Norwegian University of
Science and Technology

Real-time Testing of Operating Systems on Raspberry Pi

Amund Murstad

Master of Science in Cybernetics and Robotics
Submission date: June 2016
Supervisor: Amund Skavhaug, ITK

Norwegian University of Science and Technology
Department of Engineering Cybernetics

Preface

This thesis is written as a part of the Masters program Engineering Cybernetics at the Norwegian University of Science and Technology in Trondheim, Norway during spring 2016. The idea came from Amund Skavhaug who had supervised Jadaan Diaa in 2015. He made a FPGA based real time tester. The task is to show how it could be used to test the response time of operating systems, exploring different approaches to how the system under test should be set up, taking advantage of the cheap computing power of the Raspberry Pi.

Trondheim, 2016-13-06

Amund Murstad

Acknowledgment

I would like to thank Amund Skavhaug for making it possible to do this project, and it has been a pleasure working with him. I would also like to thank Eirik Wold Solnør for letting me use the tester to test his KybOS.

Summary

A subset of operating systems are called "real time" operating systems. While a general operating system like Windows or OSX can make no guarantee when a process gets to run, a real time one has the ability to do so. They are used for systems where the quality of the results from the operating system is not only defined by their correctness, but also by *when* they arrive. These kinds of systems are called Real-time Systems and can be divided into soft, hard and firm. If a deadline is missed in a soft system, the usefulness declines gradually. In a hard system it goes to 0 immediately, an example being a frame in a video arriving after the next one has already arrived. Firm systems are hard systems where a missed deadline is catastrophic, as in nuclear systems or some medical equipment.

This project explores the possibility to test an operating system for these kinds of requirements. It does so by using a tester developed in 2015 at NUTS by Jadaan Diaa [10]. The system that has been tested is the Raspberry Pi 3 with Ubuntu Mate as its operating system. The desire to use the Raspberry Pi comes from its great computing power in regards to price, and its easy to use methods of interacting with other equipment. These tests are carried out as a electronic ping-pong game, where the tester says "ping" and waits for the operating system to answer "pong". Different methods of setting up the operating system has been explored, using a spectrum of different methods ranging from high energy, CPU consuming methods, to CPU friendly ones.

The results show that the results improve by increasing the amount of CPU time used. This is in line with expectations. It also shows that Ubuntu Mate is not suitable for being a real time operating system, as its responses come at non-deterministic times, often being very late. Another type of test shows how the tester can be used to calculate the time it takes between to points in code, but the results here show that there is some overhead such that results need to be calculated relative to a base-case. The project has shown that the tester made by Diaa works and could be used to classify/test operating systems for real time properties.

Sammendrag

En undergruppe av operativsystemer kalles sanntids-operativsystemer. Mens generelle operativsystemer som Windows eller OSX ikke kan garantere noe om når en prosess får kjøre, kan et sanntids-operativsystem gjøre dette. De blir brukt hvor systemer hvor kvaliteten på resultater fra operativsystemet ikke bare er definert av dets korrekthet, men også av tiden det ankommer. Disse typen systemer blir kalt sanntidssystemer, og kan bli delt opp i myk, hard og streng. Hvis en operativsystemet bommer på en frist i et mykt system mister resultatet gradvis verdien sin. I et hardt system går verdien umiddelbart til 0. Et eksempel her er dersom et bilde i en videostrøm kommer etter de neste bildene alt har kommet så er det verdiløst. Strenge systemer er harde systemer hvor det å bomme på fristen leder til katastrofe, slik som for eksempel atomreaktorer eller medisinske aparater.

Dette prosjektet utforsker muligheten til å teste et operativsystem for disse typene krav. Det gjør så ved å bruke en tester utviklet av Jadaan Diaa[10] på NTNU i 2015. Systemet som blir testet er en Raspberry Pi 3 med Ubuntu Mate som operativsystem. Ønsket om å bruke Ubuntu Mate kommer fra den kraftige beregningskraften man får for lite penger, og dens enkelhet med tanke på tilkoblingsmuligheter for annet utstyr. Disse testene blir utført som elektronisk ping-pong, hvor testeren sier "ping" og venter på at operativsystemet skal svare "pong". Forskjellige metoder for å sette opp operativsystemet til å respondere på har blitt utforsket, gjennom bruk av forskjellige metoder varierende fra metoder med høyt energiforbruk, til CPU-vennlige metoder.

Resultatene viser at resultatene fra testene blir bedre av å bruke mer CPU og energi. Dette er i tråd med forventningene. De viser også at Ubuntu Mate ikke er passelig som et sanntids operativsystem etter som responsene er for uforutsigbare og ofte veldig trege. En annen type test viser hvordan testeren også kan bli brukt til å kalkluere tiden det tar mellom to punkter i koden, men resultatet viser også at det er en del overhead som gjør at resultatet er nødt til å kalkuleres relativt til et base-nivå. Prosjektet har vist at testeren laget av Diaa virker og kan bli brukt til å klasifisere/teste operativsystemer for sanntidsegenskaper.

Table of Contents

Preface i
Acknowledgment iii
Summary iv
Sammendrag v

1 Introduction **2**
1.1 Motivation 2
1.2 Previous work 3
1.3 This project 3

2 System on Chip **4**
2.1 FPGA 5
2.2 MCU 6
2.3 SUT 6
2.4 Log Files 6

3 Project setup **8**
3.1 Raspberry Pi 8
3.2 Operating system 10
3.3 Software 11

4 Data logging **15**

4.1 Self test 16
4.2 Multiple threads 16
4.3 Multi-core 17
4.4 Mutex 18

5 Response Testing **19**
5.1 Methods 20
5.1.1 Big While 20
5.1.2 Threaded 21
5.1.3 Interrupt 22

6 Results of response time testing **23**
6.1 Polling 23
6.1.1 Same core 25
6.1.2 Multiple cores 26
6.1.3 High Priority 27
6.2 Interrupt 28
6.3 KybOS 30

7 Discussion **31**
7.1 Process 31
7.2 Results 33
7.3 Future Work 34
7.4 Concluding remarks 35

A Acronyms **36**

B Code **38**
B.1 For testing the OS 38
B.2 For analyzing .rts files 45

Bibliography **50**

Chapter 1

Introduction

1.1 Motivation

Operating systems(OSs) are an often overlooked part of everyday life in 2016. Most know about the really big ones, Windows, OS X, iOS and Android, and Linux. But these does not cover every system in the modern world. Space shuttles, remote controls and micro wave ovens does not necessarily use any of these. These systems have requirements that in some way the big OSes can not deliver. The correctness of these systems, that is if it works as intended or not, is very much dependent on the OSs not just giving the right outputs, but also the time at which they arrive. We say that these kind of systems have *Real time demands* and call the operating systems that can meet these demands for *Real time systems*(RTS).[8] These kind of OSs are all around us, but are more often than not embedded inside a system with no standard user interface.

When developing these kinds of systems it is important to verify its behaviour, both in normal and in worst case scenarios. We do not want the engine of our airplane to slow down just because the OS has a lot to do at the moment serving video for the passengers. In a 'normal' OS, like Windows 10, the behaviour of slowing down is normal and acceptable at times. The OS can tell a process 'Sorry, you just have to wait', even though it really needs to get it done now. For a RTS, the non determinism in task completion is not acceptable. Therefore we need a way to test it so we can guarantee it. This needs to be done

by a specialised tester that can measure the timings of the system. The data can then be analysed and we could find if it meets our demands and specifications.

1.2 Previous work

Since we want to interfere with the system under test as little as possible introducing time management code directly in to its code is not desirable. It is much better to use an external device which can handle the time management, and just induce tiny pieces of code in to the system. The kind of equipment needed to test real time systems this way exists in the form of logic/PIC-analysers, but they are mostly specialised for a specific type of system and are expensive. The desire for a more universal system for testing, that is also cheap, lead to a master thesis here at NUTS in 2015. In his thesis, Jaadan Diaa created a system on chip with the capability to do these kinds of test using only GPIO-pins which most microcontrollers have[10]. His system came in costing just under €50, which meat the goal of the project. Diaas work is the basis of which this project is built on. His thesis was built on top of Kyrre Gonsholts work of implementing a time management unit in Verilog for using on a FPGA[9]. Kyrres solution sadly did not work when transferred to hardware. Diaa managed to find the error and transfer it to a FPGA and build the rest of the system around it.

1.3 This project

The task of this project is to explore how we can use a real-time tester to test operating systems and say something about their real-time capabilities. The tester in use will be the one created by Diaa, and his work will lay the basics. The goal is to look at different ways of implementing respond-programs on the system under test, compare their results and explain why the different methods used are better than others.

Chapter 2

System on Chip

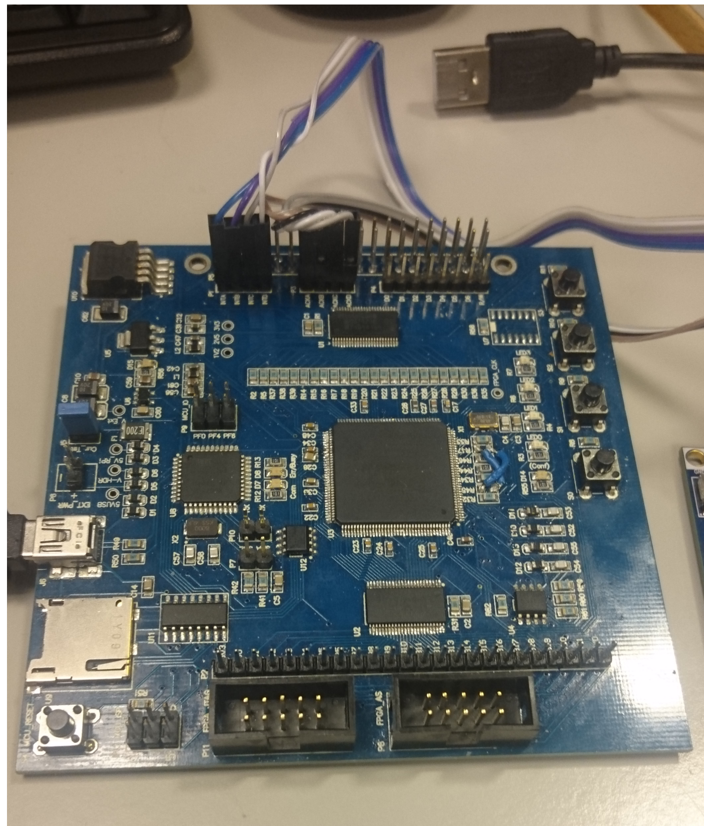

Figure 2.1: A picture of the tester .

The board in figure 2.1 is the board created by Diaa and will be referred to as 'the tester' in this report. In figure 2.2 we can see the different parts of the system. The PC controls the tester and is connected through a USB cable. Internally the MCU is communicating

with the SD card by SPI, to communicate with the FPGA UART is used and USB is going to the PC.

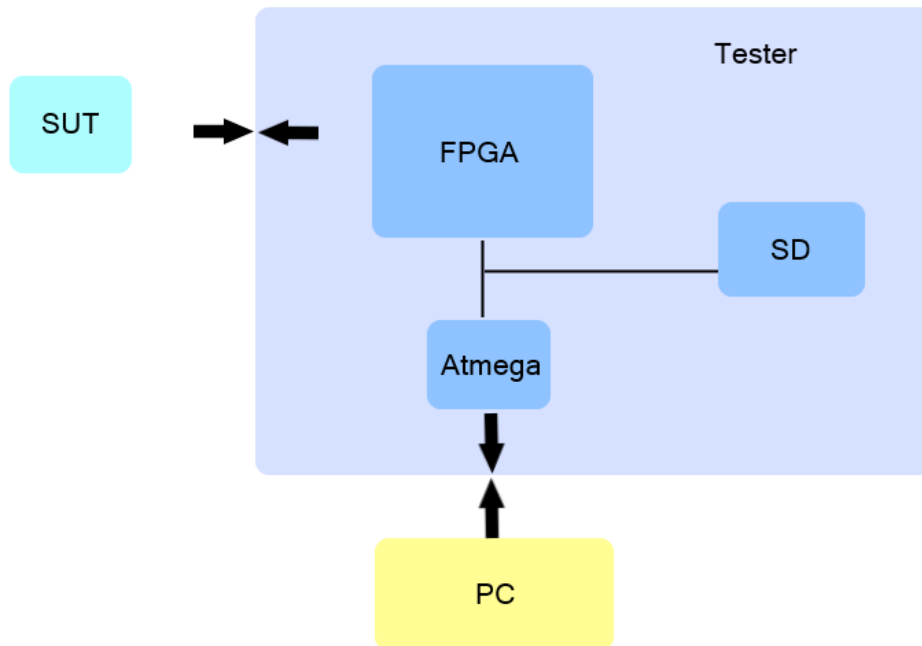

Figure 2.2: An overview of the systems modules.

2.1 FPGA

The FPGA is the core of the system. On this chip is the TMU IP of Gonsholt with slight modifications made by Diaa. The FPGA mounted is made by Altera and is of type Cyclone IV EP4CE15E22. The main reason this was chose was the low price of 23 Euro and it having just enough logic elements and memory while still being reasonably fast. Choosing this came with the drawback of having to add a soft core and hoping that the system, which was developed for a hard code, would still work.

2.2 MCU

For communication to a PC the tester need a chip between the PC and the FPGA, due to the PC most likely being much faster than the NIOS processor on the FPGA. A MCU handles the task of accessing the SD card and talking to the PC through USB. An Atmega32u4 MCU from Atmel is used. To access the USB-functionality in it an open source library called LUFA is used. The software loaded on to the MCU is simulating a COM port that the PC can connect to. This makes it simple to send commands to and from the tester, as they are simply ASCII characters.

2.3 SUT

The system under test is connected through ports on the side of the tester. There are three ports which it can connect to.

Table 2.1: The pins for connecting a system to be tested

Port	**PINS**							
P1	INTA	INTB	INTC	INTD				
P3	ACKA	ACKB	ACKC	ACKD				
P4	D0	D1	D2	D3	D4	D5	D6	R/W

Port P1 is an output from the tester while the other two are inputs. For an interrupt-acknowledgement test P1 and P2 are used. P4 is used for the system under test to trigger logging when it wants to instead of the tester controlling it as it is with the former method. It does this by putting a label on the data pins then flashing the R/W. It is important that P3, the ACK-port, is driven low when not active. Leaving this floating will make the interrupt pins not trigger, as ACK is prioritised over INT.

2.4 Log Files

The output of the tester is lines upon lines of binary code which the software pulls in to a text file (even though the ending .rts was used it is still just plain text). To make sense of

the data we first have to look at what all the bits and bytes mean. Each line of data is 64 bits long and should be split up in to pieces the way it is shown in table 2.2 to make interpret the data correctly.

Table 2.2: Formatting of data line from tester

	StartBit	**EndBit**	**Length**
Start of Frame	0	3	4
Data Type	4	5	2
Active Interrupts	6	9	4
Active ACK	10	13	4
Data	14	20	7
Timestamp	21	63	43

Chapter 3

Project setup

3.1 Raspberry Pi

Raspberry Pi is a series of mini-PCs created by the Raspberry Pi Foundation in England. Their goal is to make a cheap, power efficient, modular PC for usage in education and developing countries. It quickly got a huge following with fans of electronics due to its low cost, Linux operating system and good options of connecting peripherals. By February 2016, over 8 million units have been sold [1]. The models all share a number of GPIO(General Purpose Input-Output)-pins, USB-ports, Ethernet and memory card. The first iterations used a standard SD-card which over the generations have been replaced by the smaller microSD-card. This card is used as a system disk and contains the operating system.

Table 3.1: Comparison of the Raspberry Pis [2]

Type	CPU	Clock Frequency	Price	Ram	New from previous version
Model B	ARMv6	700MHz	$30	512MB	
Model 2	ARMv7	900MHz	$40	1GB	4 USB
Model 3	ARM Cortex-A53	1.2GHz	$40	1GB	WiFi og Bluetooth

The Raspberry Pi has 26 pins which can be seen in figure 3.1. 9 of them are power pins, 5 ground (0V), 2x3.3V and 2x5V. The 17 left over are GPIO-pins. Some of them can be used for specialised operations such as I2C or SPI-interface while others are only used as programmable pins. Listed in the figure are the different ways one can count the pins. The

P1: The Main GPIO connector							
WiringPi Pin	BCM GPIO	Name	Header		Name	BCM GPIO	WiringPi Pin
		3.3v	1	2	5v		
8	Rv1:0 - Rv2:2	SDA	3	4	5v		
9	Rv1:1 - Rv2:3	SCL	5	6	0v		
7	4	GPIO7	7	8	TxD	14	15
		0v	9	10	RxD	15	16
0	17	GPIO0	11	12	GPIO1	18	1
2	Rv1:21 - Rv2:27	GPIO2	13	14	0v		
3	22	GPIO3	15	16	GPIO4	23	4
		3.3v	17	18	GPIO5	24	5
12	10	MOSI	19	20	0v		
13	9	MISO	21	22	GPIO6	25	6
14	11	SCLK	23	24	CE0	8	10
		0v	25	26	CE1	7	11
WiringPi Pin	BCM GPIO	Name	Header		Name	BCM GPIO	WiringPi Pin

Figure 3.1: Pin nummerering

innermost is the natural counting method. The next is the BroadCom chip numbering, referencing the actual pin values on the BroadCom chip. The last one is the numbers used by the wiringPi module included on Raspberry Pi operating systems for interfacing with the GPIO-pins. All pins inn this project will refer to the wiringPi-numbers.

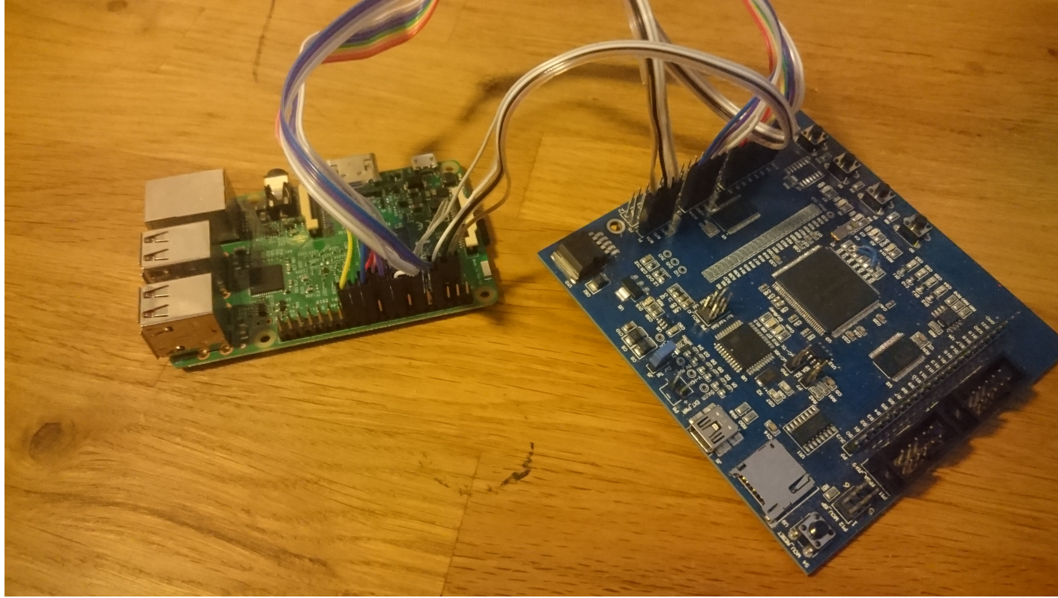

Figure 3.2: Raspberry Pi with the tester. The pins used are noted in Appendix B.1

3.2 Operating system

Most available operating systems for the Raspberry Pi are based on a form of Linux. Linux is used by many enthusiasts, and the choice to provide versions for Raspberry Pi has a lot of the honor for the great position the product is at today. The group behind it have made their own OS, called Raspbian, which is the only official operating system. This does not mean that they do not support others. They provide easy links and methods of getting operating systems made by 3rd parties, emphasising their willingness to help make a more open community. In this project Ubuntu Mate was used. This is a lightweight version of Ubuntu, which is the most adapted version of Linux today. The Mate-part of it is the desktop, simplifying it to make it run on weaker units, such as the Raspberry Pi. This results in a known environment for someone who is used to Ubuntu, while not being too slow on the Raspberry Pi. The requirements are Pentium 3 750Mhz, 8GB SD and 512MBs of ram. These are all well inn the range of the Raspberry Pi 3. All operating systems can be downloaded from their website [4].

3.3 Software

The graphical desktop of Ubuntu Mate is fully functional, but it runs on a low resolution and it feels slow on handling inputs. Therefore it was desirable to find a method to control and write files to the Raspberry Pi without using the graphical desktop. The solution was found using SSH for command line interface and SCP(based on SSH) for file management. As my main computer is running Windows, I looked for programs that could connect using these protocols. The program that was chosen was WinSCP[7]. This program connects to the remote target and gives a graphical representation of its file system as if it was a local folder, as can bee seen in figure 3.4. The program features a simple mode to share the SSH-session with PuTTY, a SSH-client for Windows. The PuTTY-window simulates the terminal window from Linux, as is shown in figure 3.3

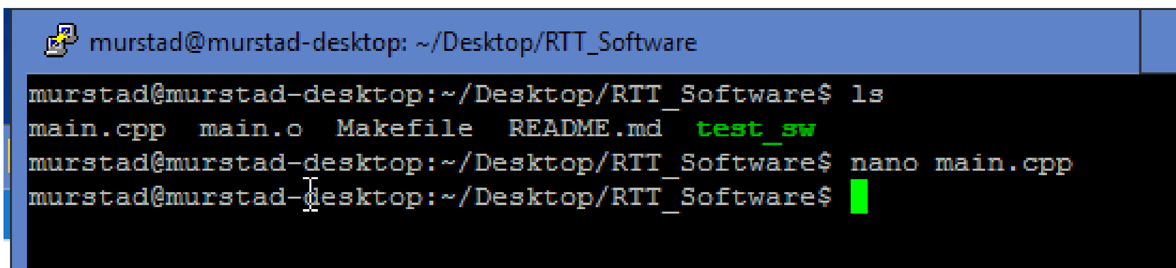

Figure 3.3: PuTTY simulates a terminal window known from the Unix world

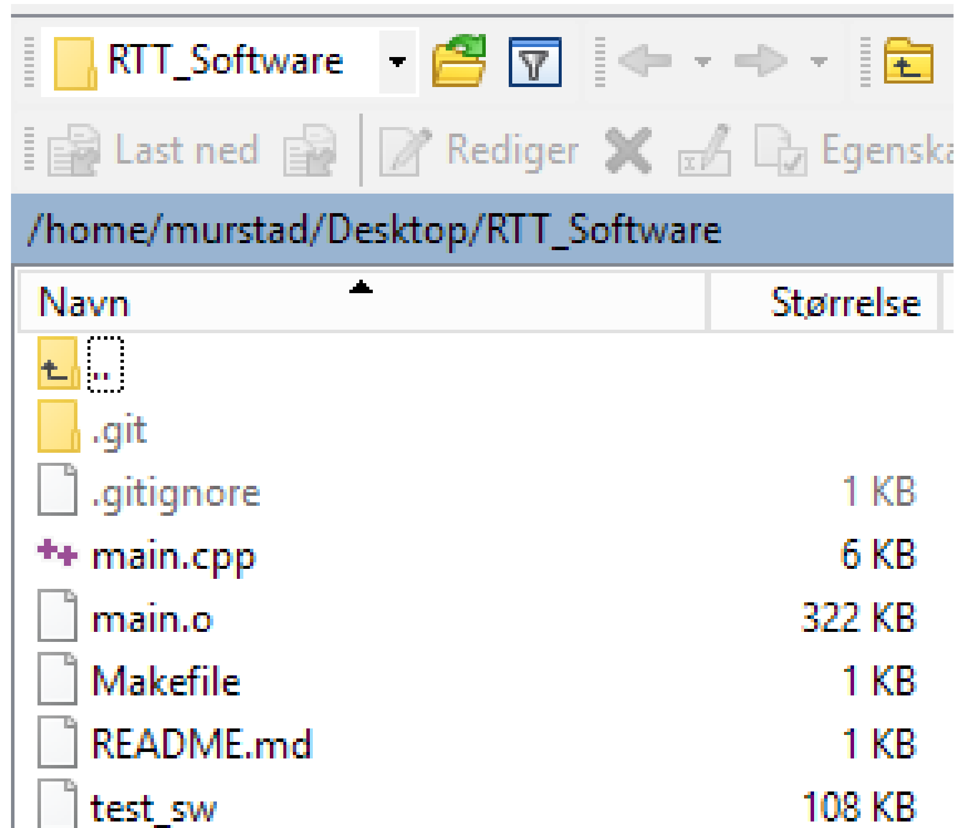

Figure 3.4: The file system of the Raspberry Pi as viewed from WinSCP

One of the best things about WinSCP is that it is very customisable in terms of how it manages the remote files. If one of the files is opened, it creates a temporary file that the user can work on. Every time it is saved, it is transferred to the Raspberry Pi. It features methods to select which program each type of files should be opened with as seen in 3.5. This makes it easy to work with and one is able to use its favourite editor for each type of file. In figure 3.6

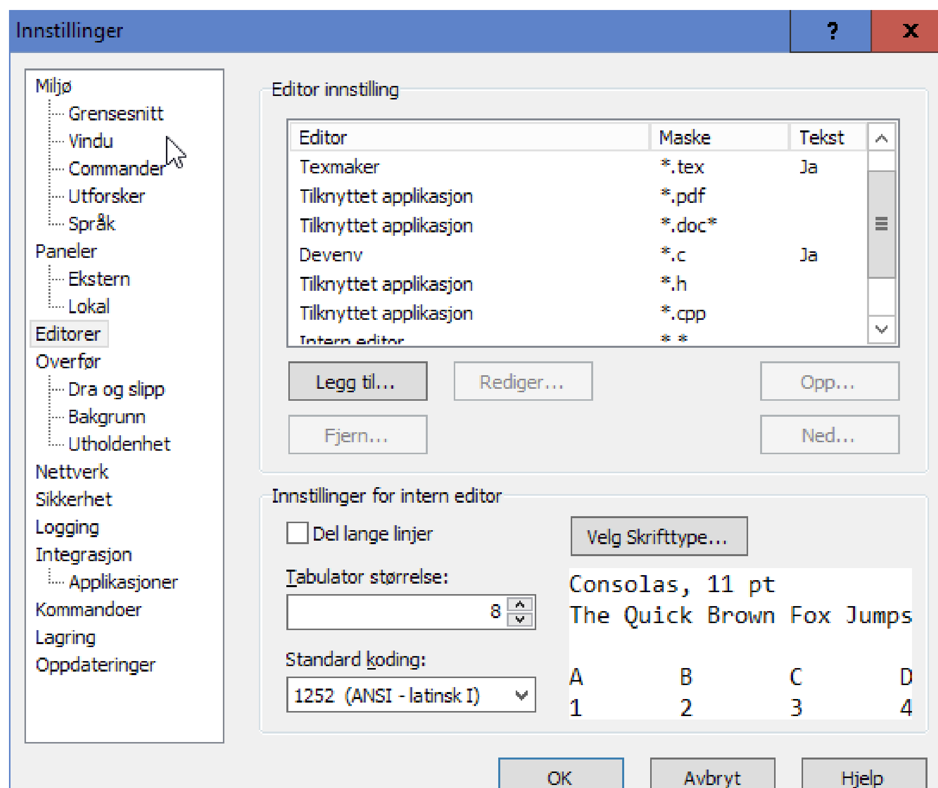

Figure 3.5: The settings screen for WinSCP. It features filters for which application should open each type of file.

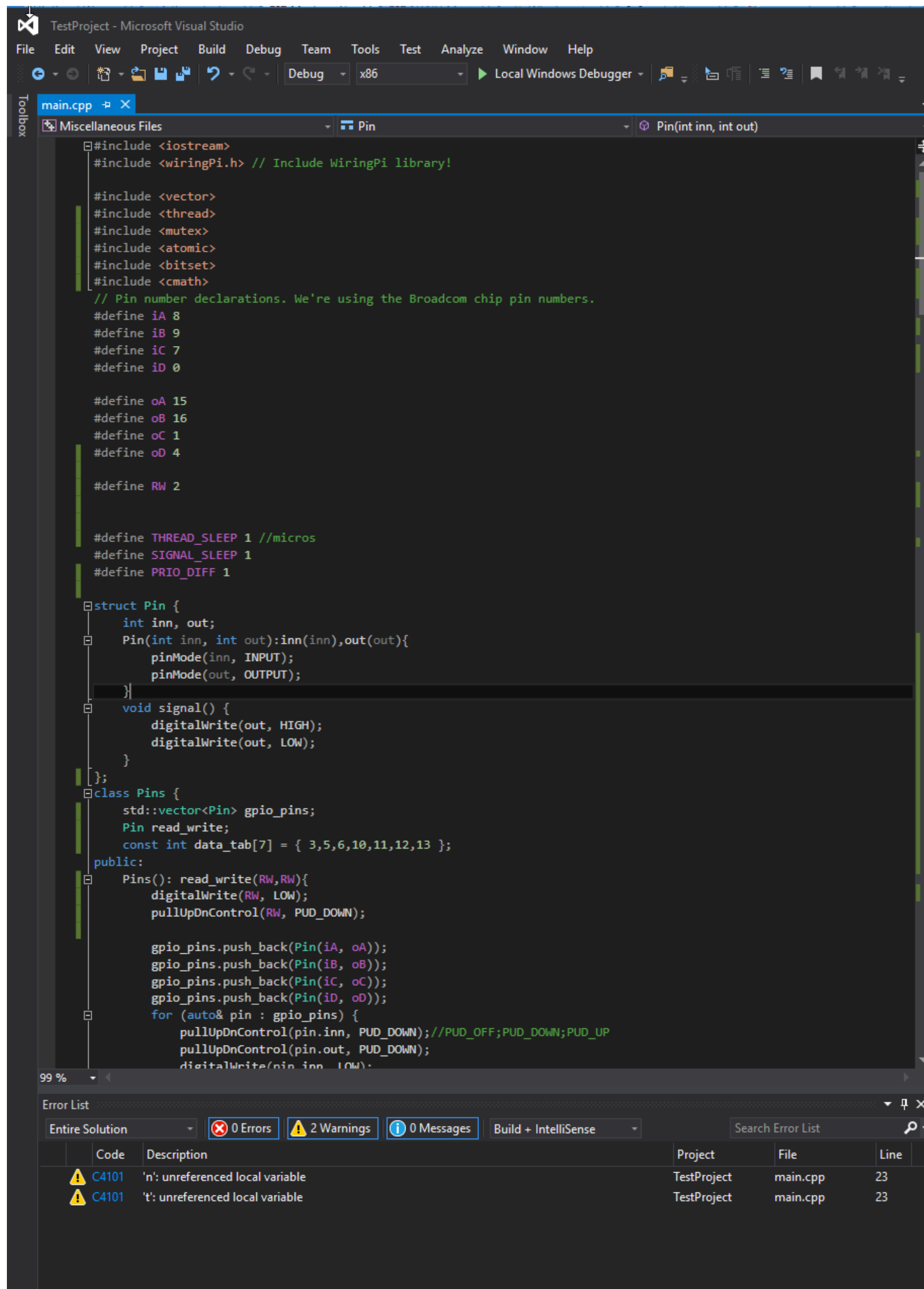

Figure 3.6: Visual Studio Community 2015 is used as the main editor for the C++ files

Chapter 4

Data logging

The tester can be told to do a log externally by toggling the R/W-pin of the system. This makes the tester log an event, and will save the state of pins D0-6 as a string of bits. This is referred to as a “Tag”. This method can be used to time the execution of a critical part of the code, the time it takes for the operating system to switch between threads or more general logging of events in code. This makes the tester a product that can be used in many different applications, not just as a response tester. To reliably time code, it is needed to find how much time the testing itself takes, as there is some overhead. The operations needed to log a given tag is given below:

1. Convert the tag to binary form
2. Iterate through the pins, setting them high or low dependant on the bit value in the tag
3. Toggle the R/W-pin to signal that a log should be performed
4. Make sure that the operating system can schedule other processes

This is not a non trivial amount of code that needs to be run for a logging. If it is known that the system only uses one tag, it can be sped up by skipping step 1 and 2.

4.1 Self test

By running code performing the task above, the results are given as below.4.1.:

Table 4.1: Self test of data logging

Type	Tag	Timestamp
11	0000001	22
11	0000001	53
11	0000001	83

This test is performed with a scaling of 3 and a clock cycle of 20ns, giving the time used as $(53-23)*20ns*3 = 1800ns$ for one logging. The thread is ran with a standard priority, and results can such be improved by raising its priority. Doing this yields a result of $27*3*20ns = 1620ns$, a minor improvement. This shows that most of the time used is used by the code itself, not by the scheduling.

4.2 Multiple threads

When laying out multiple threads on the same core, it is desired that they should all get to do their job, not blocking the other tasks. This switch between threads is not trivial, as a context switch has to be performed, saving the state of the thread and resuming another. These switches are done all the time by modern CPUs as they want to simulate work being done simultaneously. By adding a couple of threads that does the same as the self test above this switching can be observed. The threads uses individual tags, and has a tag number equal to two to the power of n (such that thread 0 uses tag 1, thread 1 uses 10 and so on). The code is included in Appendix B.1.

Table 4.2: Self test of data logging on the same core

Type	Tag	Timestamp
11	0000100	22119
11	0000010	22162
11	0000001	22205
11	0001000	22248
11	0000100	22291

From table 4.2 it is observed that the time between two log-points has gone up. A log is now performed every $43 * 3 * 20 = 2580ns$. But this is between two logs, not necessarily between to logs with the same label. As the table shows, two logs with the same tag now has a $(22291 - 22119) * 3 * 20ns = 10320ns$. Since the scheduler is set to round-robin, the tasks are all executed the same amount of times. The extra execution time does not only come from the extra amount of threads, as there are 3 new threads. Given the time found in 4.1, this should amount to $32 * 3 = 94$ extra cycles and the results give $172 - 32 = 140$ extra cycles. This can only be because the operating system now has a more difficult time scheduling and is using more time swapping between the threads.

4.3 Multi-core

To improve the results from the previous test it is possible to lay the different threads out on different cores. The Raspberry Pi has 4 cores, which allows us to run 4 threads on their own core simulating them running alone. Now concurrency is not only simulated. The threads are now executed on different physical cores at the same time. This could make the results more non deterministic as there are other threads that need to run in order for the Raspberry Pi to function correctly. Previously these threads could be switch to one of the cores that wasn't busy, but now they are all busy. This means that even though the threads want to run continuously, the operating system has to chose one of them to be scheduled later such that an important processes gets the chance to run. This situation leads to not all four threads getting to run the same amount of times, as some might have to fight more for CPU-time on the core.

Table 4.3: Excerpt from test data for threads on multiple cores

Type	Tag	Timestamp
11	0000000	5237
11	0000001	5242
11	0000110	5251

This data shows a challenge that did not exist when all the threads were ran on the same core. In table 4.3 there are two logs of special interest. The Tags 0000000 and 0000110

are not tags that any of the threads use, so the question then arises of who has written them. The case here is that two(or more) of the threads on their own core has been changing the states of the data-pins at the same time, yielding a undefined result. From the two first lines it looks like there is only 5 cycles between the logs, a time short enough that process 2 could have been under way of setting its data up on the pins while process 1 is triggering the R/W-pin.

4.4 Mutex

The part of the code that sets the pins is called a critical section, and has to be guarded against multiple processes entering at the same time. This is solved by adding a mutex as a guard. A mutex is a lock that only can be opened by the one who locks it. Other threads has to wait until it is unlocked before they can attempt locking it themselves. If N threads are all waiting to lock a mutex, it is not given who gets to lock it when it is unlocked, it is first come first serve. This means that there is still the same property of not knowing how many times one of the tests run, but now it is guaranteed that only legal tags are written to the data-pins. The price to pay for this is the extra overhead of working with a mutex-object. By reducing the amount of threads back to 1 we can compare the timings with mutex against the timings of the self test. The test in table 4.4 show that it uses 2 cycles more, which amounts to 120ns extra for the mutex-object.

Table 4.4: Self test with mutex

Type	Tag	Timestamp
11	0000001	32
11	0000001	64
11	0000001	96
11	0000001	128

Chapter 5

Response Testing

The purpose of this test is to see how fast we can get a response from the operating system, and how reliably we can tell when the response is coming. This information is useful when deciding the worst case and average case responses of a system, and telling something about its real time properties. The method used is the interrupt to acknowledgement method as seen in Figure 5.1. The result is the time it takes from the tester to put out a signal to it registering a result.

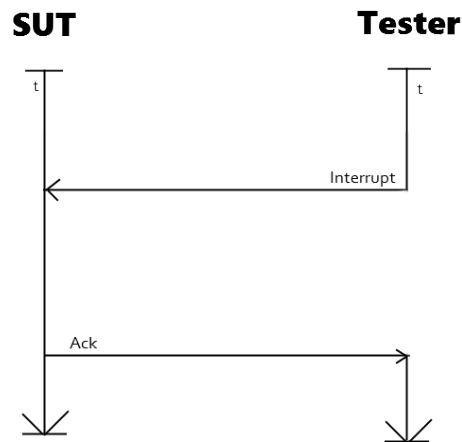

Figure 5.1: Interrupt to Acknowledgement

5.1 Methods

For the system under test, there are several different ways we can set it up to respond to the signals. The trade off is often between speed and usage of CPU. If it is known that the operating system is only going to be used for these actions, maxing out the CPU at all times will give the best results. But this will also use a lot more power and hence also produce more heat. These properties can be dealt with in certain situations, but not every. Therefore it is crucial that we also have some methods that does not hog the CPU, only working when necessary or at some interval.

5.1.1 Big While

The "Big While"-method is the first one to be looked at, and consists of one loop. This method includes hogging one core of the CPU at all time, continuously checking if there are things to be done. For a test with two pins, the process is as follows:

1. Check if input pin A is high
2. If it was high, answer the tester by setting response pin A high
3. Check if input pin B is high
4. If it was high, answer the tester by setting response pin B high
5. Go back to top and check again

Observing this method leaves us with a couple of notes. For one, since we do not have any sleeps or are yielding in any way, we run as often as the scheduler allows us. The response time of a signal is also highly dependant on where in the code we currently are. If the signal goes high just before we check it, the result will be very good. If it goes high just after we check it, it will not be answered before all the other pins are checked and the code executes the next iteration of the loop. This method will always be executed on one core, and since the Raspberry Pie has 4 cores, it will never lock completely up as the operating system can move other processes to other cores.

5.1.2 Threaded

The Big While has the problem of being non deterministic in its results since the response is dependant on where in the code it currently is. This can be improved upon by splitting up the work in to different threads and laying them out on different cores. By doing this and not yielding, the CPU will be maxed out on all cores at all time, leaving other processes with few chances to run, which might be problematic. By letting each thread yield or sleep, the other processes can run from time to time. This will result in a system that actually can be used for anything else, but will make the responses be dependant on if the core is currently running the thread or something else, and when it eventually will schedule in the thread again. By managing the priority of the threads the result can be even further improved, but this will make other threads not able to run and hence locking up the system.

1. Spawn N threads (where N is the amount of pins in use)
2. Assign each of them with a process that responds to the corresponding pin
3. Bind them to a core
4. Wait until testing is done
5. Signal all the threads that they should stop

5.1.3 Interrupt

Both the previously mentioned methods uses all of the CPU (one core or all 4). With interrupts it can be managed so that in userland, the code uses almost none of the CPU. This is the interrupt-method. Interrupts on Linux is something reserved for the kernel. When an interrupt is triggered, the kernel puts away what it was doing, handles the interrupt, then resumes its work. This method consists of the kernel triggering a sleeping thread when it receives an interrupt. All we need to do from user space is to register with the kernel what thread is to be executed when an interrupt on a specific pin is triggered.

1. Spawn N threads (where N is the amount of pins in use)
2. Register them with the kernel and tell what interrupt to look for
3. Wait until testing is done
4. Signal all the threads that they should stop

Chapter 6

Results of response time testing

The basic big-while. The Pins class is shown in appendix B.1

```
void bigWhile() {
  Pins system_pins;
  while (1) {
    for (int i = 0; i < 4; ++i) {
      if (digitalRead(system_pins[i].inn)) {
        system_pins[i].signal();
      }
    }
    std::this_thread::sleep_for(std::chrono::microseconds(THREAD_SLEEP));
  }
}
```

6.1 Polling

Putting all the threads on the same core as suggested in section 5.1.2 gives the code shown bellow. Running this on multiple cores uses 100% of all the cores, resulting slow ssh-processing. By setting the priority higher than normal, ssh functionality was turned completely off. This removes all possibilities of controlling the Raspberry PI over ssh, and such a hard reset is needed (which Ubuntu does not like as discussed in 7.1). Since the scheduling used was round robin, giving this thread a higher priority than the other threads, the

yield in the function does not matter as it is scheduled in again immediately. The results in this section is generated with Microsoft Excels histogram-function.

```
void test_thread(Pin p) {
  while (barrier) {}
  static int thread_nr = 0;
  //set_cpu(thread_nr); //use this to put each thread on new core
  thread_nr = (thread_nr + 1) % 4;
  while (running) {
    std::this_thread::yield();
    if (digitalRead(p.inn))
    {
      p.signal();
    }
  }
}

void threaded() {
  Pins pinner;
  std::vector<std::thread> threads;
  for (int i = 0; i <4; ++i)
    threads.push_back(std::thread(test_thread, pinner[i]));
  int prio = 10;
  //for (auto& i : threads)
    //set_thread_priority(i, prio);
  barrier = false;
  char a;
  std::cin >> a; //press any key to exit
  running = false;
  for (int i = 0; i < 4; ++i)
    threads[i].join();
}
```

6.1.1 Same core

Running the test on one core with normal priority 50 times and compiling together the results yields the graph shown in 6.1. From these numbers we can see that the operating system sometimes uses a long time to answer. This comes from another thread running, and the correct thread not being able to run again for some while. The CPU usage for this is one core on 100% and the rest unaffected, which means that normal system operations can be maintained.

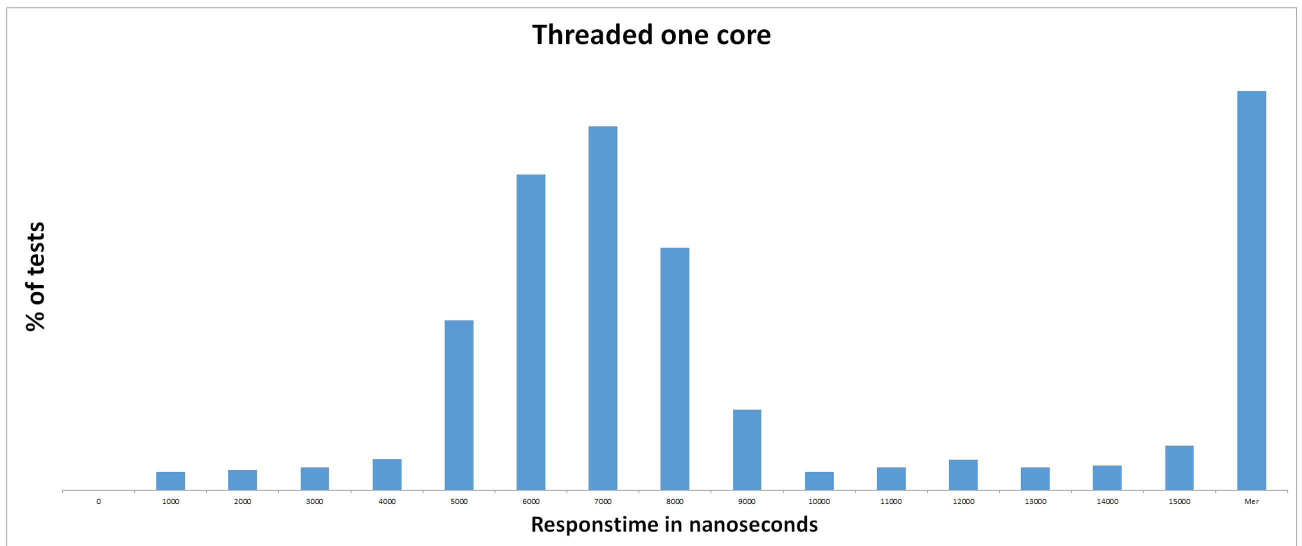

Figure 6.1: Test results on one core, multithreaded

6.1.2 Multiple cores

Running the test on multiple cores with normal priority 50 times and compiling together the results yields the graph shown in 6.2. This test uses 100% of all the cores, and yields a better result than the previous test. Still we can observe that the system sometimes fails to answer in time and that the system is not consistently responding in the same time-frame. This kind of result would be more acceptable for a soft real time system, as few responses are really late, but there are some. In a video-example this would mean that some frames are dropped (which is normal for online streaming).

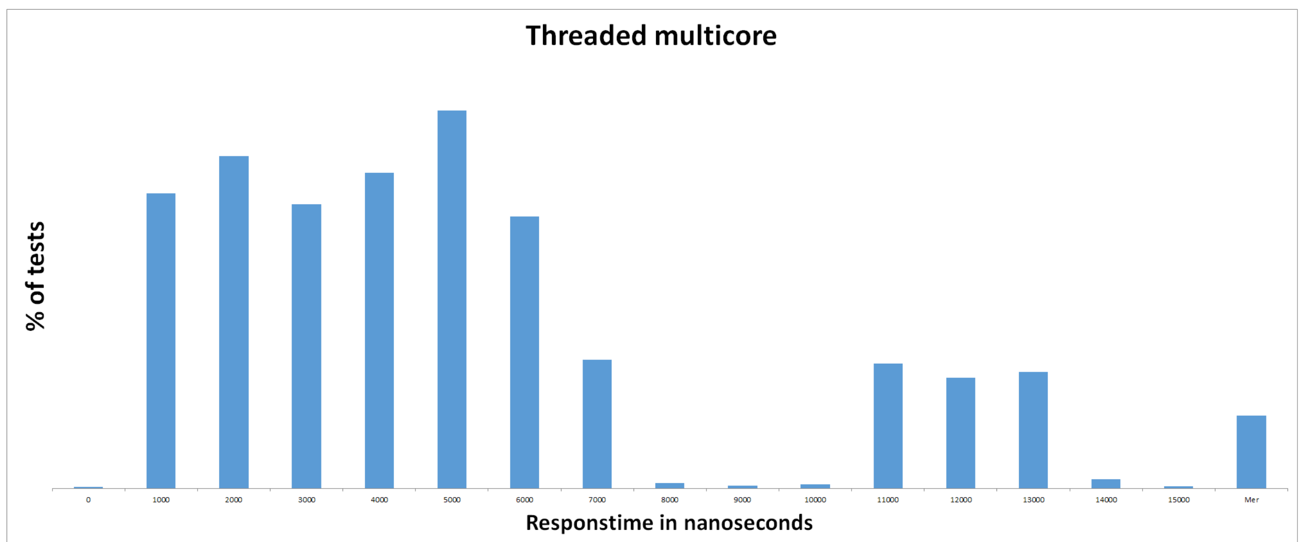

Figure 6.2: Test results on four cores

6.1.3 High Priority

Running the test on multiple cores with high priority 50 times and compiling together the results yields the graph shown in 6.3. This test uses 100% of all the cores, and locks up the system completely, requiring a hard reset. We can observe that this made all the difference. Now, since the threads are running at a higher priority, they do not have to compete with as many other tasks for getting time to run on the CPU. We still see that there are cases where the response time is large, since there are some system operations that have a higher priority than the threads. This result is the best we can get out of Ubuntu on a Raspberry PI, as it is not a real time operating system.

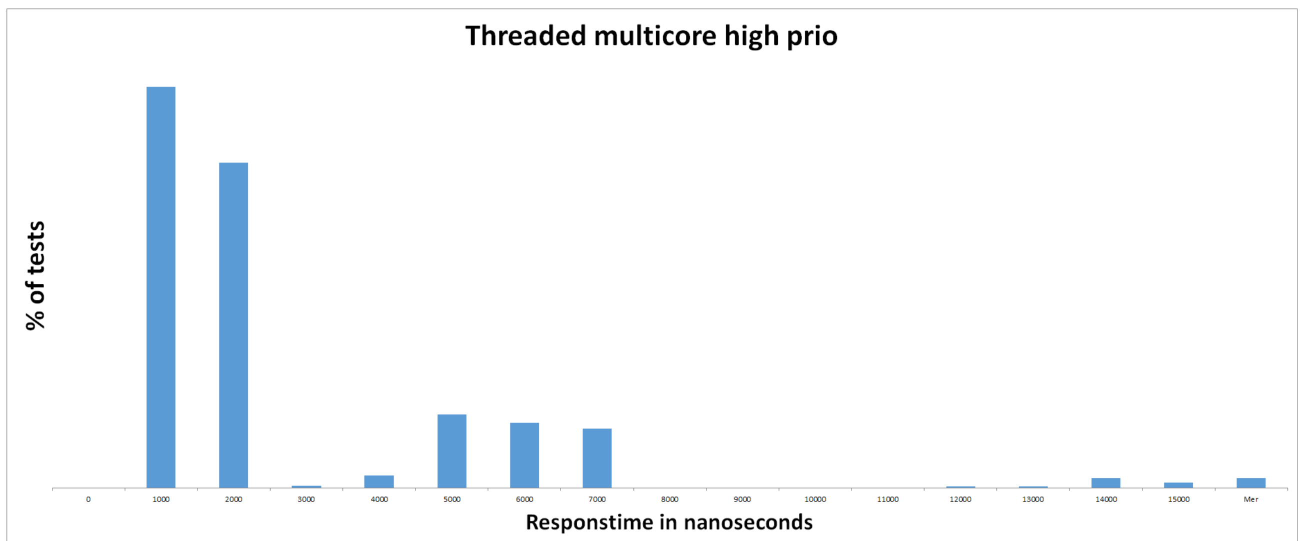

Figure 6.3: Test results on four cores

6.2 Interrupt

The setup for this test is listet below. WiringPI makes it easy to register the different tasks for the different pins, then set it in a loop that does nothing (and uses little CPU). This code runs with a CPU usage of 0.0-0.1% on one core, making it nice on the rest of the system.

```
void ansA() {
  digitalWrite(oA, HIGH);
  digitalWrite(oA, LOW);
}
void ansB() {
  digitalWrite(oB, HIGH);
  digitalWrite(oB, LOW);
}
void ansC() {
  digitalWrite(oC, HIGH);
  digitalWrite(oC, LOW);
}
void ansD() {
  digitalWrite(oD, HIGH);
  digitalWrite(oD, LOW);
}
void inter_driven() {
  Pins pinner;
  wiringPiISR(iA, INT_EDGE_RISING, &ansA);
  wiringPiISR(iB, INT_EDGE_RISING, &ansB);
  wiringPiISR(iC, INT_EDGE_RISING, &ansC);
  wiringPiISR(iD, INT_EDGE_RISING, &ansD);
  while (1) {
    std::this_thread::sleep_for(std::chrono::microseconds(1));
  }
}
```

Running the test with interrupts 50 times and compiling together the results yields the graph shown in 6.4. From these numbers we can see that even though this method should be fast, it is not. This might be wiringPis method of handling interrupts that fails, or it

might be the kernel taking a long time to pause the current thread, handle the interrupt, then bringing back the thread that was running. Going back to wiringPis source it states: "The function will be called when the interrupt triggers. When it is triggered, it's cleared in the dispatcher before calling your function, so if a subsequent interrupt fires before you finish your handler, then it won't be missed. (However it can only track one more interrupt, if more than one interrupt fires while one is being handled then they will be ignored) ".[6] This would mean that if more interrupts are triggered while one is already being handled in the kernel, they might not be recognised until much later, which could be the source.

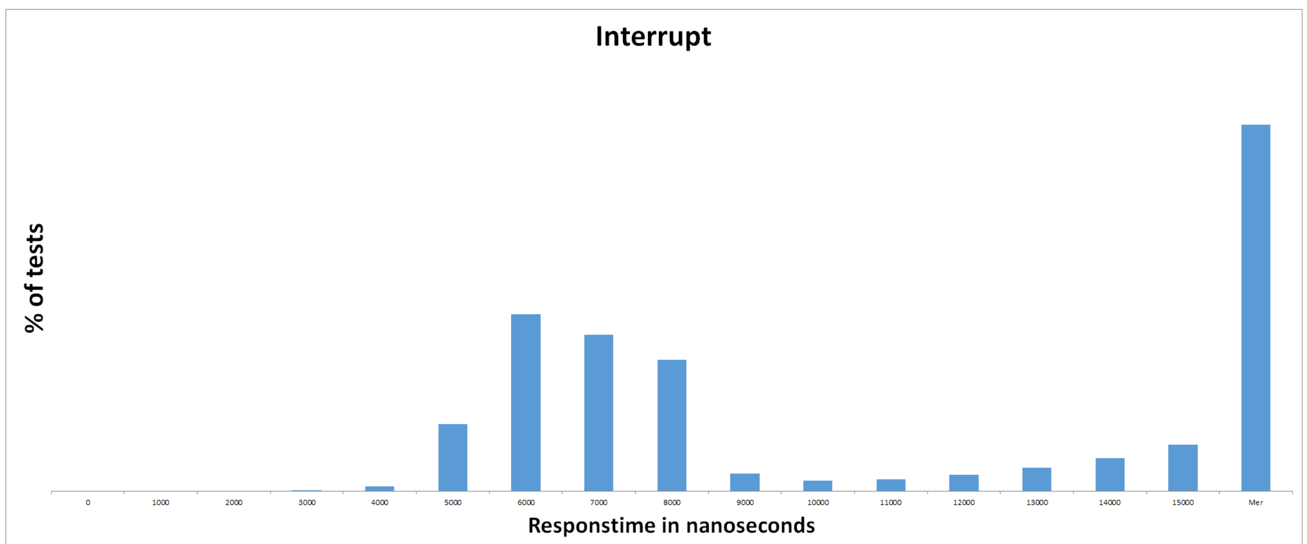

Figure 6.4: Test results with interrupt

6.3 KybOS

In Eirik Wold Solnørs master thesis KybOS [11] he created his own operating system. This gave the opportunity to work together with him and test the response time of his system. The results are shown in 6.5. The results here were kind of surprising to the both of us, as his OS had the code for responding directly as a module in the kernel, and had nothing else to do. This had led me to believe that it would be far better than what could be achieved by Ubuntu. It performed a bit better than running it multithreaded on one core in Ubuntu (the most fare comparison as his operating system only utilised one core). The results are better, as it hits worst-case scenarios way less frequently than Ubuntu and the average response time is lower. Since it is interrupt-driven and in the kernel it also uses far less CPU, making it far more power-efficient.

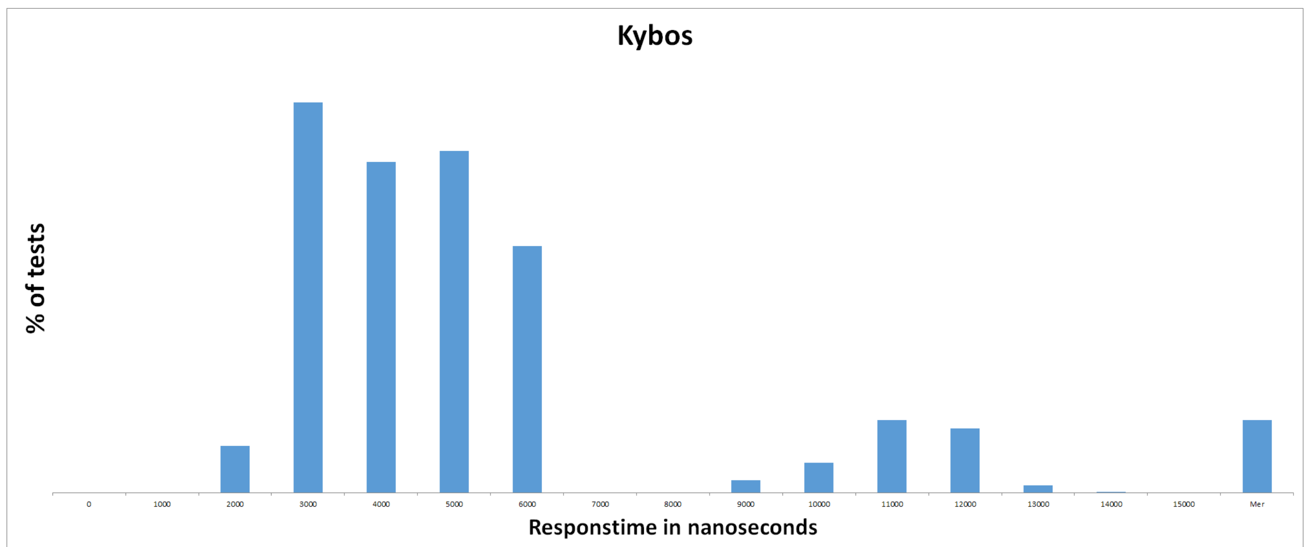

Figure 6.5: Test results on KybOS

Chapter 7

Discussion

7.1 Process

SD card

At the beginning of the project I got hold of the Raspberry Pi 3 and a Micro SD card. Believing it to be of no consequence, I did not go for the most expensive card, and bought a Kingston Micro SD 16GB UHS-I U3. After installing Ubuntu on it and setting up WiFi, installing packages and updating the included packages, it suddenly crashed. The file system locked itself up, entering read only mode. This is done by the operating system to protect your files when it senses that there is something wrong. This read only mode includes simple commands such as ls (list files) and cd (change directory) not being recognised as commands, and such the system was useless. The only things that still works are the files that happened to lay in the cache of the processor. This means that the graphical interface would not crash, but no button would work.

The fix is to unmount the SD card, put it in on another computer running Linux, fix the file system then reinstalling it into the Raspberry Pi. Under 10minutes later, the card would crash again. This time I tried to format the card and reinstall the operating system, but it happened again and again, until there was no doubt about it being the cards fault. I had initially bought two of them as I intended to have multiple operating systems, so I

tried the other one but still the same issues appeared. Searching online yielded that there is a page where user reports on compatibility between SD cards and Raspberry Pi [5]. This page has varying reports from users, with some stating that it is working, while others have not been that lucky. Getting hold of a SD card from SanDisk (same speed and capacity) did the trick, and it has not crashed since.

Ubuntu Mate

Choosing Ubuntu Mate as the operating system was a decision I started regretting after a while. As described in section 3, I had the Raspberry Pi connected to WiFi and interfaced with it over SCP and SSH. This meant that if those services were not running, I had to connect a keyboard and a screen to fix it. Unluckily, Ubuntu Mate does not like hard resets by pulling out the power, and the Raspberry Pi has no power off button. So to turn it off gently, I would have to SSH in and run a command that turns it off. But when the SSH/desktop was not responding (such as when running a lot of high priority tasks), the only way to turn it off was to pull out the power adapter.

By doing this, Ubuntu Mate goes to an emergency mode, as it was not shut down correctly. At startup it gives a message that the file system might be corrupted and keyboard input is needed to decide what to do. Checking the file system requires to unmount it and do it from another computer running Linux, so that was not desirable. It does give the option to bypass the check by entering the command "systemctl default". This again requires a keyboard to be connected. So each time I had to turn it off (without the opportunity to do it via SSH) the following process had to be done:

1. Pull out the power plug
2. Take the keyboard from the computer and plug in to Raspberry Pi
3. Insert power
4. Wait 5-10sec and enter "systemctl default"
5. Take out keyboard and reenter it into the pc

7.2 Results

Response Time

The results as shown in chapter 6 prove that Ubuntu Mate is not suitable to be deployed as a real time system. This is not ground breaking as it is not something it claims to be, but we now have data that shows it. Some of the different methods we can implement the testing has been shown, and we can see clear differences between them. It is also easy to observe a clear trend that with more computing power comes faster responses, as we unlock more cores and give higher priority. The handling of interrupts are a bit weak in these tests and I am not pleased with the way they came out. In my opinion they should not have gone into worst case scenarios as often as they did, but I do think that this is a flaw with wiringPi. On the other hand they show that we can get some OK responses whilst using almost no CPU.

Data logging

In chapter 4 it was shown how the tester could be used for timing pieces of code, and some factors that is needs to be thought about when doing such tasks. The results for the self-test was satisfactory, with a clean and consistent pattern appearing. In hindsight I believe that the result could have been even better had i compiled the C++ code with a more aggressive optimising policy, such as -O3[3]. This functionality of the tester gives it another dimension to just being a response tester. It could be useful in cases where code is running on a remote target with no way of logging timings on the target. There is some overhead in setting all the correct pins (can be improved if it is known that only one tag is written at all times), such that it is not always better to use this method than writing to a timestamp to a local file.

7.3 Future Work

Going further the most interesting thing would be to compile an own version of Linux with real time patch enabled and all unnecessary services removed, as to make the hardware perform as good as it possibly can and compare this to the results in this thesis. For the tester, a couple of ideas have emerged

Transfer speed

The tester uses about 15 seconds to transfer one log file of 512 test lines to the computer. This means that doing a series of tests takes a long time, and it feels unnecessarily slow. In my tests I did a series of 50 tests, which ended up taking 12.5 minutes just to transfer the files. Adding the time it takes to perform the test (1-2sec each) + extra overhead, it takes about 14 minutes to do them.

Continuously testing

Following the previous point, it would be handy to provide a way of continuously testing and showing the results as they come in. This would require changes on the tester as it now does a test, saves it locally, then transfers it to the computer if needed. It can not do it at the same time, which would be needed if this functionality would be implemented.

Reworking the PCB

As the PCB is now, it still has debugging pins and hardware, buttons and LEDs that are not in use. Shaving these off would make it possible to deploy the system on a smaller PCB.

7.4 Concluding remarks

This project has shown how to utilise the tester to perform two different types of tests. One was the data-logging, the other the interrupt to acceptance test. The results for Ubuntu Mate running on a Raspberry Pi 3 were in line with what was expected. Different methods of programing the Raspberry Pi to respond has been shown, and their differences theorised and proven in the results. The result that is not completely as it should have been is the result for interrupts, which in my head should have performed better than it did. The tester works as expected, and has shown that it can do what it set out to do. Overall I consider the project a success, having gotten the results that I was looking for.

Appendix A

Acronyms

ACK Accept signal or message

ASCII American Standard Code for Information Interchange

COM Communication

Config Configuration

CPU Central Processing Unit

DUT Device Under Test

FPGA Field Programmable Gate Array

GUI Graphical User Interface

HAT Hardware Attached on Top

LED Light Emitting Diode

LUFA Lightweight USB Framework for AVR

MCU Micro Controller unit

OS Operating System

PC Personal Computer

RPi Raspberry Pie

RTS Reak Time System

SoC System on Chip

SPI Serial Peripheral Interface

SUT System Under Test

UI User Interface

USB Universal Serial Bus

Appendix B

Code

B.1 For testing the OS

```
#include <iostream>
#include <wiringPi.h> // Include WiringPi library!

#include <vector>
#include <thread>
#include <mutex>
#include <atomic>
#include <bitset>
#include <cmath>
#include <iostream>
using namespace std;
// Pin number declarations. Use WiringPi chip pin numbers.
#define iA 8
#define iB 9
#define iC 7
#define iD 0

#define oA 15
#define oB 16
#define oC 1
```

```
#define oD 4

#define RW 2

#define SIGNAL_SLEEP 1
#define PRIO_DIFF 1
#define THREAD_SLEEP 1 //micros

struct Pin {
  int inn, out;
  Pin(int inn, int out):inn(inn),out(out){
    pinMode(inn, INPUT);
    pinMode(out, OUTPUT);
  }
  void signal() {
    digitalWrite(out, HIGH);
    digitalWrite(out, LOW);
  }
};

class Pins {
  std::vector<Pin> gpio_pins;
  Pin read_write;
  const int data_tab[7] = { 3,5,6,10,11,12,13 };
public:
  Pins(): read_write(RW,RW){
    digitalWrite(RW, LOW);
    pullUpDnControl(RW, PUD_DOWN);

    gpio_pins.push_back(Pin(iA, oA));
    gpio_pins.push_back(Pin(iB, oB));
    gpio_pins.push_back(Pin(iC, oC));
    gpio_pins.push_back(Pin(iD, oD));
    for (auto& pin : gpio_pins) {
      pinMode(pin.inn, INPUT);
```

```
      pinMode(pin.out, OUTPUT);
      pullUpDnControl(pin.inn, PUD_DOWN); //PUD_OFF;PUD_DOWN;PUD_UP
      pullUpDnControl(pin.out, PUD_DOWN);
      digitalWrite(pin.inn, LOW);
      digitalWrite(pin.out, LOW);
    }

  }

  Pin operator[](int n) {

    return gpio_pins[n];
  }
  void writeData(uint8_t word) {
    auto t = std::bitset<7>(word).to_string();
    for (int i = 0; i < 7; i++) {
      digitalWrite(data_tab[i], t[i] == '1' ? HIGH : LOW);

    }
    read_write.signal();
  }
};

//%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%


void bigWhile() {
  Pins system_pins;
  while (1) {
    for (int i = 0; i < 4; ++i) {
      if (digitalRead(system_pins[i].inn)) {
        system_pins[i].signal();
      }
    }
    std::this_thread::sleep_for(std::chrono::microseconds(THREAD_SLEEP));
```

```
  }
}



//%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

void set_thread_priority(std::thread& trad,int prio) {
  sched_param sch;
  int policy=SCHED_RR;
  auto thread_ = trad.native_handle();
  pthread_getschedparam(thread_, &policy, &sch);
  sch.sched_priority = prio;
  pthread_setschedparam(thread_, policy, &sch);
}

void set_cpu(int cpu_core) {
  cpu_set_t mask;
  int status;

  CPU_ZERO(&mask);
  CPU_SET(cpu_core, &mask);
  status = sched_setaffinity(0, sizeof(mask), &mask);
}



static volatile bool running;
static volatile bool barrier;

void test_thread(Pin p) {
  while (barrier) {}
  static int thread_nr = 0;
  //set_cpu(thread_nr); //use this to put each thread on new core
  thread_nr = (thread_nr + 1) % 4;
  while (running) {
    std::this_thread::yield();
```

```
    if (digitalRead(p.inn))
    {
      p.signal();
    }
  }
}

void threaded() {
  Pins pinner;
  std::vector<std::thread> threads;
  for (int i = 0; i <4; ++i)
    threads.push_back(std::thread(test_thread, pinner[i]));
  int prio = 10;
  //for (auto& i : threads)
    //set_thread_priority(i, prio);
  barrier = false;
  char a;
  std::cin >> a; //press any key to exit
  running = false;
  for (int i = 0; i < 4; ++i)
    threads[i].join();
}
//%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
void ansA() {
  digitalWrite(oA, HIGH);
  digitalWrite(oA, LOW);
}
void ansB() {
  digitalWrite(oB, HIGH);
  digitalWrite(oB, LOW);
}
void ansC() {
  digitalWrite(oC, HIGH);
  digitalWrite(oC, LOW);
}
```

```
void ansD() {
  digitalWrite(oD, HIGH);
  digitalWrite(oD, LOW);
}
void inter_driven() {
  Pins pinner;
  wiringPiISR(iA, INT_EDGE_RISING, &ansA);
  wiringPiISR(iB, INT_EDGE_RISING, &ansB);
  wiringPiISR(iC, INT_EDGE_RISING, &ansC);
  wiringPiISR(iD, INT_EDGE_RISING, &ansD);
  while (1) {
    std::this_thread::sleep_for(std::chrono::microseconds(1));
  }
}
//%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%


std::mutex mut;

void tester(int define) {
  Pins pinner;
  //set_cpu(0);
  while (barrier) {};
  while (1) {
    mut.lock();
    pinner.writeData(define);
    mut.unlock();
    std::this_thread::yield();
  }
}
void noise() {
  set_cpu(0);
  int b = 0, c = 0;
  while (barrier) {};
```

```
  while (1) {
    //std::this_thread::yield();

    ++b;
    if (b == 100) {
      cout << "100\n";
      b = 0;
    }

  }
}

void timeSection() {
  Pins p;

  while (1) {
    p.writeData(1);
    std::this_thread::yield();
  }

}

void timeCode() {
  std::vector<std::thread> threads;



  for (int i = 0; i < 4; i++) {
    threads.push_back(std::thread(tester,std::pow(2,i)));
  }
  barrier = false;
  for (int i = 0; i < 4; ++i)
    threads[i].join();

}
```

```
//%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

int main()
{
  // Setup stuff:
  if (wiringPiSetup() < 0)
  {
    fprintf(stderr, "setup␣failed\n");
  }
  running = true;
  barrier = true;
  bigWhile();
  //threaded();
  //inter_driven();
  //timeCode();
  //timeSection();
  return 0;
}
```

B.2 For analyzing .rts files

```
import sys
INT=1
ACK=2
FD=[3,5,9,13,20,63] #seperation points in a line
FL=16      #number of chars in a line
N_PINS=4
class dataLine():
  '''
  Strict-like class for holding one line of test data from the system.
  '''
  def __init__(self,binary):

    self.b=binary
```

```
    self.b=self.b[2:]
    self.type=int(    self.b[FD[0]:FD[1]],2)
    self.inter_status=  self.b[FD[1]:FD[2]]
    self.ack_status=  self.b[FD[2]:FD[3]]
    self.inter_id=int(  self.b[FD[3]:FD[4]],2)
    self.timestamp=int( self.b[FD[4]:FD[5]],2)
  def __str__(self):
    return str(self.type)+" "+str(self.inter_status)+\
    " "+str(self.ack_status)+" "+str(self.timestamp)

class datafile():
  '''
  Class holding all the datalines of the test file.
  It can calculate the response times of the logfile.
  '''
  def __init__(self,n):
    self.lines=[0]*n
    self.n=0
  def addLine(self,line):
    self.lines[self.n]=line
    self.n+=1
  def find_next_ack(self,j,inter):
    for i in range(j,self.n,1):
      if self.lines[i].ack_status[inter]=='1':
        return i
    return -1
  def response_times_intAck(self):
    res={0:[],1:[],2:[],3:[]}
    for l in range(N_PINS):
      prev_int=0
      for i in range(self.n):
        if  (i==0 and self.lines[i].inter_status[l]=='1')   or\
            (self.lines[i].inter_status[l]=='1'  and\
              self.lines[i-1].inter_status[l]=='0'):
```

```
                pos=self.find_next_ack(i,l)
                if pos>=0:
                    resTime=self.lines[pos].timestamp-self.lines[i].timestamp
                    res[l].append(resTime)
        return res


class RTTresults():
    '''
    Main class for holding info about a test.
    Includes the entire test file and methods
    for printing info and basic statistics.
    '''
    def __init__(self,filename):
        self.filename=filename
        self.fil,self.n=self.__openFile(filename)
        self.responses=self.__parse()
    def __parse(self):
        df=datafile(self.n)
        for linje in self.fil:
            binary=bin(linje)
            if int(binary[2])>0:
                dl=dataLine(binary)
                df.addLine(dl)
        return df.response_times_intAck()

    def basicAnalyze(self):
        retDict={"Max":[],"Min":[],"Avg":[]}
        for key,val in self.responses.items():
            maxi=max(val) if len(val) else 0
            mini=min(val) if len(val) else 0
            avg=sum(val)/len(val) if len(val) else 0
            retDict["Max"].append(maxi)
            retDict["Min"].append(mini)
            retDict["Avg"].append(avg)
        return retDict
```

```
  def printInfo(self):
    print self.responses
    print self.basicAnalyze()
  def __openFile(self,filename):
    file =open(filename,"r")

    text=file.read().strip()
    text=text.replace('\n','')
    text=text.replace('\r','')
    n_lines= len(text)/FL #FL=chars in one line
    lines=[0]*n_lines
    j=0
    for i in range(0,len(text),FL):
      lines[j]=int(text[i:i+FL],FL)
      j+=1
    return lines,n_lines

def main():

  if len(sys.argv)==2 and sys.argv[1][-3:]=="rts":
                        data=RTTresults(sys.argv[1])
                        data.printInfo()
        elif(len(sys.argv)==3 and sys.argv[1]=="folder"):
                import glob
                files= glob.glob(sys.argv[2]+"\*.rts")
                tot_data={"Max":[0]*N_PINS,"Min":[0]*N_PINS,"Avg":[0]*N_PINS}
                fileA=open(sys.argv[2]+"\A.txt","w")
                fileB=open(sys.argv[2]+"\B.txt","w")
                fileC=open(sys.argv[2]+"\C.txt","w")
                fileD=open(sys.argv[2]+"\D.txt","w")
                for fil in files:
                        raw=RTTresults(fil)
                        for line in raw.responses[0]:
                                fileA.write(str(line*20*3)+'\n')
                        for line in raw.responses[1]:
```

```
                    fileB.write(str(line*20*3)+'\n')
                for line in raw.responses[2]:
                    fileC.write(str(line*20*3)+'\n')
                for line in raw.responses[3]:
                    fileD.write(str(line*20*3)+'\n')
                data=raw.basicAnalyze()
                for key in data:
                    for pin in range(N_PINS):
                        tot_data[key][pin]+=data[key][pin]
            for key in tot_data:
                for pin in range(N_PINS):
                    tot_data[key][pin]/=len(files)
            print tot_data

  else:
    print "Please input a filename"
if __name__=="__main__":
  main()
```

Bibliography

[1] (2016). Blog post about the amount of sold raspberry pi. https://web.archive.org/web/20160229072534/https://www.raspberrypi.org/blog/raspberry-pi-3-on-sale/. [Online; accessed 25-September-2015].

[2] (2016). Comparrison of raspberry pi models. http://www.makershed.com/pages/raspberry-pi-comparison-chart. [Online; accessed 25-September-2015].

[3] (2016). Gcc optimizing options. https://gcc.gnu.org/onlinedocs/gcc/Optimize-Options.html. [Online; accessed 18-April-2016].

[4] (2016). Raspberry pi operating systems. https://www.raspberrypi.org/downloads/. [Online; accessed 26-February-2016].

[5] (2016). Raspberry pi sd card compatibility. http://elinux.org/RPi_SD_cards. [Online; accessed 6-December-2015].

[6] (2016). Reference of wiringpi. https://projects.drogon.net/raspberry-pi/wiringpi/functions/. [Online; accessed 25-September-2015].

[7] (2016). Winscp homepage. https://winscp.net/eng/index.php. [Online; accessed 25-September-2015].

[8] Burns, A. and Wellings, A. J. (2010). *Real-time systems and programming languages*, volume 2097. Addison-Wesley.

[9] Gonsholt, K. (2014). Implementing a time management unit for the or1200 processor. Master Thesis NTNU.

[10] Jadaan, D. (2015). Fpga based real-time systems tester. Master Thesis NTNU.

[11] Solnør, E. W. (2016). Kybos. Master Thesis NTNU.