A Low Cost Hardware in the Loop Simulator

Implemented with COTS development boards

Presented by:
Federico Lorenzi

Prepared for:
Amir Patel
Dept. of Electrical and Electronics Engineering
University of Cape Town

Submitted to the Department of Electrical Engineering at the University of Cape Town in partial fulfilment of the academic requirements for a Bachelor of Science degree in Electrical and Computer Engineering.

15th October 2013

Declaration

1. I know that plagiarism is wrong. Plagiarism is to use another's work and pretend that it is one's own.
2. I have used the IEEE convention for citation and referencing. Each contribution to, and quotation in, this report from the work(s) of other people has been attributed, and has been cited and referenced.
3. This report is my own work.
4. I have not allowed, and will not allow, anyone to copy my work with the intention of passing it off as their own work or part thereof.

Signature:.....
Federico Lorenzi

Date:.....

Acknowledgments

Firstly, I would like to thank my supervisor, Amir Patel, for proposing the topic, allocating it to me and being a great supervisor all round. It's been an interesting 3 months during which I've learnt a good deal.

Then, I'd like to acknowledge everyone who helped - especially with the little things that can so often turn into bigger problems. Names that spring to mind are Callen for providing me with a motor for performing quadrature measurements, and Kea for help with the soldering and assembly of the PCB.

To my parents, who have put up with my mess of a room, as well as my generally unfriendly state during the day - thanks for not disowning me!

While not directly related to the thesis itself, I feel Greg deserves a mention. Without his help, I can think of a few course results that might not have turned out to be so favourable.

To the community at `tex.stackexchange.com` - whenever something strange needed to be done with LaTeX, that was the place to turn to for help.

To Alice, who has put up with my constant whining and kept me supplied with cookies.

Lastly, to the countless volunteers who worked on making Linux and its surrounding ecosystem into what it is today. This project wouldn't be possible without such a solid base to work off.

Abstract

Hardware in the loop simulation is a widely used technique for testing controllers in an industrial setting. It works by allowing a controller to interact with a real-time simulation of the plant it will operate in. This is especially useful when testing a controller on a real plant is considered dangerous or not viable - such as in the case of jet engines.

Unfortunately, most hardware in the loop simulators thus far have been aimed at large companies with a suitably large budget - they often cost in the range of R200 000 and upwards. This precludes their use by more casual users, who do not necessarily need the same stringent quality guarantees that, for example, a military UAV would require.

This project aims to remedy that, by creating a low-cost real-time hardware in the loop simulator using a Beaglebone Black running Linux with Xenomai combined with Simulink. While Simulink already provides basic support for running models on the Beagleboard-xM, it is extremely limited and has unacceptable jitter. This will however be used as a reference implementation for comparing results.

To implement such a project, a hardware cape has been designed for the Beaglebone Black. It breaks out important IO such as GPIOs, PWM and quadrature inputs on screw terminals, and has transceivers for RS232, RS485 and CAN. In addition, it is equipped with a DAC. This hardware cape was coupled with a custom-built Linux/Xenomai real-time kernel, and a lightweight ArchLinuxArm GNU/Linux user space both running on the Beaglebone Black.

All of the hardware was verified manually, by controlling the devices from the GNU/Linux command line and verifying the results either on an oscilloscope or computer, where appropriate. While all of the hardware performed as expected, time constraints did not allow for a full Simulink library of interfaces to be built up.

Instead, GPIO, PWM output, ADC, and logging blocks were implemented and tested in Simulink. Finally, the system built was benchmarked and compared to the reference Beagleboard-xM - which it bettered in every benchmark performed. Perhaps most important was the jitter test, whereby the Beagleboard-xM was exposed without a real-time kernel, and had an “almost worst case” jitter figure of 1.84mS. This project improved on that by a factor of 5, resulting in a worst case jitter figure of 280μS.

Contents

1 Introduction **1**

1.1 Background to the study 1

1.2 Objectives of this thesis 2

1.3 Scope and Limitations 3

1.3.1 Hardware 3

1.3.2 Software 4

1.4 Plan of development 5

2 Literature Review **7**

2.1 Real-time operating systems 7

2.1.1 FreeRTOS 9

2.1.2 Linux 9

2.1.3 Summary 11

2.2 Hardware In the Loop 12

2.3 COTS development boards 13

2.3.1 STM32F4 Discovery 14
2.3.2 Raspberry Pi 15
2.3.3 Beagleboard-xM 16
2.3.4 Beaglebone Black 17
2.3.5 Summary 18

3 Methodology **19**

4 Design **21**

4.1 Historical Design 22
4.2 Design Overview 24
4.3 Hardware 25
4.3.1 Transceivers & ICs 26
4.3.2 PCB Design 27
4.4 Software 28
4.4.1 Arch Linux 29
4.4.2 Turnkey Operation 30
4.4.3 Simulink Integration 30

5 Implementation **33**

5.1 Hardware Implementation 33
5.1.1 PCB Assembly 34

5.1.2 Problems encountered 37

5.1.3 Modifications performed 39

5.2 Software Implementation 40

5.2.1 Linux/Xenomai Kernel 40

5.2.2 GNU/Linux Userspace 43

5.2.3 Deploying to SD card 44

5.2.4 PRU PWM Capture 48

6 Verification **50**

6.1 UART 51

6.1.1 RS232 52

6.1.2 RS485 52

6.2 GPIO 53

6.3 PWM Output 54

6.4 PWM Capture 56

6.5 ADC 58

6.6 DAC 60

6.7 EEPROM 61

6.8 Quadrature 62

6.9 SPI 64

6.10 Not completely verified 64

6.10.1 I^2C 65

6.10.2 CAN 65

7 Simulink Integration **66**

7.1 Compiler 66

7.2 Blocks created 68

7.2.1 GPIO block 68

7.2.2 PWM block 69

7.2.3 UART block 70

7.2.4 ADC block 70

7.2.5 Logging Blocks 70

7.3 Unimplemented blocks 71

7.4 Examples of blocks 72

7.4.1 GPIO example 72

7.4.2 PWM example 73

7.4.3 UART example 75

7.4.4 ADC example 76

7.4.5 Logging example 76

7.5 Running on target hardware 79

8 Benchmarking **83**

8.1 GNU/Linux performance 83

8.1.1 Results 85

8.1.2 Interpretation 87

8.2 Simulink performance 88

8.2.1 Throughput 89

8.2.2 Launch performance 90

8.2.3 Jitter 90

9 Conclusions **94**

10 Recommendations **96**

A Models & Code **101**

List of Figures

1.1 A block diagram illustrating a basic HILS setup. The Device Under Test interacts with the Hardware In the Loop Simulator via real-time communications. The HIL simulator then relays information back to a monitoring system. 1

1.2 The hardware implementation scope, separated into 3 distinct groupings. 3

2.1 A block diagram illustrating a basic RTOS setup. Here, the RTOS interfaces directly with the hardware, and controls 2 level 1 tasks, and a single level 2 task. The lower the level, the higher the priority. 8

2.2 Visualising the i-pipe concept. Interrupts are passed down between differing OS domains based on their ranking. 11

2.3 The 4 development boards that will be reviewed 13

2.4 The STM32F4 discovery board [14]. All pins from the microcontroller are broken out on dual-inline headers. 14

2.5 The Raspberry Pi. A limited selection of 3.3V logic pins are brought out on the expansion header P1, pictured here in the top left [16] 15

2.6 The Beagleboard-xM. An expansion header (flip-side, left) provides access to some of the 1.8V logic available. [20] 16

2.7 The Beaglebone Black. Most pins are broken out on the dual-inline sockets clearly visible and operate at 3.3V logic levels - with the exception of the ADC (1.8V max) [16] 17

4.1 Eye diagram showing the signal integrity of an ADG3304 at 50MHz and 3.3V to 1.8V. Notice the clear eye. The small amount of ringing is caused by the level shifter [23]. 22

4.2 Measurement of the rise-time on an HP 54615B 1GS/s 500MHz scope. . . 23

4.3 High level overview of the current system architecture running on the Beaglebone Black. The external Simulink interface is not shown. 25

4.4 Final PCB design. 28

4.5 UML diagram illustrating the initialise-on-use pattern. 31

4.6 UML diagram illustrating the design pattern followed with the S-Function builder. 32

5.1 PCB with solder paste applied. 34

5.2 PCB with components sitting on top of solder paste. 34

5.3 PCB with soldered components. 35

5.4 PCB with soldered capacitors. 35

5.5 PCB with soldered through-hole components. The solder bridges on U3 have been cleaned up. 36

5.6 Final assembled PCB. Here, it is mounted on the Beaglebone Black. . . . 36

5.7 Serial fix board mounted on the cape's UART headers, with jumpers in place. 39

5.8 UML diagram showing the flow of the PRU PWM assembly code. 49

6.1 Standard verification proceedure followed for peripherals. 51

6.2 PWM output signal. Channel 1 (yellow) has a duty cycle of 50%, whereas channel 2 (green) is 25%. Frequency is 100kHz for both. 55

6.3 Generated PWM signal at 70kHz with 50% duty cycle being sent to the Beaglebone Black. 57

6.4 Original input signal (yellow) at 5V peak-to-peak being attenuated to 1.61V (green) peak-to-peak for the ADC input. 59

6.5 Generated 12-bit sine wave from the DAC. 60

6.6 Quadrature output from motor's rotary encoder. Note that the scope was set to use a 10x probe, whereas crocodile clips were used, leading to an incorrect 10x increase in reported voltage. 63

7.1 Internal workings of the GPIO block. 68

7.2 Internal workings of the PWM output block. 69

7.3 Internal working of the UART block. 70

7.4 Internal working of the ADC block. 70

7.5 Internal workings of the text logging block. 71

7.6 Internal workings of the CSV logging block. 71

7.7 Example Simulink block diagram illustrating the use of the GPIO block. 72

7.8 Example Simulink block diagram showing the PWM output block in action. 74

7.9 Example Simulink block diagram containing the UART block. 75

7.10 Example Simulink block diagram using the ADC to transfer data to a local scope via external mode. 76

7.11 Example Simulink block diagram showing 2 text logging blocks and a single CSV logging block. 77

7.12 Step 1 of installing the Beagleboard support package - selecting the menu option. 79

7.13 Step 2 of installing the Beagleboard support package - select the package source as the internet 80

7.14 Step 3 of installing the Beagleboard support package - configure the Beagleboard package as to be installed. Note that the firmware update can be skipped, as it is only relevant for the Beagleboard-xM and not the Beaglebone Black. 80

7.15 After the support package has been installed, the model is prepared for running on the target platform. 81

7.16 The dialog for configuring the target hardware. Here, external mode is enabled (which allows communication back with MATLAB from the hardware) and the IP address is set to the Dynamic DNS hostname. 81

7.17 Finally, the sample time of the system is set to 1/500, or 500Hz. 82

8.1 Simulink block diagram used for testing CPU performance. 89

8.2 Starting point for measuring the launch performance of models on the different platforms. 90

8.3 Simulink block diagram used for jitter measurement on the Beaglebone Black. 91

8.4 Simulink block diagram used for jitter measurement on the Beagleboard-xM. 91

8.5 Scope capture for the generated pulse train from the Beagleboard-xM. . . 92

8.6 Scope capture for the generated pulse train from the Beaglebone Black. . 93

List of Tables

2.1 Comparison of a selection of freely available real-time operating systems. . 11

2.2 Comparison of a selection of low cost embedded systems boards. 18

4.1 Cost of ICs used on breakout cape. 27

8.1 Time for full operating system boot. 85

8.2 Number of processes running immediately after startup. 85

8.3 Comparison of network latency from the `ping` utility. 86

8.4 Sequential disk IO performance. 86

8.5 Results from UnixBench. 87

8.6 Time to compile UnixBench application. 87

8.7 5 minute average CPU usage when running the model described in figure 8.1. 89

8.8 Time to launch simulation. 90

9.1 Total cost of the low-cost hardware in the loop simulation system. 94

Abbreviations

ADEOS Adaptive Domain Environment for Operating Systems

COTS Commercial Off The Shelf

DAC Digital to Analog Converter

DHCP Dynamic Host Configuration Protocol

eQEP Enhanced Quadrature Encoder Pulse

HILS Hardware In the Loop Simulation

NC No Connection

NTP Network Time Protocol

PRU Programmable Runtime Unit

SMT Surface Mount Technology

SSH Secure Shell Protocol

TLC Target Language Compiler

A note on shell snippets

Throughout this report, various command line snippets appear - such as:

```
federico@desktop:~> mkdir BBBXenomai
federico@desktop:~> cd BBBXenomai
federico@desktop:BBBXenomai> cd /a/really/long/directory/path/that/has\|
/been/interrupted/with/a/backslash
```

These follow standard GNU/Linux shell terminology. Here, user `federico` is logged into the machine `desktop` and in the home folder, commonly represented as a tilda (~). A directory, `BBBXenomai` is then created and changed to. Notice that the shell prompt (the `federico@desktop`:~> part) changes depending on the current directory.

Occasionally when lines are too long they will end with a backslash (\) indicating that the next line is a continuation of the previous one.

Chapter 1

Introduction

1.1 Background to the study

Hardware In the Loop Simulation (HILS) is a well known and widely used technique for addressing the complexities in testing embedded systems. By providing a simulation of the plant involved, an embedded system can be tested with minimal to no modifications, in real world conditions.

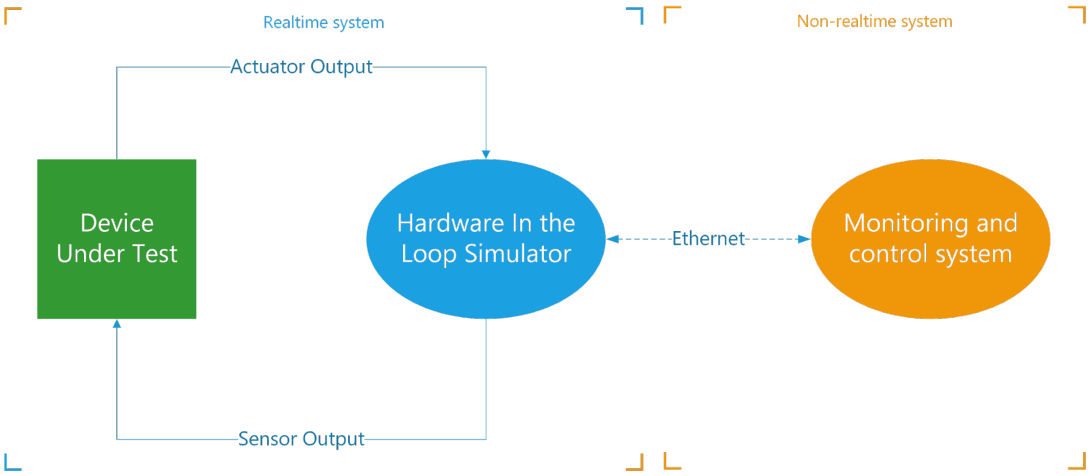

Figure 1.1: A block diagram illustrating a basic HILS setup. The Device Under Test interacts with the Hardware In the Loop Simulator via real-time communications. The HIL simulator then relays information back to a monitoring system.

While it is ideal to test an embedded system by connecting it directly to the plant, this is often either impossible or undesirable. HILS is most commonly used in situations where the plant would impose physical limitations on the testing (such as testing an automotive engine at or above its usual operating parameters) or where the plant itself is extremely expensive (such as in the case of jet engines).

1.2 Objectives of this thesis

> *"If you have to ask the price, you can't afford it*[1]*."*
>
> – J.P. Morgan

While the benefits of HILS are well known, a major barrier for more casual academic use is the cost involved. A typical HIL turnkey package, such as those from Speedgoat, are in the price range of hundreds of thousands of rand. [2]

While this price is not a problem for a large engineering company testing safety critical systems, it does present a high barrier to entry for academic or more casual hobbyist use.

Therefore, the primary objective of this thesis is to design, build and test a low-cost, real-time hardware in the loop simulation platform, using *Commercial Off The Shelf* (COTS) development boards[3] and full Simulink integration. The secondary objective is to provide a platform for rapid prototyping and development, which follows on from the primary objective.

Achieving these objectives requires the careful partitioning of the project into hardware and software components. For the hardware side, a custom board will be designed with common transceivers used by typical embedded systems - such as RS232 - to make interfacing with existing embedded systems as easy as possible. The software side will be concerned with achieving real-time performance as well as Simulink integration - such as writing the blocksets needed to utilise the hardware, and lastly turnkey like operation[4].

[1] The story goes that Morgan was asked a question about the cost of maintaining his yacht. Unfortunately, the accuracy of this is unconfirmed.

[2] There aren't any list prices available for reference, but research revealed the general cost at being over R200 000

[3] The Beaglebone Black was used specifically, reasoning as to this choice is given in section 2.3

[4] By turnkey, it is meant that the platform must be easy to use and get started with - similar to the commercially offered solutions

1.3 Scope and Limitations

As mentioned previously, it is best to split the project into hardware and software components, and deal with them individually. While occasionally these two intermix[5], for the most part they can be kept separate.

1.3.1 Hardware

Scope

The scope of the hardware component of this thesis therefore centres around building a breakout / transceiver board, and choosing the COTS development board to use as a base. After much research and consultation, a final hardware feature set was decided upon:

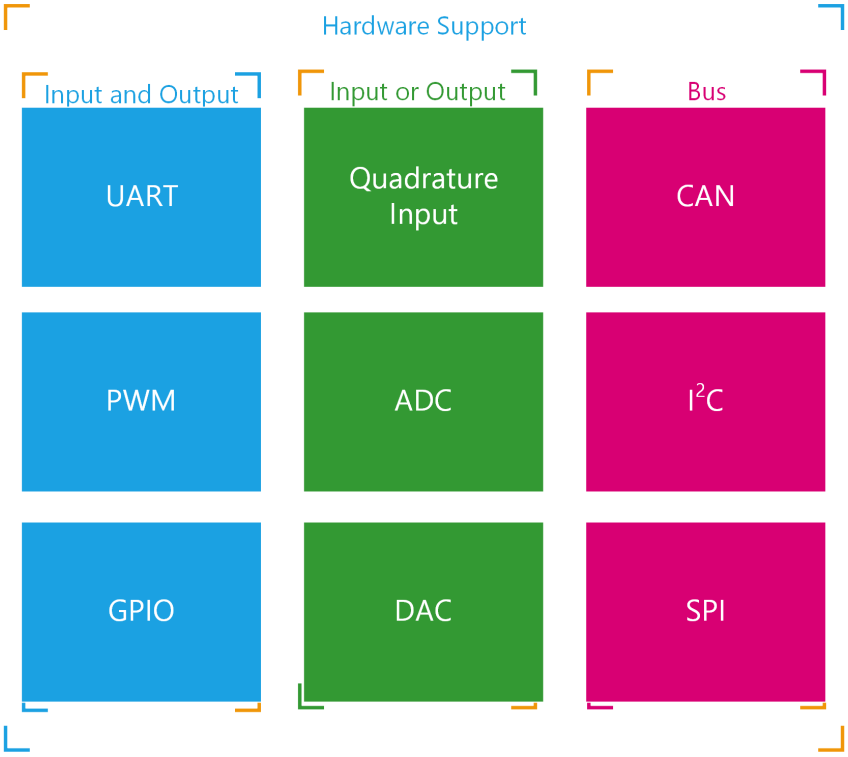

Figure 1.2: The hardware implementation scope, separated into 3 distinct groupings.

These will all be exposed on screw terminals, with the exception of RS232 being brought out on a standard DB9 connector.

[5]An example would be pin-muxing, whereby the feature of a specific processor pin is configured from a list of available options in software

Limitation

Due to the number of connector blocks required to expose the signals, the PCB produced has to be quite large, and thus expensive.

1.3.2 Software

Scope

The software component will focus on integration with Simulink, writing drivers for the hardware, as well as ensuring real-time operation. In order to ensure the lowest cost possible, Simulink's built in *Run on Target Hardware* support will be used as a basis for integration as it requires no additional licenses.

In order to access the hardware functionality, Simulink blocks will need to be produced. These are:

- Logging Block
- UART Block
- Quadrature Block
- PWM Output / Input Block
- GPIO Block
- ADC Block
- DAC Block

While the blocks mostly correspond with the hardware features provided, the notable exceptions are CAN, I^2C and SPI. The reason for this is a design decision - devices on these buses can vary wildly in terms of what data they expect, and using a block diagram environment to try and program them would complicate matters significantly. Therefore, in order to use such devices, a small S-Function will need to be written for the particular target device.

Limitation

Due to relying on *Run on Target Hardware*, which uses a restricted variant of *Simulink Coder*, it is unfortunately not possible to utilise a continuous solver for models. Should a license for *Simulink Coder* be made available, this limitation would be lifted.

In addition, due to the real-time nature, a fixed-step solver must be used.

1.4 Plan of development

This report begins with a literature review for **chapter 2** that evaluates the current state of the art in commercial off the shelf development boards as well as real-time operating systems.

This is followed by the methodology in **chapter 3**, which explains the different stages of the project and how tasks were handled and broken up.

Chapter 4 contains the design, and it covers the hardware and software design process - focusing on why particular choices were made, and how they affect the system as a whole.

Chapter 5 deals with the implementation, of both hardware and software. The hardware side focuses on the PCB and its assembly - in particular the errors made are covered. The software side deals with compiling and patching the Linux kernel appropriately, in addition to building a GNU/Linux userspace and deploying them both to the Beaglebone Black.

The verification step is performed in **chapter 6**. This is where the interaction between hardware and software is verified, either by using an oscilloscope to measure signals, or a computer to send / receive data.

Chapter 7 handles integrating the system created so far with Simulink's Run on Target Hardware support, giving arbitrary Simulink models the ability to run on the real-time platform with access to IO.

Benchmarking of the system is performed in **chapter 8**, where it is extensively compared to the standard Beagleboard-xM support provided by Simulink.

Lastly, in **chapter 9** and **chapter 10** conclusions are drawn and recommendations made from the data obtained.

Chapter 2

Literature Review

Creating a low-cost hardware in the loop simulator requires pooling together knowledge from widely varying sources. On one side, a hardware base platform needs to be selected which has suitable peripherals. On the other, a fast software stack is required to make efficient use of said hardware.

Therefore, this literature review starts out with evaluating the current state of the art in open source real-time operating systems, before looking at the current state of hardware in the loop simulation in general. Lastly, COTS development boards are assessed based on relevant criteria pertaining to the project.

2.1 Real-time operating systems

> *"Those who don't understand Unix are condemned to reinvent it, poorly."*
>
> – Henry Spencer

For simple projects and tasks, it is common to run one's application on the bare-metal. However this soon becomes problematic when different tasks or sub-applications need to interface with one another. Resources such as memory and CPU time need to be carefully controlled, ensuring that a single task doesn't end up using an unfair share, while at the same time dealing with IPC and the possibility of a runaway task[1].

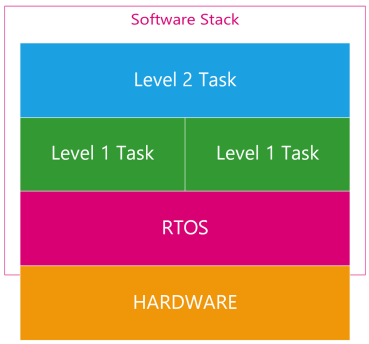

Figure 2.1: A block diagram illustrating a basic RTOS setup. Here, the RTOS interfaces directly with the hardware, and controls 2 level 1 tasks, and a single level 2 task. The lower the level, the higher the priority.

In order to prevent such unpleasantries, a common idea [1] [2] [3] is to design a layer that sits below all the regular tasks and mediates their access to the hardware and one another. Instead of directly calling `malloc()` to allocate memory, higher level tasks could instead call `myCustomMalloc()` - where it might be implemented as follows:

```
void* myCustomMalloc(size_t size, int task_id) {
        if ((allocated_memory[task_id] + size > MAXIMUM_TASK_MEMORY) ||
                total_allocated_memory + size > MAXIMUM_TOTAL_MEMORY) {
                /* Return the NULL pointer if we "cannot" allocate memory */
                return 0;
        } else {
                allocated_memory[task_id] += size;
                total_allocated_memory += size;

                /* Call the original malloc and allocate real memory. */
                return malloc(size);
        }
}
```

There are a few flaws in such a simple implementation - here, it is completely non-reentrant. While many microcontrollers are based around a single core, they utilise interrupts which are completely asynchronous and thus tend to occur at the worst possible time.

In this case, suppose `total_allocated_memory` is nearing `MAXIMUM_TOTAL_MEMORY`, and a task calls `myCustomMalloc`(). At the same time, an interrupt occurs and allocates some memory by also using `myCustomMalloc`(). The interrupt gets allocated the last available block of memory, and upon competition flow returns to normal. Unfortunately, before the interrupt had occurred line 6 had already been reached. The `myCustomMalloc`() function has now allocated more than the amount of memory it was meant to.

Of course, the example above is contrived somewhat to illustrate a point, but there are countless examples of even professional operating system programmers getting such details wrong[1].

2.1.1 FreeRTOS

FreeRTOS is a popular, open-source RTOS. It supports a wide range of platforms, including the STM32F4 as well as more fully-fledged systems such as the Raspberry Pi.

It is based around a queuing architecture, where queues form the primitives on top of which other synchronisation and communication tools are built from. In addition, it provides both preemptive and cooperative scheduling options, with a ROM footprint of around 10K. It has an extremely high quality of code, being coded to ensure deterministic operation [4].

FreeRTOS provides custom memory allocation options, which ensure determinism - unlike the standard C `malloc` and `free`.

2.1.2 Linux

> *"I started Linux as a desktop operating system. And it's the only area where Linux hasn't completely taken over. That just annoys the hell out of me."*
>
> – Linus Torvalds

[1] There was a case in 2003, where a missing equals sign would have led to a full privilege escalation vulnerability.

Linux is a popular free operating system kernel[2], originally created in 1991 by then-student Linus Torvalds.

However, it is not nearly close to what most people would consider to be an RTOS. Common GNU/Linux distributions are geared towards either server use, desktop use or the mobile space[3]. However, therein lies one of the major strengths of Linux - its flexibility. More specialized, stripped down versions can be found on most common wireless routers - often running with 16MB of RAM and 4MB of ROM. [5]

It is clear that these requirements place far higher demands on a Linux-based system than on a true lightweight RTOS such as FreeRTOS. While as mentioned above, FreeRTOS will be perfectly content with 192KB of RAM, such as found on the STM32F4 discovery board, Linux demands about 2 orders of magnitude more for a useful system [6]. This is due to both the large level of abstraction provided by a full operating system kernel, as well as the existing application compatibility that is provided. A full diagram of the Linux kernel can be found in Appendix A.

By default, Linux has absolutely no support for acting as a real-time operating system kernel; and this is not surprising[5]. However through the work of developers and researchers various *real-time patchsets*[4] have been created. These are modifications which apply to the existing Linux kernel tree, granting it real-time capabilities. While there are numerous such patchsets, they tend to fall into two separate camps - either the native preemption model or the dual-kernel model[6]. Both of these will be reviewed in more detail.

PREEMPT_RT

PREEMPT_RT takes the first approach, by making the Linux kernel entirely preemptable. It does this by making all of the internal kernel locking primitives preemptable, and additionally converts any kernel interrupt handlers into preemptable kernel threads. Furthermore, it brings out higher resolution timers - aiding user space directly.

[2]Often, the term "Linux" is used to refer to the *distribution* as a whole, such as "That computer runs Linux". This is much to the chagrin of the GNU team, who believe it should be called GNU/Linux. In this report, GNU/Linux will be used to refer to the combination of a GNU userspace running with a Linux kernel, whereas Linux will be used to refer to the kernel.

[3]Android utilises the Linux kernel

[4]With Linux, what's known as a "mainline" tree is maintained by Linus Torvalds. This is what is canonically known as Linux. However, many individuals and companies maintain their own patches or modifications to the mainline kernel. Often, they will try to reintegrate these changes, a process known as mainlining.

Xenomai

Xenomai takes the second approach. It is based off work done by the *Adaptive Domain Environment for Operating Systems* (ADEOS) project on their interrupt pipe subsystem[7]. The i-pipe system works by dealing with hardware interrupts in a deterministic and predictable fashion, depending on the level handling the interrupt[3]. It is pictured below in figure 2.2.

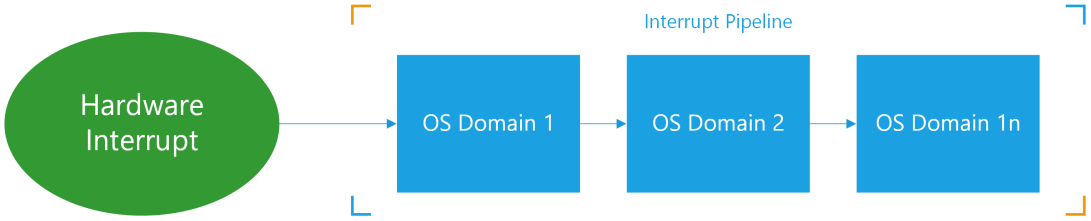

Figure 2.2: Visualising the i-pipe concept. Interrupts are passed down between differing OS domains based on their ranking.

This allows for more precise interrupt handling by essentially pushing the Linux kernel down the chain - as detailed by Yaghmour in the original paper on ADEOS [3].

2.1.3 Summary

In summary, the state of the art of relevant real-time operating systems can best be summed up by the following table:

Platform	Required Hardware	Abstraction	Real-time performance[5]
FreeRTOS	Microcontroller	Low	High
Linux	Full system (MMU)	High	Low
Linux & PREEMPT_RT	Full system (MMU)	High[6]	Medium
Linux & Xenomai	Full system (MMU)	Medium[7]	High

Table 2.1: Comparison of a selection of freely available real-time operating systems.

[5]Real-time performance can be measured in terms of average and worst case jitter

[6]PREEMPT_RT patches into the standard Linux RT functions, giving most things an automatic boost

[7]Xenomai supports similar usage to PREEMPT_RT, but to gain extra performance code must be written

2.2 Hardware In the Loop

Hardware in the loop simulation is a well known technique for the testing of embedded systems. As mentioned previous, it does this by providing an accurate simulation of the plant involved, while having the capacity to connect to the embedded system under test. All this occurs without the embedded system "knowing" that it is being tested. From its viewpoint, it is connected directly to the plant.

This allows for detailed testing, as the hardware in the loop simulator can be made to react in ways that would be dangerous or impossible to reproduce with an actual plant. For example, an engine ECU might be tested while placed under extreme temperatures whereas doing so with an actual car and driver would be considered hazardous [8].

Since hardware in the loop testing is such a useful tool, there is not much recent literature available on the subject, as it has been implemented well in industry for many years. Instead, the cutting edge of HILS lies in power electronics.

Switching electronics in the power electronics field often run at rates in the kilohertz region. Implementing a simulator on a standard processor that would be capable of handling such rates - recalling that to achieve good simulation one needs a simulation rate an order of magnitude or so greater than the control loop - is almost impossible. Instead, FPGAs are used [9] as they can provide single cycle operation for complex simulations.

Unfortunately, not much of the current literature on HILS is relevant to this project, with most useful information being sourced from the late 1990s [10].

2.3 COTS development boards

In recent times there has been an influx of low cost, high performance development boards. These development boards have made accessible processors which were previously out of reach, due to sheer quantities required or other aspects such as NDAs.

In terms of microcontrollers, the STM32F4 development board by ST is a stand-out example[11]. For more fully-fledged systems, the Beaglebone Black and Raspberry Pi have become well known in their short period of existence.

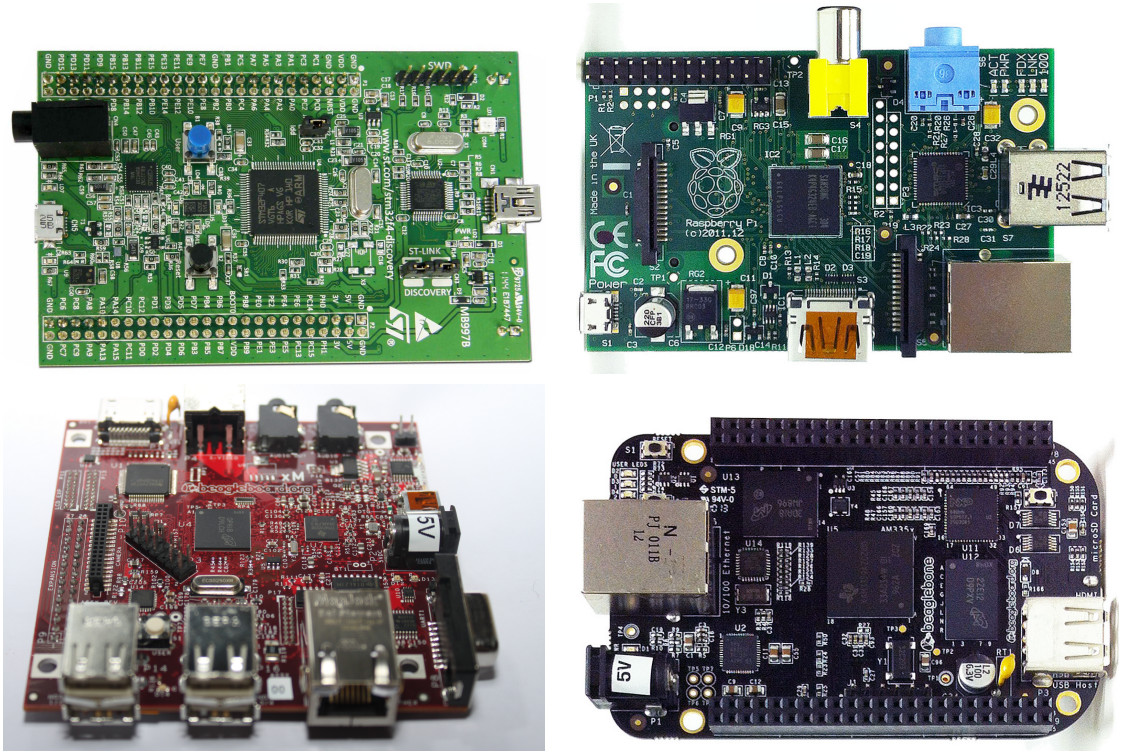

Figure 2.3: The 4 development boards that will be reviewed

Each of these options will now be analysed in more detail, with a view to be used as a base platform for the hardware in the loop simulator.

2.3.1 STM32F4 Discovery

The STM32F4 discovery was announced in late 2011 by ST Electronics. It contains an STM32F407VGT6 microcontroller, with a 32bit ARM Cortex-M4F[8], 1MB of Flash and 192KB of RAM[12]. Priced nominally at $15 [9], it is pictured in figure 2.4 below.

The STM32F4 carries the advantage of running at a 3.3V logic level, but still being tolerant of 5V signals. The majority of the pins can handle a 5V logic signal without a problem. In addition, it has an extremely wide range of onboard peripherals, including 3 ADCs, 2 DACs, as well as a wide range of digital buses (I^2C, CAN, SPI) and a large number of GPIOs and timers[13].

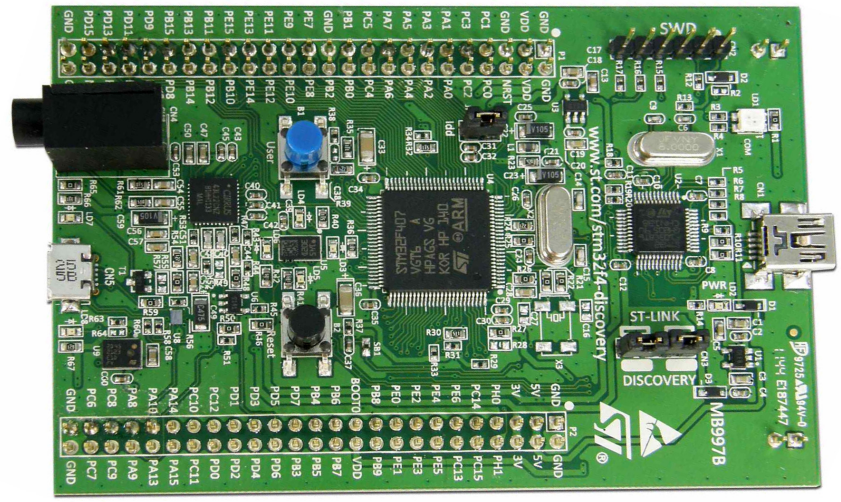

Figure 2.4: The STM32F4 discovery board [14]. All pins from the microcontroller are broken out on dual-inline headers.

In addition to the STM32F407VGT6 microcontroller on-board, the discovery board also has a wide range of sensors - such as gyroscopes, accelerometers as well as a selection of LEDs and buttons. It is coupled with an ST-LINKv2 programmer, allowing for easy development with no need to purchase additional programing tools. However, one such disadvantage of this approach is the use of pins which would normally be exposed[11].

It is impossible to disconnect these onboard peripherals from the microcontroller itself - and as such care needs to be taken when deciding on pin mapping. The extra peripherals can load down or even conflict with signals on the pins that are utilised[12].

[8] The F indicates floating point support
[9] Price obtained from digikey.

2.3.2 Raspberry Pi

The Raspberry Pi was released in early 2012 by the Raspberry Pi foundation. Aimed at being a low cost single board computer geared towards education, the Pi has an ARM1176JZF-S processor running at 700MHz, with 512MB of RAM [15] and is nominally $35[10].

It exposes a 26 pin expansion header as can be seen below in figure 2.5, which primarily has GPIO signals, as well as SPI and I^2C busses. All signals are 3.3V level, and not 5V tolerant [15].

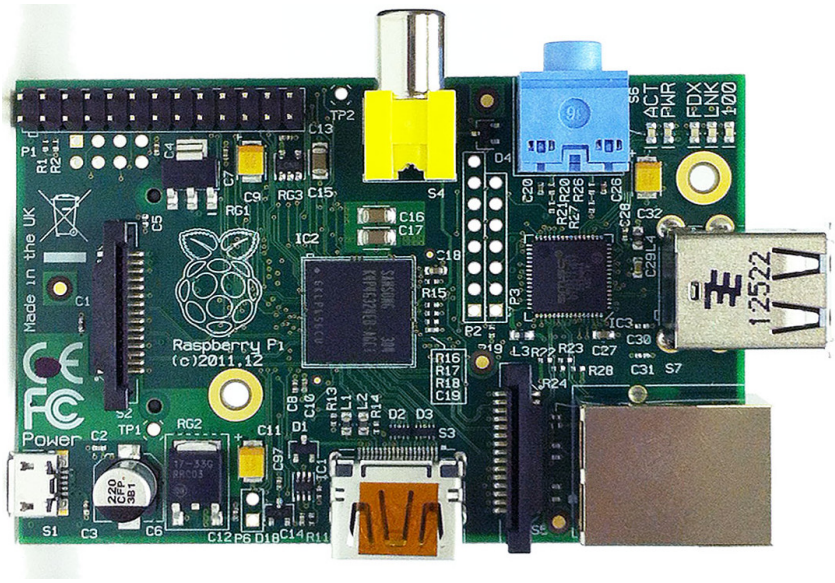

Figure 2.5: The Raspberry Pi. A limited selection of 3.3V logic pins are brought out on the expansion header P1, pictured here in the top left [16]

The Raspberry Pi has an extremely large amount of both community and commercial support behind the project. This is due to its low cost, allowing for easy experimentation and its use as a platform for research.

Unfortunately, the ARM11 core it provides - while coupled with an extremely powerful graphics processor - is a weak point. The 700MHz ARM11 core is a generation behind the Cortex-A8 which is used in other low cost development platforms, being almost 40% slower clock for clock [17] [18].

By default, the Raspberry Pi runs a Linux kernel, although a community effort has ported FreeRTOS to the platform as a bare-metal option. This port is still unstable, and has not been considered.

[10]Nominal price from the Raspberry Pi foundation.

2.3.3 Beagleboard-xM

The Beagleboard-xM was released in late 2010, as a successor to the original Beagleboard. It features a 1GHz ARM Cortex-A8 processor made by TI, as well as 512MB of RAM [19] and is nominally priced at $150[11].

The Beaglebone-xM was designed primarily to be used as a teaching tool for university students - being based on open hardware and open software - and not aimed at the so-called "maker" movement [12]. As a result of this, the Beagleboard-xM is quite limited in its external IO capabilities. It has a single 28 pin expansion connector, that brings out selected IO pins from the main processor. This can be seen in figure 2.6 below.

A major hurdle for entry level use of the Beagleboard-xM is the logic level used. Unlike common microcontrollers which use 5V or 3.3V logic, the Beagleboard-xM uses 1.8V logic [19] . While this is common for high volume manufactured consumer electronics (such as cell phones) - it is relatively rare outside of power saving microcontrollers. This presents a significant barrier to use, as any logic signals now need to be level shifted appropriately, to interface with common sensors and actuators.

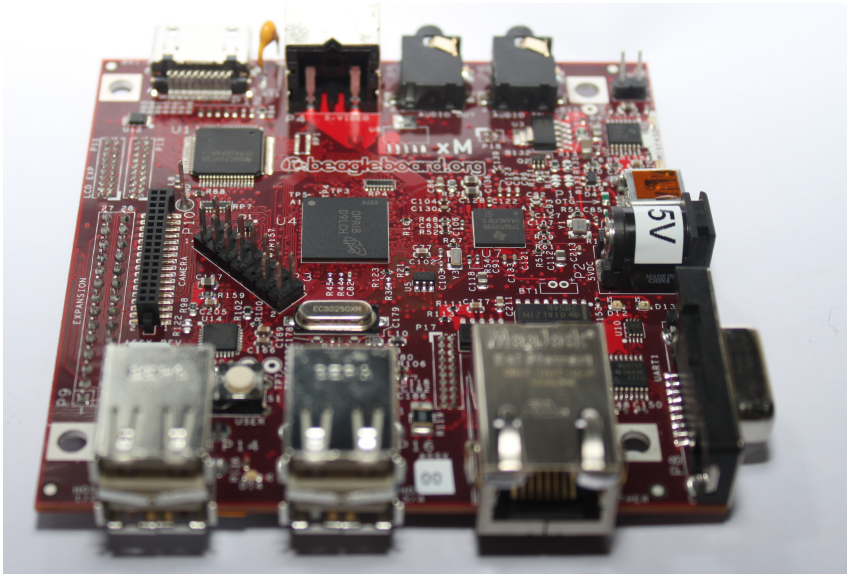

Figure 2.6: The Beagleboard-xM. An expansion header (flip-side, left) provides access to some of the 1.8V logic available. [20]

The Beagleboard-xM primarily runs GNU/Linux, although other operating systems have been shown running on it experimentally.

[11] Price from the Beagleboard community

[12] The maker movement is essentially centred around easy to get started hardware development - the Arduino is often viewed as a prime example.

2.3.4 Beaglebone Black

The Beaglebone Black was released by TI in mid 2013, and also contains a 1GHz ARM Cortex-A8 processor made by TI, with 512MB of RAM and additionally 2GB of built-in flash memory [21] nominally priced at $45.

The primary difference between the Beagleboard-xM and the Beaglebone Black is the target audience. While the Beagleboard-xM was geared towards more academic and teaching use, the Beaglebone Black is geared towards practical and "maker" use. To this end, it is equipped with a 46 pin dual-inline socket on either side of the board, for a total of 92 pins - as can be seen below in figure 2.7. This is far more than the 28 pins exposed by the Beagleboard-xM.

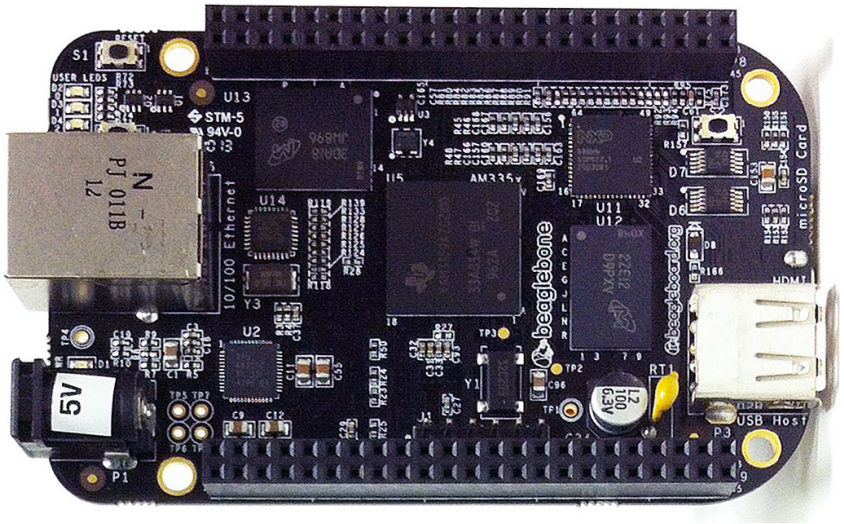

Figure 2.7: The Beaglebone Black. Most pins are broken out on the dual-inline sockets clearly visible and operate at 3.3V logic levels - with the exception of the ADC (1.8V max) [16]

In addition to the large amount of pins exposed, the Beaglebone Black has a large number of on-board peripherals - such as 2 PRU cores, a built-in ADC as well as quadrature and PWM modules. These pins also operate at a 3.3V logic level [21].

The PRU cores are of special interest when dealing with real-time control systems. They are hard real-time 200MHz ARM11 cores, with each instruction taking a single cycle and a non-pipelined design. In addition, they can communicate with the main Cortex-A8 core via DMA or interrupts, and can only be programmed in assembly. This complicates using them, but allows for precise control over fast signals.

Device tree

> *"Somebody needs to get a grip in the ARM community. I do want to do these merges, just to see how screwed up things are, but guys, this is just ridiculous. The pure amount of crazy churn is annoying in itself, but when I then get these 'independent' pull requests from four different people, and they touch the same files, that indicates that something is wrong."*
>
> – Linus Torvalds[13], in the email that sparked the device tree for ARM.

The device tree was first implemented in the Linux 3.8 kernel for the Beaglebone Black, after Linus Torvalds complained about the current state of having board dependent board file scattered all throughout the kernel tree. It is a method for managing and configuring the hardware on embedded systems [22].

The concept was extended by members of the community into the device tree overlay. The overlay feature allows device trees to be stacked upon one another, all managed by the so called `capemgr`. The current state of loaded overlays can be viewed by reading the file `/sys/devices/bone_capemgr.9/slots`, whereas a device tree overlay can be loaded by writing its name to the same file.

2.3.5 Summary

The 4 COTS boards reviewed can be be summarised into their salient features, as is done in table 2.2.

Platform	Operating System	IO capabilities	Performance	Cost
STM32F4 Discovery	None / FreeRTOS	High	Low[14]	Low
Raspberry Pi	FreeRTOS / Linux	Medium	Medium	Medium
Beagleboard-xM	Linux	Low	High	High
Beaglebone Black	Linux	High	High	Medium

Table 2.2: Comparison of a selection of low cost embedded systems boards.

[13] Linus is well known for his *direct* attitude to management

[14] The STM32F4 has excellent performance, for a microcontroller, but it cannot compete with fully-fledged processors at 1GHz+

Chapter 3

Methodology

"If you don't know where you are going, you'll end up someplace else."

– Yogi Berra

The methodology used for this project can best be equated to the test driven development in software. At each stage, a goal was set as a marker, and the implementation was iterated until that design was met. This helps to rapidly converging on a solution.

The project can thus be divided up into manageable stages, each with a clear goal.

Stage 1 focuses on choosing the correct tools and hardware to use as a base platform. This is based on the information gathered from current sources of literature in section 2.3. In particular, the outcome of the stage is a decision on the COTS development board to use as a base, as well as the operating system to place on said board.

The primary criteria involved are a mixture of performance, cost and IO capabilities. All three of these are critically important, as they form the base for the rest of the work that shall be performed.

Stage 2 concerns establishing the operating system chosen on the development board picked. First, the design of the system is carefully planned, as is done in section 4.4. Then, the software system is implemented as is done in section 5.2. The outcome of this is to have a basic operating system up and running on the chosen COTS development board, suitable to use as a base for future work.

The criteria involved in evaluating this step relate to the successful implementation of an operating system that boots and is usable for the purposes described.

Stage 3 relates to building the hardware that will be attached to the development board for the purpose of IO access. The desired peripherals laid out in the hardware scope, section 1.3.1, were carefully considered in the design described in section 4.3. This is then put into practice, with the designed PCB being assembled and verified at a hardware level in section 5.1.

The criteria for this stage centre around producing the hardware cape that will provide access to all the required hardware features.

Stage 4 revolves around integrating stages 3 and 4, namely the verification of all of the hardware peripherals by utilising them from the operating system. This is described in depth in chapter 6 where each hardware peripheral is tested from software to ensure that both the drivers and hardware is fully functional.

The goal for this stage is to have the knowledge that the hardware and software are fully functional together, and all interactions are as desired.

Stage 5 deals with tying the work done so far into Simulink. Now that a fully featured and tested environment has been created, Simulink integration can happen with the confidence that if anything fails to work, the fault lies in the Simulink integration stage, and not anywhere else on the chain. This is detailed in chapter 7.

The result of this stage is a set of Simulink blocks which can be used to allow a Simulink model to interact with embedded systems through common protocols defined in the hardware scope, section 1.3.1.

Stage 6 concludes by doing performance tests on individual parts of the system, as well as the system as whole. As described in chapter 8, the GNU/Linux performance of the system as a whole is evaluated and compared to that of a reference platform.

Even though performance tests have been carried out throughout the implementation life cycle, these tests are a more formal evaluation of the performance of the system. Their purpose is to determine both the throughput capacity and jitter of the system, and compare to the reference platform available.

Chapter 4

Design

"No battle plan survives contact with the enemy."

– Moltke the Elder

Designing and building a low cost hardware in the loop simulator requires pulling together both hardware and software components to create an integrated platform for testing and development. After careful review of the available literature presented in section 2.3 and considering the resources available, the Beaglebone Black shall be used as the hardware base platform - with GNU/Linux & Xenomai making up the foundation of the software stack.

As mentioned, the Beaglebone Black has excellent IO capabilities, in addition to its low price tag and good community support. This allows for minimal level shifting, with the only transceivers needed for RS232, RS485 and CAN busses. Additionally, a 12-bit I^2C DAC was used to provide analogue signal output.

However, this was not always the case. As will be explained the design underwent a major revision in late August.

4.1 Historical Design

It is worth noting that the Beaglebone Black wasn't a viable option when first analysed. While the hardware was released in April 2013, the initial Xenomai port was only completed near the end of August. Thus, it would have been unable to meet the real-time criteria.

Instead, the original plan involved a Beagleboard-xM coupled with an STM32F4 discovery board. As shown in table 2.2 on page 18, the STM32F4 has excellent IO capability, while the Beagleboard-xM brought with it processing resources. A PCB was designed, which can be seen in Appendix B, to break out the peripherals of the STM32F4 and link it to the Beagleboard-xM via high-speed SPI. Since the Beagleboard-xM operates at 1.8V logic levels, an Analog Devices ADG3304 bidirectional level translator was specified for use between the two boards.

The ADG3304 is a high-speed level shifter, capable of data rates of up to 50MHz. Below in figure 4.1 is an *eye diagram* of the output when shifting between 3.3V and 1.8V which shows the good level of signal integrity maintained by the level shifting process.

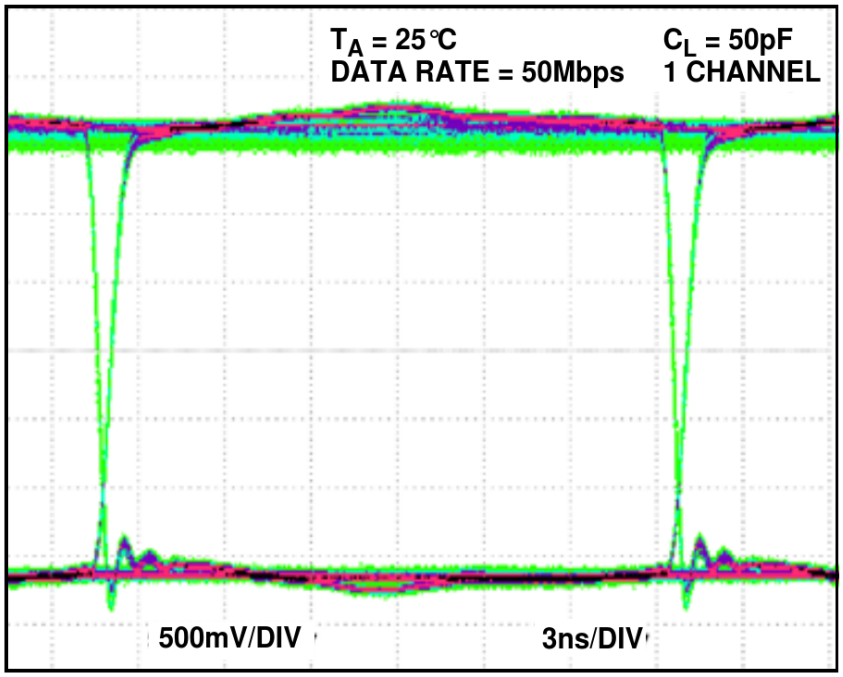

Figure 4.1: Eye diagram showing the signal integrity of an ADG3304 at 50MHz and 3.3V to 1.8V. Notice the clear eye. The small amount of ringing is caused by the level shifter [23].

The SPI bus would have run at 42MHz, or 42Mbps. Such a high data rate was chosen to minimize both jitter and transmission latency; the Beagleboard-xM would have to communicate with the STM32F4 over SPI in order to communicate with the outside world, and thus any extra delay induced is undesirable.

However, this introduced complexities at a hardware level. As a rule of thumb, one can treat traces on a PCB as a lumped element instead of a transmission line when the wavelength of the signal is smaller than $\frac{1}{10}$ of the trace length. Naïvely, one might assume that a 42MHz signal with a wavelength of 7m would not be a problem on a PCB, as $\frac{1}{10} \times 7 = 0.7m \gg$ track length

Unfortunately, this would be misguided. The high frequency components of the signal are conveyed in the rise-time of the square wave, and thus this is the information that needs to be measured. As is revealed in figure 4.2 below, the rise time is around 2.4ns, corresponding to a frequency of 420MHz [24]. This is essentially 10 times quicker than previously calculated.

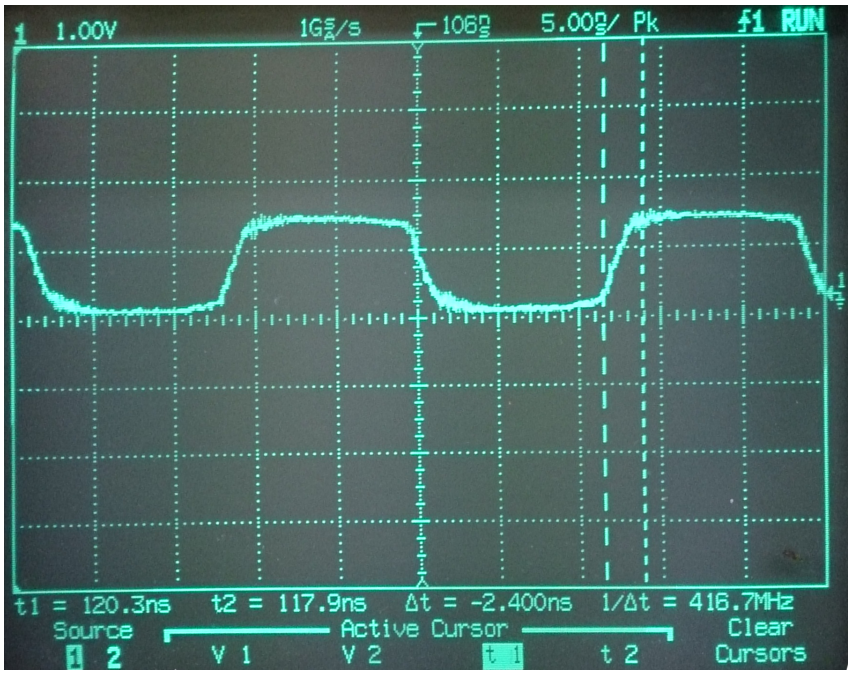

Figure 4.2: Measurement of the rise-time on an HP 54615B 1GS/s 500MHz scope[a].

[a]Unfortunately, this is just a photo of the screen of the scope. It was not possible to use the much newer and nicer Agilent 2000X scopes available in the lab, as even though they have 2GS/s ADCs, they have a -3dB bandwidth limit of 70MHz

While the STM32F4 does have slew rate control, it cannot be used for SPI at its maximum frequency [13]. Thus, the distance between the STM32F4 and Beagleboard-xM had to be kept to a minimum, in order to ensure good signal integrity and avoid transmission line effects.

As can be seen from the issues involved in simply getting the inter-board communication working, a single board solution greatly cuts down on the complexity involved. Thus, when it was revealed the Beaglebone Black would be capable of handling both the real-time computation and IO, focus was shifted to using it as a base platform. In addition, its lower cost ($45 vs $150) provides further incentives for use.

4.2 Design Overview

The current design therefore uses a Beaglebone Black as the base processor, coupled with a specialised cape[1]. This cape is a PCB that fits onto the Beaglebone Black's headers and allows for use of the IO available. It has 2 DB9 connectors, attached to RS232 transceivers, as well as numerous screw terminals exposing all the desired functionality of the other transceivers on board. A high level overview is shown below in figure 4.3

As can be seen from figure 4.3, the Beaglebone Black lies at the lowest level, with the cape providing access to its on board peripherals via suitable transceivers and terminals. Xenomai runs directly on the hardware, and using its interrupt pipe capabilities, runs the Linux kernel directly above it.

The GNU/Linux userspace runs on the Linux kernel as normal, with the simulated model being a standard GNU/Linux application. However, the simulated model has been hooked with Xenomai code, and thus can directly access the hardware for critical timing and real-time operations.

[1] Beaglebone add-on boards are referred to as *capes*

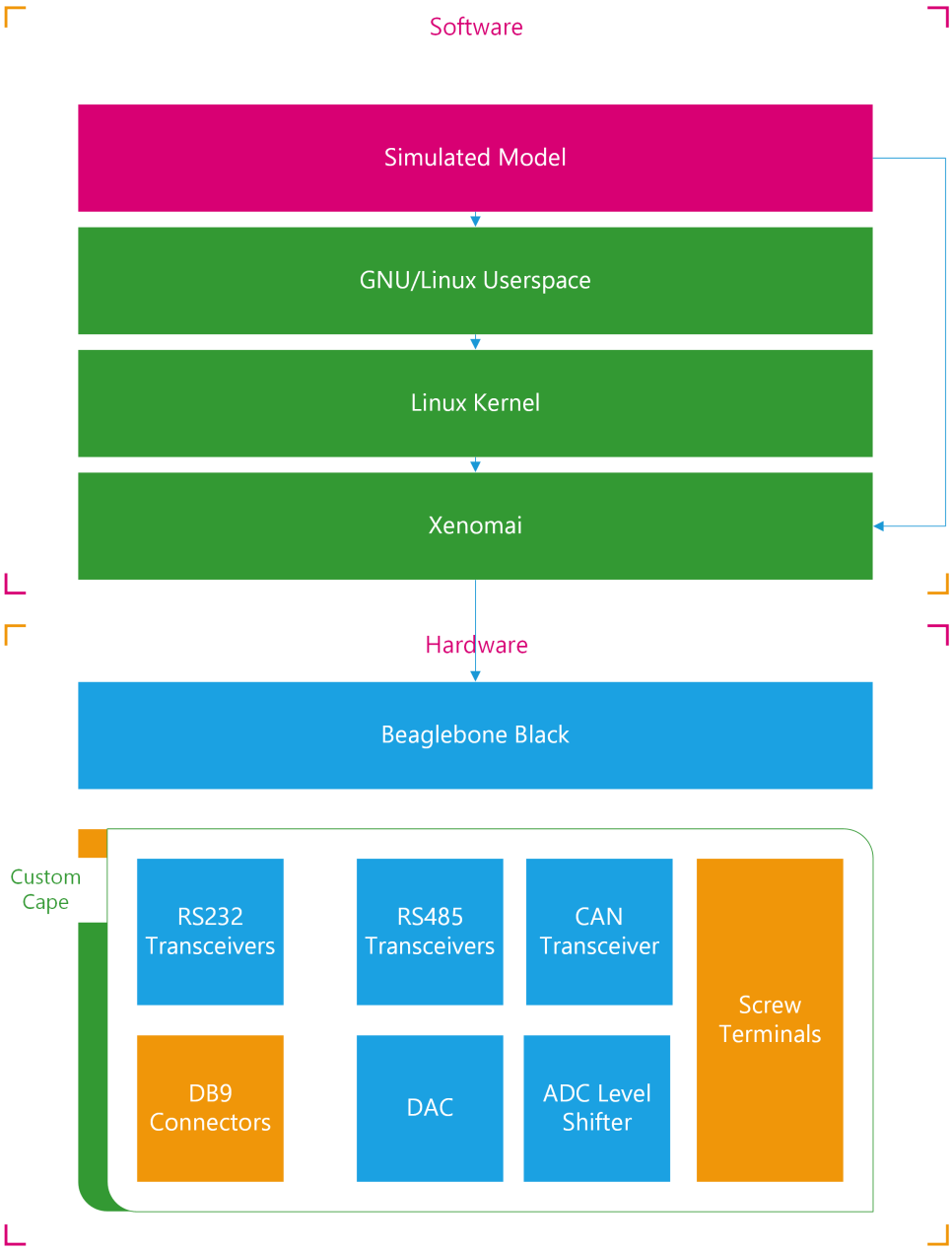

Figure 4.3: High level overview of the current system architecture running on the Beaglebone Black. The external Simulink interface is not shown.

4.3 Hardware

The hardware design component will focus on choosing suitable transceiver ICs, as well as designing the cape PCB.

4.3.1 Transceivers & ICs

In order to meet the stated objective[2] of being able to interface easily with existing hardware, transceivers are needed. Specifically, for RS232, RS485 and CAN bus. Starting with RS232, the most well known transceiver is the MAX232. It uses a charge pump topology to convert from TTL level signals to those used for RS232.

Instead, of the common MAX232, a TI version of the much newer MAX3237 was used instead. This part requires only 4 0.1μF capacitors at a 3.3V supply level, with an additional one for power supply decoupling. It has 3 receivers, and 5 transmitters as well as operation at baud rates of up to 1Mb/s [25].

For RS485, the full duplex MAX491 driver chip was selected. Unfortunately, it was not possible to find a suitable multi-transceiver full duplex RS485 chip (say, for instance an RS485 version of the MAX3237) and thus 2 MAX491s were utilised, as there are 2 UARTs available. The MAX491 is capable of speeds up to 2.5Mbps, as it is not slew rate limited. A particular advantage of this transceiver is the lack of external components required - only 2 decade capacitors of 0.1μF and 10μF were used for decoupling, as well as the 120Ω resistor required for signal termination [26].[3]

For CAN, the MAX3051 IC was chosen. It is a low power CAN transceiver that supports running off a single 3.3V supply. Just like the MAX491, it requires no additional components except for power supply bypassing and termination resistance. Once again, 0.1μF and 10μF were used for decoupling, with a 120Ω resistor providing the necessary signal termination [27].

Next, a DAC was required. The 12-bit I^{2}C MCP4725 was selected, based on its wide community support and verified pre-existing GNU/Linux support. It provides a buffered rail-to-rail voltage output, capable of sourcing up to 25mA of current [28]. Again, decade capacitors of 0.1μF and 10μF were used for decoupling and 2 pull-up resistors of 5kΩ for SCL and SDA. This I^{2}C bus is shared with the cape identification EEPROM discussed below, so it was important to ensure the addresses did not clash.

Lastly, a 24LC256 I^{2}C EEPROM was used for cape identification. The Beaglebone Black requires capes to present an identification EEPROM, available at address 0x54. This EEPROM contains information about the current cape, and specifics relating to the pin-muxing settings that need to be configured [29].

[2] See page 3

[3] RS485 is a differential protocol, and as such requires termination and care when routing traces.

Cost summary

Since cost is a primary objective, the total cost of the IC components can be summed up in table 4.1.

Component	Purpose	Price
MAX3237	RS232 Transceiver	R15.44
MAX491	RS485 Transceiver	R35.96
MAX3051	CAN Transceiver	R31.70
24LC256	256Kb EEPROM	R6.20
MCP4725	DAC	R11.62

Table 4.1: Cost of ICs used on breakout cape.

All prices were obtained locally from RS components for single quantities.

4.3.2 PCB Design

With only 5 ICs and all being relatively low speed, a standard double sided board was designed. The bottom layer was kept as free of traces as possible, so as to act as an effective ground plane when flood-filled.

Unfortunately, even with a small number of ICs a large amount of signals needed to be broken out on screw terminals, resulting in a fairly large PCB with unused space in the middle. While the ideal size for cape would have been to match the footprint of the Beaglebone Black itself, this was simply not possible while retaining the amount of signals being exposed.

Since the Beaglebone Black's ADC is only capable of 1.8V input, 2 resistor divider networks were provided on the ADC input channels. These are labeled and halve the input voltage - allowing for 3.3V input, or quarter it, allowing for 5V input. A 1.8V raw channel is broken out alongside these two.

A 3D rendeing of the top of final PCB design can be seen below in figure 4.4. The large 46 pin dual-inline are where the connectors to slot in to the Beaglebone Black are soldered on. The silkscreen has been used both as a component placement guide to aid in assembly, but also as an aid to the user, indicating each signal on each connector. The cutout leaves space for the network connector, while the slightly shorter side allows the LEDs on the Beaglebone Black to be visible.

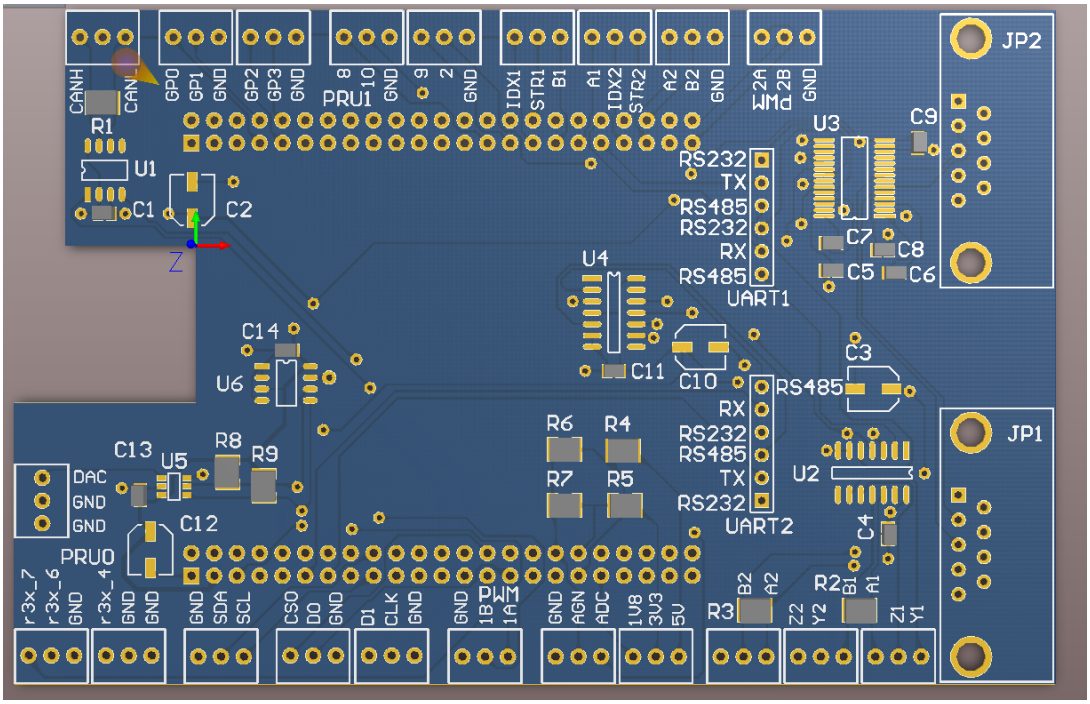

Figure 4.4: Final PCB design.

Furthermore, two headers, labelled UART1 and UART2 are exposed. These allow for the selection between RS232 and RS485 transceivers for each UART, as having both connected simultanious might have undesirable effects. Later on, in section 5.1.2 it will be explained as to how these headers were used to work around design flaws on the PCB.

4.4 Software

For software, the focus shifts to providing a stable and fast base platform for Simulink's *Run on Target Hardware* support. This shall be done by building a custom Linux kernel with the Xenomai patch set, as well as creating a custom Arch Linux based user space environment, thus providing a minimalistic base for models to run on.

4.4.1 Arch Linux

Arch Linux is a cutting edge, rolling release GNU/Linux distribution. Unlike other distributions - such as Ubuntu - which do yearly or so releases, Arch Linux is constantly updated with upstream packages[4] and changes. In addition, they are known to follow upstream closely[5].

This provides the dual benefits of having both the latest features and best performance, while at the same time making things harder to properly set up and install. Additionally, updates become more difficult, as often large changes can happen between consecutive system updates.

Consequently, this may seem like a strange choice for an embedded system. However, the benefits of having the best hardware support and performance more than make up for the difficulties in maintaining the system itself. Once a base system that works has been established, it does not need to be updated, eliminating the problem.

Toroidal

To ease the creation of a custom firmware image for the Beaglebone Black, an open source package called Toroidal was used. Toroidal[6] was created by the author during previous experimentation with GNU/Linux on an Asus Transformer TF300.

It works by generating a file system image from packages specified in a configuration file. This allows for a minimal image to be built, cutting down both on size (as the Beaglebone Black is only equipped with a 2GB eMMC for internal storage) and extraneous running processes[7].

[4] A distribution is essentially a compilation of packages, upstream refers to the original package authors

[5] Distributions like Ubuntu make their own changes to the packages they incorporate

[6] The name is a play on words, as in "Toroidal Transformer"

[7] Once again, Ubuntu is the prime example of this. There is no need for a graphical user interface on a system that doesn't have a screen attached.

4.4.2 Turnkey Operation

Another important design aspect is turnkey operation. By this, it is meant that the end platform must be easy to use - as if it were a professional project. An important design challenge to solve is that of first identifying the Beaglebone Black on the network.

While such a task is easy if one has access to a tool such as `nmap`, requiring a network scan isn't particularly user friendly. Instead, a dynamic DNS hostname is used. This dynamic DNS hostname will be updated each time the Beaglebone Black acquires a new IP address, and as such can be used to access it.

Furthermore, should a case occur where either no DHCP server[8] is available, or the Beaglebone Black is started without a network cable plugged in, it will default to have an IP address of 192.168.100.100.

Lastly, all supporting software is started automatically at boot time. This includes an NTP client[9] and the SSH server, as well as loading all of the drivers and device overlays needed[10].

4.4.3 Simulink Integration

Perhaps most important of all is the integration of the software platform with Simulink. Simulink has the ability to generate real-time C code for a target platform, either through Simulink Coder / Embedded Coder, or via its built in Run on Target Hardware support.

Simulink already has support for running models on the Beagleboard-xM. It is this support that has been used as a basis for integration. By providing the same environment, the custom Arch Linux distribution can be used by Simulink with no modifications.

Run on Target Hardware works by first generating C code, then copying it and compiling on the device. This allows for a much simpler development setup, as it eliminates the headache of cross-cross-compilation[11].

[8] The DHCP server is responsible for handing out IP addresses on a network
[9] The Beaglebone Black does not have an onboard RTC and thus cannot keep time if unplugged.
[10] Loading these manually is quite the pain, but is done in section 6 for verification purposes.
[11] Windows -> Linux -> ARM

However, Run on Target Hardware only provides support for the model, it does not support any of the hardware features available on the Beaglebone Black. To achieve support, custom S-Functions were designed for each peripheral. These S-Functions are designed as minimal interfaces between Simulink and native C code. By splitting the design, the testing and debugging of interface code is possible outside of Simulink.

S-Function Builder

Due to the complexity in writing S-Functions with *Target Language Compiler* (TLC) support from scratch, the S-Function builder - which provides a streamlined environment to generate the TLC code automatically - was used. Unfortunately, the S-Function Builder does not provide access to initialise or shutdown methods; it only allows you to specify code that will be called upon each iteration of the model's execution. To work around this, without modifying the code by hand, the initialise-on-use design pattern was utilised, as is pictured in the UML diagram in figure 4.5.

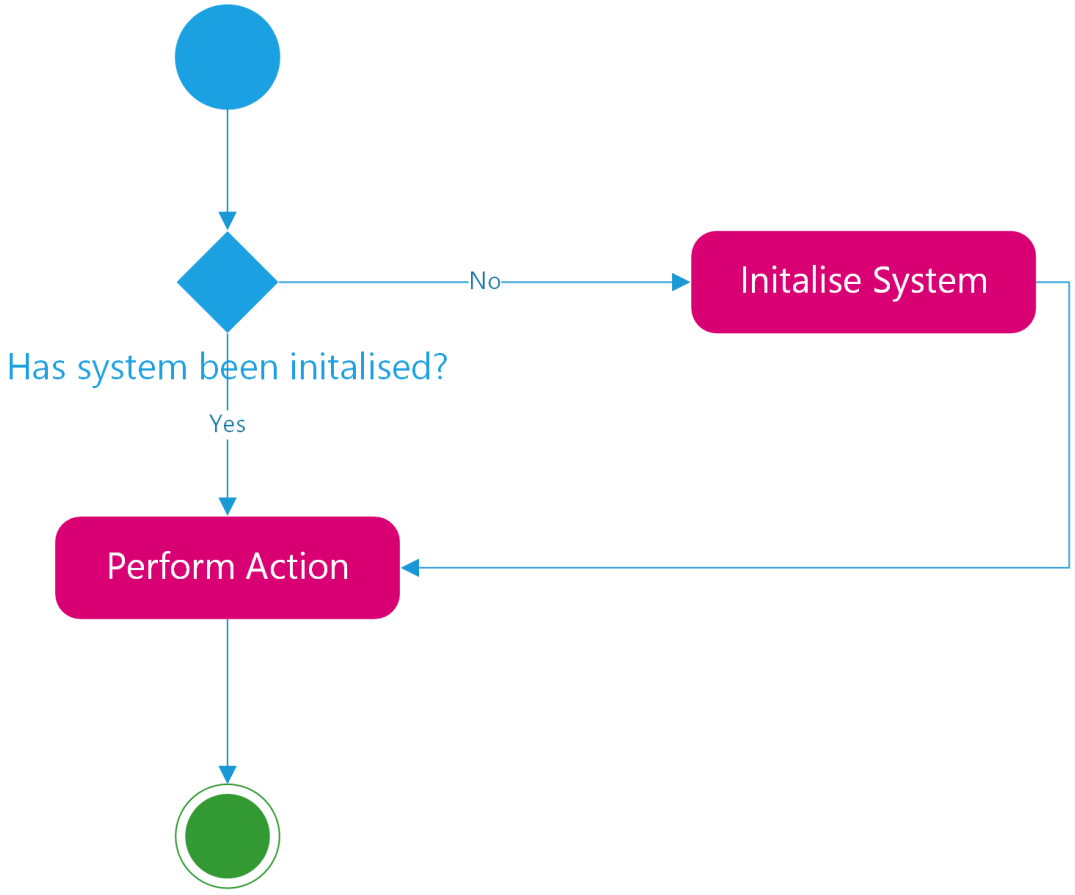

Figure 4.5: UML diagram illustrating the initialise-on-use pattern.

Furthermore, the architecture of the system is arranged such that the minimal amount of code possible is dealt with in Simulink. Instead, the majority of the functionality is contained in header files which the S-Function builder includes. This once again allows for easy testing, by separating the code that shall be executed in the model from the model itself. Such an example of testing can be seen in the code listing below, where the GPIO read functionality is tested.

```
1 #include "gpio.h"
2
3 int main(void) {
4         int gpio_result = gpio_read(0);
5
6         return 0;
7 }
```

The S-Function builder interfaces with the peripherals in the exact same method as the testing code - here it would use `gpio_read()` to read a GPIO pin and assign the output to the block output.

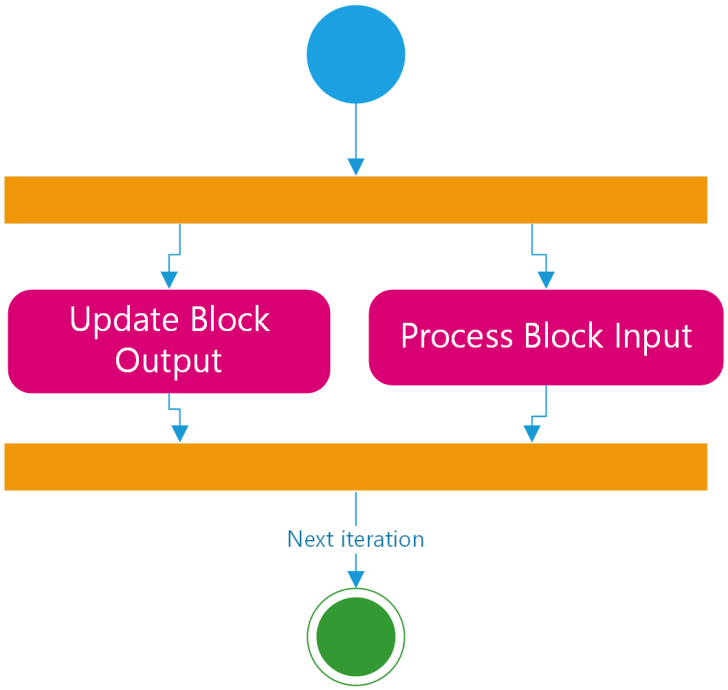

Figure 4.6: UML diagram illustrating the design pattern followed with the S-Function builder.

Chapter 5

Implementation

> *"In theory, there is no difference between theory and practice. But, in practice, there is. "*
>
> – Jan L. A. van de Snepscheut

After establishing a suitable design plan in the previous chapter, it can now be put into effect. Once again, there is a fairly clearly defined line between hardware and software initially. However, this will change later at the verification stage, where the interaction between the hardware and software components of the system is explicitly tested.

5.1 Hardware Implementation

The primary portion of the hardware implementation phase is focused on assembling and testing the PCB that had been designed. As mentioned previously, a PCB cape with various transceivers as well as screw terminals and DB9 connectors was produced.

5.1.1 PCB Assembly

While soldering through-hole components is relatively simple, some of the components involved came in fine pitched SMT packages. These require a more careful assembly procedure. While it is possible to solder these components with a regular soldering iron, the two most common approaches are hot air reflow or oven reflow. In this case, hot air reflow was used[1]. First, the PCB was prepared with solder paste, as can be seen in figure 5.1 below. This was applied using a stencil provided by the PCB manufacturer.

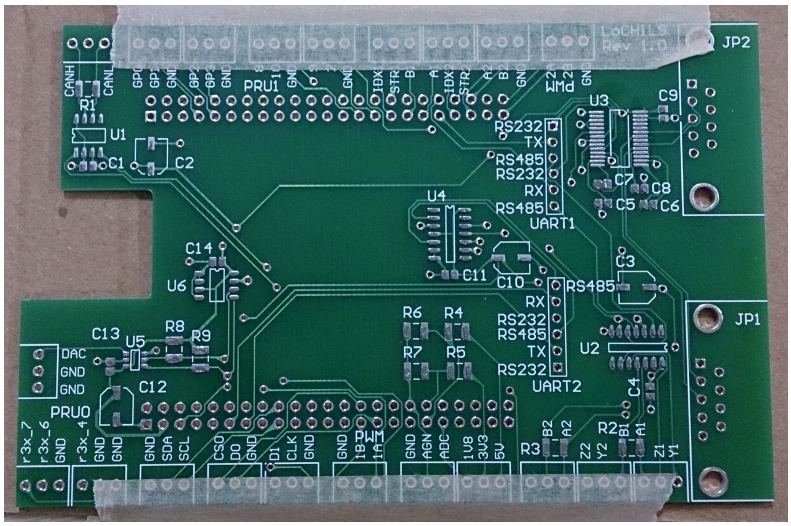

Figure 5.1: PCB with solder paste applied.

Next, the components were carefully placed in position using tweezers. This needed care in order to avoid smudging of the solder paste; and hence shorts; from occurring. This can be seen below in figure 5.2.

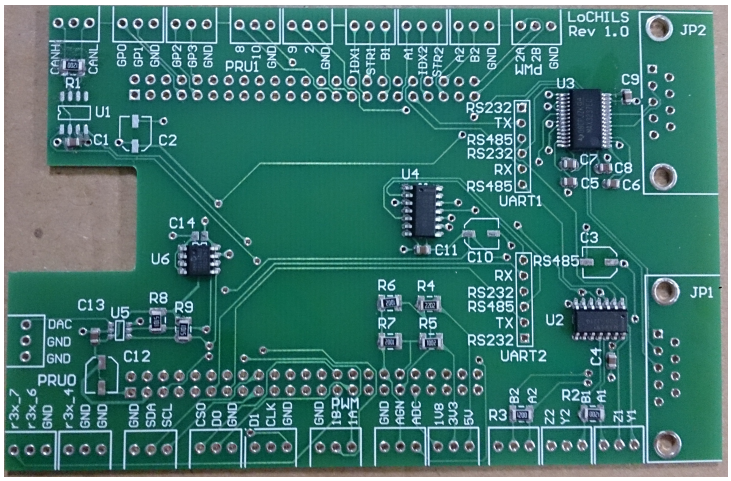

Figure 5.2: PCB with components sitting on top of solder paste.

[1]Oven reflow was originally considered, but unfortunately the lab containing the oven was closed when the PCB was assembled

At this point, the PCB is ready for hot air reflow. A hot air rework station in the tomography lab was used, on the lowest airflow setting. This was held over the components, until the solder started to reflow. Due to the PCB's soldermask as well as surface tension, any ICs which were positioned slightly incorrectly corrected themselves. No shorts were observed, except for the RS232 transceiver (U3). Due to the very fine pitch of this device, there were numerous solder bridges over the pins, as can be observed in figure 5.3 below.

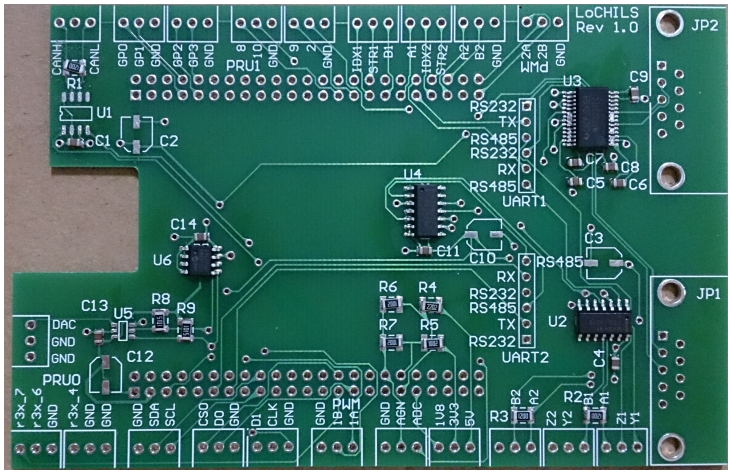

Figure 5.3: PCB with soldered components.

Next, the SMT electrolytic capacitors were soldered. Due to their size, as well as sensitivity to heat, these were soldered normally using an iron.

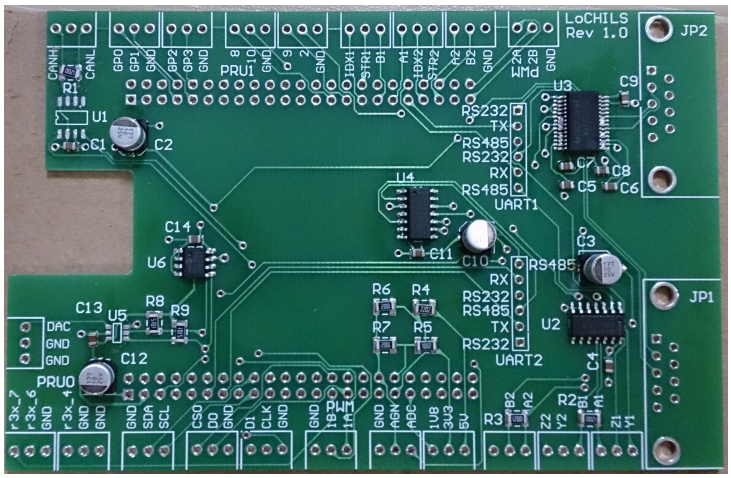

Figure 5.4: PCB with soldered capacitors.

As can be seen in figure 5.4 above, the SMT capacitors are now in place. However, 2 components are still missing - the CAN controller[2] (U1) and the DAC (U5). Due to supply issues, these components were not available when the board was being assembled, and were added at a later stage.

[2]As an amusing anecdote, free samples of the CAN IC were requested from Maxim at the same time as the order was placed with RS-components. The free samples arrived first.

Next, the solder bridges on U3 were cleaned using solderwick and flux. This process draws out the excess solder[3]. Finally, the through-hole components were soldered in place, leaving an almost-finished board, as can be seen in figure 5.6

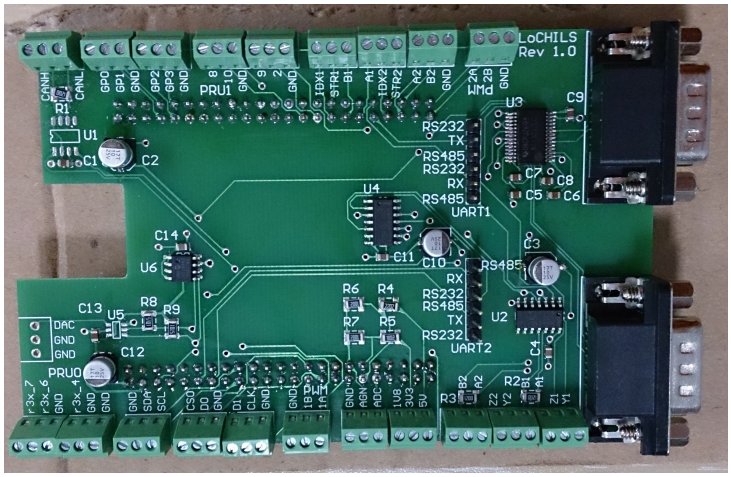

Figure 5.5: PCB with soldered through-hole components. The solder bridges on U3 have been cleaned up.

The PCB was finished a few days later, when the CAN controller and DAC arrived. These were soldered on using hot air, resulting in the finished PCB.

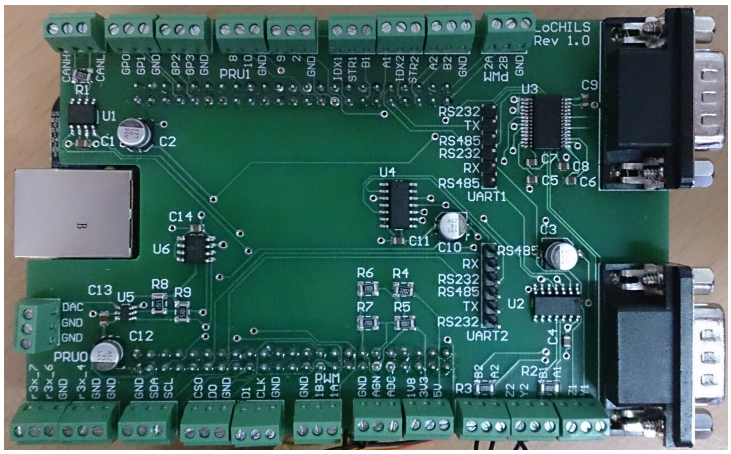

Figure 5.6: Final assembled PCB. Here, it is mounted on the Beaglebone Black.

[3]Unfortunately, it drew out too much solder and caused an issue with the transceiver, which will be discussed in the next section

5.1.2 Problems encountered

RS232 Transceiver

As mentioned above, solder bridges had formed on the RS232 transceiver (U3) due to the fine pitch of the device. These were cleaned with solderwick, but upon initial testing the RS232 transceiver appeared to not work properly. The RX function worked fine, but transmission didn't work at all. Inspecting the TX line with a oscilloscope revealed that it was simply sitting at 0V, and not changing - even when an input signal was verified as being presented on the input to the transceiver.

At first, the problem was believed to lie with either the $\overline{\text{SHDN}}$ or $\overline{\text{EN}}$ pins. Confusingly, due to these pins being inverted, a 0 logic level is required at $\overline{\text{EN}}$ to have the chip enabled, and a 1 logic level is required at $\overline{\text{SHDN}}$ to have the chip not be in shutdown mode. These voltages were verified as being as the correct values on chip, so this was ruled out as a cause.

The problem was finally discovered when probing the V+ and V- charge pump outputs of the chip. Nominally, these voltages should be at +5.5V and -5.5V respectively. However, they were only sitting at half their nominal values - indicating an issue with the charge pump capacitors. Further probing eventually revealed the cause - the pad for a charge pump capacitor was not soldered down properly and thus not making contact. However, when it was probed the extra force pushed the pin down onto the pad and resulted in a connection and the chip working. This was manually fixed with a soldering iron, and the RS232 transceiver had no further issues.

RS485 Transceivers

The RS485 transceivers chosen were unfortunately 5V parts. This went unnoticed during the redesign (as mentioned in section 4.1), and was only picked up after the board had already been ordered. Originally, the transceivers were supposed to interface with an STM32F4 - which is 5V tolerant.

However, the Beaglebone Black is not 5V tolerant on any pin. In fact, 5V runs a severe risk of permantly damaging the board. Fortunately, since the RS485 transceivers operate off TTL logic levels, the 3.3V of the Beaglebone Black is enough to trigger a high logic level on the transmit pin. The problem lies on the receive pin, where the 5V outputted by the transceiver would damage the Beaglebone Black.

Luckily, since the pins were broken out on the UART headers, a small custom board could be created to perform logic level shifting. This is described further on in section 5.1.3.

DB9 Pinout

Unfortunately, while the footprint used for the DB9 connector was correct, the pins turned out to be inverted compared to the actual connector used. As a result, pin 5 - nominally GND - was connected through to pin 1 on the DB9 connector, whereas pin 1 - NC - was connected to pin 5 of the connector.

Fortunately, this too was fairly easy to resolve. A set of custom adapters was created, as detailed in section 5.1.3

UART header swapped

Continuing with the trend of serial related problems, a further issues is that the RS232 pins brought out to the UART headers were swapped around. Connecting the UART to RS232 on UART1 actually connects it to UART2, while the same is true in reverse for UART2's header. This problem only affects RS232 - the RS485 transceiver wiring is correct.

This was fixed in the same custom board as used for solving the RS485 level shifting problem, which is described in section 5.1.3.

5.1.3 Modifications performed

Serial board

In order to fix both the problem relating to the 5V RS485 transceivers, as well as the RS232 transceiver lines going to the opposite labelled UART, a small board was designed to clip on to the existing UART header and provide the necessary modifications.

While it would have been ideal to use a proper PCB for this task, time was not permitting - and thus a quick solution was devised and implemented on stripboard.

This board swaps the RS232 lines around, ensuring that the jumpers for UART1 control the first DB9 connector, and similarly for UART2. In addition, a voltage divider provides level shifting of the RS485 RX signals.

It is important to note that low value resistors were used in making up the voltage divider. While these draw a relatively significant current, they ensure sharp edges remain - and testing as performed in section 6.1.2 at baud rate of 115200 did not reveal any problems. The final board, mounted on the cape's UART headers, is pictured below in figure 5.7.

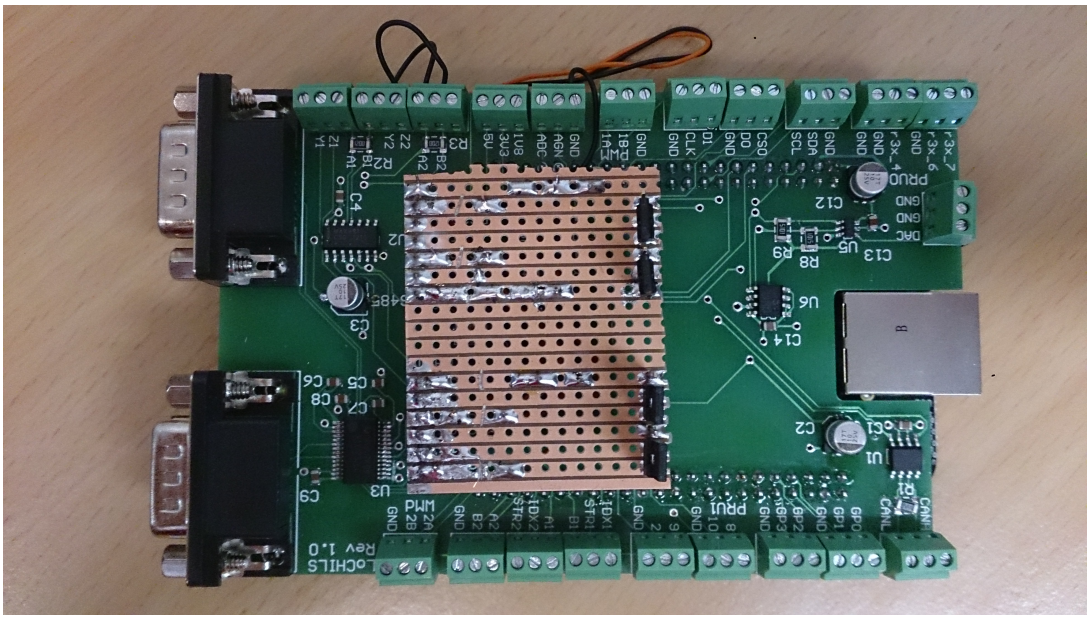

Figure 5.7: Serial fix board mounted on the cape's UART headers, with jumpers in place.

Serial connectors

In order to fix the serial connectors, two simple adapters were made which correctly maps the pins. This is done by using a male to male DB9 connector, with modified cabling to achieve the correct pinout.

5.2 Software Implementation

The software implementation stage focuses on compiling the Linux/Xenomai kernel, as well as building the GNU/Linux user land that will be used. Additionally, the steps required to create a MicroSD card and transfer the image to the internal memory of the Beaglebone Black are detailed.

5.2.1 Linux/Xenomai Kernel

In order to produce a Xenomai based kernel, the relevant Linux kernel tree needs to be prepared, and then patched with real-time support. The first step is ensuring that a suitable cross-compiler is available on the system that will be used to compile the kernel - which can be pretty much any recent GNU/Linux based system. Thanks to the ArchLinuxArm team, there is a prebuilt compiler package suitable for use. This can be extracted:

```
1 f@d:~> wget http://archlinuxarm.org/builder/xtools/x-tools7h.tar.xz
2 f@d:~> tar -xvf x-tools7h.tar.xz
```

Now that the toolchain has been extracted, the patching and compilation of the kernel can take place. The first step involved is to create a working directory where all files will be kept.

```
1 f@d:~> mkdir BBBXenomai
2 f@d:~> cd BBBXenomai
```

The ipipe and Xenomai sources can now be retrieved. The ipipe sources contain the full standard Linux kernel tree, which will later be patched with the Beaglebone Black specific hardware support.

```
f@d:BBBXenomai> git clone git://git.xenomai.org/ipipe-jki.git
f@d:BBBXenomai> cd ipipe-jki; git checkout -b 3.8 origin/for-upstream/3.8; cd ..
f@d:BBBXenomai> git clone git://git.xenomai.org/xenomai-2.6.git
```

Next, two helper packages will be downloaded. The `beaglebone-xenomai` package contains helper scripts to ensure compilation of Xenomai for the Beaglebone Black, whereas the `meta-begaleboard` package contains specific patches and fixes for the Beaglebone Black. Due to ongoing development changes, a specific version[4] is checked out. Lastly a small patch is applied to ensure clean compilation.

```
f@d:BBBXenomai> git clone https://github.com/DrunkenInfant/beaglebone-xenomai
f@d:BBBXenomai> git clone https://github.com/beagleboard/meta-beagleboard.git
f@d:BBBXenomai> cd meta-beagleboard;
f@d:meta-beagleboard> git checkout 50316366dd4f75027ee5291b65a9bbcfa9a9e840
f@d:meta-beagleboard> patch -p1 < ../beaglebone-xenomai/\
        meta-beaglebone-xenomai.patch
f@d:meta-beagleboard> cd ..
```

Once that has been completed, the `beaglebone-xenomai` package needs to be modified to point to the correct `meta-beagleboard` directory. To do this, open up `beaglebone-xenomai/apply-beaglebone-patches.sh` in an editor, and change the `META_BEAGLEBONE_ROOT` line to be `META_BEAGLEBONE_ROOT=~/BBBXenomai/meta-beagleboard`. After performing this modification, the Xenomai and Beaglebone patches can now be applied to the kernel source tree

```
f@d:BBBXenomai> cd ipipe-jki
f@d:ipipe-jki> source ../beaglebone-xenomai/apply-beaglebone-patches.sh
f@d:ipipe-jki> ../xenomai-2.6/scripts/prepare-kernel.sh --arch=arm
```

[4] A commit, in GIT terms

Once the patches have been successfully applied, the kernel is almost ready for compilation. First, the default kernel configuration file is loaded, and then LoCHILS specific patches are applied. These include quadrature support, increased I^2C speed as well as a few bug fixes. The patch file can be located on the CD attached with this document.

```
f@d:ipipe-jki> cp ../meta-beagleboard/common-bsp/recipes-kernel/\
        linux/linux-mainline-3.8/beaglebone/defconfig .config
f@d:ipipe-jki> patch -p1 < ../custom-lochils.patch
```

Finally, the kernel and associated modules are ready for compilation. Depending on the speed of the machine being used, this process may take a while. The `-j5` argument should be tweaked to be `-jX` where X is the number of CPU cores available plus one. Here, `-j5` is suitable for a quad core machine.

```
f@d:ipipe-jki> make -j5 ARCH=arm \
        CROSS_COMPILE=~/x-tools7h/bin/arm-linux-gnuabihf- uImage modules
```

Once finished, a cross compiled Linux uImage[5] will be present under arch/arm/boot.

[5] uImage is the preferred kernel image format of u-boot, the bootloader used.

5.2.2 GNU/Linux Userspace

Having compiled the kernel in the previous section, the GNU/Linux ArchLinuxArm userspace can now be created. The first step involved is to retrieve the toroidal toolkit. This is not the original toolkit, instead it has been forked[6] from the original to accommodate Beaglebone Black specific changes.

```
1 f@d:BBBXenomai> git clone git@bitbucket.org:cb22/bbxm-toroidal.git
2 f@d:BBBXenomai> cd bbxm-toroidal
```

(Here, the name "bbxm-toroidal" is an artifact from the old design, as detailed in section 4.1. Due to the similarities between the Beagleboard-xM and Beaglebone Black, the existing work done on preparing the user space environment was kept)

Toroidal uses a GNU/Linux trick known as userspace binary emulation to achieve the easy cross-creation of userspace images. While common emulators emulate a full CPU and system, Toroidal uses `binfmt-misc` combined with `qemu-arm-user` to create an environment whereby ARM binaries can be run natively on an X86 or X86_64 system.

`binfmt-misc` allows the user to specify a custom exection handler for binaries. Normally, these binaries are executed automatically by the system, which would clearly fail for ARM binaries on an X86 based system. However, with `binfmt-misc` it is possible to specify an application that invokes the binaries. By setting this to `qemu-arm-user` - an ARM Linux userspace emulator - GNU/Linux binaries for different platforms can be seamlessly run[7]. This is coupled with the `chroot` command to build up a fully self contained filesystem image from scratch.

The following steps need to be performed as the root[8] user, as they create a virtual environment for the operating system image to be created. Then `toroidal.sh` script can be run. This automatically fetches and installs the latest required packages, and builds up a file system image suitable for deployment.

[6] Forking in git refers to the process of taking an existing repository, and starting a new one using that previous repository as a starting point

[7] There is, of course, a performance penalty.

[8] In GNU/Linux terminology, root is the superuser - equivilent to Administrator on Windows systems. However, root can also refer to the base of the GNU/Linux file system, as in the root partition. This is similar to the C: drive in Windows.

```
f@d:bbxm-toroidal> su
root@d:bbxm-toroidal# ./toroidal.sh
```

The previous step normally takes around 20 minutes to run, as it is building the image from binary packages - as opposed to the kernel compilation in the previous section which compiled from source code.

Once completed, a completed user space built to specification is available under the `build/rootfs` folder.

5.2.3 Deploying to SD card

Now that the kernel has been compiled and the userspace environment built, the two can be placed on a MicroSD card suitable for deployment to the Beaglebone Black. This process requires a minimum of a 2GB MicroSD card, preferably of a high class[9].

First, the MicroSD card needs to be partitioned. The Beaglebone Black requires a small FAT16 filesystem as a boot partition. It loads the first stage bootloader which in turn loads the primary bootloader, uboot, from this partition. uboot is then capable of loading the kernel from the much bigger EXT4[10] filesystem on the MicroSD card.

Again, since these steps deal with manipulating filesystems, they need to be performed as the root user. Note that `/dev/mmcblk0` is the path to the device representing the MicroSD card. It is extremely important to get this correct (and it can vary depending on system), as there is the potential for data loss.

[9] The class of a MicroSD card specifies the speed. Class 10 or above is ideal.
[10] EXT4 is a common filesystem for GNU/Linux.

```
f@d:bbxm-toroidal> su
root@d:bbxm-toroidal# fdisk /dev/mmcblk0
Welcome to fdisk (util-linux 2.23.2).
Command (m for help): n
Partition type:
   p   primary (0 primary, 0 extended, 4 free)
   e   extended
Select (default p): p
Partition number (1-4, default 1): 1
First sector (2048-4095999, default 2048):
Using default value 2048
Last sector, +sectors or +size{K,M,G} (2048-4095999, default 4095999): +32M
Partition 1 of type Linux and of size 32 MiB is set

Command (m for help): t
Selected partition 1
Hex code (type L to list all codes): 6
Changed type of partition 'Linux' to 'FAT16'

Command (m for help): a
Selected partition 1

Command (m for help): n
Partition type:
   p   primary (1 primary, 0 extended, 3 free)
   e   extended
Select (default p):
Using default response p
Partition number (2-4, default 2):
First sector (67584-4095999, default 67584):
Using default value 67584
Last sector, +sectors or +size{K,M,G} (67584-4095999, default 4095999):
Using default value 4095999
Partition 2 of type Linux and of size 1.9 GiB is set

Command (m for help): w
The partition table has been altered!
```

After the MicroSD card has been succesfully partitioned, the partitions need to be formatted with the correct filesystems. The commands below create the FAT16 filesystem on the first boot partition, and the primary GNU/Linux EXT4 filesystem on the second partition.

```
root@d:bbxm-toroidal# mkfs.vfat /dev/mmcblk0p1
root@d:bbxm-toroidal# mkfs.ext4 /dev/mmcblk0p2
```

Next, the partitions can be mounted[11]. Two directories are created, `/mnt/boot` and `/mnt/main` for the filesystems to be mounted in.

```
root@d:bbxm-toroidal# mkdir /mnt/boot
root@d:bbxm-toroidal# mkdir /mnt/main
root@d:bbxm-toroidal# mount /dev/mmcblk0p1 /mnt/boot
root@d:bbxm-toroidal# mount /dev/mmcblk0p2 /mnt/main
```

The bootloader files can now be placed on to the boot partition. This is done by first downloading the bootloader archieve from the ArchLinuxArm website, and then extracting it to the boot partition.

```
root@d:bbxm-toroidal# wget http://archlinuxarm.org/os/omap/BeagleBone-\
        bootloader.tar.gz
root@d:bbxm-toroidal# tar -xvf BeagleBone-bootloader.tar.gz -C /mnt/boot
root@d:bbxm-toroidal# umount /mnt/boot
```

Once completed, the bootloader partition is unmounted. This is the equivilent of *Safely remove hardware* on Windows, and causes the disk cache to be flushed - ensuring that no data is lost. Now, the main part of the system can be copied across to the MicroSD card. It is important to do this step in a way that preserves all the permissions of the filesystem. As such, the `tar` command is used.

[11] Mounting a partition in GNU/Linux is the act of making the contents of the filesystem available under a specific directory.

```
root@d:bbxm-toroidal# tar -cpf - build/rootfs/* | tar -xvpf - -C /mnt/main
```

Lastly, the compiled kernel is copied into its final directory and the main partition is unmounted.

```
root@d:bbxm-toroidal# cp ../ipipe-jki/arch/arm/boot/uImage /mnt/main/boot/
root@d:bbxm-toroidal# umount /mnt/main
```

The MicroSD card is now ready to be used, and can be placed into the Beaglebone Black to boot the operating system. Once the operating system has started up from the MicroSD card, it is possible to move it to the internal eMMC storage of the Beaglebone Black - freeing up the MicroSD card slot.

This is done by accessing the Beaglebone Black via a serial terminal, and then executing the following commands in order to clear the existing contents of the internal memory, before replacing it with a copy of what is on the MicroSD card.

```
root@beagle# mkfs.ext4 /dev/mmcblk1p2
root@beagle# mount /dev/mmcblk1p2 /mnt
root@beagle# tar -cpf - --exclude /mnt /* | tar -xvpf - -C /mnt
root@beagle# umount /mnt
```

After completion of this step, the Beaglebone Black can be rebooted and the MicroSD card removed. The custom userspace and kernel will now boot off the internal memory of the board.

5.2.4 PRU PWM Capture

While the Beaglebone Black is capable of native PWM capture, using its eCAP module, there is unfortunately no driver support for this under Linux. Instead of implementing a driver, a decision was taken to rather implement a custom PWM input capture module.

Normally, this would be impossible due to the sheer latencies involved in such a system. Even if it were implemented, the maximum supported frequency would be low, and nowhere near what a dedicated hardware module would provide. It is here where the Beaglebone Black's PRUs come into play.

As mentioned before in the literature review (section 2.3.4), the Beaglebone Black has 2 hard real-time PRU cores, each running at 200MHz with direct access to GPIO pins. Thus, a custom assembly implementation a PWM input capture module was written and utilised on a PRU core. The crux of this implementation is shown below:

```
1  COUNT_ON: // Count the time the signal is high
2      QBBC COUNT_ON_DONE, r31.t8      // If it is low, we are done.
3      ADD r10, r10, 1                 // Increment r10, which counts on time
4      QBBS COUNT_ON, r31.t8           // Branch back if the pin is still high
5
6  COUNT_ON_DONE: // Once the signal goes low, store the couter to RAM.
7      SBCO r10, CONST_PRUDRAM, 0, 4   // Store r10 in DRAM 0 bytes offset
8      LDI r10, 0                      // Reset r10 back to 0
9      QBA COUNT_OFF                   // Start counting the off time
10
11 COUNT_OFF: // Count the time the signal is low
12     QBBS COUNT_OFF_DONE, r31.t8
13     ADD r11, r11, 1
14     QBBC COUNT_OFF, r31.t8
15
16 COUNT_OFF_DONE: // Once the signal goes high, store the counter to RAM.
17     SBCO r11, CONST_PRUDRAM, 4, 4
18     LDI r11, 0
19     QBA COUNT_ON
```

The flow of operations can also be represented in UML diagram form, as is shown in figure 5.8.

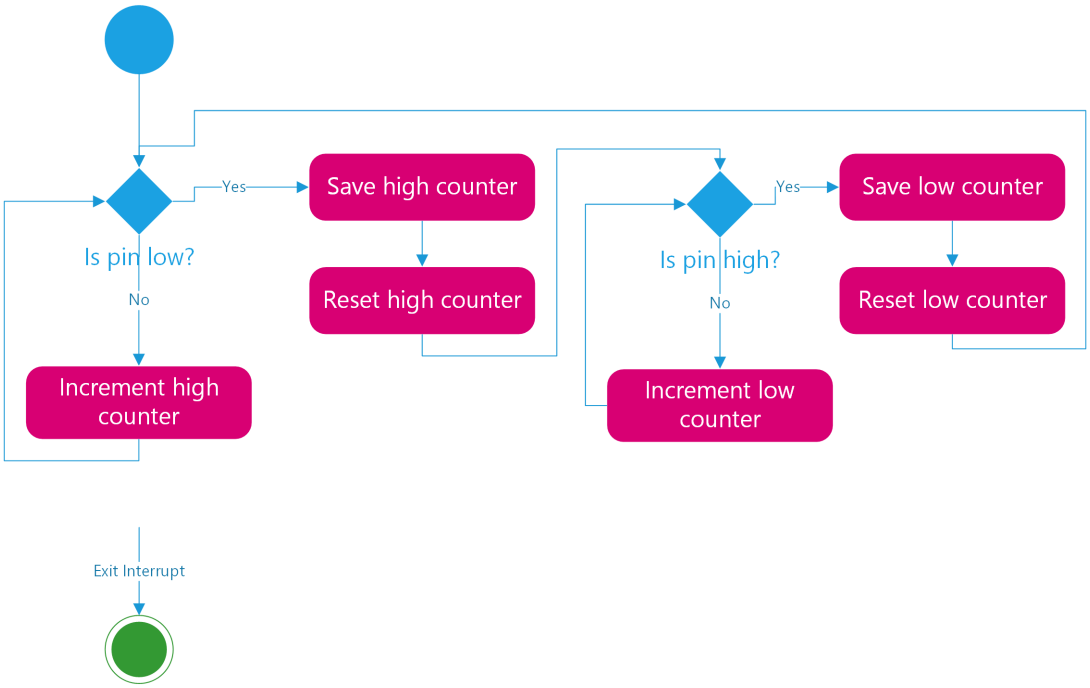

Figure 5.8: UML diagram showing the flow of the PRU PWM assembly code.

It works by counting the number of loop iterations that the specific pin is in either a high or low state for. Once the pin transitions state, the counter is reset and copied to shared memory, where it can be accessed directly from GNU/Linux user space using the PRU libraries provided by TI.

Chapter 6

Verification

"The problem with troubleshooting is that trouble shoots back."

– Author Unknown

Before proceeding with testing the Simulink blockset integration, it is important to know that all the peripherals are fully functioning. This step greatly assists with debugging, as it helps clarify exactly where a problem lies. If, for example, the UART block was not functioning in Simulink, the problem might lie at any level of the stack. It could be the RS232 transceiver, the UART pin-muxing, the Linux driver or the Simulink integration itself.

Thus, in order to verify the peripherals involved, a standardized testing procedure was followed. A high-level overview of this procedure is given in figure 6.1 below.

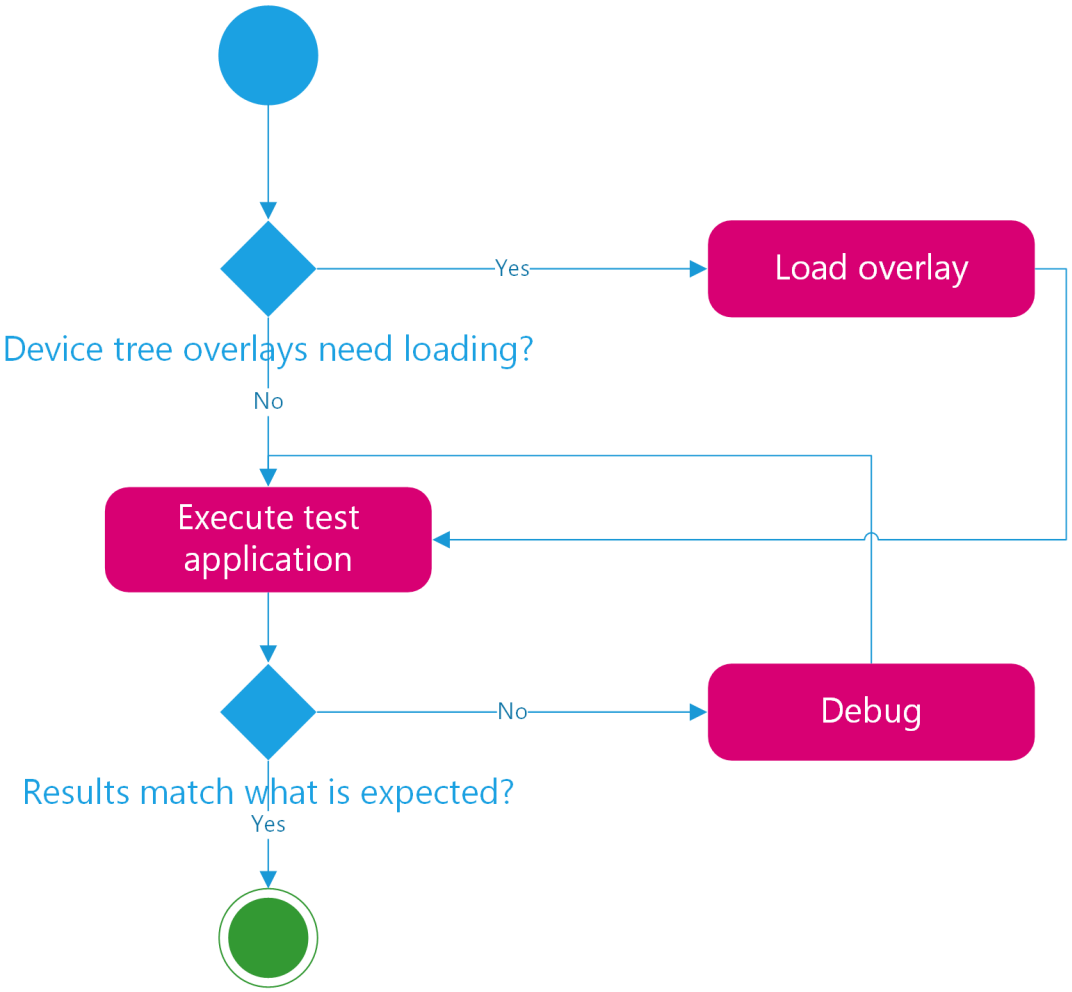

Figure 6.1: Standard verification proceedure followed for peripherals.

6.1 UART

The cape exposes 2 UARTs from the Beaglebone Black, UART4 and UART5. To verify these, first the relevant device tree overlay files were loaded:

```
1 root@beagle:~# echo tty04_armhf.com > /sys/devices/bone_capemgr.9/slots
2 root@beagle:~# echo tty05_armhf.com > /sys/devices/bone_capemgr.9/slots
```

After performing these commands, Linux exposes UART4 and UART5 as `/dev/tty04` and `/dev/tty05` respectively. This allows for the use of standard serial terminal software to verify the UART.

Using an FTDI USB TTL serial cable, the TTL UART was connected directly to a computer for testing. A serial terminal[1] was then invoked on the Beaglebone Black at a baud rate of 115200:

```
root@beagle:~# picocom -b 115200 /dev/ttyO4
```

and similarily on the PC used for testing. Characters were then typed on both terminals and verified to be received on the other end, indicating that the serial link was fully functional. This process was repeated for the second UART.

6.1.1 RS232

Once the problem regarding the RS232 transceiver described in section 5.1.2 had been fixed, it was verified. This was performed in a similar manner to the generic TTL UART described above; however, the hardware test setup was different.

Now, the UART configuration jumpers were wired to link the TX and RX lines of both UARTs to the RS232 transceiver. Next, a USB serial cable was connected to the DB9 connectors by using a null-modem cable[2]. Then, as above, picocom was used to ensure data could be sent and received successfully.

6.1.2 RS485

RS485 was a bit trickier to verify, as there was not a USB RS485 adapter available on hand. Instead, once the voltage problem described in section 5.1.2 had been fixed, RS485 was verified by loopback.

In this case, Y1 and Z1 were connected to A2 and B2; while Y2 and Z2 were connected to A1 and B1. Then, two instances of picocom were launched on the Beaglebone Black, and similar character tests were performed. All data was sent and received successfully.

[1] Picocom is a light weight serial terminal application available for GNU/Linux
[2] The DB-9 connectors were configured as DTE

It is important to note that while this passed the basic RS485 test, indicating that the transceivers were indeed working, it was an extremely simple setup in terms of RS485. No advanced features, such as multimaster or noise rejection were tested, due to a lack of test equipment. However, it can be inferred that such features - while unverified - should be working.

6.2 GPIO

There are 4 pins broken out on the cape that are dedicated for GPIO. While technically other pins could be used as GPIOs by reconfiguring the device tree, this was not done. The 4 GPIOs first need to be exported to userspace:

```
root@beagle:~# echo 66 > /sys/class/gpio/export
root@beagle:~# echo 67 > /sys/class/gpio/export
root@beagle:~# echo 68 > /sys/class/gpio/export
root@beagle:~# echo 69 > /sys/class/gpio/export
```

This will create the relevant directories /sys/class/gpio/gpioXX, where XX corresponds to each GPIO pin exported. In these directories there are 2 files of interest. `direction` controls whether the GPIO acts as an input or output, and `value` either reads in the input or sets the output level.

In order to test the GPIO pins, they were first assigned as outputs and set to a high logic level. This was observed on an oscilloscope, before they were set back to a low logic level.

```
root@beagle:~# echo out > /sys/class/gpio/gpio66/direction
root@beagle:~# echo 1 > /sys/class/gpio/gpio66/value
...
root@beagle:~# echo 0 > /sys/class/gpio/gpio66/value
```

This process was repeated for the 4 GPIOs explicitly broken out on the cape. Next, they were tested in input mode. A low logic level was applied to each pin, before the value was read out. Then, a high level was applied and the value read in again.

```
root@beagle:~# echo in > /sys/class/gpio/gpio66/direction
root@beagle:~# cat /sys/class/gpio/gpio66/value
0
root@beagle:~# cat /sys/class/gpio/gpio66/value
1
```

As can be seen from the above snippet, the GPIO successfully reports 0 when presented with a low input, and 1 when presented with a high input.

6.3 PWM Output

4 PWM outputs are exposed on the cape, and split into two groups. Each group can have an independent period [3], while each channel can only have a differing duty cycle. In order to perform verification, the relevant device tree overlay files were loaded:

```
root@beagle:~# echo am33xx_pwm > /sys/devices/bone_capemgr.9/slots
root@beagle:~# echo bone_pwm_P8_45 > /sys/devices/bone_capemgr.9/slots
root@beagle:~# echo bone_pwm_P8_46 > /sys/devices/bone_capemgr.9/slots
root@beagle:~# echo bone_pwm_P9_14 > /sys/devices/bone_capemgr.9/slots
root@beagle:~# echo bone_pwm_P9_16 > /sys/devices/bone_capemgr.9/slots
```

The first command initialises the Beaglebone Black's PWM subsystem, while the subsequent commands expose each PWM interface. The PWM interface can be accessed in the folder `/sys/devices/ocp.2/pwm_test_PX_XX`, where X_XX refers to the header and pin number of the specific PWM pin.

[3] While each group can have an independent period, a bug in the Linux kernel prevented this from working - this was fixed.

There are numerous files in this folder, but the two relevant ones are `period` and `duty` which set, in nanoseconds, the period and duty cycle of the PWM signal. In order to verify the PWM output, each PWM channel was connected to an oscilloscope and measured directly. Figure 6.2 below illustrates the testing of PWM interfaces 1A and 1B, at a rate of 100kHz.

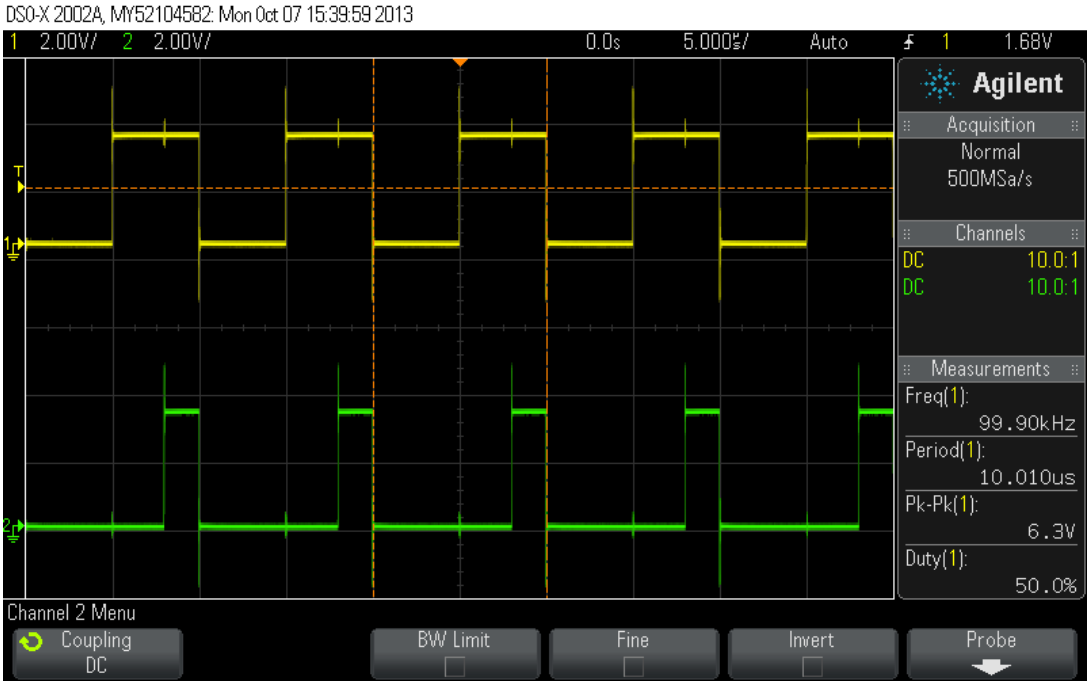

Figure 6.2: PWM output signal. Channel 1 (yellow) has a duty cycle of 50%, whereas channel 2 (green) is 25%. Frequency is 100kHz for both.

The ringing seen in the above figure is a result of the test setup used; specifically the inductance of the ground clip and length of wire used to connect the screw terminal to the oscilloscope's probe.

6.4 PWM Capture

As mentioned in section 5.2.4, PWM capture has been implemented by using the PRU subsystem of the Beaglebone Black. Thus, in order to use it, the PRU subsystem must first be initialised. This is detailed below:

```
root@beagle:~# modprobe uio_pruss; sleep 2
root@beagle:~# rmmod uio_pruss; sleep 2
root@beagle:~# echo cape-bone-pru > /sys/devices/bone_capemgr.9/slots
root@beagle:~# sleep 2
root@beagle:~# modprobe uio_pruss; sleep 2
root@beagle:~# echo cape-bone-pru > /sys/devices/bone_capemgr.9/slots
root@beagle:~# sleep 2
root@beagle:~# rmmod uio_pruss; sleep 2
root@beagle:~# modprobe uio_pruss
```

Ideally, it should be possible to simply load the customised `cape-bone-pru` device tree overlay. Unfortunately, due to bugs in the PRU subsystem, the above commands are needed to ensure reliable loading of the driver.

Once the PRU system has been initialised, the application can be loaded onto the processor itself. This is done by using a special helper application, which loads the compiled binary to the PRU core, and allows for interaction - either by interrupts or memory mapping.

The waveform shown below in figure 6.3 is fed into the PRU input pin, followed by starting the PRU application. This was a custom test application that prints out the on and off time (in loop iterations) for the PWM signal. By mapping these values with the known frequency of the signal, it is possible to establish a mapping between loop iterations and wall clock time. This would also be possible, due to the nature of the PRU cores, by instruction counting the assembly code.

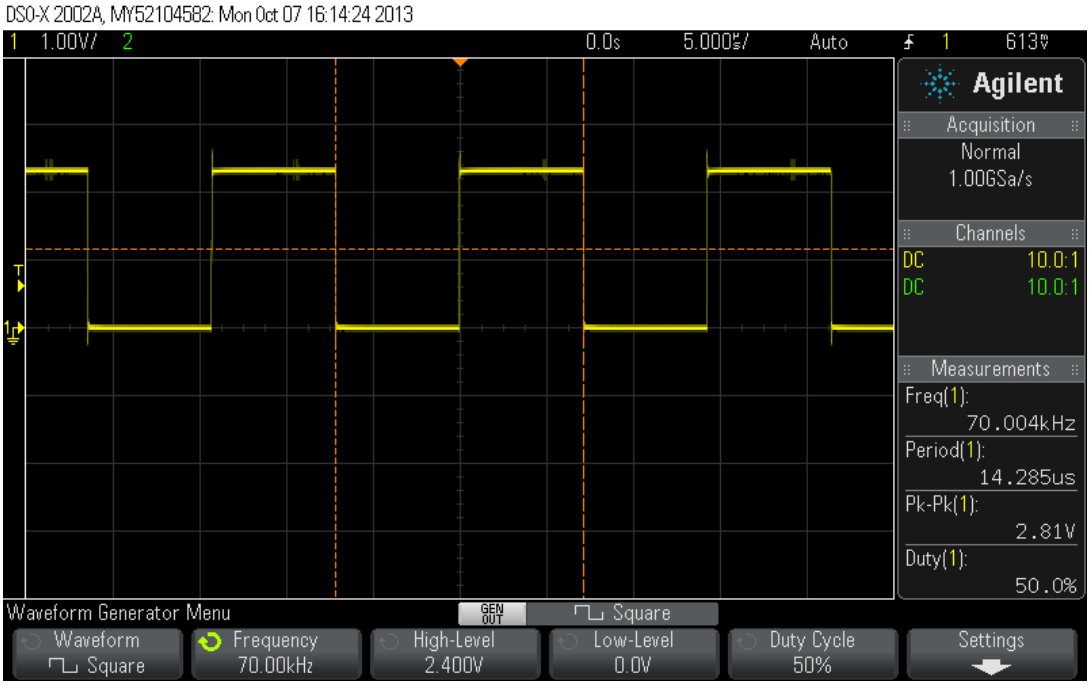

Figure 6.3: Generated PWM signal at 70kHz with 50% duty cycle being sent to the Beaglebone Black.

```
root@beagle:~# ./pru_pwm_capture
...
Value of pruDataMem_int[0] (on time) is 475
Value of pruDataMem_int[1] (off time) is 475
...
```

Clearly with a 50% duty cycle, the on and off times should be the same. This test was repeated with varying duty cycles and frequencies, and all performed as expected. For reference, the wall clock time of a loop iteration of the demo application can be worked out using:

$$t_{\text{iter}} = \frac{t_{\text{on}}}{\#\text{ on iterations}} \tag{6.1}$$

$$t_{\text{iter}} = 15\text{nS} \tag{6.2}$$

With the PRU core running at 200MHz, this corresponds to three cycles per iteration - which matches up with counting the assembly instructions involved.

6.5 ADC

The Beaglebone Black has a 100ksps multiplexed ADC, with 8 input channels. While it was originally designed to be used with touch screen controllers, it is still a fully functioning ADC, and can be used for other purposes.

To verify the ADC hardware, the relevant device tree overlay needs to be loaded:

```
root@beagle:~# echo BB-ADC > /sys/devices/bone_capemgr.9/slots
```

This creates the directory `/sys/devices/ocp.2/44e0d000.tscadc/tiadc/iio:device0`. The files `in_voltageX_raw`, where X corresponds to the ADC channel can then be read, resulting in a number in the range of 0 to 4095. This corresponding to the full scale of the ADC; 0V to 1.8V.

This was tested by applying a known voltage at the ADC input channel, and reading back the voltage by issuing the command:

```
root@beagle:~# cat /sys/bus/iio/devices/iio:device0/in_voltage2_raw
2048
```

Next, the voltage divided channels were tested. These are ADC channels exposed on the cape that have a pre-installed voltage divider; to ease use with higher voltages. Shown in figure 6.4 below, is a 5V peak to peak signal being attenuated to 1.61V for the ADC. In addition, this was done with a script to continuously read out the value of the ADC:

```
root@beagle:~# read_adc () {
>         cat /sys/bus/iio/devices/iio:device0/in_voltage5_raw
>         read_adc
> }
root@beagle:~# read_adc
2048
...
^C
```

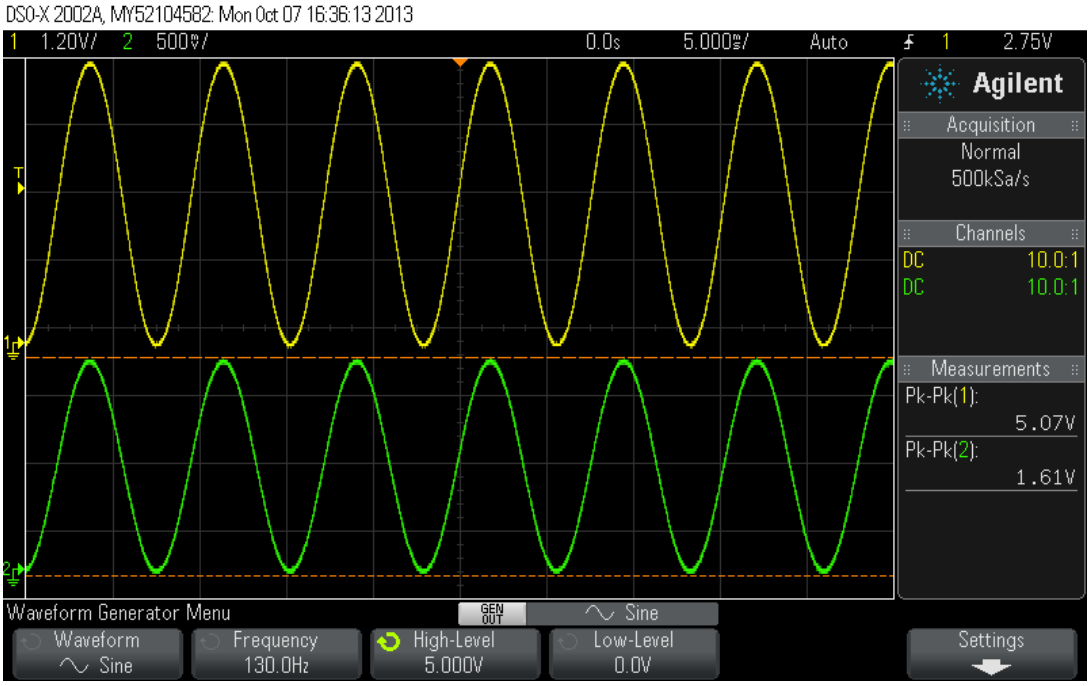

Figure 6.4: Original input signal (yellow) at 5V peak-to-peak being attenuated to 1.61V (green) peak-to-peak for the ADC input.

6.6 DAC

Since the DAC is a standard I^{2}C device, no loading of device tree overlays was required. It sits on the I^{2}C-2 bus, which is also shared with the cape identification EEPROM.

Normally, this bus runs at 100kHz. however, a modification was made to instead run the bus at the 3.4MHz high speed mode[4]. While this is unsupported by the EEPROM, the transition only occurs after the EEPROM has been read - and thus allows for a much faster update rate for the DAC.

Pictured below in figure 6.5 is the DAC outputting a 12-bit sine wave, generated from a lookup table. This was done by executing a custom test application, originally modified from sample code for the Raspberry Pi:

```
1 root@beagle:~# python2 DAC_test.py
```

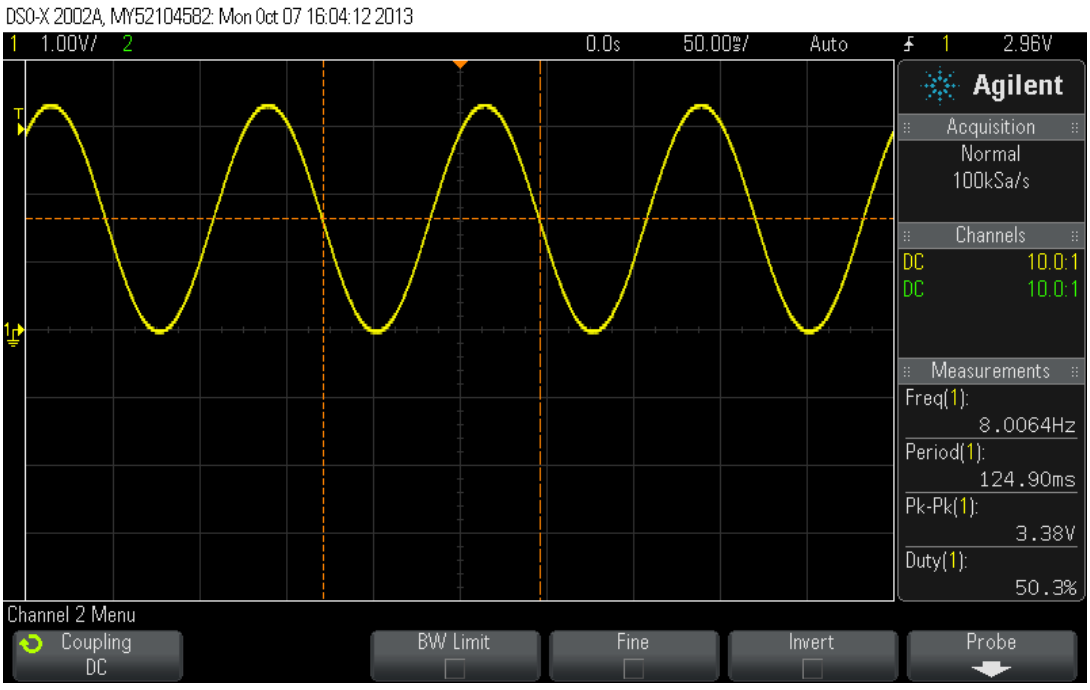

Figure 6.5: Generated 12-bit sine wave from the DAC.

[4]This modification is included in the LoCHILS patch applied when compiling the kernel as per section 5.2

6.7 EEPROM

The EEPROM is again an I^2C device running on the I^2C-2 bus. It is accessable in the folder `/sys/devices/ocp.2/4819c000.i2c/i2c-1/1-0054` by default. Here, a file `eeprom` allows for both read and write access to the EEPROM.

It can be tested by writing random data to the EEPROM and reading it back, verifying the data was the same. First, a randomly generated 256Kb data file is generated, along with a hash.

```
root@beagle:~# dd if=/dev/urandom of=/tmp/random_test bs=1000 count=32
32+0 records in
32+0 records out
32000 bytes (32 kB) copied, 0.003668 s, 8.7 MB/s

root@beagle:~# md5sum /tmp/random_test
988e2b1d59803ff775f31260eae30b09  /tmp/random_test
```

Next, this file is written to the EEPROM and the device is rebooted. Then, it is read back into a file, and a hash is calculated again on the read back data.

```
root@beagle:~# cd /sys/devices/ocp.2/4819c000.i2c/i2c-1/1-0054/
root@beagle:1-0054# cat /tmp/random_test > eeprom
root@beagle:1-0054# reboot
...
root@beagle:1-0054# cat eeprom > /tmp/random_test_readback
root@beagle:1-0054# md5sum /tmp/random_test_readback
988e2b1d59803ff775f31260eae30b09  /tmp/random_test_readback
```

Since the hashes match, the data has been successfully written to and read from the EEPROM.

6.8 Quadrature

By default, the Linux kernel for the Beaglebone Black does not have quadrature input support. Instead, this was added in the LoCHILS kernel patch, as detailed in section 5.2 [30].

The quadrature input module was tested by hooking up a motor with a rotary encoder to the A and B inputs of the first quadrature capture module. After loading the device tree overlay file:

```
root@beagle:~# echo bone_eqep1 > /sys/devices/bone_capemgr.9/slots
```

After loading the device tree overlay, a new directory `/sys/devices/ocp.2/48302000.epwmss/48302180.eqep` is available. This corresponds to the first eQEP hardware module on the Beaglebone Black, and contains 3 relevant files: `mode`, `period`, `position`.

The eQEP module supports two modes of operation, absolute and relative. Absolute mode is more useful for positioning, and shows the current position, whereas relative mode is more useful for velocity. It shows the number of encoder movements per `period`.

In this case, velocity measurement is more useful for the motor to be tested. Thus, the eQEP `mode` was set to relative mode and the `period` was set to 100ms:

```
root@beagle:~# cd /sys/devices/ocp.2/48302000.epwmss/48302180.eqep
root@beagle:48302180.eqep# echo 1 > mode
root@beagle:48302180.eqep# echo 100000000 > period
```

Next, the `position` can be read, indicating the number of encoder movements per period.

```
root@beagle:48302180.eqep# cat position
-531
```

The end result, -531, indicates that 531 counter-clockwise encoder movements occurred during the 100ms period. Reversing the polarity of the motor - while keeping the speed the same, results in 531 being read out. This corresponds to 531 clockwise encoder movements per period, and is the expected result. The corresponding input waveforms can be seen below in figure 6.6.

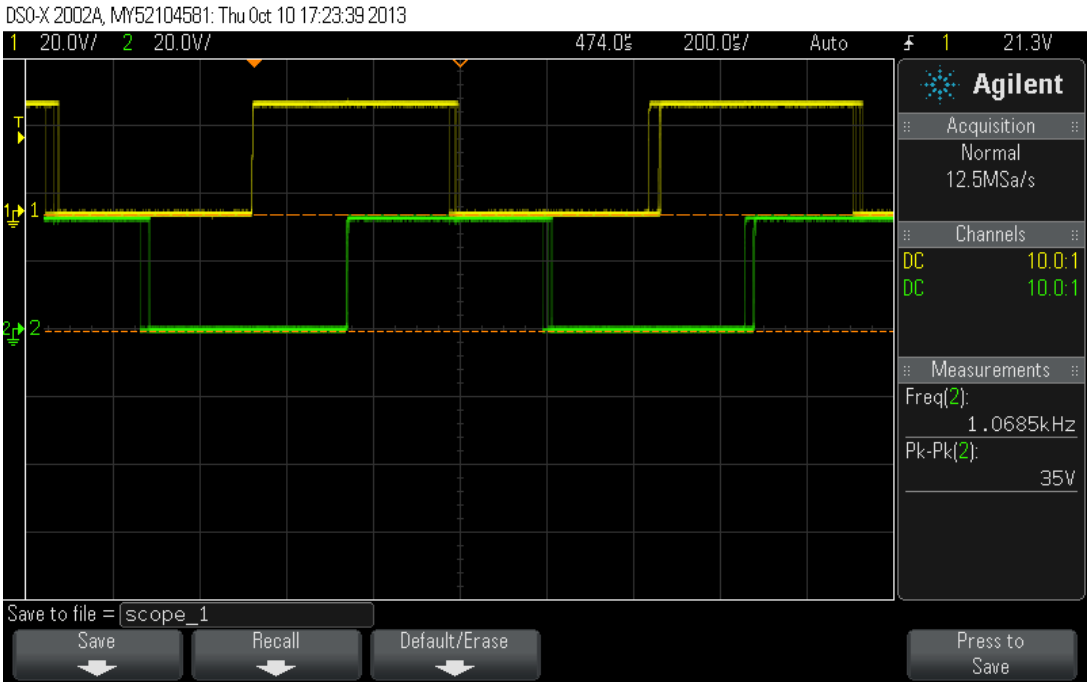

Figure 6.6: Quadrature output from motor's rotary encoder. Note that the scope was set to use a 10x probe, whereas crocodile clips were used, leading to an incorrect 10x increase in reported voltage.

Lastly, absolute mode was tested. While a motor isn't exactly the ideal test platform (a rotary encoder knob would have been ideal), it can still be used to verify the system. The eQEP module was placed in to absolute mode, and the position reset to 0:

```
root@beagle:48302180.eqep# echo 0 > mode
root@beagle:48302180.eqep# echo 0 > position
```

Next, the motor was turned on. Observing the position output, it was steadily increasing - indicating the motor moving in a clockwise fashion. The motor was then switched off, and the position output remained constant. Lastly, it was turned on again but with the polarity reversed, and a steady decrease in the position output was observed. This behavior matched what was expected from the quadrature capture module.

6.9 SPI

Unfortunately, since no SPI devices were available for testing (and time did not permit programming a microcontroller for use), a loopback SPI test was used. First, the device tree overlay was loaded, and the user space SPI access bus was created:

```
root@beagle:~# echo BB-SPI0-01 > /sys/devices/bone_capemgr.9/slots
```

Next, a standard SPI loopback test was run using the spidev sample application. For this, the D0 and D1 lines on the cape were connected to each other.

```
root@beagle:~# spidev_test -D /dev/spidev1.0
spi mode: 0
bits per word: 8
max speed: 500000 Hz (500 KHz)

FF FF FF FF FF FF
40 00 00 00 00 95
FF FF FF FF FF FF
FF FF FF FF FF FF
FF FF FF FF FF FF
DE AD BE EF BA AD
F0 0D
```

6.10 Not completely verified

Unfortunately, the resources were not available to test 2 of the peripheral buses - I^2C and CAN. Their basic operation was verified by loading the suitable device tree overlays and ensuring the device nodes were created.

6.10.1 I²C

Due to the fact that the EEPROM and DAC are both I²C peripherals (albeit on a different bus), it can be reasoned that I²C support is fully functioning. The device tree overlay was successfully loaded, and the Linux bus driver was created:

```
root@beagle:~# echo BB-I2C2 > /sys/devices/bone_capemgr.9/slots
root@beagle:~# ls /dev/i2c-2
/dev/i2c-2
```

6.10.2 CAN

Similar to I²C, the CAN device tree overlay was loaded, and the presence of the device checked by

```
root@beagle:~# echo BB-DCAN1 > /sys/devices/bone_capemgr.9/slots
root@beagle:~# ip link set can0 up type can bitrate 500000
root@beagle:~# ifconfig can0 up
```

Chapter 7

Simulink Integration

> On two occasions I have been asked, - *"Pray, Mr. Babbage, if you put into the machine wrong figures, will the right answers come out?"* In one case a member of the Upper, and in the other a member of the Lower House put this question. I am not able rightly to apprehend the kind of confusion of ideas that could provoke such a question.
>
> – Charles Babbage

After implementing a suitable software platform and verifying that hardware access is working as expected, the Simulink portion of the work can be dealt with.

7.1 Compiler

Simulink requires that suitable `mex` files are available on the host platform - even though they are destined for the target ARM platform. This requires the installation and configuration of a suitable compiler. As mentioned in the documentation, the Windows SDK is well suited and provided for free by Microsoft.

However, one needs to be careful during the installation process. If the host computer already has the `Visual C++ 2010 redistributable` installed, it needs to be uninstalled before attempting to install the SDK. This is caused by the SDK including an older version of the redistributable, which causes the setup to fail if a newer version is installed [31].

After this step has been completed, MATLAB needs to be updated to use the new compiler. This can be done by executing the `mex -setup` command from the primary MATLAB window:

```
>> mex -setup

Welcome to mex -setup.  This utility will help you set up
a default compiler.  For a list of supported compilers, see
http://www.mathworks.com/support/compilers/R2013a/win64.html

Please choose your compiler for building MEX-files:

Would you like mex to locate installed compilers [y]/n?

Select a compiler:
[1] Microsoft Software Development Kit (SDK) 7.1 in
    C:\Program Files (x86)\Microsoft Visual Studio 10.0

[0] None

Compiler: 1

Please verify your choices:

Compiler: Microsoft Software Development Kit (SDK) 7.1
Location: C:\Program Files (x86)\Microsoft Visual Studio 10.0

Are these correct [y]/n? y

Done . . .
```

7.2 Blocks created

6 blocks were successfully created and tested, which will be described in more detail in the sections below. For reference, the code that is referred to below (eg, `gpio.h`) can be found on the CD provided with this document, in addition to the library containing the blocks themselves.

7.2.1 GPIO block

The GPIO block consists of the GPIO S-Function, which interfaces with the `gpio.h` library, as well as data type converters. The type conversion is necessary since Simulink signals tend to be of type `double`, whereas the GPIO S-Function expects `uint8`. The direction of the GPIO pins is configured via mask parameters.

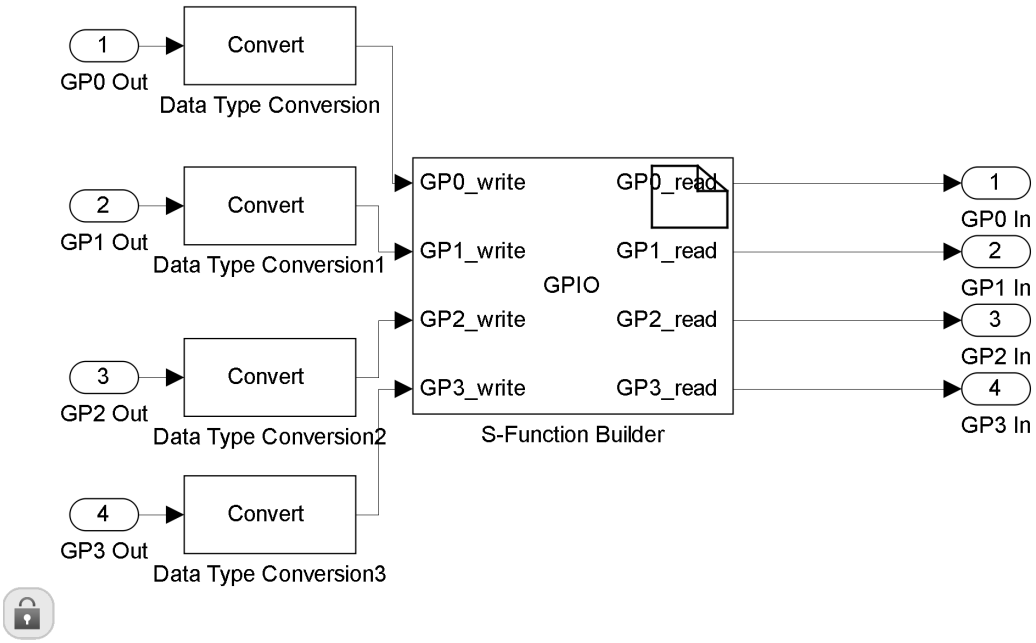

Figure 7.1: Internal workings of the GPIO block.

7.2.2 PWM block

Here, the PWM block simply takes the input ports and passes them on to the PWM S-Function. The PWM S-Function interfaces with the `pwm.h` library.

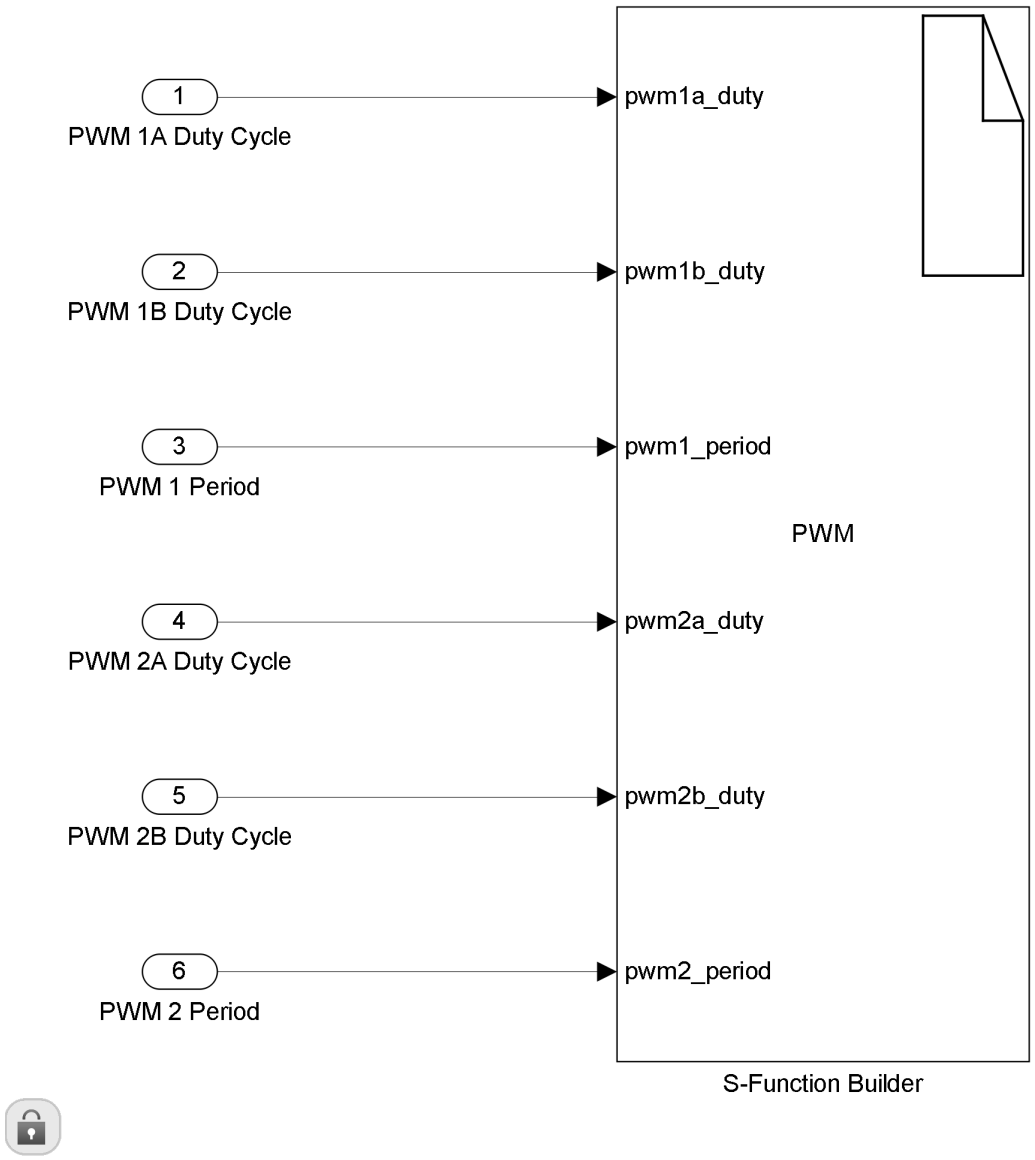

Figure 7.2: Internal workings of the PWM output block.

7.2.3 UART block

The UART block requires a data input and length input. This data input is padded with zeros to match the port size of the S-Function, before being passed in to the UART S-Function itself, which interfaces with the `uart.h` library.

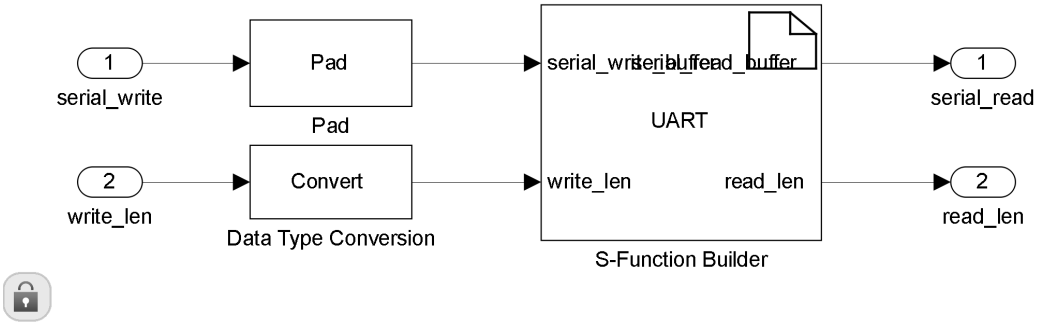

Figure 7.3: Internal working of the UART block.

7.2.4 ADC block

The ADC block is extremely simple. It is a small wrapper around `adc.h` which provides the analog values read in.

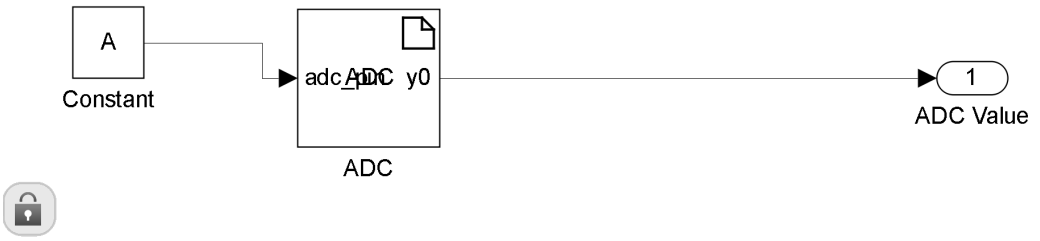

Figure 7.4: Internal working of the ADC block.

7.2.5 Logging Blocks

Two logging blocks were created, for differing purposes. The first, Text Logger, was designed to log arbitrary text data. It uses the same interface as the UART, requiring a data in and length in. This then gets written to a file containing the current date and time, along with the unique ID specified.

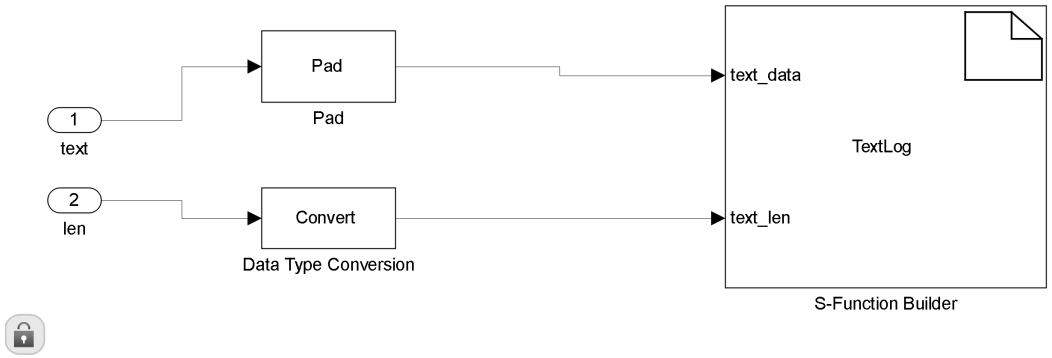

Figure 7.5: Internal workings of the text logging block.

The CSV Logger was designed for logging data. It takes in an array of doubles, again with a length parameter, and writes them to a CSV formatted file on the Beaglebone Black.

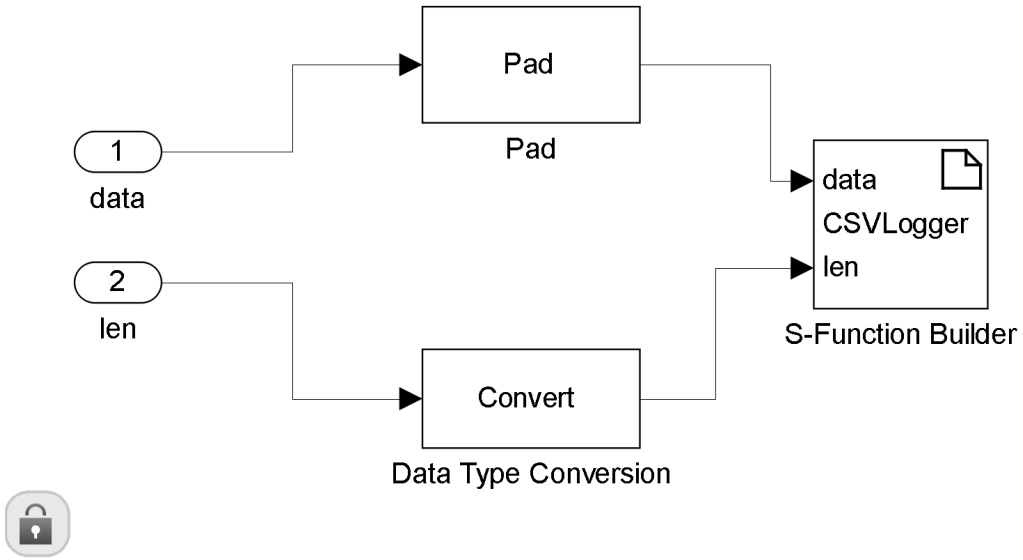

Figure 7.6: Internal workings of the CSV logging block.

7.3 Unimplemented blocks

Unfortunately, due to time constraints, the quadrature, PWM input and DAC blocks were not completed in time. However, the extra work required to implement these blocks is minimal - perhaps a day at most. This is because the underlying support is present, and the hardware has been verified.

7.4 Examples of blocks

In addition to the blocks, example models to illustrate and test their usage were also created. These examples are included on the CD provided.

7.4.1 GPIO example

Here, the GPIO block is used as an output, with the input coming from a pulse generator. The pulse generator generates a square wave at a fixed frequency which is then fed into the first GPIO output. Connecting an oscilloscope to the port labelled *GP0* on the cape reveals the pattern as expected.

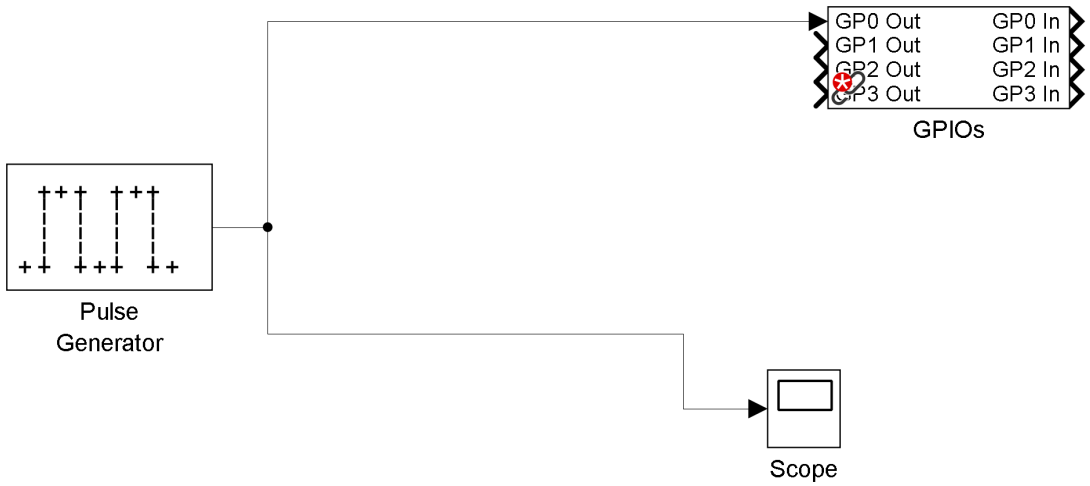

Figure 7.7: Example Simulink block diagram illustrating the use of the GPIO block.

Internally, the GPIO block automatically performs data type conversion - as binary values are the only ones that make sense. It is hidden under a mask that allows for the configuration of the specific direction of the GPIO port.

7.4.2 PWM example

A slightly more complicated example was created to test the PWM output capabilities. Here, a MATLAB function outputs an incrementing number on its output, `y`, which feeds into the duty cycle of PWM port 1A.

```
function y = fcn()
%#codegen

persistent count;

if isempty(count)
    count = 2000000;
end

count = count + 3600;
y = count;
```

It starts at 2 000 000 and ramps up to 20 000 000 over 5000 iterations, or 10 seconds at 500Hz. The period is fixed at 20 000 000 nanoseconds, and so this corresponds to starting at a duty cycle of 10% and increasing to 100%. As mentioned previously, the duty cycle input to the block is a nanosecond value and not a percentage value.

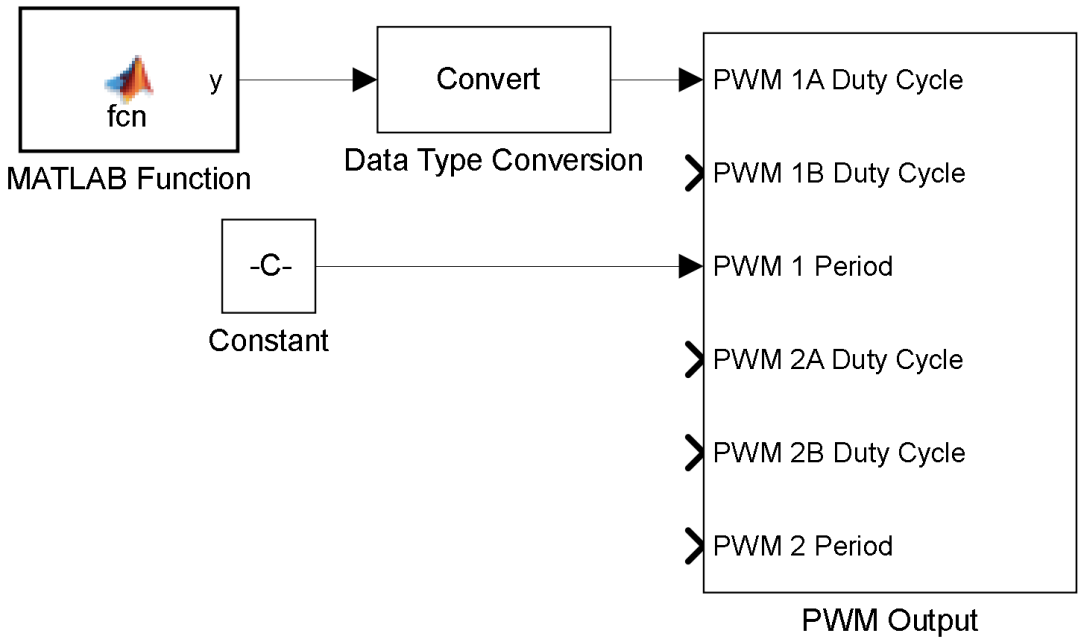

Figure 7.8: Example Simulink block diagram showing the PWM output block in action.

7.4.3 UART example

Demonstrating the UART is again performed with the help of MATLAB function code. As shown in figure 7.9, a MATLAB function is connected to the write ports of the UART block. This MATLAB function outputs once the data to be written, as well as the length of the data. On the other side of the block, the read length is connected to an `if` block. Should the length not be 0, as is the case when data has been received, the triggered subsystem is run - which stores the received data to the MATLAB workspace.

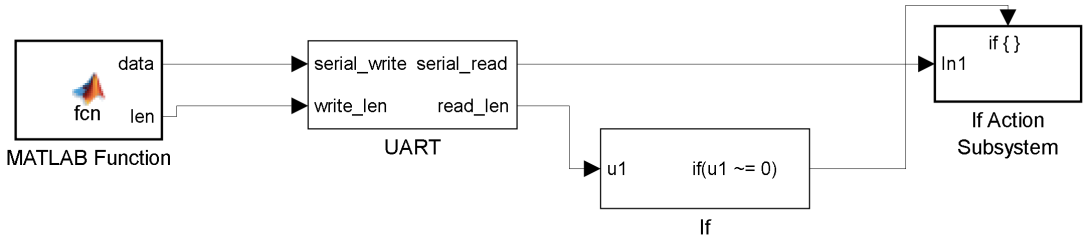

Figure 7.9: Example Simulink block diagram containing the UART block.

The code for the MATLAB function is shown below. Effectively, it puts `dataout` on the output bus, and specifies a length of 7. This can be checked by connecting a serial terminal to the Beaglebone Black. When the model starts, the terminal will read `dataout`, and any data sent through the terminal will be saved into the MATLAB workspace.

```
1  function [data, len]= fcn
2  %#codegen
3
4  % Use a persistent variable so we only send the data once.
5  persistent sentData;
6  if isempty(sentData)
7     sentData = 0;
8  end
9
10 if sentData == 0
11     data = uint8('dataout');
12     len = 7;
13     sentData = 1;
14 else
15     % Data needs to be set on both code paths!
16     data = uint8('dataout');
```

```
17
18      % Length controls what actually gets sent out by the block. If 0,
19      % nothing goes out.
20      len = 0;
21  end
```

7.4.4 ADC example

The ADC is presented in a much simpler example than the other blocks. Here, it is directly connected to a Simulink scope. Due to Simulink's external mode, the scope will be updated with the values that the ADC reads in. Thus, a pot can be connected to the analog input channel, the ADC voltage reference, and analog ground. Turning the pot results in the ADC value changing and being directly represented on the scope.

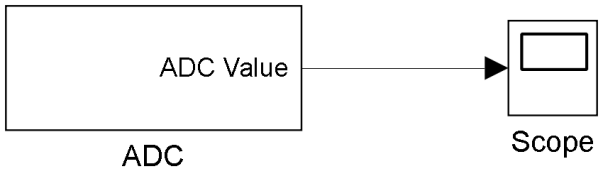

Figure 7.10: Example Simulink block diagram using the ADC to transfer data to a local scope via external mode.

7.4.5 Logging example

The logging example consists of both logging blocks, CSV and Text logging. It has two text logging blocks, configured with different block names which thus log to different files. Again, MATLAB functions are used to create the data that will be logged.

The code for the first MATLAB function block feeding the CSV logging block is shown below. It puts an array of numbers with length 3 out, which then gets stored in CSV file by the CSV logging block.

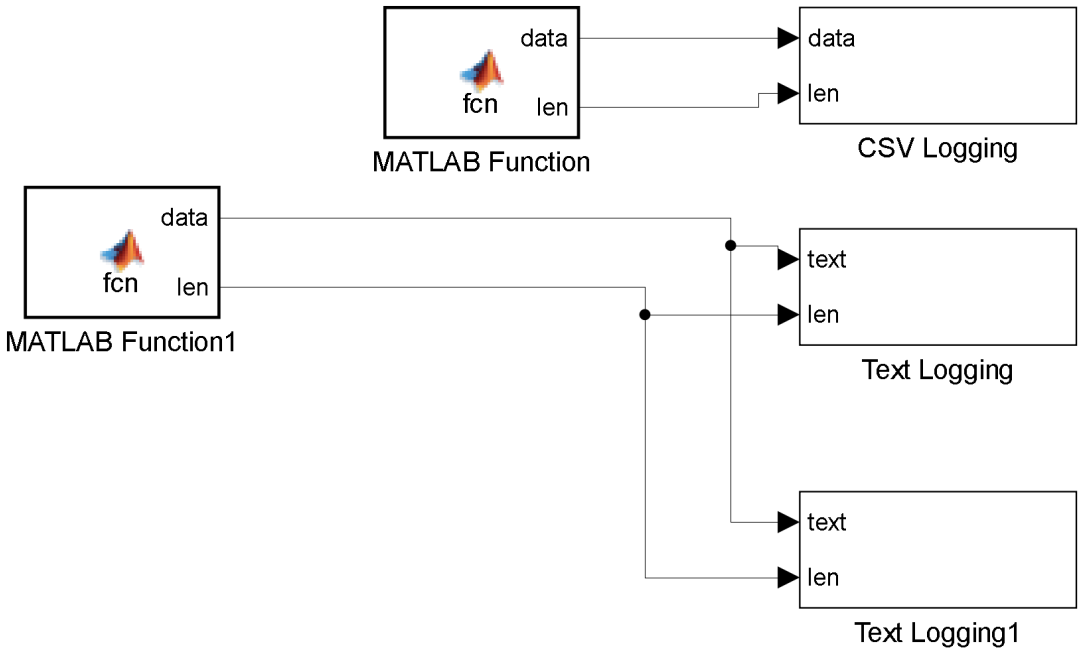

Figure 7.11: Example Simulink block diagram showing 2 text logging blocks and a single CSV logging block.

```
function [data, len]= fcn
%#codegen

% Use a persistent variable so we only send the data once.
persistent sentData;
if isempty(sentData)
   sentData = 0;
end

if sentData == 0
    data = [1.234, 14.412, 345324.1];
    len = 3;
    sentData = 1;
else
    % Data needs to be set on both code paths!
    data = [1.234, 14.412, 345324.1];

    % Length controls what actually gets sent out by the block. If 0,
    % nothing goes out.
    len = 0;
end
```

Next, the code for the second MATLAB function block is shown below. Essentially, it does the same thing, as `uint8()` converts text to an array of `uint8`. Both of these snippets of code operate in the same manner as the serial block, allowing easy connection between the different blocks.

```
function [data, len]= fcn
%#codegen

% Use a persistent variable so we only send the data once.
persistent sentData;
if isempty(sentData)
   sentData = 0;
end

if sentData == 0
    data = uint8('logdata');
    len = 7;
    sentData = 1;
else
    % Data needs to be set on both code paths!
    data = uint8('logdata');

    % Length controls what actually gets sent out by the block. If 0,
    % nothing goes out.
    len = 0;
end
```

7.5 Running on target hardware

Having created the supporting blocks, Simulink's *Run on Target Hardware* support can now be initalised. This is performed by installing the Beagleboard support package, as is pictured below

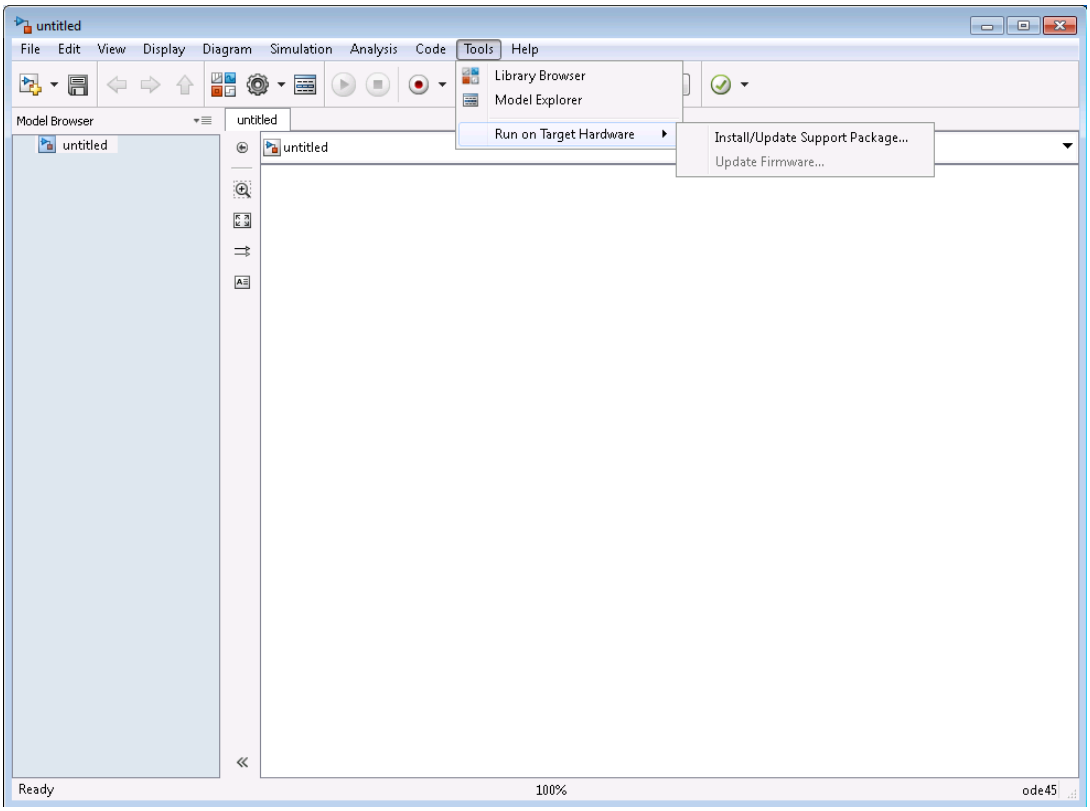

Figure 7.12: Step 1 of installing the Beagleboard support package - selecting the menu option.

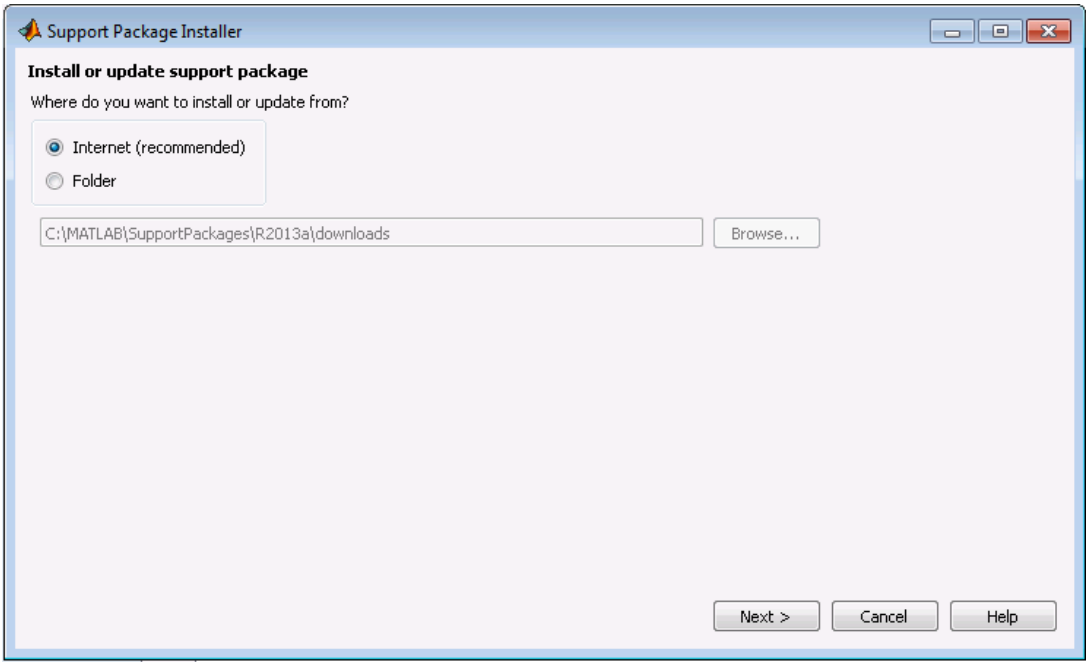

Figure 7.13: Step 2 of installing the Beagleboard support package - select the package source as the internet

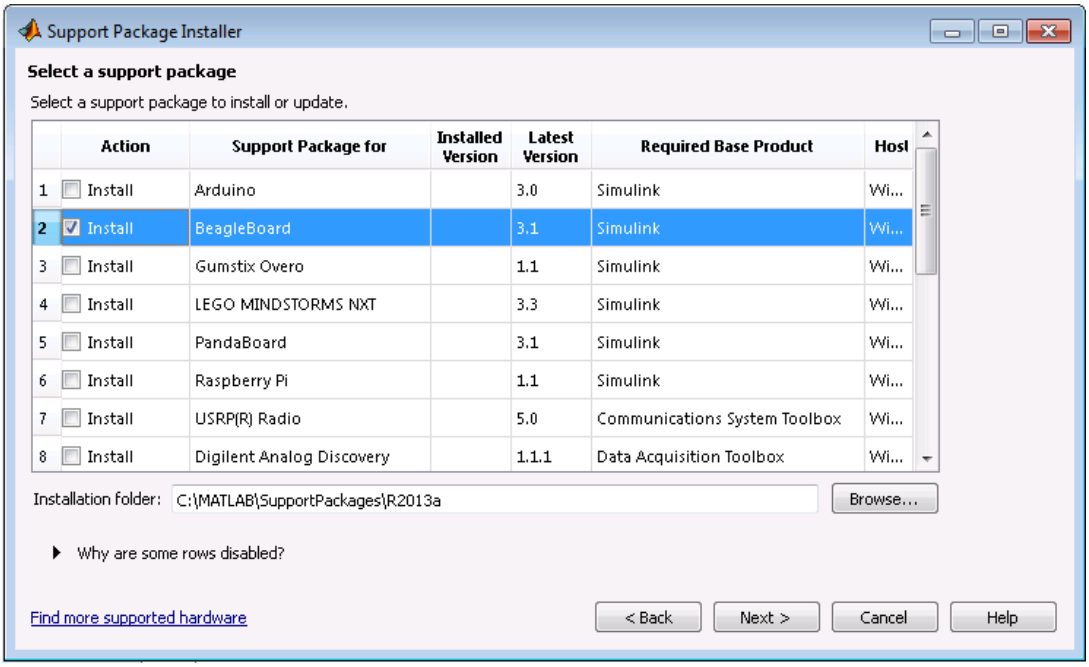

Figure 7.14: Step 3 of installing the Beagleboard support package - configure the Beagleboard package as to be installed. Note that the firmware update can be skipped, as it is only relevant for the Beagleboard-xM and not the Beaglebone Black.

Once the Beagleboard support package has been installed, the model can be prepared for use on the target platform. This involves setting the target hardware parameters, such as hostname, as well as the model parameters like sample time.

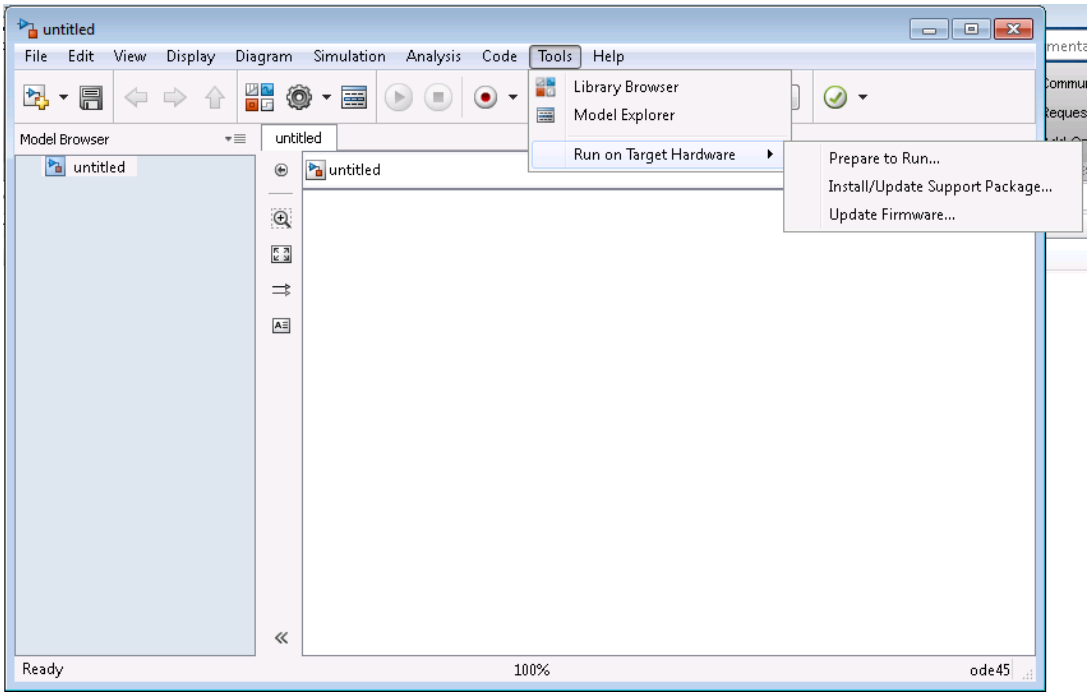

Figure 7.15: After the support package has been installed, the model is prepared for running on the target platform.

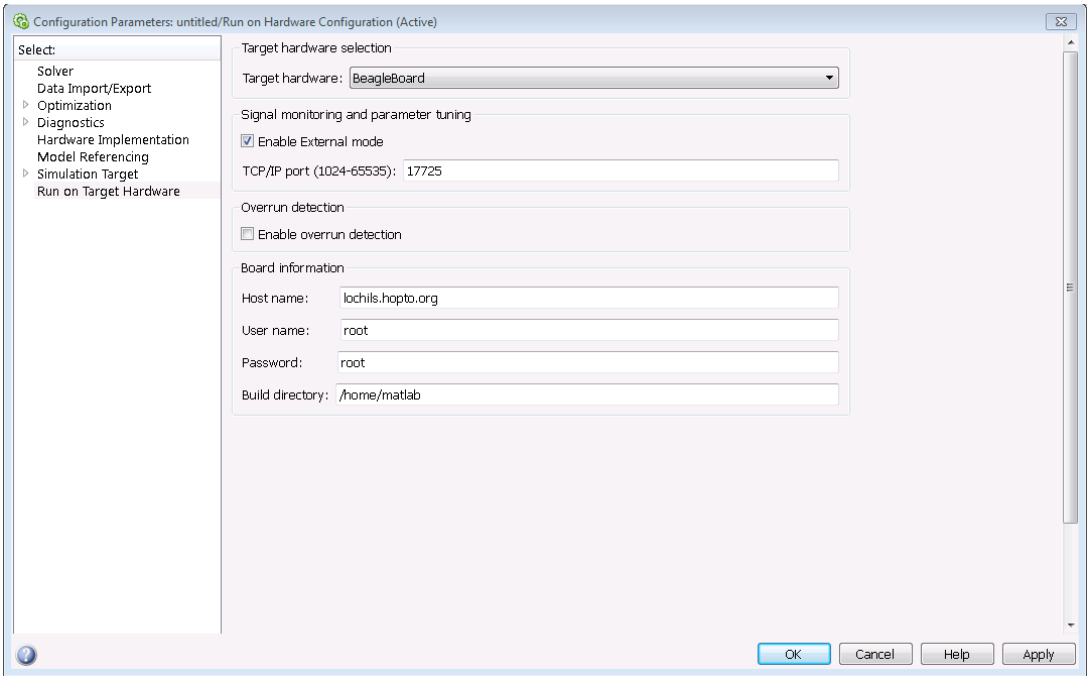

Figure 7.16: The dialog for configuring the target hardware. Here, external mode is enabled (which allows communication back with MATLAB from the hardware) and the IP address is set to the Dynamic DNS hostname.

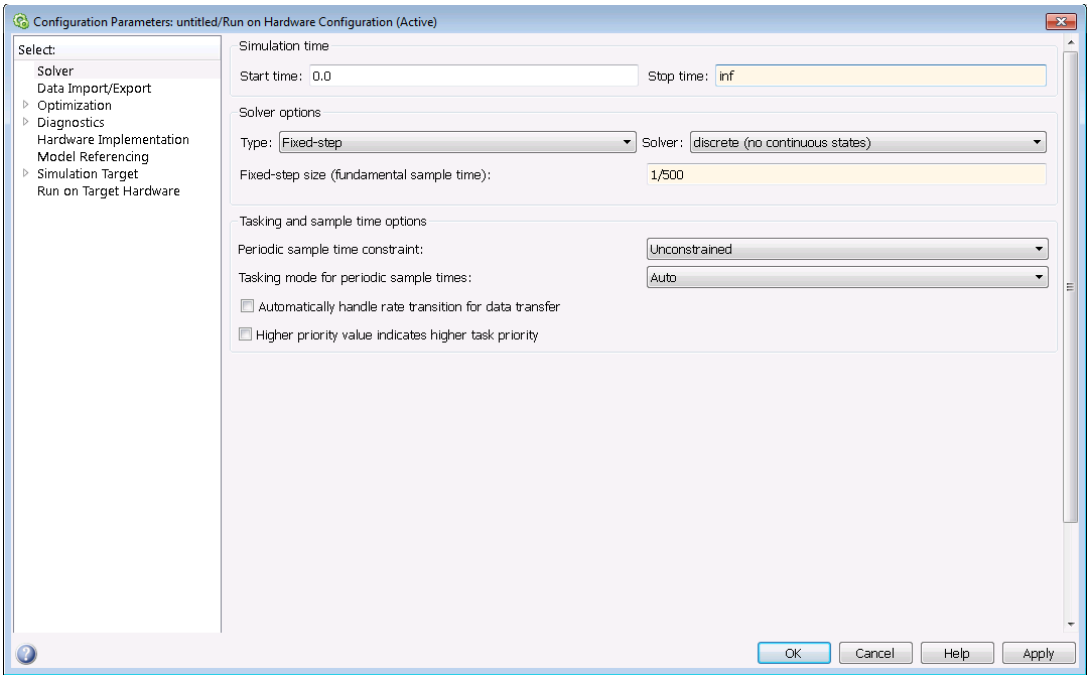

Figure 7.17: Finally, the sample time of the system is set to 1/500, or 500Hz.

Chapter 8

Benchmarking

"If you don't care about quality, you can meet any other requirement."

– Gerald M. Weinberg

While the previous chapter focused on the implementation and verification of individual components of the system, this chapter focuses on the system performance as a whole. In addition, comparisons will be drawn with the standard Simulink Run on Target Hardware support for the Beagleboard-xM.

In order to make this possible, a MicroSD card was loaded with the latest Simulink Ubuntu image and placed into the Beagleboard-xM for testing.

8.1 GNU/Linux performance

The first array of tests performed focuses on the relative GNU/Linux performance of the two platforms. This is split up into 6 different tests.

The first is the boot time measurement test. This test reflects how long the system takes to boot up from being in a completely off state. This is mostly a test of the efficiency of the userspace design - as a user space environment that starts more services will take longer to load. Furthermore this test is also dependent on disk read performance.

Next, the number of tasks at idle immediately after bootup finishes is evaluated. A higher number of tasks indicates that more services have been started, and thus an increased probability of interruption when a CPU bound task is running.

This is followed by a test of the network latency of both systems. The latency of the wired Ethernet ports will be measured using the `ping` utility.

Fourthly, disk benchmarks are performed on both systems. These benchmarks measure purely the raw disk performance, without taking into account the CPU or other factors.

Fifthly, a blended CPU / memory / disk test is run. This test reports on the strength of the CPU at evaluating number crunching tasks, as well as relative memory bandwidth combined with disk performance. It also takes into account the surrounding operating system and compiler[1]. Since this a proper benchmark, it only needs to be run once - as internally it performs multiple runs to ensure consistent results.

Lastly, a sample open source project is compiled[2]. This test mimics real world use, by being a blend of both disk, CPU and memory usage. In particular as was described in section 7.5, this is highly relevant for the running of the model - as it is compiled on the target platform.

[1]Newer compilers tend to have additional optimisations, so that running a benchmark compiled with a different compiler can produce different results.

[2]It just so happens that the project chosen was the benchmark used in test #5

8.1.1 Results

The bootup test was performed by connecting the power to the system, then measuring with a stopwatch the time until the login prompt appears on the serial console. This test was repeated 3 times to ensure an accurate measurement of the boot time. Each time, the system was shut down properly using the `poweroff` command, to ensure that a file system check[3] did not have to occur on startup.

Run	Beagleboard-xM Ubuntu	Beaglebone Black ArchLinux
1	63 seconds	24 seconds
2	60 seconds	24 seconds
3	61 seconds	26 seconds
average	61.3 seconds	24.6 seconds
std dev	1.52	1.15

Table 8.1: Time for full operating system boot.

The process test was performed by measuring the number of processes present on the system after bootup, using `top`[4] via the serial console. While the presence of `top` will add an extra process to the count, this makes no relative difference as it is added for both systems. This test was repeated 3 times, with the exact same results being obtained for each run. This is expected, as the boot process should be fairly deterministic.

Run	Beagleboard-xM Ubuntu	Beaglebone Black ArchLinux
1	99 processes	55 processes
2	99 processes	55 processes
3	99 processes	55 processes
average	99 processes	55 processes
std dev	0	0

Table 8.2: Number of processes running immediately after startup.

The network latency was calculated by sending 20 ICMP packets to the Ethernet router that the devices were plugged in to. This was done by using the `ping` command. The average of these 20 round-trips was noted, and the test itself was repeated 3 times. While the speed of the router can play a part in influencing this result, it would again be equal for both systems and thus a relative comparison is still valid.

[3] Even if such a check did occur, the impact would likely be minimal, as both devices are using journaled file systems, which can regain consistency extremely quickly.

[4] `top` is a command used for displaying running processes and similar statistics, similar to the Task Manager in Windows.

Run	Beagleboard-xM Ubuntu	Beaglebone Black ArchLinux
1	1.213 msec	0.482 msec
2	1.213 msec	0.489 msec
3	1.208 msec	0.451 msec
average	1.211 msec	0.474 msec
std dev	0.003	0.02

Table 8.3: Comparison of network latency from the `ping` utility.

The disk throughput test was performed by reading in 256MB of sequential data from the primary storage device to memory. This was done by using the `dd` command. Before performing each instance of this test, it was important to make sure that the Linux disk cache had been flushed - otherwise the data might be read from cache instead.

```
1 root@beagle:~# sync; echo 3 > /proc/sys/vm/drop_caches
```

Run	Beagleboard-xM Ubuntu	Beaglebone Black ArchLinux
1	8.1MB/s	19.3MB/s
2	8.3MB/s	19.0MB/s
3	8.8MB/s	19.3MB/s
average	8.4MB/s	19.2MB/s
std dev	0.36	0.17

Table 8.4: Sequential disk IO performance.

It is important to note that this benchmark is only measuring sequential read performance, and not write performance nor random disk performance.

The UnixBench index was then calculated by running the UnixBench v5 benchmark. This tested all round system performance by producing an index score based on conventional benchmarks such as Whetstone and Dhrystone, as well as real world tasks, such as pipe throughput, process creation and system call overhead. Each test was performed multiple times by the UnixBench tool, and thus only a single run was required to get a score with a good confidence.

Beagleboard-xM Ubuntu	Beaglebone Black ArchLinux
73.8 UnixBench Index	89.7 UnixBench Index

Table 8.5: Results from UnixBench.

Finally, the compilation test was performed. This test centred around the compilation time required for the UnixBench benchmark used in the previous test. It was compiled 3 times, each time being freshly extracted from the source archive, in addition to flushing the disk cache - for the same reasons as test 4.

Run	Beagleboard-xM Ubuntu	Beaglebone Black ArchLinux
1	43 seconds	28 seconds
2	45 seconds	25 seconds
3	43 seconds	26 seconds
average	43.6 seconds	26.3 seconds
std dev	1.15	1.5

Table 8.6: Time to compile UnixBench application.

8.1.2 Interpretation

From the results gathered in the above section, it is clear that the Beaglebone Black comprehensively outperforms the Beagleboard-xM when it comes to benchmarks. There are numerous reasons for this, the primary one being that even though the Beagleboard-xM and Beaglebone Black both ship with the same Cortex-A8 core as their main CPU, the Beagleboard-xM is limited to 800MHz. This is due to the old version of Ubuntu that was chosen by MathWorks as their reference implementation. In contrast, the Beaglebone Black runs at the full 1GHz clock speed.

Furthermore, the two systems are running differing kernels. The Beagleboard-xM comes with a 3.2 kernel, whereas the Beaglebone Black utilises the 3.8 kernel. As the Linux kernel is continuously undergoing improvements, this likely plays a factor in the results.

The disparity in disk IO can easily be explained by the fact that while the Beagleboard-xM requires an external MicroSD card, the Beaglebone Black has an onboard 2GB eMMC memory chip - allowing for operation without the need to rely on an external MicroSD [21].

The superior network latency of the Beaglebone Black is due to the architecture of the board itself. While the Beagleboard-xM uses an onboard USB Ethernet chip, the Beaglebone Black has an on-processor Ethernet PHY. This contributes greatly to the performance in this regard, completely eliminating the overhead of USB from dealing with the network. It is interesting to note that should the Beagleboard-xM's USB bus become saturated, network performance will suffer heavily. Whereas in the case of the Beaglebone Black, it will be completely unaffected [19] [21].

Lastly, the boot performance and number of processes at startup can be attributed to the differing GNU/Linux distributions. The customised ArchLinuxArm based distribution that was created for the Beaglebone Black is optimised for performance and real-time operation, and thus beats out the standard Ubuntu setup provided by MathsWorks.

8.2 Simulink performance

Having established a base level of performance for the GNU/Linux systems running on the Beaglebone Black and the Beagleboard-xM, the two can now be compared in the context of running Simulink models.

8.2.1 Throughput

In order to quickly create a complex model for testing the CPU performance of the differing systems, the `fdatool` was used to design a high-order filter. Higher order filtering is known to be particularly CPU intensive and thus is a suitable tool for comparing raw CPU throughput between systems. Figure 8.1 below illustrates the testing setup used. Two sine wave sources, one of 25Hz and one of 50Hz are summed together before being passed through a 1000^{th} order bandstop filter that takes out the range of 20Hz to 30Hz.

Figure 8.1: Simulink block diagram used for testing CPU performance.

This model was then run on both systems for a period of 5 minutes, during which the CPU load was monitored. The results of this can be seen below in table 8.7.

Beagleboard-xM Ubuntu	Beaglebone Black ArchLinux
23% CPU Load	12% CPU Load

Table 8.7: 5 minute average CPU usage when running the model described in figure 8.1.

8.2.2 Launch performance

To measure the launch performance[5] of each system, the model used for throughput testing in section 8.2.1 was retained, but instead the time taken from clicking the *Run* button as shown in figure 8.2 to the time when the model actually starts processing data is measured. This was done with a stopwatch, as the times involved are fairly large.

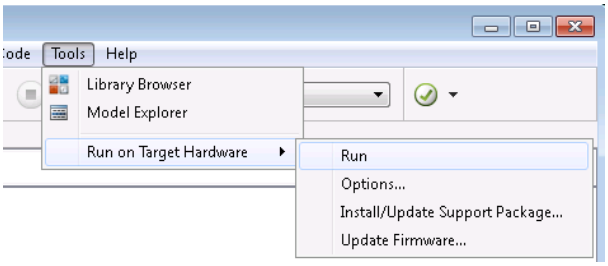

Figure 8.2: Starting point for measuring the launch performance of models on the different platforms.

Three runs were performed on each platform, with the results shown below in table 8.8

Run	Beagleboard-xM Ubuntu	Beaglebone Black ArchLinux
1	119 seconds	81 seconds
2	111 seconds	76 seconds
3	113 seconds	75 seconds
average	114 seconds	77 seconds
std dev	4.16	3.21

Table 8.8: Time to launch simulation.

8.2.3 Jitter

In order to measure and quantify the real-time nature of the system, jitter measurements were performed. These were done by toggling a GPIO pin to create a pulse train signal. While this is clearly an application more suitable for the hardware PWM supported provided by the Beaglebone Black, the task was explicitly performed using manual GPIO toggling to see the varying affects of jitter on the system.

[5]Launch performance refers to how long the model takes to start running on the platform.

To do this, the simulation sampling rate was set at 500Hz, and a simple model loaded onto both the reference Beagleboard-xM and the Beaglebone Black. These two models differ only in their hardware output support. The Beaglebone Black's model uses the GPIO output block created as part of this thesis and can be seen in figure 8.3, whereas the Beagleboard-xM uses the reference Simulink *GPIO Write* block as in figure 8.4.

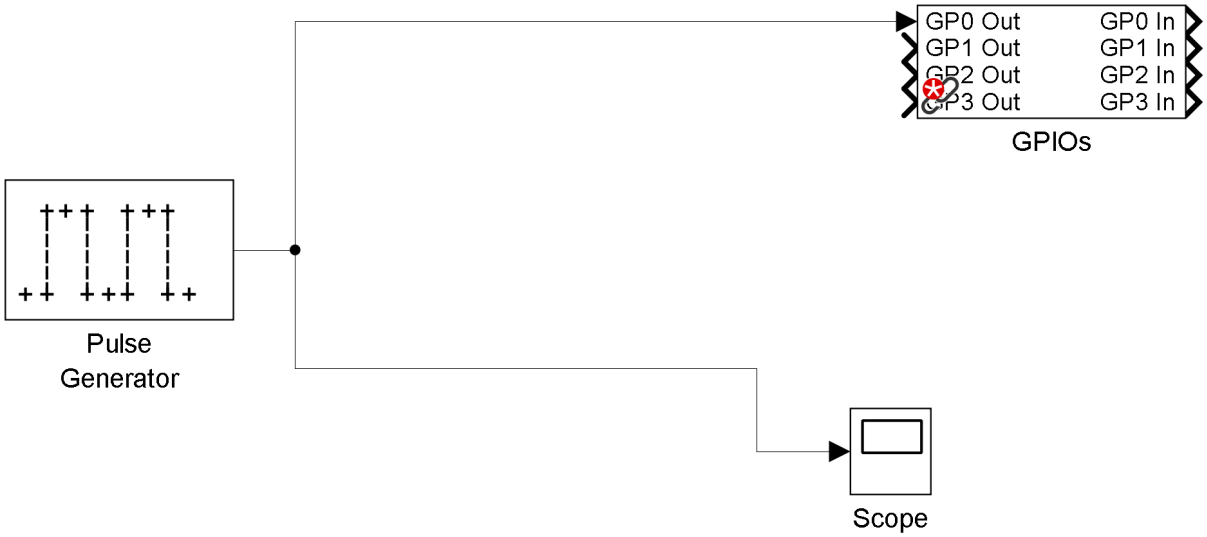

Figure 8.3: Simulink block diagram used for jitter measurement on the Beaglebone Black.

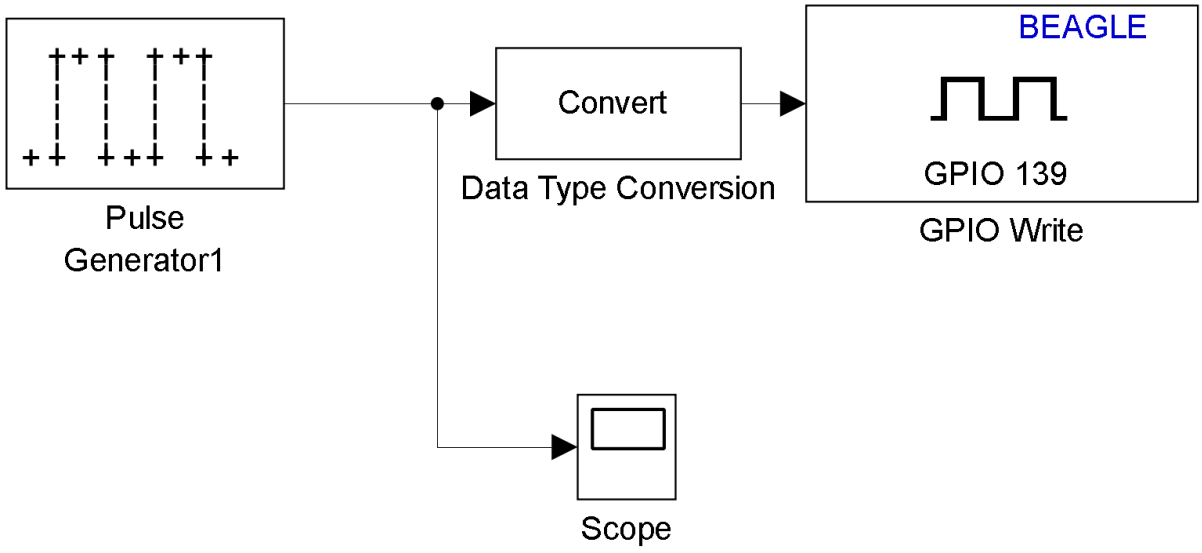

Figure 8.4: Simulink block diagram used for jitter measurement on the Beagleboard-xM.

Here, the pulse block was configured with an amplitude of 1 - corresponding to a high logic level, a period of 6 samples and a pulse width of 3 samples. as mentioned previously, the sample frequency was fixed at 500Hz.

Next, each model was loaded on to the respective target hardware platform, and executed. The GPIO pins being toggled were connected to an oscilloscope to be measured. In order to suitably measure the jitter, the oscilloscope was put into variable persistence mode, where the previous signal persists on the screen for a maximum of 60 seconds. This essentially creates a rolling average of the waveform on the screen, easily allowing things such as jitter to be detected. First, the output from the model running on the Beagleboard-xM is shown below in figure 8.5.

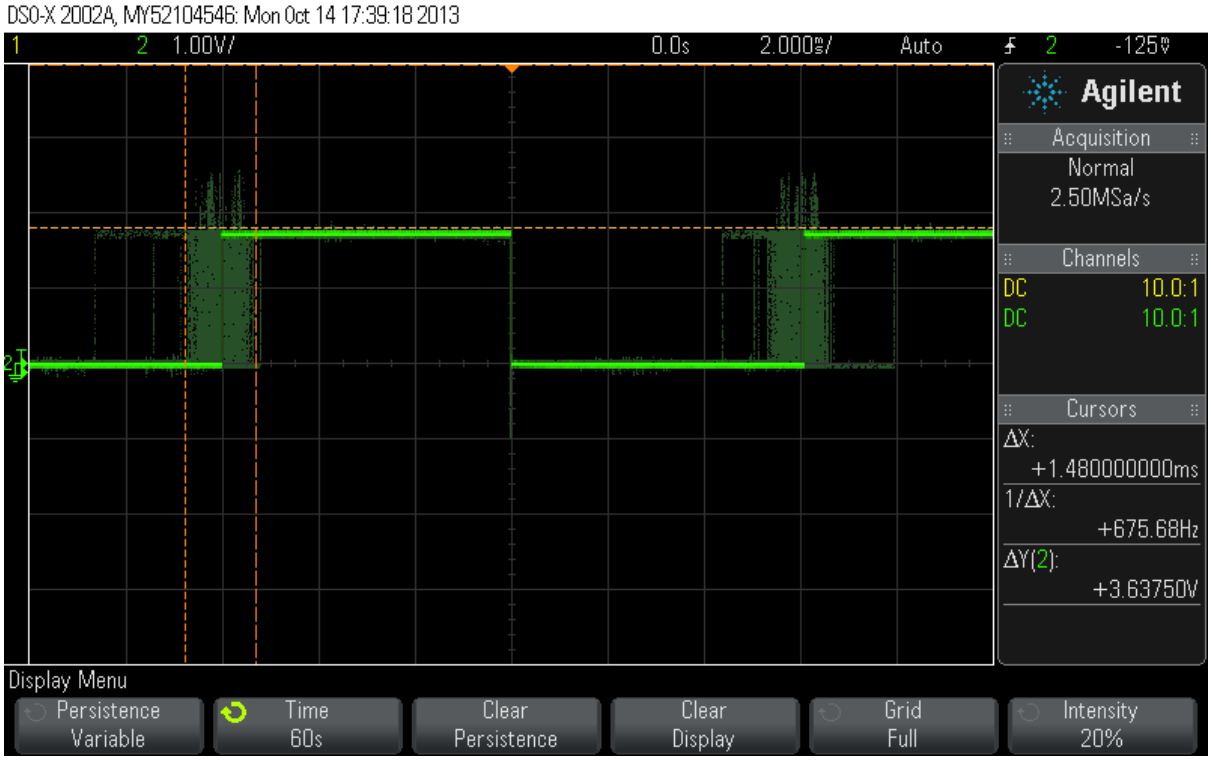

Figure 8.5: Scope capture for the generated pulse train from the Beagleboard-xM.

The large green area in between the two cursors shows the timebase jitter present on the signal. Since the signal is slow compared to the speed of the oscilloscope, the effects of trigger jitter can be ignored. While the total green area does show the "almost worst case"[6] jitter, the average case jitter cannot be directly quantified in this way. Unfortunately, the oscilloscope used did not have the capability for either mask testing or saving the full persistence history in a workable format for analysis.

However, the average jitter could be seen when viewing the scope display in person. In the Beagleboard-xM case, it was particularly large, as the signal visibly moved between the two extremes. The opposite however was true for the Beaglebone Black, for which the same measurement can be seen below in figure 8.6.

[6]Since the Beagleboard-xM system is not real-time, the worst case jitter is unbounded. You can see a faint line off to the left of figure 8.5 as a case of extreme jitter.

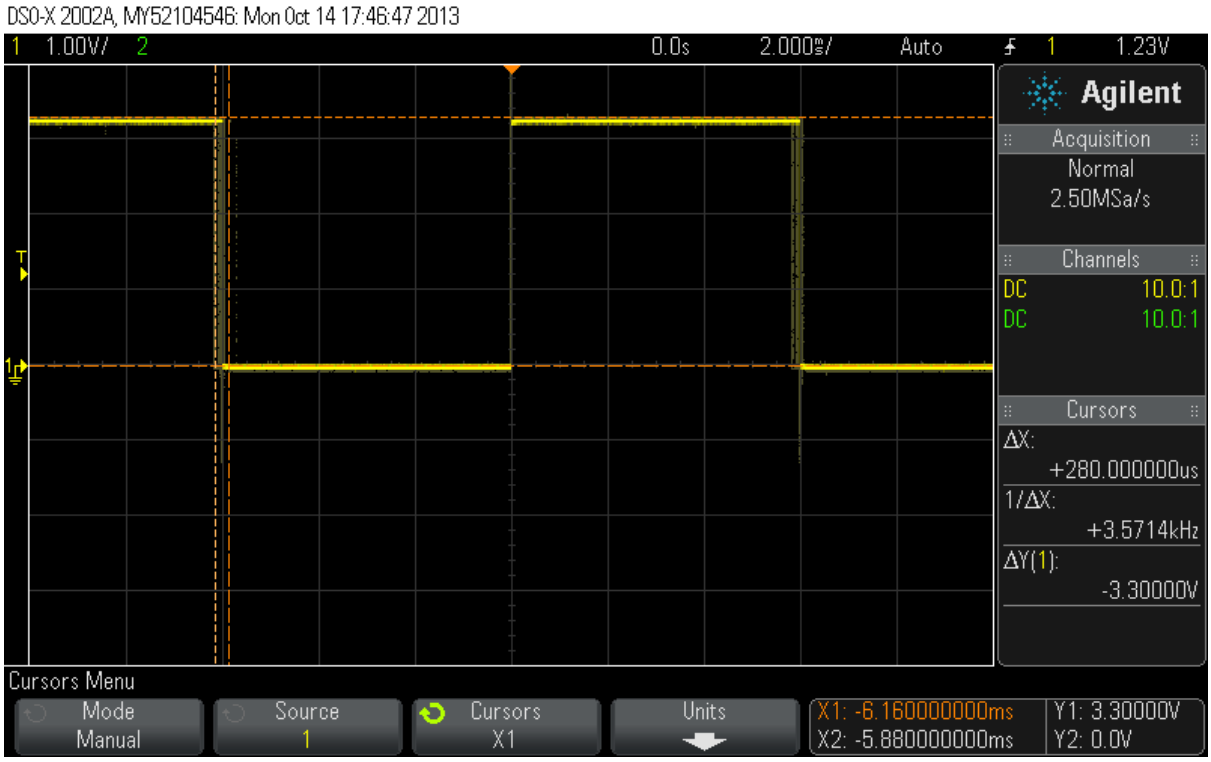

Figure 8.6: Scope capture for the generated pulse train from the Beaglebone Black.

Here, the true worst case jitter of the Beaglebone Black is greatly superior. It sits at 280μS compared to 1.48mS for the Beagleboard-xM as pictured in figure 8.5. Furthermore, the signal appeared far more stable on the screen - indicating a much improved average jitter too.

Chapter 9

Conclusions

> Hofstadter's Law: *"It always takes longer than you expect, even when you take into account Hofstadter's Law."*
>
> – Douglas Hofstadter

The stated objective of this project was to develop and build a low-cost, real-time hardware in the loop simulator. The degree to which it has succeeded can be judged on the three components that make up the project as a whole: the low-cost, the real-time and the hardware in the loop simulation capabilities.

The first, cost, is easy to evaluate. Combining data from previous sections results in the total cost for the system, as shown in table 9.1.

Component	Price
ICs[1]	R100.92
Passives[2]	R33.12
PCB[3]	R831.68
Beaglebone Black	R545.63
Software [4]	R0.00
Total	R1511.35

Table 9.1: Total cost of the low-cost hardware in the loop simulation system.

[1] See table 4.1 for IC cost breakdown

[2] The majority of the cost of passive components was made up by the screw terminals

[3] While the PCB was R831.68 when produced locally in once off quantity, it is possible to obtain 10x from a Chinese PCB fab for $45 with a 30 day lead time.

[4] All software used was either open source, developed for the thesis, or specifically chosen from existing licensed software at UCT.

The entire system, with cape and Beaglebone Black costs a total of R1511.35. For comparison, the Beagleboard-xM alone is R2026.64[5]. Furthermore, a major component of the system cost - the custom PCB - could have been manufactured overseas at a fraction of a cost. Low cost PCB services such as SeeedStudio or ElectroDragon offer 10 PCBs of the size required for $45, or about R45 per PCB. This is total cost reduction of R785, and would bring the total system cost down to R726 - resulting in a sub-R1000 hardware in the loop simulator.

Next, the real-time ability of the project can be evaluated by drawing comparisons to the original Beagleboard-xM system provided by Mathworks. Here, figures 8.5 and 8.6 clearly show the superior jitter abilities of the Beaglebone Black and custom setup used. The jitter has been measurably lowered by a factor of 5 - a significant improvement on the original system.

Lastly, the hardware in the loop simulation capabilities of the system can be evaluated. These centre around the throughput, ease of use of the system and functionality. As demonstrated by the table 8.7 on page 89, the CPU usage of the Beaglebone Black is superior to that of the standard Beagleboard-xM. This directly translates into an improvement in throughput, as a more complex model can be run with a lesser impact on resources.

In terms of ease of use, the existing Simulink Run on Target Hardware interface has been retained, with additional measures such as the Dynamic DNS hostname mentioned in section 4.4.2 put in place to aid use. The functionality of the system is directly related to both the hardware capabilities and software interfaces.

While the hardware was fully verified for all functionality set forth in the objectives, unfortunately time did not permit all of the Simulink blocks to be implemented, as was detailed in section 7.3. However, both the hardware support and low level driver support is in place, meaning that the only thing missing is the Simulink integration. It is estimated that this work could be completed in about a days worth of work.

In conclusion, while the final blockset is more limited than would have been desired, the project as a whole can be considered a success. Performance was greatly improved over the standard reference implementation and functionality was successfully added all while keeping the cost to a minimum.

[5]Price from RS-components.

Chapter 10

Recommendations

"The best way to predict the future is to implement it."

– Alan Key

While the project was mostly a success, the low hanging fruit that stands out for improvement is the completion of the blockset for Simulink access to the device peripherals. By completing the missing blocks, namely the DAC, quadrature input and PWM input the system would be turned into a truly powerful platform for hardware in the loop simulation.

Another point that could be worked on is the configuration options available. The blocks provided offer the basic configuration options - things such as pull up / pull down resistors are set once at boot time and cannot be configured later.

A PWM capture driver could be written, allowing the use of the built in PWM capture module. This would free up a PRU core for other computation - which leads into the next point of making the PRUs more accessible. Should a block be created which allows access to program the PRU core, the use of it would be eased greatly. Furthermore, ready made PRU blocks could be shared.

It is also physically possible to utilise all of the pins provided on the board as GPIOs. For an application that is GPIO heavy, the ability to remap pins as GPIOs arbitrarily could be extremely useful.

Lastly, while the current implementation takes advantage of the increase in Linux timer stability from Xenomai, it does not fully utilise the Xenomai stack. Modifying Simulink's base TLC files to produce fully Xenomai laden could could offer a further improvement in jitter.

Bibliography

[1] M. Samek, 'Building bare-metal arm systems with gnu', *Embedded. com*, 2007.

[2] R. Barry, *Real time application design using freertos in small embedded systems*, 2003.

[3] K. Yaghmour, 'Adaptive domain environment for operating systems', *Opersys inc*, 2001.

[4] R. Barry, *Using the FreeRTOS real time kernel: a practical guide*. Real Time Engineers Limited, 2009.

[5] M. Franke, 'A quantitative comparison of realtime linux solutions', *Chemnitz University of Technology*, 2007.

[6] K. Koolwal, 'Myths and realities of real-time linux software systems', in *Proceedings of the 11th Linux Symposium-Dresden, Germany*, 2009.

[7] J. H. Brown and B. Martin, 'How fast is fast enough? choosing between xenomai and linux for real-time applications', *Rep Invariant Systems*, 2010.

[8] R. Isermann, J. Schaffnit and S. Sinsel, 'Hardware-in-the-loop simulation for the design and testing of engine-control systems', *Control Engineering Practice*, vol. 7, no. 5, pp. 643–653, 1999.

[9] B. Lu, X. Wu, H. Figueroa and A. Monti, 'A low-cost real-time hardware-in-the-loop testing approach of power electronics controls', *Industrial Electronics, IEEE Transactions on*, vol. 54, no. 2, pp. 919–931, 2007.

[10] R. W. Allen, T. J. Rosenthal, B. L. Aponso, D. H. Klyde, F. G. Anderson and J. Chrstos, 'A low cost pc based driving simulator for prototyping and hardware-in-the-loop applications', *SAE Paper*, vol. 980222, pp. 23–26, 1998.

[11] G. Brown, 'Discovering the stm32 microcontroller', *Cortex*, vol. 3, p. 34, 2012.

[12] ST. Electronics. (2012). Arm cortex-m4 32b mcu+fpu, [Online]. Available: http://www.st.com/st-web-ui/static/active/en/resource/technical/document/user_manual/DM00039084.pdf (visited on 11th Sep. 2013).

[13] ——, (2012). Stm32f4discovery stm32f4 high-performance discovery board, [Online]. Available: http://www.st.com/web/en/resource/technical/document/datasheet/DM00035129.pdf (visited on 11th Sep. 2013).

[14] TKJ. Electronics. (2013). Stm32f4 discovery board, [Online]. Available: http://blog.tkjelectronics.dk/wp-content/uploads/STM32F4-DISCOVERY_Board.jpg (visited on 1st Oct. 2013).

[15] Broadcom. (2013). Raspberry pi manual, [Online]. Available: http://www.raspberrypi.org/wp-content/uploads/2012/02/BCM2835-ARM-Peripherals.pdf (visited on 4th Oct. 2013).

[16] MCM. Electronics. (2013). Beaglebone black, [Online]. Available: http://blog.mcmelectronics.com/image.axd?picture=/2013/05/BeaglePiUno_Comparison.jpg (visited on 1st Oct. 2013).

[17] A. Holdings, 'Arm1176jzf-s technical reference manual', *Revision r0p7*, 2009.

[18] M. Baron, 'Cortex-a8: high speed, low power', *Microprocessor Report*, vol. 11, no. 14, pp. 1–6, 2005.

[19] G. Coley, 'Beagleboard-xm system reference manual', *Revision A2. Beagle-Board. org*, 2010.

[20] Wikimedia. Commons. (2013). Beagleboard-xm, [Online]. Available: http://upload.wikimedia.org/wikipedia/commons/8/8e/BeagleBoard_xM.JPG (visited on 1st Oct. 2013).

[21] G. Coley, *Beaglebone black rev a5c system reference manual*, 2013.

[22] J. Cooper. (2013). Device tree background, [Online]. Available: http://learn.adafruit.com/introduction-to-the-beaglebone-black-device-tree/device-tree-background (visited on 2nd Oct. 2013).

[23] Analog. Devices. (2013). Low voltage, 1.15v to 5.5 v, 4-channel, bidirectional logic level translator, [Online]. Available: http://www.analog.com/static/imported-files/data_sheets/ADG3304.pdf (visited on 1st Oct. 2013).

[24] M. S. Sharawi, 'Practical issues in high speed pcb design', *Potentials, IEEE*, vol. 23, no. 2, pp. 24–27, 2004.

[25] Maxim. Integrated. (2003). 3.0v to 5.5v low power rs-232 transceiver, [Online]. Available: http://datasheets.maximintegrated.com/en/ds/MAX3222-MAX3241.pdf (visited on 4th Oct. 2013).

[26] ——, (2003). Low-power, slew-rate-limited rs-485/rs-422 transceivers, [Online]. Available: http://www.usconverters.com/downloads/max491.pdf (visited on 4th Oct. 2013).

[27] ——, (2012). +3.3v, 1mbps, low-supply-current can transceiver, [Online]. Available: http://datasheets.maximintegrated.com/en/ds/MAX3051.pdf (visited on 4th Oct. 2013).

[28] Microchip. (2007). 12-bit digital-to-analog converter with eeprom memory in sot-23-6, [Online]. Available: https://www.sparkfun.com/datasheets/BreakoutBoards/MCP4725.pdf (visited on 4th Oct. 2013).

[29] ——, (2005). 256k i2c cmos serial eeprom, [Online]. Available: http://ww1.microchip.com/downloads/en/devicedoc/21203n.pdf (visited on 4th Oct. 2013).

[30] N. Lewis. (2013). Driver and api for the enhanced quadrature encoder pulse (eqep) decoder found on the am33xx series ti socs, [Online]. Available: https://github.com/Teknoman117/beaglebot/tree/master/encoders (visited on 2nd Oct. 2013).

[31] Mathworks. (2013). Why does the sdk 7.1 installation fail with an "installation failed" message on my windows system?, [Online]. Available: http://www.mathworks.com/support/solutions/en/data/1-MOAV25/index.html?solution=1-MOAV25 (visited on 4th Oct. 2013).

Appendix A

Models & Code

For the code used, please see the attached CD.