Erlpiphany: Erlang Interface to the Epiphany SDK

Version: 1.1

Introduced in: January 2015

Authors: Mark A Fleming (mark_fleming@ieee.org).

Erlpiphany is the Erlang interface to the Epiphany eHAL library. With it an Erlang node can communicate with the Epiphany chip on a Parallella board to initiate and interact with parallel programs running on the Epiphany.

Contents

Introduction

Erlang provides several means of communicating with resources external to an Erlang node. An Erlang Port is a mechanism similar to the C function 'execv()' that can spawn an external program and communicate with the program via its standard input and output. Although simple to use, the programmer is responsible for converting ("marshalling") Erlang terms to a byte stream sent to the program and reassembling the result byte stream into Erlang terms. By using the 'ei' library one can create a linked in *port driver* that replaces the external program and communicates via the Port. A loadable library of routines will execute much faster, but a fault in the library code can crash the Erlang node.

An Erlang node can also incorporate external C libraries by creating a Natively Implemented Functions (NIF) loadable library. With a NIF implementation there is no need to manually translate Erlang terms to and from a byte stream as with a Port implementation. A NIF library is well suited to providing a straightforward interface to a foreign library of functions. However, like a loadable Port implementation, a fault in the library code can bring down the Erlang node.

Finally, an Erlang node can communicate with a node implemented in a different language, such as Java or C. The Erlang 'ei' library provides a means of creating a node written in C, or C-node, that can communicate with other Erlang nodes using standard

message passing or remote procedure calls. The 'jinterface' library provides the same capability for Java programs. A node implementation isolates an Erlang node from any library fault while providing rapid communication.

The **Erlpiphany** application provides a NIF library that acts as a wrapper for the eHAL Epiphany library. The application includes a set of C functions in 'c_src/ehal.c' that provide the NIF interface and the corresponding 'src/ehal.erl' module that provides an Erlang function for each eHAL library function. More information can be found in the 'ehal' documentation page.

Epiphany SDK Library

The NIF library implementation provides wrapper functions to access the Epiphany eHAL library. Every effort has been made to hew to the name and argument list convention of the original C functions. This was done to simplify the porting of C code to Erlang, to maintain familiarity as much as possible for the experienced SDK user, and to avoid the usual temptation to "improve" the original source arrangement.

Many of the SDK's defined values and enumerations can be found in the Erlang 'include/ehal.hrl' file. The file defines macros to represent such constants as E_REG_CONFIG (use ?E_REG_CONFIG). A small deviation from the SDK standard is to use the atoms 'true', 'false', 'ok' and 'error' in place of the defined integer values E_TRUE, E_FALSE, E_OK and E_ERR respectively.

Module functions use the Erlang convention of returning the atoms 'ok' and 'error' or the tuple '{ok, Result}'. For example the 'e_open/4' function returns '{ok, Dev}' while the 'e_read/5' function returns '{ok, {Ssize, Data}}'.

The correspondence between Epiphany SDK C functions and Erlang functions is shown in the next table.

C Function	Erlang Function	Erlang Return
`e_init(char *hdf)`	`e_init(String|0)`	`ok | error`
`e_get_platform_info(e_platform_t *platform)`	`e_get_platform_info()`	`{ok, Platform} | error`
`e_finalize()`	`e_finalize()`	`ok | error`
`e_open(e_epiphany_t *dev, unsigned row, unsigned col, unsigned rows, unsigned cols)`	`e_open(Row, Col, Rows, Cols)`	`{ok, Dev} | error`
`e_close(e_epiphany_t *dev)`	`e_close(Dev)`	`ok | error`
`e_alloc(e_mem_t *mbuf, off_t base, size_t size)`	`e_alloc(Base, Size)`	`{ok, Mbuf} | error`
`e_free(e_mem_t *mbuf)`	`e_free(Mbuf)`	`ok | error`
`e_read(void *dev, unsigned row, unsigned col, off_t from_addr, void *buf, size_t size)`	`e_read(Dev|Mbuf, Row, Col, From_addr, Size)`	`{ok, {Ssize, Data}} | error`
`e_write(void *dev, unsigned row, unsigned col, off_t to_addr, const void *buf, size_t size)`	`e_write(Dev|Mbuf, Row, Col, To_addr, Buf, Size)`	`{ok, Ssize} | error`

e_reset_system()	e_reset_system()	ok | error
e_reset_group(e_epiphany_t *dev)	e_reset_group(Dev)	ok | error
e_start(e_epiphany_t *dev, unsigned row, unsigned col)	e_start(Dev, Row, Col)	ok | error
e_start_group(e_epiphany_t *dev)	e_start_group(Dev)	ok | error
e_signal(e_epiphany_t *dev, unsigned row, unsigned col)	e_signal(Dev, Row, Col)	ok | error
e_halt(e_epiphany_t *dev, unsigned row, unsigned col)	e_halt(Dev, Row, Col)	ok | error
e_resume(e_epiphany_t *dev, unsigned row, unsigned col)	e_resume(Dev, Row, Col)	ok | error
e_load(char *executable, e_epiphany_t *dev, unsigned row, unsigned col, e_bool_t start)	e_load(String, Dev, Row, Col, true|false)	ok | error
e_load_group(char *executable, e_epiphany_t *dev, unsigned row, unsigned col, unsigned rows, unsigned cols, e_bool_t start	e_load_group(String, Dev, Row, Col, Rows, Cols, true|false)	ok | error
e_get_num_from_coords(e_epiphany_t *dev, unsigned row, unsigned col)	e_get_num_from_coords(Dev, Row, Col)	{ok, Corenum} | error
e_get_coords_from_num(e_epiphany_t *dev, unsigned corenum, unsigned *row, unsigned *col)	e_get_coords_from_num(Dev, Row, Corenum)	{ok, {Row, Col}} | error
e_is_addr_on_chip(void *addr)	e_is_addr_on_Chip(Addr)	true | false
e_is_addr_on_group(e_epiphany_t *dev, void *addr)	e_is_addr_on_group(Dev, Addr)	true | false
e_set_host_verbosity(e_hal_diag_t verbose)	e_set_host_verbosity(Verbose)	{ok, PreviousLevel} | error
e_set_loader_verbosity(e_loader_diag_t verbose)	e_set_loader_verbosity(Verbose)	{ok, PreviousLevel} | error

In general, Erlang integer terms are used wherever an integer is used in an SDK function and an Erlang list of characters is used for an SDK string. The Erlang string does not need to be null terminated.

Where structures are used or returned by an SDK function, the corresponding Erlang function uses a binary term. This allows the Erlang programmer to pass around the structure without concern for its size or content. The exception to this rule is the `e_platform_t` structure returned by the `get_platform_info()` function. The Erlang function returns a list of key-value tuples, where the key is the platform structure member name. Thus, `platform.row` is represented as `{row, Row}` in a list.

When dealing with memory buffers for the `e_read` and `e_write` SDK functions, the Erlang functions also use binaries. Consult the Erlang documentation for information on constructing and disassembling binaries. Examples are given in the Erlpiphany 'ehal' module documentation as well.

NIF Library

The 'src/ehal.erl' module provides the Erlang interface to the Epiphany SDK eHAL

function set. The module is compiled to a BEAM file that is kept in the 'ebin' directory. More detail for each function and its arguments can be found in the 'ehal' module documentation.

The C code that implements the link to the Epiphany eHAL library can be found in the c_src directory. The code is compiled to a loadable library `ehal.so` and kept in the 'priv' directory. A module that uses the library should include both the 'ebin' and 'priv' directories in its load path by using the `code:add_path/1` function.

Building Erlpiphany

This release of Erlpiphany was built and tested using the current version of Erlang (http://www.erlang.org) available at the time, OTP 17.4. The resulting NIF library was also tested using the version of Erlang available as an Ubuntu package.

To build the Erlpiphany package, download the source from github, change to the root directory and simply type 'make' to compile all files and place the results in their corresponding directories. As currently distributed, all code and examples have been compiled, so a build is not strictly necessary unless a change is made to the source.

Example Programs

The 'examples' directory contains a number of example programs supplied with the Epiphany SDK. For each of the examples, a corresponding Erlang program has been written to communicate with the Epiphany chip. As a quick introduction, try the following commands to run the hello_world example:

```
cd examples/hello_world
./build.sh
./run.sh
./runerl.sh
```

The first command compiles both the C and Erlang source files. The second command runs the standard C main program. The third command will run the Erlang version of the hello_world program by setting up the environment variables and then running the eScript `runerl.escript` in the Debug directory.

Example programs currently include

1. **Hello World** A simple program that selects a random eCore, loads a program, and verifies program operation. The Erlang version `hello_world.erl` duplicates the original C program functionality.
2. **Dot Product** Creates a workgroup containing all eCores on an Epiphany chip and loads a program to compute a vector cross product on each core in parallel. The `main.erl` program replicates the original C program main behavior, while the `dotproduct.erl` program launches a separate Erlang process per eCore to load a program and compute cross products. The parent process prints the order in which each eCore process returns its eCore result.

Using Erlpiphany

Once you've examined and run the example Erlang programs, the next step is to develop and operate programs of your own.

Scripting

Scripting is the simplest approach to programming an Epiphany chip with Erlang. The 'examples/scripts' directory provides a starting point for scripting. You can write Erlang eScripts that access the eHAL library directly as shown in the `scripts/info.escript` file. To invoke the script, type

```
./escript.sh ./info.escript
```

to run the program and print out the Platform info list. The 'escript.sh' file provides all setups needed to run an eScript as root. The eScript itself is quite simple.

```
#!/usr/bin/env escript
%% -*- erlang -*-
main([]) ->
    code:add_path("/home/linaro/Work/erlpiphany"),
    code:add_path("/home/linaro/Work/erlpiphany/ebin"),
    code:add_path("/home/linaro/Work/erlpiphany/priv"),
    application:load(erlpiphany),
    ok = ehal:e_init(0),
    ok = ehal:e_reset_system(),
    {ok, Platform} = ehal:e_get_platform_info(),
    io:format("Platform info: ~p~n", [Platform]),
    ok = ehal:e_finalize().
```

The two add_path/1 calls establish the location of the erlpiphany application and the `ehal.beam` file respectively. the call to application:load/1 insures that the ehal NIF loadable library is available when the Erlang `hello_world` program is invoked.

For Erlang to find the `e-hal.so` loadable library and access the Epiphany chip, the Erlang eScript program must be run with certain established environment variables. These environment variables are preestablished in the Linaro parallella account.

Just as shell scripts are used to launch pre-compiled C programs, Erlang scripts ("eScripts") can also be used to run Erlang modules compiled to BEAM files. One such example script can be found in the Hello World example, as shown below.

```
#!/usr/bin/env escript
%% -*- erlang -*-
main([]) ->
  code:add_path("/home/linaro/Work/erlpiphany"),
  code:add_path("/home/linaro/Work/erlpiphany/ebin"),
  application:load(erlpiphany),
  hello_world:main().
```

Establishing Node Communication

Erlang eScripts are useful for quick prototyping but have their limitations in terms of speed. For more complex applications, you'll need to develop one or more Erlang modules.

To develop and test an Epiphany application you'll need to start an Erlang node. as in

```
erl -cookie mysecret -sname ehal_node
```

Rather than setting the node cookie on the command line, you can put the text in `~/.erlang.cookie` so that all of your Erlang nodes use the same cookie value.

Try running the above command in the scripts directory to start an Erlang node, then start a node in another window with the command

```
erl -sname linaro
```

In the 'examples/scripts' directory type the folowing commands in your Erlang 'linaro' node

```
(linaro@linaro-nano)1> net_adm:world().
['linaro@linaro-nano', 'ehal_node@linaro-nano']
(linaro@linaro-nano)2> rpc:call('ehal_node@linaro-nano', os, cmd, ["./info.escript"]).
```

The net_adm:world/0 command establishes communication with all Erlang nodes running on the Parallella board that use the same security cookie. You can connect to nodes on remote hosts by putting the host names in the file `~/.host.names` with each host name on a separate line, surrounded by single quotes and terminated with a period. Another way to connect is to use the net_adm:world_list/1 to specify one or more hosts as in

```
net_adm:world_list(['mynotebook', 'workstation.domain.org']).
```

Notice the rpc call returns the same output as when the command is run in the bash shell. You can use this approach to run eScripts that act as wrappers for larger programs you develop. You can also use this remote procedure call to invoke Erlang functions on the Parallella board from your workstation.

Invoking Commands

Lets try remotely running an example program that has already been developed and compiled. Using your local Erlang node and the Erlang node running in the scripts directory, use the following commands to access and run the hello world demo.

```
('linaro@linaro-nano'>3 rpc:call('ehal_node@linaro-nano', file, set_cwd, ["../hello-
world/Debug"]).
ok
('linaro@linaro-nano'>4 rpc:call('ehal_node@linaro-nano', hello_world, main, []).
```

Notice you get the expected hello world demo output on your local console. Once you have a working Erlpiphany application, you can invoke the application remotely from your workstation after starting and connecting to an Erlang node running on your Parallella board.

Overview

Generated by EDoc, Apr 13 2015, 14:58:22.