_NEW_FUN_EXT
ERL_FUN_EXT
ERL_EXPORT_EXT

Decode using `ei_decode_fun()`.

ERL_SMALL_INTEGER_EXT
ERL_INTEGER_EXT
ERL_SMALL_BIG_EXT
ERL_LARGE_BIG_EXT

Decode using either `ei_decode_char()`, `ei_decode_long()`, `ei_decode_longlong()`, `ei_decode_ulong()`, `ei_decode_ulonglong()`, or `ei_decode_bignum()`.

ERL_LIST_EXT
ERL_NIL_EXT

Decode using either `ei_decode_list_header()`, or `ei_decode_iodata()`.

ERL_STRING_EXT

Decode using either `ei_decode_string()`, or `ei_decode_iodata()`.

ERL_MAP_EXT

Decode using `ei_decode_map_header()`.

ERL_PID_EXT

Decode using `ei_decode_pid()`.

ERL_PORT_EXT

Decode using `ei_decode_port()`.

ERL_NEW_REFERENCE_EXT

Decode using `ei_decode_ref()`.

ERL_SMALL_TUPLE_EXT
ERL_LARGE_TUPLE_EXT

Decode using `ei_decode_tuple_header()`.

Instead of decoding a term you can also skipped past it if you are not interested in the data by usage of `ei_skip_term()`.

`int ei_init(void)`

Initialize the `ei` library. This function should be called once (and only once) before calling any other functionality in the `ei` library.

On success zero is returned. On failure a posix error code is returned.

`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(unsigned release_number)`

In general, the `ei` library is guaranteed to be compatible with other Erlang/OTP components that are 2 major releases older or newer than the `ei` library itself.

Sometimes an exception to the above rule has to be made to make new features (or even bug fixes) possible. A call to `ei_set_compat_rel(release_number)` sets the `ei` library in compatibility mode of OTP release `release_number`.

The only useful value for `release_number` is currently `21`. This will only be useful and have an effect if **bit strings** or **export funs** are received from a connected node. Before OTP 22, bit strings and export funs were not supported by `ei`. They were instead encoded using an undocumented fallback tuple format when sent from the emulator to `ei`:

`Bit string`

The term `<<42, 1:1>>` was encoded as `{<<42, 128>>, 1}`. The first element of the tuple is a binary and the second element denotes how many bits of the last bytes are part of the bit string. In this example only the most significant bit of the last byte (128) is part of the bit string.

`Export fun`

The term `fun lists:map/2` was encoded as `{lists,map}`. A tuple with the module, function and a missing arity.

If `ei_set_compat_rel(21)` is **not** called then a connected emulator will send bit strings and export funs correctly encoded. The functions `ei_decode_bitstring` and `ei_decode_fun` has to be used to decode such terms. Calling `ei_set_compat_rel(21)` should only be done as a workaround to keep an old implementation alive, which expects to receive the undocumented tuple formats for bit strings and/or export funs.

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.

`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)`

Types:

`ei_x_buff`

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, ... )`

Types:

`ei_x_buff`
`erlang_pid`

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.

Change:

Since OTP 26.2 maps can be encoded with syntax like `"#{k1 => v1, k2 => v2}"`.

`int ei_x_free(ei_x_buff* x)`

Types:

`ei_x_buff`

Deallocates the dynamically allocated content of the buffer referred by `x`. After deallocation, the `buff` field is set to `NULL`.

`int ei_x_new(ei_x_buff* x)`

`int ei_x_new_with_version(ei_x_buff* x)`

Types:

`ei_x_buff`

Initialize the dynamically realizable buffer referred to by `x`. 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`.

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.

User Supplied Socket Implementation

By default `ei` supplies a TCP/IPv4 socket interface that is used when communicating. The user can however plug in his/her own IPv4 socket implementation. This, for example, in order to communicate over TLS. A user supplied socket implementation is plugged in by passing a callback structure to either `ei_connect_init_ussi()` or `ei_connect_xinit_ussi()`.

All callbacks in the `ei_socket_callbacks` structure **should** return zero on success; and a posix error code on failure.

The `addr` argument of the `listen`, `accept`, and `connect` callbacks refer to appropriate address structure for currently used protocol. Currently `ei` only supports IPv4. That is, at this time `addr` always points to a `struct sockaddr_in` structure.

The `ei_socket_callbacks` structure may be enlarged in the future. All fields not set, **needs** to be zeroed out. Currently the following fields exist:

`flags`

> Flags informing `ei` about the behaviour of the callbacks. Flags should be bitwise or:ed together. If no flag, is set, the `flags` field should contain `0`. Currently, supported flags:

`EI_SCLBK_FLG_FULL_IMPL`

If set, the `accept()`, `connect()`, `writev()`, `write()`, and `read()` callbacks implements timeouts. The timeout is passed in the `tmo` argument and is given in milli seconds. Note that the `tmo` argument to these callbacks differ from the timeout arguments in the `ei` API. Zero means a zero timeout. That is, poll and timeout immediately unless the operation is successful. `EI_SCLBK_INF_TMO` (max `unsigned`) means infinite timeout. The file descriptor is in blocking mode when a callback is called, and it must be in blocking mode when the callback returns.

If not set, `ei` will implement the timeout using `select()` in order to determine when to call the callbacks and when to time out. The `tmo` arguments of the `accept()`, `connect()`, `writev()`, `write()`, and `read()` callbacks should be ignored. The callbacks may be called in non-blocking mode. The callbacks are not allowed to change between blocking and non-blocking mode. In order for this to work, `select()` needs to interact with the socket primitives used the same way as it interacts with the ordinary socket primitives. If this is not the case, the callbacks **need** to implement timeouts and this flag should be set.

More flags may be introduced in the future.

`int (*socket)(void **ctx, void *setup_ctx)`

Create a socket and a context for the socket.

On success it should set `*ctx` to point to a context for the created socket. This context will be passed to all other socket callbacks. This function will be passed the same `setup_context` as passed to the preceding `ei_connect_init_ussi()` or `ei_connect_xinit_ussi()` call.

> **Note:**
>
> During the lifetime of a socket, the pointer `*ctx` **has** to remain the same. That is, it cannot later be relocated.

This callback is mandatory.

`int (*close)(void *ctx)`

Close the socket identified by `ctx` and destroy the context.

This callback is mandatory.

`int (*listen)(void *ctx, void *addr, int *len, int backlog)`

Bind the socket identified by `ctx` to a local interface and then listen on it.

The `addr` and `len` arguments are both input and output arguments. When called `addr` points to an address structure of length `*len` containing information on how to bind the socket. Upon return this callback should have updated the structure referred by `addr` with information on how the socket actually was bound. `*len` should be updated to reflect the size of `*addr` updated. `backlog` identifies the size of the backlog for the listen socket.

This callback is mandatory.

`int (*accept)(void **ctx, void *addr, int *len, unsigned tmo)`

Accept connections on the listen socket identified by `*ctx`.

When a connection is accepted, a new context for the accepted connection should be created and `*ctx` should be updated to point to the new context for the accepted connection. When called `addr` points to an uninitialized address structure of length `*len`. Upon return this callback should have updated this structure with information about the client address. `*len` should be updated to reflect the size of `*addr` updated.

If the `EI_SCLBK_FLG_FULL_IMPL` flag has been set, `tmo` contains timeout time in milliseconds.

> **Note:**
>
> During the lifetime of a socket, the pointer `*ctx` **has** to remain the same. That is, it cannot later be relocated.

This callback is mandatory.

`int (*connect)(void *ctx, void *addr, int len, unsigned tmo)`

Connect the socket identified by `ctx` to the address identified by `addr`.

When called `addr` points to an address structure of length `len` containing information on where to connect.

If the `EI_SCLBK_FLG_FULL_IMPL` flag has been set, `tmo` contains timeout time in milliseconds.

This callback is mandatory.

`int (*writev)(void *ctx, const void *iov, long iovcnt, ssize_t *len, unsigned tmo)`

Write data on the connected socket identified by `ctx`.

`iov` points to an array of `struct iovec` structures of length `iovcnt` containing data to write to the socket. On success, this callback should set `*len` to the amount of bytes successfully written on the socket.

If the `EI_SCLBK_FLG_FULL_IMPL` flag has been set, `tmo` contains timeout time in milliseconds.

This callback is optional. Set the `writev` field in the the `ei_socket_callbacks` structure to `NULL` if not implemented.

`int (*write)(void *ctx, const char *buf, ssize_t *len, unsigned tmo)`

Write data on the connected socket identified by `ctx`.

When called `buf` points to a buffer of length `*len` containing the data to write on the socket. On success, this callback should set `*len` to the amount of bytes successfully written on the socket.

If the `EI_SCLBK_FLG_FULL_IMPL` flag has been set, `tmo` contains timeout time in milliseconds.

This callback is mandatory.

`int (*read)(void *ctx, char *buf, ssize_t *len, unsigned tmo)`

Read data on the connected socket identified by `ctx`.

`buf` points to a buffer of length `*len` where the read data should be placed. On success, this callback should update `*len` to the amount of bytes successfully read on the socket.

If the `EI_SCLBK_FLG_FULL_IMPL` flag has been set, `tmo` contains timeout time in milliseconds.

This callback is mandatory.

`int (*handshake_packet_header_size)(void *ctx, int *sz)`

Inform about handshake packet header size to use during the Erlang distribution handshake.

On success, `*sz` should be set to the handshake packet header size to use. Valid values are `2` and `4`. Erlang TCP distribution use a handshake packet size of `2` and Erlang TLS distribution use a handshake packet size of `4`.

This callback is mandatory.

`int (*connect_handshake_complete)(void *ctx)`

Called when a locally started handshake has completed successfully.

This callback is optional. Set the `connect_handshake_complete` field in the `ei_socket_callbacks` structure to `NULL` if not implemented.

`int (*accept_handshake_complete)(void *ctx)`

Called when a remotely started handshake has completed successfully.

This callback is optional. Set the `accept_handshake_complete` field in the `ei_socket_callbacks` structure to `NULL` if not implemented.

`int (*get_fd)(void *ctx, int *fd)`

Inform about file descriptor used by the socket which is identified by `ctx`.

> **Note:**
>
> During the lifetime of a socket, the file descriptor **has** to remain the same. That is, repeated calls to this callback with the same context `should` always report the same file descriptor.
>
> The file descriptor **has** to be a real file descriptor. That is, no other operation should be able to get the same file descriptor until it has been released by the `close()` callback.

This callback is mandatory.

Data Types

`ei_cnode`

Opaque data type representing a C-node. A `ei_cnode` structure is initialized by calling `ei_connect_init()` or friends.

`ei_socket_callbacks`

```
typedef struct {
    int flags;
    int (*socket)(void **ctx, void *setup_ctx);
    int (*close)(void *ctx);
    int (*listen)(void *ctx, void *addr, int *len, int backlog);
    int (*accept)(void **ctx, void *addr, int *len, unsigned tmo);
    int (*connect)(void *ctx, void *addr, int len, unsigned tmo);
    int (*writev)(void *ctx, const void *iov, int iovcnt, ssize_t *len, unsigned tmo);
    int (*write)(void *ctx, const char *buf, ssize_t *len, unsigned tmo);
    int (*read)(void *ctx, char *buf, ssize_t *len, unsigned tmo);
    int (*handshake_packet_header_size)(void *ctx, int *sz);
    int (*connect_handshake_complete)(void *ctx);
    int (*accept_handshake_complete)(void *ctx);
    int (*get_fd)(void *ctx, int *fd);
} ei_socket_callbacks;
```

Callbacks functions for a *User Supplied Socket Implementation*. Documentation of each field can be found in the *User Supplied Socket Implementation* section above.

`ErlConnect`

```
typedef struct {
    char ipadr[4]; /* Ip v4 address in network byte order */
    char nodename[MAXNODELEN];
} ErlConnect;
```

IP v4 address and nodename.

`Erl_IpAddr`

```
typedef struct {
    unsigned s_addr; /* Ip v4 address in network byte order */
} Erl_IpAddr;
```

IP v4 address.

`erlang_msg`

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

Information about a message received via `ei_receive_msg()` or friends.

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)`

Types:

`ei_cnode`

`ErlConnect`

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.

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)`

Types:

`ei_cnode`

`ErlConnect`

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

`int ei_close_connection(int fd)`

Closes a previously opened connection or listen socket.

int ei_connect(ei_cnode* ec, char *nodename)
int ei_xconnect(ei_cnode* ec, Erl_IpAddr adr, char *alivename)
int ei_connect_host_port(ei_cnode* ec, char *hostname, int port)
int ei_xconnect_host_port(ei_cnode* ec, Erl_IpAddr adr, int port)

Types:

ei_cnode
Erl_IpAddr

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. The `ei_xconnect_host_port()` function provides yet another alternative that will work even if there is no EPMD instance on the host where the remote node is running. The `ei_xconnect_host_port()` function requires the IP address and port of the remote node to be specified. The `ei_connect_host_port()` function is an alternative to `ei_xconnect_host_port()` that lets the user specify a hostname instead of an IP address.

- `adr` 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.
- `port` is the port number 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, unsigned creation)

int ei_connect_init_ussi(ei_cnode* ec, const char* this_node_name, const char *cookie, unsigned creation, ei_socket_callbacks *cbs, int cbs_sz, void *setup_context)

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

int ei_connect_xinit_ussi(ei_cnode* ec, const char *thishostname, const char *thisalivename, const char *thisnodename, Erl_IpAddr thisipaddr, const char *cookie, unsigned creation, ei_socket_callbacks *cbs, int cbs_sz, void *setup_context)

Types:

ei_cnode

Erl_IpAddr

ei_socket_callbacks

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 name of the C-node (the name before '@' in the full node name).
- `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.

> **Note:**
>
> The type of the `creation` argument was changed from `short` (16 bit) to `unsigned int` (32 bit) in OTP 25. This should cause no practical problem other than maybe a compiler warning.

- `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 name of the local C-node (the name before '@' in the full node name). Can be `NULL` (from OTP 23) to get a dynamically assigned name from the peer node.
- `thisnodename` is the full name of the local C-node, that is, `mynode@myhost`. Can be `NULL` if `thisalivename` is `NULL`.
- `thispaddr` if the IP address of the host.
- `cbs` is a pointer to a callback structure implementing and alternative socket interface.
- `cbs_sz` is the size of the structure pointed to by `cbs`.
- `setup_context` is a pointer to a structure that will be passed as second argument to the `socket` callback in the `cbs` structure.

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:

```
unsigned 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)`

`int ei_connect_host_port_tmo(ei_cnode* ec, char *hostname, int port, unsigned ms)`

`int ei_xconnect_host_port_tmo(ei_cnode* ec, Erl_IpAddr adr, int port, unsigned ms)`

Types:

`ei_cnode`

`Erl_IpAddr`

Equivalent to `ei_connect`, `ei_xconnect`, `ei_connect_host_port` and `ei_xconnect_host_port` 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_listen(ei_cnode *ec, int *port, int backlog)`

`int ei_xlisten(ei_cnode *ec, Erl_IpAddr adr, int *port, int backlog)`

Types:

`ei_cnode`

`Erl_IpAddr`

Used by a server process to setup a listen socket which later can be used for accepting connections from client processes.

- `ec` is the C-node structure.
- `adr` is local interface to bind to.

- `port` is a pointer to an integer containing the port number to bind to. If `*port` equals `0` when calling `ei_listen()`, the socket will be bound to an ephemeral port. On success, `ei_listen()` will update the value of `*port` to the port actually bound to.
- `backlog` is maximum backlog of pending connections.

`ei_listen` will create a socket, bind to a port on the local interface identified by `adr` (or all local interfaces if `ei_listen()` is called), and mark the socket as a passive socket (that is, a socket that will be used for accepting incoming connections).

On success, a file descriptor is returned which can be used in a call to `ei_accept()`. On failure, `ERL_ERROR` is returned and `erl_errno` is set to `EIO`.

`int ei_make_pid(ei_cnode *ec, erlang_pid *pid)`

Types:

`ei_cnode`

`erlang_pid`

Creates a new process identifier in the argument `pid`. This process identifier refers to a conseptual process residing on the C-node identified by the argument `ec`. On success `0` is returned. On failure `ERL_ERROR` is returned and `erl_errno` is set.

The C-node identified by `ec` must have been initialized and must have received a name prior to the call to `ei_make_pid()`. Initialization of the C-node is done by a call to `ei_connect_init()` or friends. If the name is dynamically assigned from the peer node, the C-node also has to be connected.

`int ei_make_ref(ei_cnode *ec, erlang_ref *ref)`

Types:

`ei_cnode`

`erlang_ref`

Creates a new reference in the argument `ref`. This reference originates from the C-node identified by the argument `ec`. On success `0` is returned. On failure `ERL_ERROR` is returned and `erl_errno` is set.

The C-node identified by `ec` must have been initialized and must have received a name prior to the call to `ei_make_ref()`. Initialization of the C-node is done by a call to `ei_connect_init()` or friends. If the name is dynamically assigned from the peer node, the C-node also has to be connected.

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

Types:

`ei_cnode`

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)`

Types:

`ei_cnode`

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)`

Types:

`erlang_msg`

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)`

Types:

`erlang_msg`

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)`

Types:

`ei_x_buff`
`erlang_msg`

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.

`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)`

Types:

`ei_x_buff`
`erlang_msg`

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)`

Types:

ei_cnode

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)

Types:

ei_cnode

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

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

Types:

ei_cnode

ei_x_buff

erlang_msg

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.

The `ei_xrpc_to()` function is equivalent to `ei_rpc_to()` when its `flags` parameter is set to `0`. When the flags parameter of `ei_xrpc_to()` is set to `EI_RPC_FETCH_STDOUT`, stdout (standard output) data are forwarded. See the documentation for the flags parameter for more information about the `EI_RPC_FETCH_STDOUT` flag.

`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 an `ei_rpc_from()` call the result consists of a version magic number and a 2-tuple. The 2-tuple can be in one of the following two forms:

  `{rex,Reply}`
  : This response value means that the RPC has completed. The result value is the `Reply` term. This is the only type of response that one can get from an RPC triggered by a call to `ei_rpc_to()` or `ei_xrpc_to()` without the `EI_RPC_FETCH_STDOUT` flag. If the RPC was triggered by a call to `ei_xrpc_to()` with the `EI_RPC_FETCH_STDOUT` flag set, then all forwarded stdout data has been received.

  `{rex_stdout,StdOutUTF8Binary}`
  : This response value can only be obtained if the RPC call was triggered by a call to `ei_xrpc_to()` with the `EI_RPC_FETCH_STDOUT` flag set. This response value means that forwarded stdout data has been received. The stdout data is stored in a binary and is UTF-8 encoded. One may need to call `ei_rpc_from()` multiple times to read all the stdout data. The stdout data is received in the same order as it was written. All forwarded stdout data have been received when a `{rex,Reply}` tuple has been obtained from an `ei_rpc_from()` call.
- `flags` The flag `EI_RPC_FETCH_STDOUT` is currently the only flag that is supported by `ei_xrpc_to()`. When `EI_RPC_FETCH_STDOUT` is set, the called function is executed in a new process with a group leader that forwards all stdout data. This means that stdout data that are written during the execution of the called function, by the called function and by descendant processes, will be forwarded (given that the group leader has not been changed by a call to `erlang:group_leader/2`). The forwarded stdout data need to be collected by a sequence of calls to `ei_rpc_from()`. See the description of the `x` parameter for how `ei_rpc_from()` is used to receive stdout data. See the documentation of the the I/O protocol, for more information about the group leader concept.

> **Note:**
>
> The flag `EI_RPC_FETCH_STDOUT` only works when interacting with a node with a version greater or equal to OTP-24.

`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`. The functions `ei_rpc_to()` and `ei_xrpc_to()` returns 0 if successful, otherwise -1. When failing, all four 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)`

Types:

`ei_cnode`

`erlang_pid`

Retrieves a generic 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. Do **not** modify this structure.

On success a pointer to the process identifier is returned. On failure `NULL` is returned and `erl_errno` is set.

The C-node identified by `ec` must have been initialized and must have received a name prior to the call to `ei_self()`. Initialization of the C-node is done by a call to `ei_connect_init()` or friends. If the name is dynamically assigned from the peer node, the C-node also has to be connected.

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

Types:

`erlang_pid`

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)`

Types:

`erlang_pid`

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)`

Types:

erlang_pid

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)

Types:

erlang_pid

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 retrieved from the `ei_cnode` structure by calling `ei_self(cnode_pointer)`.

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

Types:

erlang_pid

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)

Types:

erlang_pid

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)

Types:

ei_cnode

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)

Types:

ei_cnode

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)`

Types:

`ei_cnode`

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

ei_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 **ei_global_names(ec,fd,count)`

Types:

```
ei_cnode *ec;
int fd;
int *count;
```

Retrieves a list of all known global names.

- `ec` is the `ei_cnode` representing the current cnode.
- `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 ei_global_register(fd,name,pid)`

Types:

```
int fd;
const char *name;
erlang_pid *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 ei_global_unregister(ec,fd,name)`

Types:

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

Unregisters a name from `global`.

- `ec` is the `ei_cnode` representing the current cnode.
- `fd` is an open descriptor to an Erlang connection.
- `name` is the name to unregister from `global`.

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

`int ei_global_whereis(ec,fd,name,pid,node)`

Types:

```
ei_cnode *ec;
int fd;
const char *name;
erlang_pid *pid;
char *node;
```

Looks up a name in `global`.

- `ec` is the `ei_cnode` representing the current cnode.
- `fd` is an open descriptor to an Erlang connection.
- `name` is the name that is to be looked up in `global`.

The `pid` parameter is a pointer to a `erlang_pid` that the function will update with the pid associated with the global name, if successful.

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 `ei_connect()` if necessary.

On success, the function returns 0, updates the `erlang_pid` pointed to by the pid parameter, and the `node` parameter is initialized to the node name where `name` is found. On failure, a negative number is returned.

erl_call

Command

`erl_call` makes it possible to start and/or communicate with a distributed Erlang node. It is built upon the `Erl_Interface` library as an example application. Its purpose is to use 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` except only a subset of all terms are allowed. The allowed term types are: `list` (and `string` representation of list, that is "example"), `tuple`, `atom` and `number`.

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.

`-address [Hostname:]Port`

(One of `-n`, `-name`, `-sname` or `-address` is required.) `Hostname` is the hostname of the machine that is running the peer node that `erl_call` shall communicate with. The default hostname is the hostname of the local machine. `Port` is the port number of the node that `erl_call` shall communicate with. The `-address` flag cannot be combined with any of the flags `-n`, `-name`, `-sname` or `-s`.

The `-address` flag is typically useful when one wants to call a node that is running on machine without an accessible epmd instance.

`-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.

`-fetch_stdout`

(**Optional.**) Executes the code, specified with the `-a` or `-e` option, in a new process that has a group leader that forwards all stdout (standard output) data so that it is printed to stdout of the `erl_call` process. This means that stdout data that are written during the execution of the called code, by the code and by descendant processes, will be forwarded (given that the group leader has not been changed by a call to `erlang:group_leader/2`).

The printed data is UTF-8 encoded.

This option is only relevant together with the option `-a` or `-e`.

See the documentation of the I/O protocol, for more information about the group leader concept.

> **Note:**
>
> This option only works when `erl_call` is interacting with a node with a version greater or equal to OTP-24.

`-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` or `-address` is required.) Has the same meaning as `-name` and can still be used for backward compatibility reasons.

`-name Node`

(One of `-n`, `-name`, `-sname` or `-address` is required.) `Node` is the name of the peer 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`.

`-no_result_term`

(**Optional.**) Do not print the result term. This option is only relevant together with the options `-a` and `-e`.

`-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.

`-R`

(**Optional.**) Request a dynamic random name, of the hidden node that `erl_call` represents, from the peer node. Supported since OTP 23. Prefer `-R` over `-r` when doing repeated requests toward the same peer node.

`-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` or `-address` is required.) `Node` is the name of the peer 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`.

`-timeout Seconds`

(**Optional.**) Aborts the `erl_call` process after the timeout expires. Note that this does not abort commands that have already been started with `-a`, `-e`, or similar.

`-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>,
                  []}]
```

To forward standard output without printing the result term (**again, the input ends with EOF (Control-D)**):

```
erl_call -s -e -sname madonna -fetch_stdout -no_result_term
io:format("Number of schedulers: ~p~n", [erlang:system_info(schedulers)]),
io:format("Number of logical cores: ~p~n", [erlang:system_info(logical_processors_available)]).
^D
Number of schedulers: 8
Number of logical cores: 8
```