

Embedded Erlang Simulation

Rickard Olsson & Reza Javaheri

02-08-2011



CHALMERS

Bachelor's thesis in Computer Science
Datavetenskapligt Program
Department of Computer Science
Chalmers University of Technology

Supervisors:
Hans Svensson, Chalmers
Robert Virding, Erlang Solutions

Abstract

The goal of this thesis is to develop simulators for device drivers and to create and describe a work flow for developing on Erlang Embedded using simulators. The motivation is that frequently testing code on hardware is inconvenient, takes time and only is possible if you have access to the hardware. By simulating simple device drivers like serial ports, buttons and leds, this thesis aims to prove that a better development environment can be created.

A development work flow describes how the current work flow with Erlang Embedded can be improved by using simulators. The idea is that you should be able to switch between real hardware and simulated mode by just changing an environment variable and providing necessary configuration files. No changes in code are required.

A recorder tool was developed to support simulation of devices that generate data such as sensors. The recorder can also be used to quickly create a simulator replaying data traffic that have been recorded from an application previously. Taking advantage of `dbg` module, the recorder captures messages that a process receives or sends off without any modification to the source code. Using the recorder tool, no API or application logic is needed to simulation a sensor.

The thesis was conducted at Chalmers University of Technology under supervision of Erlang Solutions Ltd. Erlang Embedded Simulation is available at Github [19] and is licensed under: Apache License Version 2.0, January 2004 [1].

Contents

1	Acknowledgements	5
2	Introduction	6
2.1	Erlang	6
2.2	Erlang Embedded	6
2.3	Simulation in Erlang	7
3	Definitions	8
4	Problem Description	10
4.1	Scope	10
4.2	Deliverables	10
5	Methodology	12
5.1	Approach	12
5.2	Measuring Quality of Simulators	12
5.3	Simulation	12
6	Related Work	13
6.1	Meck	13
6.2	Erlang Embedded	13
6.3	GNU Xnee	13
6.4	Semi-formal Development of a Fault-Tolerant Leader Election Protocol in Erlang	13
7	Results	14
7.1	Simulators	14
7.1.1	Serial Stub Simulator	15
7.1.2	Button Stub Simulator	15
7.1.3	LED Stub Simulator	15
7.1.4	Stub Monitor	15
7.1.5	Serial Demo	16
7.2	Erlang Recorder	16
7.3	Measurements	17
7.3.1	Simulator Accuracy	17
7.4	Workflow	18
7.5	Convenience Scripts	21
7.5.1	Ember	22
7.5.2	Internet Over USB-OTG	22

8 Conclusion	23
8.1 Correctness of Simulation	23
8.2 Improvement of using Simulators	23
9 Future Work	24
9.1 Runtime function call replacement	24
9.2 Stdio Interface	24
9.3 Recording and Replaying Messages Containing PIDs	24
9.4 Creating Finite State Machines from logs	25
9.5 Remote Testing	25

1 Acknowledgements

Robert Virding For daily discussions and feedbacks, helping us to focus on the interesting parts, providing Erlang expertise and suggesting improvements.

Hans Svensson For showing interest in our thesis and supervising it as well as providing helpful feedback.

Ulf Wiger For helping us finding a thesis topic and providing feedback.

Fredrik Andersson, Gustav Simonsson, Henrik Nordh For introducing us to Erlang Embedded and helping us to set up Beagle Board.

Erlang Solutions For giving us the opportunity to work on our thesis in their office in Stockholm.

Henrik Sandklef For feedback and ideas for future work, e.g. interfacing to other languages.

2 Introduction

Our thesis claims that you can shorten development time and improve quality by using simulators. The three main issues that are addressed in this thesis are: creating tools to support development and use of simulators, some basic simulators, and a work flow describing how to use and develop simulators in Erlang. By comparing workflows with and without simulators, we have identified several benefits of using simulators. Time can be saved because the program binary does not need to be deployed to hardware for testing or demonstration purposes as frequent.

If the proper simulator is available, the software can be tested also during development when there is no access to the hardware. When working with simulators a main concern is whether the simulator conforms to the hardware. Provided with the identical input data, the hardware and the simulator should result in the same output in the same order with the same relative intervals in a deterministic application.

2.1 Erlang

Erlang is a high-level, functional language and runtime system with built-in support for concurrency, distribution and fault-tolerance. The Erlang virtual machine acts like an OS and supports concurrent processes, scheduling, memory management and etc. Concurrency is implemented regarding the Actor Model [23]. Processes do not share data, instead, they communicate by sending messages. When a message is sent to a process it goes into that process' inbox, where it can be retrieved by the receiving process [22, p. 1-9].

As embedded devices get more powerful, the applications they run tend to grow in complexity which makes Erlang an interesting choice for embedded development. Currently, Erlang is mostly used on high end [22, p. 2] servers but Ericsson has some releases for embedded devices. However, they are tailored for Ericsson and less interesting outside Ericsson [16, p. 14].

2.2 Erlang Embedded

Erlang Embedded is a prototype based on two thesis projects conducted by Fredrik Andersson, Fabian Bergström, Gustav Simonsson and Henrik Nord at Erlang Solutions[8]. Erlang embedded provides a linux distro and an Erlang release that is tailored for embedded hardware. Currently Erlang Embedded exists as a prototype and while technically it works well with running Erlang, there is yet no development environment or work flow available.

The hardware supported at the moment is Beagle Board and Gumstix and the distro is Ångström. The Erlang release is a stripped down and compressed version of the original release, at the size of 2.8MB. By configuring different garbage collectors, memory usage can also be reduced [16, p. 14].

Erlang Embedded passed all OTP tests except for the memory intensive tests and file permissions. However, both of those tests are irrelevant on the embedded hardware. The reasons are that the memory is supposed to be limited and the file format on the memory card has no support for such file permission operations. The Erlang Embedded team is currently working on a new release. Support for the Panda Board may also be added in the future.

2.3 Simulation in Erlang

There are four ways in which Erlang communicates with the outside world: Distributed Erlang, Ports, Linked-in drivers and NIFs [5]. Ports in Erlang are written in C and are hidden from Erlang behind an Erlang API, acting as an interface to the C program. The API starts the external C program and starts listening for messages from that process. Ports are one of the common ways to hook up the Erlang application to an external peripheral. Due to the latter, simulation of devices communicating through a port to an Erlang application is the focus of this thesis. The API is unaware of whether it is communicating to the real hardware using a port or an Erlang process acting as a driver. Although, this is only possible if drivers are very well hidden behind an API.

All communications between processes are being carried out through message passing. The data traffic of an arbitrary process can be traced and captured using an Erlang built-in tool called `dbg`. The message data, relevant timestamp and the recipient of that message can be used to simulate the behavior of that specific process. This can be done by impersonating that process and resending all the recorded messages. Needless to say the correct order and the exact delays between each two messages are kept intact. It is also easy for simulators to use the replayer to replay messages since it only needs to read a log file with messages and send a message to the specified process after a given delay.

3 Definitions

Name	Description
PID	Process identifier. PIDs are returned by the function <code>spawn/3</code> which starts a new process in Erlang. The PID can be used to identify a process, send messages to it and to register it under a name. [7]
dbg	"This module implements a text based interface to the <code>trace/3</code> and the <code>trace_pattern/2</code> BIFs. It makes it possible to trace functions, processes and messages on text based terminals." [3]
Port	"Ports provide the basic mechanism for communication with the external world from Erlang's point of view. They provide a byte-oriented interface to an external program. When a port has been created, Erlang can communicate with it by sending and receiving lists of bytes, including binaries." [6]
Linked-in driver	Also known as Port Driver. "The external program resides in another OS process than the Erlang runtime system. In some cases this is not acceptable, consider for example drivers with very hard time requirements. It is therefore possible to write a program in C according to certain principles and dynamically link it to the Erlang runtime system, this is called a linked-in driver." [5]
NIF	Native Implemented Functions. "A NIF library contains native implementation of some functions of an Erlang module. The native implemented functions (NIFs) are called like any other functions without any difference to the caller." [4]
Message Passing	Parallel communication where processes can send and receive messages
USB OTG	Universal Serial Bus On-The-Go
IO	Input/Output
stdio	Standard Input/Output
NMEA	A communication specification used by GPS

Name	Description
FSM	Finite State Machine
API	Application Programming Interface
LED	Light-emitting diode
GPS	Global Positioning System
Actor Model	The Actor Model[23], which was first proposed by Carl Hewitt in 1973 , takes a different approach to concurrency that should avoid the problems caused by threading and locking.

4 Problem Description

Testing code frequently on hardware is inconvenient, takes time, and is only possible if you have access to the hardware. Previously, no documented development workflow for Erlang Embedded existed. By simulating simple devices like serial ports, push buttons, LEDs, GPS navigation devices and creating a development workflow, this thesis aims to prove that a better development environment can be created.

4.1 Scope

The thesis focused on the following topics:

- How simple stubs as simulators can be used to allow testing on desktop environments. Switching between hardware and software environments should be possible without any changes in code and with minimum configuration.
- Capture messages on hardware and replay them later on hardware or simulator.
- How to put together a work flow in order to provide an improved development process for Erlang Embedded.
- Simplify the process of uploading code onto the Beagle Board e.g. *make deploy* or *make test*.
- Measuring the quality of supervisors by using logs to compare results and estimating quality using hand-waving analysis. This was carried out by observing and comparing the old workflow with the new one.

The following topics are outside the scope:

- Simulation of environment and resources.
- Linked-in drivers and NIFs.

4.2 Deliverables

- Simulation library with a few simulators and support for creating new simulators.
- Workflow for Erlang Embedded.

- Convenience scripts for setting up hardware and deploying code to simplify development.
- Tool for recording and replaying messages.

5 Methodology

5.1 Approach

A development work flow was described by studying service-oriented paradigms, Erlang's characteristics and by using simulators with Erlang. Deeper knowledge in simulation and development was gained by implementing a simulation library as well as simulators and to use these to develop sample applications. The sample applications demonstrated our progress through out the project.

The simulators were written in Erlang that reusable code was abstracted to generic modules. The recorder utilized Erlang `dbg` to record messages.

5.2 Measuring Quality of Simulators

The quality of simulators and our approach can be measured by using the recording and replaying tools that we created. Recorded driver outputs on the hardware mode can be replayed on simulation mode. If the driver outputs are the same in both modes, the application output on hardware and simulation mode can be compared to show the conformity of the simulator. This was the quantitative approach used to measure quality and conformity of the simulators.

Qualitative research was also performed where the assumption was that using simulators is an improvement. By observing the development of a sample application, we should be able to make a proposition whether simulators can improve development process.

To support the qualitative research, one could use the tools and workflow to create e.g. a GPS simulator, as well as a driver and measure the outcomes. Some measurements like development time, number of errors on the hardware in comparison with the simulators can be used to support the proposition that using simulators is an improvement.

5.3 Simulation

Simulation is mimicking the behavior of a real thing or a process. There are usually three main issues in simulation that should be dealt with.

- Find a valid source of information.
- Use a proper simplifying approximations and assumptions within the simulation.
- Precision and validity of the simulation outcomes[15, p. 452].

6 Related Work

6.1 Meck

Meck is a mocking library for Erlang which allows you to replace functions in order to simplify testing [20].

6.2 Erlang Embedded

Our thesis started mainly as a continuation of the Erlang-Embedded project. "Erlang-Embedded is a platform for developing embedded Erlang applications. The project started as a collection of thesis projects at Uppsala University, under the supervision of Erlang Solutions [8]."

6.3 GNU Xnee

Gnu Xnee is another related project to our thesis. "GNU Xnee is a suite of programs that can record, replay and distribute user actions under the X11 environment. Think of it as a robot that can imitate the job you just did. GNU Xnee is licensed under GNU GPLv3 [10]."

6.4 Semi-formal Development of a Fault-Tolerant Leader Election Protocol in Erlang

The article discusses how to create the most generic finite state machine out of recorded trace files [21]. Recording has a great value for embedded simulation, but also elsewhere. Some interesting future work would be to create a FSM out of recording logs. Perhaps the recorder tool created as the result of this thesis can be used to collect message traffic to and from a specific FSM. Those recordings can be used later to reverse engineer the FSM using the algorithm proposed in the article mentioned above. The reverse engineered FSM produced in the latter step can be used to simulate that specific FSM.

7 Results

7.1 Simulators

A generic module to support the use of simulators and three simple device simulators for led, button and serial communication were created. All simulators are implemented as finite state machines, using the OTP behavior `gen_fsm`. The generic module lets one define different actions depending on values of OS variables. This is used to start the simulator if an OS variable (`EMBEDDED_ENV`) is set to "sim", if not a port is started instead. This allows the same code to be run in a simulated environment or on the real hardware.

Interaction with the port is handled by message passing. If hardware mode is set, the function returns the pid to a port, otherwise a pid to a simulator is returned. This is done in the API of the port, written in Erlang, which means that if the simulator handles the messages correctly then the processes that are using the API will not be able to notice if it is a simulator or a port (unless they run `is_port/1`). This makes simulation transparent, and allows the same code to be run on target hardware or in a simulated environment. The developer of the simulator needs to support the same functionalities defined by the same API as the port providing proper replies, if a reply is needed.

A simple workflow could be described as.

- Step one is to design an API for the driver, implement a stub simulating the driver using the API. Test the simulator using your application, and redesign the API and/or simulator until you get a good result. Specifications and limitations of the actual hardware should be taken into consideration.
- Step two is to create the driver and follow the API. Run tests for the driver. When it seems stable, upload the code to your hardware and supply the `open_port` arguments to the simulator. The idea is to lower the amount of testing and debugging required at this stage. Note that you should also run application tests on the hardware.

At the moment there are some limitations e.g. `serial_stub:start_port` should be used instead of `open_port`. In addition, `port_commands` can not be simulated, these have to be changed to message passing instead. Moreover, simulators such as GPS and other devices generating data are harder to simulate and our solution to this problem is to use a tool for recording messages on hardware and replay them later in simulators. This involves log

files with timestamps and messages that the simulator can replay. See the recorder section for more information.

7.1.1 Serial Stub Simulator

Serial communication was simulated by simply using distributed Erlang over two nodes. The node to send messages to needs to be defined as an application variable. Setup involved creating two nodes and adding the corresponding nodes names to the vm args. This simulation applies for simulating other communication as well. To improve the serial simulator baud rate restrictions could be added to provide a more accurate scenario, but it is not the focus of this thesis.

Since a serial driver was already available, the simulator could reuse the driver's API [17].

7.1.2 Button Stub Simulator

To simulate the user button on the board, a simple API, a simulator together with the driver was created. The API consists of a start function that starts listening for button push and release messages. The button driver reads the button events and sends messages to its listener pid (the process that started the driver) when the button is pushed it sends a push messages and when the button is released a release message is sent. Messages from the driver are in binary form and messages from the API are Erlang terms. The simulator's API has a push function that allows the button to be pushed and released after a specified delay in millisecond, which can be used for testing.

7.1.3 LED Stub Simulator

The LED also has an API, a simulator and a driver. The driver listens for messages to turn the LED on and off. Sending messages to the API can turn on or off the light, which changes the brightness of the LED in hardware mode and the internal state of the simulator in simulated mode.

7.1.4 Stub Monitor

The LED and button simulator send messages to a stub monitor that is used for printing the internal state of the simulator as it changes. This provides demonstration and manual testing of the simulators. Stub monitor is not meant to be used by automated tests.

7.1.5 Serial Demo

To test and demonstrate the simulators a demo application was created, consisting of two Beagle Boards [2] connected over serial port, sending messages to each other. If a button is pushed on one Beagle Board, the led will light on the other. The application can be run on Beagle Board or desktop without any modifications. There is only an OS variable that needs to be set to determine whether the real hardware should be used or the stubs.

7.2 Erlang Recorder

After the first release new peripherals were assessed and the focus was on a USB GPS. This simulator would be quite different from previous simple state simulators, since it generates data. When a push button is pushed, simply it notifies when its state has changed in form of an event that another process might be listening to and storing its state. A GPS generates data, it asynchronously sends data over serial port, in form of e.g. NMEA messages every 500 ms. In order to simulate this behavior an application for recording and replaying messages was created. The idea is to record the messages sent from the Erlang port, and replay them in the simulator.

The recorder tool is used for recording the outgoing and incoming messages sent between processes and is based on a debugging tool called `dbg` [3]. The purpose is to create data-generating simulators. Messages can be recorded, or in the lack of hardware dummy messages can be written manually. Messages can be recorded with `dbg` by just specifying which process and what kind of message one would like to record. Erlang recorder is a layer on top of `dbg`, with more filtering options for messages, different message format and relative timestamps. The timestamps inform the recorder how long it should wait before retrieving and replaying the next message from the log.

Recording the outgoing messages from the port did not work using `dbg`. To resolve this problem, messages were recorded from the port as they arrived to the receiving process mailbox instead. Conveniently, the message protocol for ports includes the port's pid in the messages, which made it easy to filter out the port messages from other messages sent to the API.

Another use case for the recorder is to create application log files on the hardware and in the simulated mode, which can be used to measure conformity of the simulator. In the serial demo the application reads data from a port and sends that data to the serial driver that writes to the serial port. By recording the driver and application output on hardware we can replay the recorded messages from the driver in simulated mode and record application output. If the application log file from hardware mode and simulated mode

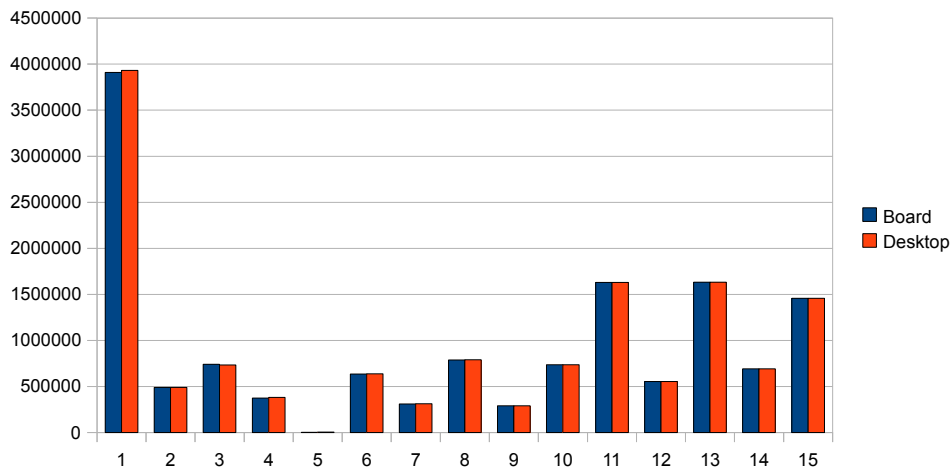
are being compared, the result can be used for measuring the quality and conformity of the simulator.

7.3 Measurements

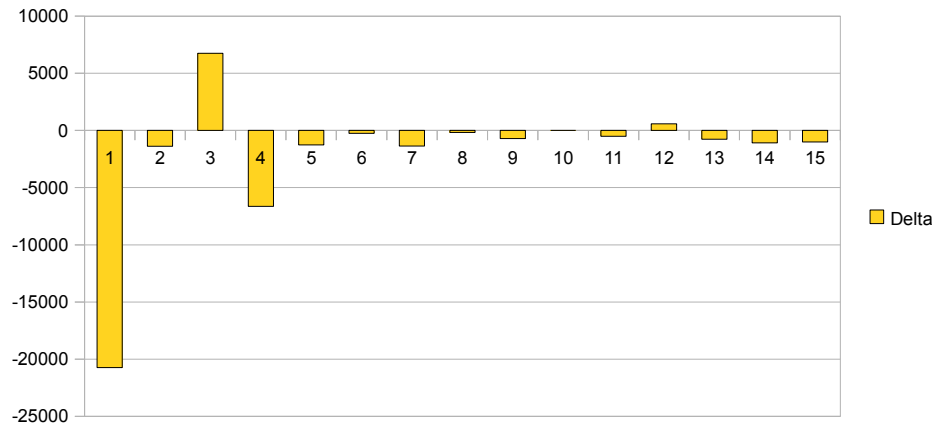
The difference between hardware application output and simulation application output given the same recorded hardware output should be as small as possible. In simple systems the sequence of messages should be the same. Time between messages should be relatively equivalent, timestamps on Beagle Board are likely to be larger since it is a slower machine.

One assumption that could be made is that writing a stub in Erlang and testing the code on desktop is a faster development approach than uploading the code to hardware and test it. One can try to validate this by researching how much time it takes to write a stub, how many errors slip through, errors found on target that testing with the stub did not catch. If one deploys to different targets, how much needs to be changed in the application, rather than just changing code at the stub level?

7.3.1 Simulator Accuracy



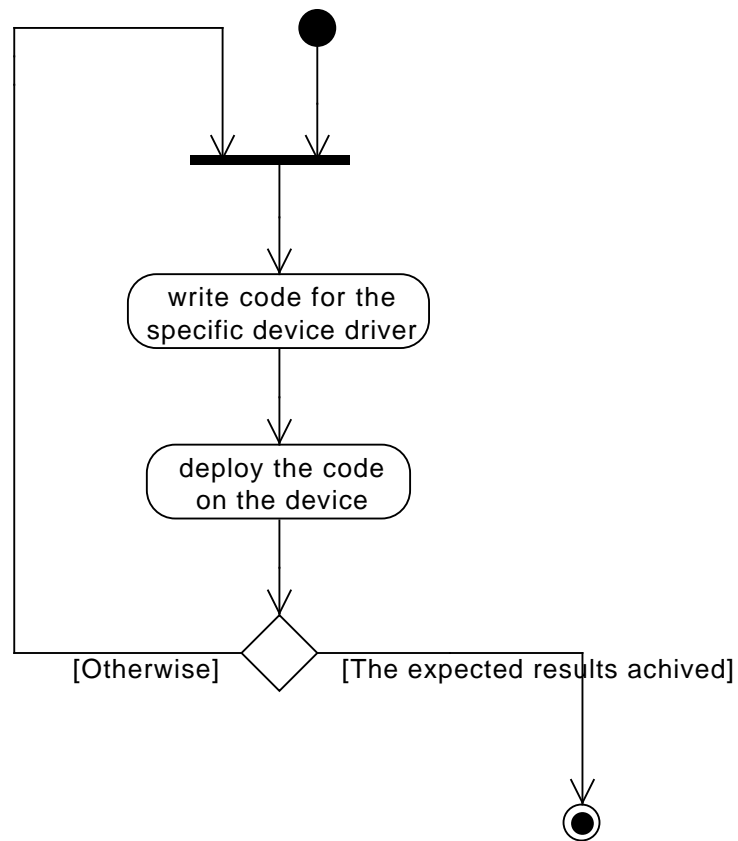
The above illustration compares delays before receiving a sequence of messages on a desktop computer and a Beagle board. Y-axis is time in micro second and X-axis denotes samples. The difference is more clear in the diagram below.



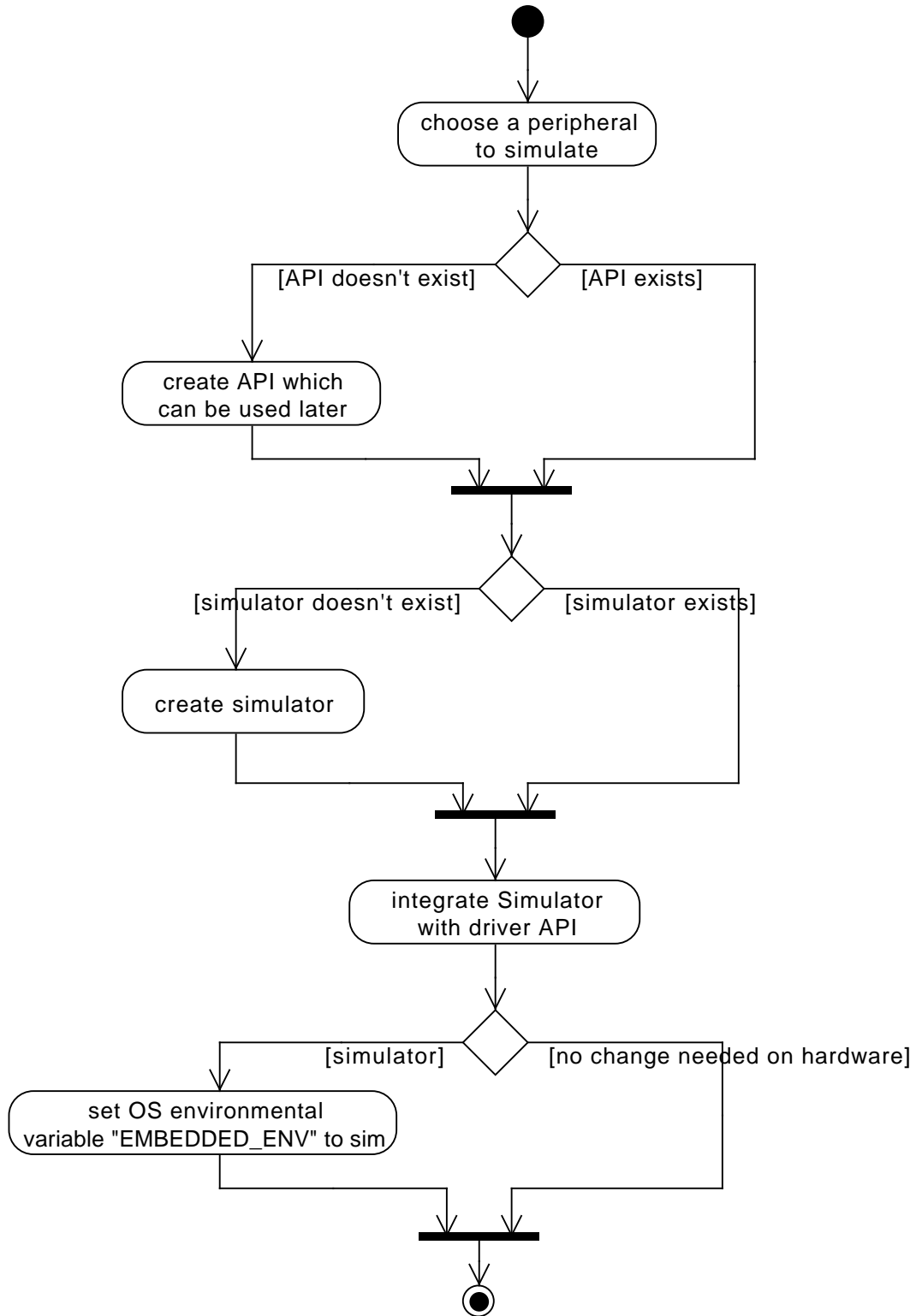
The minus values on the Y-axis mean that the Beagle Board was behind the desktop machine and it responded to the same message with more delay. It can be inferred that initialization was relatively slow on Beagle board.

7.4 Workflow

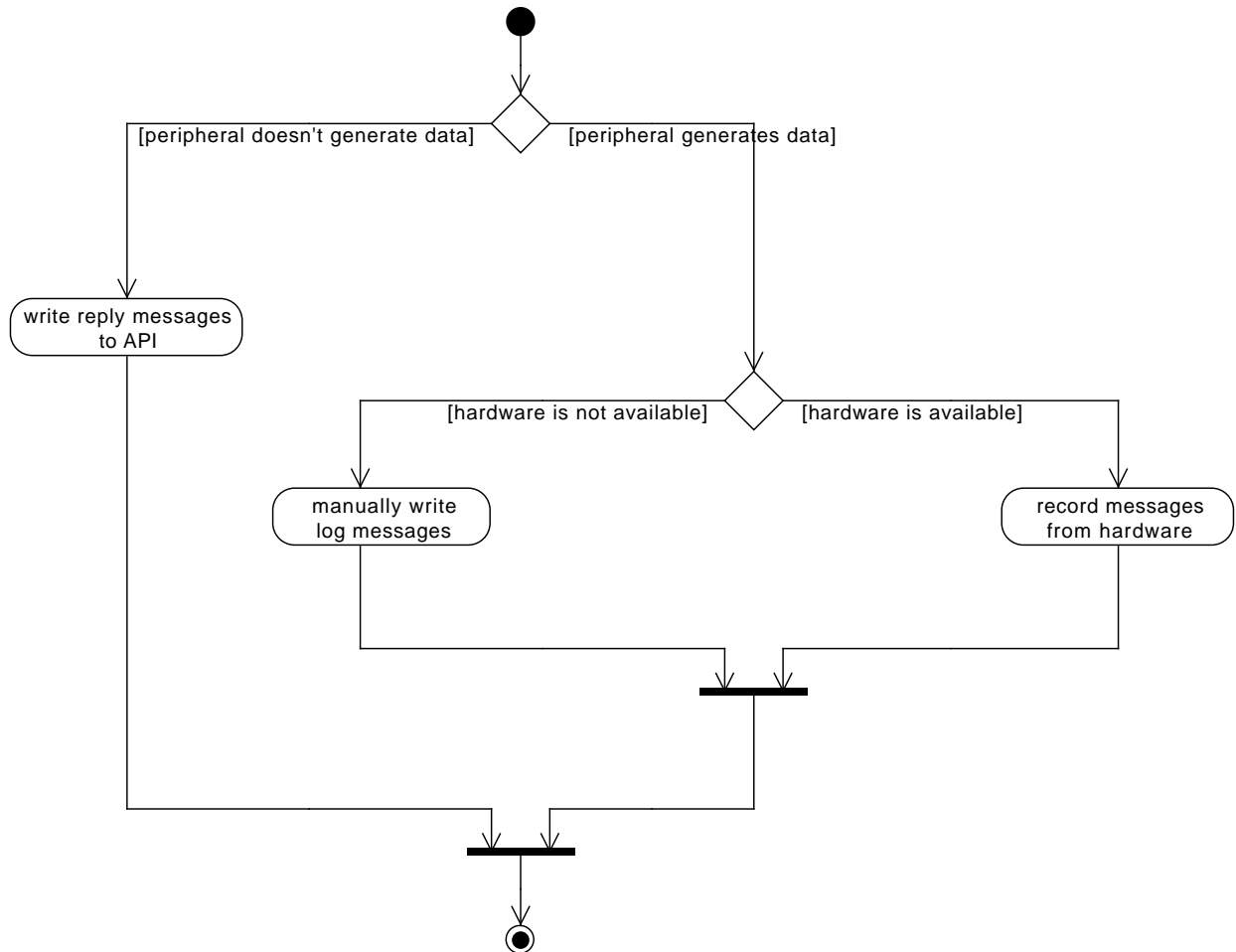
Here we start with one possible way of developing software for a different platform than the development environment. Abstracted enough, such a development process, inevitably, bears a tedious loop of development and deployment to the device.



The above loop iterates over and over until the expected results are achieved. The illustration above shows one way of developing an application without using a simulator.



The "create simulator" box is elaborated in the following illustration:



The workflow describes how development on Erlang Embedded using our tools is intended to be carried out. The workflow is meant to work as a development process to guide developers using simulators. The improvement in the new workflow is that you do not need to put code on hardware to test it. It shows how the API can be reused, how you would do if you already have a simulator and what to think about when creating and using these simulators.

7.5 Convenience Scripts

Some scripts were created to make the development process easier. The focus was the Angstrom Linux running on a Beagle Board hardware, so the produced scripts are consistent with that environment.

7.5.1 Ember

Ember is a bash script that does some of the boring development stuff so you do not have to. You can set up a config file with ssh and project variables and Ember can automate the process of deploying code for you and setting up the network interfaces(to be implemented).

The Erlang Embedded simulators library is generic so one can choose yourself what OS variables to use and the library will simply provide you with the option to call different function depending on your own setup and environment variables.

7.5.2 Internet Over USB-OTG

The script sets up IP based networking interface between the Beagle Board and the host machine. Further it uses NAT to provide the Beagle Board with Internet connection.

8 Conclusion

8.1 Correctness of Simulation

It turned out that it is possible to create simulators for Beagle Board using Erlang Embedded. Moreover, simulation of some data generator peripherals (such as sensors) can be done by recording the data packets that they send. The recorded data can be used later to simulate that specific peripheral. The underlying tracing capability of Erlang made it considerably easy to record the messages passing through a port and consequently producing a simulator for such a data generator device became trivial. Comparison of time-stamped reply messages on the hardware and in simulated environment for the demo application reveals that the simulators created in this thesis project behave relatively the same.

8.2 Improvement of using Simulators

The reasoning was that it takes more time to upload code on hardware to test it than to test it on desktop using simulators. During development of the sample applications the drivers that were created were fairly simple and consisted of few lines of code, but still took long time because they needed to be tested on the hardware. When the simulators were created development of the sample application went pretty quick, and it was easy to debug the application logic. If this would have been done on hardware not only would it have taking longer because of deployment, but it could be harder to identify which errors belonged to the driver and which belonged to the application. Integration of two Beagle Boards running the serial demo was also easier, since it required no setup of two Beagle Boards with a serial cable. One problem that could not be discovered on simulators was the need for a keep alive signal for the serial communication, since this issue was unknown during the creation of the simulator. Other benefits of simulation are that development can be started before hardware is available and/or at places where the hardware will never be available.

9 Future Work

9.1 Runtime function call replacement

By replacing function calls used for the hardware related code, you could write code for hardware as usual and start ports and use port commands. In a simulated environment, the function calls that start the port and execute port commands would be replaced by calls to the simulators instead. This simplifies the usage of testing with simulators and also makes it possible to easily use simulators in already existing code.

A downside is that when replacing function calls at runtime there is no explicit call to a specific simulator. This means that it might be unclear which simulator to start and what is the module and function name that will start it. One option could be to use a naming convention related to the name of the executable file run by `open_port`. Another option is to use a config file mapping ports to simulators. This way the call to `open_port` could be replaced with e.g. `serial_stub:start` instead.

Meck could be used to replace function calls [20].

9.2 Stdio Interface

Provide an interface via stdio so that tools and simulators created with Erlang Embedded Simulation can be used by other languages, like C or Java.

9.3 Recording and Replaying Messages Containing PIDs

It is a nontrivial problem to record and replay messages containing PIDs. PIDs consist of three positive integers, but are not represented in Erlang as a specified data structure. Because of this it is not possible to turn a PID into a string or use `file:consult/1` to turn it to a term. Other terms can be stored as strings in files and turned into terms. Another problem is that processes get assigned to different PIDs when running on different nodes. The concept of a PID is that it is unique. There is no guarantee that the PID of a process that was recorded will be assigned to a corresponding process or even exists when starting a new node. This means that all processes present in a recording needs to be registered, otherwise messages can not be replayed correctly since the PIDs have changed. PIDs inside the message also needs to be parsed and replaced with registered process names. Alternatively these processes could be stored with an initial call to `process_info/1`, and the PID could be replaced with the specific initial call that started the process.

One problem with replacing PIDs with registered names is that if `is_pid/1` is run on the message it will not give the desired effects. A solution would be to replace PIDs back again during replay. It would be necessary to keep track of which were registered names originally and which were PIDs. A problem with using `initial` call from `process_info` is that there may be several processes spawned from the same initial call, which makes it hard to separate the processes. In some cases the PID assignment might not be relevant as long as processes are spawned from the same initial call.

The problem might be circumvented by identifying common messaging patterns, e.g. synchronous Request-response[13] or publish/subscribe[14] are two messaging patterns. In request-reponse the PID in outgoing messages should be mapped to incoming send message. In publish/subscribe the PID in the outgoing messages, is a PID that has subscribed at some point.

9.4 Creating Finite State Machines from logs

Recording has great value for embedded simulation, but also elsewhere. Some interesting future work would be to create finite state machines out of recording logs of more complex applications. When a simulator receives a message it could consult a log file and perform a response, which could result in a reply. If several options are available the simulator could randomly choose the response which was most common during recording. Simulators could be fully generated from logs or finite state machines could be manually created were access to logs files could be used for any purpose.

9.5 Remote Testing

Running the test suite on a desktop and uploading the tests one by one to the Beagle Board could be a more convenient way of testing. It would also abstract log generation to desktop and might shorten time for running tests on hardware. A desktop computer needs to be connected through e.g. `ssh` to the embedded device. When the test suite server has been started on the desktop the tests will one by one be transferred to a test client on the embedded device and ran independently. All output would be piped through the `ssh` tunnel to the test server on the desktop, where it would be put together.

References

- [1] Apache License, Version 2.0. <http://www.apache.org/licenses/LICENSE-2.0.html>, March 2011.
- [2] BeagleBoard.org - default. <http://beagleboard.org/>, March 2011.
- [3] Erlang – dbg. <http://www.erlang.org/doc/man/dbg.html>, March 2011.
- [4] Erlang – erl_nif. http://www.erlang.org/doc/man/erl_nif.html, April 2011.
- [5] Erlang – Overview. <http://www.erlang.org/doc/tutorial/overview.html>, March 2011.
- [6] Erlang – Ports and Port Drivers. http://www.erlang.org/doc/reference_manual/ports.html, April 2011.
- [7] Erlang – Processes. http://www.erlang.org/doc/reference_manual/processes.html, April 2011.
- [8] Erlang Embedded. <http://www.erlang-embedded.com/>, March 2011.
- [9] Erlang Programming Language, Official Website. <http://www.erlang.org/>, March 2011.
- [10] Introduction — GNU Xnee. <http://www.sandklef.com/xnee/>, March 2011.
- [11] LaTeX project: LaTeX – A document preparation system. <http://www.latex-project.org/>, February 2011.
- [12] The GNU General Public License v3.0 - GNU Project - Free Software Foundation (FSF). <http://www.gnu.org/licenses/gpl.html>, March 2011.
- [13] Transport Message Exchange Pattern. http://www.w3.org/2000/xml/Group/1/10/11/2001-10-11-SRR-Transport_MEP, March 2011.
- [14] XEP-0060: Publish-Subscribe. <http://xmpp.org/extensions/xep-0060.html#intro-howitworks>, March 2011.
- [15] "Winsberg E.". "simulations, models and theories: Complex physical systems and their representations". *"Philosophy of Science 68 (Proceedings)"*, 2011.

-
- [16] Fabian Bergström Fredrik Andersson. Development of an Erlang System Adapted to Embedded Devices. February 2011.
 - [17] Tony Garnock-Jones. tonyg/erlang-serial - GitHub. <https://github.com/tonyg/erlang-serial>, March 2011.
 - [18] Olof Lindroth Niklas Een. The Erlang Port Recorder. September 1997.
 - [19] Reza Javaheri Rickard Olsson. EmbeddedErlang/Embedded-Erlang-Simulation - GitHub. <https://github.com/EmbeddedErlang/Embedded-Erlang-Simulation>, February 2011.
 - [20] Erlang Solutions. Meck. <https://github.com/eproxus/meck>, February 2011.
 - [21] Koen Claessen Thomas Arts and Hans Svensson. Semi-formal Development of a Fault-Tolerant Leader Election Protocol in Erlang. 2005.
 - [22] Francesco Cesarini & Simon Thompson. *Erlang Programming*. O'Reilly, 1 edition, 2009.
 - [23] Ruben Vermeersch. Concurrency in Erlang Scala The Actor Model. <http://ruben.savanne.be/articles/concurrency-in-erlang-scala>, January 2009.