CMSC 498 Final Paper

John L Conway IV

May 13, 2010

Contents

1 Introduction

I had always thought school was the place you will learn what you need for your career. You learn how to make programs, how computers work, then you get a job and apply that knowledge. What you do learn in the workplace, is just how they are changing. I could never have been more wrong.

I started my current internship at Aegis, Inc. in September of 2009. I showed up to work on the second day of classes of my senior year ready to apply all the knowledge I had gained in class over the past three years. I knew they had a project ready for me to work on, but I had no idea what exactly that was. What I then proceeded to find out was that I had much more to learn at work then I ever could in school.

This paper will start with a description of my company and an overview of my project. I will then chronicle my steps while working on my project since last September and all of the different aspects of computers and working that I have encountered and learned from.

1.1 Aegis, Inc.

I work for a company called Aegis Incorporated located in Columbia, Maryland. They do a lot of government contracting as well as some products developed and sold in house. I found out about the company from a friend who was able to get me an interview at the end of last summer. I gladly took the position as it seemed to offer a challenging environment to work as well as a good chance at a job after college.

The company is owned and operated by two partners, Robert Corley and Ross Rigby. It is a fairly small company with about 25 employees total. I work in the research and development lab under my supervisor Michael Fay.

1.2 The Project

One of their in house products that they sell is the Network Receiver Controller, or NRC. The device takes inputs from eight receivers and allows clients to connect and control the data flow from those connections. Each connection, or channel, can be controlled individual by a connected client and options such as filtering and kurtosis values can be performed or calculated on the data.

My project is the next evolution of this product line. The latest NRC, the NRC 24, only supports sample rates of 48 kHz or lower. For the new generation, they wanted to support receiver input rates up to 192 kHz which puts the NRC into VHF range. Upping the sample rate poses a lot of problems to the system since a much greater amount of processing power is needed. Since the original design could not handle the load, the new NRC, called the NRC Flex would use microprocessors on each channel to perform processing on the data. This frees up the central processors to perform other functions as needed for the whole system.

I have been the main developer on the project thus far. I have had a large amount of help and guidance from my supervisor Michael as well as others in the company. However, the bulk of the actual work has been mine.

I was given the project at its infancy when I started my internship. The overall design and specs had been laid out. The microprocessor had been choses and as well as an analog to digital converter (ADC). A limited amount of work had been done to start the beginning kernel driver, however, not much headway had been made.

Overall, my project is split in a couple of different main sections, which I will use as the layout to this paper. The first task was to write a kernel driver and take data collected from the driver and send it into user space in linux. The second phase is to create a user level program to take the data and process it. Once it has been processed it will be send out an ethernet port. The final step, which will not be discussed in the paper as I have not gotten to that step yet, is write the central processing code to handle all the data from the channels

sent over the ethernet port as well as client connections and receiver information.

1.3 Learning Objectives

When I started working on the project, we set out a couple of goals for what I would focus my learning on. These goals follow the flow of the project closely and will be referenced throughout the paper.

1. To learn kernel and embedded systems programming.
2. To learn networking and programming of client server systems.
3. To learn new coding techniques and working with code outside of school.
4. Work on developing on a new project and apply skills learned in school to the new skills learned in the workplace.

2 Writing a Kernel Driver

The first task for the project was to create a Linux kernel driver. The design of the NRC has an ADC connected to the microprocessor by an I^2S bus which must be controlled by a driver to get the data off the bus. This is an awful lot of acronyms and tech talk for one sentence so I will explain step by step how this works.

2.1 The Basic Low Level Design

Since the NRC Flex takes data input from a receiver, the data must go into something to begin processing. This something is the ADC that will take analog signals and output a digital signal that a computer can understand. Once this magical step is performed by a microprocessor board, the data must be sent to the microprocessor that I will be programming to perform processing. The ADC we are using uses a special bus to output its data, and the microprocessor I am using will also decode that bus signal.

2.1.1 I^2S Bus

The bus is an Integrated Interchip Sound (I^2S) bus and is shown in Figure 1. This bus provides an easy way to send data at the rates we need (around 192kHz) and be able to decode that signal on the receiving end easily and accurately. The bus has three lines on it, the Left Right Clock (LRCLK), the Bit Clock (BCLK) and the Data Line. The data line is the actual data being transferred and the BCLK tells whatever is decoding the bus when to sample for each bit. Thus, 24 clock cycles on the BCLK is 24 bits of data. The LRCLK determines which channel the data is for, be it the right or left channel (since the I^2S bus is normally for audio signals). When we speak of a 192 kHz signal, we are referring to the speed of the LRCLK and how many samples we can receive.

2.1.2 The Microprocessor

The microprocessor will take the input data from the I^2S bus and decode the signal for data processing. We are using an Atmel AVR32 family processor to perform this decoding and processing of the data. This chip supports an embedded Linux system that is given to a developer along with a toolchain to create programs for the device.

The toolchain and development software is all created with Buildroot. This is an open source compilation platform that uses a structure of make files to build the entire embedded environment that can loaded onto the flash of a system. Buildroot handles downloading and compiling all the packages and programs required for an embedded Linux system, as well as download and compile the kernel, c library, bootloader, and filesystems. Since we are given a Buildroot system directly from Atmel, the system will also patch all the packages and system files so it will run on our specific board. Overall, the Buildroot system is a great help in building the development platform on my host machine as well as a custom Linux environment that can be copied onto my embedded system.

2.2 SSC and I2S

To start my first task I was given not as much information as I would have liked. I was thrown into a project and told that the driver to decode the I^2S data was going to need to be written and no one knew how to do that. So I dove headfirst into documentation to try to figure out what I needed to do. Since I knew very little about the AVR32 architecture and the I^2S bus, I needed to first figure out as much as I could about each of them. I spent a good month pouring over diagrams, Wikipedia, and countless datasheets on the AVR32. As an aside, whoever says Wikipedia is not a good resource has never used it for computer science information. It is one of the most helpful websites for understanding computer related topics, algorithms, and different architectures and how they work.

After pouring over the documentation I was still at a loss for how the system worked and how to begin to code a kernel driver for this system. I began to then look at other kernel drivers and how they worked. I found a few sound drivers that seemed to do what I was looking for. I began to look at the code and try to decipher what was going on and what parts of the code did what I was looking for. This is was an interesting and difficult process, one that I am glad I had to do.

Looking over and working with others code is one of the hardest things I have found with computer programming. Deciphering what is going on when you didn't put the time into writing it is something that is not often taught in school. When you do have to look at others code, it is usually not as complicated as something like a kernel driver. The difficulty of deciphering what was going on was quite a surprise to me. However, it was a great experience that continues to help me as I constantly have to work with other's code.

After a week or so of looking through kernel drivers, I started to get a feel for how everything was going work. The I^2S was controlled through the Synchronous Serial Controller (SSC). This was a hardware driven peripheral on the AVR32 that decoded the I^2S signal based on inputs and options that were put into a register. Once I started to figure out how to use the register and the basic controls given by another kernel driver, I was able to start writing the driver.

As I began writing the driver, I was constantly referring to books I had bought for my Operating Systems class. Having taken the class and gaining the basics of the kernel helped tremendously in my ability to write my own kernel driver and navigate my way around the kernel.

After finishing the first draft of the driver, I realized that it was not working the way I had intended. The AVR32 could not keep up with the interrupts of each sample coming into the SSC at 48kHz. This was quite a blow to have worked so hard on a driver and it fail with flying colors.

2.3 Direct Memory Access

So I went back to documentation to find out what I was doing wrong and how I could fix this problem. I was again reading the datasheets from Atmel, lots of Google searching, Wikipedia, but also a new resource that would turn out to be a tremendous help. I found a site called AVR Freaks that had a forum just for the AVR32. These forums had a plethora of information on all aspects of the AVR32. This would turn out to be a constant resource that I would continually turn to when I needed help on a problem.

My research lead me to Direct Memory Access (DMA). DMA allows the microprocessor to transfer data into ram without intervention of the processor and therefore freeing up tremendous amounts of CPU cycles. The problem now was figuring out how to get the DMA to work.

The Atmel chip uses another peripheral, set up much like the SSC, called the Peripheral DMA Controller, or PDC. The PDC when set up will automatically transfer data from the SSC into a buffer sent to the PDC. One of the tricky parts however is that the memory address of the buffer has to be a hardware address not a normal operating system address we get when we allocate arrays normally. So I had to dig into other kernel drivers again to find out how to make this work.

I spent about another month of trying to get the PDC to work before it actually did. I was now getting data only when the buffer was full, but the driver would still crash and burn occasionally with no apparent reason and did not have much uptime. This was becoming very frustrating as I couldn't track down where all the bugs were coming from. The whole driver was a ton of code that was everywhere from all the things I kept trying that either worked or didn't as I tried to get the PDC and SSC working.

2.4 The Final Driver

This is where I think the greatest improvement in my coding style happened. This was the key moment in my internship where things started working, clicked, and I started to understand where I was going wrong.

As I was working on trying to find bugs, Corley came in to see how I was doing. After explaining my frustration at how things were not working he started looking over the code with me. As we looked through trying to find issues, he started giving me pointers on how to set up my code, different ways of structuring functions, and a new way to handle the flow of my code. The biggest thing was handling the flow of function by a return value, so if anything goes wrong we can clean up and return, but always returning at the end of function. In this way, we check for errors at each step, then before the next step we check to return value to see if any of the previous things tripped an error. If so we clean up and head right down to the bottom of the function since nothing else will get done with the return value holding an error.

This, among lots of other little things, including a macro to hide printf() functions and print based on a verbosity, have changed the way I code. As a result I can code more accurate programs much quicker than I was able to before. It has been one of those defining moments of my school and professional career where I suddenly get how to code effiently without forgetting too many small things and while always building a production quality program even for the simplest things. I am certain that at this point I have been avoiding many bugs that I normally would have produced.

After restructuring the program I recompiled and ran it. It took about a day of working with Corley to go through my driver and restructure things. In the end, without logically changing anything or hunting down bugs, it worked. The driver worked exactly how I expected it too. The best thing was, all the bugs were gone by just changing the style of the code.

2.5 Testing

It now came to the point where I needed to start testing the device. This is one of the aspects of my internship I was not very familiar with. I had never done much testing on projects before except just seeing if the narrow constraints we were given worked. I actually would have liked to have had a class on testing programs and creating testing scripts and how to apply them to our programs.

The driver did not require as much testing as the later user code would, mainly because the driver did one and only one job. Get data to from the SSC to userspace. Thus, the testing on the driver involved verifying that the data over the I^2S was the same data that was being sent to user space. To do this we needed to send a signal over the I^2S bus and then save the data off the driver I had written.

So I wrote a test program to grab the data we needed from the driver and save it to a file so we can then verify the data. During the first tests I was using music off my iPod (since there was an iPod connector already hooked into the system when I came) and then trying to listen to the music again by importing the saved file into Audacity. These tests proved to be hard since the signal was music. Another lesson was learned here: known signals should be simple and easy to verify. Music does not satisfy the simple aspect in any way.

I was then given a small device that would output I^2S data in the form of a decrementing counter. This was simple and verifiable, especially since all I had to do was verify the previous sample was equal to one more than the next. So now, instead of transferring files into Audacity, I could not just write a program that would check if each sample was a decrement of the previous.

We were able to verify that most of the data was correct, however, we did find that the I^2S cable we had made to connect the I^2S output device to my microprocessor was not shielded and random bits were being inserted into my data. After making a better cable, we found almost 100% data integrity, which was a great self confidence boost. I had written a

kernel driver that worked.

2.6 Lessons from Writing a Kernel Driver

Writing the kernel driver was a challenging and rewarding project. I was excited when I first learned that's what my first task at my internship would be. I am a fan of low level programming so it fit well with my interests. Although it was interesting and rewarding, it had it's moments of frustration.

The main lessons I learned in this process were coding style and testing. I grew tremendously in my coding style and feel that I learned more in how to code well than how to code a kernel driver or a test program. All the programs, both for school and work, that I have written since writing this driver have been much easier to read, analyze, and have been written with fewer bugs than my previous programs. This has also allowed to me write quicker test programs, which is a great benefit since these programs are made to be used a few times to verify something then never used again.

I have also learned a lot on how to approach reading and understanding other's code. This has especially driven me to produce much more readable code since other people will more than likely have to read my code. After seeing some horribly documented and formatted code, I feel it is my obligation to write readable and well documented code.

Overall I feel I have accomplished my original goal of learning kernel and embedded systems programming, as well as to learn new coding techniques and apply skills from school in my work at Aegis. I learned the complexity of embedded systems and kernel development. I feel much more confident and comfortable working with the Linux kernel and writing drivers for it. I have also grown as a programmer and a computer scientist through my newly gained knowledge of new coding techniques, testing techniques, and overall design an development processes. The great thing about feeling accomplished after all I had done so far is that I still had a lot more to do and much more to learn.

3 Writing the Userspace Code

With the kernel driver now complete, it was time to move on to the next phase of the project. The next phase was to get the data from the driver, process it, then send it out an ethernet port. The processing mainly involved running filters on the data, though finding kurtosis and time stamping could also be done.

To begin I was directed towards the previous generations code. The old NRC code has most of the code written and I could then take it and change what I needed to work in this instance. It seemed pretty straightforward, so I dived right in.

3.1 Looking at Previous Code

I began to take a look at the previous code. There was a lot of C++ classes and some documentation that I had to start sorting through. I found a group of Java Doc style documentation and began to take a look at that.

The documentation was for the common classes that all aspects of the old code used. These common classes included things like a Socket Manager class and a Logger class. There was a lot of code that didn't make much sense to me yet and I started to become aware of the fact that I still didn't quite understand the overall big picture view of the project.

This is where I learned another important lesson. You need to know what your project is if you expect to get anywhere. I was pouring over the documentation and code looking for a specific thing, but bypassing a lot of useful information because I didn't understand the big picture. So after a good week of reading, I got Michael to give me the overview of the project. And boy did that make a difference. Once I understood what we were trying to do in the end, it became much clearer how I should approach what I was doing.

I decided after careful review of all the documentation and code that what I need was already almost there. All I needed to do was copy the code and start modifying things to

work with my driver and the AVR32 architecture. Then I could strip out things I didn't need and have a system ready to go.

So again, I dove headfirst into writing code, though this time, it wasn't from scratch. That did prove to be interesting since I was again reading other people's code, trying to understand it, then modifying it to suit my needs.

I started ripping things out I didn't need, adding things in, and just overall tearing apart the code. After a week or so I had something to try. I just needed to get the code on AVR32 and run it.

3.2 Cross Compiling

Here is where I ran into trouble again. When I went to compile the code, I had no idea what to do. The driver code was fairly straightforward, and I had written my own makefile for that. However, I had a project now that was already organized and set up to make. But I needed to change the makefiles to use my cross compiler, c library and other packages that worked for the AVR32. This proved to be a challenge that would end up teaching me a lot about makefiles.

I quickly found that I had to trace variables and things through many different makefiles to find out where they were declared. I also had to read a few books on Make and how large projects use makefiles. This lead me to take the plunge and subscribe to O'Reilly Books online. This service is a great resource that has every O'Reilly book as well as thousands others that you can view for about $20 a month. It is an invaluable resource to programmers at any level.

After figuring out the basics of Make I was able to change the compiler options and start compiling. It still didn't work. I need Xerces, and XML parser library. So I went into Buildroot to add the package so it would compile.

Again, I started to learn something new at this point. Adding Xerces, which should have

been a simple matter of clicking a check box in a configuration menu, did not end up being so easy. The packages were no longer where they should have been online, and when I did find them, they wouldn't compile either. Turns out, there is a bug in Buildroot where the Xerces package at the version I need didn't work. This was my first bug that I encountered that I didn't create. Also, other programs and packages are not only not perfect, but can have flaws that are huge. Through messing with Buildroot and the Xerces package, I learned that not everything is perfectly written and that the way some people decide to have their program work is not always the right way. I have a love/hate relationship with Buildroot now, and although it is helpful in many occasions, it is frustrating and annoying in many others.

3.3 Testing

Through a lot of frustration with Buildroot, I was able to get the program compiled and learned a tremendous amount about cross compiling as well. I copied the new executables onto the AVR32 and started to run them. With a few tweaks and bug fixes, the program was running beautifully.

It was time to start testing. I had to now verify that data was going from the driver to a client PC without any data integrity issues. So, I wrote a test client to connect to my new userspace code and started grabbing data. Everything was going ok and data integrity was good. So I decided to turn on Filtering and start testing that.

3.4 Looking at Performance

The program that I was writing needs to be able to handle about 8 connections with about 4 different kinds of filters. When we started testing with 1 filter and 1 connection, I found that I was using a lot of CPU cycles. So I started adding channels until I started loosing

data integrity. I was only able to handle 2 channels before it became too much processing for the AVR32 to handle.

So I needed to see why it was taking so much time to process data. I started looking for a code profiling tool that I could use to profile where the code was spending its time. After about a week of searching for tools to use, I found that few tools would work when cross compiled on the AVR32. So I was forced to recompile the code to run on x86 processors and profile on my host machine so I could at least get an idea of what was going on, but knowing full well that it was not 100% accurate since the x86 instruction set would work much differently than that AVR32. Using a profiler was something I had never done before and I welcomed the opportunity to learn how to use them. It proved a very educational experience and I am glad to now know of the benefits and drawbacks to code profiling.

What I did find through profiling was that the program was spending all its time in two functions. Those happened to be the functions where the filtering was taking place. So I started to investigate why the filter was taking so much processor time. As I looked into the code I realized we were using floating point computations. So I started to look at the AVR32 datasheets to look up floating point performance. To my surprise I found out that the AVR32 does not have a Floating Point Unit (FPU). So, the compiler was translating the floating point operations into integer operations, but that means an exponential increase in computations compared to all integer or floating point with a FPU computations.

Again I was taken aback at how many things you take for granted when developing for full scale computers. Embedded systems bring a whole new set of challenges from compiling to doing floating point calculations and counting clock cycles to make sure your code is as efficient as possible.

Since the AVR32 didn't have a FPU and we needed to make a lot of floating point calculations, we started to look at processors that did have an FPU. Turns out there are not many. After searching for a day for microprocessors that would do the trick I started

looking at what others had done in this situation. I went back to the AVR Freaks forum and found that many people, especially those doing Digital Signal Processing (DSP), which is basically what I am doing, convert floating point operations to fixed point.

I had never head of fixed point until now. I looked up what fixed point operations were and the light bulb went off. Floating point, integers, and fixed point all made sense. As well as the complexities with floating point operations and why they need their own processing unit. Granted I had learned about that in CMSC 411, Computer Architecture, but it didn't click until now. I have had a lot of these epiphanies as I have been working outside of school.

After spending some time looking into fixed point it looked like that was what we had to do. I ran some tests on my code to see if the processor could handle that much processing, and for the most part it could. However, there was still a problem. Too many clock cycles were being used but I couldn't place why since there were so many things going on in the code I had taken.

I started to find out what was going on by performing some benchmarks to see what the processor could handle. I started by writing a program that would take data out of the driver, then send it out over the ethernet. That was it. Take a buffer and a sent it over the network. This program used less than a percent of cpu. So, next, I modified the program to split up the buffer, go over each sample and then put them back together in an output buffer to send over the network. This would simulate the processing that would be needed to set up each sample to be filtered. This program showed little gain in CPU use. So, lastly, I added the sample computations that would be done when the samples are filtered (though it wasn't actually being filtered since we needed a lot of precompiled data to filter correctly, so we used dummy data). This showed an increase of CPU use, but no where near what the old code was using.

3.5 Rewriting the Code

So a hard decision had to be made and another lesson learned. Do we rewrite the code? We decided that may be the only way to get the performance we needed off the processor. Whether or not this has paid off has yet to be seen as we have not gotten far enough along to determine if it was the right decision, but so far it is looking good.

I began to rewrite the code from scratch. Again, finding out more and more about how I approach coding and problems. Over about 3 weeks of writing code, I found that my ability to design code that would work well when written was not quite where it should be yet. I spent a few days coming up with a design for the whole system, writing down how it all worked, but when I started coding found out my design would never work. I did this quite a few times, and each time I became much more careful about how I tackled designing. I am not where near where I should be in this aspect of coding, however, it was nice to realize it was a skill I needed to work on.

When it came time to start testing the code and seeing how it performed versus the old code, the results were favorable. We were able to sustain 8 open connections and handling 8 filtering options at the same time.

With good results it became time to see where we were and what to do next. Up until this point we had been using a 48 kHz sample rate to transfer the data, since this was easier to verify. So we decided to up the sample rate to 96 kHz and 192 kHz (the optimal final sample rate) and see how it performed. As I started testing at these rates, I noticed that I was occasionally dropping samples. After closer inspection, it appeared I needed to put transmitting the data into a separate thread since the transmit part of the system would occasionally take too long and I would then miss getting data from my driver.

So I took a week or so and rewrote the program to work as a multi-threaded application. This gave me the results I needed, and my driver was telling me I wasn't dropping samples. So I started to test the data and see how my data integrity was.

3.6 Retesting

The first test I ran was not promising. The data had many samples that were wrong with no rhyme or reason to where they occurred. I started testing some more and decided to try the new code at 48 kHz. When I switched the data rate to 48 kHz, the data was correct. It was at this point that I realized that I never verified the data when I upped the sample rate. I had assumed that since my code works at 48 kHz, it must work at 96 kHz and 192 kHz since I didn't have to change any code. Well it turns out that assuming anything with computers is a bad idea.

I had to fess up to Michael that I never really tested the data integrity when we originally upped the sample rate before I wrote the threaded program. So we started to take a look at where the data could have been corrupted. I broke out some old test programs that tested data integrity on the AVR32 that I used to initially test the I^2S driver. When I started to run the program I realized that I needed to make some modifications, since along the way I had changed the buffer sizes and some other small things.

When I started to make some changes to the test program I realized how far I had come in my coding. The test program was a mess and barely work if at all with the new changes. It was amazing to see what I had created only a few months ago and how much I had learned and changed over that short period of time.

With the test program rewritten, I was able to conclude that my new userspace code was not causing the issues. So it was time to take a step back towards the source of the data and see what was going on. Since we cannot tap the data off the SSC before it gets to my driver, we went directly to the device that was outputting the I^2S data. That data was correct. So somewhere between the cable that was carrying the signal to the device driver, something was happening. We decided then to try to see what we could find out by using a different I^2S source.

The Texas Instruments ADC we were going to use in the final product was perfect. So

we input a known sine wave at 2 kHz and tried to see what came out. At 48 kHz the data was flawless. However, at 96 kHz the data was not correct. It was close but there were a lot of inconsistencies around the wave.

At this point we were using a signal generator to generate the sine wave and Audacity to analyze the wave on the other side. This isn't the most accurate setup, but it's close enough to see when samples are off. However, another problem arose. We couldn't test the data at 192 kHz. Audacity couldn't handle the speed and none of our logic analyzers would sample and decode I^2S signal at a high enough rate. We have still not found a solution to this problem of testing the data at 192 kHz.

Since the 96 kHz signal was wrong as well, we decided to continue trying to find the problem at that rate first. As we traced back through the system we found that out input signal was not as clean as we had hoped. The signal from the signal generator was being sent through a box that made the signal balanced then the signal was input into the ADC. The box was not outputting as clean a signal as we had originally thought. At this point it appears that at 96 kHz, the data integrity is fairly good. However, since we can't test at 192 kHz yet, I can't say for sure we are done. Plus, the original device that was outputting the I^2S signal is having bit shift issues at both 96 kHz and 192 kHz. Currently, these are the problems we are still trying to solve.

4 Conclusions

Throughout the past two semesters I have gone through a tremendous amount of situations, problems and bugs. I have written a Linux kernel driver, learned signal processing, done some computer engineering type things, and overall done way more with computers than I could have imagined. Taking this internship has been the best decision I have ever made and provided me with a lot of experience.

I came into the internship with a few goals. I feel that I have accomplished all of them and more. I got to write a kernel driver and learn about embedded programming. Along the way on accomplishing this task I learned about coding style, reading code, and the obvious Linux kernel internals. I also got to get a glimpse of how embedded systems work, how to program for them, and how the hardware is set up, especially in regards to peripherals like the SSC and PDC. I will say that taking CMSC 421, Operating Systems, and CMSC 411, Computer Architecture, were a great help in this aspect of my internship.

I also wanted to learn networking and programming of client and server systems. I have not only created one or two client server systems with my userspace code, but I also learned how TCP works over its predecessor UDP. I also learned how to use select() to monitor multiple connections. I learned how to pass data from two computers using a network. I had never taken a networks course so being able to learn hands on was a great experience.

I also wanted to learn new coding techniques and work with code outside of school. I wanted to accomplish this since all the code I had worked with until now had been in classes where the scope was not very broad and the coding wasn't to difficult. I wanted to see what "real" code looked like. I wanted to write code that was going to be used in something that other people depended on. Although my code written for this project has not been released, I know that it is the foundation of this new product which is gratifying to know. I have also gotten to look at large amounts of code from the linux kernel to previous incarnations of

the project I am working on and get exposure to code that was not written for projects in school.

Lastly, I wanted to work on developing something and apply what I had already learned. I found out that what I had already learned wasn't much compared to what I was going to learn, but it has been a rewarding experience to apply what I did know. Knowing C from CMSC 201 and C++ from CMSC 202 was a great benefit and I used that knowledge extensively in my coding. I also used a lot of the assembly concepts I learned from CMSC 313 when analyzing floating point operations and how my code translated into instructions to be processed.

Overall, this has been a rewarding experience. I have learned much more than I was able just at school and feel much more prepared to enter the workforce in the coming weeks. It is also reassuring to know that I will be able to continue my work on this project as a full time employee after graduation. I look forward to all the new things I will learn as I continue with the project.

Figure 1: The I^2S Bus