University of Nevada, Reno

RealPi - A Real Time Operating System on the Raspberry Pi

A thesis submitted in partial fulfillment of the
requirements for the degree of Master of Science
in Computer Science and Engineering

by

Samuel Francis Delaney

Dr. Dwight Egbert, Thesis Advisor
Dr. Frederick C. Harris, Jr., Co-Thesis Advisor

December, 2018

THE GRADUATE SCHOOL

We recommend that the thesis
prepared under our supervision by

SAMUEL FRANCIS DELANEY

Entitled

RealPi - A Real Time Operating System on the Raspberry Pi

be accepted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE

Dr. Dwight Egbert, Advisor

Dr. Frederick C. Harris, Committee Member

Dr. Xiaoshan Zhu , Graduate School Representative

David W. Zeh, Ph.D., Dean, Graduate School

December, 2018

Abstract

Academia has always sought to ride the line between established thought and new developments. No much more so than in the terms of technology. Universities seek to teach using a known and proven method and resource but also stay relevant with new technologies to provide students the knowledge they will need to be competitive in the work place or graduate field. This thesis will explore how the University of Nevada Reno approaches this problem with its Real Time Operating system course. Namely on how using the established Micro cOS II Real time Operating System with the new builder phenomena the Raspberry Pi we can overcome the challenge of updating a tried and true lesson plan updated to use technology relevant and interesting to the students of today.

Dedication

for Jenn, Cadence, Kerrigan and McKenna

Acknowledgments

I would like to thank the following people who stuck with me over the many years it took to get here.

My wife Jennifer who never let me give up on attaining this goal.

My committee members Dr Dwight Egbert and Dr Frederick Harris who have preserved with me, providing tempered guidance and advice.

Bill Groesbeck for all the laughs and comradeship that only Computer Information Engineering majors can understand.

Thank you to the members of The Nevada Army National Guard for all of the support.

Contents

Abstract i

Dedication ii

Acknowledgments iii

1 Introduction 1

2 Background and Related Work 3
2.1 Operating Systems 3
2.1.1 Kernel 3
2.1.2 Context Switching 4
2.1.3 Scheduler 5
2.1.4 Multitasking 6
2.2 Real-Time Computing 7
2.2.1 Real-Time Operating Systems 9
2.2.2 MicroC/OS-II 11
2.3 Related Work 11

3 Hardware **12**
3.1 Target System: Raspberry Pi 12
3.2 ARM Processor 13
3.2.1 Instruction Set 13
3.2.2 Base Board 15
3.3 Whole package approach 15

4 Implementation **16**
4.1 Tool chain 16
4.1.1 Make files 18
4.1.2 Text Editors 19
4.2 Making the Kernel 19
4.2.1 Compiling with GCC 19
4.2.2 Early Goals 20
4.2.3 CPU Port 21
4.3 Micro cOS-II examples 22

4.4 Running Micro cOS-II on the Pi 24

5 Conclusions **26**
5.1 Future Work 26
5.2 Summary 27

Bibliography **29**

A Raspberry Pi Zero How To **31**
A.1 Introduction and Set Up 31
A.2 Development Environment 34
A.3 Building the Kernel 35
A.4 Running the Kernel 35

B Raspberry Pi 1 How To **40**
B.1 Introduction and Set Up 40
B.2 Development Environment 42
B.3 Building the Kernel 43
B.4 Running the Kernel 44

C Raspberry Pi 2 and 3 How To **50**
C.1 Introduction and Set Up 50
C.2 Development Environment 52
C.3 Building the Kernel 53
C.4 Running the Kernel 54

Chapter 1

Introduction

All Computer Science and Computer Engineering disciples have encountered the “real time” catchphrase at least once in their career. Whether that experience equates to a simple buzz word or a fundamental understanding is beyond the scope of this article but none the less the concept has a claim to key concepts in the computing world.

As technology has advanced and computers become faster and more efficient they allow us more room to error in our programming and still produce acceptable response times. Even the most observant user will be unable to distinguish the difference of execution time measured in a few clock cycles. However, as students continue to develop their programming chops they will encounter problems were devices are stripped to bare essentials to keep costs down but still expect timely program execution. This is the realm of the Real Time Operating System course at UNR.

Students during the semester are introduced to the basic fundamentals of Real Time computing and how a real time operating system operates within those fundamentals. To facilitate the process the Micro cOS II Operating system is used for its scalability, availability and robust functionality. Miciruium also provides text books written by the designer of the Micro cOS II Operating System which are used by the students. Included with the text books are example programs written by Jean Jacques displaying the functionality and features of his Operating System. It is using these examples that UNR students begin to understand the underlying mechanisms that go into a real time operating system.

But as time heals all wounds it also destroys all circuitry. For many years stu-

dents in the Real Time Operating Systems course would use the 8051 micro-controller as the development platform for their course work. No one will dispute the 8051's place in computer development history but it's practicality of use has begun to wane significantly. Students today need a board more in line with the products that todays software development companies are utilizing and the Raspberry Pi was a development platform identified to address this issue. Utilizing the Pi hardware which is more in line with today's architecture but still maintaining the ease of use offered by the 8051 lead to the process of modifying the code operating the 8051 and moving it, also known as "porting", to the Raspberry Pi. This port will enable students to run the same simple real time operating examples provided by Jean Jacques but on a board that will stay pace with them their academic careers.

To best convey this process from idea to working project, this thesis will be broken out as follows: Chapter 2 will cover background concepts and other related work to establish a firm foundation and understanding of the topic. Chapter 3 will cover the hardware utilized and provide a high level view of specific critical components. Chapter 4 will shift focus from hardware to software and the actually porting of the Micro cOS II Operating system utilizing a starting code base provided from GitHub [7], and Chapter 5 will provide a summary of lead into future work.

Chapter 2

Background and Related Work

2.1 Operating Systems

At first glance, when the term Operating System is used it triggers a response of Windows or Mac. Its an option clicked on websites for downloads, or to spark debates on INTERNET forums. To the programmer or computer enthusiast an Operating System is a much more fascinating and complicated creature.

Looking back at the first computers there was no operating system outside of the user. Early computers were of a simple design in regards to their use. A computer required a program and input. In this program the user needed to include how the program was going to interact with the hardware of computer, i.e. how to interpret if a switch had been pulled or how to turn on a bulb. These instructions to interface with the physical computer had to be included in every program. In addition to having to include interface code a user had to wait until all jobs in the queue before him were completed before any sort of inputting could be started[15].

2.1.1 Kernel

As time went on and as new innovations took place the need for an automated system to handle the routine tasks required of all programs led to the development of the Operating System. While still a far cry from the systems in place today, the earliest operating systems served as the go between an application or program and the hardware [16]. With the development of the kernel which contained support programs

that applications could use to interact with displays and input devices. Programs use the kernel to interact with every component of a computer, most important of which would be the CPU. By allowing the computer programmer the ability to focus solely on the program at hand without the burden of also writing compiling and testing code to interface with the hardware proved to be invaluable. By using established working code, development time was significantly reduced and errors within hardware modules became virtually nonexistent. With this bridge crossed the next problem to be tackled was multitasking.

As useful as the support programs or libraries were it was still an inconvenience to only run one program at a time. Without the ability to run multiple programs there can become dead times as jobs that take only 3 minutes to run must wait for a job that runs for 6 hours to complete leading to an incredible amount of work time lost. It was then at this time a scheduler was developed. The scheduler is the component of the operating system that selects what jobs are run and when. Though there are many different methods of scheduling the main concept is that each job is given a certain amount of time to run before it is paused and a new job is allowed to run, this is known as context switching.

2.1.2 Context Switching

Context Switching, simply put, is when the OS places the current job on hold and proceeds to work on another job before switching back [8]. This requires a few key steps to be accomplished without error however. First the current job whether it be thread or process must be saved in its current state in order to be restarted from that exact spot. This requirement for returning to the same execution state is critical, any deviation or skipped steps from the prior execution could lead to corrupt data or an error in execution resulting in a failed job state. To prevent this catastrophe key elements of the current process must be stored such as the Program Counter, memory mappings, etc. It is important to note however that while the overall notion of a context switch is the same between different operating systems the exact procedures

and cost (computational time, recourses saved) may vary.

2.1.3 Scheduler

With the ability to context switch between tasks it is the scheduler that informs the CPU which job to run. It also falls upon the scheduler to decide when to switch between jobs. While a fundamental reaction might be to divide computation time up evenly between the tasks until all jobs are complete. This type of procedure, known as a scheduling algorithm, is the Round Robin Scheduling algorithm. However this type of scheduling is in most cases highly inefficient as it requires heavy overhead to constantly perform a context switch. For this reason the development of other algorithms became a necessity.

Scheduling algorithms

It is important to clarify that there is no best algorithm. Each algorithm presented may or may not be the best approach for a given system. The burden of selecting an algorithm to yield maximum performance lies upon the programmer after analyzing the tasks needed to run, how these tasks are implemented, and how complex a scheduler that can be supported. As we saw in the previous section the algorithm that works on each task for a set time before switching to another task is the Round Robin algorithm [1].

This algorithm performs well as no process must wait on another to complete but as the number of tasks running simultaneousness increases so does the wait time between each process. In contrast to the round robins even distribution of work there is the most basic scheduler algorithm, the First In First Out algorithm. This scheduler mimics the early computer method of working of jobs in the order they are received and only switching once the current task has completed. Though simple this algorithm offers no benefit of prioritizing tasks as high priority tasks must still wait for every task in the queue before it to complete. This also presents a problem of starvation, a task that is unable to complete due to denial to a resource, as if

a task never completes no other task will be granted access to the CPU. Another algorithm used for scheduling is the Earliest Deadline First algorithm. With this form of scheduling whenever a scheduling event occurs, such as a task completes, a new task enters, etc. The scheduler will then scan all the current tasks and switch to the task whose deadline is closest to finishing. Keep in mind the priority of the task at this point is still irrelevant, but this algorithm ensures that the most critical task in regards to time is receiving the most attention from the CPU. There are many other algorithms that can be employed to schedule tasks, each with its pros and cons. The last algorithm we will discuss in this paper is the Fixed priority preemptive scheduler. This algorithm assigns each task a priority ranking from highest to lowest. For each priority there exists a corresponding queue where tasks of that priority will wait for access to the CPU. The rules of the queue are that tasks are taking from the highest queue first and once complete will then service the lower priority queues. However if a low priority task is running and a new higher level task enters a queue. The lower level task must pause and allow the higher level task access to the CPU. This method ensures that the most critical task is allowed access to CPU in contrast to the task with the fastest approaching deadline [12].

2.1.4 Multitasking

We have explored how a system can switch tasks through context switching and how the scheduler can be implemented to perform these context switches these merely make up the bigger picture of multitasking. Multitasking is the ability to run multiple tasks at the same time which was the goal in the development of the scheduler. Though so far all multitasking has been done by the operating system. These types of systems are referred to as Preemptive multitasking systems.

Preemptive Multitasking

In a preemptive multitasking system the scheduler is the key figure in deciding which tasks run, for how long, and if they will be interrupted [10]. Most Operating systems

today use the preemptive multitasking scheme as it provides a higher guarantee that high priority tasks will be given CPU access and providing a seamless flow of multiple tasks as those tasks that may be waiting for resources are not hogging the CPU in an idle state.

Cooperative Multitasking

The other method of multi-tasking in that of Cooperative multitasking. In this scheme the scheduler is demoted to merely a module who starts tasks. All context switching is controlled by the tasks themselves. In order to accomplish this each task that is run in a cooperative schema may only perform a context switch if the current task allows it. For example a task may require keyboard input to proceed so a mechanism is included that allows the task to give up its CPU to another task until it receives the data it needs to proceed [2]. It becomes readily apparent however that this method can be prone to CPU hogging if a task is poorly written or if a task included no mechanism to perform a context switch! As such, cooperative multitasking while still used is reserved for operating systems specifically tasked for this type of operation none of which would be found in a off the shelf computer store.

2.2 Real-Time Computing

All computer scientists and programmers have at some point to deal with the concept of time. When developing new code the best programmers strive to limit the amount of time and resources used by the computer. It is at this time the analysis of algorithms takes place and the discussion of limits and big-O notation arise. Speed can always be increased with the addition or upgrading of hardware. More memory and a faster processor can accelerate the production of output. Many professional gamers and industry professionals who require mass amounts of data to be processed quickly are ever on the look out for the latest hardware. Its apparent that through top tier hardware and efficient code we can process data fast but is it considered real time? The answer is: it depends. This response is what leads to most of the confusion what

comprises a real-time system.

For a system to be classified as real-time it must follow a simple rule: Any task must be completed and correct before its deadline is hit. It seems that if this is all that is required that almost every computer is running in real-time. Everyday users open their email, play video games, and run other applications without issue or users walking away in frustration as they never complete the task but the fact is that there is no guarantee that any of these tasks will complete in a set time constraint. Browsers may take time to load waiting for resources to become available, a system with many other applications may not prioritize the latest task resulting in longer compute times. This lack of a guaranteed result in a set time prevents system from being considered real-time. Perhaps the best example of a real-time system is the pacemaker. One of the methods a pacemaker may use to regulate heart rate is that of Bi-ventricular Pacing to treat Congestive Heart Failure. In a healthy heart the left ventricle will receive an electrical pulse to compress the chamber, and the right ventricle will receive a pulse as well in order to contract in unison. In a patient suffering from Congestive Heart Failure however the right ventricle either receives a delayed or absent pulse preventing the heart from pumping in unison. When the ventricles fail to pump in unison the rate of blood flow becomes compromised leaving the patient at risk of death or disability. With the use of a pacemaker configured to use Bi-ventricular pacing an embedded device will detect the pulse in the left ventricle and then within milliseconds send a pulse to the right ventricle to cause it to contract. In this system there is essentially one task: When a pulse is detected send a pulse. The deadline for this task is configured by the doctors assessing the patient but it will be under second in time. Its readily apparent that this is a real time system as the pacemaker will always send a pulse, completing the task, before the deadline is approached. If it does not the consequences are catastrophic.

While other systems may have fast hard-drives, large amounts of memory, over-clocked processors capable of performing billions of calculations a second we may call them high-performance systems, but the amount of work a system can do is not

relevant to an real-time system. Real-time systems can only be called real-time if it can guarantee that all tasks run on the system will be completed and correct before a deadline has elapsed [3].

2.2.1 Real-Time Operating Systems

With a base understanding of an operating system and the concepts of real-time computing we can discuss what a real-time operating system is and does. A real-time system must complete a task within a set time guideline. A programmer can write code that is constrained to run in so many machine cycles (Obviously this becomes more difficult with more complex programs as big O complexity increases) but how to ensure the Operating System will not schedule another task or more causing our real-time task to fail its deadline? It is for this purpose there are real-time operating systems.

A real-time operating system servers as a facilitator for real-time applications by using an advanced scheduler that ensures the most important task is giving priority in execution time. There are different methods in how this scheduler approaches executions of tasks.

Scheduling algorithms

A few of the most prevalent types of scheduling algorithms found in a RTOS are cooperative scheduling, preemptive scheduling, and earliest deadline first scheduling. Despite the algorithm that is implemented an RTOS requires that this scheduler not run into a situation where it can become starved for jobs or resource hogging. As a result most scheduling algorithms in real time systems are highly advanced and are key components of the system.

Hard vs Soft Time Operating Systems

Depending on the scheduling algorithm and the tasks being performed it becomes apparent that situations may arise where a task is not completed by its deadline. Based on our earlier definition of a real time operating system this would be an

unacceptable situation. However a system may perform close enough to this standard that it is still considered a real time system. We classify this discrepancy by describing the system as a hard or soft real time system [11].

In a hard real time system the deadline is absolute. All tasks that are run on the system must return the correct outcome by the deadline. If a single task is not completed by its deadline the system has failed. We call a system with this ability to always produce the correct output for every input a deterministic system. Of all the real time classifications hard is the most strict and therefore the hardest to implement. Few systems exist that qualify as a hard real time system though those that do usually have simple tasks. The pace maker example from earlier would qualify as a hard real time system as a single missed pulse could cause series consequences for the patient.

In contrast a soft real time system allows for frequent missed deadlines as long as the task can be completed reasonably quickly. The outcome of the task must also still retain some value even after the deadline. An example of this type of system would be a weather system. This weather system may be analyzing multiple components such as humidity, temperature, and wind speed each of which would require a separate task to process. In this system if a task does not meet the deadline but perhaps completes a second afterwards we can still use the data as it still relevant. However the longer the delay from deadline to completion the usefulness of this data will degrade.

While the line that separates a hard real time system from a soft system is absolute there exists a third classification that one could argue IS the line of separation. This is appropriately termed a firm real time system. In a firm real time operating system we are allowed to miss deadlines, though quite infrequently, but if a deadline is missed the value of the task has no value and the task should be discarded. An example of this system would be the playing of audio. If the system misses the occasional sound byte it will likely go unnoticed by the user. However there is no benefit in playing the missed byte at a later time hence its value is zero.

2.2.2 MicroC/OS-II

The main real time operating system that this paper will focus on is the MicroC/OS-II The Real Time Kernel. While it has been replaced commercially by a new version MicroC/OS-III version II still remains a main stay choice. Part of this viability is no doubt due to its free licensing for personal or educational use, but MicroC/OS-II has other key features that appeal to hobbyists and large developers alike. First of these features is the scalability of the OS, by scalability it means that the developer can choose what features to build into the kernel and leave out unused modules keeping the kernel size as small as possible. This is also known as avoiding bloat or space dedicated to programs that are never used. For scheduling MicroC/OS-II uses a preemptive scheduler so all context switching is performed by the OS itself. In addition to being preemptive it is also deterministic ensuring that all tasks will be completed on time and correctly [9].

2.3 Related Work

The concept of running a Real Time Operating System on the Raspberry Pi has been milling around on many INTERNET forums since the Raspberry Pi was released. The project of porting an RTOS to the Raspberry Pi had been discussed though most users found that through virtualization of the RTOS that the performance was within the threshold needed for the hobbyist. Most predominate of these virtualizations was FreeRTOS. FreeRTOS is a real time operating system offered as an open source and free RTOS solution for a multitude of processors. Built as a collaboration effort from leading chip companies and possessing a small foot print, the OS has found wide spread use through out the industry, with a significant presence in the micro-controller market. For this reason alone it makes sense that this would be an entry point for having a RTOS running on the Raspberry Pi and to date is the most complete ported RTOS available for the Raspberry Pi [6].

Chapter 3

Hardware

3.1 Target System: Raspberry Pi

While Real Time Operating systems differ greatly from their bigger more well known counter parts the hardware used is essentially the same. The main component that will be the focus of this project is the CPU. The CPU or Central Processing Unit is the brains of the computer. Though all CPUs essentially perform the same function their construction is unique form manufacturer to manufacturer. It is at this point it becomes essential to port the operating system to this hardware. Porting software is simply the task of writing the interface code that will allow the software to use the hardware. In cases of embedded systems this file is sometimes referred to as the board support package file or BSP file.

While computer enthusiasts have been readily abundant since the first do it yourself computers were released in the early 70s, the initial cost was ever on the rise. It become a considerable investment to build a computer even if one only desired low end components. In response to this absence of a low cost all in one computer the Raspberry Pi was developed. Developed in the United Kingdom by the Raspberry Pi foundation, the main goal of the Pi was to create a cost effective platform to teach children Computer Science. By providing an all in one board which only requires a SD card, monitor, input device to run it was a practical and affordable device. Another appeal of the Pi is that it is almost unbrickable or it is unable to be programmed in such a way that it will never run correctly again. The Pi was built with a key feature in

mind that students and hobbyists and even professional programmers make mistakes in the programs they write, by having no Read Only Memory this prevents the user from flashing firmware that might turn the Pi into an inoperable board. Another appealing factor is the small power draw from the device as it can be powered by a usb port from another computer or from a small 5v charger [5].

3.2 ARM Processor

The Raspberry Pi employs the use of an ARM 11 computer organized chip. Namely the Broadcom BCM2835. When we say "computer organization" we are specifying the way a CPU was designed to handle a instruction set. In the simplest of terms it is the road map of how all the parts that make up a CPU talk to one another to perform the CPU task. Each chip manufacture has an established chip organization which is usually how one can identify the type of CPU. In our case the ARM 11 is named for its manufacturer, ARM Holdings.

3.2.1 Instruction Set

To no great surprise the Instruction Set contains instructions. However it is important to classify these instructions and how they are implemented. When discussing an instruction set in computing we are referring to the instructions that the software will give the hardware to execute.

Complex Instruction Set

The first set of instructions we will discuss are the Complex Instruction Set. These aptly named instructions provide the operator a array of instructions that provide more than one simple task per instruction. For example, a complex instruction may load a value from memory, perform an arithmetic function and then store the resultant value in another location.

The ability to perform multiple steps in one instruction provides obvious advantages. Namely the reduction of instructions needed being the most apparent. An

additional benefit is the standardization of simple tasks that would exist as subroutines in other instruction set formats. Additionally by combining multiple actions into a single instruction program footprints are smaller saving the amount of memory and disk storage needed [13].

Reduced Instruction Set

In contrast to the Complex Instruction Set, the umbrella term Reduced Instruction Set was created. At it's root, the reduced instruction set is architecture in which instructions have very finite and specific tasks. The example provided in the complex instruction set was that of a instruction pulling a value from memory, performing a arithmetic function and then storing the resulting value. In a reduced instruction set the same process would be replicated in three separate instructions: One to pull the value from memory, one to perform a arithmetic function and lastly one to store the value. While at face value this seems like a step backwards as the program must now specify each step of a process there are several benefits of the reduced instruction set [4].

Firstly is the power provided to the architect. By having explicit control over every part of the process the programmer has absolute control over the machine cycles being used to perform functions. In skilled hands a program written in a more efficient way than being forced to use memory cycles not needed from a complex instruction set. Secondly, in smaller scale applications such as microcontrollers, the need for robust instructions in minimized, and by using a subset of reduced instructions producing a lean deployment package for development. Lastly, by removing the load/store memory calls from instructions other than those explicitly designed to access memory provides the opportunity for faster processes and more efficient pipelining.

Ultimately as both schools of thought have advanced it has become unclear which architecture is superior as super computers from all era's have utilized some form of either architecture.

3.2.2 Base Board

While there are many components that constitute the entire aspect of the Raspberry Pi, to address each in detail would be verbose and unnecessary as from the project stand point we will address all components except for the CPU as generic devices. The reason for this is two fold. First, by using generic handlers we are able to increase the portability of the project as new versions of the Raspberry Pi are released. Second, it keeps the code simple allowing students to focus on the real time aspects of the project and not get lost in peripheral devices.

3.3 Whole package approach

If the devil is in the details than why not remove them? It definately would seem that we are ignoring the subtle aspects of the Rasperry Pi but it's just a different approach to the same problem. This approach allows to look to the Pi as more of a integrated circuit rather than a complex controller. We are simply concerned about power, ground, inputs and outputs. With the drivers handled and such a low level they are essentially transparent pass through objects that we need not concern ourself and the students with.

Chapter 4

Implementation

4.1 Tool chain

Before we can delve into the workings of the Raspberry Pi and Micro/UCOS-II we need to be able to convert our source files into a language that the Raspberry Pi can use. This is accomplished through the use of a tool chain. A tool chain, like it's namesake, is a group of tools, in our case programs, that when linked or grouped together create a software program. One of the most prevalent examples tool chain use is the Integrated Development Environment or IDE. Most IDEs like Microsoft's Visual Studio provide the ability to edit code, compile and link objects. The more advanced of these toolchains build "hooks" into the developers programs that allow on the fly debugging. It is worth to note however that not all IDEs provide all these features and may only provide subsets of the previous mentioned capabilities.

Many developers use IDEs for their powerful abilities, convenience and features. To all of our dismay, IDE's are not usually bundled with computer systems and need to be acquired and installed by the end user. Most of the advanced IDE's on the market today are not free and can be cost prohibitive per license. IDE's also require substantial system requirements to support the capabilities provided and can add additional size or "bloat" to the executables they create.

As the Raspberry Pi is a minimalist computer which is marketed as a cost effective tinker/development platform, it makes sense to use a minimalist approach to develop our software. As such for the toolkit used in this project we will be using readily

available freeware. Just because we are going free does not mean we have to sacrifice usability and convenience. Earlier I mentioned that a toolchain needs to build, link and compile our executable and while we could scour the INTERNET finding the best tools for each step we would then run into another issue: Interconnectivity. It should surprise no one that not all tools operate the exact same way and when we are building our own toolchain it can become a task of itself to just get the parts in the toolchain to work together. Another fun fact for trivia night is that not all tools produce the correct files for the CPU architecture in use so we also have to make sure the tools we are assembling even work for our board! Worry not dear reader as for this project we are going to skip that and use a free toolchain that does all the steps and works on the processor we need it to. That toolchain is called Yagarto.

The Yagarto Toolchain was developed in 2006 to allow hobbyists and other individuals a free way to compile and deploy ARM based programs. Until Yagarto was developed there were very few Windows applications that could compile ARM based programs without either being very expensive or so difficult to work with that development was hindered. To address this issue Yagarto was created and considered feature compete five years later. Since it's development new toolchains and IDE's have been developed to deploy ARM programs however the simplicity of use and proven track record keep Yagarto a stable of ARM hobbyists [17].

Now that I have sold you on the Yagarto as our front runner toolchain it is now time to disclose an inconvenient truth. While the Yagarto toolchain is monetarily free we will pay in the lack of a very specific feature: Integrated Development. For those who have only written code in Visual Studio you may need to brace yourselves for in this project we need to make the use of "make" files and a text editor to build our project. But "Wait!" you say, "I thought Yagarto did all of the work for us!" And it does, just the work that we can't see. You and I still need to write the code that Yagarto will turn into an actual executable file and to do that we need a program that allows us to type and a way to tell Yagarto what files we want built and how to build them.

4.1.1 Make files

Imagine you are a carpenter who owns a wood shop and makes custom built to order tables. What would you need to make these tables? First obviously you need tools; saws, rulers, sanders, etc. You also probably should know how to use these tools or else you wouldn't be a very good carpenter. We would also need wood but I'll get to that later, but for now we have a carpenter, his tools and the ability to use them in order to make things. This should sound very similar to our Toolchain as basically they are one and the same. Tools that allow us to make things. Now about the wood, can't make tables with out it, but since they are really a resource used to make tables we don't include them in our "toolchain". In our project the wood is our C++ and Header files. Not part of the toolchain, but obviously necessary to project completion. With our tools(toolchain) we can now turn wood(C++/h files) into tables(executables)! Among the more astute reading this, you may have noticed the assumption being made and at this point we are unable to make tables, we have no instructions on how to build the tables! For our build to order table business to do anything we need input from the user on how the table should be built, not in the aspect of how to put nails in boards, but in the sense of should it be rectangle? or a circle? Should it sit 4 or 12? By specifying this in our build order we provide the necessary information to allow the Carpenter to turn wood into the table we want and not just random planks of board glued together. This is no different in our toolchain, without the base instructions, or what we will call a make file, our toolchain has no ability to build anything. This is a very simplified version of the capabilities of makefiles but for this project you would not be wrong to say that the make file is simply telling our toolchain what files to use as resources, in what way we want them built, how we want them linked, and finally how to package them into our final executable.

4.1.2 Text Editors

Without the ability to directly edit our code in Yagarto, we will need to utilize a text editor to modify the C++/h files needed to build our program. For this program Notepad++ was used, however this I will leave up to the reader and personal preference. My only word of warning is that you ensure you are using a text editor that does not insert "invisible" characters into your code. Wordpad and Microsoft Word and prime examples of text editors you DO NOT want to use unless you really know what you are doing. By using editors such as these you will spend countless hours wondering why your compiler is reporting "illegal character" errors and "Unexpected End Of Line" warnings. So while I suggest you stick with what you know, if you have no experience then look into Notepad++, Vim, or just plain Notepad.

4.2 Making the Kernel

As discussed earlier in this paper, for anything to work in any meaningful fashion we are going to need a Kernel. And not just any kernel will do, we need a kernel that runs an ARM architecture and the Yagarto toolchain provides the ability to do this with their arm-eabi-gcc compiler.

4.2.1 Compiling with GCC

The Yagarto toolchain uses the GCC compiler, basically allowing us to use C++, to build and link objects which are then packaged into a Kernel image based on the ARM architecture specified in our make file. While Yagarto will provide us that ARM kernel shell, without the specific operating system code it is not enough to run the processor much less a full board. To actually allow the target board, in our case the Pi, to run we need to provide the operating system itself.

Micro c/OS-II, while aged, was still written with the concept that operating systems should not have specific hardware dependencies. As such we are able to use the source code as is from the developer Micurium. This is a huge boon as any

changes made to the Micro c/OS-II code would prompt significant testing changes to ensure we didn't accidentally alter functionality. However, we are simply not able to simply drop in the code and hit run. There are two other key components that we need to address to bridge the cap from our high level source code to the low level assembly that drives the hardware. Specifically the board support package or bsp and the port of the OS CPU file. It is these two parts that will allow our software the knowledge it needs about the hardware to fully utilize the device. However, we need to eat this elephant one bite at a time and the board support package and cpu port will be discussed once we are further along in the process.

4.2.2 Early Goals

Before bumping into the unknown blindly I set out to see if any Pi Enthusiasts or those in Academia had dabbled with the MicroC/OS-II system and could provide any ground work to build off of. During this research I found two projects that would benefit me greatly. The first was a bare metal programming of the Raspberry Pi to run simple tasks written by a developer named Dave Welch[14]. The term bare metal has many meanings to many people so I should clarify that in this context it is the most simple programing of a device we can do; very reminiscent of the old paper programs where we need to tell the device how to everything from storing data to turning LEDs on and off. Mr Welch's simple programs however were critical in my development as it provided the memory map that would allow me to access the hardware components on the Pi. Remember the Board Support Package I mentioned earlier? The memory addresses provided by Mr Welch were essential in building the BSP as they provided all the memory address and simple calls that could be used to access the hardware on the Raspberry Pi. The second project was found on GitHub developed by Weng Kai which allowed a basic port of the Micro C/OS-II operating system to the original Raspberry Pi[7]. With his port I was able to see how he structured the build in order to utilize the Yagarto toolchain in a user friendly manner and also create a build directory that wouldn't confuse entry level programmers. It also provided me with a

safe starting point for my build. This is critical as experience in industry has taught me that when one is working over many files in many locations it is quite easy to lose hours of work because you don't know what change may have "broke" your code. By having a working model and then building in slow steps from it you can allows quickly revert to a working program and analyze why your change is causing problems.

The first step to build my working prototype was simply to ensure an SD Card with the Micro C/OS-II image would boot and begin execution of Mr Kai's simple program. This step could best be described as merely merging two projects into one, as the memory map provided from Dave Welch's Bare Metal programming as was able to create the start up sequence that would boot the Pi and then simply point to the task switching program written by Mr Kai. As all things go, this turned out to be slightly more difficult than expected. The first issue to arise was that the code provided by Mr Kai, wasn't complete. The main working parts were there but upon the first attempt to build the amount of declaration errors was significant. This of course required the tedious task of ensuring all C++ and header files were linking appropriately and also modernizing some of the data types used through out the program. All understandable as this original code was written in 2005. With this complete we can move to the next step: addressing the issue of creating a CPU port.

4.2.3 CPU Port

The CPU port itself is relatively simple in relation to our goals as we are mainly concerned with two things, the Stack and the Registers. These two components are essentially the life blood of the project and are needed to perform any form of context switching. While a true port of a CPU would be a massive en-devour encompassing all aspects of the CPU our port only requires from us the where and how to do the context switching. This all takes place over two files. One file is a C++ file that is simply providing the memory location for the Stack. The memory locations used are the same ones provided by all ARM ports used by Micerium. The other file is an assembler source file and requires us to actually think as it is a hair more difficult.

We now must tell the CPU how to to perform the context switching tasks using the stack we specified in our C file. We will take solace in the fact that we really only need to handle five aspects: 1) Storing the current working registers 2) Save our state 3) Handle an interrupt 4) Load a new state 5) Load new working registers

Utilizing other ports provided by Micerium we can see that each port essentially performs the same functions in context switching, it's just a matter of syntax. It's essentially just a matter of ensuring your compiler understands the assembly language you are using. For example, if a function's intent it to put this value in register 2, we simply need to write the corresponding assembly for our compiler to take a value and place it in register 2.

With what essentially amounted to code house keeping complete I was able to build a kernel that would boot from the microSD card and start Micro c/OS II with two tasks. These tasks would simply write "Task 1" and "Task 2" back and forth as each interrupt was triggered but again the goal here was to have something that worked and I could go back to when it stopped working. Now my goal was to move from two simple tasks interrupting one another to examples provided to the UNR students.

4.3 Micro cOS-II examples

The textbook used by the University of Nevada Reno's real time operating class is Micro cOS II, the Real Time Kernel by Jean Lebrasse. In this book are five examples that display the more predominant features of the Micro cOS II operating system. The University of Nevada Reno students will compile and run these samples with minor modifications to understand the basics of the OS. However as previously stated in this paper, the programs were meant to be compiled with a Borland 4.5 compiler and run on a system executing the DOS operating system, both of which are hard to come by in 2018.

To port these examples the following objectives were identified: Remove DOS entry and return protocols, remove and replace Borland specific functionality with

gcc functionality and remove screen video display with serial port communications.

As I'm a big supporter of one stone bird removal methodology, the removal of DOS from the programs is trivial as in the previous steps we created a bootable image which loads uCOS II directly. We simply have no need for the DOS system and none of the calls used in the examples affect the real time system. The calls are simply to preserve the state of the DOS stack and relinquish control of the CPU so we are free to simply remove these lines of code out right.

Our stone is currently two for two and in our next step of removing Borland dependencies it's going to put up a strong argument for going three for three. Aside from one one function call, every Borland command we encounter is used to write our Graphical User Interface to the DOS shell. As an interesting note, the way this was accomplished was done by creating an artificial frame buffer that was then loaded to the terminal window like a GPU would load to a display. Since we are not using DOS there is no need to write to a DOS window, so these Borland calls aren't needed and can simply be removed. At this point of the operation we can compile the code and set it running. The only downside is that we removed all graphical attributes of the program so we could only go by faith that are program is executing correctly. As any undergraduate can attest, faith based programming does not credit bestow so we must implement our last identified objective of moving all output to the serial port.

The challenge of this step is to switch from writing to the display buffer used by Borland and writing the data to the serial port in the same matter. Why are we simply unable to to simply reuse this code? We do have access to the video memory after all since our bare bones implementation provides us direct access. The first reason is that I was instructed to use the serial port, but the more practical reason is that the Borland frame buffer functions much differently than a true frame buffer. In our Pi and most bare bones applications we have access to the physical frame buffer but at this level it is individual pixel access. In the Borland DOS window a character map was utilized providing the needed fonts and line positioning. This slight difference may sound trivial but lets look at why the difference is actually

staggering. Let's say we want to write the letter 'A' to the screen on the fifth line, four spaces indented. In the Borland system we simply call the write character function with three arguments provided: The character to write, the 'x' position and the 'y' position, so DispChar(A,5,4), and we are done! In the bare bones implementation we have to write to each individual pixel, so we have two options. We could calculate the position of each pixel in the letter in the x/y plane, and then write a color value to each pixel. But we must then also keep track of every spot and spacing. Or we could precalculate all this information through building a font type that we could call upon but this would also require a controller to maintain all the running data of each character. Ultimately though both approaches are resource intensive, time consuming and outside of the scope of this paper. By utilizing the serial port we can simply pass strings to the host terminal and let the terminal window handle all the graphical work. Another benefit is all of the data we need to display is already in a string format so the only work required from us is to transmit this data using the standard serial communication protocol.

Despite all of this tinkering in the code the key gaol is still met. The original uCOS-II calls remain unchanged and the RTOS functionality is unaffected leaving us with a clean port of the operating system. This was critical to the project as it provides the students with the same code referenced in the Micro cOS II text book and provided source code.

4.4 Running Micro cOS-II on the Pi

With the examples modified to remove all DOS references, Borland dependencies and information being passed through the serial port we are now able to move the software into our kernel image. The easiest way to create our bootable SD card with our kernel is to inject our kernel into a prebuilt bootable SD card. Confused? Don't worry it's super simple. In our project we will keep things easy and use a Raspberry Pi bootable SD Card. The Raspberry Pi usually comes with a preloaded SD Card with the Raspian OS loaded. However we will assume that we need to build one. To

do this we need to download the Raspbian OS from the Raspberrypi Organization website[5]. With OS in hand we can utilize a SD writing tool to burn the Raspbian OS to the SD card. As always one should test this SD card to ensure a successful imaging of the SD card. Once verified it is now possible to insert our kernel image. After we make it of course.

To build the kernel we need to run our make file. As there are millions of ways to create a makefile we won't attempt to cover any of them in this paper and the make file provided will perform all of the necessary functions to take the code and package it into a kernel file. This kernel file requires no other modification and is ready to drop into SD card. This operation requires nothing other than opening our SD card on our development PC and replacing the kernel on the SD card with our new kernel.

With the kernel in place we now have a complete OS and as a sense of poetic irony the fanfare of running is as subtle as the OS itself. As the Pi has no native display we will rely on a host PC to run the Putty Serial application so we can see our tasks executing. Using a serial to GPIO cable attached to both the Pi and our host PC we need only to apply power to the Pi and let the program run. By observing the Putty terminal we will shortly see the application begin to populate the terminal window with a graphical output and run the loaded project.

As stated this is just the starting point for the UNR students. This project is only providing the code needed to execute the programs as provided from the textbook. From here students will be tasked with reverse engineering the code in order to understand its capabilities and design changes in order to affect the tasks priorities and outputs. However, those who plan ahead and keep a working copy on hand as they implement their changes will as always have an easier go of it.

Chapter 5

Conclusions

5.1 Future Work

During the development of this project the Raspberry Pi foundation certainly did not rest on its laurels. With the growth of the home brew programming niche and with increasing demand for better boards new versions of the Raspberry Pi were released every few months. Foremost of these was the Pi Zero, a compact and cheap reproduction of the original Pi but with built in Wifi. With no significant changes to the CPU no changes were required to run the current port of the Micro cOS II RTOS. For cost purposes and sheer novelty of the power to size ratio, the Pi Zero will most likely be the candidate for UNR students to utilize during the Real Time Operating Systems Course.

While the new Raspberry Pi Zero provides a cost efficient board with full capabilities to meet the requirements for the Real Time Operating Course at UNR, there is still work that could be accomplished. The Pi 3 provides a 64 bit processor and even more ram than the Pi or Pi 2. However, of interesting note is that the Pi 3 does not natively run in a 64 bit environment even though it utilizes a 64 bit processor. Work has already begun amongst tinkerers in the community to fix this issue and I do believe that it would prove a challenge for another graduate student to see what performance enhancements could be gained by running uCOS-II natively in a 64 bit environment.

Another limitation found during the development of this project was the direct

writing to the frame buffer pixels. With the development of “hats” - small boards that attach directly to the pi, usually with LCD displays - it would be a great enhancement to develop a graphics library and BSP to replace the serial communications and allow the Pi to operate completely independently. In addition to the video port, many of the other internal hardware components were not added to the bsp as they were not needed. Future projects could implement these devices and allow the uCOS-II tasks to operate them and build real applications for academia or industry. The functionality of the Raspberry Pi was barely tapped in this project, and as a working base the sky is truly the limit for future projects to build upon. .

5.2 Summary

This thesis provided a look into how the fine line between academia and practical application can become more of a haze than a defined line. By exposing students to new hardware but teaching thoroughly tested and reviewed software methodology we can both educate and motivate hand in hand.

We started by assembling all the components necessary to build our program. Preliminary research assisted us with the establishment of a simple working prototype. From there we assembled the tools needed to build a bootable kernel that could be used in the Raspberry Pi. Key to all of this was keeping University students and their needs in mind. Aside from the board itself nothing in this project had to be purchased. Starting with simple freeware programs we were able to construct a toolchain that would build a working kernel. Under the education license we could access all the components of the uCOS-II software for use. Finally with a few modifications this build could be converted to work in the Linux operating environment.

With the tools assembled we then focused on modifying our code to work with the Raspberry Pi hardware and CPU. Utilizing other ports as a guide, the build of the stack was complete and our program had access to the hardware components needed to run a simple program. From a working simple program we then added the working examples provided from Micrium.

Finally, with ported software we were able to assemble a bootable SD card that would run a Raspberry Pi that would provide a host PC running a Putty terminal our graphical interface that mirrored the DOS version identically. With the port complete the hope is to allow University students to build their knowledge and understanding of Real Time Operating systems on a system that will grow with them as they progress through their undergraduate careers.

Bibliography

[1] Remzi Arpaci-Dusseau and Andrea Arpaci-Dusseau. *Operating Systems: Three Easy Pieces*. Arpaci-Dusseau Books, 2014.

[2] Joe Bartel. Non-preemptive multitasking. *The Computer Journal*, (30):37–39, 2011.

[3] M. Ben-Ari. *Principles of Concurrent and Distributed Programming*. Prentice Hall, first edition, 1990.

[4] Margarita Esponda and Rojas Ra'ul. The risc concept - a survey of implementations. *Institut fuer Informatik*, Technical Report B-91-12, 1991.

[5] Raspberry Pi Foundation. Raspberry pi, Last accessed Mar 2017. `https://www.raspberrypi.org/`.

[6] FreeRTOS. Freertos, Last accessed Mar 2017. `https://www.freertos.org/`.

[7] Weng Kai. Github, 2013. `https://github.com/fmlab/ucos_RaspberryPi`.

[8] Linfo. Context switch definition, September 2013. `http://www.linfo.org/context_switch.html`.

[9] Micrium. Micrium embedded software, Last accessed Mar 2017. `https://www.micrium.com/rtos/kernels/`.

[10] riscos.info. Preemptive multitasking, November 2009. `http://www.riscos.info/index.php/Preemptive_multitasking`.

[11] K.G Shin. Real-time computing: a new discipline of computer science and engineering. *Proceedings of the IEEE*, 82(1):6–24, 1994.

[12] William Stalling. *Operating Systems Internals and Design Principles*. Prentice Hall, forth edition, 2004.

[13] Jon Stokes. Risc vs cisc: the post-risc era. *Arstechnica*, 2004.

[14] David Welch. Github, 2016. `https://github.com/dwelch67/raspberrypi`.

[15] Richard Wexelblat. *History of Programming Languages*. ACM Monograph Series. Academic Press, first edition, 1981.

[16] W. Wulf, E. Cohen, W. Corwin, A. Jones, R. Levin, C. Pierson, and J. Pollack. Hydra: the kernel of a multiprocessor operating system. *Communications of the ACM*, 17(6):337–345, 1974.

[17] Yagarto. Yagarto, Last accessed Mar 2017. `http://www.yagarto.org/`.

Appendix A

Raspberry Pi Zero How To

A.1 Introduction and Set Up

At this point of the paper we will skip theory and process in favor of implementing Micro cOS-II on the Pi Zero. In the appendix that follows all steps will be laid out in order to allow a student with the required materials to build their own development station for the Raspberry Pi Zero.

The most critical part of this process is ensuring that all the required materials and software are accounted for before we even begin to load any images or compile any code. For this project you will need the following physical devices:

- Host PC (Desktop or Laptop) with USB ports
- Raspberry Pi Zero W with breadboard and pins
- Adafruit USB to Serial adapter
- MicroSD Card

With the hardware acquired we can now download the required software tools we will need to build our kernel. This list is not absolute as all developers may choose different tools to perform the same task, but as of this writing the tools below are readily available, free, and will work with this project.

- UnetBootin

Figure A.1: Required Hardware sans PC

- Yagarto Toolchain
- Notepad++
- Putty

To set up our development environment first we will start with the Pi Zero and then move to the PC. The Pi Zero may have shipped with a preloaded MicroSD card with Raspbian already loaded. If so great! Step one done. However if you did not receive a preloaded MicroSD Card please do the following:

- Download UnetBootin from SourceForge and save it to your Host PC
- Download Raspbian from https://www.raspberrypi.org/downloads/raspbian/
- Open UnetBootin and select the Disk Image radio button.
- Insert the MicroSD card into the Host PC
- In image, select the unzipped Rasbian image file

- For Type select USB drive and select the Micro SD Card drive letter
- Leave all other options alone and press OK

The process will take some time so be patient as the SD card is formatted and loaded with the Raspbian image. It's essential that this process complete without errors as it is building the bootable image we need to run our kernel.

With the Pi handled we now turn our attention to the Host PC. Our first order of business will be to download and install our text editing software. As stated before any text editor will do as long as you the operator are comfortable with how it handles cpp and header files. For this project NotePad++ was used as a easy to find free text editor that has wide spread use across the world.

Our next piece of software used to be preinstalled on Windows machines so there is a chance that the host PC might already have the application PuTTY installed. If not, the folks at https://www.putty.org/ still maintain a vigil for the SSH terminal loved by all, overlooked by many. It is a stand alone application so after downloading the executable to our desktop for easy access we are done.

Lastly we need to install our toolchain software. There are two methods of doing this. First we could download all the source and build the compiler ourselves, or we can trust the installer from SourceForge. There are certainly pros and cons to each approach but as the first method requires a significant knowledge of package building and compiler options specific to your PC the risk of error is phenomenal. So for this project download the installer executable from SourceForge here: https://sourceforge.net/projects/yagarto/ and give it a good once over with the antivirus before installing.

The other perk of this installer is that is also providing us all the gcc libraries required to compile on ALL ARM architecture, not a specific chip as seen with other tool chains. Once the installer has finished there is an additional step we can utilize to simplify our life. All computers have a path variable which stores "shortcuts" to executable files allowing the user to access that executable even if they are not working

in that executable's directory. Task Manager is an example of this. In our make files when we build the kernel we will be performing multiple calls to the toolchain. In order to save ourselves some pain we will add the path of the bin folder located in the Yagarto2012 directory. (The one you just installed)

A.2 Development Environment

With all our tools in place we can know build our testbed or development environment. From this project you will be provided the a directory with the following directory structure:

The project can be found here: https://github.com/samj3sus/Real-Pi

- BSP (Board Support Package)
- Build (Holds the object files once created)
- H (Header files reside here)
- Init (Initialization files are stored here, such as startup.s)
- Lib (Library files reside here)
- Port (Holds the ported CPU files)
- usrApp (Holds the example programs you will modify)
- uCos (Empty Folder)

No there is no typo on the last item. Due to distribution rites I am unable to provide you a copy of the uCOS-II Software. However, as required the textbook required by the course is the official Micro cOS-II textbook, it contains a CD with the uCOS-II files needed to populate the folder in our project. Simply copy the files directly into the uCos folder in the project directory (Do not add any subdirectories) and our development environment is complete.

A.3 Building the Kernel

With the makefile provided there is no need to make any modifications unless a different compiler was installed or the toolchain directory was not added to the PATH variable. With all MAKE files ours has a clean function and it's always a good practice to start with a clean run. To clean and build our kernel open a terminal window and navigate the project base directory. Once there we will type the following commands.

- make clean (Let this run to completion, should be quick)
- make (Will run considerably longer)

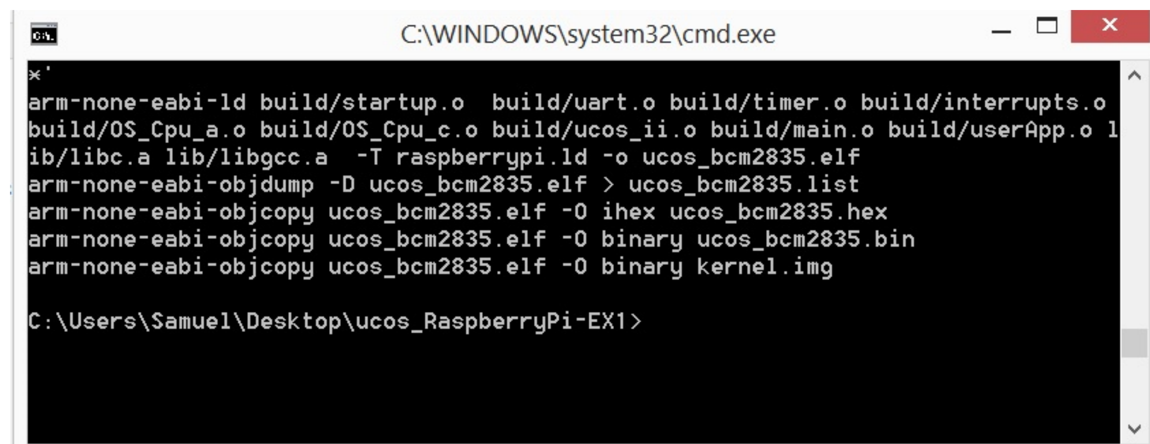

Figure A.2: A successful build output

If we navigate back to our base directory we will now see a new file called kernel.img and no awards for guessing that this is our new kernel file for the SD card.

A.4 Running the Kernel

To run our project on the Pi we need to open the SD card loaded with Raspbian that we created earlier and see it's base directory. In the directory we can see there are two kernel files, kernel and kernel7. We need only replace the file kernel with our new kernel.img as the kernel7 applies to the Raspberry Pi 2 which is very much a horse of a different color. With the file replaced we can eject the SD Card and load it into

the Raspberry Pi Zero. If we simply turned on the Pi now, it would run! However we want to see what it is doing so we turn back to the Host PC to run a shell for the Pi to output to.

On our host PC we need to run the PuTTY SSH application. Upon loading we will get a configuration screen that is asking for a few pieces of information before we can run.

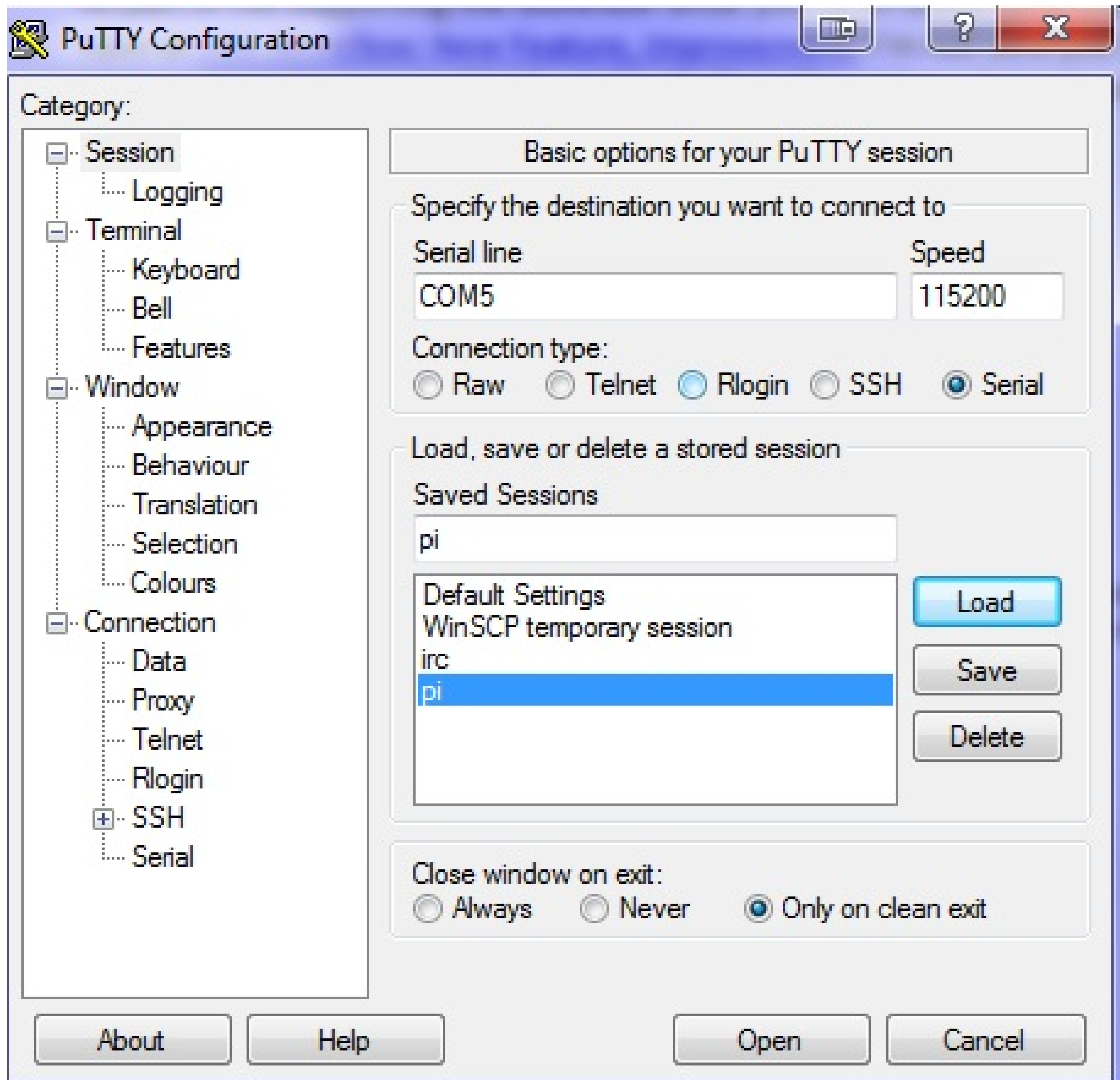

Figure A.3: Sample Putty Configuration

- Speed - Set to 115200 Baud (It works don't ask questions)

- Connection Type - Serial (We need the ANSI escape sequences)
- (What are the escape sequences? Ask Dr Egbert)
- Serial Line - Set this to the COM your Pi is using (Check Device Manager)

Once you have loaded your information click ‘Open’ and a terminal window will appear.

With PuTTY running we can now start our Pi and behold the glory of tasks! Following the pinout of our Pi we will connect our USB Serial cable to our PC. Keep flipping the cable USB connector 180 degrees until it is firmly secured in the host PC. Now connect the pins to the Pi Zero as seen in Figure A.4. Once the power pins have been connected the PuTTY terminal screen will now start running the uCOS example loaded in the usrApp files. If all the steps were completed correctly the image in Figure A.5 should appear on your PuTTY terminal. With this you now have a complete working environment in which to modify the usrApp code and develop your own uCOS-II applications!

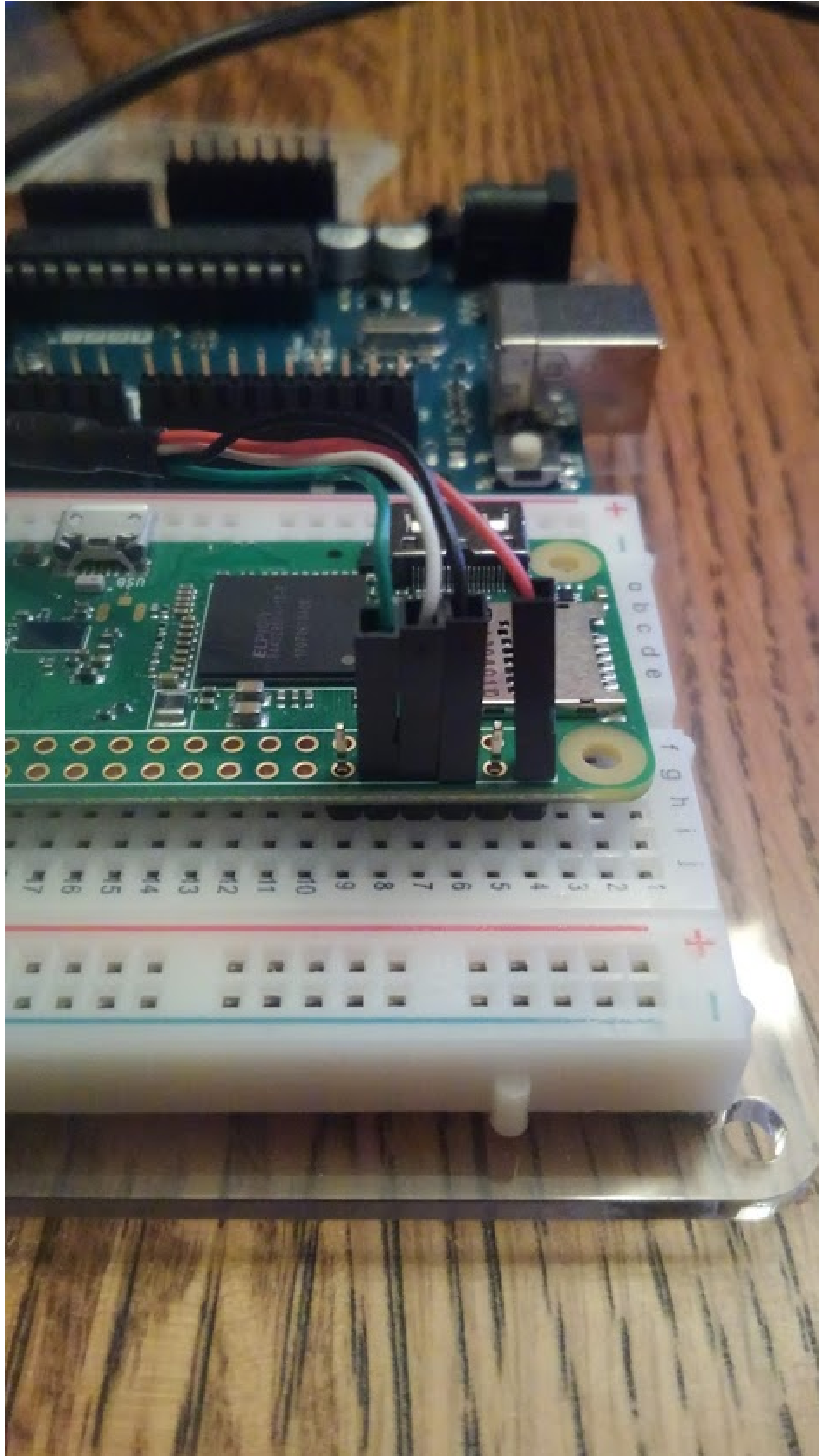

Figure A.4: Pin connections

```
COM4 - PuTTY
                         uC/OS-II, The Real-Time Kernel
                               Jean J. Labrosse

                                  EXAMPLE #1

0672046112363849592183752945 51806490810499610899039201943388667363183471280465 41
7732477364609558114399597293405729088948008904854787531354769851069090387786887 0
5305787310617664387805221186297262438948745344064749395244096114185064831495134 8
2254348476071284354785360859910759618741480341525293981411932063836020802762390 7
22733873829955001 347142225670051136110975237913919885747228710346087513470807 93
8611207497531916918109464315292485373936404431146673686882470049069684353569481 8
6615359097295415749509661935037085781505813969202607272545627139105521924905529 9
2934409806485440347945968571460482010708138231053386627291083436656197866189164 9
7812569628947189643956666352869071234226518110044489306770037735313906454564124 7
626204423330782835854919511490669653 919770318353915886332422851140552175978113
1109988364227125837057713016357748443995012093989224658219631650365039942054938 9
41986974195824621878295003903671325283511478828345517614214987840312490758660 56
17043989953277906687996732127798 33026753301341377712867178087013449463949230452
04515980358227284264965226677758025913857608326807992762470377218790836650761 21
2857395745909769396587145753518843702737111974231581076910816805689712051192003 0
1061556436195005066216736815264893893616188249352785575730876633659217778948455 8

#Tasks           : 13      CPU Usage: 19  %
#Task switch/sec: 1
                            <-PRESS 'ESC' TO QUIT->
```

Figure A.5: Running Example

Appendix B

Raspberry Pi 1 How To

B.1 Introduction and Set Up

In the appendix that follows all steps will be laid out in order to allow a student with the required materials to build their own development station for the Raspberry Pi 1.

The most critical part of this process is ensuring that all the required materials and software are accounted for before we even begin to load any images or compile any code. For this project you will need the following physical devices:

- Host PC (Desktop or Laptop) with USB ports
- Raspberry Pi
- Adafruit USB to Serial adapter
- MicroSD Card

With the hardware acquired we can now download the required software tools we will need to build our kernel. This list is not absolute as all developers may choose different tools to perform the same task, but as of this writing the tools below are readily available, free, and will work with this project.

- UnetBootin
- Yagarto Toolchain

- Notepad++
- Putty

To set up our development environment first we will start with the Pi and then move to the PC. The Pi may have shipped with a preloaded SD card with Raspbian already loaded. If so great! Step one done. However if you did not receive a preloaded SD Card please do the following:

- Download UnetBootin from SourceForge and save it to your Host PC
- Download Raspbian from https://www.raspberrypi.org/downloads/raspbian/
- Open UnetBootin and select the Disk Image radio button.
- Insert the SD card into the Host PC
- In image, select the unzipped Rasbian image file
- For Type select USB drive and select the Micro SD Card drive letter
- Leave all other options alone and press OK

The process will take some time so be patient as the SD card is formatted and loaded with the Raspbian image. It's essential that this process complete without errors as it is building the bootable image we need to run our kernel.

With the Pi handled we now turn our attention to the Host PC. Our first order of business will be to download and install our text editing software. As stated before any text editor will do as long as you the operator are comfortable with how it handles cpp and header files. For this project NotePad++ was used as a easy to find free text editor that has wide spread use across the world.

Our next piece of software used to be preinstalled on Windows machines so there is a chance that the host PC might already have the application PuTTY installed. If not, the folks at https://www.putty.org/ still maintain a vigil for the SSH terminal

loved by all, overlooked by many. It is a stand alone application so after downloading the executable to our desktop for easy access we are done.

Lastly we need to install our toolchain software. There are two methods of doing this. First we could download all the source and build the compiler ourselves, or we can trust the installer from SourceForge. There are certainly pros and cons to each approach but as the first method requires a significant knowledge of package building and compiler options specific to your PC the risk of error is phenomenal. So for this project download the installer executable from SourceForge here: https://sourceforge.net/projects/yagarto/ and give it a good once over with the antivirus before installing.

The other perk of this installer is that is also providing us all the gcc libraries required to compile on ALL ARM architecture, not a specific chip as seen with other tool chains. Once the installer has finished there is an additional step we can utilize to simplify our life. All computers have a path variable which stores "shortcuts" to executable files allowing the user to access that executable even if they are not working in that executable's directory. Task Manager is an example of this. In our make files when we build the kernel we will be performing multiple calls to the toolchain. In order to save ourselves some pain we will add the path of the bin folder located in the Yagarto2012 directory. (The one you just installed)

B.2 Development Environment

With all our tools in place we can know build our testbed or development environment. From this project you will be provided the a directory with the following directory structure:

The project can be found here: https://github.com/samj3sus/Real-Pi

- BSP (Board Support Package)
- Build (Holds the object files once created)
- H (Header files reside here)

- Init (Initialization files are stored here, such as startup.s)
- Lib (Library files reside here)
- Port (Holds the ported CPU files)
- usrApp (Holds the example programs you will modify)
- uCos (Empty Folder)

No there is no typo on the last item. Due to distribution rites I am unable to provide you a copy of the uCOS-II Software. However, as required the textbook required by the course is the official Micro cOS-II textbook, it contains a CD with the uCOS-II files needed to populate the folder in our project. Simply copy the files directly into the uCos folder in the project directory (Do not add any subdirectories) and our development environment is complete.

B.3 Building the Kernel

With the makefile provided there is no need to make any modifications unless a different compiler was installed or the toolchain directory was not added to the PATH variable. With all MAKE files ours has a clean function and it's always a good practice to start with a clean run. To clean and build our kernel open a terminal window and navigate the project base directory. Once there we will type the following commands.

- make clean (Let this run to completion, should be quick)
- make (Will run considerably longer)

If we navigate back to our base directory we will now see a new file called kernel.img and no awards for guessing that this is our new kernel file for the SD card.

B.4 Running the Kernel

To run our project on the Pi we need to open the SD card loaded with Raspbian that we created earlier and see it's base directory. In the directory we can see there are two kernel files, kernel and kernel7. We need only replace the file kernel with our new kernel.img as the kernel7 applies to the Raspberry Pi 2 and 3 which is very much a horse of a different color. With the file replaced we can eject the SD Card and load it into the Raspberry Pi. If we simply turned on the Pi now, it would run! However we want to see what it is doing so we turn back to the Host PC to run a shell for the Pi to output to.

On our host PC we need to run the PuTTY SSH application. Upon loading we will get a configuration screen that is asking for a few pieces of information before we can run.

- Speed - Set to 115200 Baud (It works don't ask questions)
- Connection Type - Serial (We need the ANSI escape sequences)
- (What are the escape sequences? Ask Dr Egbert)
- Serial Line - Set this to the COM your Pi is using (Check Device Manager)

Once you have loaded your information click 'Open' and a terminal window will appear.

With PuTTY running we can now start our Pi and behold the glory of tasks! Following the pinout of our Pi we will connect our USB Serial cable to our PC. Keep flipping the cable USB connector 180 degrees until it is firmly secured in the host PC. Now connect the pins to the Pi as seen in Figure A.4. Once the power pins have been connected the PuTTY terminal screen will now start running the uCOS example loaded in the usrApp files. If all the steps were completed correctly the image in Figure A.5 should appear on your PuTTY terminal. With this you now have a complete working environment in which to modify the usrApp code and develop your own uCOS-II applications!

Figure B.1: Required Hardware sans PC- Pi 1

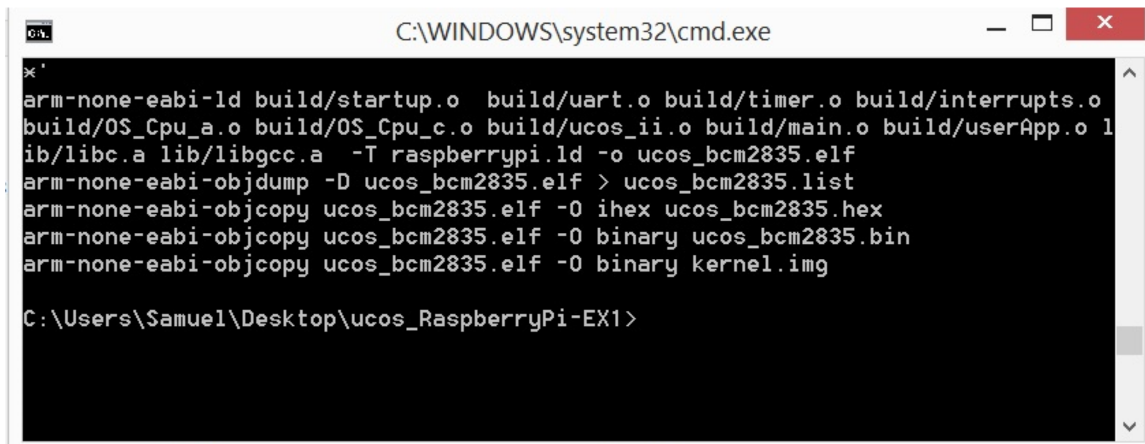

Figure B.2: A successful build output

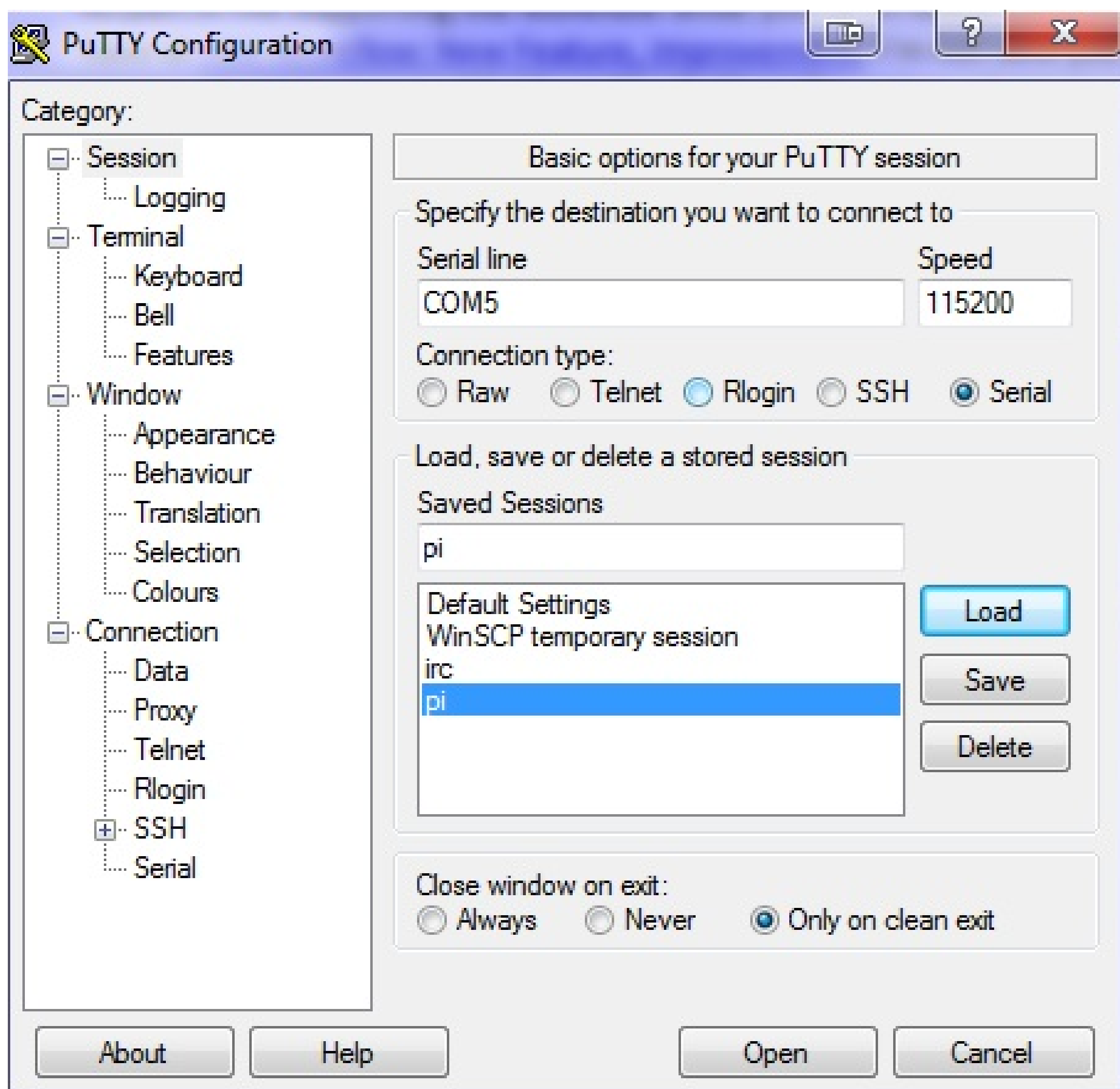

Figure B.3: Sample Putty Configuration

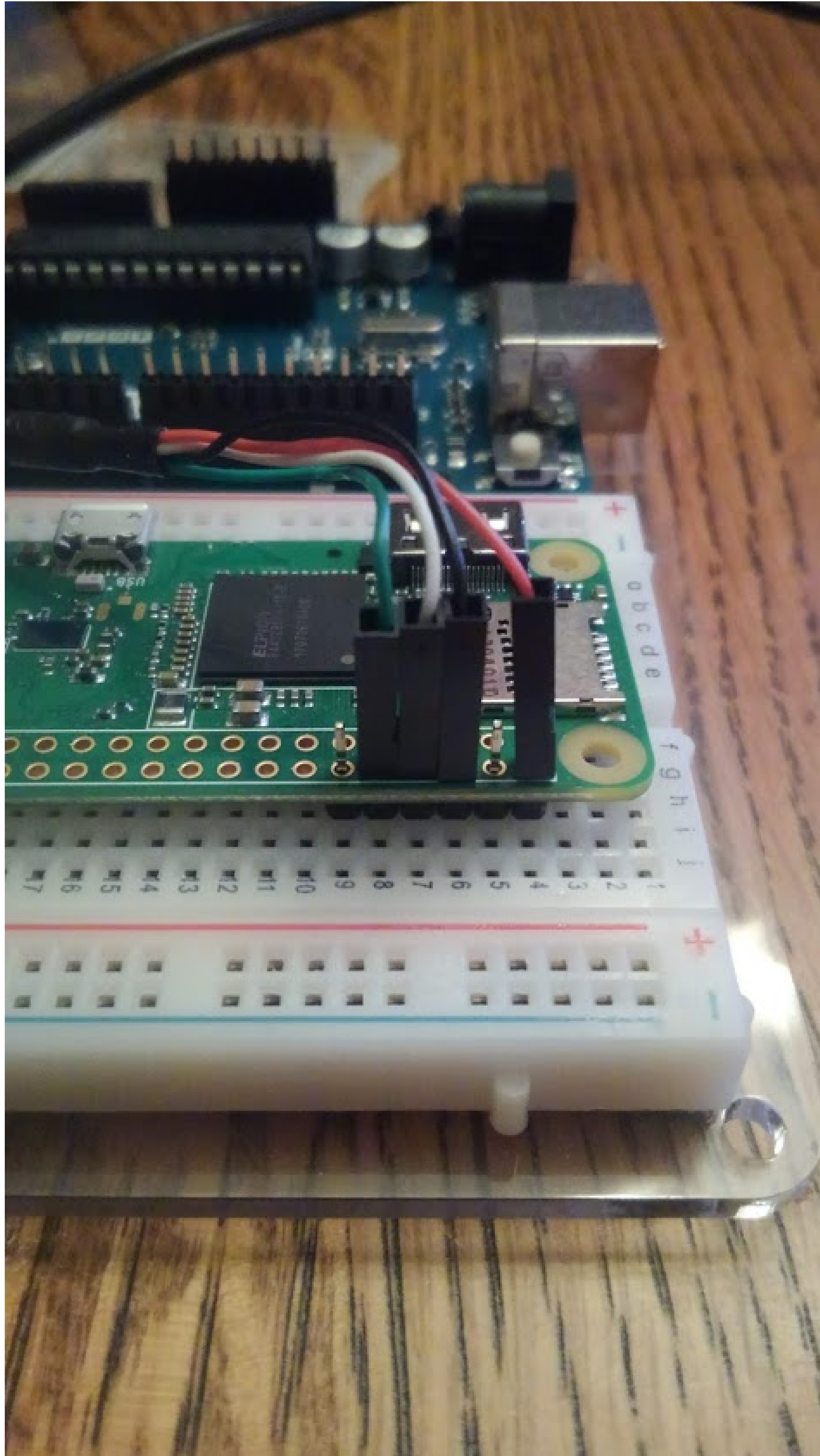

Figure B.4: Pin connections

Figure B.5: Running Example

Appendix C

Raspberry Pi 2 and 3 How To

C.1 Introduction and Set Up

At this point of the paper we will skip theory and process in favor of implementing Micro cOS-II on the Pi3. (If using the Pi2 follow all the same instructions as the project has been configured to run on both boards) In the appendix that follows all steps will be laid out in order to allow a student with the required materials to build their own development station for the Raspberry Pi 3.

The most critical part of this process is ensuring that all the required materials and software are accounted for before we even begin to load any images or compile any code. For this project you will need the following physical devices:

- Host PC (Desktop or Laptop) with USB ports
- Raspberry Pi 3
- Adafruit USB to Serial adapter
- MicroSD Card

With the hardware acquired we can now download the required software tools we will need to build our kernel. This list is not absolute as all developers may choose different tools to perform the same task, but as of this writing the tools below are readily available, free, and will work with this project.

- UnetBootin

- Yagarto Toolchain
- Notepad++
- Putty
- Latest GNU Embedded ARM Tools (Needed for the Pi 3's Cortex A53 processor)

To set up our development environment first we will start with the Pi 3 and then move to the PC. The Pi 3 may have shipped with a preloaded MicroSD card with Raspbian already loaded. If so great! Step one done. However if you did not receive a preloaded MicroSD Card please do the following:

- Download UnetBootin from SourceForge and save it to your Host PC
- Download Raspbian from https://www.raspberrypi.org/downloads/raspbian/
- Open UnetBootin and select the Disk Image radio button.
- Insert the MicroSD card into the Host PC
- In image, select the unzipped Rasbian image file
- For Type select USB drive and select the Micro SD Card drive letter
- Leave all other options alone and press OK

The process will take some time so be patient as the SD card is formatted and loaded with the Raspbian image. It's essential that this process complete without errors as it is building the bootable image we need to run our kernel.

With the Pi handled we now turn our attention to the Host PC. Our first order of business will be to download and install our text editing software. As stated before any text editor will do as long as you the operator are comfortable with how it handles cpp and header files. For this project NotePad++ was used as a easy to find free text editor that has wide spread use across the world.

Our next piece of software used to be preinstalled on Windows machines so there is a chance that the host PC might already have the application PuTTY installed. If not, the folks at https://www.putty.org/ still maintain a vigil for the SSH terminal loved by all, overlooked by many. It is a stand alone application so after downloading the executable to our desktop for easy access we are done.

Lastly we need to install our toolchain software. There are two methods of doing this. First we could download all the source and build the compiler ourselves, or we can trust the installer from SourceForge. There are certainly pros and cons to each approach but as the first method requires a significant knowledge of package building and compiler options specific to your PC the risk of error is phenomenal. So for this project download the installer executable from SourceForge here: https://sourceforge.net/projects/yagarto/ and give it a good once over with the antivirus before installing.

In addition to the Yagarto toolchain we will need to update the Yagarto tool with the latest GNU Embedded tools found here: https://developer.arm.com/open-source/gnu-toolchain/gnu-rm/downloads/. Once the installer has finished we can then move these new gcc compilers into our Yagarto directory replaceing the execuatables in place. There is an additional step we can utilize to simplify our life. All computers have a path variable which stores "shortcuts" to executable files allowing the user to access that executable even if they are not working in that executable's directory. Task Manager is an example of this. In our make files when we build the kernel we will be performing multiple calls to the toolchain. In order to save ourselves some pain we will add the path of the bin folder located in the Yagarto2012 directory. (The one you just installed)

C.2 Development Environment

With all our tools in place we can know build our testbed or development environment. From this project you will be provided the a directory with the following directory structure:

The project can be found here: https://github.com/samj3sus/Real-Pi

- BSP (Board Support Package)
- Build (Holds the object files once created)
- H (Header files reside here)
- Init (Initialization files are stored here, such as startup.s)
- Lib (Library files reside here)
- Port (Holds the ported CPU files)
- usrApp (Holds the example programs you will modify)
- uCos (Empty Folder)

No there is no typo on the last item. Due to distribution rites I am unable to provide you a copy of the uCOS-II Software. However, as required the textbook required by the course is the official Micro cOS-II textbook, it contains a CD with the uCOS-II files needed to populate the folder in our project. Simply copy the files directly into the uCos folder in the project directory (Do not add any subdirectories) and our development environment is complete.

C.3 Building the Kernel

With the makefile provided there is no need to make any modifications unless a different compiler was installed or the toolchain directory was not added to the PATH variable. With all MAKE files ours has a clean function and it's always a good practice to start with a clean run. To clean and build our kernel open a terminal window and navigate the project base directory. Once there we will type the following commands.

- make clean (Let this run to completion, should be quick)
- make (Will run considerably longer)

If we navigate back to our base directory we will now see a new file called kernel.img and no awards for guessing that this is our new kernel file for the SD card.

C.4 Running the Kernel

To run our project on the Pi 3 we need to open the SD card loaded with Raspbian that we created earlier and see it's base directory. In the directory we can see there are two kernel files, kernel and kernel7. We need only replace the file kernel7 with our new kernel7.img as the kernel applies to the Raspberry Pi 1 and zero. With the file replaced we can eject the SD Card and load it into the Raspberry Pi 3. If we simply turned on the Pi now, it would run! However we want to see what it is doing so we turn back to the Host PC to run a shell for the Pi to output to.

On our host PC we need to run the PuTTY SSH application. Upon loading we will get a configuration screen that is asking for a few pieces of information before we can run.

- Speed - Set to 115200 Baud (It works don't ask questions)
- Connection Type - Serial (We need the ANSI escape sequences)
- (What are the escape sequences? Ask Dr Egbert)
- Serial Line - Set this to the COM your Pi is using (Check Device Manager)

Once you have loaded your information click 'Open' and a terminal window will appear.

With PuTTY running we can now start our Pi and behold the glory of tasks! Following the pinout of our Pi we will connect our USB Serial cable to our PC. Keep flipping the cable USB connector 180 degrees until it is firmly secured in the host PC. Now connect the pins to the Pi as seen in Figure A.4. Once the power pins have been connected the PuTTY terminal screen will now start running the uCOS example loaded in the usrApp files. If all the steps were completed correctly the image

in Figure A.5 should appear on your PuTTY terminal. With this you now have a complete working environment in which to modify the usrApp code and develop your own uCOS-II applications!

Figure C.1: Required Hardware sans PC- Pi 3

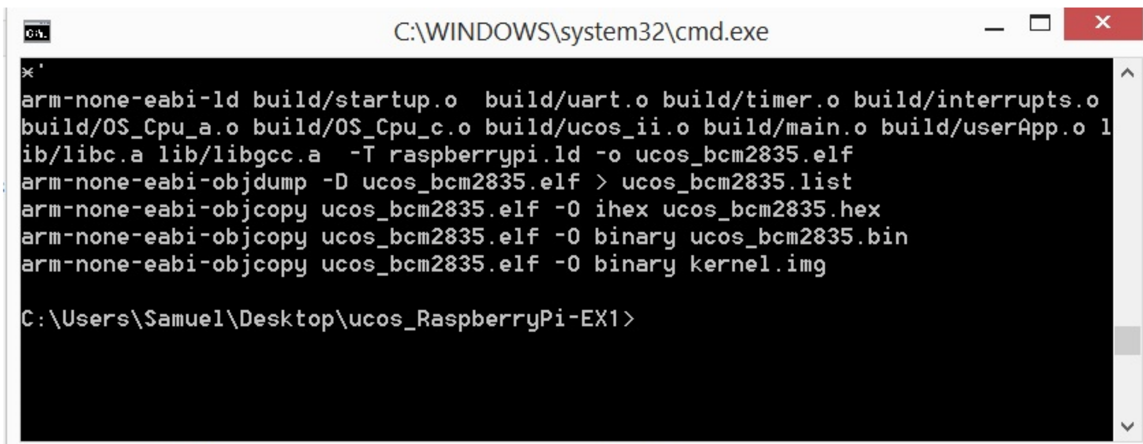

Figure C.2: A successful build output

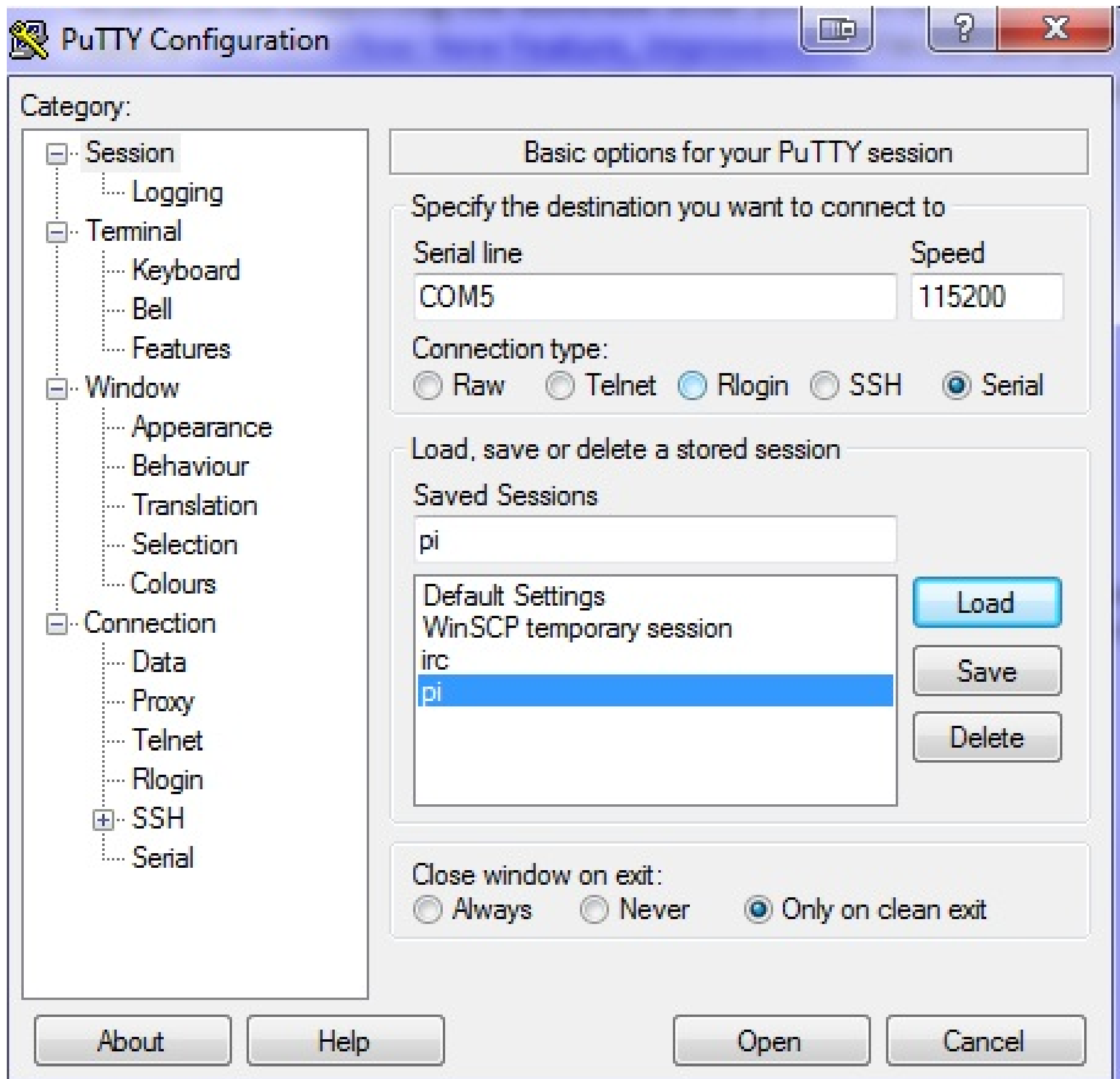

Figure C.3: Sample Putty Configuration

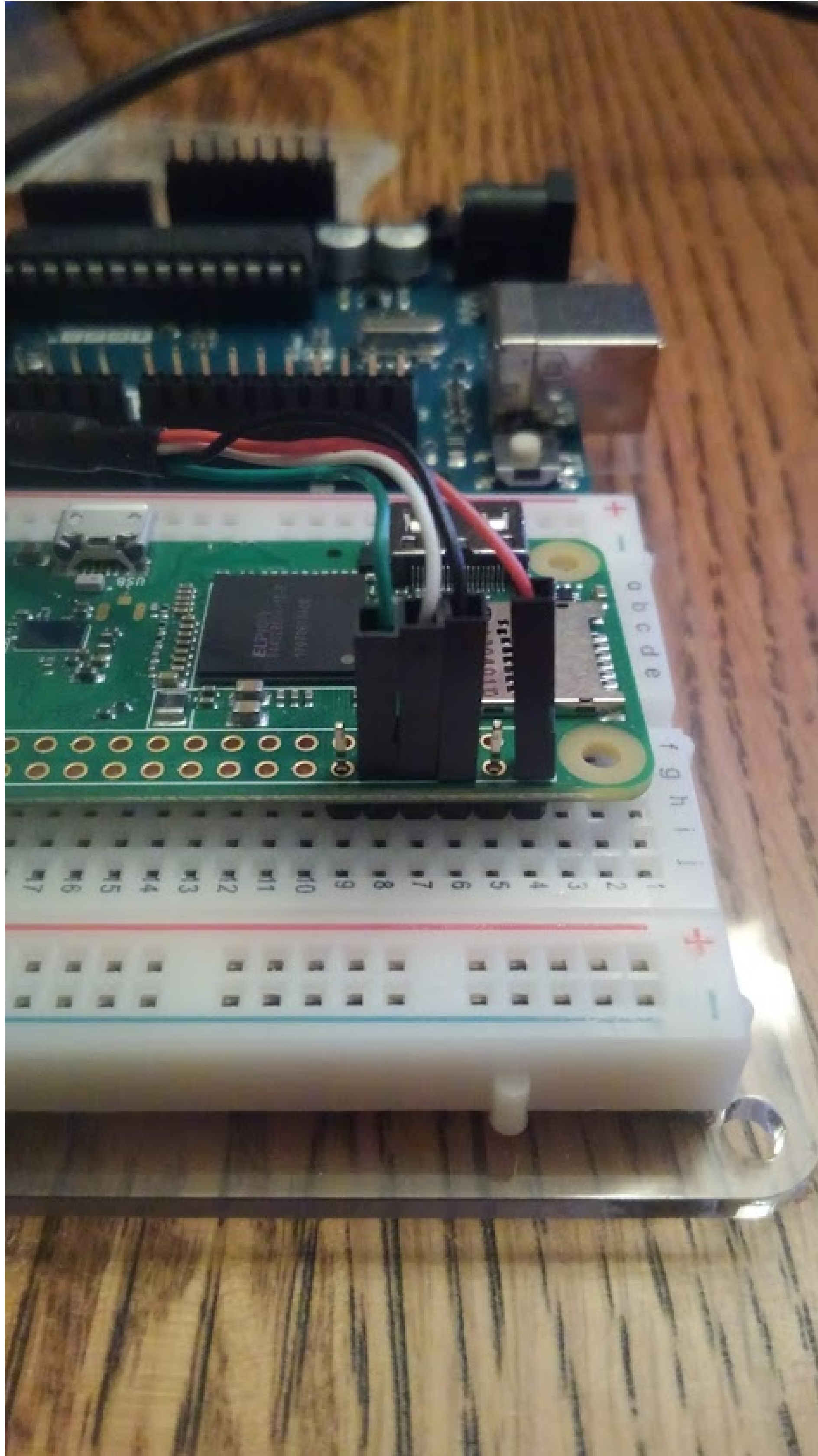

Figure C.4: Pin connections

```
COM4 - PuTTY

                         uC/OS-II, The Real-Time Kernel
                                Jean J. Labrosse

                                   EXAMPLE #1

06720461123638495921837529 4 5180649081049961089903920194338866736318347128046541
77324773646095581143995972934057290889480089048547875313547698510690903877868870
53057873106176643878052211862972624389487453440647493952440961141850648314951348
22543484760712843547853608599107596187414803415252939814119320638360208027623907
2273387382995500 1 347142225670051136110975237913919885747228710346087513470807 93
86112074975319169181094643152924853739364044311466736868824700490696843535694818
66153590972954157495096619350370857815058139692026072725456271391055219249055299
29344098064854403479459685714604820107081382310533866272910834366561978661891649
78125696289471896439566663528690712342265181100444893067700377353139064545641247
62620442333078283585491951149066965 3 91977031835391588633242285114055217597811 3
11099883642271258370577130163577484439950120939892246582196316503650399420549389
4198697419582462187829 50 039036713252835114788283455176142149878403124907586605 6
1704398995327790668799673212779 8 33026753301341377712867178087013449463949230452
0451598035822728426496522667775 80 2591385760832680799276247037721879083665076121
28573957459097693965871457535188437027371119742315810769108168056897120511920030
10615564361950050662167368152648938936161882493527855757308766336592177789484558

#Tasks           : 13      CPU Usage: 19  %
#Task switch/sec: 1
                          <-PRESS 'ESC' TO QUIT->
```

Figure C.5: Running Example