NTNU – Trondheim
Norwegian University of
Science and Technology

Development of an Embedded Test Platform

A Real-Time Programmers Perspective

Henrik Finnland Foss

Master of Science in Cybernetics and Robotics
Submission date: June 2014
Supervisor: Amund Skavhaug, ITK

Norwegian University of Science and Technology
Department of Engineering Cybernetics

MASTER THESIS

Development of an Embedded Test Platform

A Real-Time Programmers Perspective

Author:
Henrik Finnland Foss

Supervisor:
Amund Skavhaug

June 9, 2014

Preface

This thesis is a report of my master thesis project in Engineering Cybernetics at the Norwegian University of Science and Technology (NTNU). It is a study into different topics concerning embedded and real-time systems. The goals were twofold, one part concerned with hardware and creating a platform for testing such topics, and the other studying real-time behaviour of a complex operating system patched to be fully preemptible.

The project started in January 2014, and has presented me with the opportunity to work with both familiar and unfamiliar subjects. The project was specified as a set of goals instead of a tight problem description. This meant that I was able to shape my own path and could thus investigate deeper into topics which presented themselves as challenging.

I would like to give a special thanks to my supervisor, Amund Skavhaug, for excellent guidance.

Henrik Finnland Foss
Trondheim, June 9, 2014

Summary

As the embedded world grows day by day, more complex operations are carried out in this environment. The embedded market is no longer dominated by the 8-bit segment, and future embedded and real-time programmers will thus need a wider range of knowledge and skills. This project is a study into several aspects of embedded real-time systems. The first goal was to create a versatile test platform for future programmers to learn real-time and embedded programming principles. The second goal was to study GNU/Linux in an embedded real-time environment. The purpose of the first goal was to create an improved alternative to the course assignment in TTK4147, Real-Time Systems, where today's exercises are either outdated or run on virtual machines. The purpose of the second goal was to investigate the potential for a complex operating system to offer hard real-time characteristics.

This project has created an embedded platform for testing embedded and real-time principles. The platform consists of a single-board computer together with a stackable cape by own design, containing a 8-bit microcontroller circuit. The cape, called the External Response Tester, was programmed to perform testing on the responsiveness of the connected hardware, and it was shown that it did so with precision. We also expanded on the operating systems catalogue for the single-board computer as we ported a fully preemptible Linux kernel, as well as the real-time operating system FreeRTOS to work on the board. This complete educational platform was found to be capable of unifying and replacing most of the TTK4147 course assignments.

The operating system GNU/Linux, more importantly the fully preemptible PREEMPT_RT version, was evaluated in terms of usage in hard real-time systems. An analysis that compared results from the ordinary Linux kernel, the PREEMPT_RT patched Linux kernel, and FreeRTOS was carried out. We concluded that the fully preemptible Linux version is not "definitely unsuitable", and its applicability depends on the requirements of a given real-time application.

Sammendrag

Innebygde datasystemer finnes i dag i mange ulike applikasjoner, og antallet øker sterkt. Videre utfører slike datasystemer stadig mer komplekse operasjoner. Tidligere var dette markedet dominert av 8-bit mikrokontrollere, i dag benyttes stadig mer komplekse prosessorer. Dette fører til at utviklere av innebygde datasystemer og sanntidssystemer trenger et bredere spekter av kunnskap og ferdigheter. Det første målet med denne oppgaven var å lage en allsidig og moderne test-plattform som programmerere kan bruke for å lære seg viktige prinsipper innenfor fagområdene. Det andre målet var å studere operativsystemet GNU/Linux for bruk i innebygde sanntidsapplikasjoner. Bakgrunnen for at vi satte oss det første målet var at dagens øvingsopplegg i TTK4147 Sanntidssystemer, er enten utdatert eller kjøres på virtuelle maskiner. En ny og moderne plattform ville kunne forbedre store deler av dette øvingsopplegget. Bakgrunnen for at vi satte oss det andre målet var at vi ville undersøke potensialet for å ha et komplekst operativsystem som GNU/Linux til å levere harde sanntidsegenskaper.

Dette prosjektet har utviklet en plattform som muliggjør testing av prinsipper fra innebygde datasystemer og sanntidssystemer. Plattformen består av en enkelt-brett datamaskin sammen med en 8-bit mikrokontroller-krets designet på et PCB brett. Denne mikrokontroller-kretsen har muligheten til å stables oppå enkelt-brett datamaskinen, eller fungere for seg selv. Den er programmert til å utføre tester som undersøker reaksjonsevnen til programvare kjørende på ekstern maskinvare. Dette viste vi at den utførte med god nøyaktighet. Vi utvidet også operativsystem katalogen for enkelt-brett datamaskinen med en fullt avbrytbar GNU/Linux versjon, og implementerte sanntids-operativsystemet FreeRTOS. Vi viste at denne komplette plattformen hadde alle egenskapene som skal til for å samle øvingsopplegget i TTK4147 til og utføres på en enkelt plattform.

Den fullt avbrytbare versjonen av GNU/Linux ble evaluert i forhold til sine sanntids-karakteristikker. En analyse sammenlignet resultater fra den ordinære GNU/Linux versjon, den full avbrytbare versjonen og FreeRTOS. Vi konkluderte med at den avbrytbare versjonen av Linux har et potensiale til å benyttes i sanntidsapplikasjonener, men det avhenger av applikasjonen.

Contents

Preface i
Summary ii
Sammendrag iii

1 Introduction **1**
1.1 Background 1
1.2 Project goals 2
1.3 Main contributions 2
1.4 Report layout 3
1.5 Literature and Related Work 3

2 Real-time Systems **5**
2.1 Interrupts 5
2.2 Latency 8
2.2.1 Worst case latency and boundedness 9
2.2.2 Latency vs throughput 9
2.3 Scheduling 9
2.3.1 Priorities 10

3 Hardware **15**
3.1 ARM 15
3.1.1 ARM vs x86 16
3.1.2 Other architectures 16
3.2 BeagleBone Black 16
3.2.1 The AM3358 processor 18
3.2.1.1 The exception vector 18
3.2.2 Programmable Realtime Unit 18

3.2.3 The AM3358 Pin Multiplexer (Pinmux) . . . 19
3.3 Atmel 8-bit AVR . . . 20
3.3.1 Modified Harvard Architecture . . . 20
3.3.2 Atmel Studio and the JTAG ICE . . . 21
3.4 Saleae Logic . . . 22

4 Operating Systems **23**
4.1 GNU/Linux . . . 23
4.1.1 Naming . . . 24
4.1.2 Kernel Architecture . . . 24
4.1.3 Kernel and User space . . . 25
4.1.4 Microkernel vs Monolithic kernel . . . 26
4.1.5 Linux Standard Base . . . 27
4.1.6 Preemption and real-time viability . . . 28
4.1.7 Scheduling . . . 28
4.1.7.1 Priorities . . . 28
4.1.8 Device Tree . . . 29
4.1.9 General-purpose input/output Control . . . 29
4.1.9.1 Sysfs . . . 30
4.1.9.2 Memory mapping . . . 31
4.1.9.3 Kernel module . . . 31
4.1.9.4 Polling . . . 31
4.1.10 Distributions . . . 32
4.2 QNX . . . 34
4.2.1 Neutrino microkernel . . . 34
4.2.2 Threads and processes . . . 35
4.2.3 Thread lifecycle . . . 35
4.2.4 Scheduler . . . 36
4.3 FreeRTOS . . . 38
4.3.1 Tasks in FreeRTOS . . . 38
4.3.2 Lifecycle . . . 38
4.3.3 FreeRTOS scheduler . . . 38

5 Hardware solution **41**
5.1 Proposed solution . . . 41
5.2 External Response Tester . . . 42
5.2.1 AVR butterfly . . . 42
5.2.2 BeagleBone Cape . . . 42
5.2.3 Prototyping . . . 43
5.2.3.1 Schematics . . . 45
5.2.4 PCB design . . . 48
5.2.4.1 Altium designer . . . 48
5.2.4.2 Manufacturing files . . . 48
5.2.4.3 Final pinout description . . . 49
5.3 ERT application . . . 51
5.3.1 How to use . . . 51
5.3.2 Code explanation . . . 53
5.3.2.1 Defines and globals . . . 53
5.3.2.2 UART setup . . . 53
5.3.2.3 Timer 1 and ICP setup . . . 54
5.3.2.4 Timer 0 setup . . . 54
5.3.2.5 Input/Output setup . . . 54
5.3.2.6 The main loop . . . 55
5.4 Discussion on the creation of the External Response Tester cape . . . 55

6 GNU/Linux on the BeagleBone **57**
6.1 Boot process . . . 57
6.1.1 Bootloader . . . 57
6.1.2 SD card . . . 58
6.2 Debian GNU/Linux . . . 59
6.3 Compiling the Linux kernel . . . 59
6.3.1 Configure pinmux driver . . . 61
6.3.2 Device Tree setup . . . 62
6.3.3 General configuration settings . . . 65
6.3.4 Further optimizations . . . 66
6.4 Developing on the BeagleBone Black . . . 67
6.4.1 Cross-compilation with Eclipse . . . 67

6.4.2 BeagleBone pinout for interracting with the ERT cape 68
6.4.3 GPIO test application 68
6.4.3.1 The response test 68
6.4.3.2 The maximum single pin toggle test 69
6.4.3.3 The code 70
6.4.4 Developing for the PRU subsystem 71
6.4.4.1 Starting and loading programs to the PRU 71
6.4.4.2 The PRU single pin maximum toggle test 72
6.4.4.3 PRU busy-wait test 72
6.5 Benchmarking tools 72
6.5.1 Cyclictest 73
6.5.2 Stress 73
6.5.3 Hackbench 73
6.6 Discussion on the implementation of GNU/Linux 73

7 FreeRTOS and additional OS support **75**
7.1 QNX 75
7.2 FreeRTOS 75
7.2.1 Porting FreeRTOS to AM335x 76
7.2.1.1 Configuring the tick interrupt 76
7.2.1.2 ARM exception vector 77
7.3 FreeRTOS test application 78
7.3.1 GPIO interrupt 78
7.3.2 Initialization of output GPIO and UART0 79
7.3.3 Tasks 80
7.4 Discussion on the implementation of additional OS support 80

8 Test setup and results **83**
8.1 The maximum single pin toggle test 83
8.1.1 Setup 83
8.1.2 Testing 83
8.1.3 Results 84
8.2 Cyclictest 86
8.2.1 Setup 86
8.2.2 Testing 86

8.2.3 Results . . . 87
8.3 Cape response testing . . . 92
8.3.1 Setup . . . 92
8.3.2 Testing . . . 92
8.3.3 Results . . . 93
8.3.3.1 Basic preemption Linux kernel . . . 93
8.3.3.2 PREEMPT_RT patched Linux kernel . . . 95
8.3.3.3 FreeRTOS . . . 97
8.3.3.4 PRU . . . 98

9 Discussion **99**
9.1 I/O latency in GNU/Linux . . . 99
9.2 System latency in GNU/Linux, PREEMPT_RT vs basic low-latency preemptable kernel . . . 100
9.3 Working with the BeagleBone Black and ERT cape . . . 102

10 Conclusions and recommendations **105**
10.1 Conclusions . . . 105
10.2 Recommendations for future work . . . 105

A Acronyms **107**

B Appendix **109**
B.1 Appended external storage . . . 109
B.2 How to boot the different OS on BeagleBone Black . . . 110
B.2.1 Booting Linux . . . 110
B.2.1.1 Swapping the kernel . . . 110
B.2.1.2 Log in . . . 110
B.2.2 Booting QNX and FreeRTOS . . . 110
B.2.2.1 QNX . . . 111
B.2.2.2 FreeRTOS . . . 111
B.3 ERT code files . . . 112
B.3.1 defines.h and globals . . . 112
B.3.2 main.c . . . 115
B.3.3 io.c . . . 122

B.3.4 usart.c 124
B.3.5 timer.c 127

Bibliography **129**

1 Introduction

The definition of a real-time system is, "the correctness of the system behavior depends not only on the logical results of the computations, but also on the physical instant at which these results are produced"[1]. This area in computer science originated from a growing usage of computer systems in control applications. Until the 2000s most real-time systems were implemented through homebrew Operating Systems(OS) tailored to each specific problem. Today the popular solution is to use supported commercial or/and open source OS, and thus, reducing lead time in product development. However, the high quantity of both different hardware platforms and OS makes it necessary for designers to have good knowledge of real-time systems. The main purpose of this thesis will be to create an embedded test platform, which future students can use as a basis for their education into embedded and real-time systems.

1.1 Background

TTK4147 Real-Time Systems is a course lectured at NTNU. The purpose of the course is to educate the students in how to analyse and evaluate runtime requirements for embedded real-time systems. There are 11 exercises and one miniproject, which counts towards the final grade in the subject, these are practical tasks which often consists of coding for specific operation systems and platforms. However, several of these platforms are either outdated or run on virtual machines, and will thus, not necessarily represent what the students will experience in the outside the university.

[1] Real-Time Systems[17]

1.2 Project goals

This thesis will create an embedded test platform compatible with several OS. We will through both examples and quantifiable test results present how the platform can be used by students. It will hopefully be as an education to for how to program for modern 32-bit embedded architectures, both in general and in terms of real-time principles. We will also take a closer look into one of the OS that will run on the platform, GNU/Linux, and especially the state of its kernel for use in hard real-time systems.

To achieve our goals the problem will be separated into several parts:

- Draw inspiration from similar work done in the past.
- Acquire and review the necessary background material.
- As far as the time permits, implement a solution.
- Evaluate the result and the choices made throughout the process.

1.3 Main contributions

The main contributions of this thesis are:

- The first openly available fully functional port of FreeRTOS to the BeagleBone Black single-board computer[2].
- A fully functional real-time patched Linux kernel version 3.14 for the BeagleBone Black.
- Created a new basis for the exercises in the course TTK4147 Real-Time Systems, for the Department of Engineering Cybernetics at NTNU.

[2] Github repository: https://github.com/henfos/BBBFreeRTOS

1.4 Report layout

This report contains 10 chapters, including this introductory chapter, acronyms and an appendix.
Chapters 2, 3 and 4 are chapters which will focus on theory. First, we will present real-time system topics, followed by theory concerning the hardware we chose to work with, before an OS chapter will explain topics specific to each OS, which are relevant to the software implementation. Chapter 5, 6 and 7 will then focus on the practical implementation carried out in this thesis. The hardware solution with the creation of the External Response Tester is shown first. Then we will show how GNU/Linux was set up and used, before we dedicate a chapter to showing the setup and usage of additional supported OS. Each of these chapters will end with a section discussing the implementation. Chapter 8 contains all the quantifiable results obtained during this thesis. The chapter also includes a description of how each experiment was set up, its testing process and results. We finish with a discussion of the results in chapter 9, which is followed by conclusions and recommendations in chapter 10.

The thesis is written with the expectation that the reader has basic knowledge of operating systems, programming and hardware. Thus, not all terms and topics will be fully elaborated and the reader is therefore encouraged to look up any unknowns. All figures are, unless otherwise noted, made by the author.

All the resources referenced in the thesis are located in the appended CD/USB drive, as well as a PDF version of the thesis. The structure of this appended drive can be found in Appendix B.1.

1.5 Literature and Related Work

There has been no similar work done in the past at this campus, which we have used. Most of the theory on real-time systems topics was gathered from the TTK4147, Real-Time Systems, course material [27], [2]. Apart from these books information was gathered through various Internet sources and communities.

2 Real-time Systems

The introductory chapter stated that the correctness of a real-time system not only on the logical results of the computations, but also on the physical instant at which these results are produced. The study of real-time systems is important because many time critical operations has an inherent need for guaranteed performance. Examples of such operations include air traffic control systems and car safety features. These systems have computational deadlines that must be met, regardless of system load. Hence, predictability is the most important feature in these systems. When we are talking about real-time systems there are two different versions, soft and hard. Soft real-time systems are those which continue even if deadlines are not met, however the system's quality of service is degraded. Hard real-time are those systems where a failure to meet deadlines leads to catastrophic system failures. This thesis will have a focus on the latter systems, and real-time will carry the meaning of hard real-time unless otherwise specified. This chapter will present important topics concerning real-time systems, which will be relevant later on in the thesis. Topics that are presented in this chapter are for the most part covered by the curriculum of TTK4147, [27] and [2].

2.1 Interrupts

Creating a device that fulfills some real-time requirements would be easy if it never relied on any external information. Then all operations would be completely deterministic, and thus it would be straightforward to guarantee performance. However, most systems will work with one or more external resources, and this fact complicates matters significantly. There are several methods for communicating with external sources, a processor can for instance continuously poll for information.

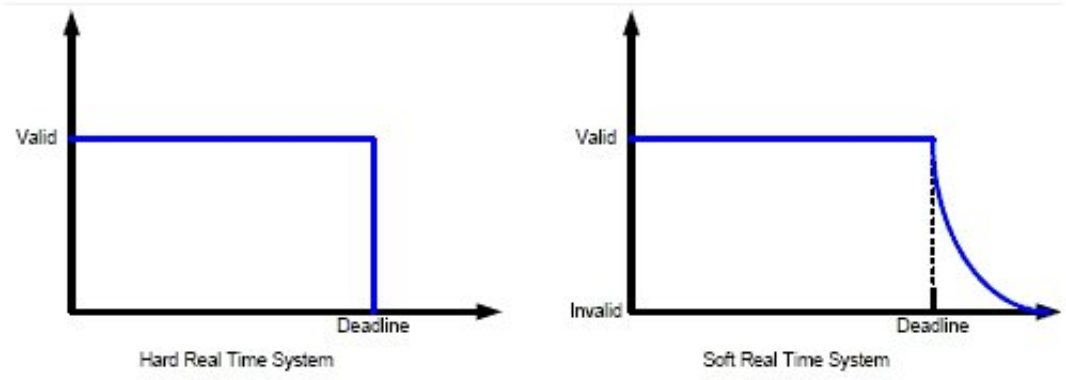

Figure 2.1: Validity of system depending on response time in question to its deadline.

Polling is the process of continuously checking the external resources for some state change. However, this means that the processor will not be able to do anything else as it might miss an event. Instead of polling, we will therefore for the most part use interrupts.

An interrupt is a signal (emitted by hardware or software), which is handled by the processor in a different way than an ordinary signal. Its purpose is to reduce wasting valuable time polling for a resource. Interrupts enables the processor to perform other tasks until an interrupt occurs. It is able to free the processor because it has the ability to break off normal code execution to run some specialized handler code. These specialized handlers are called Interrupt Service Routines (ISR). An example of is when waiting for some data on a bus. Instead of polling continuously to check if there is some new data, we will add an extra line on the bus, which the external resource will trigger when it posts data on the bus. This enables us, by connecting this line to a processors interrupt controller, to use all the processing power for whatever purpose, because at the instance the interrupt line triggers, the code will break normal execution to run a handler, which receives the data. The break from normal code execution will create some overhead, however, this technique does by far still outweigh polling techniques. The programmer will ,however, have to be aware of high interrupt loads, as this can cause serious performance issues.

The use of interrupts does, as stated above, free the processor to perform other operations while waiting for some external device. However, it will affect the real-time capabilities based on the number of interrupts sources, as well as the

already stated context switching overhead and the size of the interrupt handler. This makes it difficult for complex systems to meet the requirement of guaranteed performance. For example, it may be impossible for a complex operating system to specify any worst case latencies.

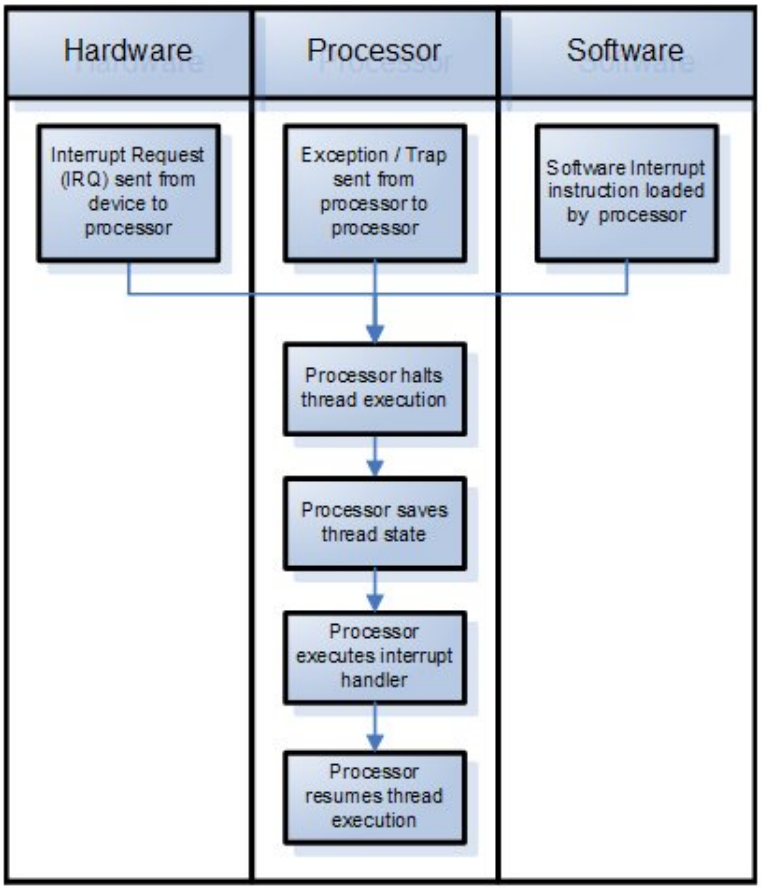

Figure 2.2: Processor handling of different interrupt sources. Courtesy of [1]

[1] http://en.wikipedia.org/wiki/File:Interrupt_Process.PNG

2.2 Latency

Latency is the time interval between a request is made, until it is serviced.

Interrupt latency

Interrupt latency is the time from when an interrupt is generated to when the source of the interrupt is serviced. The minimum interrupt latency depends on the hardware in use, more importantly the interrupt controller circuit. The maximum interrupt latency depends on software, and in a non-RTOS it can be difficult to gain any knowledge on this time interval. There is also the factor that most processors will allow interrupts to be turned off, this is to protect certain critical code sections. However, with the use of a RTOS we can guarantee that the interrupt latency will never exceed a predefined maximum.

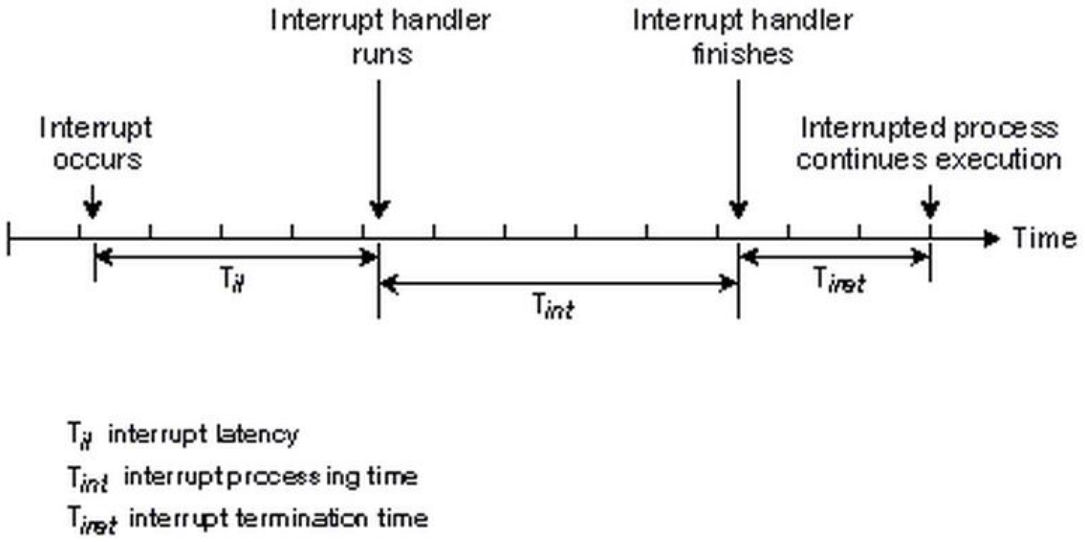

Figure 2.3: Interrupt latency is only one of the factors in the cost of handling an interrupt. Courtesy of [1]

The interrupt latency (T_n in Figure 2.3) and the interrupt termination time (T_{iret} in Figure 2.3) are the two factors which are determined by hardware and the OS. The interrupt processing time (T_{int} in Figure 2.3) is the time it takes to run the code in the interrupt service routine (ISR). This code will often set some flag such that the

[1] http://www.qnx.com/developers/docs/qnx_4.25_docs/qnx4/sysarch/microkernel.html

operations requested by the interrupt can be done in normal code execution. Other times, however, this code can be rather heavy, and thus, significantly increase the overall interrupt cost.

I/O latency

I/O latency is the full time from a process asks to do some I/O operation, until it is actually conducted. When barebone programming on 8-bit microcontrollers (MCUs), like the Atmel AVR, the programmer will mostly have direct control over I/O. Thus, there will only be bound, deterministic, hardware latency. However, when introducing a complex OS like GNU/Linux then I/O control can be hidden through several layers, thus, these systems are prone to more significant I/O latency.

2.2.1 Worst case latency and boundedness

The total worst case latency for a task is a key property when evaluating real-time characteristics of a system. Further, it is important that if the worst case latency can not be mathematically proven, that the empirical results can indicate an upper bound on latency.

2.2.2 Latency vs throughput

Throughput is the amount of data transferred or processed in a specified amount of time. Latency and throughput are two terms that are often confused and sometimes used interchangeably. The reality is that low latency often comes at the cost of less throughput. Low latency requires the OS to frequently break normal code execution to check if there is a request, from either software or hardware, that needs to handled. Breaking from normal code execution creates overhead, and this overhead subtracts of the systems total throughput. To clarify, a responsive system is not the same as a high performance system.

2.3 Scheduling

In the event of one single task with given processing time, it is straightforward to evaluate if it meets real-time requirements. However, often we have several

tasks running concurrently. The processor then has to context switch between the differents tasks. Scheduling is the method by which a task is given access to system resources. The choice of scheduler is both an important and difficult one. For example, a advanced scheduler may enable a lot different lower priority tasks to run, and still keep requirements for the higher priority tasks. This may however cause larger overhead, which reduces system throughput capabilities.
Figure 2.4 shows the different categories of schedulers. This thesis is as earlier stated focused on hard real-time problems. Static schedulers are schedulers with priorities which are fixed pre-run time, while dynamic has the ability of changing the priorities of each task during run-time. Non-preemptive (or co-operative) schedulers are schedulers where once a task is given processing time, it will run until itself decides to yield, or uses its time-slice. Preemptive schedulers, on the other hand, allow a task to force a context switch whenever it is ready to run, and there is a lower priority task which currently is running.

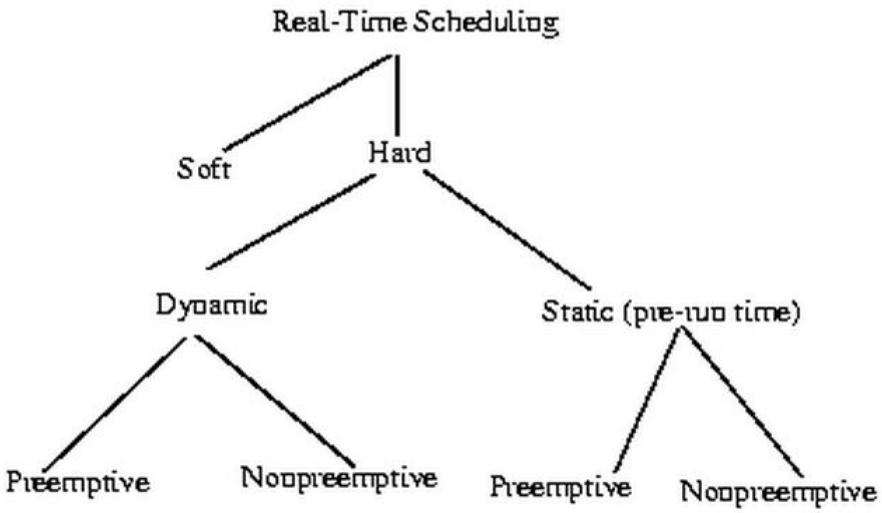

Figure 2.4: Different groups of schedulers. Courtesy of [1]

2.3.1 Priorities

Real-time systems these have multiple threads or tasks. Some may perform display and housekeeping operations, others are tasked with crucial control operations. With the use of priorities we can make sure that the lesser important operations do not

interfere with the crucial ones. Priorities are often a simple number which tells the scheduler which position in the ready queue it should take, se Figure 2.5. They can also be static or dynamic. Static priorities are set pre-runtime and can not be changed. Dynamic priorities however, give the system the possibility to change the priority in runtime, which often can be very useful.

[1] http://users.ece.cmu.edu/~koopman/des_s99/real_time/
[2] http://www.qnx.com/developers/docs/6.3.0SP3/neutrino/prog/images/readyq.jpg

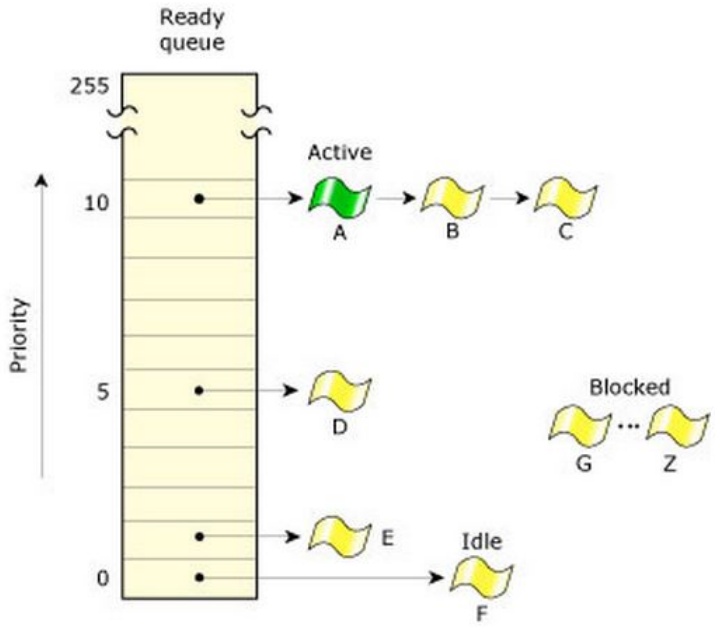

Figure 2.5: Tasks waiting in different ready queues depending on priority, while other tasks are blocked. Courtesy of [2].

Priority inversion

Priority inversion is a scenario where a high priority task is not able to run due to involvement by a lower priority task. Take the scenario with three threads, one with high (H), medium (M), and low (L) priority. There exists also a resource (R) used by the H and L threads. Figure 2.6 shows a scenario where thread H is not able to run due to thread L controlling the needed resource R. Further, since there is a thread M running, thread L is not able to finish its operations with R. This makes it impossible to say when thread H will be able to run again, something that should not happen since H has the highest priority. This is an example of priority inversion.

This is a classic problematic scenario in real-time system, and there are several ways to solve this problem. However, there have been several cases where this have been forgotten and lead to serious errors through starvation of higher priority tasks. Most famous is the Mars lander "Mars Pathfinder" incident [1]. When the rover started to gather meteorological data it did so through a low priority task, its data was then published on an information bus. Another high priority task, which ran bus management also frequently accessed this information bus. The low priority task

[1] What really happened on Mars? [16]

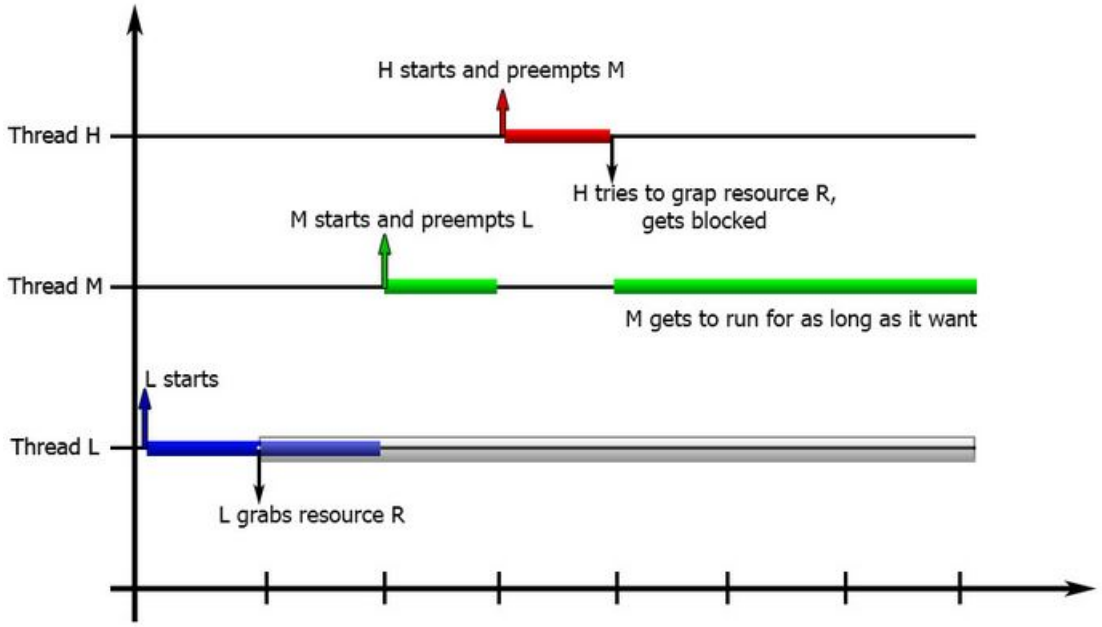

Figure 2.6: Example of priority inversion.

would then control the bus resource like in Figure 2.6, and a medium priority task would preempt it with the consequence of keeping the high priority task blocked when it needed the bus. A watchdog timer would then go off, which caused a system wide reset. The rover ran a version of VxWorks, a RTOS which is created by Wind River. A VxWorks mutex object, like the one used to lock access to the information bus, accepts a boolean parameter that indicates whether priority inheritance should be performed by the mutex. Priority inheritance causes the low priority task to temporarily be assigned the priority of the highest waiting priority task while holding the resource. Thus, the medium priority task can not preempt and prevent the low priority task from finishing. Once the low priority task releases the resource it returns to its original low priority and the high priority task waiting preempts it, see Figure 2.7. By uploading a small program, which changed this boolean parameter of the information bus mutex, no further system resets occurred.
There are several other solutions to this problem, but the important thing is to know about the scenario since there is no foolproof method to predict the situation.

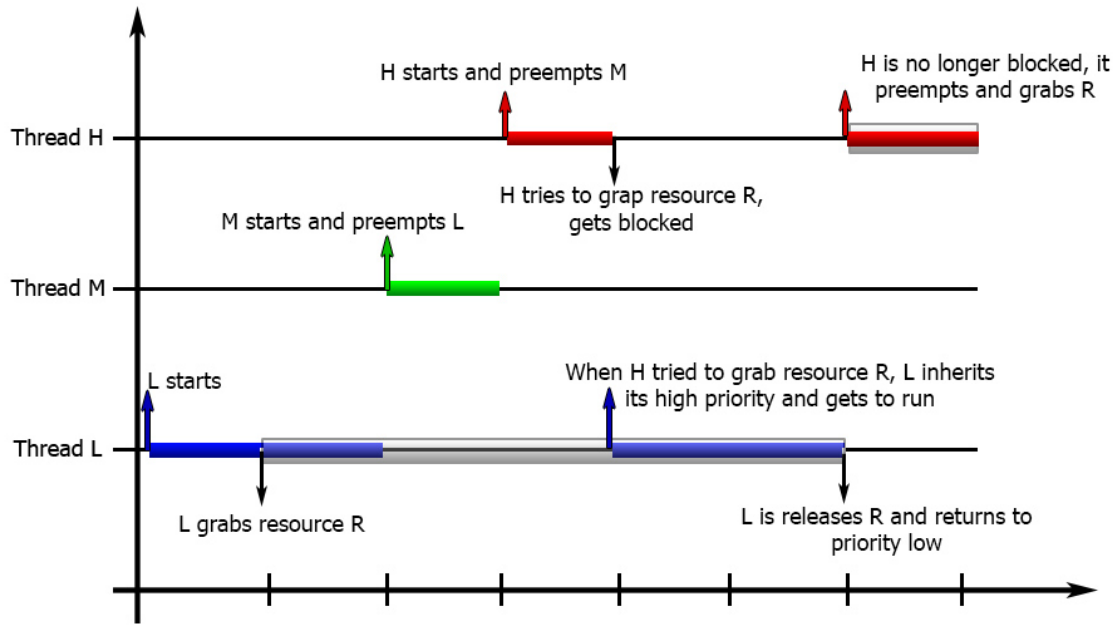

Figure 2.7: Solution to problem by using priority inheritance.

3 | Hardware

The main criteria for choosing the hardware platform was that it would be both modern, and versatile. This was based on problem formulation 1.2, which states that the test platform would be used in education for learning embedded and real-time principle. Thus, it was important for the architecture to support multiple OS, such that all the OS already used in the course TTK4147 exercise program, would be supported by this new platform. Thus the choice fell, on using the popular ARM architecture. This basis would be complemented by a preprogrammed external unit, this would be used as a benchmarking and analysis tool. This interacting unit was chosen to be built around an 8-bit Atmel AVR MCU, due to previous experience with such units.
This chapter will present the different architectures and hardware, which are relevant for this thesis.

3.1 ARM

The ARM architecture was first introduced in the mid 1980s. In 1985, VLSI produced the first ARM silicon, known as ARM1. ARM is an IP company, which does no manufacturing. Their primary business is selling IP cores, in particular chip layout designs. The licensees then base their MCUs and CPUs on these ARM cores. ARM designs are based on reduced instruction set computing (RISC) architecture, as opposed to a complex instruction set computing (CISC) architecture. The difference will be presented in the next subchapter. The ARM architecture had a 60% market share, as of 2012 in the embedded processor market[1], a number which is expected to grow to 68% in 2016. ARM offers a powerful ecosystem, where their cores range from

[1] ICD: ARM market share [12]

costly high-end 64-bit cores to cheap low-powered cores. These low-powered cores have in later years been steadily attracting more traditionally 8/16-bit projects over to ARM. They also offer a full range of development tools and software.

3.1.1 ARM vs x86

The best known computer architecture today is the x86, the x86 has since 2006 been the de facto processors in desktop and laptop computers. In 2006 Apple moved, as the last of the large personal computer companies, from PowerPC to the x86 architecture. The x86 is in contrary to ARM based on CISC architecture. CISC enables a single instruction to execute several low-level operations. CISC attempts to minimize the number of instructions per program, while RISC reduces cycles per instruction at the cost of the number of instructions per program. The x86 has historically had greater emphasis on speed and performance than power consumption, and therefore not been a serious contender in the embedded market. The higher power consumption comes in part from the large overhead needed to maintain the large ROMs needed for CISC. However, Intel is making strides to gain a foothold inside this market. Their latest Atom low power architecture[2] looks to be ARM's first serious contender in the mobile embedded market[3]. This could be the first push to create very low powered x86 processors, ARM does as it stands today, have a much larger ecosystem when it comes to the embedded market.

3.1.2 Other architectures

There are several other architectures available for the embedded market, including the Atmel AVR, the Microchip PIC, the Texas Instrument MSP430, and several others. Most of these are 8 or 16-bit specialists, and their 32/64-bit architectures are not as popular as their less advanced counterparts.

3.2 BeagleBone Black

We needed a versatile base to enable the running of several OS on the same testing solution. As explained at the start of this Chapter, the ARM architecture was selected

[2] Bay Trail, launched Q3 2013
[3] The Bay Trail tested [26].

to create this platform. The choice of hardware therefore fell on the BeagleBone Black single-board computer, seen in Figure 3.1. This board has a 1GHz ARM Cortex-A8 processor produced by Texas Instruments. It is a popular open-source community supported development platform, which provides a lot of resources to support development. This board features:

- AM3358 1GHz ARM Cortex-A8
- 512MB DDR3 RAM
- 2GB 8-bit eMMC on-board flash storage
- 3D graphics accelerator
- NEON floating-point accelerator
- 2x PRU 32-bit microcontrollers
- USB, Ethernet, HDMI, 2 x 46 pin headers

Figure 3.1: The BeagleBone Black. Courtesy of [1].

[1] http://beagleboard.org/Products/BeagleBone+Black

3.2.1 The AM3358 processor

The BeagleBone Black has a AM3358 Sitara ARM Cortex-A8 processor. The ARM Cortex-A8 is a processor core implementing the ARMv7-A 32-bit instruction set architecture, and it runs at 1GHz. The Cortex-A8 is one of the most widely used cores in mobile devices, and Texas Instruments recommends it as "Ideal for home automation, industrial automation, enterprise/educational tablets, portable navigation devices and networking".

3.2.1.1 The exception vector

ARM microprocessors are able to respond to an interrupt with a context switch, this breaks normal code execution to run some special handler routine. All these interrupts are on ARM processors called exceptions (including hardware reset). When an exception occurs, the processor saves the context and then jumps to a vector table in memory, which contains addresses to where the exceptions should be handled. This is the Exception Vector Table (EVT), and it includes Reset, Data Abort, Prefetch Abort, Undefined Instruction, Normal Interrupt (IRQ), and Software Interrupt (SWI) exceptions.

3.2.2 Programmable Realtime Unit

The MCU chip comes with a second generation Programmable Realtime Unit (PRU) subsystem, which features a dual 32-bit RISC core, 8KB data memory, 8KB instruction memory and 12KB shared RAM, see Figure 3.2. Its instruction set is small and deterministic, which means all instructions are executed in a single cycle, except accessing external memory. This is a subsystem integrated separately from the ARM core, allowing independent operation. The PRU can be set up by the main Cortex core, and the BeagleBone community has written a package which creates PRU support for GNU/Linux systems. The package includes the PRU assembly complier, *pasm*, and its source code. It also provides a userspace driver to load applications. Documentation and example applications are also included. The package can be found on github [4].

[4] Am335x_pru_package, [18]

[1] The PRU reference guide, http://mythopoeic.org/BBB-PRU/am335xPruReferenceGuide.pdf

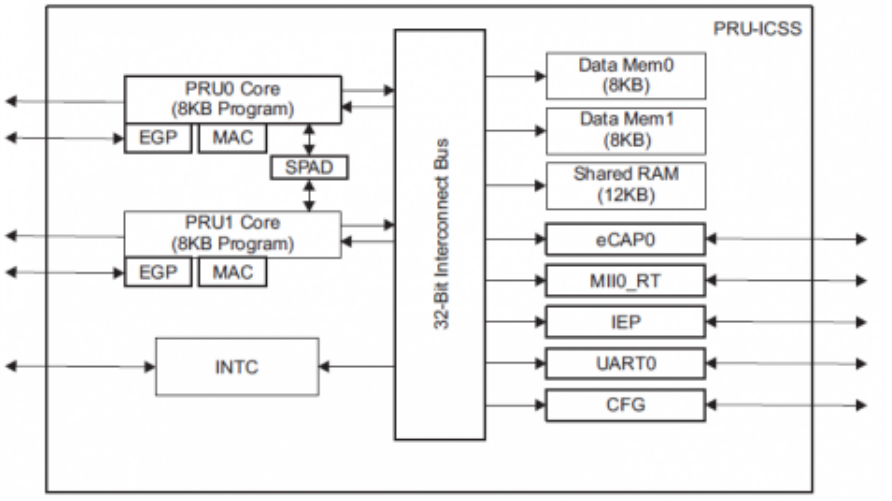

Figure 3.2: The PRUSSv2. Courtesy of [1].

3.2.3 The AM3358 Pin Multiplexer (Pinmux)

The AM3358 chip has fewer pins on its package than the internal logic provides functionality. The chip therefore provides a configurable pin multiplexer, which gives the user a choice to what logic should be available. Each pin can be set to one out of eight modes. Some pins have to be set in some predetermined mode to initialize the board, however, a lot of pins have a range of modes providing the developer with flexibility. Figure 3.3 is included to illustrate the vast number of different modes, which are choosable for the BeagleBone Black headers. The pinmux can be configured at boot, and at normal runtime.

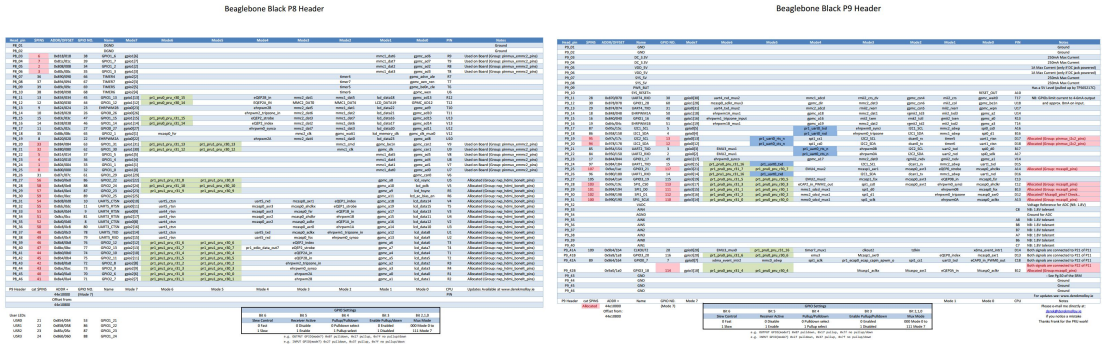

Figure 3.3: The different possible modes for the BeagleBone black header pins. Courtesy of Derek Molloy [1].

3.3 Atmel 8-bit AVR

The Atmel AVR architecture was developed by two students[5] at the Norwegian Institute of Technology (now Norwegian University of Science and Technology - NTNU)[6]. These students further developed the architecture when the technology was acquired by Atmel. Their 8-bit MCUs deliver high speed (1 MIPS/MHz), a large range of pins (6-100), and a large range of peripheral set options. Combined with free and inexpensive development tools the AVR has gained a large user base in the 8-bit MCU market.

3.3.1 Modified Harvard Architecture

The Atmel AVR MCU is a harvard architecture machine[7]. This is one of two main types of digital computer architectures. The characteristic of the Harvard Architecture is that it maintains a distinct separation between code and data spaces. The alternative architecture is the von Neumann architecture, which has shared signals and memory for code and data. The difference between these are that the harvard architecture is able to access memory and data simultaneously, enabling the possibility of completing an instruction in a single cycle. However, the code memory of the harvard architecture is typically read-only memory, which makes it impossible for program contents to be modified by the program itself. This is something easily done by the von Neumann Architecture where all the memory is read-write, as seen in Figure 3.4.

Modified harvard architectures allows for the contents of its instruction memory to be accessed just as if it was data. This enables the architecture to support high performance concurrent data and instruction memory access, while also supporting tasks like loading a program from disk storage as data and then executing it. The AVR adopts such a modified version of the harvard architectures through special instructions.

[1] GPIO Programming on ARM Embedded Linux, http://derekmolloy.ie/beaglebone/beaglebone-gpio-programming-on-arm-embedded-linux/

[5] Alf-Egil Bogen and Vegard Wollan

[6] The Story of AVR [1]

[7] Data in Program Space [21]

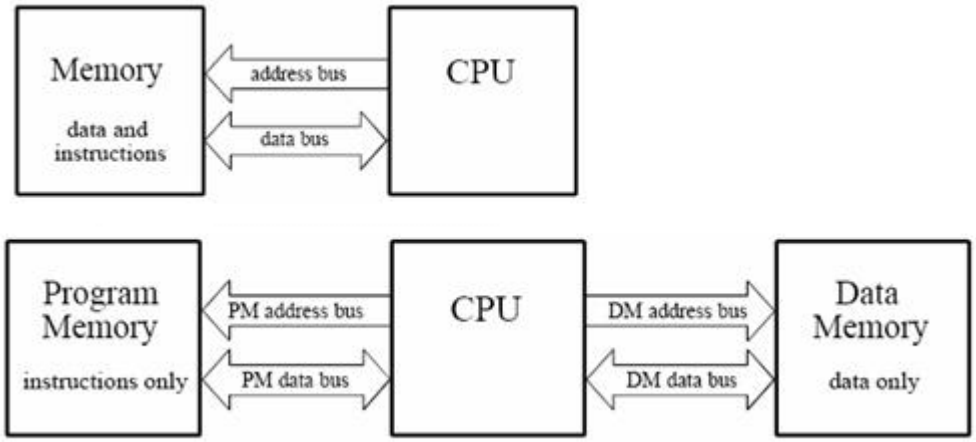

Figure 3.4: The main difference between the Harvard (bottom) and von Neumann (top) arcitecture.

3.3.2 Atmel Studio and the JTAG ICE

Atmel Studio (seen in Figure 3.5) and the JTAG ICE (seen in Figure 3.6 are development tools enabling programming of the Atmel AVR MCUs. The Atmel Studio is a Integrated Development Environment (IDE) created by Atmel for developing and debugging assembly or C/C++ applications for Atmel MCUs. It is free of charge and is integrated with the Atmel Software Framework (ASF). ASF provides a large collection of embedded software to keep developers from reinventing the wheel.

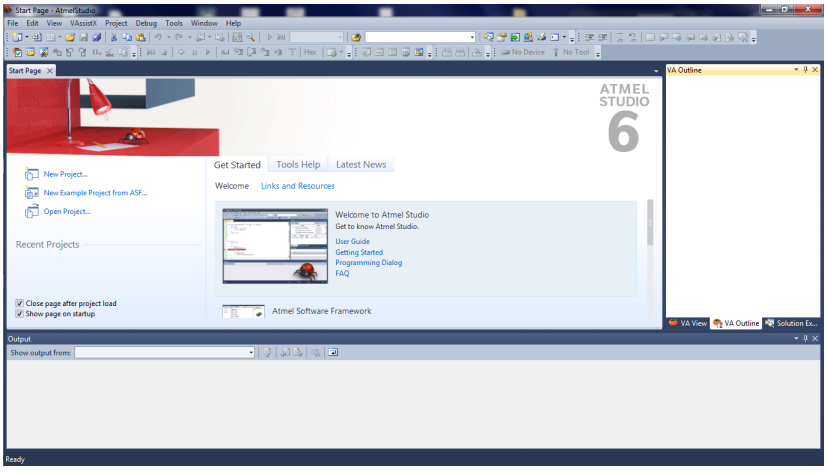

Figure 3.5: The graphical user interface of Atmel Studio.

The JTAGICE3 is a development tool for debugging and programming Atmel MCUs. It supports JTAG, aWire, SPI and PDI interfaces, has hardware and software programmable breakpoints, and is powered by and interfaces with the host computer via USB.

Figure 3.6: The JTAGICE3.

3.4 Saleae Logic

Saleae Logic (seen in Figure 3.7) is a logic analyser used to record, view, measure and interpret digital signals, featuring 8 channels and 9 probes, and with a sampling frequency up to 24MHz. It supports both newer GNU/Linux distributions and Windows, which makes it highly flexible. It is a handy tool in both prototyping and to document results, and is used throughout this thesis.

Figure 3.7: The logic analyzer used throughout this thesis.

4 Operating Systems

As stated in the introduction of Chapter 3, the ARM platform was chosen for its modernity and versatility. These are also the reasons for focusing mostly on the GNU/Linux OS. FreeRTOS and QNX will, however, also be presented as alternatives to GNU/Linux, albeit as more specialized Real-Time OS (RTOS). The exercise program of the course TTK4147 utilizes all three OS in some manner, either through virtual computers or different hardware platforms. By enabling support of these on a single target we will streamline the educational resources.

This chapter will present the three OS which will be compatible with the hardware platform, and are relevant for this thesis.

4.1 GNU/Linux

The presentation of the Unix/GNU/Linux is based on [1]. The first production of Unix was installed in early 1972 and was by the start of the 1980s a leading force in commercial startups. It had achieved its market position through its popularity in academic circles. However, since Unix was a commercial product they wanted to prohibit illegal copying and redistribution. Therefore, as most manufacturers, they stopped distributing source code and began using copyright and restrictive licenses. This caused free software activist and Harvard graduate Richard Stallman to launch the GNU Project in 1983. GNU is a recursive acronym meaning "GNU's Not Unix", and its aim was to create a completely free Unix compatible "Unix-like" OS. By 1991 they had completed most mid-level portions of the OS. However, the team struggled to make progress with their kernel, GNU Hurd. Due to an ambitious design there were severe implementation problems. Then, in 1991 Linus Torvalds used the GNU's

[1] 2.1. History of Unix, Linux, and Open Source / Free Software[31]

development tools to produce his Linux kernel, originally only intended to work on the Intel 386(486) processors. However, Linux quickly became a popular community driven project with Torvalds as chief architect. The existing programs from the GNU project where then ported, resulting in a complete computer OS composed entirely of free software.
Today, there are several GNU/Linux distributions, were an OS is built on top of the Linux kernel. These range from distributions for the most powerful supercomputers, to the smallest embedded systems.

4.1.1 Naming

There is some naming controversy regarding Linux; should it refer to the kernel only, or to the entire operating system? "Linux" has become a far more widespread name than its partner, GNU, and this even though popular distributions like Ubuntu uses almost equal amounts of code from both (8-9% of total LOC[2]). This thesis will however use the term GNU/Linux as a general term to all OS based of the Linux kernel, quoting Richard Stallman "*people tend to think it's all Linux, that it was all started by Mr. Torvalds in 1991, and they think it all comes from his vision of life, and that's the really bad problem*".

4.1.2 Kernel Architecture

Linux is a monolithic kernel, this can be observed in Figure 4.1, and will be elaborated on later. The GNU/Linux OS is therefore somewhat more complex than other OS that will be used in this thesis project. Device drivers and kernel extensions run in kernel space with full access to the hardware. The kernel does support the loading of kernel modules in runtime. Device drivers can thus be loaded or unloaded while running the system. As the Linux kernel is also a file based system, user applications can interact with hardware through files. Device drivers are mapped to the */dev* and/or */sys* directories, while processes are mapped to the */proc* directory.
The Linux kernel supports:

- True preemptive multitasking
- Virtual memory

[2] How much GNU in GNU/Linux [6]

- Shared libraries
- Demand loading
- Memory management
- TCP/IP
- Threading

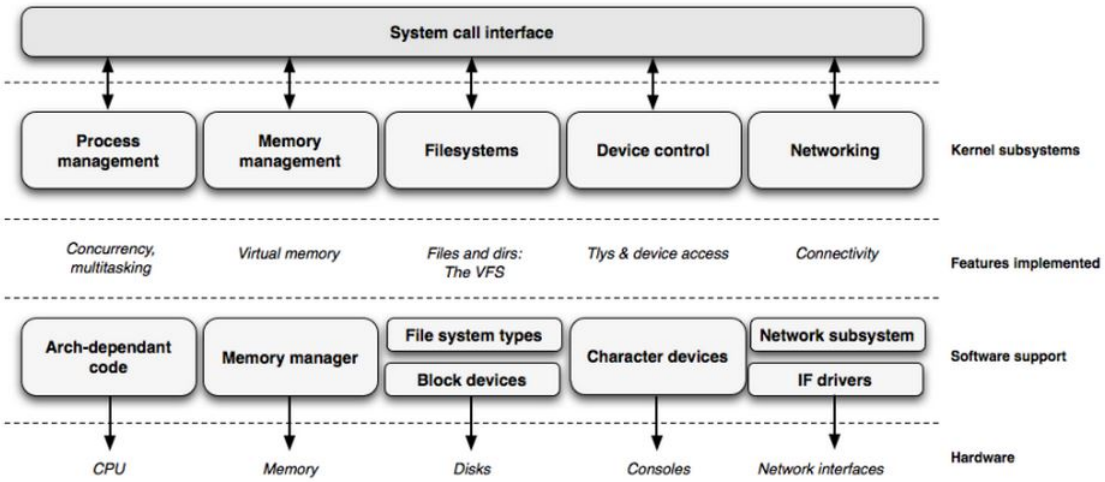

Figure 4.1: Functions of the Linux kernel. Courtesy of [1].

4.1.3 Kernel and User space

A GNU/Linux OS can be divided into two levels - User space and Kernel space, as illustrated in Figure 4.2. The OS divides the virtual memory between these two modes, thus protecting data and functionality from faults. The purpose of the kernel space is to run the kernel and most device drivers. Device drivers are programs which interacts with some device attached to the processor. The kernel space creates a stable foundation for the user space, which runs the normal user processes. The user space processes can only access a small part of the kernel via the system interface.

[1] http://knowstuffs.wordpress.com/tag/kernel-architecture/

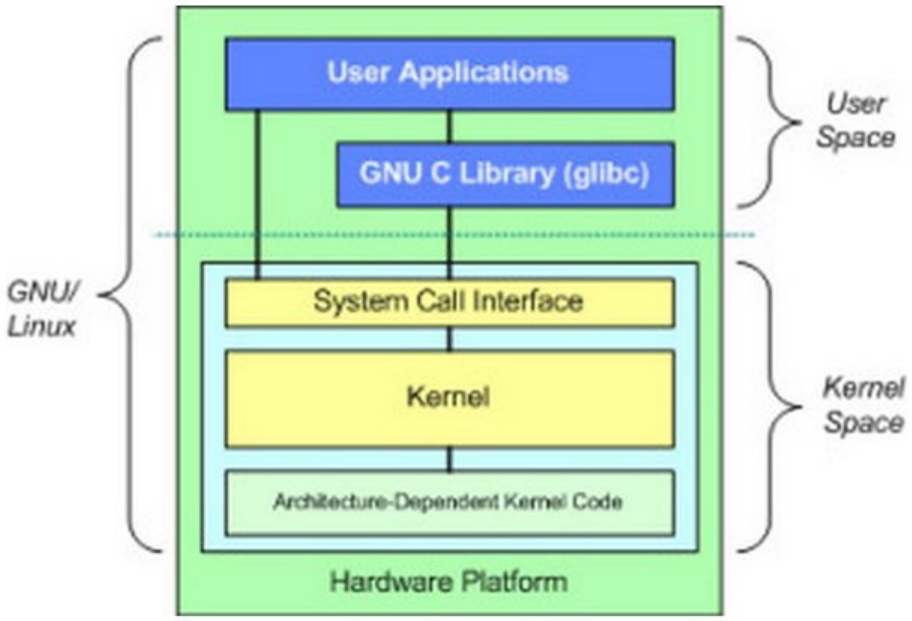

Figure 4.2: Kernel and user space and the different levels of each. Courtesy of [1].

4.1.4 Microkernel vs Monolithic kernel

A microkernel differs from a monolithic kernel in that it only the most basic functions are available from system calls, the kernel is broken down into separate processes, known as servers. This structure is implemented in many real-time operating systems. The practical purpose of using a microkernel is to sacrifice some performance for reliability. In a monolithic structure, a service is obtained by a single system call. In a microkernel structure a service is obtained through Interprocess Communication (IPC), this causes overhead due to the required context switch. However, since the microkernel is divided into different servers, if one fails, the other servers will work efficiently. In critical operations as control systems often are, this is an extremely important feature.

The idea of microkernels did not breakthrough until the end of the 1980's. Windows and GNU/Linux (and most UNIX like) OS are therefore in essence monolithic. Since the introduction of microkernels, most monolithic kernels have become more of a hybrid solution, see Figure 4.3. However, since they are in their core still a monolithic kernel the problem with this structure remains. These kernels needs to be completely foolproof or else the user will experience complete system crashes, an experience

made famous by the infamous Windows bluescreen.

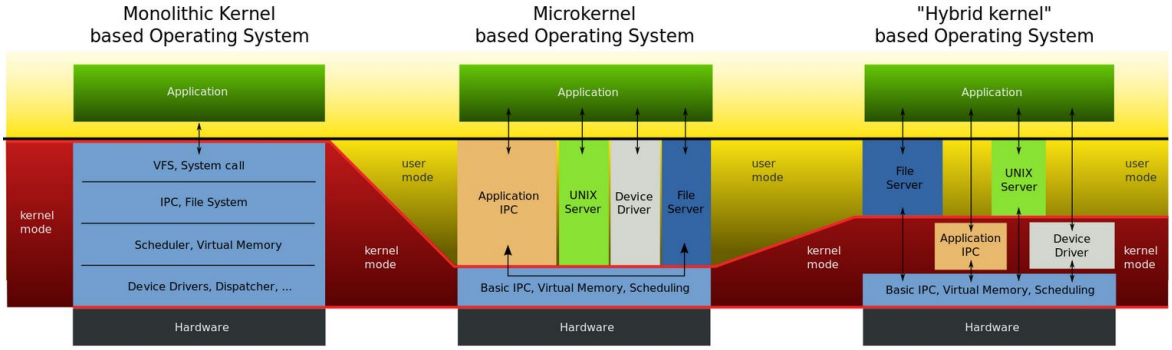

Figure 4.3: Figure showing how a microkernel differs from the standard monolithic kernel. Courtesy of [1].

4.1.5 Linux Standard Base

Linux Standard Base (LSB) is a standardized method for creating software system structure. LSB is a superset of the Portable Operating System Interface (POSIX). POSIX is an IEEE standard[3] first released in 1988, with the latest revision in 2008. The standard contains specifications for Unix-like operating system environments. It is implemented as an extra layer between the OS and applications, thus creating code compatibility between OS. It includes specifications for the command line, scripting, user-level programs, services, program-level I/O services and threading. LSB extends this specification with its stated goal being: *"to develop and promote a set of open standards that will increase compatibility among Linux distributions and enable software applications to run on any compliant system even in binary form. In addition, the LSB will help coordinate efforts to recruit software vendors to port and write products for Linux Operating Systems.*[4] *"*

[1] http://en.wikipedia.org/wiki/File:OS-structure2.svg
[3] 1003.1-2008 - Portable Operating System Interface (POSIX(R))[13]
[1] http://knowstuffs.wordpress.com/tag/kernel-architecture/
[4] The Linux foundation, Linux Standard Base[7]

4.1.6 Preemption and real-time viability

This thesis investigates themes relevant to real-time systems, for GNU/Linux to be fully relevant in this category implies that preemption is a necessity. Towards Linux kernel version 2.6 preemption had moved from availability only in user space to the possibility of interruption of kernel code. However, there were still sections in the kernel code, which could not be preempted. The PREEMPT_RT patch may be a solution. This patch supports full preemption of critical sections, by making in-kernel locking-primitives spinlock and rwlock pre-emptible. The drawback is that some device drivers can stop functioning because they are dependent on non-pre-emptible sections, which now are not possible. The creation of non-pre-emptible sections is still possible, but only thrugh using raw spinlocks. However, this should only be used in bounded situations, as Steven Rostedt, maintainer of the stable version of the PREEMPT_RT patch explains: *"If there's an unbounded latency that's in PREEMPT_RT, we consider that a bug, and work hard to fix it.*[5]*."*. Rostedt also specifies that the patch is to provide GNU/Linux with something very close to a hard real-time OS: *"I will be the first to tell you that I wouldn't want the PREEMPT_RT kernel to be controlling whether or not the plane I'm flying on crashes. But it's good enough for robotics, stock exchanges, and for computers that have to interface with hard real-time software. PREEMPT_RT has been used on computers that have gone into space."* The complexity of Linux makes PREEMPT_RT a hardening of Linux real-time capabilities, but it's far from mathematically provable.

4.1.7 Scheduling

To support real-time scheduling the kernel contains three scheduling classes named SCHED_FIFO (first-in-first-out), SCHED_RR (round-robin) and SCHED_DEADLINE, which implements earliest deadline first.

4.1.7.1 Priorities

The PREEMPT_RT patch makes several changes to interrupt handling, locking mechanism, and also the scheduler. As mentioned in the last section, the kernel implements the scheduling classes, and these implement a strict priority order. The

[5] Steven Rostedt, interview [3]

priorities vary from 0 to 99, with inverted priority values, i.e. 0 is the highest priority. While in the standard Linux kernel, process priorities are dynamic. The scheduler keeps track and adjusts a process priority periodically.

4.1.8 Device Tree

The growing popularity of SoCs and Linux for ARM devices created some problems in the early 2010s. Each SoC or board had its own hardware-specific code, this caused major problems as drivers, which could be shared with other SoC families, had been put under board-specific code. The duplication of code spawned a famous Linus Torvalds quote: "Gaah. Guys, this whole ARM thing is a f*cking pain in the ass. " in 2011[6]. This spawned a project to build a single Linux kernel, which would boot on different ARM SoCs. The project made use of device trees for describing hardware to solve this problem. A device tree is a data structure (seen in Figure 4.4), which enables the description of most board design aspects, thus enabling hardware specification to be read at boot time and dynamically configure the device drivers. The Device Tree description includes:

- The number and type of CPUs.
- Base addresses and size of RAM.
- Busses and bridges.
- Peripheral device connections and GPIO set up.
- Interrupt controllers and IRQ line connections.

4.1.9 General-purpose input/output Control

The move from 8-bit architecture to a more advanced 32-bit architecture will almost always also induce a move to a more advanced OS like GNU/Linux. I/O operations will then be hidden from barebone programming. There are basically two ways to control a General-Purpose Input/Output (GPIO) pin from userspace in GNU/Linux: memory mapping or sysfs. And if we want to use a GPIO pin as an interrupt source,

[6]Linus Torvalds, Linux kernel mailing list [29]

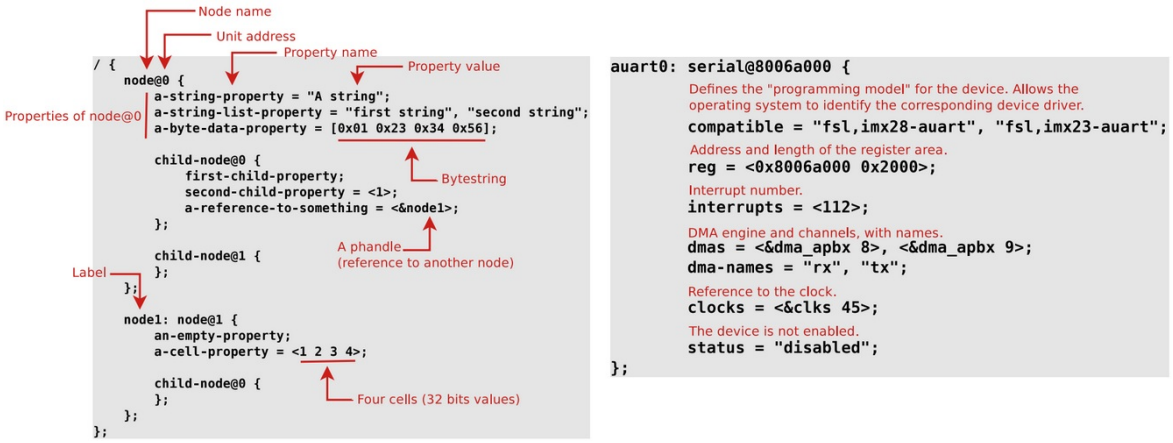

Figure 4.4: General device tree structure and an example of implementation.

we also have to options. Either use polling combined with sysfs files in userspace or by writing a kernel module.

4.1.9.1 Sysfs

Sysfs is a virtual file system provided by Linux. It describes the devices known to the system and is by default mounted on */sys*. The purpose of each file is to pass on information to the kernel so that it can perform a specific command. Platforms which implement the abstraction layer *gpiolib* provides files mounted on */sys/class/gpio* for interacting with the GPIOs in userspace. Linux provides a program called echo for displaying a line of text[7], it can also push be used to push a line of text into a file. Echoing a pin number to */sys/class/gpio/export* or *..unexport* will expose or remove a pin from userspace. For example, *echo 10 > export* will export the control of pin 10 to userspace and create a file */sys/class/gpio/gpio10* which can be given commands to. Available commands are:

- *direction*, with arguments in or out. For if the pin should be input or output.
- *value*, with arguments 0 or 1. Low or high output.
- *edge*, with arguments none, rising, falling or both. These are arguments for selecting the signal edge for using a gpio pin with the polling function poll.

[7] http://linux.about.com/library/cmd/blcmdl1_echo.htm

- *active_low*, with arguments 0 or 1. Nonzero value will invert the value attribute for both reading and writing.

4.1.9.2 Memory mapping

Memory mapping associates a range of user-space addresses to device memory. Using the POSIX-compliant *mmap()*[8] we can open */dev/mem* and map a device's physical address space into a process's virtual address space. Thus, by reading or writing to that assigned address range, we are actually accessing the device. Memory mapping can therefore provide quick and easy access to a device for performance-critical application. It does, however, require the application to run as superuser, which can cause trouble if the programmer makes a mistake in the addressing. There are no possible interrupt handling, and no protection against simultaneous access.

4.1.9.3 Kernel module

When trying to use a GPIO as an interrupt source, the most direct way is to do it through the kernel and implement a kernel module. Modules are pieces of code that can be loaded and unloaded into the kernel upon demand, and are often device drivers[9]. This makes it is possible to extend the features of the kernel without the need to reboot or recompile. The code running in the kernel can map I/O to interrupts, and register handlers to those interrupts, among a lot of other features.

4.1.9.4 Polling

Most interrupt handling gets carried out in the kernel. However, by using a kernel driver in combination with the sysfs interface and poll, we can also do I/O event driven operations in user space.
The *poll()* system call can block a process until any of a given set of file descriptors becomes available for reading or writing. As we earlier presented 4.1.9.1 we mentioned how it GPIOs could be controlled through such file descriptors. We can then create an event driven GPIO driver in user space by, exporting pins, setting them to be edge triggered, opening their files and passing the file descriptors to a *poll()* call.

[8] Linux Device Drivers, chapter 13 [25]
[9] The Linux Kernel Module Programming Guide, Chapter 1 [25]

4.1.10 Distributions

As previously stated a GNU/Linux distribution is an operating system built on top of the Linux kernel. Distributions often include some Package Management System (PMS), where each package contains a specific application or service. They include compiled code, with installations and removal handled by the PMS, thus packages which are dependent of other packages will be detected, thereby easing the installation process. The PMS will also handle upgrading packages and continually check that all the dependencies are fulfilled. The distributions typically contain a range of packages, and the system administrator can also add packages not included in the distribution. There is a huge number of distributions fitting most imaginable applications. There are commercially backed distributions like Fedora and Ubuntu, and entirely community-driven distributions, such as Debian and Arch Linux. Figure 4.5 shows the distribution timeline from 1993 until today. Each line represents its own distribution of GNU/Linux. The figure is not meant to be closely studied, but instead provide an illustration to the vast size of the GNU/Linux community.

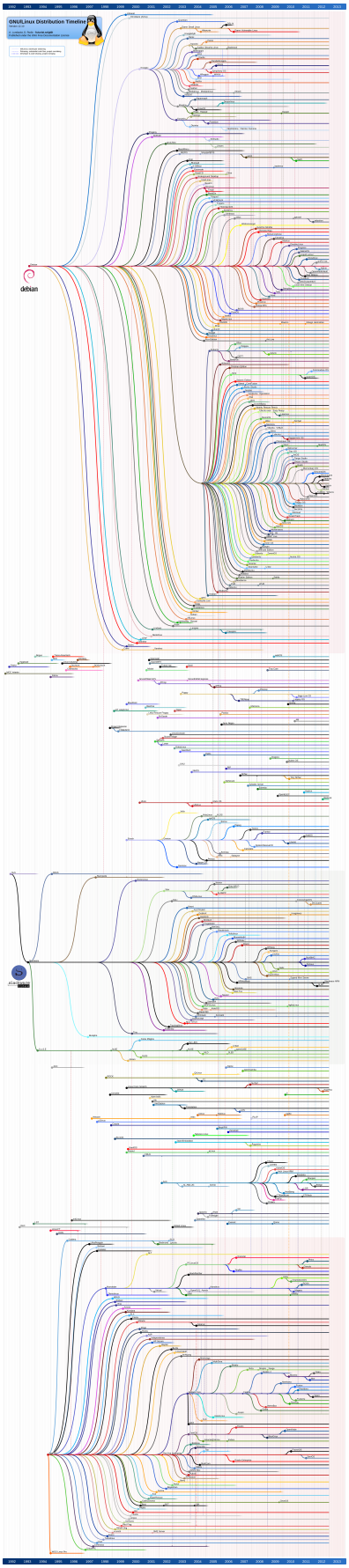

Figure 4.5: General Device Tree structure and a example of implementation.

4.2 QNX

QNX Neutrino [10] is a real-time operating system (OS), and its primary market target is embedded systems. QNX was developed in the early 1980s as a commercial Unix-like OS. However, instead of an monolithic kernel, QNX implements a microkernel. The OS can be used in a large variety of platforms including x86, PowerPC, ARM and many others.

4.2.1 Neutrino microkernel

The microkernel implements core POSIX features along with an IPC message-passing service. As the implementation is a microkernel, then the file system, networking and similar functions are provided by optional servers which can be configured at compilation. The kernel is primarily coded in C, and according to QNX, performance goals are achieved by: *"successively refined algorithms and data structures, rather than via assembly-level peep-hole optimizations"*. QNX Neutrino is a fully preemptible OS, and the entire OS is based upon kernel calls to support[11]:

- Threads
- Message passing
- Signals
- Clocks
- Timers
- Interrupt handlers
- Semaphores
- Mutual exclusion locks (mutexes)
- Condition variables (condvars)
- Barriers

[10] http://www.qnx.com/products/neutrino-rtos/neutrino-rtos.html
[11] The QNX microkernel [22]

Fully preemptible means that even kernel operations as message passing can be preempted and resumed with no harm done. However, within a system call there are some minor critical sections, which turns off interrupts (see Figure 4.6). The minimal complexity of the kernel helps as it makes it possible to place an upper bound on the longest non-preemptible code path. Interrupts and preemption are disabled in very brief intervals (hundreds of nanoseconds).

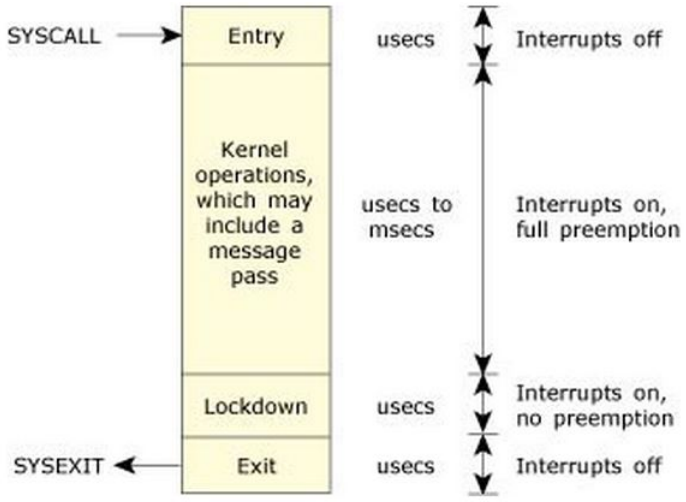

Figure 4.6: Example that show when a system call is preemptible, and when interrupts are on/off. Courtesy of [22].

4.2.2 Threads and processes

Any given process always contains at least one thread, and can be thought of a "container" for threads through defining an address space, which all of the threads share. It is noteworthy to mention that even though all threads in a process share address space, thus each thread are allowed private data. This private data can for instance be the thread ID protected within the kernel, or that each thread has a stack for its own use.

4.2.3 Thread lifecycle

An executing thread can be described as either "running", "ready" or "blocked". There are however a lot of different states within "blocked", as shown in Figure 4.7. Except

for the “running” state in which a thread has processing power, and “ready” where the thread is waiting to acquire time by the scheduler, the rest are other “blocked” states. In these cases the threads are waiting for some other factors like sleep, semaphores or resources, before they are able to return to the ready queue.

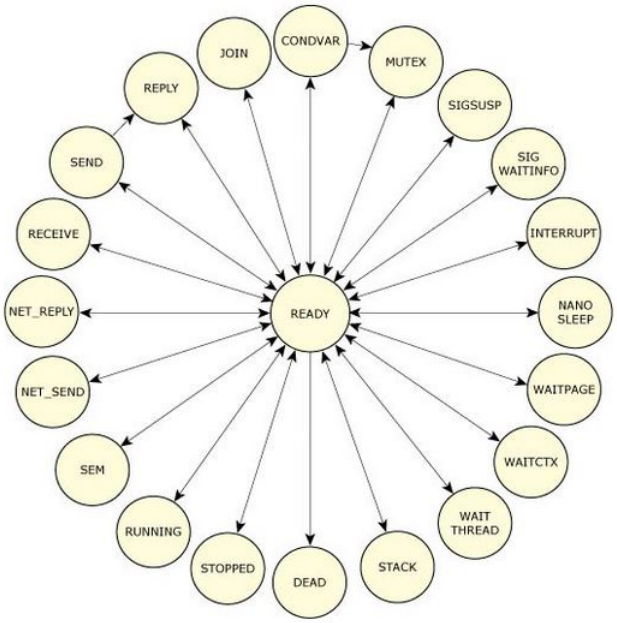

Figure 4.7: Possible thread states. Courtesy of [22].

4.2.4 Scheduler

The scheduler is invoked whenever a decision on which thread is to be given processing time next. This happens whenever the running thread is blocked, preempted, yields or uses its allocated time-slice. There are 0-255 priority queues, with 255 being the highest priority. When several threads are ready, QNX provides three scheduling algorithms:

- First in, first out (FIFO) scheduling
- Round-robin scheduling
- Sporadic scheduling

These algorithms can be changed by within each thread, this way different threads can use the scheduling that fits its purpose best.

In FIFO scheduling the first thread entering the ready state will be the first to execute at the given priority. It will execute until it either blocks, yields or is preempted by a higher-priority thread.

This holds also for the round-robin scheme, however, in this case the threads are also given a time-slice. If they do not relinquish control within this set timeslice they will be forced back into the ready queue, giving the next thread access.

The sporadic scheduling algorithm is more complicated. It is used to provide a capped limit on the execution time of a thread, within a given period of time. The threads priority can dynamically change between a normal (foreground) priority and a low (background) priority. As in FIFO scheduling a thread runs until it is blocks, yields or preempted. The difference is that the a thread is allocated a fixed amount of time it is allowed to run at normal priority (N) before dropping to low priority (L). By also controlling the replenishment period for normal priority execution, and the value of the low priority, sporadic scheduling can be a powerful tool for handling aperiodic events, without missing hard deadlines of other threads. An example is illustrated in Figure **??**.

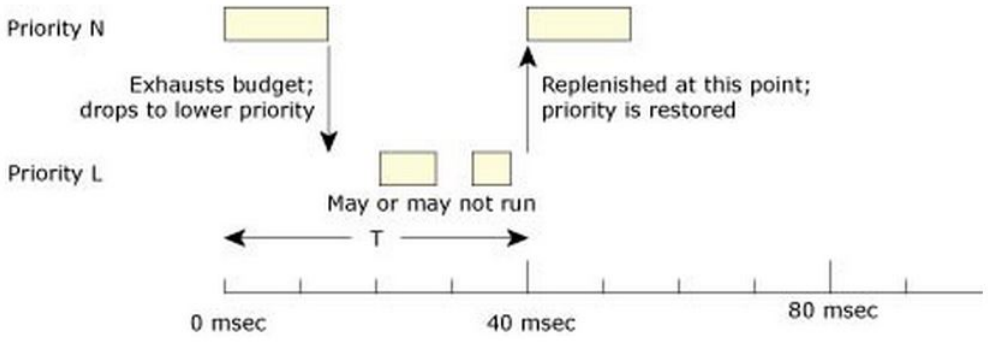

Figure 4.8: Thread drops in priority until its budget is replenished. Courtesy of [22].

4.3 FreeRTOS

FreeRTOS[12] is a real-time operating system and its primary market target is again embedded systems. However, FreeRTOS is as its name indicates a free, open sourced OS. It implements a simpler microkernel then QNX were its features include, preemptive tasks, support for 34 hardware architectures, a small footprint, and it is written in C. The basic non-commercial version does however, not include support for network communication, external hardware drivers, or a file system.

4.3.1 Tasks in FreeRTOS

There are no software restrictions to the number of tasks that can be created in FreeRTOS, though the practical number of maximum tasks will be limited by hardware and memory. Tasks are created before the a call starts the scheduler. Any created task shall always be wrapped in an infinite loop, or to invoke vTaskDelete, which free all allocated memory to this task by kernel.

4.3.2 Lifecycle

In FreeRTOS a task is either running or not, and only one task can run at any given time. When not running a task will be in one of three states, as depicted in Figure 4.9. When a task is delayed or waiting for another task (through semaphores or mutexes) it is said to be “Blocked”. A task can also be suspended, when a task gets suspended it will stay in that state until it gets resumed, by either a task or the kernel. The last state in the "Not running" superstate is the ready state. This is where all the tasks which are not waiting for any events, but there is a equally or higher prioritized task running at that time. If a task with higher priority than the one currently running enters the ready state, and preemption is enabled, then the schedular will force a context switch.

4.3.3 FreeRTOS scheduler

The scheduler has the responsibility of choosing which task in “Ready” states is to be given processor time. The FreeRTOS scheduler operates as an ISR at a given tick

[12] http://www.freertos.org/

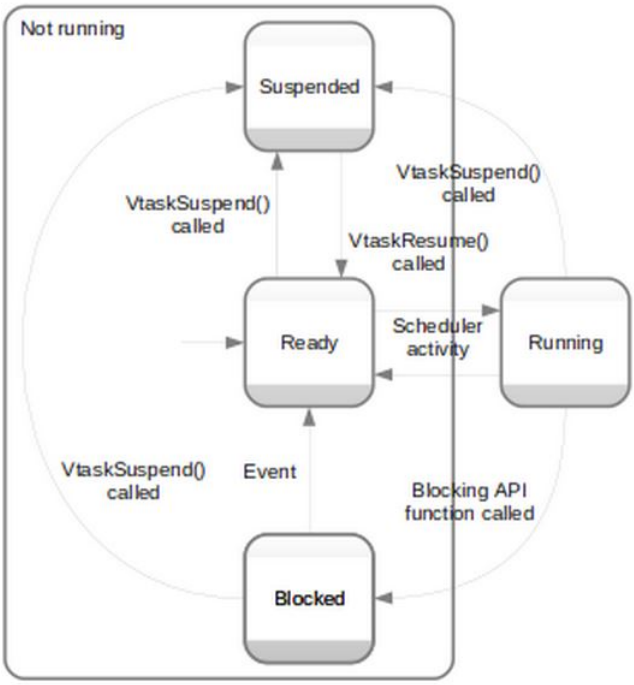

Figure 4.9: Lifecycle of FreeRTOS tasks. Courtesy of [19].

period. This tick period is defined in the kernel header files. Figure 4.10 shows the scheduler algorithm run in the ISR. The first operation is to reset the counter timer, this is to initialize the next tick period. If the scheduler is co-operative, then the only action is to increment the tick count. However, if the scheduler is preemptive then the scheduler can force a context switch. Therefore the context of the current task is saved before the tick count is incremented. The scheduler then checks if the tick incrementation caused any higher priority blocked tasks to unblock. If so, then a context switch is executed before context is restored, and the ISR returns.

The scheduler is started through a call to the vTaskStartScheduler() function. This should be the last function called in main, after all the required tasks have been created. It then creates the IDLE task, this is the task running with at lowest priority. Further, it sets up the time interrupt to invoke the scheduler. Since this is hardware dependent, the configuring occurs in the Hardware Abstraction Layer (HAL), this is the part which requires porting when compiling FreeRTOS for new platforms. After this is done, context is restored and the tasks will begin to run.

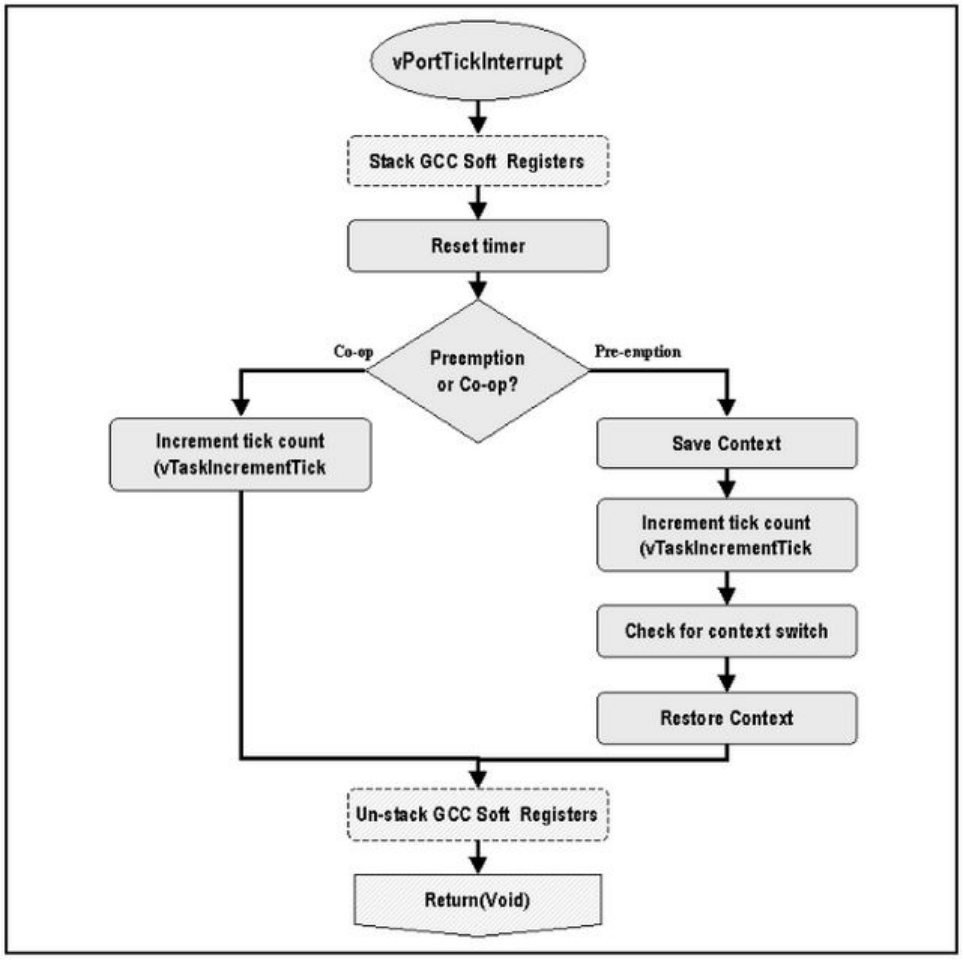

Figure 4.10: ISR scheduler algorithm for FreeRTOS. Courtesy of [19].

5 Hardware solution

5.1 Proposed solution

The problem formulation 1.2 states that this thesis show create an embedded test platform. Figure 5.1, which is based on the hardware and software choices made in Chapters 3 and 4, shows a sketch of the proposed hardware solution to this problem. It depicts the BeagleBone Black, together with SD cards representing the possibility for multiple OS, and the boards expansion, the External Response Tester (ERT) cape. This chapter will present how the ERT was made and programmed.

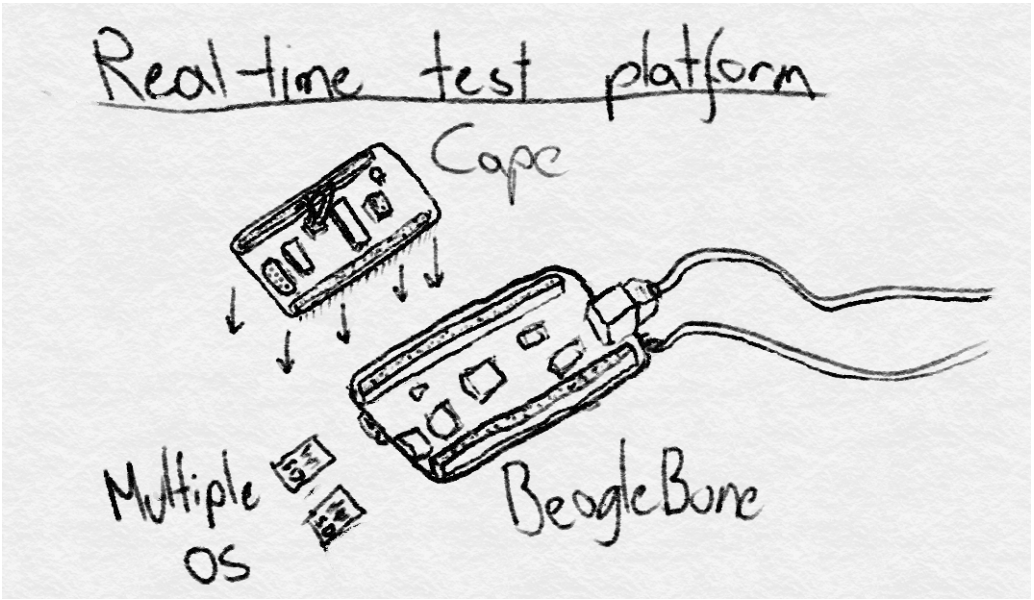

Figure 5.1: Sketch of hardware solution.

5.2 External Response Tester

As mentioned in the introduction of Chapter 3, the Beaglebone platform was chosen for being a modern embedded development platform with a range of compatible OS. However, there was need to create an equally versatile device, which could be used to simulate external sources that the BeagleBone applications could interact with. Presently the exercises in the course TTK4147 Real-time systems use the AVR butterfly evalution tool to enable these kind of tests.

5.2.1 AVR butterfly

The AVR butterfly is a ATmega169 MCU with a small LCD screen and a four-direction joystick, shown in Figure 5.2.

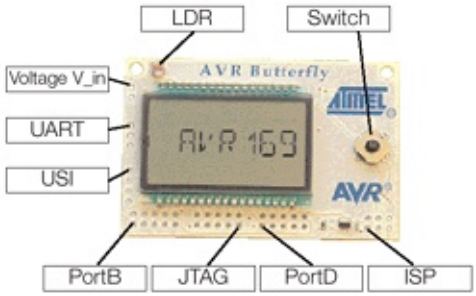

Figure 5.2: The AVR butterfly. Courtesy of [1].

In the course assignments the butterfly is programmed to be a real-time response tester. It transmits out a signal on 1 to 3 output pins, and then waits for, and times the response on 1 to 3 (other) input pins. Each test runs either 10, 100 or 1000 such timed responses on each pin. The results of such a response test is then be placed on the butterfly UART.

5.2.2 BeagleBone Cape

This project set out to create some device with similar, but extended features compared to the AVR butterfly. A BeagleBone cape is a device which can be plugged into the board headers extending its features. Many different capes exist through

[1] http://www.atmel.com/tools/avrbutterfly.aspx?tab=overview

the BeagleBone community, these range from LCD screens to motor control. Thus, to enable cape prototyping we took advantage of such a community cape, the BeagleBone Breadboard shown in Figure 5.3. This was used to ensure that the design performed according to our expectations, before creating and ordering a Printed Circuit Board (PCB).

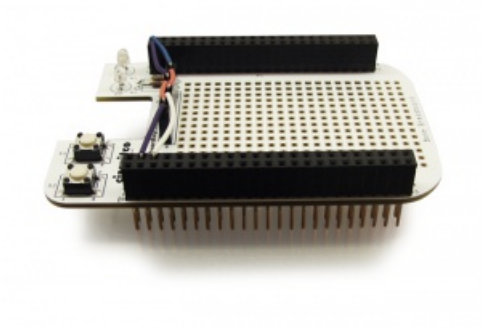

Figure 5.3: The BeagleBone Breadbord. Courtesy of [1]

5.2.3 Prototyping

The ERT cape features can be summarized as follows:

- PCB printed BeagleBone Black compatible cape.
- 3 output signal and input response lines to the BeagleBone.
- Create output signals based on an internal clock prescaled and set up by the BeagleBone.
- UART communication to the BeagleBone, and to a possible external computer.
- Fully deterministic behaviour.

The key component for the prototyping was the Atmel ATMega168, which was chosen to be the MCU for this project. The 8-series MCU has an extensive feature set while still being delivered in a rather compact 28-pin DIP chip. DIP stands for Dual in-line package, which features long vertical pins ideal for breadboard prototyping, a typical DIP package is shown in Figure 5.4.

[1] http://elinux.org/CircuitCo:BeagleBone_Breadboard

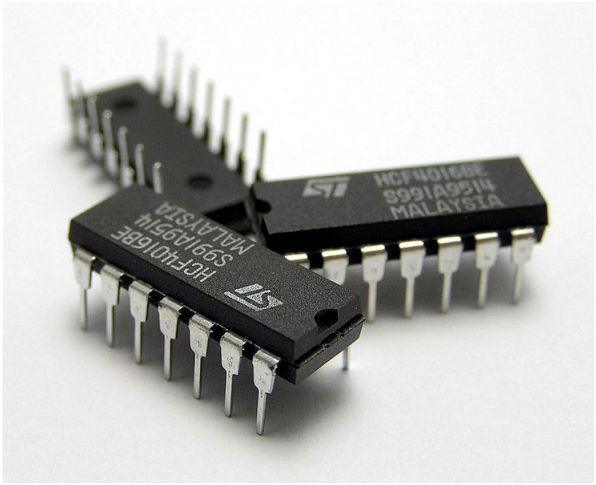

Figure 5.4: Typical DIP package.

The most important feature for this project is that the MCU features an Input Capture Pin (ICP). The ICP provides an edge-triggered read of a 16-bit timer running on the MCU, with an optional trigger of the ICP interrupt routine. This enables a time measure of a signal sent from an output pin to the BeagleBone, until a signal is received on the respective input pin. The challenge, however, is to be able to set up the hardware and software such that the ICP always has stored the correct timer value when the subsequent pin operation grabs this value.

5.2.3.1 Schematics

The schematics of the prototype (Figure 5.5) was created by a program called Altium designer, which we will return to in greater detail under the PCB design section.

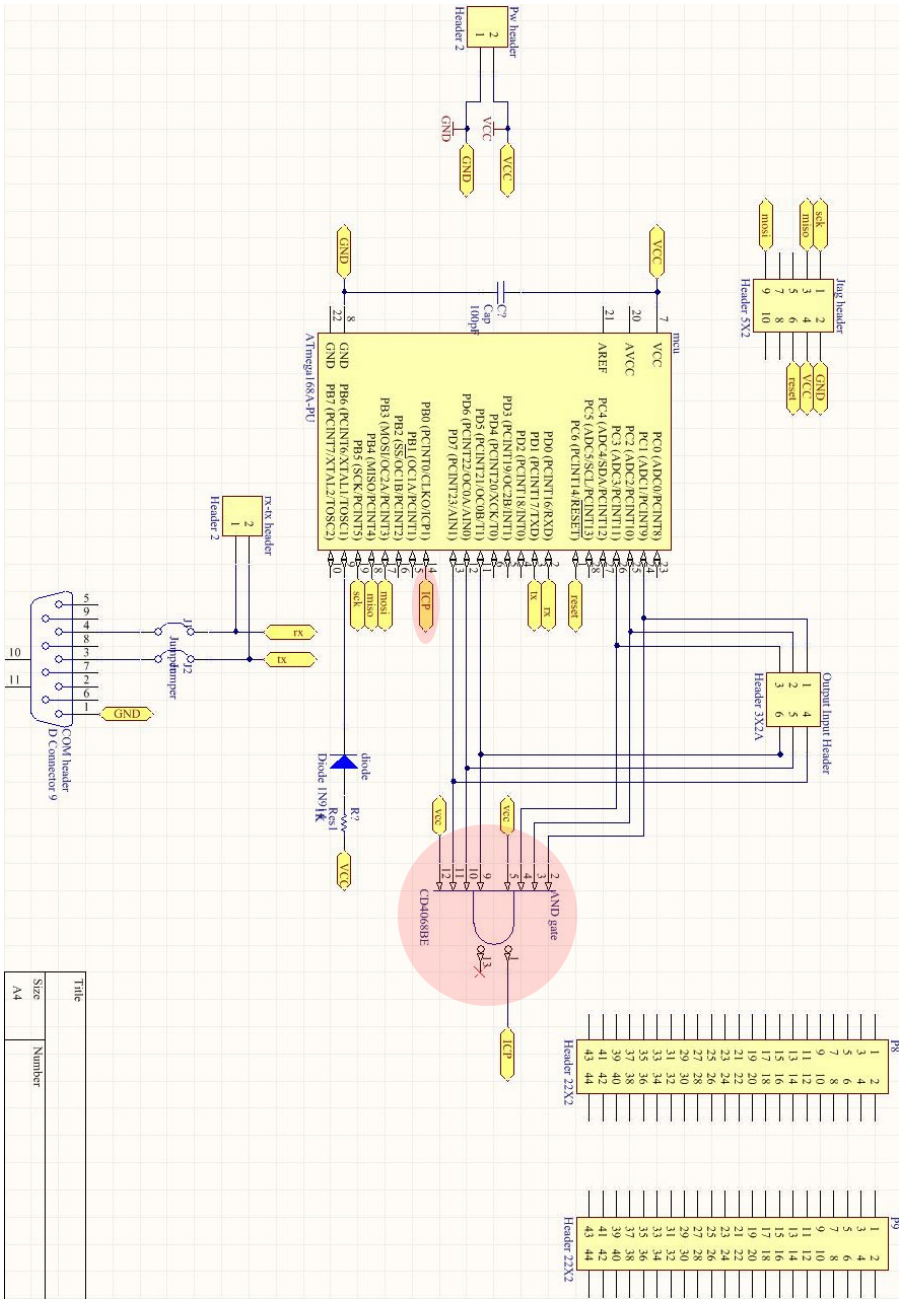

Figure 5.5: Prototype schematics. From Altium designer.

Explanation of the design presented in Figure 5.5

Every signal is initiated by pulling a line from 1 to 0. Thus, by connecting every input and output signal line to an AND gate, the ICP pin will get a negative edge every time one of the signal lines gets drawn to ground. The AND gate has 8 inputs, where 6 is used by the signal lines, as depicted by component CD4068BE and its connections, highlighted in the schematic figure above. The C port on the MCU with pins 1 to 3 is used as input, while the D port with pins 5 to 7 is used as output. It should be noted that if one signal lines is low, then all subsequent signals sent or received until it goes high again, will record the same timestamp. Therefore, it is important that all signals be sent with a short pulse. Thus, this inaccuracy will be insignificant in regards to the entire response time.

The input and output pins, as well as the tx/rx pins of the UART, are connected to separate headers on the board instead of directly to the BeagleBone P8 and P9 headers. The reason for this is to be flexible in which GPIO and UART pins to use, thus ensuring compatibility across different OS and later revisions of the BeagleBone board.

The design also includes a JTAG header for programming, separate COM port header for UART transmission with external PC, and a diode for the final design proposal. The Figure 5.6 shows the prototyping testbench with the BeagleBone Breadboard cape, JTAGE ICE3 and Saleae Logic analyser.

Test example

To explain further we include an example on how the cape works when sending three signals, one on each output pin, and then timing the response from the BeagleBone. We will return to how both devices were programmed later. The screenshot in Figure 5.7 was taken with the Saleae Logic analyser, which was connected to the ICP and the input/output lines of the cape. The BeagleBone was set to respond to a signal as fast as possible. The three output signals from the cape can be seen on channels 0 to 2 in the figure. We also observe that the ICP line (channel 7) gets drawn to ground for each signal as expected. The responses from the BeagleBone can be viewed on channels 3 to 5. The response times were measured by the logic analyser to be 0.60ms, 1.54ms, and 3.43ms, while the cape measured and wrote timestamps equal to 0.60ms, 1.55ms and 3.44ms on the UART. The cape timer has a resolution of 0.008ms.

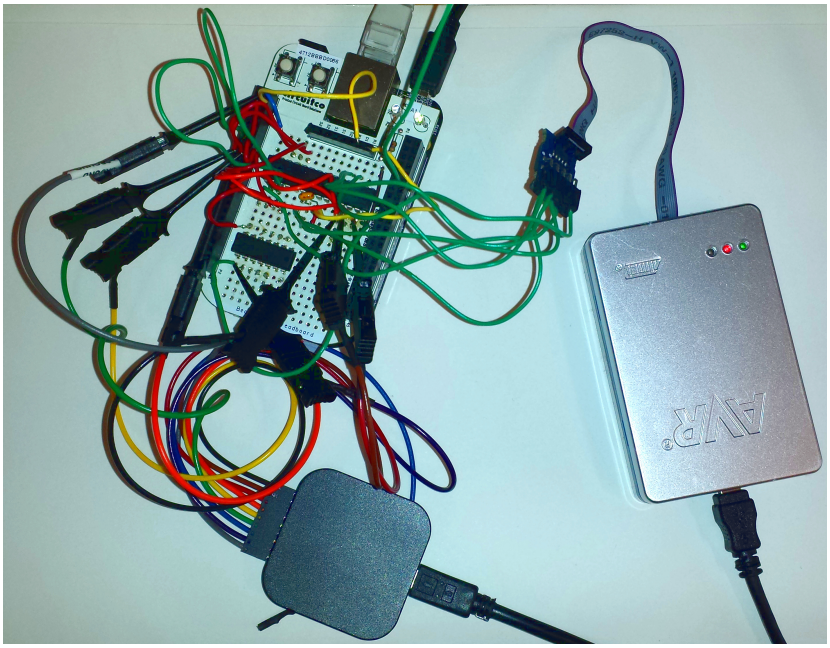

Figure 5.6: Prototype setup with breadboard, AVR JTAG ICE3 programmer and the logic analyser with connected probes.

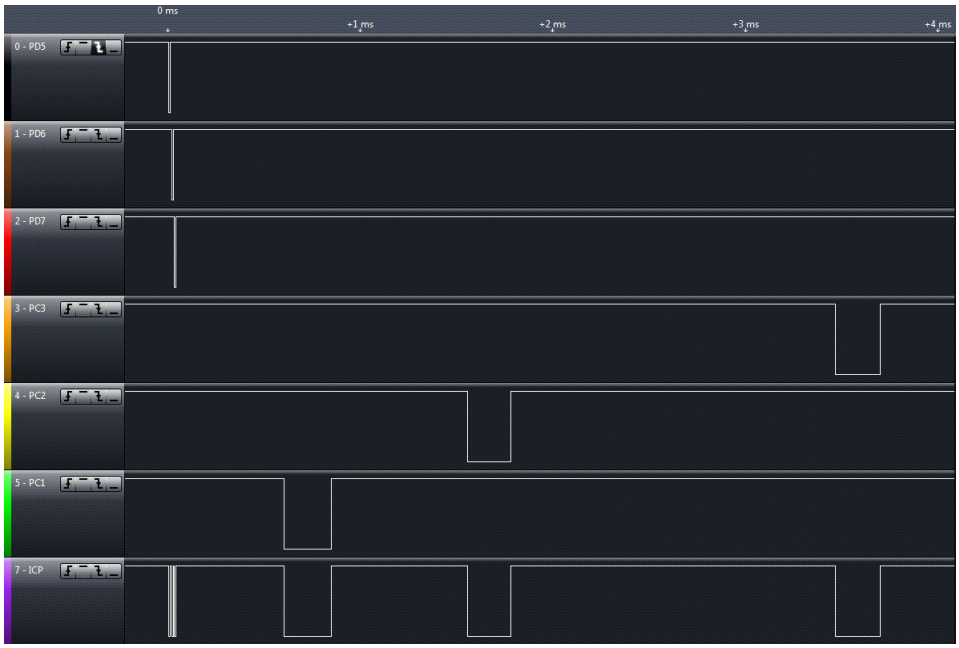

Figure 5.7: Logic analysis of signals and their responses.

5.2.4 PCB design

A PCB connects integrated circuits (IC) through layers of copper laminated onto a non-conductive substrate like silicon.

5.2.4.1 Altium designer

Altium designer is an electronic design automation software package for PCB, FPGA and embedded software design. It was used to create the cape PCB board in this project. The PCB design process starts by creating the schematics for our layout. The schematic used to create the PCB design was depicted in an earlier Figure 5.5. The largest IC manufacturers like Atmel and Texas Instruments provide component libraries, which integrates with Altium Designer, easing the design process significantly. These components also include their PCB footprint, thus when creating a PCB file and importing the schematic, all the components with their connections are available. In order to place the components on the PCB we first needed to acquire the dimensions of the final board. This was done by obtaining the original PCB files[1] for the BeagleBone Black and measuring the P8 and P9 header distance. After creating the same space between the main headers in our design we could begin placing the different components and draw the connection paths. There was no need for more then two layers, and we chose to continue with DIP sized components as we had experienced no sizing constraints.
The end result was the layout which can be seen in Figure 5.8. The final project files are detailed in Appendix B.1.

5.2.4.2 Manufacturing files

To manufacture the PCB we chose to use iTead Studio[2], which produced ten PCB boards at the expense of $20 before shipping. The manufacturing files they needed was Gerber files and drill files. These were created by using Altiums fabrication outputs and then emailed to iTead Studio. The finished product can be seen in Figure 5.9.

[1] BeagleBone Black, latest production files [5]
[2] iTead Studio, 10cm x 10cm green PCB [15]

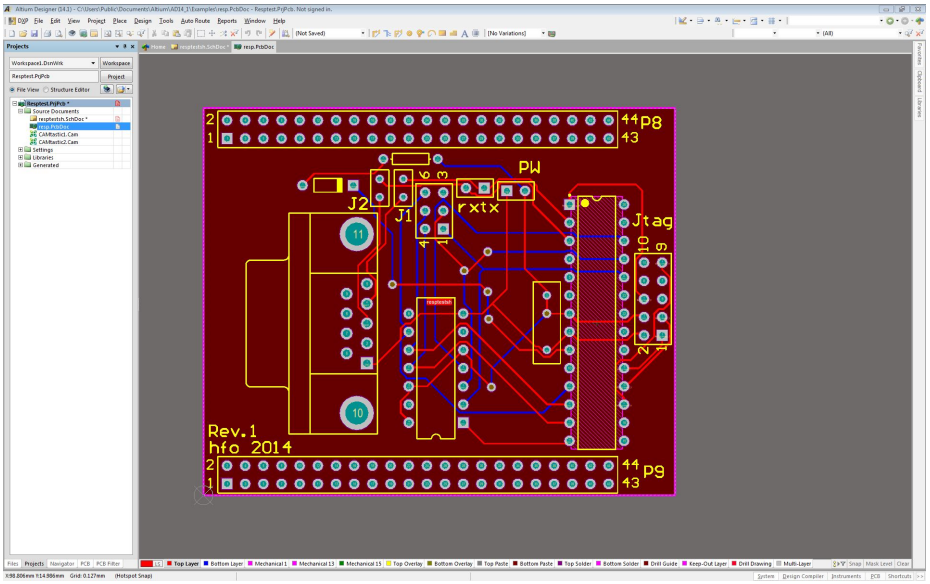

Figure 5.8: Altium designer in PCB mode.

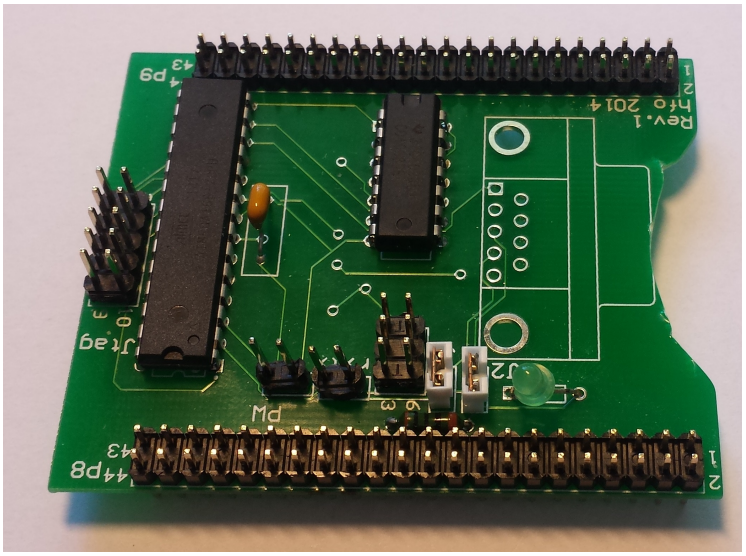

Figure 5.9: The manufactured PCB design with soldered components.

5.2.4.3 Final pinout description

Figure 5.10 shows the functionality of the different headers on the ERT cape, and the Figure 5.11 depicts how it is used in conjunction with the BeagleBone Black. Note that the input and output pins don't have the correct numbering on the final board. Input pin number 3 is channel 1 and pin number 1 is channel 3. Output pin number

6 is channel 1, pin number 5 is channel 2, and pin number 6 is channel 3.

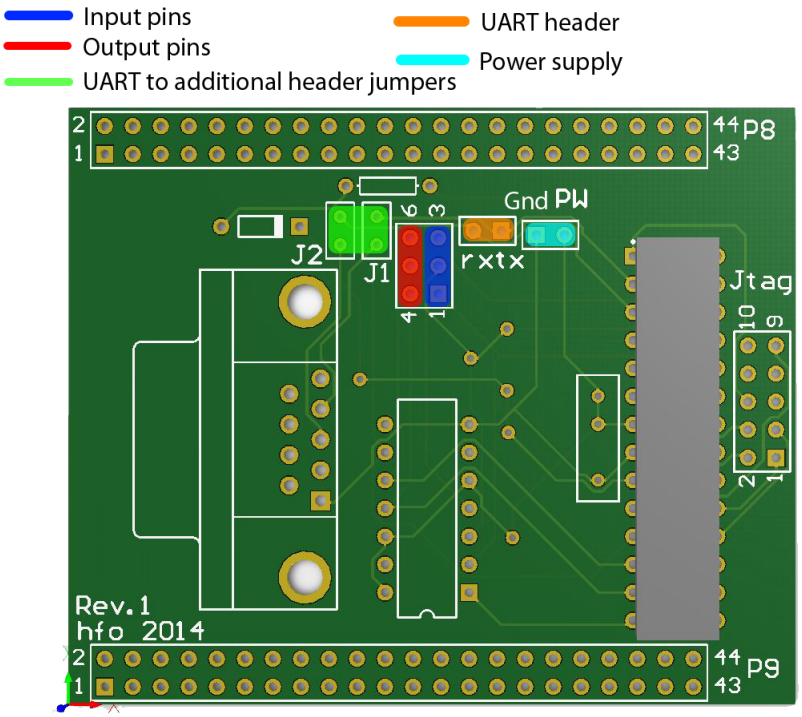

Figure 5.10: Pinout of the ERT cape board.

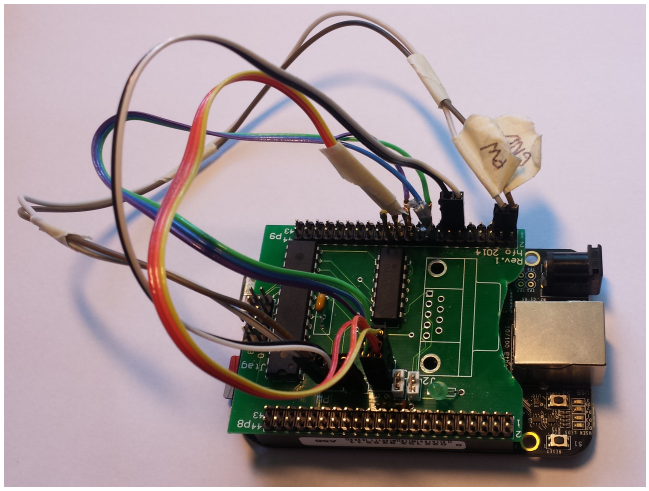

Figure 5.11: The ERT cape stacked on top of the BeagleBone Black.

5.3 ERT application

The main purpose of the ERT is to perform response testing of the BeagleBone through sending a signal and timing the response. There are six signal lines which connect to the BeagleBone GPIO, and thus, three channels containing one output line and one input line. Each of these are set to default high, and each event where one or more are dragged to ground, also pulls the ICP pin to ground. This was done because all six signal lines are also connected to an AND gate as shown in the previous schematic, Figure 5.5. To then be able to send three output signals and time the response, we will need:

- Intialize UART to let the BeagleBone specify the test that the ERT should perform.
- Intialize the three pins C1, C2, C3 as inputs, with interrupts.
- Intialize the three pins D5, D6, D7 as outputs with pull-up.
- Initialize timer1, and initialize ICP to store timer1 when pulled to ground.
- Initialize timer0 to trigger new output signals.
- Create a main loop that waits for all three responses and then places the result on the UART.

5.3.1 How to use

The ERT cape plugs onto the top of the BeagleBone, when powered it enters a simple state machine, seen in Figure 5.12. The configurable UART keywords are as follows (all the keywords have to be ended with '
n' for the ERT cape to recognize them):

- T:1000/100 or 10, configures the number of tests to run.
- C:1/2 or 3, configures the use of 1-3 channels.
- R:64/256 or 1024, configures the prescalar of the timer responsible for measuring the response interval (timer1).

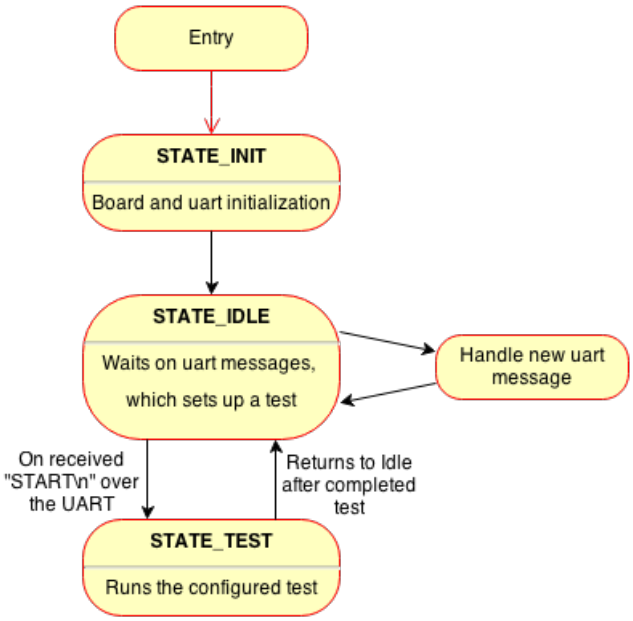

Figure 5.12: The simple state machine which the ERT enters at power up.

- F:R, configures the ERT cape to run in free-running mode. Free-running mode causes the ERT cape to send a new set of output signal one millisecond after receiving response on the last set of input signals. This is instead of waiting for timer0 to specify when to send new output signals.

- START, drives the ERT cape from IDLE state to TEST state, and it executes a test depending on how it was configured.

After a test is finished it returns to the IDLE state, it then prints the test results as a histogram. Timer1 has a tick resolution of 8, 32 or 128μs depending on the prescalar value chosen in the configuration. The histogram has three arrays, one for each channel, which each have 128 slots. Each slot has consists of 3 ticks, making each slot size, 24, 96 or 384μ. Every result received iterates one of these slots depending on its calculated time interval, thus creating a time interval. If the result overshoots the number of slots, it will increment an overflow counter, which is printed last. The Figure 5.13 depicts an example of the ending of such a histogram print.

```
116      t1: 1       t2: 0       t3: 0
117      t1: 3       t2: 0       t3: 0
118      t1: 2       t2: 1       t3: 0
119      t1: 0       t2: 2       t3: 0
120      t1: 0       t2: 7       t3: 0
121      t1: 0       t2: 0       t3: 0
122      t1: 0       t2: 2       t3: 0
123      t1: 0       t2: 2       t3: 0
124      t1: 0       t2: 0       t3: 0
125      t1: 1       t2: 0       t3: 0
126      t1: 3       t2: 0       t3: 0
127      t1: 2       t2: 1       t3: 0
128  ovf t1: 11      t2: 32      t3: 0
```

Figure 5.13: An example of the ending of a histogram print, after a successfully run test.

5.3.2 Code explanation

5.3.2.1 Defines and globals

Defines and globals for the project can be found in Appendix B.3.1. All the declared globals are defined at the top of main.c, which we will return to later.

5.3.2.2 UART setup

The ATMega168 has only one UART transceiver which utilities pins D0(RX), D1(TX). It is initialized by setting the baud rate, data format, enabling receiver and transmitter, as well as enabling the receiver interrupt in USARTInit(). The baud rate was set at 38400, which with the MCUs clock rate at 8MHz gives an error of 0.2%[3]. The data frame was set to 8 bits and 1 stop bit. The code can be found in usart.c, appended and referred to in Appendix B.3.4.
The receive interrupt routine ISR (USART_RX_vect) (see Appendix B.3.4) stores each reveived character in a 15 character buffer. It does so until it receives an end-of-line character ('\n'), and it then sets a boolean flag that tells the main loop that it has received a message. It also copies the buffer such that it will not be corrupted by new messages until the main loop gets to process it.
The new message is handled in the main loop by calling handle_usart_trans() (see Appendix B.3.4), which parses the message for known keywords. Keywords are normally used to define and set up a test that will be performed.

[3] AVR baud calculator [32]

5.3.2.3 Timer 1 and ICP setup

Timer 1 has a true 16-bit design and is initialized with a prescalar of optional value 1024, 256 or 64. The 8MHz clock rate combined with this prescalar this gives a timer resolution of 8μs, with an timer overflow every 0,52s. The timer is extended in software by iterating an unsigned short (16-bit) variable every hardware overflow by using the overflow interrupt. This gives a software overflow each 34360s or roughly every 10 hours. We also clear any pending interrupts, before enabling the ICP and overflow interrupts. Further we ensure that B0, which is the ICP pin, is set as input. The can be found in timer.c, appended and referred to in Appendix B.3.5.
The two interrupt routines are also located in timer.c, TIMER1_CAPT_vect copies the input captured timer value. TIMER1_OVF_vect implements the software overflow counter.

5.3.2.4 Timer 0 setup

Timer 0 has a 8-bit design. It is set up to trigger an interrupt on a compare match register, and its ISR routine flags the main loop such that it sends new output signals to the BeagleBone. Both the prescalar and the compare match register are configurable, and thus also the output signal frequency. The compare match register can also be changed at runtime to make a test of varying frequency. This gives flexibility to the testing. The code can be found in timer.c, appended and referred to in Appendix B.3.5.

5.3.2.5 Input/Output setup

When signals are sent and received the time is stored in a struct for each channel. This struct includes two variables for storing both the timer counter and overflow counter at send time, and two variables for the same at receive time. The struct declaration can be found in defines.h, appended and referred to in Appendix B.3.1.
The three output pins are set as outputs when the board is initialized (INIT_BOARD(), found in main.c, Appendix B.3.2), and they have their own functions, which sends signals and stores the send time (send_interrupt(byte pin) B.3.3).
The three input pins are set as inputs and pull-up when the board is initialized (INIT_BOARD(), found in main.c, Appendix B.3.2). When the program enters the state which runs tests, it performs a function called init_test() (See Appendix B.3.2),

which further initializes the pins to trigger the PCINT1 interrupt when either has a state change. The PCINT1 interrupt routine will be triggered by any state change, so the routine always checks if there has been a positive edge on one of the three pins, and if so, stores the receive time of that/those pin/pins, see ISR (PCINT1_vect) in main.c, Appendix B.3.2. The reason the PCINT1 interrupt handler detects positive edges, and the ICP (Section 5.3.2.3) detects negative edges, is to ensure that the ICP interrupt handler has copied out the correct timer count before the PCINT1 routine stores it.

5.3.2.6 The main loop

The main loop can be found in main.c, Appendix B.3.2. It implements the state machine depicted in the earlier presented Figure 5.12. STATE_INIT initializes the board and the UART, before changing state to STATE_IDLE. STATE_IDLE waits for received UART messages, and then use them to configure the application for running a test. On receiving the START keyword the state changes to STATE_TEST. STATE_TEST initializes the test by setting up timer and interrupts used. It then waits for Timer0 to trigger output signals to the BeagleBone. After these are sent, it waits for all the configured channels to get a response. If it does not get a response until the next set of output signals is meant to be sent, it will abort the test. It stores the time interval/s in the histogram, before it starts over and wait for Timer0 to trigger a new round of output signals. This loop continues until it has run the configured amount of tests, it then returns to STATE_IDLE after printing the histogram and disabling timers and interrupts. The exception to this code execution is if the free running flag is set, then it will automatically send a new set of output signal when it has received a response on all the channels, thus not using Timer0.

5.4 Discussion on the creation of the External Response Tester cape

The ERT cape was created in a two step process. First we created a prototype on the BeagleBoard Breadbord, before manufacturing and soldering the final board. A limitation of the BreadBoard is its limited size, and thus lack of through-holes. It can only support packages which are 14 pins wide, limiting the available MCUs when

prototyping. It would in hindsight have been both more efficient and provided more options prototyping on a completely separate breadboard. This would have allowed a more complex circuit to be built, and most sizing constraints could have been solved by using other packages for the PCB design. The BeagleBoard BreadBoard did however, give us a good indication to what a final product may look like.
The schematics of the prototype was used to create a PCB design, a process which was completely new to us. Nonetheless, using an intuitive tool like the Altium Designer facilitated the design of a board. It is clear to us that designing simple PCB designs without any high-speed requirements can be performed by most with some engineering experience. It was in fact so straightforward that this project should have spent less time on the breadboard, and instead more time to iterate on the cape design. There were features the cape should have had, that was not needed when prototyping on the breadboard, and thus not implemented in the final design. Prototyping was done only with the GNU/Linux operating system, which had no use for the UART header. This is however used for booting both QNX and FreeRTOS. Support for extending this header through the cape would have been beneficial, together with adjusting the mechanical layer of the production files such that top the board did not interfere with Ethernet connector. A workaround for the missing header support is to not plug the cape on top of the BeagleBone. However, disregarding these minor faults in the design, the ERT performed perfectly, delivering results like the prototype example in Section 5.2.3.1.

6 GNU/Linux on the BeagleBone

This chapter will show how GNU/Linux was utilized on the BeagleBone platform. Both how it was set up, and the programming, which followed to perform tests both internally and externally with the ERT cape. Appendix B.2.1 describes how to boot and log into the distributions/kernels compiled in this implemention.

6.1 Boot process

When the BeagleBone's AM3358 processor powers up it starts loading a program from one of multiple sources, depending on external triggers. The Linux bootloader is set up such that holding the boot button while powering up will make the board boot from the SD card.

6.1.1 Bootloader

A bootloader is a program that loads an operating system or some other software for the computer. The BeagleBone depends on u-boot, the Universal Boot Loader for this task. By default, the processor will boot from the MMC1 interface first (this is the onboard eMMC), followed by MMC0 (MicroSD), UART0 and USB0. However, by pressing the boot switch we can bypass the eMMC and boot straight from the MicroSD card. When the processor accesses the card it starts a three step boot process. The MicroSD card has to be a FAT32/16 partition, which contains at least three files, one for each boot stage. The stage 1 bootloader is the MLO file which is the X-loader, this is provided by the board manufacturer Texas Instruments. This fits entirely on on-chip memory and configures the external memory such that a more advanced bootloader can be run. This is the u-boot.bin file, which is the stage 2 bootloader u-boot. This is a more complex bootloader, which performs several

initialization task before it starts the Linux kernel. The last file to be loaded is the uImage, which is the Linux kernel loaded by u-boot. The Linux kernel mounts the Linux root filesystem and the OS system is started, see Figure 6.1.

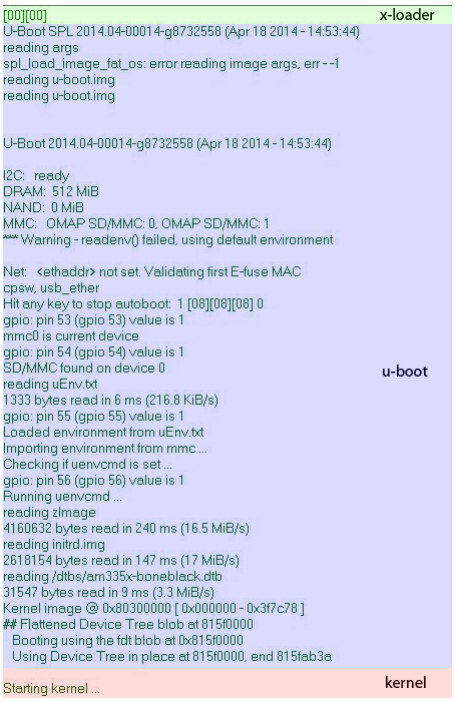

Figure 6.1: The x-loader loads u-boot, which again loads the kernel.

6.1.2 SD card

The SD card has to be formatted to FAT32/16 to be readable by the BeagleBone. The X-loader (MLO file) has to be copied first to a freshly formatted card. After this, the u-boot and kernel binaries can be copied. More complex distributions with package managers often comes with scripts which sets up and partitions the SD card for the developer.

6.2 Debian GNU/Linux

The Debian project[1] combines GNU tools, the Linux kernel, and other important free software to form a software distribution called Debian GNU/Linux. It includes a package manager with an extremely large user base and frequent upgrades. In this project two Debian GNU/Linux distributions are used, one with a kernel compiled with basic preemption and the other compiled with the PREEMPT_RT patch. Both kernels have the Debian BeagleBoardDebian[2] distribution on top.

6.3 Compiling the Linux kernel

There exists a rather wide range of precompiled GNU/Linux images for the BeagleBone Black. However, there exists none with the PREEMPT_RT patched Linux kernel. There are some useful repositories which enabled easy compilation of the basic preemption version of Linux. This makes the kernel preemptible to some degree, but not fully preemptible as we are aiming for with the PREEMPT_RT patch. Therefore, we concluded that there were a need to compile a kernel image ourself. The process of compiling a Linux kernel can be comprised into a few steps:

1. Grab the toolchain needed for the selected platform, for example the GCC ARM Cross Toolchain.

2. Grab a version of the Linux kernel via git from http://kernel.org or some other repository like Linus Torvalds Linux.

3. Apply patches to the kernel for selected platform (optional).

4. Add extra drivers and device trees for wanted features (optional).

5. Configure the kernel, run *Make menuconfig* or equivalent, to browse and toggle kernel and driver options.

6. Compile the kernel (*Make)*, modules, firmware and device trees (*Make modules_install* and *Make install*).

[1] http://www.debian.org/intro/about
[2] BeagleBoardDebian wiki [4]

We based our kernel of one of the Linux 3.14 repositories, more specifically with Robert C Nelsons BeagleBone patch repository[3]. This repository contains build scripts to ease the building process and it also contains a PREEMPT_RT patch. The PREEMPT_RT patch removed from the patching script due to some errors in its implementation, however, after correcting these errors in the patch we managed to apply it successfully. After applying RT patch file located under *patches/rt/* the option to use the PREEMPT_RT preemption mode became visible when configuring pre-build, see Figure 6.2. The build was then carried through and a image together

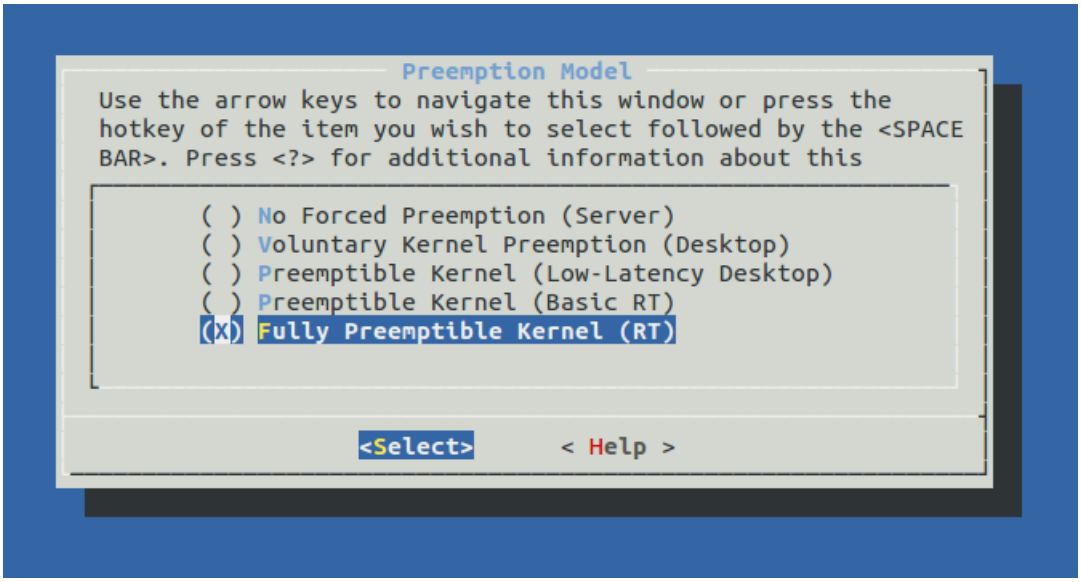

Figure 6.2: This image shows the PREEMPT_RT option when configuring.

with additional device tree and modules files was created. It also successfully booted on the BeagleBone, see Figure 6.3. However, compiling a working kernel is one thing, to make it have the necessary features is something else. The Linux 3.14 repository for the BeagleBone was the only one we managed to compile with PREEMPT_RT, however, a lot of features were lacking as most patches was not ported. A possible explanation for these difficulties was the 3.14 kernel version as late as April 26, 2014. Errors included no driver to configure the pinmux from device trees. Uart nr.4, which is used to communicate with the cape uses was not enabled, and any attempt to interract with the PRU subsystem threw a bus error.

[3]Robert C Nelson, build script repository [20]

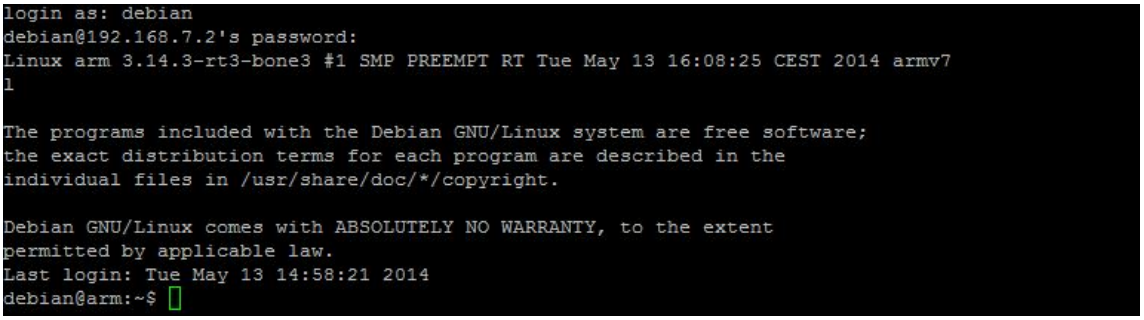

Figure 6.3: Successful boot of PREEMPT_RT kernel for BeagleBone Black.

6.3.1 Configure pinmux driver

It is possible to configure the pinmux in runtime through software, but it requires memory mapping and knowledge of how the pinmux works. We wanted the pinmux to be set up through the device tree such that users (students), who will use the platform can focus on the subjects most relevant to embedded and real-time topics. Earlier versions of the BeagleBone Linux kernel like the 3.8 version had a pinmux helper driver. Implementing this to our kernel was straightforward. The kernel version 3.8 branch of Robert Nelsons repository included patch files which implements a pinmux helper driver, it is located at *linux-dev/patches/not-capebus/0100-Pinmux-helper-driver.patch*. Thus, by following the instructions of this patch file it was successfully ported it to the 3.14 kernel. First, a c file containing the driver is put in the *drivers* folder of the kernel source. We could have placed it in any folder inside the *drivers* folder, however, choosing a logical location will help later in the configuration. It was placed it in, */drivers/misc/pinmux_helper/bone-pinmux-helper.c*. This c file is then added to the closest makefile in the hierarchy, in this case, */drivers/misc/Makefile*. Further the configure option (to include it in the kernel build), is added to *Kconfig* file in the same folder */drivers/misc*. It is this *Kconfig* file which is compiled to create the configure menu of the kernel, and thus, from our folder placement the pinmux driver appeared under the misc devices option, see Figure 6.4.

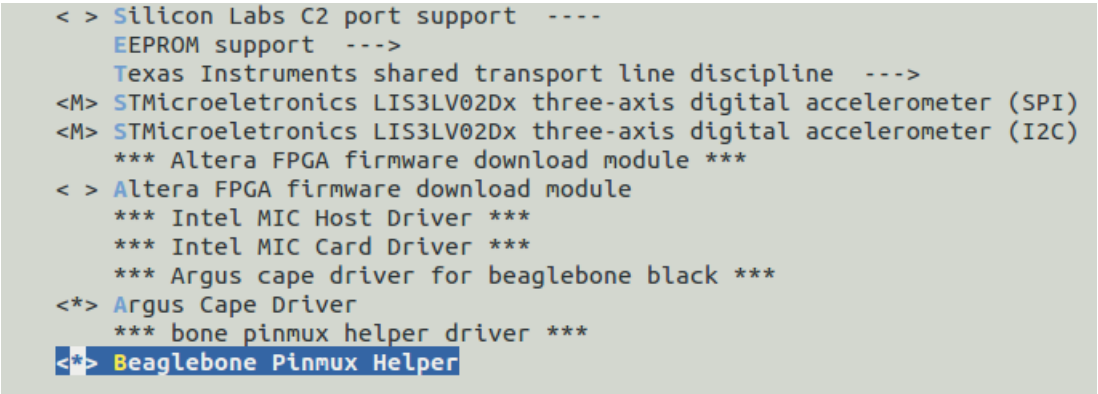

Figure 6.4: This figure depicts the option to include the pinmux helper driver in the configure menu.

6.3.2 Device Tree setup

Device trees was presented in Section 4.1.8. This section describes how device trees is used in a kernel implementation. After adding the needed pinmux driver we could add to the device tree, to configure GPIOs, and enable the PRU and UART. In earlier versions of the kernel there where a driver named capemanager which enabled device tree overlays to be loaded at runtime. However, as this was lacking in kernel version 3.14 and we had to compile the kernel anyway, it was decided to implement an additions to the device tree at boot time. At boot time the system is configured to load the *AM335x-boneblack.dtb* file as the device tree. This is configured within the *uEnv.txt* file located with u-boot on the SD card. The source file of this device tree blob (.dtb) is located at *KERNEL/arch/arm/boot/dts/*. Several changes were made, as explained in the following.

am335x-boneblack.dts

This is the source file for *am335x-boneblack.dtb*, which is compiled each time the kernel was rebuilt. The following changes were made. First, we added am335x-boneblack-ttyO4.dtsi and am335x-ttk4147-cape.dtsi (as seen in Figure 6.5) to the include list of the file. These files, which we will elaborated on later, enables the PRU subsystem and UART4, and they also configure the pinmux as needed. Second, we also disabled the HDMI label in this file (status = "disabled"), thus freeing 20 pins on the pinmux, see Figure 6.6.

```
10  #include "am33xx.dtsi"
11  #include "am335x-bone-common.dtsi"
12  #include "am335x-boneblack-ttyO4.dtsi"
13  #include "am335x-ttk4147-cape.dtsi"
```

Figure 6.5: Includes in the am335x-boneblack.dts file.

```
90  hdmi {
91      compatible = "ti,tilcdc,slave";
92      i2c = <&i2c0>;
93      pinctrl-names = "default", "off";
94      pinctrl-0 = <&nxp_hdmi_bonelt_pins>;
95      pinctrl-1 = <&nxp_hdmi_bonelt_off_pins>;
96      status = "disabled";
97
98      panel-info {
99          bpp = <16>;
100         ac-bias = <255>;
101         ac-bias-intrpt = <0>;
102         dma-burst-sz = <16>;
103         fdd = <16>;
104         sync-edge = <1>;
105         sync-ctrl = <1>;
106         raster-order = <0>;
107         fifo-th = <0>;
108         invert-pxl-clk;
109     };
110 };
```

Figure 6.6: The modified HDMI label in the am335x-boneblack.dts file.

am33xx.dtsi

Am33xx.dtsi is included in AM335x-boneblack.dts, it includes pin setup, and device and interface setup. Further we added a label named *pruss* (see Figure 6.7) under *ocp*, which stands for On Chip Peripheral. It initiates the compatible key, which binds the correct device driver (compatible = "ti,pruss-v2";), it also sets the address range used by the device (reg = <0x4a300000 0x080000>;). In addition interrupts for the device are configured. For now it is disabled (status = "disabled"), as we do enable of the devices we need in the a later include file.

am335x-boneblack-ttyO4.dtsi

Am335x-boneblack-ttyO4.dtsi is included in AM335x-boneblack.dts, it extends the UART4 label, which was defined in the am33xx.dtsi file. It binds it to the UART4 pins (also defined in the am33xx.dtsi), before enables the device, see Figure 6.8.

am335x-ttk4147-cape.dtsi

Figure 6.9 shows the AM335x-ttk4147.dtsi file. First we, created two labels under the extended am33xx_pinmux label, which are defined by default in the am33xx.dtsi file.

```
402 ▼       pruss: pruss@4a300000 {
403             compatible = "ti,pruss-v2";
404             ti,hwmods = "pruss";
405             ti,deassert-hard-reset = "pruss", "pruss";
406             reg = <0x4a300000 0x080000>;
407             ti,pintc-offset = <0x20000>;
408             interrupt-parent = <&intc>;
409             status = "disabled";
410             interrupts = <20 21 22 23 24 25 26 27>;
411         };
```

Figure 6.7: The pruss label.

```
1   &uart4 {
2       pinctrl-names = "default";
3       pinctrl-0 = <&uart4_pins>;
4
5       status = "okay";
6   };
```

Figure 6.8: The AM335x-boneblack-ttyO4.dtsi device tree file.

The first new label contains pins for GPIO (resp_pins) and the second contains pins for the PRU (pru_gpio_pins). The first hexadecimal number when assigning a pin is the pinmux address, the second sets the operating mode of pinmux and configures any pullup/pulldown circuit. The next fragment implements a label, *gpio_helper*, under ocp, which is compatible with the bone-pinmux-helper driver, which we implemented in the last subsection, Section 6.3.1. The next sections extends already created labels. The first extends and enables the helper label we just created (status = "okay";), as well as binding the GPIO pins to the label (pinctrl-0 = <&resp_pins>;). The last extends *pruss*, a label we just created in AM335x-bone.dtsi, and binds the PRU pins (pru_gpio_pins) to it before enabling the device.

```
&am33xx_pinmux {
    pru_gpio_pins: pinmux_pru_gpio_pins {
        pinctrl-single,pins = <
            0x03c 0x16
            0x043 0x16
            0x0b8 0x26  /* pr1_pru1_pru_r31_6 MODE6 */
            0x0b0 0x26  /* pr1_pru1_pru_r31_4 MODE6 */
            0x0a8 0x26  /* pr1_pru1_pru_r31_2 MODE6 */
            0x0bc 0x15  /* pr1_pru1_pru_r30_7 MODE5 */
            0x0b4 0x15  /* pr1_pru1_pru_r30_5 MODE5 */
            0x0ac 0x15  /* pr1_pru1_pru_r30_3 MODE5 */
        >;
    };
    resp_pins: resp_cape_pins {
        pinctrl-single,pins = <
            0x15c 0x17  /* gpio0_5 P9_17 OUTPUT MODE7 pullup */
            0x158 0x17  /* gpio0_4 P9_18  */
            0x154 0x17  /* gpio0_3 P9_21  */
            0x150 0x27  /* gpio0_2 P9_22 INPUT MODE7 */
            0x180 0x27  /* gpio0_14 P9_26  */
            0x184 0x27  /* gpio0_15 P9_24  */

           /* OUTPUT  GPIO(mode7) 0x07 pulldown, 0x17 pullup, 0x?f no pullup/down */
           /* INPUT   GPIO(mode7) 0x27 pulldown, 0x37 pullup, 0x?f no pullup/down */
        >;
    };
};

/ {
    ocp {
        gpio_helper: gpiohelper {
            compatible = "bone-pinmux-helper";
            status = "disabled";
        };
    };
};

&gpio_helper {
        pinctrl-names = "default";
        pinctrl-0 = <&resp_pins>;
        status = "okay";
};

&pruss  {
        status = "okay";
        pinctrl-names = "default";
        pinctrl-0 = <&pru_gpio_pins>;
};
```

Figure 6.9: The AM335x-ttk4147.dtsi device tree file.

6.3.3 General configuration settings

In the last subsections we have presented how the kernel configurations have been used to include a driver and further to choose the preemption model. Other modifications were also made to the default configuration set by Robert C Nelsons build scripts. The Texas Instruments PRUSS driver was added, see in Figure 6.10. Besides, the timer frequency was set to its maximum, 1000Hz, see Figure 6.11.

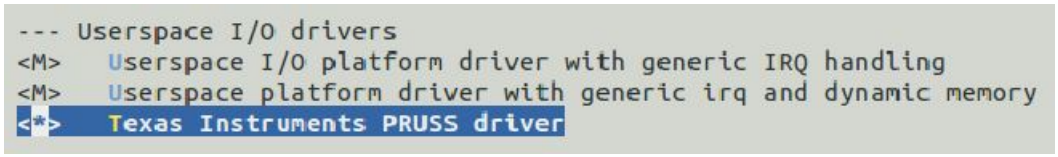

Figure 6.10: Kernel configuration, adding the PRUSS driver.

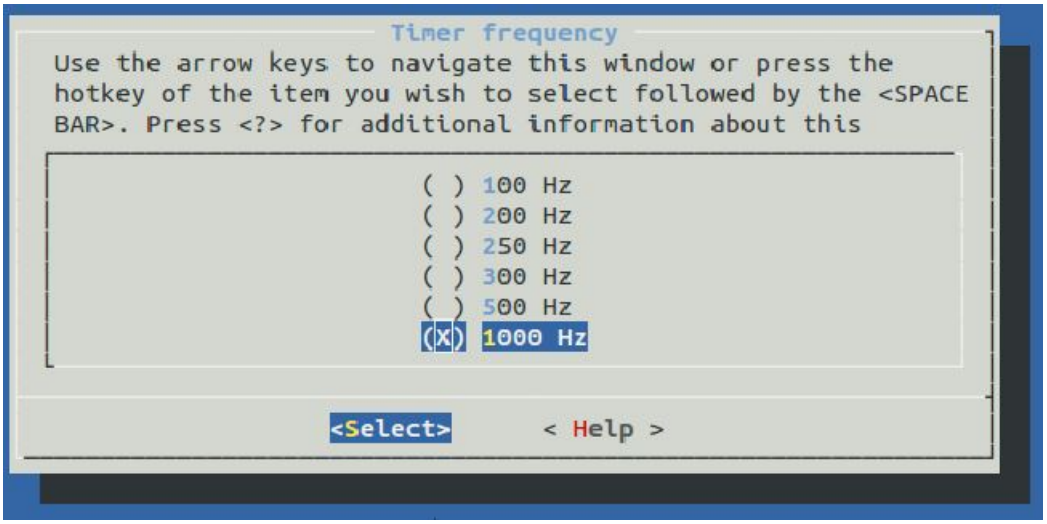

Figure 6.11: Kernel configuration, setting maximum timer frequency.

6.3.4 Further optimizations

After all the above changes were implemented, the kernel was compiled and testing began. Results (which will be presented later) showed that a kernel with no load performed slower then a kernel on full load. The most obvious thought was that the processor went into some kind of sleep mode, which made it react slower in response tests. Therefore, a new kernel was compiled that disabled the power management features we could locate in the kernel configuration. To allow changes in these features the Typical OMAP configuration was unchecked, see Figure 6.12.

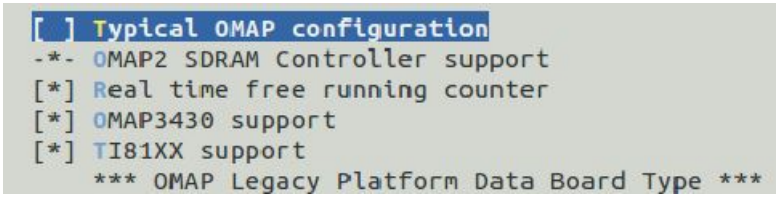

Figure 6.12: Kernel configuration, turning of Typical OMAP configuration.

The following options were then set:

- HZ_PERIODIC = y.
- PM_RUNTIME = n.
- PM_AUTOSLEEP = n.

HZ_PERIODIC keeps the scheduling ticks running at a constant rate, even when it is not needed, lowering overhead for handling events by an idle system. PM_RUNTIME is turned off to keep I/O devices from entering into an energy-saving state after a period of inactivity. PM_AUTOSLEEP is turned off to keep the kernel from triggering a global sleep state whenever there are no active wakeup sources.

6.4 Developing on the BeagleBone Black

After the successful compilation of the kernels, that had the features we required, the actual code development could begin. The code was developed in a structured manner to facilitate later use. Development was done through cross-compilation with Eclipse on a host computer running Ubuntu.

6.4.1 Cross-compilation with Eclipse

Eclipse is IDE used to develop applications for multiple programming languages, including C and C++. It is free and works on multiple platforms. We used Eclipse on a host computer running Ubuntu to cross-compile for the BeagleBone Black. Cross-compilation is the act of compiling code for one computer system, often known as the target, on a different system called the host. What follows is a point by point instruction on how to set up cross-compilation for the BeagleBone:

- Grap an ARM gcc compiler at: https://releases.linaro.org/latest/components/toolchain/binaries/.
- Extract the compiler to a folder of ones own choosing, in this case *bbb-compiler/*.
- In Eclipse, create a new c project and choose Linux GCC.

- Open project properties, and go to C/C++ Build and then to Settings.
- At GCC C Compiler, the command field should be set to: *bbb-compiler /gcc-linaro-arm-linux-gnueabihf-(version_nr_and_date)_linux/bin/ arm-linux-gnueabihf-gcc*.
- At Linker, the command field should be set to: *bbb-compiler/gcc-linaro -arm-linux-gnueabihf-(version_nr_and_date)_linux/bin/arm-linux-gnueabihf-gcc*.
- At Assambler, the command field should be set to: *bbb-compiler/gcc-linaro -arm-linux-gnueabihf-(version_nr_and_date)_linux/bin/arm-linux-gnueabihf-as*.

After compiling the project the build folder will now contain a file ,which will run on the BeagleBone Black. Files can either transfered the files manually using *scp* or similar, or by remotely running and debugging via SSH and *gdb*. There are several guides available on the Internet, and hence we do not detail this further.

6.4.2 BeagleBone pinout for interracting with the ERT cape

Figure 6.13 illustrates which pins are utilized on the BeagleBone Black headers.

6.4.3 GPIO test application

The test applications purpose is to offer the user benchmarking tool, both internally and in conjunction with the external testing cape. The application can run a response test with the ERT cape, or run a maximum pin toggle test. These tests will be explained further in the next subsections. It also has the ability set up a thread, which reads and outputs the serial port at */dev/ttyO4*. This is UART4 which is connected to the ERT cape. The Eclipse project files are appended to the project and detailed in Appendix B.1.

6.4.3.1 The response test

The purpose of the response test is to measure the time it takes for the BeagleBone Black to respond to some external source. When choosing to run a response test, the application will set up the three input pins, and write responses on their state changes as fast as possible to the respective output pins. The user can choose

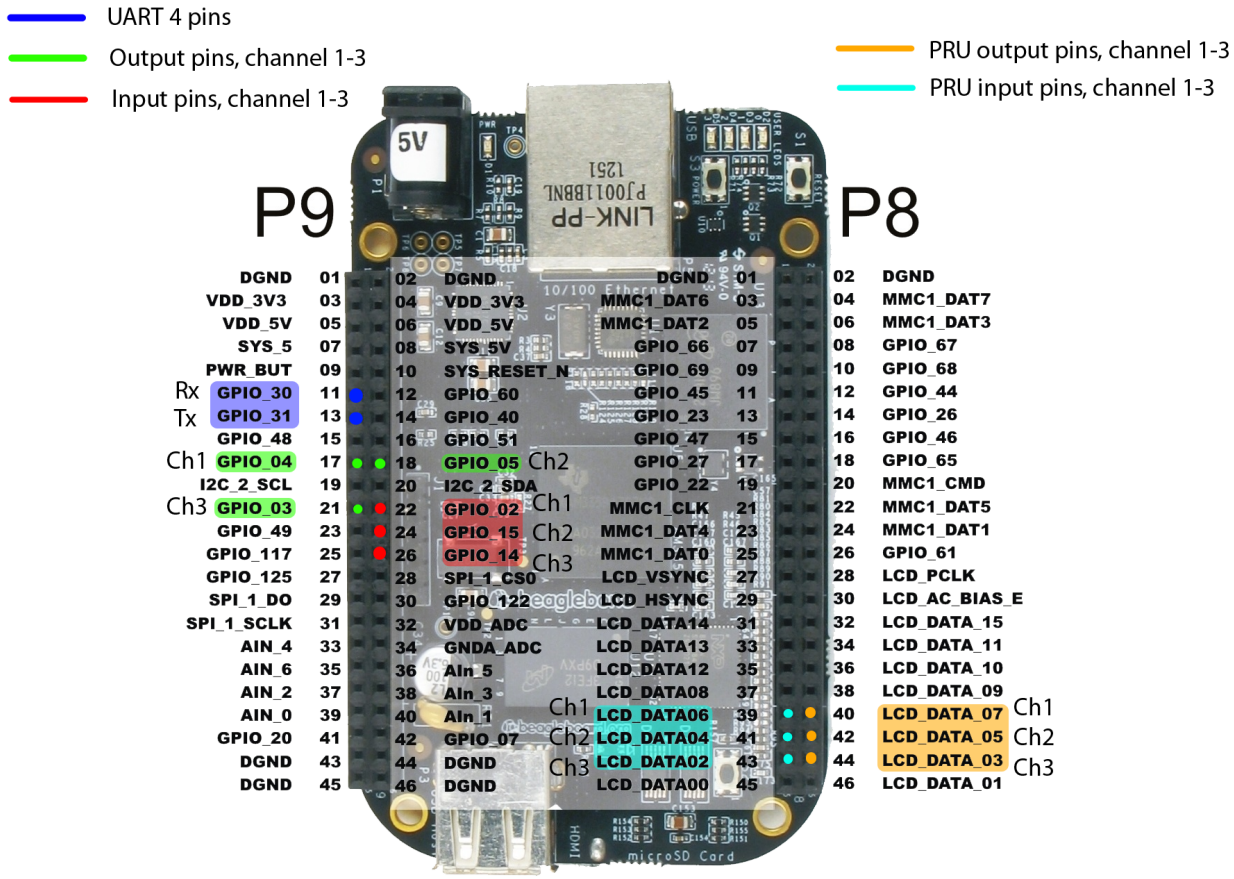

Figure 6.13: This Figure depicts the pins set up in the manipulation of the Device Tree, which are utilized by the BeagleBone applications.

between running busy-wait threads or using Linux polling, as well as choosing the pin control interface. The interfaces available are the sysfs GPIO driver, and direct memory mapped control. Finally the, the user can choose to run the test thread at maximum priority.

6.4.3.2 The maximum single pin toggle test

The purpose of this test is to evaluate the I/O latency, as explained in 2.2, by measuring the time interval of state changes when toggling as fast as possible. When choosing to run a toggle test, the application will set up the channel 1 output pin, and toggle it on and off as quickly as possible. The user can choose between using the sysfs GPIO driver, or the memory mapped GPIO control. The user can also in this

```
root@arm:/opt/bbb# ./BBB_new -h

        /*****************************/
        /*******GPIO TEST PROGRAM*******/
        /*****************************/
        /*****************************/
        /*********** VER 1.0 ***********/
        /*****************************/

Valid arguments:
-pmax, run test threads at maximum priority
-resp, external response test
        -bw, busy-wait
                -mem, use the memory mapped interface
                -sysf, use the sysfs interface
        -poll, polling
-togl, max output toggle test on gpio channel 1
        -mem, use the memory mapped interface
        -sysf, use the sysfs interface
-ser, serial port tty04 outputs to console
```

Figure 6.14: This figure shows the user help of the GPIO test application.

case choose to run the at maximum priority.

6.4.3.3 The code

gpiodriver.c

The simplest GPIO interface is the default BeagleBone GPIO driver through the virtual file system sysfs, previously described in Section 4.1.9.1. To utilise this through a C application we applied the userspace gpiodriver written by RidgeRun, modified by Derek Molloy, before further modifications were done by this project. This includes a set of functions to export/unexport, set direction, value and edge triggering through the sysfs interface.

gpiomemdriver.c

The other interface used to control GPIOs from userspace utilizes memory mapping. Memory mapping was previously described in Section 4.1.9.2. Gpiomemdriver includes functions to map GPIO banks to memory, and also the functions need then interact with this. It includes setting, clearing, configuring and reading pins.

outputdriver.c

Outputdriver is a wrapper that contains functions which generalize the use of the gpiodriver and the gpiodrivermem functions for output purposes.

inputdriver.c

Inputdriver is also a wrapper, however, it includes functions for input purposes. Apart from a generalized read function, it also includes a function, which initializes

the polling thread in the same file. This function accepts a callback function as argument. This callback function will be called by the polling thread every time it detects a negative edge on one of the input pins. This is to give the programmer flexibility in how it chooses to handle the event. For this application the callback function was implemented such that it toggles the corresponding output pin as fast as possible, with the interface chosen by the user, it is located in *main.c*. Polling was previously presented in Section 4.1.9.4.

serial.c

The serial file includes an initialization function, which sets up the UART4 for communication with the ERT cape. It assigns this to a global serial file descriptor variable, which is declared in its header file.

It also includes the function that is run as its own thread and prints all messages received on the UART.

main.c

Main includes the main function, which parses input arguments and then call the needed functions from *serial.c*, and *input/outputdriver.c* to perform the tests specified by the user.

It also includes the thread function run by the busy-wait response test, and the callback function, which is passed on to the inputdriver for polling response tests.

6.4.4 Developing for the PRU subsystem

The PRU compiler *pasm* can be downloaded together with both documentation, examples and a userspace driver can be downloaded from github[4]. The applications developed for the PRU subsystem was coded in assembly, and the instructions available can be found at the wiki page of Texas Instruments [5].

6.4.4.1 Starting and loading programs to the PRU

The PRU subsystem is by default turned of when booting one of the Linux kernels compiled in this project. A script called *startpru.sh* was used to manually turn it on, it is appended and referred to in Appendix B.1. If forgotten, Linux will throw a bus

[4] PRU package [18].

[5] PRU Assembly Instructions http://processors.wiki.ti.com/index.php/PRU_Assembly_Instructions

error when trying to load a new program. To load a compiled program to the PRU use the *loadPruBin* program located in the *pru* directory, appended to the project and referenced in the Appendix B.1.

6.4.4.2 The PRU single pin maximum toggle test

The SoC PRU was presented in Section 3.2.2. The PRU runs at 200MHz with most instructions deterministically finishing each clock cycle. This equals 5ns per instruction and a possible pin toggling rate of 100MHz. The logic analyser used to measure the toggling rate (presented in Section 3.4) operates at a maximum sampling frequency of 24MHz, thus, the PRU had to be delayed to be measured correctly. The test application loops therefore through a total of 41 instructions between each toggle. The PRU toggle code *toggleprutest.p* is appended to the project and detailed in Appendix B.1.

6.4.4.3 PRU busy-wait test

To enable testing in conjunction with the ERT cape, we implemented a simple busy-wait scheme on three channels. A read loop continuously checks the state of the three input pins. If one of the pins has a state change from high to low, the PRU branches to respond on the corresponding channels output pin. The hope is that the PRU will be so quick that it won't miss any activity from the much slower AVR ATmega MCU on the ERT cape. Since each instruction takes 5ns to perform and the maximum code gap between each channel read is 52 instructions, the PRU should in theory respond to any signal kept low for 0.1μs. The PRU busy-wait code *busywaittest.p* is appended to the project and detailed in Appendix B.1.

6.5 Benchmarking tools

The following benchmarking tools were acquired from external sources as opposed to the above mentioned applications.

6.5.1 Cyclictest

Cyclictest[6] is a high resolution test program for measuring the time between an event occurs, until is handles. It performs similar tasks as ERT cape. However, by contrast it does everything internally in pure software.

6.5.2 Stress

Stress[7] is a simple workload generator, it can be used to impose CPU, memory, I/O and disk stress on the system.

6.5.3 Hackbench

Hackbench[8] can be used as a benchmark or a stress test for the Linux kernel schedular. It creates a specified number of schedulable entities which will communicate with each other. It then measures the time interval for sending data back and forth.

6.6 Discussion on the implementation of GNU/Linux

This chapter presented the implementation of GNU/Linux, and the applications that would run on it. We also showed how to use and program the PRU subsystem through GNU/Linux. The difficulty of compiling a Linux kernel depends on the circumstances, compiling for a popular platform that provides build scripts makes the process straightforward and doable for anyone with some GNU/Linux experience. Difficulties occur, however, when there is a need for extra features, see step 4 in Section 6.3. A few days is a reasonable estimate to get a grip on both the kernel file structure, and how to work with device trees. A factor, which proved very useful, was the BeagleBone community. Community driven development platforms provides the user with immense amounts of resources, this project especially utilized Stack Overflow[9] and the BeagleBoard section of Google

[6] Cyclictest [11].
[7] Stress, workload generator [30].
[8] Hackbench [24].
[9] http://stackoverflow.com/tour

Groups[10]. Official documentation is often cumbersome, and challenging topics like device trees are best taught, according to out experience, through practical examples. The task of creating the test application, after we managed to compile the fully functional kernels, was a rather simple task. We did try keep most of the files generalized so they would be helpful to later users, wrapping the tests into a single program instead of several different, was also done with the same purpose. One undertaking, which was never completed, was the serial interface with the ERT cape. The commands to set up and perform a test were performed manually in a terminal with *echo*, this was due to time constraints because of more important tasks.
Developing for the PRU subsystem was, in contrary to the test application, more challenging. The idea of having a 100% predictable subsystem, delivering two 32-bit cores running at 200MHz, is very exciting from a real-time programmer's perspective. There will hopefully be released a C compiler in the future, as the potential user-base then would increase tenfold. Only a limited amount of processors presently include the subsystem, and even a large community as the BeagleBone lack a lot of resources to exploit the full potential of it. This project only touched the surface of the PRU, which is reflected in the two very simple tests programs presented in Section 6.4.4. This is also partly due to the large amount of topics we investigated, a project work fully dedicated to the PRU potential is thus an interesting topic for further work.
We will study the results created from the GNU/Linux test application in Chapter 8, and they will be further discussed in Chapter 9.

[10] https://groups.google.com/forum/#!forum/beagleboard

7 | FreeRTOS and additional OS support

The test results which will be produced by the GNU/Linux distributions will have less value if not compared with other OS. This chapter will show the implementation of the other two OS, namely FreeRTOS and QNX. For how to boot the finished products of this implementation on the BeagleBone Black, see Appendix B.2.2.

7.1 QNX

QNX Software Systems work with Texas Instruments and support the AM335x processor. They deliver a Board Support Package (BSP) with source code and pre-built images. They also provide MLO and u-boot binaries to ensure easy creation of a bootable SD card. All this together with a User Guide can be found at QNX wiki pages[1]. The catch with using QNX is that it requires a license, to install and use the QNX Software Development Platform. However, the platform is then available to perform the exercises of the TTK4147 course which involves QNX. Figure 7.1 shows QNX booted on the BeagleBone Black.

7.2 FreeRTOS

To enable cross compatibility with the exercises of the TTK4147 course we also needed FreeRTOS support. There exists multiple repositories openly available[2]. We tested several, and none of them were close to functioning, only providing barebone

[1] BeagleBone Black QNX BSP wiki [28]
[2] BeagleBone FreeRTOS repositories[8],[10],[9]

```
.section .startup,"ax"
         .code 32
         .align 0

    b     _start                    /* reset - _start          */
    ldr   pc, _undf                 /* undefined - _undf       */
    ldr   pc, _swi                  /* SWI - _swi              */
    ldr   pc, _pabt                 /* program abort - _pabt   */
    ldr   pc, _dabt                 /* data abort - _dabt      */
    nop                             /* reserved                */
    ldr   pc, [pc,#-0xFF0]          /* IRQ - read the VIC      */
    ldr   pc, _fiq                  /* FIQ - _fiq              */

_undf:  .word 0x4020FFE4                /* undefined               */
_swi:   .word 0x4020FFE8          /* SWI                     */
_pabt:  .word 0x4020FFEC                /* program abort           */
_dabt:  .word 0x4020FFF0                /* data abort              */
_fiq:   .word 0x4020FFFc                 /* FIQ                    */

__undf: b     .                         /* undefined               */
__pabt: b     .                         /* program abort           */
__dabt: b     .                         /* data abort              */
__fiq:  b     .                         /* FIQ                     */
```

Figure 7.1: Jerrings repository, exception vector table set up.

gpio programming without any FreeRTOS features. The systems lacked several features including, there were no configured system ticks, registers were wrong and the exception vector table misplaced causing the board to freeze if any exception or interrupt was triggered.

7.2.1 Porting FreeRTOS to AM335x

To enable use of FreeRTOS we had to set it up ourself. The BeagleBone repository of Jerrings [8] was used as a base for our port. The AM335x Technical Reference Manual (TRM) is the bible when porting, it details the integration, the environment, the functional description, and the programming models for each peripheral and subsystem in the device.

7.2.1.1 Configuring the tick interrupt

Several of the FreeRTOS source files are cross platform compatible, those which are not, are located under *Source\portable\"Compiler"\"Platform"*. The FreeRTOS source files for this project are appended and their location is referenced in Appendix B.1. One of the files in this folder is the *port.c* file. This file includes a function *prvSetupTimerInterrupt*, which is resposible for configuring and starting the tick interrupt, as well as ensuring that the interrupt controller is ready to receive it. The interrupt controller module was set up by performing a soft reset, setting it to free

running mode, and the interrupt threshold was disabled to ensure capturing of all interrupts. The DMTIMER module contains 8 programmable timers. DMTIMER2 was set up to produce a tick every 1ms using a compare match register. This triggers interrupt number 68 on the processor, which subsequently was unmasked from the interrupt mask register.

7.2.1.2 ARM exception vector

When compiling and booting after configuring the tick interrupt, we could observe that the tick interrupt fired at the correct rate, however, when trying to enter the assigned *vIRQHandler*, the board froze. The same could be observed if we manually triggered a SWI, which should have run the SWI handler *vPortYieldProcessor*. This led us to suspect that the exception vector was unavailable or incorrectly set up. Exception vectors was introduced in Section 3.2.1.1. When compiling FreeRTOS we use a linker script *Demo\AM3359_BeagleBone_GCC\omap3-ram.ld*, it links together the different parts of compiled code, and specifies where it should be put in memory. It is this script which makes the boot command in B.2.2 start the OS at memory address 0x80500000 instead of 0x81000000 as QNX. This script also specifies that the entry point for the OS is not main, but instead a section in the assembly coded *boot.s* file, located in the same folder. It is this file which sets up the stacks for each user mode and is responsible for the exception vector. The Jerrings repository had the vector table set up as shown in Figure 7.2.

This code makes the assumption that processor will search for the exception vector table at this section. And further it makes use of handler addresses, 0x4020FFE4 and so on, which could be tampered with. Problems may occur because this code is loaded by u-boot, and if u-boot wants to use exceptions and starts moving or tampering with the default exception vector handler addresses[3]. To ensure that the exception vector table would be correctly set up changes were made, as depicted in Figure 7.3.

Instead of going through the default exception handlers we make the ISR from the FreeRTOS callable in the boot file using the *extern* calls at the top of the file. Then, before branching to main, we change the location were the processor should look for the vector table to a label called *_vector_table*. Finally we placed this label at the top

[3]The default RAM Exception Vectors can be found in Section 26.1.3.2 of the AM335x TRM [14].

```
.section .startup,"ax"
        .code 32
        .align 0

    b     _start                    /* reset - _start          */
    ldr   pc, _undf                 /* undefined - _undf       */
    ldr   pc, _swi                  /* SWI - _swi              */
    ldr   pc, _pabt                 /* program abort - _pabt   */
    ldr   pc, _dabt                 /* data abort - _dabt      */
    nop                             /* reserved                */
    ldr   pc, [pc,#-0xFF0]          /* IRQ - read the VIC      */
    ldr   pc, _fiq                  /* FIQ - _fiq              */

_undf:  .word 0x4020FFE4                /* undefined               */
_swi:   .word 0x4020FFE8          /* SWI                     */
_pabt:  .word 0x4020FFEC                /* program abort           */
_dabt:  .word 0x4020FFF0                /* data abort              */
_fiq:   .word 0x4020FFFc                 /* FIQ                    */

__undf: b     .                         /* undefined               */
__pabt: b     .                         /* program abort           */
__dabt: b     .                         /* data abort              */
__fiq:  b     .                         /* FIQ                     */
```

Figure 7.2: Jerrings repository, exception vector table set up.

of our exception vector table and made it call the handlers directly.

7.3 FreeRTOS test application

A FreeRTOS test application was created to enable comparison between a pure RTOS and GNU/Linux.

7.3.1 GPIO interrupt

The following list details the register manipulation to enable GPIO interrupts for interacting with the ERT cape.

- Enable the clock of the GPIO 0 bank, Section 8.1.12.2.3 in the TRM.
- Set the pinmux to set three input pins, Section 9.3.1.50 in the TRM.
- Enable interrupts on those specific pins, Section 25.4.1.8 in the TRM.
- Enable falling-edge detection on the same pins, Section 25.4.1.22 in the TRM.
- Unmask the GPIO 0 bank interrupt (nr.45), Section 6.5.1.38 in the TRM.
- Add detection of the specific interrupt in the interrupt handler, then mask out which pin on the bank created the interrupt, Section 25.4.1.6 in the TRM.

```
    .extern vIRQHandler
    .extern vPortYieldProcessor
    .extern DATA_ABORT
.....
......
        /* Set V=0 in CP15 SCTRL register - for VBAR to point to vector */
       mrc    p15, 0, r0, c1, c0, 0    @ Read CP15 SCTRL Register
        bic    r0, #(1 << 13)       @ V = 0
        mcr    p15, 0, r0, c1, c0, 0    @ Write CP15 SCTRL Register

        /* Set vector address in CP15 VBAR register */
        ldr     r0, =_vector_table
        mcr     p15, 0, r0, c12, c0, 0  @Set VBAR
.....
......
.section .startup,"ax"
         .code 32
         .align 0
_vector_table:  b     _start    /* reset - _start       */
        ldr   pc, _undf         /* undefined - _undf        */
        ldr   pc, _swi          /* SWI - _swi               */
        ldr   pc, _pabt         /* program abort - _pabt    */
        ldr   pc, _dabt         /* data abort - _dabt       */
        nop                     /* reserved             */
        ldr   pc, _irq          /* IRQ - read the VIC       */
        ldr   pc, _fiq          /* FIQ - _fiq               */

_undf:  .word __undf
_swi:   .word vPortYieldProcessor
_pabt:  .word __pabt
_dabt:  .word DATA_ABORT
_irq:   .word vIRQHandler
_fiq:   .word __swi

__undf: b     .                         /* undefined                */
__pabt: b     .                         /* program abort            */
__dabt: b     .                         /* data abort               */
__fiq:  b     .                         /* FIQ                      */
__irq:  b     .                         /* data abort               */
__swi:  b     .                         /* FIQ                      */
```

Figure 7.3: The altered code sections to correctly set up the exception vector.

The first four items on the list were implemented under *prvSetupHardware* in *main.c*. The unmasking of the interrupt was added to *port.c*, next to the unmasking of the tick interrupt. At last, the main ISR handler *vIRQHandler* in *portISR.c* was changed to set three flags, one for each input channel, in case of an interrupt on those specific pins.

7.3.2 Initialization of output GPIO and UART0

The function *prvSetupHardware* also sets up three GPIO as output, and then initialize them as high. These three pins together with the input pins are pin compatible with the GNU/Linux application, this pinout was earlier illustrated in Section 6.4.2. A limited UART driver for UART0 was also written in *serial.c*, which enables putting strings on the transmitter, thus enabling some debugging.

7.3.3 Tasks

Two tasks were implemented, the first *vBlink* introduces stress to the system. It blinks some LED, however, instead of calling vTaskDelay to sleep, it loops, and thus remaining in the ready or running state. The other task *vRespTask1* loops while continuously checking if one the channel interrupt flags get set. It responds to a flag by toggling the corresponding channel output pin. The *vRespTask1* task also has the possibility of sleeping between each loop, this is done by un-commenting the *vTaskDelayUntil* call at the loop end. *vTaskDelayUntil* is an absolute time sleep call, it accepts a parameter which specified when it last woke up, and goes to sleep for a period relative to this parameter instead of when the task calls the function.

7.4 Discussion on the implementation of additional OS support

Two additional OS, besides to GNU/Linux, were implemented for the BeagleBone Black. QNX Neutrino support was trivial as QNX delivers a BSP for the single-board computer. Therefore, we decided to focus more on FreeRTOS support. When researching this project we found several FreeRTOS for the BeagleBone repositories as mentioned in Section 7.2. One of these were tested early on; the board initialized, launched the binary file, and the LEDs blinked at a steady rate as they were supposed to. This was mistakenly taken as a sign of a fully functioning FreeRTOS. In the GNU/Linux discussion in the last chapter we mentioned how the BeagleBones community is a key feature, however, the above case illustrates the importance verifying community driven resources.
The fact that none of the FreeRTOS ports would work, meant that we had to perform this port ourself, nonetheless, the process of porting FreeRTOS was a very gratifying experience. When working with QNX and GNU/Linux we had taken topics like the bootloader and primary OS initialization for granted. To properly set up FreeRTOS meant working closely with these topics, and thus acquiring a deeper understanding of the AM335x processor. We managed to use timers and interrupts through working with the processor registers, and thus, this also gave a deeper understanding on how the drivers in the GNU/Linux OS for the same platform work, as they perform similar

tasks.
We will look at the results created from the FreeRTOS test application in Chapter 8, and these will be discussed in Chapter 9.

8 Test setup and results

8.1 The maximum single pin toggle test

The following test will benchmark the different methods for GPIO control on the BeagleBone platform. Its purpose is extract information to enable highlighting of any advantages or disadvantages of each method through measuring their I/O latency. I/O latency was presented in Section 2.2.

8.1.1 Setup

The test setup involved the BeagleBone platform and a logic analyser connected to a GPIO pin. The property we set out to benchmark was how fast each interface could toggle a pin, i.e. the pins I/O latency.

8.1.2 Testing

Three interfaces were tested. First, the sysfs virtual Linux file system. Sysfs was presented in Section 4.1.9.1. Second, we used memory mapping through */dev/mem* to gain access to the GPIO module of the BeagleBone. Memory mapping in Linux was presented in Section 4.1.9.2. Last, we used the PRU subsystem to control the pin toggling. The GNU/Linux test application was presented in Section 6.4.3, and the PRU test application in Section 6.4.4.2. The test with the three different interfaces are shown in Figures 8.1, 8.2 and 8.3. Note that following each figure the pin toggling rate is included below each caption text.

8.1.3 Results

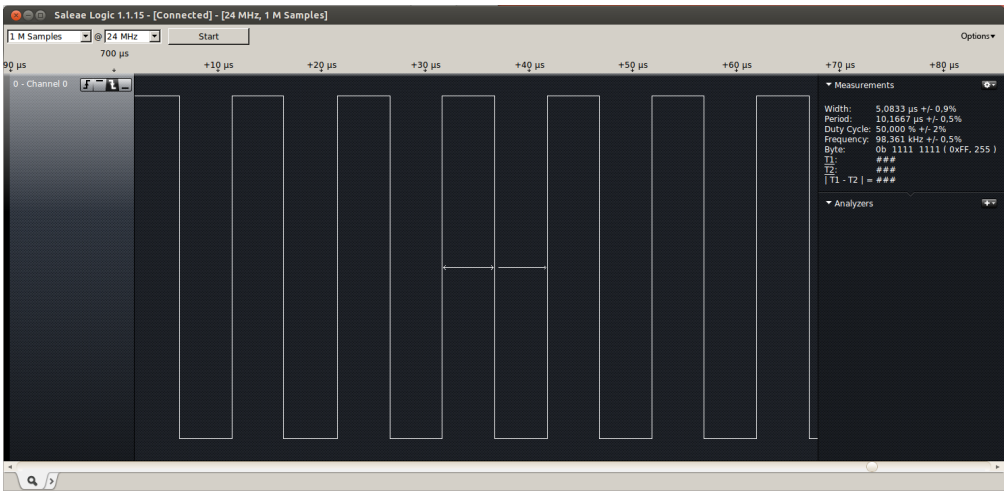

Figure 8.1: Toggling a pin through the virtual Linux file system sysfs.

Sysfs test	
Frequency	0.098MHz

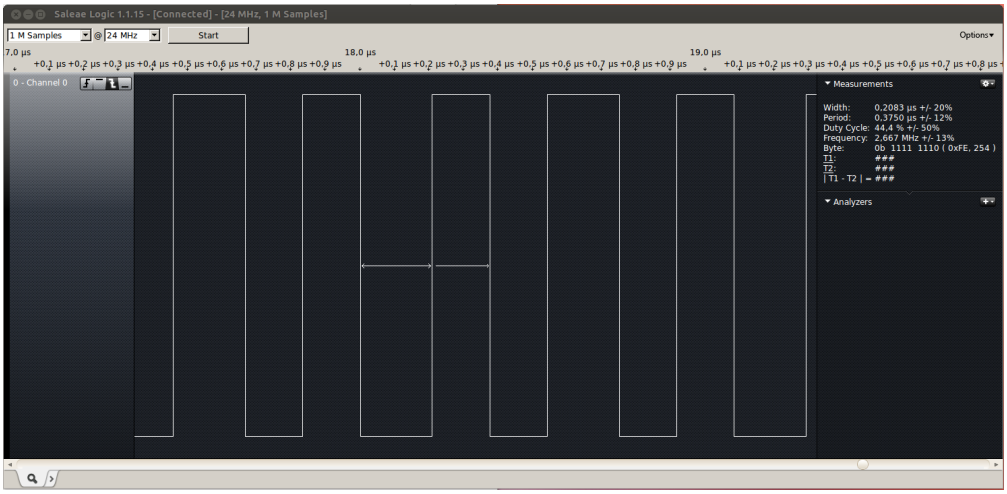

Figure 8.2: Toggling a pin through memory mapping.

Memory mapped test	
Frequency	2.67MHz

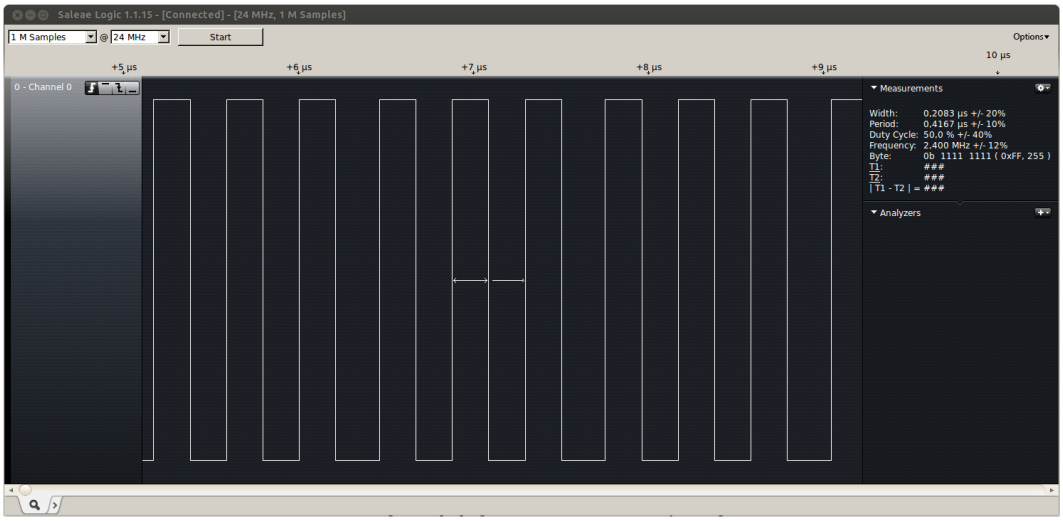

Figure 8.3: Toggling a pin through the PRU subsystem.

PRU test	
Frequency	2.40MHz
Frequency x 41	98.4Mhz

8.2 Cyclictest

The purpose of this test is to benchmark response testing with different kernels to highlight their advantages and disadvantages.

8.2.1 Setup

The benchmarking tools used in this setup was presented in Section 6.5. Cyclictest was used to perform large scaled tests on the response time of the system. Cyclictest was run with 1 million tests (-l1000000), at max priority (99), with locked memory allocation (-m), high precision nanosleep (-n), single thread (-t1) and with an interval of 400 microseconds (-i400). (-q) and (-h400) are output specific options only. CPU stress was used to load the CPU to its maximum (–cpu 1). Further, hackbench was used for a more elaborate load scheme with 20 groups (-g 20) of threads (-T) creating load by communicating with each other through pipes. The complete commands are shown in Figure 8.4.

Test params				
Test cmd	*cyclictest -l1000000 -m -n -t1 -p99 -i400 -h400 -q*			
CPU stress	*stress --cpu 1*			
Hackbench	*hackbench -T -g 20 -l 100000000*			

Figure 8.4: The commands used for testing.

8.2.2 Testing

The tests were carried out for both the basic preemption and PREEMPT_RT kernel, first without stress, then only CPU stress, and finally with the hackbench stress scheme. We also ran tests to check if the non-sleep optimized kernels made any difference.

The first three Figures, 8.5, 8.6 and 8.7, show the difference in response time, for the different kernels, with difference stress levels. Figure 8.6 includes an additional dataset not produced by this project, but by an external source with equal test setup[1]. Figure 8.8 shows the difference between a CPU stressed vs. a non-stressed test.

[1] Cyclic test performed by a google boards user, PREEMPT 3.8 kernel [23]

Finally, the two Figures 8.9 and 8.10, depicts the non-sleep optimized kernel vs. the original kernel.

All test data, which was created in this test, is appended and referred to in Appendix B.1.

8.2.3 Results

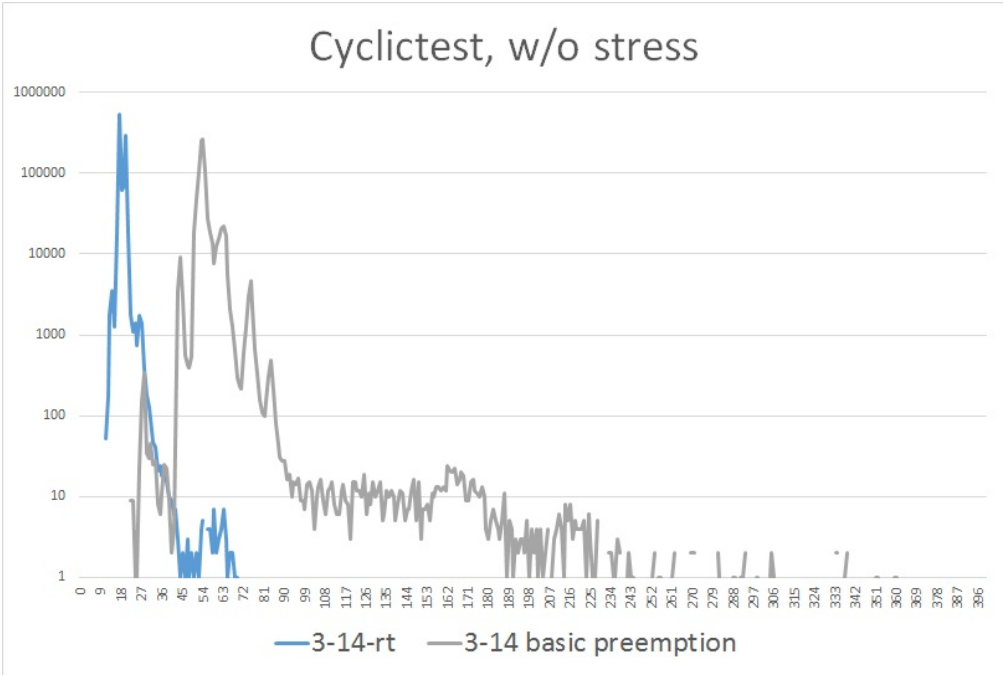

Figure 8.5: Cyclictest without stress.

3-14-rt preempt_rt	
Sum tests	1000000
Avg	18
Min	11
Max	73

3-14 basic preemption	
Sum tests	1000000
Avg	54
Min	20
Max	823

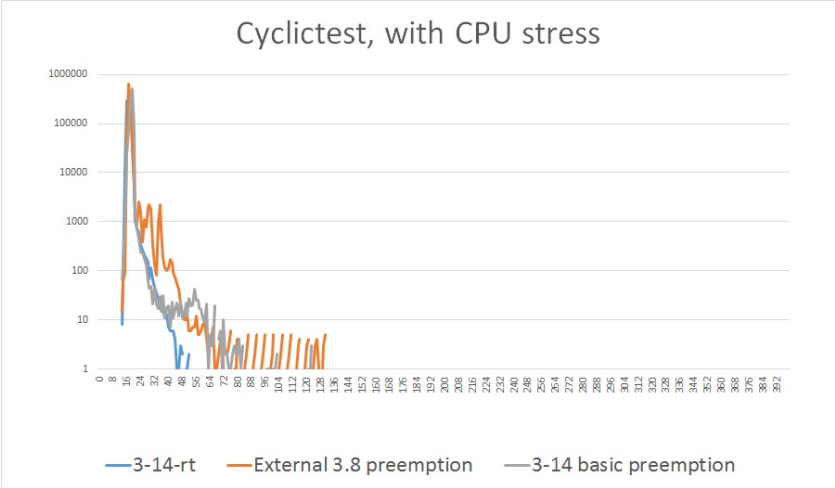

Figure 8.6: Cyclictest with CPU stress. The 3.8 kernel data is from an external source.

3-14-rt preempt_rt	
Sum tests	1000000
Avg	17
Min	13
Max	54

External 3-8 preemption	
Sum tests	1000000
Avg	17
Min	13
Max	262

3-14 basic preemption	
Sum tests	1000000
Avg	18
Min	13
Max	261

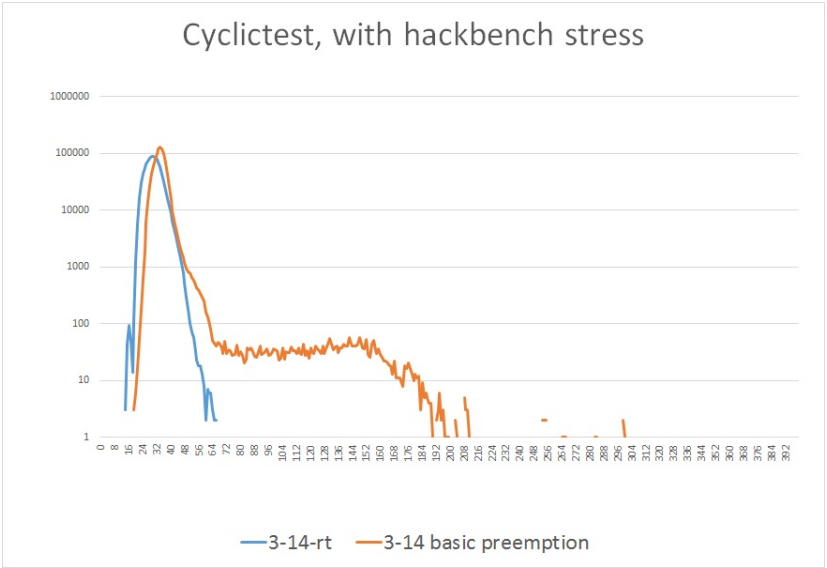

Figure 8.7: Cyclictest with hackbench stress.

3-14-rt preempt_rt	
Sum tests	1000000
Avg	30
Min	14
Max	78

3-14 basic preemption	
Sum tests	1000000
Avg	34
Min	19
Max	547

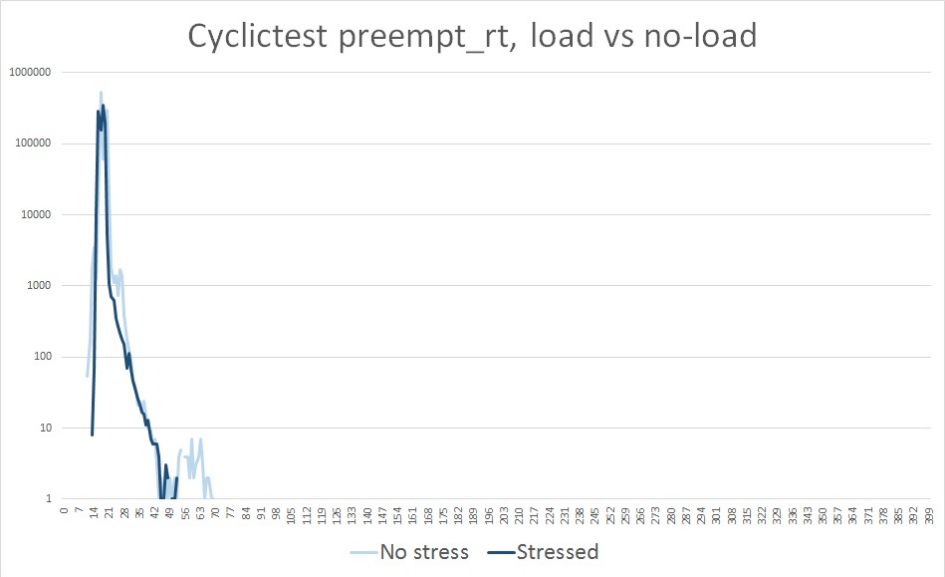

Figure 8.8: Cyclictest data showing the difference between a idle and CPU loaded system.

preempt_rt no-stress	
Sum tests	1000000
Avg	18
Min	11
Max	73

preempt_rt, stressed	
Sum tests	1000000
Avg	17
Min	13
Max	54

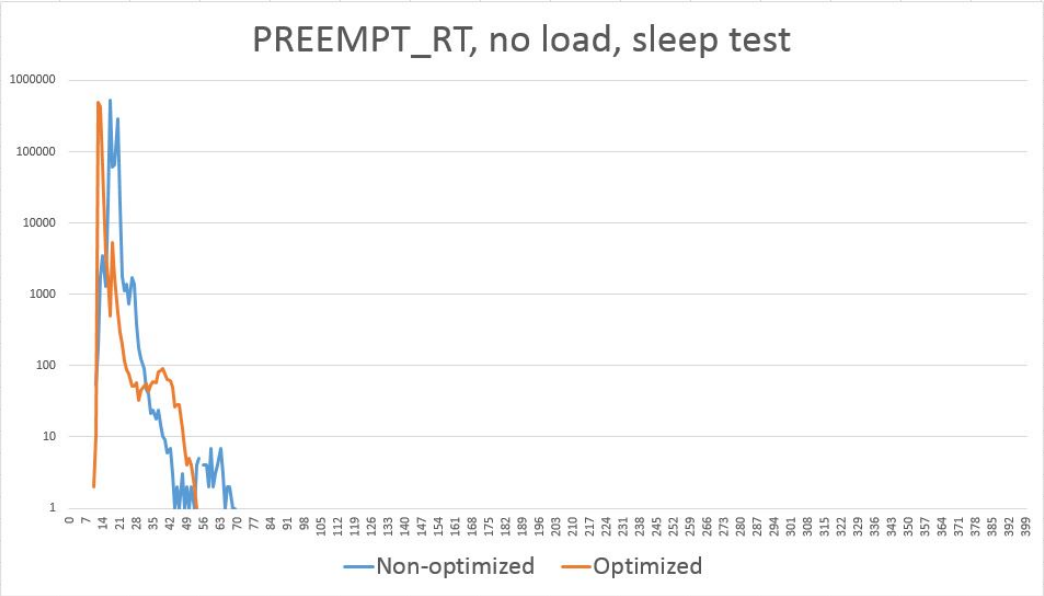

Figure 8.9: Normal vs sleep optimized kernel, not stressed

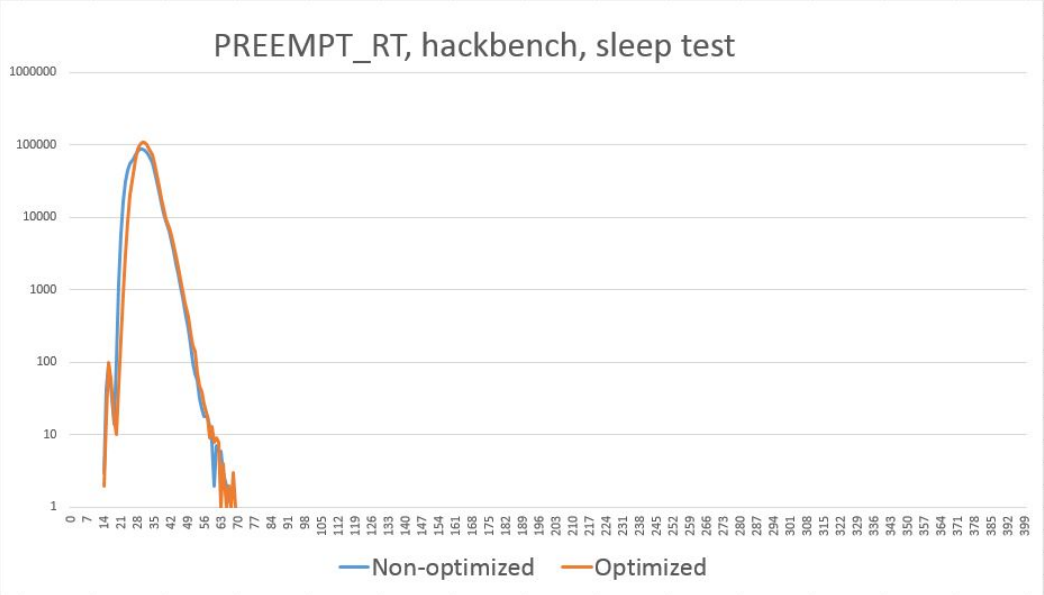

Figure 8.10: Normal vs sleep optimized kernel, hackbench stressed

preempt_rt, no-optimized, no stress	
Sum tests	1000000
Avg	18
Min	11
Max	73

preempt_rt, optimized, no-stress	
Sum tests	1000000
Avg	12
Min	10
Max	67

8.3 Cape response testing

The following test will benchmark the time interval the BeagleBone Black uses to respond to external signals under different circumstances.

8.3.1 Setup

The ERT cape was connected to the BeagleBone through the input and output pins matching the pin layout in Section 6.4.2 and Section 5.2.4.3. Note that when testing with the PRU, the BeagleBone utilizes different input and output pins. The cape was set up to send 1000 signals on each channel, giving a total of 3000 signals when utilizing three channels. The signals were sent at a rate of once per 32ms, however, in cases were this was too fast, the cape would instead set to free-running mode. The timer resolution was set to R:64, a prescalar of 64. This gave the output histogram a resolution of 3ms over 128 slots, each with a 24μs resolution.

8.3.2 Testing

Testing was carried out across four different platforms, basic preemption Linux kernel, PREEMPT_RT patched Linux kernel, FreeRTOS and the PRU subsystem. The tests we ran with the Linux kernels were performed with and without stress created by hackbench (same as in 8.2), while running the test application shown in Section 6.4.3, in response test mode. Testing with the PRU was implemented with the busy-wait scheme laid out in Section 6.4.4.3, without any applied stress. Testing on FreeRTOS was executed with the test application presented in Section 7.3, with both the busy-wait and periodic polling schemes.

All results are, if not specified otherwise, depicted with a logarithmic scale with base 10 on the y-axis. The test data, which was created in this test, is appended and referred to in Appendix B.1.

8.3.3 Results

8.3.3.1 Basic preemption Linux kernel

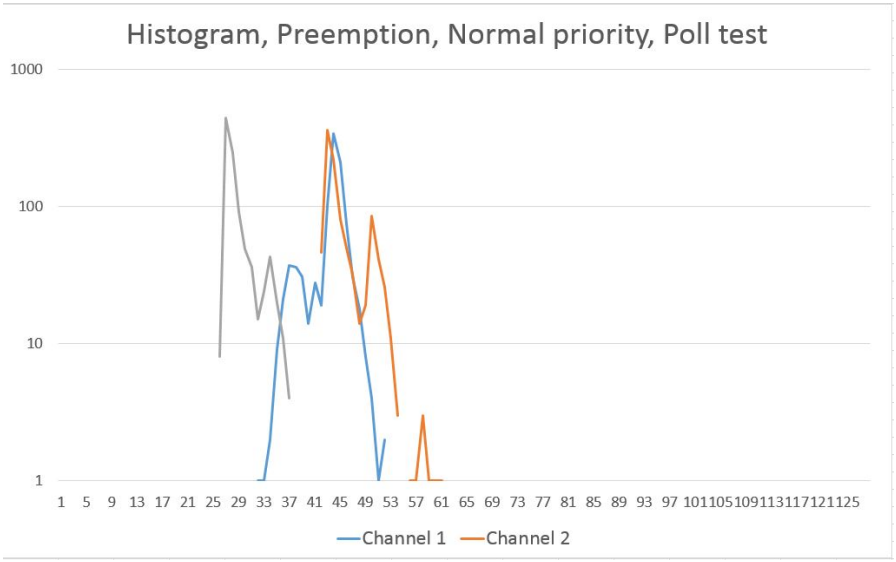

Figure 8.11: Polling scheme, 1000 tests on three channels.

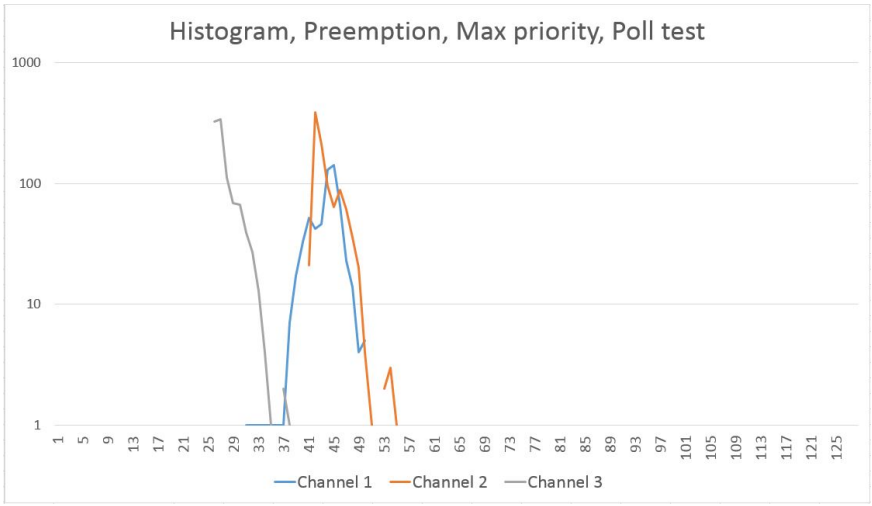

Figure 8.12: Polling scheme, 1000 tests on three channels, max priority.

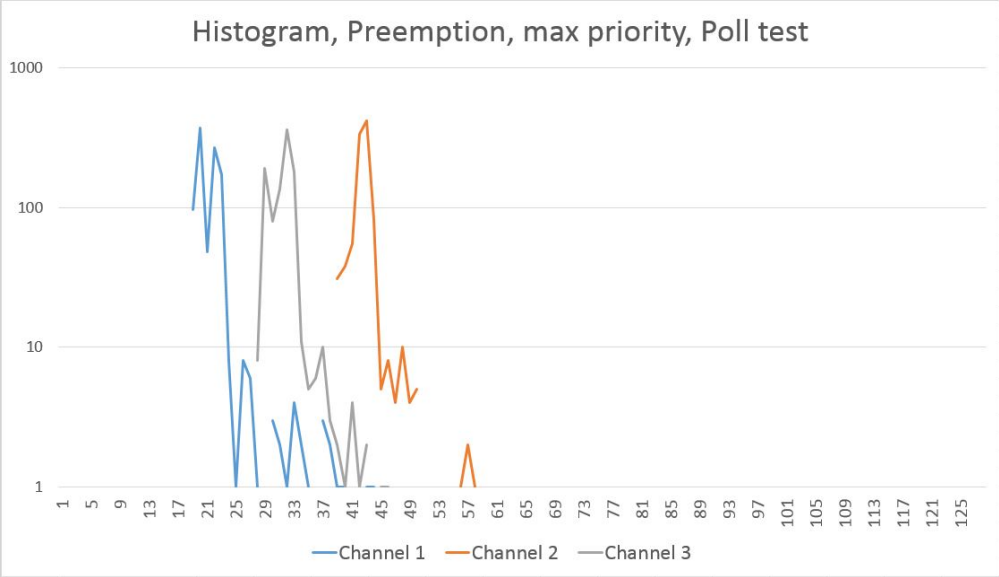

Figure 8.13: Polling scheme, 1000 tests on three channels, max priority, with hackbench stress.

Busy-wait and hackbench without priority change

The busy-wait scheme and the test running normal priority with hackbench stress failed to provide any tangible results.

8.3.3.2 PREEMPT_RT patched Linux kernel

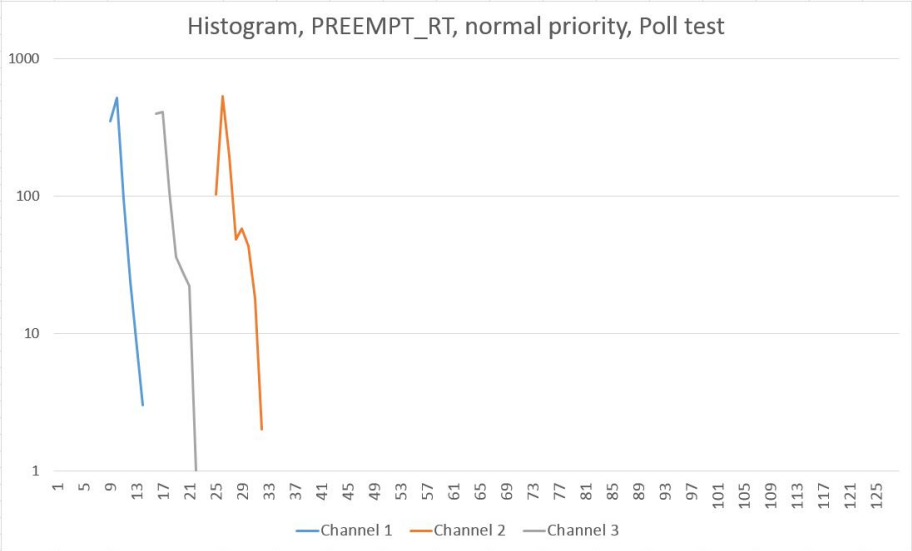

Figure 8.14: Polling scheme, 1000 tests on three channels.

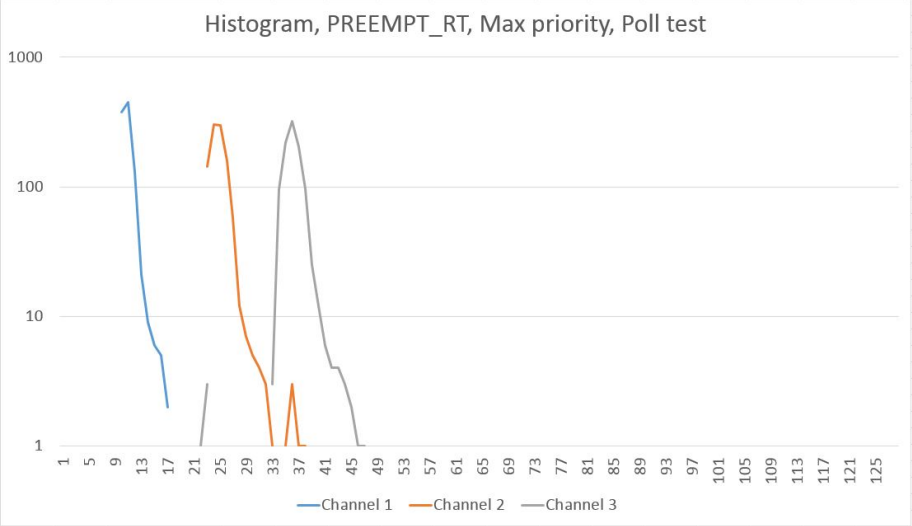

Figure 8.15: Polling scheme, 1000 tests on three channels, max priority.

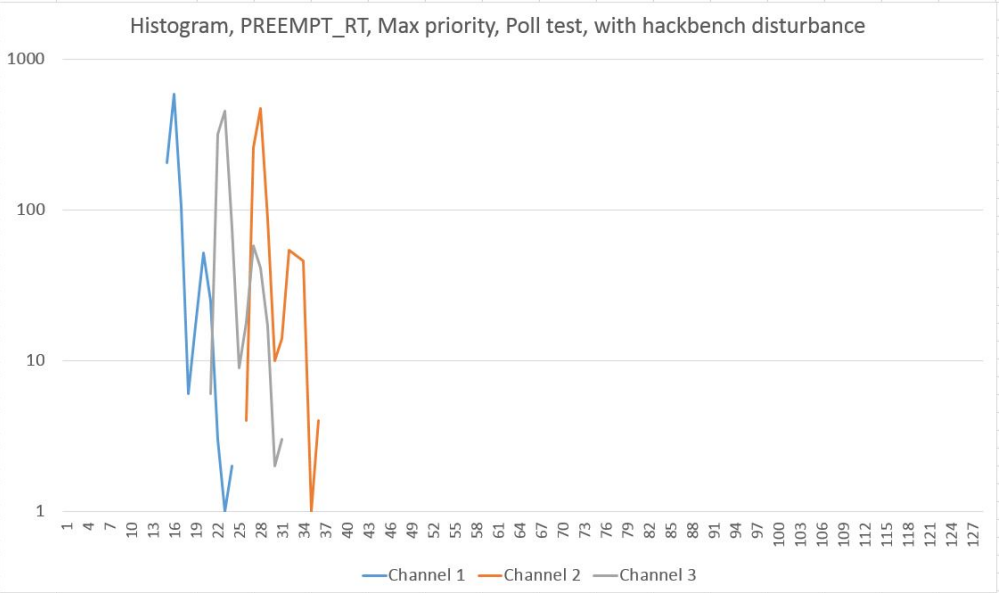

Figure 8.16: Polling scheme, 1000 tests on three channels, max priority, with hackbench stress.

Busy-wait and hackbench without priority change

The busy-wait scheme and the test running normal priority with hackbench stress failed to provide any tangible results.

8.3.3.3 FreeRTOS

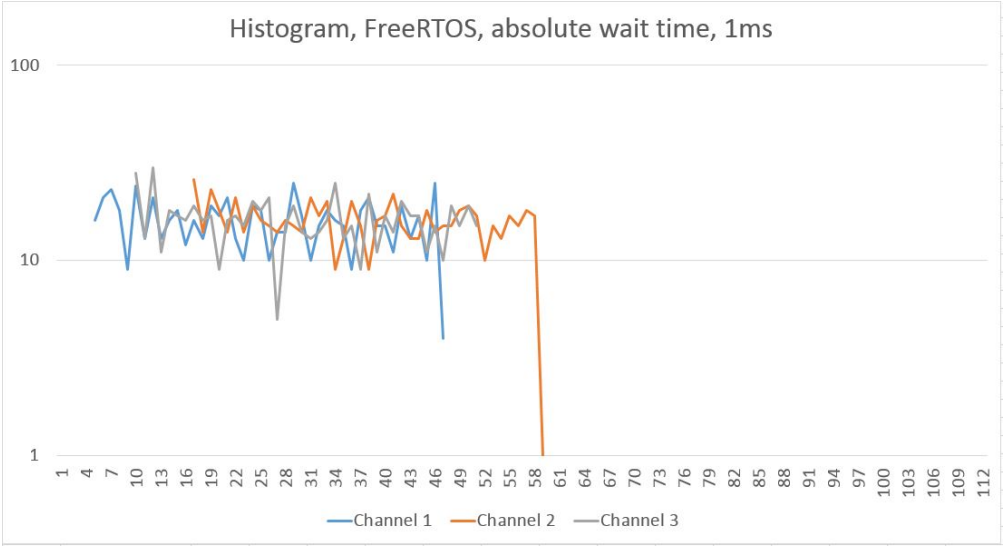

Figure 8.17: 1000 tests on three channels, interrupts with absolute periodic polling at 1ms.

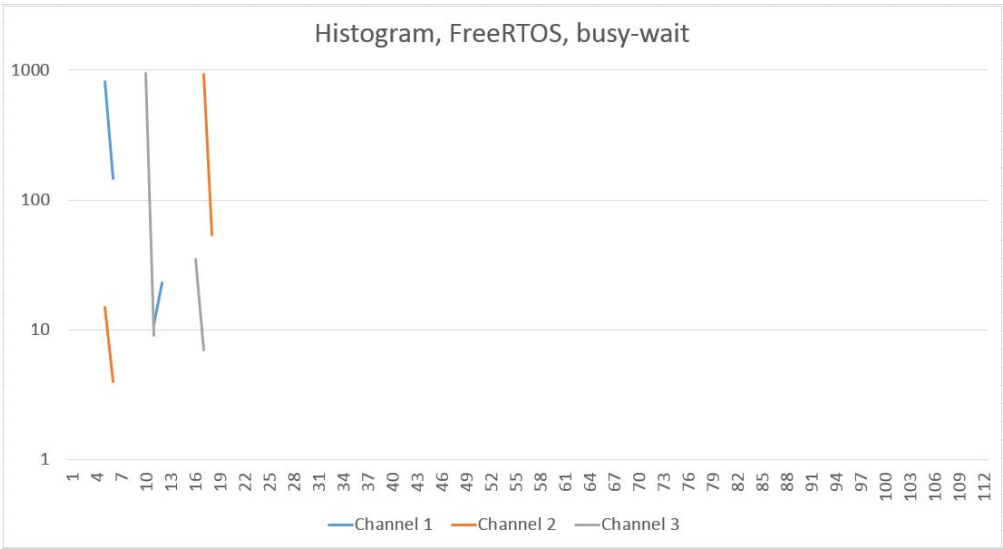

Figure 8.18: 1000 tests on three channels, interrupts with busy-wait scheme.

8.3.3.4 PRU

We were unable to run any meaningful test with the ERT cape. The PRU did responded, however, the response time was either measured as zero or too fast for the cape to detect. Figure 8.19 illustrates this, and was created with the logic analyser together with the ERT and PRU running a test.

Figure 8.19: Logic analyzer showing the immediate response of the PRU. The top line is the ICP value, the next three lines are the ERT capes output on channel 1 to 3, with the PRU response on channel 1 to 3 on the last three lines.

9 Discussion

9.1 I/O latency in GNU/Linux

While the toggle test is not very relevant to real-time topics, it is meant as an introduction to embedded programming on 32-bit MCU's with a complex OS. It shows a new added dimension to GPIO control, which did not exist when programming 8-bit microcontrollers, namely that there are different ways of accessing GPIOs, and the fact they all have pros and cons. Communicating with a driver in kernel space through sysfs (introduced in Section 4.1.9.1) provides an understandable syntax, and it is a secure way of doing it as there is seldom a need for superuser privileges. However, it requires writing a kernel driver, and all interaction will induce a context switch. Memory Mapping (introduced in Section 4.1.9.2) is the direct opposite, superuser privileges are required so the programmer can cause unforeseen problems, and memory mapping can be quite obscure for the untrained eye. It does, however, provide easy and responsive access to GPIO, the results showed that memory mapped control (see Results 8.2) was 27 times faster than control with communication through the sysfs virtual file system (see Results 8.1). A simple test like the toggle test and its results, can if studied give insight into how the Linux kernel operates.
We also performed a toggle test with the processors PRU subsystem (introduced in Section 3.2.2), it gave 35 times faster control than memory mapping and 1000 times faster control than through sysfs (see Results 8.3). This PRU test was meant to show how GNU/Linux needs not be a debilitating factor if we require a really deterministic and fast interface. Modern processor chips are often not only a simple processing core, they also have different subsystems for different tasks.

9.2 System latency in GNU/Linux, PREEMPT_RT vs basic low-latency preemptable kernel

The cyclictest measures the amount of time that passes between when a timer expires, and when the thread, which set the timer, actually runs. Thus, it measures system latency. We observe by Figures 8.5, 8.6, and 8.7, how the PREEMPT_RT patch substantially improves worst case scenarios regarding system latency. It is important when judging the real-time characteristics of non-mathematically provable schedulers, that the latency appears to be bounded, boundedness was explained in Section 2.2.1. It is difficult to estimate a bound for the basic preemption results because of the tail observed, see for instance Figure 8.7. Figures 8.6 and 8.7 have an average system latency almost equal to the PREEMPT_RT version, however, in real-time systems we are not so worried about average latency, but instead worst case latency. It is this latency, which can potentially wreck a well designed real-time system. Therefore, even though the two systems have nearly the same average latency, the PREEMPT_RT delivers much better results with 5 to 10 times lower worst case scenarios than the basic preemption kernel.
An interesting observation was made for the PREEMPT_RT kernel, a non-stressed system delivered slightly worse results than a loaded systems. This is shown in Figure 8.5, where we have an almost as high worst-case as the hackbench stressed Figure 8.7, and an inferior average to maximum latency ratio of all the PREEMPT_RT tests. This last factor is important as a lower average to maximum latency ratio means that the latency is to a higher degree bounded. The results of the PREEMPT_RT kernel with no load vs. loaded are shown in Figure 8.8. We suspected that the difference was due to the OS having the processor put in sleep state, an attempt to disable such features were implemented and the kernel was then recompiled, see Section 6.3.4. The difference in test results between the optimized and the non-sleep optimized kernels is observed in Figures 8.9 and 8.10. When the system is stressed with the hackbench tool we observe that the removal of sleep features had no effect. This was as expected since the system would never enter a sleep state when heavily stressed. The non-loaded system did show differences, however, not as we expected. The experiment with turning off the sleep features was expected to bring down the worst-case scenario latency, instead, it had an greater impact on the average

latency. Thus, there were no significant improvement regarding the boundedness of the latency test results. The lower average latency does, however, increase the throughput of the system, but most likely at a cost of higher power usage because of the disabled sleep features. It should be noted that there might be other processor features, which this project did not identify, that could improve on the boundedness of the non-loaded system.
The externally sourced test result in Figure 8.6 is included as a verification of our testing scheme, delivering results with striking similarities to the 3.14 preemption kernel.

Response testing with the ERT cape

Further latency testing was carried out in conjunction with the ERT cape, the results were presented in Section 8.3.3. These results support the statements in the discussion above, the PREEMPT_RT kernel gives predictable and bounded results for all three test response channels, while the basic preemption kernel results depicts a more chaotic response. The most interesting figures to compare are Figure 8.13 and Figure 8.16. We can observe that both tests tend to use the same time responding to Channel 1, however, while PREEMPT_RT manages to keep all of its response times in this area, basic preemption occasionally more than doubles its response time. This increase in response time, compared to PREEMPT_RT response time, also propagates and get larger for each channel, a very undesired property in a real-time system.
The results with the ERT cape are harder to interpret than the cyclictest results, as the difference in results were greater for those tests. Nonetheless, response testing with an external module does give another opportunity, to compare results with a pure RTOS like FreeRTOS. Results from testing with FreeRTOS on the Beaglebone Black were shown in Section 8.3.3.3, and while the PREEMPT_RT patch improves on Linux predictability, we can observe that FreeRTOS is in a league of its own. The busy-wait results, depicted in Figure 8.18, show how the response intervals have three distinct slots, which it responds in, one at 5-6, then at 10-12, and again at 16-18. This is exactly the behaviour expected from a RTOS busy-wait loop where the only disturbance source is the system tick interrupt, which has a bounded ISR. FreeRTOS shows its prowess again with the periodic polling test, depicted in Figure 8.17. 1ms represents 42 histogram slots and Channel 1 responds between

slots 5 to 47, Channel 2 from 17 to 59, and Channel 3 from 10 to 51, each with its own ĩms window of response. The reason Channel 1 does not start respond at 0 is the interrupt overhead, while Channel 2 has the interrupt overhead, as well as the Channel 1 response overhead, and so on.
We also implemented a busy-wait scheme without interrupts for the PRU. This is risky because the PRU has to read the pin as low before it returns high, or else it will miss the event. However, since the PRU is exceedingly fast it is actually possible to implement a busy-wait scheme, which can guarantee response as long as the input signal is kept low for a minimum amount of time. For our busy-wait scheme implemented in Section 6.4.4.3 the signal had to be kept low for at least 0.1μs, which is much less than signal width sent from the ATMEGA MCU on the cape. The PRU had thus no problems providing near instantaneous responses, which can be observed in the results in Figure 8.19. A more optimistic implementation was the pure busy-wait tests through GNU/Linux, which failed to provide any results, even when testing only one channel. The total overhead was just too large for the BeagleBone to observe the state change.

9.3 Working with the BeagleBone Black and ERT cape

The community linked to the BeagleBone Black is important in terms of education, since support is always only a few keystrokes away. However, such community driven boards often provide simple hardware, or do not feature components for embedded environments. The BeagleBone Black does, however, have an industrial grade processor featuring top of the line technology. Working with such a device provides students with expertise, which they can use to their advantage when entering employment, thus it makes the platform a lot more exciting to work with. With extended OS support it also has the ability to be a single point embedded platform for a complete exercise program for the course TTK4147, Real-Time Systems.

The implementation of the ERT was discussed in Section 5.4, the ERT cape performed perfectly, and delivered the results that this project required. The use of the ICP feature makes it is very precise, it is also more flexible than its counterpart, the AVR butterfly response tester. It features several timer resolutions, and it also features a free-running mode such that timer overflow will not abort a test.

The ability to store the test results as a histogram instead of posting the results continuously is also a useful feature, as it prevents serial communication from interfering with a test.

10 Conclusion and recommendations

10.1 Conclusions

The BeagleBone Black together with the ERT cape is a solution capable of unifying the TTK4147 course assignments. It delivers the multiple OS needed, as well as a well functioning cross-platform benchmark tool.
There is no doubt that PREEMPT_RT improves on the real-time characteristics of Linux, and it does a good job of creating bounded latencies. However, it should be noted that as long as worst case latencies can not be explicitly proven, it is difficult see GNU/Linux delivered in any life-critical applications. While basic preemption is not suited for hard real-time systems, we can conclude that GNU/Linux patched with PREEMPT_RT is not "definitely unsuitable". Its applicability depends on the requirements of a given application, and this OS should thus not be ignored as a potential candidate when starting a project with real-time requirements.
Extending the features of GNU/Linux with a PRU subsystem or similar is also an option for projects which wants the best of both, a highly resourced OS and deterministic behaviour.

10.2 Recommendations for future work

This thesis has studied at a diverse set of topics, which could by themselves been studied in a thesis. However, we recommends two items as most interesting for future work:

1. Fully implement a course assignment for TTK4147, Real-Time Systems, on the BeagleBone Black in conjunction with the ERT cape.

2. Perform a deeper investigation on the properties of the PRU on the BeagleBone Black, and evaluate how it can be used as a real-time subsystem in conjunction with a complex OS like GNU/Linux.

A | Acronyms

ASF Atmel Software Framework

BSP Board Support Package

CISC Complex Instruction Set Computing

DIP Dual In-line Package

ERT External Response Tester

EVT Exception Vector Table

FIFO First In, First Out

GPIO General-Purpose Input/Output

IC Integrated Circuit

ICP Input Capture Pin

IDE Integrated Development Environment

ISR Interrupt Service Routine

IRQ Normal Interrupt

MCU Microcontroller

OCP On Chip Peripheral

OS Operating System

PCB Printed Circuit Board

Pinmux Pin Multiplexor

PMS Package Management System

RISC Reduced Instruction Set Computing

RTOS Real-Time Operating System

SWI Software Interrupt

TRM Technical Reference Manual

UART Universal Asynchronous Receiver/Transmitter

B Appendix

B.1 Appended external storage

The appended storage unit contains the following folders:

1. GNU_Linux/
   (a) linux-dev-am33x-v3.14/, contains the kernel.
   (b) tester_application/, contains the eclipse files for the GPIO test application.
   (c) pru/, contains the PRU support package, and the source files for the PRU applications.
2. FreeRTOS/
   (a) RTOS_Bone-master/, contains the FreeRTOS kernel and additional source files.
3. QNX/, contains the QNX BSP user manual, as well as an test image, MLO and u-boot binaries.
4. Hardware_related _files/
   (a) AltiumDesigner/, the Altium designer project files used to design the PCB design.
5. Results/
   (a) Cyclictest_results/, results from the cyclictest.
   (b) ERT_results/, results from the ERT tests.

B.2 How to boot the different OS on BeagleBone Black

B.2.1 Booting Linux

There should be a red microSD-card appended to this report, it contains the PREEMPT_RT kernel version of GNU/Linux. Plug it into the BeagleBone Black and hold the s2 boot button while powering to start GNU/Linux.

B.2.1.1 Swapping the kernel

In the appended external storage unit we can find the Linux kernel development files, located at *GNU_Linux/linux-dev-am33x-v3.14/*. Further, the *deploy/cores* folder contains zipped files of all compiled kernels used in this project. To change the kernel on the SD-card, first unzip the wanted kernel and plug in the SD-card. Then run the following commands:

- sudo cp -v ./unzipped_folder/kernel_version.zImage /media/boot/zImage
- sudo tar xfov ./unzipped_folder/kernel_version-dtbs.tar.gz -C /media/boot/dtbs/
- sudo tar xfv ./unzipped_folder/kernel_version-firmware.tar.gz -C /media/rootfs/lib/firmware/
- sudo tar xfv ./unzipped_folder/kernel_version-modules.tar.gz -C /media/rootfs/

B.2.1.2 Log in

SSH is set up on the GNU/Linux distributions such that a terminal can be acquired using an internet cable. The USB to uart cable used for QNX and FreeRTOS (B.2.2) can also be used, this works because ttyO0 is set by default as terminal. Log in as: debian, password: temppwd.

B.2.2 Booting QNX and FreeRTOS

The black microSD-card appended to this report is intended for booting FreeRTOS and QNX. Both these run without any ethernet set up. However the uart0 header

will provide a terminal. Use the FTDI to TTL cable (see Figure B.1) which followed this report to get a console through a USB port of the host computer. Standard serial settings are: 115200 baud, 8 data bits, no parity, and 1 stop bit (115200 8N1).

Figure B.1: The standard FTDI to TTL cable.

B.2.2.1 QNX

In the appended external storage unit we can find the QNX board support package, located at *QNX/*. Further, the *deploy/* folder contains the MLO, u-boot, and binary file to start qnx on the BeagleBone. Remember to copy the MLO first to a clean version of the microSD-card. Plug in the card, power up the board, and press the s1 "Reset" button, and s3 "Power" button. This will cause the board to load and run u-boot. At the u-boot command prompt type: *uenvcmd=mmcinfo;fatload mmc 0 81000000 ifs-ti-am335x-beaglebone.bin; go 81000000;*. QNX should now boot.

B.2.2.2 FreeRTOS

In the appended external storage unit we can find the FreeRTOS development files, located at *FreeRTOS/*. The makefile for the project is located at *Demo/AM3359_BeagleBone_GCC/*, this folder also includes the MLO, u-boot, and binary file to start FreeRTOS on the BeagleBone. Further, it is the same procedure as for QNX, however, the u-boot command is changed to: *mmcinfo;fatload mmc 0 80500000 rtosdemo-a.bin; go 80500000;*.

B.3 ERT code files

B.3.1 defines.h and globals

```
/*
 * defines.h
 *
 * Created: 23.05.2014 13:06:37
 *  Author: henrifo
 */
#define F_CPU 8000000UL

#include <stdio.h>
#include <string.h>
#include <avr/io.h>
#include <avr/interrupt.h>
#include <util/delay.h>
#include <inttypes.h>

#ifndef DEFINES_H_
#define DEFINES_H_


// Misc definitions
#define byte   uint8_t
#define bool   int
#define FALSE 0
#define TRUE   1
#define RESOLUTION_64 1
#define RESOLUTION_256 2
#define RESOLUTION_1024 3

//Test definitions
#define NORMAL_TEST 0
#define VARYING_TEST 1
#define NODELAY_TEST 2
#define KAMIKAZ_TEST 3

// Bit location definitions
#define ICP1_BIT      0x01
```

```
#define PCINT9_BIT      0x02
#define PCINT10_BIT     0x04
#define PCINT11_BIT     0x08

// Pin definitions
#define PIN_PCINT1      PINC

// Usart definitions
#define FOSC 8000000
#define BAUD 38400
#define MYUBRR FOSC/16/BAUD-1

//State variables and definitions
#define STATE_INIT 0
#define STATE_TEST 1
#define STATE_IDLE 2

// Structs
typedef struct  interrupt_info
{
  unsigned short     send_time_cnt;
  unsigned short     send_time_ovf;
  unsigned short     rec_time_cnt;
  unsigned short     rec_time_ovf;
} interrupt_info;


#endif /* DEFINES_H_ */
```

ap/main/defines.h

```
/*
 * globals.h
 *
 * Created: 24.05.2014 18:55:11
 *  Author: henrifo
 */


#ifndef GLOBALS_H_
#define GLOBALS_H_
```

```
12 #include "defines.h"

14

   // Int measure variables
16 extern volatile byte         pcint1_history;
   extern volatile interrupt_info     int_info1;
18 extern volatile interrupt_info     int_info2;
   extern volatile interrupt_info     int_info3;
20 extern volatile byte     responded_channels;
   extern volatile unsigned short     timer1_uShrt;
22 extern volatile unsigned short     timer1_ovfcnt_uShrt;

24 //USART buffer
   extern volatile char  usart_buffer[15];
26 extern volatile char  usart_buffer_copy[15];
   extern volatile byte  usart_buffer_position;
28 extern volatile bool  usart_received_trans;
   extern volatile byte     send_flag;
30



32

   //Setup variables
34 extern byte STATE;
   extern byte TEST_TYPE;
36 extern short NR_OF_TESTS;
   extern bool TEST_CHANNEL_1;
38 extern bool TEST_CHANNEL_2;
   extern bool TEST_CHANNEL_3;
40 extern bool FREE_RUNNING;

42 //Timer variables
   extern volatile byte send_freq; //  FOSC/PRESCALAR*SEND_FREQ
44 extern volatile byte resolution;

46 //Histogram overflows
   extern volatile short testOvf1;
48 extern volatile short testOvf2;
   extern volatile short testOvf3;
50

   #endif /* GLOBALS_H_ */
```

ap/main/globals.h

B.3.2 main.c

```
/*
 * GccApplication1.c
 *
 * Created: 09.04.2014 17:34:35
 *  Author: henrifo
 */

#include "defines.h"
#include "globals.h"
#include "timer.h"
#include "usart.h"
#include "io.h"

//******** DEFINE GLOBAL VARIABLES *********//
// Int measure variables
volatile byte       pcint1_history    = 0xFF;
volatile interrupt_info   int_info1;
volatile interrupt_info   int_info2;
volatile interrupt_info   int_info3;
volatile unsigned short   timer1_uShrt    = 0;
volatile unsigned short   timer1_ovfcnt_uShrt = 0;
volatile byte       responded_channels = 0;
//USART buffer
volatile char usart_buffer[15];
volatile char usart_buffer_copy[15];
volatile byte usart_buffer_position = 0;
volatile bool usart_received_trans =  FALSE;
volatile byte   send_flag = FALSE;

//Timer var
volatile byte resolution=RESOLUTION_64;

//Setup variables
byte STATE = STATE_INIT;
```

```
byte TEST_TYPE       = NORMAL_TEST;
short NR_OF_TESTS    = 5;
bool TEST_CHANNEL_1 = FALSE;
bool TEST_CHANNEL_2 = FALSE;
bool TEST_CHANNEL_3 = FALSE;
bool FREE_RUNNING = FALSE;

//#define CONSOLE_PRINT 1;
byte abort = FALSE;

short testVar1[112] = {0};
short testVar2[112] = {0};
short testVar3[112] = {0};
volatile short testOvf1 = 0;
volatile short testOvf2 = 0;
volatile short testOvf3 = 0;
char tmpbuf1[50];

//Send timer
volatile byte send_freq = 0xFF; //  FOSC/PRESCALAR*SEND_FREQ
//******** END: DEFINE GLOBAL VARIABLES *********//

void print_result();

void init_test()  {
  //Reset histogram
  memset(testVar1,0, sizeof(testVar1));
  memset(testVar2,0, sizeof(testVar1));
  memset(testVar3,0, sizeof(testVar1));
  testOvf1 = 0;
  testOvf2 = 0;
  testOvf3 = 0;
  int_info1.rec_time_cnt=0; int_info1.rec_time_ovf=0; int_info1.send_time_cnt=0; int_info1.se
  int_info2.rec_time_cnt=0; int_info2.rec_time_ovf=0; int_info2.send_time_cnt=0; int_info2.se
  int_info3.rec_time_cnt=0; int_info3.rec_time_ovf=0; int_info3.send_time_cnt=0; int_info3.se
  abort = FALSE;
  //Setup PCINT
  pcint1_history = PIN_PCINT1; //initialize PCINT history to detect edges
  PCMSK1  = 0b00001110; //setup PCINT1-2 will trigger PCINT0 interrupt
  PCICR = (1<<PCIE1); //Enable PCINT0 interrupt
  //End
```

```
  //Setup input capture
  init_icp1_timer();
  //End
  //Setup signal timer
  init_timer0();
  //End
}

void exit_test()  {
  //Turn off PCINT interrupts
  PCICR = 0;
  //Turn off ICP interrupts
  TIMSK1  = 0;
  //Turn off compare match interrupt
  TIMSK0  =  0;
  //Go out of test state
  STATE = STATE_IDLE;
}

void INIT_BOARD() {
  CLKPR = (1<<CLKPCE); //Enable system prescaler change
  CLKPR = 0x00;     //Set prescaler to 0, enables 8Mhz

  //Setup input and output pins
  DDRC  &= 0b11110001;
  DDRD   = 0b11100000;
  PORTD |= 0b11100000;
  DDRB  |= _BV(DDB6); //PB6 output
  //End

  //Uart pins
  DDRD |= 0b00000010; //ensure tx as output
  DDRD &= 0b11111110; //ensure rx as input
  //End
}

int main(void)
{
  unsigned long i=0, j=0, ret=1;
  unsigned int ovf1, ovf2, ovf3;
  unsigned int time1, time2,time3;
```

```
117   memset(tmpbuf1, 0, 32);


119
      DDRB  |= _BV(DDB6); //PB6 output
121   while (i<20)  {
        _delay_ms(10);
123     PORTB ^= _BV(DDB6);
        i++;
125   }
      //
127
      //clear some int flags?
129   sei();

131   while (1) {
        switch(STATE) {
133       case STATE_INIT:
            INIT_BOARD();
135         USARTInit(MYUBRR);
            STATE = STATE_IDLE;
137         USARTWriteChar('I');
            USARTWriteChar('\n');
139         break;
          case STATE_IDLE:
141         //Handle received uarts
            if (usart_received_trans) {
143           handle_usart_trans();
              usart_received_trans = FALSE;
145         }
            //End
147         break;
          case STATE_TEST:
149         cli();
            init_test();
151         sei();
            USARTWriteChar('S');
153         USARTWriteChar('\n');
            _delay_ms(1000);
155
            for (i=0; i<NR_OF_TESTS; i++)
157         {
```

```
        if (FREE_RUNNING==FALSE) {
159       while (!send_flag) {
            //Loop and wait
161       }
          send_flag = FALSE;
163     }
        else
165     {
          _delay_ms(2);
167     }

169     if (TEST_CHANNEL_1)
          send_interrupt(PD5);
171     else if (TEST_CHANNEL_2) {
          send_interrupt(PD5);
173       send_interrupt(PD6);
        }
175     else {
          send_interrupt(PD5);
177       send_interrupt(PD6);
          send_interrupt(PD7);
179     }
        if (TEST_CHANNEL_1) {
181       while(1) {
            if (responded_channels == 1) {
183           responded_channels = 0;
              ovf1 = int_info1.rec_time_ovf-int_info1.send_time_ovf;
185           time1 = int_info1.rec_time_cnt-int_info1.send_time_cnt+ovf1*65535;
              j = time1 / 3;
187           if (j<112)
                testVar1[j]++;
189           else
                testOvf1++;
191           break;
            }
193         else if (send_flag && FREE_RUNNING==FALSE) {
              abort = TRUE;
195           break;
            }
197       }
        }
```

```
199             else if (TEST_CHANNEL_2)  {
                  while(1)  {
201                 if (responded_channels == 2)  {
                      responded_channels = 0;
203
                      ovf1 = int_info1.rec_time_ovf-int_info1.send_time_ovf;
205                   time1 = int_info1.rec_time_cnt-int_info1.send_time_cnt+ovf1*65535;
                      j = time1 / 3;
207                   if (j<112)
                        testVar1[j]++;
209                   else
                        testOvf1++;
211                   ovf2 = int_info2.rec_time_ovf-int_info2.send_time_ovf;
                      time2 = int_info2.rec_time_cnt-int_info2.send_time_cnt+ovf2*65535;
213                   j = time2 / 3;
                      if (j<112)
215                     testVar2[j]++;
                      else
217                     testOvf2++;
                      break;
219                 }
                    else if (send_flag && FREE_RUNNING==FALSE)  {
221                   abort = TRUE;
                      break;
223                 }
                  }
225             }
                else if (TEST_CHANNEL_3)  {
227               while(1)  {
                    if (responded_channels ==3 )  {
229                   responded_channels = 0;
                      ovf1 = int_info1.rec_time_ovf-int_info1.send_time_ovf;
231                   time1 = int_info1.rec_time_cnt-int_info1.send_time_cnt+ovf1*65535;
                      j = time1 / 3;
233                   if (j<112)
                        testVar1[j]++;
235                   else
                      {
237                     testOvf1++; }
                      ovf2 = int_info2.rec_time_ovf-int_info2.send_time_ovf;
239                   time2 = int_info2.rec_time_cnt-int_info2.send_time_cnt+ovf2*65535;
```

```
                j = time2 / 3;
                if (j<112)
                  testVar2[j]++;
                else
                {
                  testOvf2++;
                }
                ovf3 = int_info3.rec_time_ovf-int_info3.send_time_ovf;
                time3 = int_info3.rec_time_cnt-int_info3.send_time_cnt+ovf3*65535;
                j = time3 / 3;
                if (j<112)
                  testVar3[j]++;
                else
                {
                  testOvf3++;
                }

                break;
              }
              else if (send_flag && FREE_RUNNING==FALSE)  {
                abort = TRUE;
                break;
              }
            }
          }
          if (abort)  {
            USARTWriteChar('A');
            USARTWriteChar('B');
            USARTWriteChar('T');
            USARTWriteChar('\n');
            break;
          }
        }
        USARTWriteChar('E');
        USARTWriteChar('\n');
        if (ret != -1)
          print_result();
        exit_test();
        break;
      default:
        break;
```

```
281     }
  }
283 }


285
void print_result() {
287   int i,j,ret;
  short hist1, hist2, hist3;
289   for  (i=0; i<113; i++)  {
    if (i==112) {
291     hist1 = testOvf1;
      hist2 = testOvf2;
293     hist3 = testOvf3;
      USARTWriteChar('O');
295     USARTWriteChar(':');
      USARTWriteChar('\t');
297   }
    else
299   {
      hist1 = testVar1[i];
301     hist2 = testVar2[i];
      hist3 = testVar3[i];
303   }
    ret = sprintf(tmpbuf1,"t1:\t%u\tt2:%u\tt3:\t%u\n",hist1,hist2,hist3);
305   for (j=0; j<ret; j++) {
      USARTWriteChar(tmpbuf1[j]);
307   }
  }
309 }
```

ap/main/main.c

B.3.3 io.c

```
1 /*
 * io.c
3 *
 * Created: 23.05.2014 13:09:53
5 *  Author: henrifo
 */
```

```
#include "defines.h"
#include "globals.h"

ISR (PCINT1_vect)
{
  byte pin_pcint1 = PIN_PCINT1; //Lagre slik at forandringer underveis ikke skal ha noe aa si
  byte pcint1_positive_edges = ((pcint1_history^pin_pcint1) & pin_pcint1);
  pcint1_history = pin_pcint1;
  //CHANNEL 3
  if (pcint1_positive_edges & PCINT10_BIT)  {
    int_info3.rec_time_cnt=timer1_uShrt;
    int_info3.rec_time_ovf=timer1_ovfcnt_uShrt;
    int_info3.rec_time_cnt=timer1_uShrt;
    int_info3.rec_time_ovf=timer1_ovfcnt_uShrt;
    responded_channels++;
  }
  //CHANNEL 2
  if (pcint1_positive_edges & PCINT9_BIT) {
    int_info2.rec_time_cnt=timer1_uShrt;
    int_info2.rec_time_ovf=timer1_ovfcnt_uShrt;
    int_info2.rec_time_cnt=timer1_uShrt;
    int_info2.rec_time_ovf=timer1_ovfcnt_uShrt;
    responded_channels++;

  }
  //CHANNEL 1
  if (pcint1_positive_edges & PCINT11_BIT)  {
    int_info1.rec_time_cnt=timer1_uShrt;
    int_info1.rec_time_ovf=timer1_ovfcnt_uShrt;
    int_info1.rec_time_cnt=timer1_uShrt;
    int_info1.rec_time_ovf=timer1_ovfcnt_uShrt;
    responded_channels++;
  }
}


//Comment!
void send_interrupt(byte pin) {
  PORTD &= ~(1<<pin); //Set pin low
  if (pin==PD5) {
    int_info1.send_time_cnt=timer1_uShrt;
```

```
    int_info1.send_time_ovf=timer1_ovfcnt_uShrt;
49    int_info1.send_time_cnt=timer1_uShrt;
    int_info1.send_time_ovf=timer1_ovfcnt_uShrt;
51  }
  else if (pin==PD6)  {
53    int_info2.send_time_cnt=timer1_uShrt;
    int_info2.send_time_ovf=timer1_ovfcnt_uShrt;
55    int_info2.send_time_cnt=timer1_uShrt;
    int_info2.send_time_ovf=timer1_ovfcnt_uShrt;
57  }
  else if (pin==PD7)  {
59    int_info3.send_time_cnt=timer1_uShrt;
    int_info3.send_time_ovf=timer1_ovfcnt_uShrt;
61    int_info3.send_time_cnt=timer1_uShrt;
    int_info3.send_time_ovf=timer1_ovfcnt_uShrt;
63  }
  PORTD |= (1<<pin); //Set pin high
65 }
```

ap/main/io.c

B.3.4 usart.c

```
1 /*
 * usart.c
3 *
 * Created: 12.04.2014 10:49:24
5 *  Author: henrifo
 */
7 #include "defines.h"
#include "globals.h"
9

11 ISR (USART_RX_vect) {
  int i = 0;
13  usart_buffer[usart_buffer_position] = UDR0; //read usart into buffer
  usart_buffer_position++;
15  if (usart_buffer[usart_buffer_position-1]=='\n')  { //if end character then set received tr
    strcpy(usart_buffer_copy, usart_buffer);
17    usart_received_trans  = TRUE;
```

```
    //ECHO MESSAGE
    //for (i = 0; i < usart_buffer_position-1; i++) {
    //USARTWriteChar(usart_buffer[i]);
    //}
    //ECHO MESSAGE END
    usart_buffer_position = 0;
  }
  else if (usart_buffer_position>15)  { usart_buffer_position = 0; } //iterate buffer position
}

void handle_usart_trans() {
  if (strncmp(usart_buffer_copy, "F:R",3)==0) {
    FREE_RUNNING = TRUE;
  }
  if (strncmp(usart_buffer_copy, "T:MAX",5)==0) {
    NR_OF_TESTS=65000;

  }
  if (strncmp(usart_buffer_copy, "T:1000",6)==0)  {
    NR_OF_TESTS=1000;

  }
  else if (strncmp(usart_buffer_copy, "T:100",5)==0)  {
    NR_OF_TESTS=100;
  }
  else if (strncmp(usart_buffer_copy, "T:10",4)==0) {
    NR_OF_TESTS=10;
  }
  else if (strncmp(usart_buffer_copy, "R:64",4)==0) {
    resolution=RESOLUTION_64;
  }
  else if (strncmp(usart_buffer_copy, "R:256",5)==0)  {
    resolution=RESOLUTION_256;
  }
  else if (strncmp(usart_buffer_copy, "R:1024",6)==0) {
    resolution=RESOLUTION_1024;
    USARTWriteChar('t');
    USARTWriteChar('\n');
  }
  else if (strncmp(usart_buffer_copy, "C:1",3)==0)  {
    TEST_CHANNEL_1=TRUE;
```

```
59      TEST_CHANNEL_2=FALSE;
        TEST_CHANNEL_3=FALSE;
61    }
    else if (strncmp(usart_buffer_copy, "C:2",3)==0)  {
63      TEST_CHANNEL_2=TRUE;
        TEST_CHANNEL_1=FALSE;
65      TEST_CHANNEL_3=FALSE;
    }
67  else if (strncmp(usart_buffer_copy, "C:3",3)==0)  {
        TEST_CHANNEL_2=FALSE;
69      TEST_CHANNEL_1=FALSE;
        TEST_CHANNEL_3=TRUE;
71  }
    else if (strncmp(usart_buffer_copy, "START",5)==0)  {
73      STATE = STATE_TEST;
    }
75 }

77 //This function is used to initialize the USART
   //at a given UBRR value
79 void USARTInit(uint16_t ubrr_value)
   {
81   /*Set baud rate */
     UBRR0H = (unsigned char)(ubrr_value>>8);
83   UBRR0L = (unsigned char)ubrr_value;
     /* Enable receiver and transmitter */
85   UCSR0B = (1<<RXEN0)|(1<<TXEN0)|(1<<RXCIE0);
     /* Set frame format: 8data, 1stop bit */
87   UCSR0C = (3<<UCSZ00);

89   //stdout = &mystdout;
   }
91

   void USARTWriteChar(char data)
93 {
     if (data == '\n')
95    USARTWriteChar('\r');
     /* Wait for empty transmit buffer */
97   while ( !( UCSR0A & (1<<UDRE0)) ) {
       //Do nothing
99   }
```

```
  /* Put data into buffer, sends the data */
101  UDR0 = data;
}
```

ap/main/usart.c

B.3.5 timer.c

```
/*
 * timer.c
 *
 * Created: 11.04.2014 16:20:40
 *  Author: henrifo
 */


#include "defines.h"
#include "globals.h"

//flytt til timer.c
// Timer 1 input capture interrupt service routine
ISR (TIMER1_CAPT_vect)
{
  timer1_uShrt = ICR1;
}
//flytt til timer.c
// Timer overflow isr
ISR (TIMER1_OVF_vect)
{
  timer1_ovfcnt_uShrt++;
}


//flytt til timer.c
ISR (TIMER0_COMPA_vect)
{
  //send_freq -= 10;
  OCR0A = 0xFF;
  send_flag = TRUE;
}
```

```
34 int init_icp1_timer() {
   if (resolution==RESOLUTION_64)
36   TCCR1B  = (1 << CS11) | (1 << CS10);    // Timer clock = system clock / 64 (1 << ICNC1) <
   else if(resolution==RESOLUTION_256)
38   TCCR1B  = (1 << CS12);    // Prescaled to 256
   else if(resolution==RESOLUTION_1024)
40   TCCR1B  = (1 << CS12) | (1 << CS10);    // Prescaled to 256
   TIFR1 = 1 << ICF1;            // Clears ICF1 / pending interrupts
42 TIMSK1  = (1 << ICIE1 | 1 << TOIE1);    // Enable timer1 capture event interrupt
   DDRB  &=  ~(1<<PB0);            // Ensure PB0/ICP1 as input
44 return  0;
}
46
int init_timer0() {
48 OCR0A  = 0xFF;
   TCCR0B = (1 << CS02 | 1 << CS00); //1024 prescalar
50 TIMSK0 = (1 << OCIE0A); //Enable output compare match A interrupt
   TCCR0A = 1 << WGM00;//(1 << COM0A0 | 1 << WGM01 | 1 << WGM00);
52 //TIMSK0 = (1 << TOIE0);
}
```

ap/main/timer.c

Bibliography

[1] Atmel. The story of avr. https://www.youtube.com/watch?v=HrydNwAxbcY, 2008.

[2] Alan Burns and Andy Wellings. Real-time systems and programming languages, fourth edition, 2009.

[3] Libby Clark. Interview of steven rostedt. https://www.linux.com/news/featured-blogs/200-libby-clark/710319-intro-to-real-time-linux-for-embedded-developers, 2013.

[4] Beagleboard community. Beagleboarddebian wiki. http://elinux.org/BeagleBoardDebian.

[5] Beagleboard community. Beaglebone black, latest production files. http://elinux.org/Beagleboard:BeagleBoneBlack#LATEST_PRODUCTION_FILES_.28C.29.

[6] Pedro Côrte-Real. How much gnu is there in gnu/linux. http://pedrocr.pt/text/how-much-gnu-in-gnu-linux/, 2011.

[7] The Linux Foundation. Linux standard base. https://wiki.linuxfoundation.org/en/LSB.

[8] github user: jerrinsg. Freertos port for beaglebone. https://github.com/jerrinsg/RTOS_Bone.

[9] github user: steve kim. Freertos port for beaglebone black. https://github.com/steve-kim/BeagleBone.

[10] github user: wayling. porting freertos to beaglebone. https://github.com/wayling/FreeRTOS_Beaglebone.

[11] Thomas Gleixner. Cyclictest - high resolution test program, 2013.

[12] IDC. Arm to take 68% of embedded processor market. http://www.electronicsweekly.com/news/components/arm-to-take-68-of-embedded-processor-market-2013-02/, 2013.

[13] IEEE. 1003.1-2008 - standard for information technology - portable operating system interface (posix(r)). http://standards.ieee.org/findstds/standard/1003.1-2008.html, 2008.

[14] Texas Instruments. Am335x arm cortex-a8 microprocessors (mpus) technical reference manual (rev. j). http://www.ti.com/lit/pdf/spruh73, 2013.

[15] iTead Studio. itead studio, 2layer green pcb 10cm x 10cm. http://imall.iteadstudio.com/open-pcb/pcb-prototyping/im120418003.html.

[16] Mike Jones. What really happened on mars? http://research.microsoft.com/en-us/um/people/mbj/mars_pathfinder/mars_pathfinder.html, 1997.

[17] Kanaka Juvva. Real-time systems, 1998.

[18] Jason Kridner and BeagleBone community. Am335x pru package, 2014.

[19] Nicolas Melot. Study of an operating system: Freertos. http://stiff.univ-brest.fr/~boukhobza/images/stories/Documents/Teachings/OSM/expo/FreeRTOS_Melot.pdf.

[20] Robert C Nelson. Beaglebone black, linux-dev repository. https://github.com/RobertCNelson/linux-dev.

[21] Nongnu.org.

[22] QNX. The qnx neutrino microkernel. http://www.qnx.org.uk/developers/docs/6.4.0/neutrino/sys_arch/kernel.html.

[23] quik...@gmail.com. Cycletest done by a poster on google boards with a preempt linux v3.8. https://groups.google.com/d/msg/beagleboard/gJ_iFT2IwEQ/VITftfymIhQJ, 2014.

[24] Rusty Russell, Yanmin Zhang, Ingo Molnar, and David Sommerseth. Hackbench - scheduler benchmark/stress test, 2010.

[25] Peter Jay Salzman, Michael Burian, and Ori Pomerantz. The linux kernel module programming guide, v2.6.4. http://tldp.org/LDP/lkmpg/2.6/html/, 2007.

[26] Anand Lal Shimpi and Brian Klug. The bay trail preview: Intel atom z3770 tested. http://www.anandtech.com/show/7314/intel-baytrail-preview-intel-atom-z3770-tested/4, 2013.

[27] William Stallings. Operating systems, seventh edition, 2012.

[28] QNX Software Systems. Texas instruments am335x beaglebone/beaglebone black board support package. http://community.qnx.com/sf/wiki/do/viewPage/projects.bsp/wiki/TiAm335Beaglebone.

[29] Linus Torvalds. Re: [git pull] omap changes for v2.6.39 merge window. https://lkml.org/lkml/2011/3/17/492, 2011.

[30] Amos Waterland. Stress - workload generator, 2013.

[31] David A. Wheeler. Secure programming for linux and unix howto, 3rd edition, 2003.

[32] WormFood. Wormfood's avr baud rate calculator. http://www.wormfood.net/avrbaudcalc.php?postbitrate=38400&postclock=8&hidetables=1.