University of Victoria
Faculty of Engineering
ELEC 499 Report
5 April 2007

The Design of a USB Based Oscilloscope

Prepared for:

Paul Kraeutner
Project Supervisor
pkraeutner@shaw.ca

Prepared by:

Leah Dickinson	**Darcy Metz**	**Brian Walts**	**Aaron Willis**
00-24892	**01-24231**	**98-04542**	**03-26317**
ldickins@uvic.ca	**dmetz@uvic.ca**	**bwalts@uvic.ca**	**awillis@uvic.ca**

Abstract

With the availability of the USB 2 interface chips, and high speed A/D converters, there is a greater availability of standard engineering instruments implemented as USB based computer peripherals. This means that the hobbyists and engineering students will want such tools for their home computers to enable simple waveform evaluation. Four University of Victoria Electrical Engineering students were involved in a project to design a USB based oscilloscope as part of their graduating requirements.

This oscilloscope would be connected to the desktop, or laptop computer, and use the computer screen to display signal information. Electronics students and hobbyists working on home projects would use it. The oscilloscope would be able to process signals up to the 5 MHz range, while maintaining a low cost. It would use the high speed USB 2.0 port to connect to the computer.

This document covers all details of the project. The project is introduced listing the goals and implementation plans for the project. The project management is described. The project is then divided along logical lines, and each section is described in detail. The investigations into a hardware system to develop the oscilloscope on are detailed. The design work for future hardware development is presented. The firmware development is described and listed. The software system design and development are included. The integration and testing procedures are detailed. Each section is concluded with the results of the work completed during this term, and recommendations for future development. The report ends with an overall conclusion, and recommendations for future developments for the whole project.

Some hardware design work was completed, but no hardware was built this term. The Firmware and Software were developed and tested with an off the shelf USB development board. Future work would include realization of hardware design, and modifications to firmware and software.

Table 1: Table of Figures

1 Overview

USB peripherals are commonly used computer devices, most computers have USB ports on the easily accessed front panel these days. This makes USB a handy method of data transfer, and good target for engineer tools such as oscilloscopes, frequency generators and signal analyzers for the electrical hobbyists and engineering students. These devices are inexpensive compared to traditional tools with display modules and power supplies. Engineering students and hobbyist will want such tools for their home computers to enable simple waveform evaluation. Four 4th year university engineering students completed a project to design a USB 2.0 High speed oscilloscope. The team was called Novice Engineering Research Devices. The product was named NERDScope. This report documents the design and Implementation NERDScope.

The report first introduces the design concept and the project objectives. Next it details the management for the project, and the design rational used to make decisions for the project. The report covers the implementation of the project, breaking it up into three logical sections: Hardware, Firmware, and Software. This is followed by a Testing and Results summary, a System Integration description, and finally the Conclusions and Recommendations for any future works to be completed on this project.

1.1 Introduction

Computer based engineering instruments have become especially attractive since the advent of the USB ports. These ports make it easy to plug an instrument into the computer; the instrument often will not need a separate power supply, as the USB port will supply it. The computer handles the display so the instruments are cheaper to build and buy; a separate CRT, or LCD is not needed. The instruments are small, easily portable, and not as susceptible to damage due to the elimination of a display module. This decrease in price puts many of the instruments into the feasible price range of electronics students and hobbyists. There are many such instruments available that work in the 100kHz range and only a two to three hundred dollars to purchase. USB 2.0 made it possible to achieve higher data transfer rates, and instruments that work in the MHz range, but they are all more expensive. Market research has shown that PC based oscilloscopes that work up to the 5 MHz range are around $600.00 US dollars. Four graduating University of Victoria Engineering students have chosen to design a USB based oscilloscope for a 499 project as part of their graduating requirement. The NERDScope was designed to work into the 5 MHz range, but the target price for it was $200.00 US which should put it into the financial range of most students and hobbyists.

1.2 Objectives

- Design a USB 2.0 oscilloscope
    - High speed data transfer (up to 480 Mbites/sec)
    - Up to 5 MHz in frequency
    - Selling price around $200.00
    - Easy to use, just plug it in and use it
    - Provide a platform for future development of the oscilloscope
- Set up a website to market the oscilloscope

- Attend a poster presentation to present the oscilloscope

1.3 Theory

There are two types of traditional oscilloscopes: Capture oscilloscopes, and continuous oscilloscopes. As the names imply, the capture oscilloscopes will capture and display a single sweep when triggered. Any changes to the signal after the capture are not displayed. This will not display any transient spikes or other anomalies in the signals. The continuous oscilloscopes display the continuously changing signal. They will show transient spikes and changing signals but can be more difficult to obtain a useful display of the changing signal. The NERDscope is like a marriage of the two types of oscilloscopes. The NERDscope is continually capturing and displaying a sweep of the signal. Transients and changes are captured and displayed, then replaced by the next sweep across the display. Any sweep could be captured and stored in memory. A next generation version could store the data into a large buffer for future display and comparisons.

A quick search on the net showed the price range for some of the USB based oscilloscopes:

- Link Instruments with a 100 MHz bandwidth $950.00
- TiePie engineering with a 5 MHz BW and more grater than 600 Euroddollars
- PicoScope with a 2 MHz BW almost $400.00 US
- Cleverscope with a 25 MHz BW $1000.00 US
- USBee up to a 8 MHz BW $550.00 US

The NERDScope was designed to have a 5 MHz bandwidth, but it was designed to sell for approximately $200.00 US. This would make the NERDScope highly competitive in the USB based oscilloscope market.

2 Project Management

2.1 Division of Labour

The labour was divided among group members along logical system boundaries. The group was divided first into two teams: hardware and software. The roles and responsibilities of individual members are outlined below.

Leah Dickinson – Hardware Design

Responsibilities:

- Design and testing of filter and data acquisition circuit,
- Compilation of progress reports,
- Documentation of the hardware team's efforts,
- Integration of software team's documentation into final report,
- Development of web site content,
- Production of demonstration poster,
- Test and troubleshoot of custom PCB board if required.

Brian Walts – Hardware Design

Responsibilities:

- Design and testing of a custom USB transceiver PCB implementation,
- Integration of hardware subsystems onto custom PCB,
- Development of web site content,
- Production of demonstration poster,
- Documentation of Hardware team's design,
- Assembly of custom PCB board if required,
- Test and troubleshoot of custom PCB board if required.
- Manufacture of two-layer PCB if required.

Aaron Willis – Firmware Design

Responsibilities:

- Design, testing, and implementation of firmware system,
- Design of USB chip to peripheral interface,
- Integration of software to firmware interface library,
- Documentation of firmware design,
- Design of demonstration poster,
- Implementation, testing, and documentation of hardware and software used for project demonstration.

Darcy Metz – Software Design

Responsibilities:

- Design and documentation of overall system,
- Design and implementation of website,
- Development of website content,
- Design, testing and implementation of software system,
- Documentation of software design,
- Design of demonstration poster,
- Implementation, testing, and documentation of hardware and software used for project demonstration.

No project manager was assigned, and each group member was given substantial autonomy with regard to their project portion design and scheduling. This approach was taken in order to promote simultaneous development by all group members as well as minimizing project management overhead. The drawback to this type of organization is that no individual is tasked with the responsibility for the overall project schedule and integration across system boundaries.

2.2 Communication

It was decided that weekly meetings would be held throughout the project to coordinate progress and efforts. All communication outside of weekly meetings was through group broadcast email.

2.3 Design Decision Methodology

The effect of design decisions on the following factors were utilized as qualitative criteria to guide decision making:

- Adherence to manufacturer recommendations
- Realization of performance objectives
- Cost including both production and development costs
- Project schedule

- Adherence to generally accepted design practices

- Ease of use
- Subsystem independence/modularity/encapsulation
- System integration
- Ease of development
- Ease of testing
- Ease of required parts acquisition
- Accommodation of future upgrades
- Marketability

The first four items are listed in order of importance and have priority over all other considerations. With these exceptions, the remaining factors are listed in order of approximate priority. The priority ranking for all factors indicates the guideline used in weighing tradeoffs amongst them. It should be noted that for all but the initial four items, the priority ranking was flexible to some degree and individual design decisions are explained in their respective Design Rationale sections of this report.

3 Design Rationale

The key to this project was transferring data through the USB lines at a high rate of transfer. The theoretical maximum is 480 MB/s. The data must be put into a USB transceiver, and data must be passed to the computer using handshaking techniques. A second goal was to display the incoming as a representation of an analog voltage signal. Another third goal was to create the hardware system to acquire the said data, read it into the USB chip. This hardware system would also incorporate any hardware support needed for the USB chip to operate. If created this hardware system would be designed to have a bandwidth from 0 to 5 MHz. If this hardware system would prove too expensive, or complicated to be completed during the scheduled project time, a development board for the USB chip would be purchased to be used to transfer the data. There was considerable time and effort put into the hardware design as the team would have considered the project more of a success if a completed product had been produced. The team views this project as a long-term effort, and any hardware design completed this term that does not get implemented could be used in the future. The design rationale for the individual systems are listed in the appropriate system section of this document.

4 Proposed Design

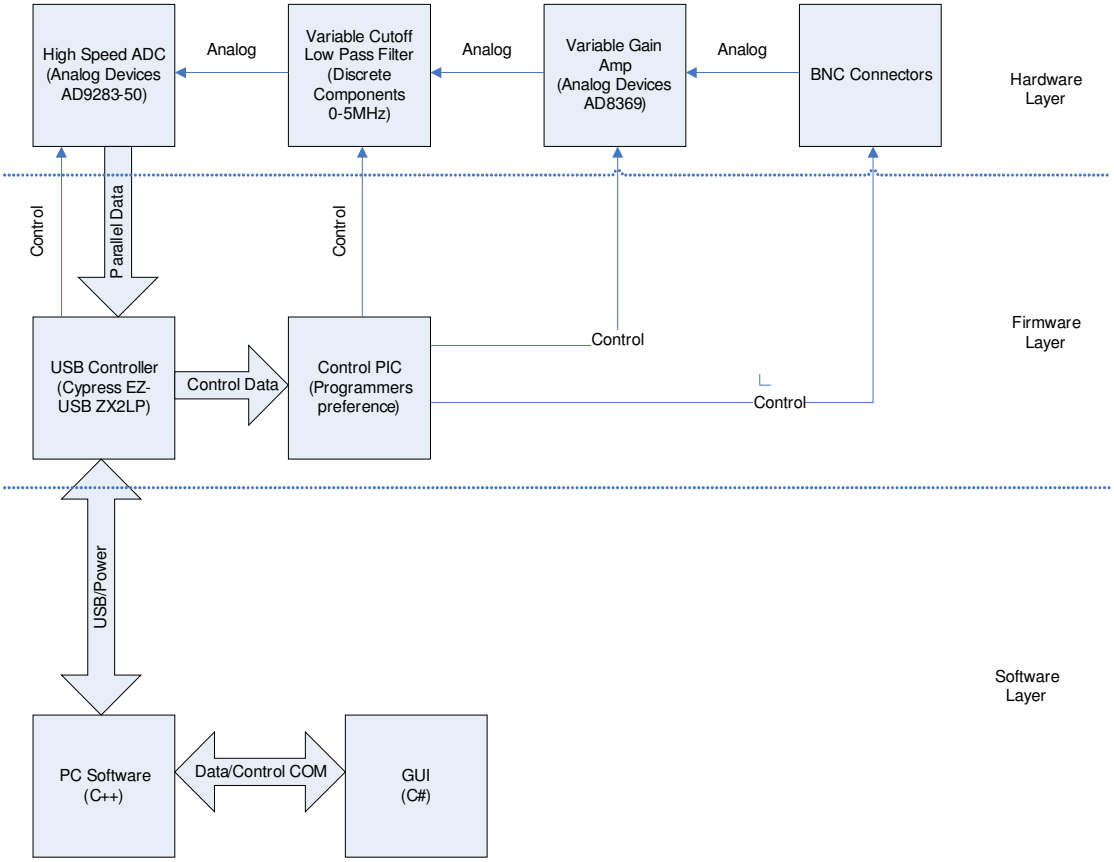


Figure 1Design Concept

This project divides logically into three sections: Hardware, Firmware, and Software.

4.1 Hardware

4.1.1 Overview

4.1.1.1 Introduction

The job of the hardware portion of the project was to amplify and filter an analog signal, convert it to an eight bit digital signal, and read this signal into the USB chip. After this point the firmware would come into play converting the data into USB data streams and sending to the computer on the USB data lines.

4.1.1.2 Objectives

- Use USB 2.0 port for communications with PC
- Use the USB 5V line to power the oscilloscope
- Capture and digitalize analog signals in frequencies up to 5 MHz
- Inexpensive to build, around $100.00

4.1.1.3 Theory

Whenever digital hardware designs include high frequencies, noise is an issue. All standard noise precautions should be followed, this includes bypass capacitors on the power supply and ground leads to all devices, keeping traces as short as possible, and separating analog and digital signals as much as possible. Digital signal processing boards are inherently noisy due to the alias frequencies created when sampling the signals. Any noise present on the PCB could cause errors in the digitalized signal to be sent to the computer and result in faulty representation of the measured signal by the oscilloscope. To minimize the effects of noise in the circuit, not only were the traditional precautions adhered to, and an anti-alias filter calculated, the data sheets of the high frequency components were studied for hints to reduce noise problems. Also low noise digital devices were chosen for the signal conditioning.

Before the filtered signal to be analyzed could be digitized, it needed to be adjusted to an appropriate level for the A/D. To obtain the best resolution from the A/D, the signal should have an amplitude matching the full amplitude of the A/D input. The reference level also should be the same for the A/D as the output of the filtering section.

4.1.2 Design Rational

Protel was chosen as a development environment, as one hardware team member had previous experience with it, and the other had some limited experience with it. It is also used commonly within the technical community. Protel is a software package that links schematics with Printed Circuit Board development and will produce Bill of Material documents and Gerber files used by the PCB manufactures.

Designing the PCB on a two-layer board and manufacturing the board at home was discussed. This solution would be cheaper than farming it out to a board manufacturer. It would also have a shorter turn around time, giving the software team a board to start development work on. The preparation for the PCB manufacture would be less complicated with this solution, but unfortunately the datasheets for both the USB and the A/D chips chosen specify the need for a four-layer board to keep reduce noise, so this solution was discarded. It was decided design the hardware for a four-layer board and obtain factory price estimates to see if this was a feasible solution for the project.

4.1.3 Design Implementation

The hardware section was further divided into two sections; the USB chip and the high speed A/D chip both required a four-layer board to eliminate noise issues, so they were grouped together in the high-speed section. The gain and filtering were the portions that made up the analog section of hardware. Designing the PCB in Protel took considerably longer than anticipated. At the end of February it became apparent that the entire hardware PCB would not be completed on schedule for the project; it was decided to only put the USB chip and A/D on the four-layer PCB, and design a two-layer daughter board to be manufactured at home with the filter and gain section. This daughter board adds flexibility to the project, as different boards could be developed for different applications: very low voltage input for analysis of noise problems, mid-range voltage input for display of TTL and CMOS logic signals, and high voltage input range for analysis of higher voltage analog signals.

4.1.3.1 Gain and Filtering

4.1.3.1.1 Overview

4.1.3.1.1.1 Introduction

A frequency analysis was conducted to determine the required anti-aliasing filter for the hardware design. The maximum A/D sample frequency of 20 MHz was used for this analysis.

The incoming signal could have different amplitude ranges, so some method of adjusting the gain is necessary to get a better resolution on the smaller signals, and keep the peaks on the larger signals. Two basic methods of achieving variable gain were explored, it was decided to use a variable gain amplifier chip.

4.1.3.1.1.2 Objectives

- Input impedance 50Ω
- Constant gain into the 5 MHz range
- Adjustable gain
- Reduction of alias frequency to smaller than resolution of A/D
- Out put voltage be compatible with A/D
- Be powered from USB 5V line

4.1.3.1.1.3 Theory

This board would introduce some noise due to the required connections to the rest of the hardware system. Many oscilloscope probes have 50Ω impedance, the board would introduce less noise if it has a matched input impedance.

The Maximum sample frequency for the A/D is 20MHz. The problematical alias frequency would be 15 MHz (20-5). According to the data sheets, both the A/D and the VGA gains were constant and stable well beyond this point. The anti-alias filter should reduce a 4V signal at 15 MHz to lower that the resolution of the A/D. In this case the signal needed to be reduced to below 16mV, or about –48 dB. To ensure the signal was flat in the 5 MHz range, the 3 dB point was chosen to be 5.3 MHz, a nine pole low-pass filter would be required. If an A/D with a sample frequency of 48 MHz, could be found, the theoretical maximum for the USB chip, then a three pole low-pass filter would be required.

4.1.3.1.2 Design and Design Rational

Due to time restraints and difficulties in board design the variable gain system was never implemented however parts were considered and some preliminary design work done.

The part originally chosen for the Variable Gain Amplifier was the AD605. This part allows voltage scaling of –14dB to 34dB with an output voltage range of 1 to 4 Volts. The problem with using this part was that the ability to detect the DC voltage present on the input signal was lost. Another issue with this part was that it has a limited output range of 3Vpp. All other chips originally investigated that could operate with this high of a bandwidth had much lower output voltage range typically under 1Vpp therefore this chip proved to have the best output range available. The loss of the ability to detect the DC component of the signal was deemed acceptable for prototyping purposes as continued research into this system was taking too much time. When this chip was chosen, the A/D was selected based on this choice.

Another solution discussed yet not fully developed was to put two switches in the system to determine the input path. Both switches would be controlled by the PIC. One switch would be closed to start. The switches would choose either a fixed negative gain amplifier, or a unity gain amplifier. Only one input would be chosen at a time. The rest of the system would be hooked up to a similar VGA yet one with both positive and negative voltage swings. This VGA would have a much lower output voltage range yet would allow us to capture the DC voltage present on the signal. This method was determined to require too much design time and as such was never fully developed.

After a mid-project consultation with the project supervisor, it was decided to divide hardware into two boards; one high-speed board with the USB chip and D/A, one two-layer board to be manufactured locally by a team member. The supervisor suggested the AD8331/2 chip, this was subsequently investigated and it was decided to use the AD8331 chip for the two-layer VGA solution for a few reasons. The active input impedance matching made it simple to match the input for a typical oscilloscope probe. The output could be matched to the full swing of the chosen D/A input, this would require less software tweaking for integration. This solution also was not realized. The schematic for the VGA section with the AD8331 chip is included in Appendix A.1. (see Figure 24: VGA System Schematic)

To use a variable gain it is necessary to have a method of requesting different gains. Three separate methods for doing this were explored.

The first method was to have the USB chip calculate the required reference and output it on the appropriate USB chip pins. However when operating in GPIF mode, all of the I/O lines available would already in use if there were two oscilloscope probes in use. A latch would be required to communicate the required gain range. The USB chip must be taken out of GPIF mode to calculate the reference and it would stop capturing incoming data. This would cause the oscilloscope to miss a significant amount of the data and increase the time until a useful signal is displayed after changing the gain.

The second method was to have the CPU do the calculations and send the results through the USB chip. This method would also require a latch. There is also concern that since the oscilloscope would be operating in windows the CPU could already be a bottleneck with the other windows processes it would be running. This solution would just increase the bottleneck problem.

It was decided that the best solution would be to use a separate PIC chip. A PIC chip would calculate the required analog voltage for the job, and convert it to a digital signal. This would decrease the time until a usable signal would be displayed. The digital signal would be sent to an D/A, and from there to the gain selection line of the VGA. A latch would not be required with a PIC. The following figure depicts the communication flow for a gain change.

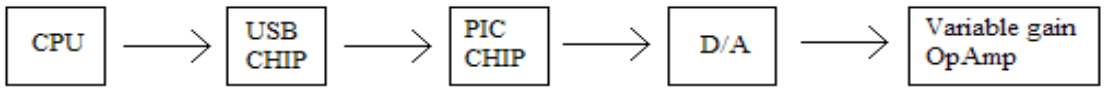


Figure 2: Communication Flow for Gain Change

The gain selections section was never designed.

The oscilloscope was required to work in the 5 MHz range, so the filter circuit required a faster than average Op-amp. After some investigations the LT1213 was chosen, it inexpensive, and has a gain-bandwidth product of 28 MHz.

4.1.3.1.3 Testing Plan

To test the VGA and filter section, apply a 5 VDC apply an adjustable voltage supply to the gain line of the VGA. Adjust the voltage set the gain to some preset value. Apply an AC signal frequency around 100 kHz with an appropriate amplitude to the input of the VGA. Monitor the output of the filter section with an oscilloscope. Adjust the voltage to the gain line to the maximum and minimum values for the VGA. Monitor the output should change according the data sheet for the VGA used. Set the gain to the middle. Adjust the frequency of the analog signal from 0 to 5 MHz. Monitor the output the output level should remain steady for the entire range of 0 to 5MHz. Perform this test with the gain set for maximum, and minimum.

4.1.3.1.4 System Integration

To integrate the analog section into the hardware system, the PIC section for selecting the required gain must still be designed, including firmware instructions. The regulated 3.3V, DGND and AGND lines for he PCB must be connected to the appropriate devices in the analog section.
A regulated 5 V signal must be applied to the 5V supply of the VGA chip. The output of the gain and filter section should be directly connected into the D/A input of the high-speed section.

4.1.3.1.5 Conclusions

- This section was not implemented, but was investigated for future works.

4.1.3.1.6 Recommendations

The PIC section for selecting the gain of the VGA should designed. The analog section should be built on a two-layer board using the AD8331 for the VGA. This section should be evolved and perfected using the proven development board before committing it to a four-layer board.

4.1.3.2 Analog to Digital Conversion System

4.1.3.2.1 Overview

4.1.3.2.1.1 Introduction

The A/D system is designed to convert the incoming analog voltage waveform to a digital parallel signal at speeds in sync with the capture speed of the USB chip.

4.1.3.2.1.2 Objectives

- High speed data transfer, synchronized with the USB data transfer rate
- Conversion of 5 MHz signal
- High sample rate, at least 4 times the highest data rate
- Low noise

4.1.3.2.1.3 Theory

When designing the A/D system it was important to keep in mind both timing and the speed of operation. One of the critical overall design goals was to be able to sample an incoming signal at speeds of 5MHz. To be able to accurately reproduce a repetitive sinusoidal signal according to nyquist theory there must have at least 2 samples per cycle. In practical application for an oscilloscope one is trying to determine more then just if a signal is present. With only two samples present for a signal, the representation would be very spiky, and there would be no assurance that the peak amplitudes are shown. Because of these issues it was decided that there would be at least 4 samples per cycle required to be able to reproduce a signal with reasonable accuracy. Therefore, for a 5 MHz incoming signal an A/D which could run at least 20 MHz would be required.

4.1.3.2.2 Design and Design Rationale

Choosing an A/D chip was a very difficult and time-consuming task. The most important criteria was that the A/D chip be capable of operating at the high speed of at least 20 Mega-samples /second. There are several chips available on the market that can operate at high speeds however almost all of them require very low input signals of the order of 1Vp-p.

The available time to design the PCB was getting short, so it was decided to use the TLC5510A for the A/D chip. This chip has a larger input range so there should be less noise present, it also meant less conditioning would be needed before the A/D. This chip uses an 8 bit parallel port for the output, which is what the USB chip would require for an input, again less integration hardware would be needed. The TLC5510 operates at 20 Mega-samples/second. This meets the required four-times sampling rate. It meant that the anti-alias filter would need a shaper roll-off, but this was an acceptable consequence. This A/D had the largest input voltage swing that we could find that operated in the desired frequency range. The output from the chosen VGA would range from 1-4V, therefore this A/D would not use the full out-put range. Also the software would need to convert the 2.5V code from the A/D to be zero, and all other voltage codes appropriately adjusted.

4.1.3.2.3 Testing plan

To test the A/D system, a 5VDC signal must be applied to the USB 5V line. 20 MHZ digital signal must be applied to the clock line, either using the clock signal provided by the USB IF line, or by not connecting that line and using a separate input signal. An analog signal with approximately 1kHz frequency, and a voltage swing from +1V to +4V must be applied to the ANALOG IN pin. A digital analyzer should be attached to the D1 to D7 lines, and the output examined. If no logic analyzer is available, two or more oscilloscope probes can be attached to the data lines individually and the output monitored. If possible the tests should be carried out in a range of frequencies from 0 to 5 MHz.

4.1.3.2.4 System Integration

This system was designed on the same board as the USB chip, so further integration would be needed on that side. The ANALOG IN line must be connected to the output of the filter and gain section, it should be connected with the shortest line possible.

4.1.3.2.5 Conclusion

Due to the high cost of the four-layer board required by the manufacturer's specifications, this section was not built. It was designed, and a PCB lay-out produced for future development purposes.

4.1.3.2.6 Recommendations

Continue with the 5510 A/D, and the four-layer board design. Try to get the board ordered early in the project to some of the cheaper options with a longer turn over time may be used.

4.1.3.3 USB System

The whole project would be based on the USB chip selection. Three USB chips were investigated for this selection.

4.1.3.3.1 Overview

4.1.3.3.1.1 Introduction

The USB chip had hardware jobs had firmware jobs to perform. The most important job was to act as a transceiver, and transfer high speed data on the USB data lines. A second job would be to read data into the data buffers. It must only read the data when the data ready, or have some method to discard erroneous data.

4.1.3.3.1.2 Objectives

- High speed USB data transceiver
- Data collection and temporary storage

4.1.3.3.1.3 Theory

The transceiver and data acquisition hardware design will be based on which USB chip is chosen. the preference would be a chip that most of the peripheral components and does not require much extra development to create the USB transceiver. The Firmware design will also be dependent on the USB chip.

4.1.3.3.2 Design and Design Rationale

The most important hardware decision was what USB chip to base the design against. Three chips were considered: The Phillips ISP 1581, the Cypress FXS 2, and the Cypress SX2 chips. Both Phillips and Cypress FXS 2 chips offered an integrated solution; no other hardware would be required for transceiver operation.

The Phillips solution was less expensive than the other two; it also had more onboard FIFO available. Unfortunately the Phillips solution had very little in the way of development tools, or documentation. This would greatly increase the time to develop the software. There was no renumeration feature available. The Phillips website said the chip discontinued, so there was some concern about future availability of the chip, and technical support available.

Cypress has substantially more development tools including software and application notes. Cypress also has a free software library that integrates with C#. This library implements many features, which would make the software development for the Cypress chip much easier. Cypress also has by far the largest selection of third party development boards available. Cypress has a software library available that is in second version, the original version was for visual studio v6, the current is for visual studio v8.

The Cypress SX2 chip was slightly cheaper than the FXS 2 chip, but it did not include an integrated solution, so after adding the appropriate hardware to create the transceiver, it was a more expensive solution than the FXS 2 chip.

The most important consideration in this decision was the time factor. It was decided that the available development tools more than made up for any deficiencies in the Cypress chips. It would be much easier to develop the product in the time frame available. The included software and documentation proved to be invaluable. The Cypress FXS 2 chip was chosen over the SX2 since the hardware development would be simpler and less expensive.

The type of package to be used was discussed, the QFN package was smaller in real estate, but it required a special heat transfer area below the package. A Copper fill would need to be designed into the PCB under the package, and a 5X5 array of vias would be needed within this heat transfer area. The vias add cost to the PCB. It was decided this was too complicated and expensive. The SSOP package was chosen. The 56-pin version of the available Cypress USB chips was chosen, it had all the pins needed to meet the performance objectives. This chip was also cheaper to buy, and required less traces on the board, so it was cheaper and faster to develop, test, and manufacture.

The datasheets (see Appendices on CD) for the Cypress specify that an external 24 MHz crystal with a +/- 100-ppm tolerance. An appropriate ASV crystal was chosen. The data sheet also specify that 12 pf capacitors with 5% tolerance be used, and state that the leads to the crystal be only 3 pf per side. This dictates that the crystal must be located on the same board as the USB chip. It was decided to locate the crystal as close physically as possible to the appropriate USB chip pins. The following is a diagram from the Cypress data sheets.

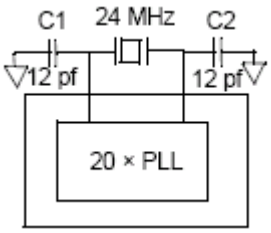


Figure 3 External 12MHz Crystal for on –chip Oscillator

The datasheet for the Cypress chip also list PCB lay-out recommendations that should be followed to maintain signal quality. These were considered when designing the PCB.

- At least a four-layer impedance controlled are required to maintain signal quality
- To control the impedance, maintain trace widths and trace spacing.
- Minimize stubs to minimize reflected signals
- Connections between the USB connector shell and signal ground must be done near the USB connector
- DPLUS and DMINUS trace lengths should be kept to within 2 mm of each other in length, with a preferred length of 20-30 mm
- A solid ground plane should be maintained under the DPLUS and DMINUS traces, the plane should not be split under these traces, including no vias placed on these trace routings
- Isolate the DPLUS and DMINUS traces from all other signal traces by no less than 10 mm

4.1.3.3.3 Testing plan

To test the USB section, the A/D section must be working. Connect the system to the host

computer via a USB cord. All voltage levels should be tested. The IFCLK pin should be checked to confirm the appropriate frequency is present.

4.1.3.3.4 System integration

The computer must be connected to the USB chip using the socket on the board, and a USB cable. The USB system was designed to be built on the same board as the D/A system, so no further integration would be needed for the D/A. A gain control circuit would need to be designed and connected to the USB system for gain to be adjusted with the computer.

4.1.3.3.5 Conclusion

This board was found to be too expensive to be built in the time remaining after it had been designed.

4.1.3.3.6 Recommendations

The design process should continue. The high-speed PCB with USB, and A/D should be ordered early in the project using some of the cheaper options with a longer turn over time. This board could be built and tested running the software that has been designed in this session of the project.

4.1.3.4 Board design

The board design was constantly evolving throughout the project as the design for the individual systems progressed. Each system imposed constraints on how the board was to be designed. It was decided at the beginning of the project to use the Protel software package to design the board as it is a commonly used industry tool for board design. Also all the fabrication companies are able to use protel99SE Gerber files to build PCBs.

4.1.3.4.1 Overview

4.1.3.4.1.1 Introduction

As the design progressed the board designs evolved. The evolution went as follows:

- Single two-layer board
- Single four-layer board
- Single four-layer board with a two-layer daughter board
- Development board with breadboard.

Each of the board designs is discussed and evaluated in the sections below.

4.1.3.4.1.2 Objectives

- Create a board design to encompass all hardware systems
- Create a board design that was inexpensive to manufacture
- Create a board design with low noise present
- Provide a board within the project time line for the software team to use for data transfer

4.1.3.4.1.3 Theory

The motivation for designing a PCB for the project is to realize a finished product. It would also be less noisy than a breadboard circuit designed for development purposes. It was not known at the beginning of the project whether a PCB was a viable option for software development purposes. The hardware team investigated several solutions for a development

board, the preference was to design and build a custom board for the project.

4.1.3.4.2 Design and Design Rationale

It was decided that if a custom board were to be developed, solder bridges would be used for jumpers. These jumpers would be a development aid; used to connect tools to the board when needed. They would be disconnected when not needed. This would introduce less noise than traditional multi-post jumpers or dip switches.
It was decided to include a 8-pin jumper with certain lines jumpered from the USB chip for development purposes:

- IFCLK could be used to synchronize with downstream peripherals. The GPIF is controlled by the IFCLK, as mentioned in the firmware section of this document. By breaking this line out, an off board clock could be used to take advantage of the allowable external clock frequencies of 5-45 MHz. This slower speed would allow easier development by allowing the use of a slower A/D.
- PWR could be used to power any downstream circuits, it could also be used as a reference for any debugging tools. This does have the possibility of adding noise to the circuit, and if the maximum current draw for the bus is exceeded, the bus will shut down
- CTL 0 was to be used a an output enable signal to allow sharing of the data bus lines between multiple devices: a high on CTL 0 line would enable the A/D to put data on the lines, a low would allow the Cypress to put data out for the PIC, for gain adjustment.
- Additional CTL lines could be used for debugging , they would show the states in the GPIF state machine. The 56 pin package was the cheaper option, but it did not include Gstate pins.
- RDY could be used by the PIC for handshaking when requesting a change of gain.

4.1.3.4.2.1.1 Two-Layer Single Board

It was originally intended to design and build a single two-layer board. This two-layer board would be cheap and could be manufactured by us. It has significantly better noise performance then a breadboard and would allow us to use surface mount parts, which is how most high frequency devices are manufactured.

Early on in the design it was discovered that the USB chip datasheets recommended a four-layer board be used to keep noise down. After performing some research online an example of a two-layer board design using the same chip was found so this option was not at first rejected. The ease and low cost of the manufacture was still considered high priority. As the research was conducted in A/D chips it was discovered that A/D chips that designed for the high frequencies required, required four-layer PCB designs. With the four-layer boards prescribed for two major chips in the hardware design the idea of a single two-layer board was discarded, and it was decided to use a single four-layer board design.

4.1.3.4.2.1.2 Four-Layer Single Board

During the initial discussions, some drawbacks to the single four-layer board concept were noted: this would go against the modular design principle, there would not be a separation between systems, which could cause more troubleshooting difficulties. This also limited the future upgrades; the data acquisition section would not be replaceable. Any errors in creating the board had the potential to render the entire board useless, so this could end up being a more expensive solution. The benefit to the system noise level was deemed to outweigh the downsides to creating one PCB on a four-layer board. Noise was a considerable concern and

the elimination of all but the oscilloscope probe and the USB cable was an attractive option. A single four-layer board would ensure that noise issues would be common throughout the circuit. A huge advantage to having a four-layer board is that it allows for separate power and ground planes greatly reducing noises present in the system.

During this design process, parts were still being changed as certain parameters become more critical. However some parts were finalized such as the USB chip, and simple voltage regulators. These were placed in the Protel schematic and the chip designs put into the Protel libraries. This process was taking considerably longer than anticipated, time was running out and the Filter and Gain section still had not being incorporated.

Some PCB manufacturer companies were investigated to get an estimate of the four-layer board price. It was noted that the price was increased by the square inch, and by the number of through holes. It would be a much less expensive board if the gain and filter sections were not included. It was a logical step to then have the USB chip and the A/D design on the four-layer board and place the variable gain system and the filtering system, which were less noise sensitive on a separate two-layer board. The two-layer board could be manufactured at home while awaiting the arrival of the four-layer board. This would invite noise into the system, but the time limitations had become a larger concern.

4.1.3.4.2.1.3 Single four-Layer Board with Two-Layer Daughter Board

Producing a four-layer board and a two-layer daughter board would allow the hardware team to focus on getting the four-layer board finished. This solution would still meet the manufactures' recommendations for using a four-layer board. This would provide a development board for firmware/software team to begin the testing, debugging and benchmarking process using the four-layer board and signal generator. The hardware group would be free to finalize an optimum design for the filtering and variable gain systems. The schematic for the four-layer board is shown in the Appendix A.3.1. Four-Layer Board PCB Schematic.

The schematic encapsulates the A/D and the USB systems. It has all the circuitry required to power and set up both these systems to run in the modes required. The USB chip was originally designed to operate in the Port Mode, however, this was questioned in consultations with the software team. After further investigations of Cypress USB documentation, it was decided that a better solution was to have the USB running in the GPIF mode. The GPIF mode was needed for high speed data transfer as it removes the 8051 from the data path. The circuit was re-designed to reflect this decision. The entire circuit is powered by the USB connection to the computer. USB provides a regulated 5 VDC according to the documented standards. For the devices used on board this had to be regulated to a 3.3V signal for power to the analog portions of the devices, and a separate 3.3V signal for the digital power of the devices. It was also required to create a stable 4 V reference for the A/D chip chosen. The LMS1585A was chosen for 3.3V regulation, mainly because it is a simple chip and has been used successfully in other projects. The data sheet is included with the appendices on CD. The LT1461 was chosen for similar reasons, and that data sheet is also included on the CD. The schematic has several breakout connectors allowing easy implementation of possible design changes. After this schematic had been approved by all team members the four-layer PCB was developed. The four-layer board layout is shown Appendix A.2.2.Four-Layer Board PCB Lay-out.

When designing the PCB some of the key points to consider were:

- The data sheet specifications with reference to the D+ and D- lines (see 4.1.3.3.2).
- How to set up the ground planes. An overall ground plane was required as well as an analog ground plane. According to best practices the two planes should connect at only one point to minimize noise and eliminate ground loops.

Once the PCB was finished it was Geber files needed to be created. The manufactures use the Gerfiles to manufacture the boards. Once this was completed the files sent out for quotes. It was found that the use of the split plane object used in Protel for ground and power planes was not compatible with the methods used by manufacturers. The ground and power planes had to be redesigned, this caused further routing problems due to more restrictive clearance issues. By the time all of these were resolved there was so little time left for the project, there would have only been under one week in which to test and assemble the board prior to presentation. Also it was found that the board would cost 339.00 USD because there was only going to be one board manufactured. Had the board gone into production it would have cost 12.71 USD per board for 100 of them. At this point it was decided that if there was going to be any data transferred over the USB, the best solution would be to purchase an off the shelf development board from Quick USB.

4.1.3.4.2.1.4 QuickUSB

The Quick USB development board was chosen for the large library of implemented firmware and software that assist fast development. The decision was based on the cost of development of the four-layer custom board, and the time remaining in the project schedule. This solution was in keeping with the modular idea; hardware modules could be designed at a later date to replace modules in the Quick USB development environment.

4.1.3.4.3 Conclusions

A single two layer board design that can guarantee operation at the high frequencies required is not possible due to the manufacturer's recommendations. The single four-layer board is a large project in itself, and easily slowed down by unforeseen problems. The four-layer high-speed board and a two-layer daughter-board would be an easier development path to follow. When the two-layer board is perfected it could be combined with the high-speed devices on a single four-layer board.

4.1.3.4.4 Recommendations

The hardware investigations and design efforts completed in this project should be used to continue the hardware development of the USB based oscilloscope. The high-speed four-layer PCB design should be ordered using a less expensive, longer turn-around time. The board should be ordered within one month of requiring it. A two-layer daughter board should be designed and manufactured. These two boards should be the next step in the hardware development process.

4.1.4 Testing Plan and Results Summary of Hardware

There was no hardware design implemented for this project. The software was tested on the Quick USB development board and the process is detailed in the Firmware section of the report.

The following is a cost analysis for the three of the development board options: single four-layer board, one four-layer board and one two-layer board, Quick USB development board
Single four-layer board

- Prototype four-layer board manufacturing $450.00
    - This design was not finished, so an estimate was not received, a ball-park figure of $450.00 can be used as it requires a larger board, and more through holes the actual figure may be higher.
- Total parts cost $88.00

The total cost of the single four-layer board would be $550.00 or more.

One four-layer board and a two-layer daughter board

- Prototype four-layer board manufacturing $380.00
- Prototype two-layer board manufacturing $50.00
- Total parts cost $104.00

The total cost of the single four-layer board would be $534.00.

Quick USB development board

- QuickUSB Module QUSB2 $149 US
- QuickUSB Adapter Board QUSBAB $79 US

The total cost of the Quick USB development board solutions was $230.00 US

It is important to note that if this oscilloscope were to become a marketable product it would minimize construction time to have the whole circuit on one four-layer board. Also if the company were to be selling over 100 of these the additional costs for the two-layer board with connectors would be much greater than having it all on one larger four-layer board. The plan to develop on a single four-layer board was discarded, however the finished product would be entirely on a single four-layer board. A price break-down for a single four-layer board for manufacturing follows. The price break-down is based on building 100 boards for retail purposes.

One four-layer boards

- one four-layer board manufacturing $20.00
- total parts cost per board $90.00

The total cost of the single four-layer board would be around $100.00 per board if 100 were manufactured. This leaves $100 per board for development costs (engineering costs), and packaging hardware. The Software would be sent over the net, anyone who wanted a hard copy would need to purchase that separately. This should keep the price of the oscilloscope within the desired price target of around $200.00.

4.1.5 System Integration Summary of Hardware

When the hardware system is built, the integration should be as simple as plugging it into the computer using a USB cable. The Firmware will be installed by the software in the computer once it has been published to an install file. During development stage, the manufacturer supplied firmware drivers will need to be installed. An oscilloscope probe will need to be plugged into the BNC jack on the board and used to obtain signals to be analyzed. In the final stage, the hardware system will be one four-layer board.

4.1.6 Conclusions Summary of Hardware

No hardware devices were manufactured during this term, but a considerable effort was put into investigating and designing hardware solutions that may be used for future development. An off the shelf USB development board was purchased and may be used as a proven platform for future development.

4.1.7 Recommendations Summary of Hardware

Recommendations have been listed for each sub-section within the hardware section. One key recommendation is to order a four-layer PCB for the USB and A/D section very early into the project so a longer turn-around time would be acceptable, this will decrease the manufacture price considerably, and the remainder of the development time could be spent on finalizing the VGA and filter sections. The VGA system should be built on a two-layer board, this would be a cheap way to develop and perfect this section. The gain selection section should be developed and added to the VGA system. These recommended boards should be used to replace the development board in the long run.

4.2 Firmware

4.2.1 Overview

4.2.1.1 Introduction

The firmware is software designed to be run on the 8051 CPU integrated into the Cypress FX2 silicon. This software manages data transfer between the Cypress data bus and the Cypress side of the USB cable.

4.2.1.2 Objectives

- Implement firmware capable of acquiring 8 bit samples from a downstream a/d system
- Implement firmware capable of transferring acquired data across USB cable to PC.

A fully functioning oscilloscope would require □i-directional transfer to the data acquisition hardware and would likely require generalized I/O from the 8051 CPU; both of these features would be realized through 8051 software design. As this project involved the creation of a simplified design with the focus on High Speed USB 2.0 compliant data acquisition only, neither data output nor generalized 8051 I/O were designed.

4.2.1.3 Theory

This section is a partial summary of information contained in the Cypress document number AN4067, *Endpoint FIFO Architecture of EZ-USB FX/FX2*. See also Appendix B1. The Cypress FX2 chip is a fully featured chip. Only the features that directly pertain to the scope of this project will discussed in this report.

In this section the term external devices refers to devices that communicate with the Cypress chip through port I/O pins configured either as discrete I/O or as a data bus; these devices are on the opposite of the Cypress chip in regard to data path from the host PC. All references to

onboard components refer to components within the Cypress chip.

The operation of the Cypress chip is logically divided into two domains: the USB domain and the Peripheral domain. Both the USB and Peripheral domains are configured and controlled by the onboard 8051 CPU.

The USB domain implements the transfer of data between the 4KB onboard FIFO storage and the host PC device. This domain consists of the USB transceiver and the Dysart USB Engine; it shares both the onboard FIFO storage and 16KB CPU ram with the peripheral domain. These on chip components can be seen in Figure 4 below. The Peripheral domain implements the transfer of data between external devices and the onboard FIFO storage. This domain consists of an eight or sixteen bit data bus, a nine bit address bus, a maximum size six bit control output bus, a maximum six bit input ready bus, and a GPIF engine. As stated above, the Peripheral domain shares both the onboard FIFO storage and 16KB CPU ram with the USB domain.

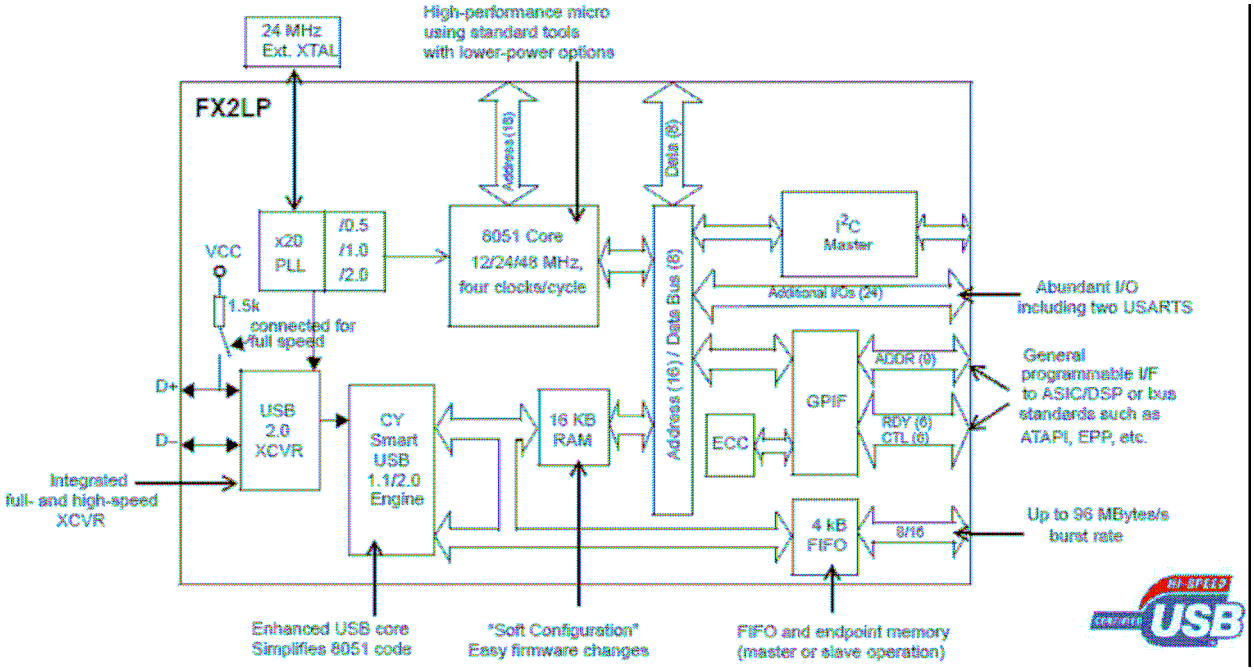


Figure 4: Cypress FX2 Block Diagram

4.2.2 Design rationale

Due to the implementation of the USB hardware with the QuickUSB development board, the firmware design sections of the Peripheral Domain and USB Domain consider only a QuickUSB implementation. This decision was made to ensure rapid development of a working prototype. Since the QuickUSB hardware implementation is shipped with an EEPROM based software library that encapsulates most of the firmware design, only configuration is required at this level. The disadvantage of the QuickUSB platform is the decrease in flexibility: custom waveforms at the Peripheral Domain level cannot be defined, and access to the 8051 CPU is limited to only a small subset of the CPU level registers normally exposed; no firmware coding is possible. It has been confirmed with Bitwise Systems that the QuickUSB firmware can be erased from the development board EEPROM which would allow development directly within

the 8051 CPU; the procedure has not been investigated further, however, and it is unclear whether the QuickUSB firmware could be re-installed after such a procedure.

Initial Cypress platform implementations were designed according to the methodology outlined in the Cypress *EZ-USB FX2 GPIF Primer* document. See also Appendix B 2. As it recommends, the firmware required to implement design decisions was added to a known good implementation referred to as a *Firmware Frameworks* project. The *Firmware Frameworks* projects are created and compiled with Keil uVision2 development environment. Both the Firmware Frameworks and an evaluation copy of the Keil uVision2 development environment are on the accompanying CD, see Appendix B 3.

The Cypress platform implementations were written at a basic level and did not represent a completed or tested implementation. This code was written based only on available documentation; it was written without the benefit of knowledge gained through explorative programming on actual Cypress silicon as no working hardware implementation was available. This code represented the level at which testing, explorative programming, and further refinement of the software could begin. As such, its value to subsequent development efforts was minimal; for the sake of brevity it was not included in this document. The investigation required to produce these small modifications of code, however, was instrumental in the rapid prototype development with the QuickUSB platform required at the end of this project; it is recommended that subsequent groups work through the design methodology outlined in the GPIF Primer if considering a migration to a Cypress firmware platform.

4.2.3 Design Implementation

All firmware implementations are discussed in their respective sections below.

4.2.3.1 Peripheral Domain

4.2.3.1.1 Overview

4.2.3.1.1.1 Introduction

The Peripheral Domain firmware defines the functionality required to transfer data between an external device and either the onboard 8051 CPU for slower applications, or between an external device and the onboard FIFO storage.

4.2.3.1.1.2 Objectives

- Implement firmware capable of acquiring 8 bit samples from a downstream A/D system.

4.2.3.1.1.3 Theory

This section is an abbreviated summary of information contained in the Cypress document number AN4067, *Endpoint FIFO Architecture of EZ-USB FX/FX2* as well as information contained in the Cypress document number 001-13670, *EZ-USB Technical Reference Manual*. See Appendix B 1 and Appendix B 4. As previously described, external devices are on the

opposite side of the Cypress chip with respect to data flow from the PC host.

The peripheral domain can operate in one of three modes: Port Mode, Slave Mode, or Master Mode. Each of these modes implements a different method of transferring data across the onboard I/O pins between an external device and the onboard FIFO storage.

The Port Mode operation of the Peripheral Domain involves the use of the 8051 explicitly to transfer data; all data transferred in this mode passes through the 8051 CPU. This mode is characterized by the 8051 CPU's access to data in the FIFO storage module. Code can be written to fill an endpoint in the FIFO, or to process endpoint data sent from the PC host. In addition, all port I/O pins are available to the 8051 CPU in order to communicate with external devices. This mode is the slowest transfer mode available as the 8051 CPU is directly involved with the data path.

In Master Mode the General Programmable Interface or GPIF is used to configure the transfer operation of data between the onboard FIFO storage and external devices. As seen in Figure 5 below, the GPIF is a state machine that serves as a master to a slave external device and defines the control signal waveforms that enable transfers across the data bus.

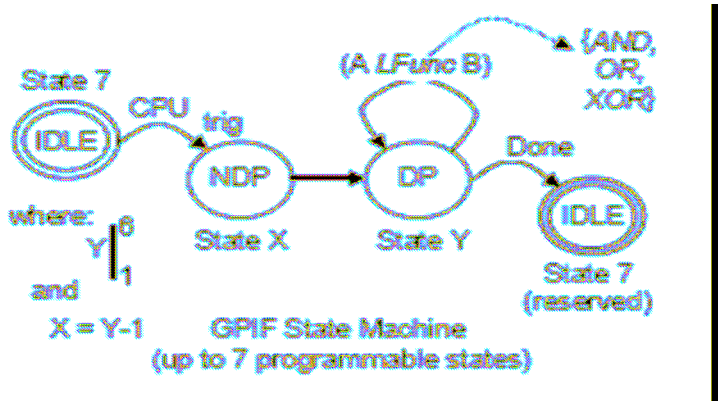


Figure 5:GPIF State Machine Diagram

The GPIF state machine is able to define the logic for six control signals, an address bus, and four input ready signals. In GPIF mode, Port B serves and as 8 bit data bus and can be combined with Port D to define a sixteen bit bus. Firmware in the 8051 CPU oversees the GPIF operation by defining the appropriate waveform and initiating a data transfer transaction, but remains out of the data path. While in Master Mode, the GPIF engine can be synchronized by an internally generated clock at either 30 or 48 MHz, or by an externally supplied clock operating between 5 and 45 MHz. The Master Mode is suitable for high speed data transfer.

In Slave Mode, the waveforms required to transfer data between the onboard FIFOs and an external device are created by the external device. The onboard FIFOs act as slaves to the off chip master. Similar to the Master Mode, while operating in Slave Mode the 8051 CPU serves only to configure the transaction but remains out of the data path. For this reason, Slave mode is also suitable for high speed data transfer.

The modes of operation are shown graphically in Figure 6 below.

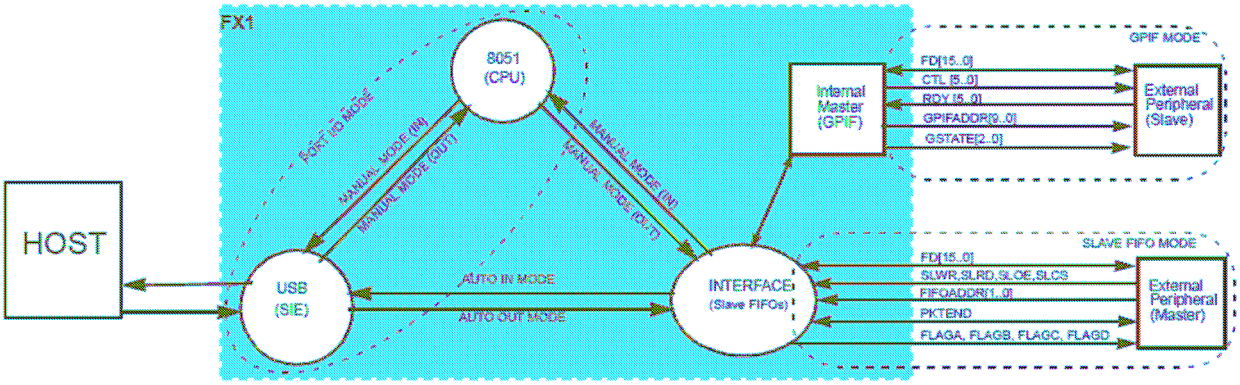


Figure 6: Peripheral Domain Modes of Operation

4.2.3.1.2 Design rationale

4.2.3.1.2.1 Peripheral Mode Selection

It was decided that this project would use Master Mode and the GPIF engine to control the transfer between an external device and the onboard FIFO storage. Using the onboard GPIF engine enabled the high speed transfers required to meet the performance objectives; this would not have been possible using Port Mode as the insertion of the 8051 CPU into the data path would have slowed down the transfers substantially. Additionally, since the GPIF is integrated into the Cypress chip, using it allowed a simpler and lower cost design than could have been achieved by utilizing the Slave Mode.

4.2.3.1.2.2 Data Bus Width Selection

It was decided that the data bus would be eight bits wide only. This size of bus was adequate to meet the performance objective of using eight bit samples. Matching the bus to the sample size increased the performance compared to the use of a sixteen bit bus as the unused bits would not have to be removed at the host side.

4.2.3.1.2.3 Waveform Specification

It was decided that a simple read waveform would be developed to be used in conjunction with the downstream A/D chip. This waveform would read one sample per clock cycle to maximize performance. As the project was designed as a continuously sampling oscilloscope, the read waveform was designed to continue indefinitely once started, exiting the state machine only if the transfer was aborted, a hardware level debug stop signal was generated, or if the destination FIFO was full.

It was decided that an additional handshake centered read waveform would be developed. This waveform was developed with the intention of being used for testing purposes as it would allow an asynchronous transfer that did not require that the external device have access to the GPIF clock signal. Additionally, it was felt that this read waveform would serve as the model for a future

handshake centered read waveform designed to communicate with an external microcontroller sharing the data bus.

4.2.3.1.2.4 GPIF Clock Source Selection

It was decided that during development, the GPIF engine would be controlled by an internally generated clock signal. This resulted in easier and less costly development.

4.2.3.1.3 Design Implementation

This section describes the use of information provided in the *QuickUSB User Guide* documentation. See Appendix B 5 for more information.

4.2.3.1.3.1 Peripheral Mode Selection

The QuickUSB implementation allows access to selection of the Peripheral Domain mode in two ways: through a compiled code library shipped with the product (See Section Firmware to Software Interface), and through an executable, QuickUSB Diagnostics; both products are shipped with the QuickUSB package. See Appendix B: Firmware Apendices.

In order to select GPIF Peripheral Mode the QuickUSB setting SETTING_FIFO_CONFIG, located at QuickUSB setting address 3 must be modified. Specifically bits 1 and 0 of the Least Significant Byte must be set to 1 and 0 respectively. The following pseudocode, using the QuickUSB library functions QuickUsbReadSetting and QuickUsbWriteSetting would set setting bits as defined by user input bit masks.

```
Variables:
DeviceHandle        //an integer value returned from QuickUsbOpenDevice function
                    //this pseudocode assumes that a device has already been opened
Address             //a two byte value corresponding to the QuickUSB setting address
FIFOConfigSetting   //a storage address for the setting

Call QuickUsbReadSetting (DeviceHandle, Address, FIFOConfigSetting)
FIFOConfigSetting = FIFOConfigSetting OR BitSetMask
FIFOConfigSetting = FIFOConfigSetting AND BitClearMask

Call QuickUsbWriteSetting (DeviceHandle, Address, FIFOConfigSetting)
```

Pseudocode 1: Modifying a Setting Using QuickUSB Library

To set GPIF master mode Pseudocode 1 would be implemented with Address = SETTING_FIFO_CONFIG or Hex 03, BitSetMask = Hex 0002, and BitClearMask = Hex FFFE.

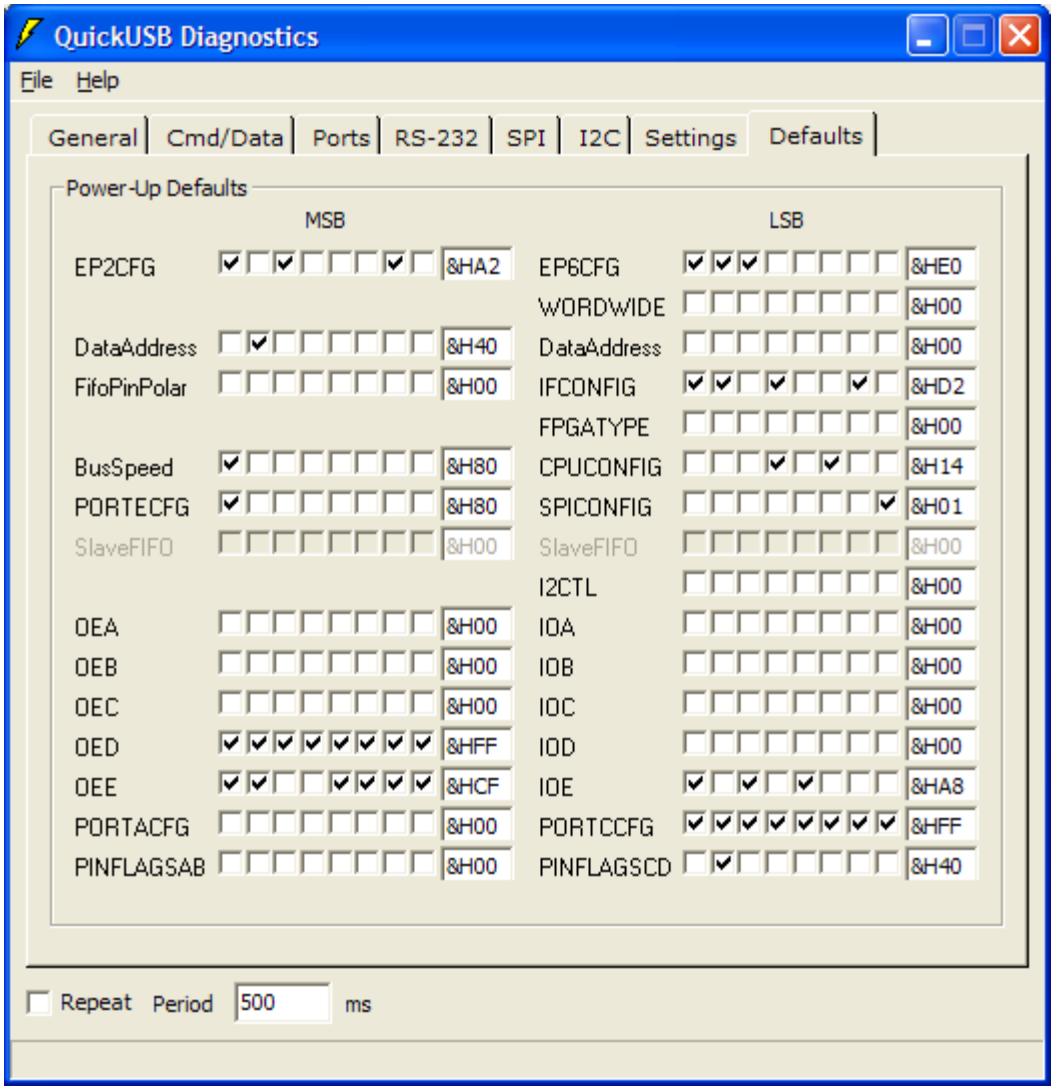


Figure 7: QuickUSB Diagnostics Program Defaults Tab

The Peripheral Mode can also be set using the QuickUSB Diagnostics program; the Default Settings tab of this program is shown in Figure 7 above. The Default Settings define non-volatile settings that are implemented when the power to the USB device is cycled; on cycling the Default settings replace those settings in the Settings tab. Checking IFCONFIG bit 1 and unchecking IFCONFIG bit 0 sets the Peripheral Mode to GPIF Master Mode. While in the program, hovering the mouse over these two bits displays a tool tip that shows possible alternative settings.

4.2.3.1.3.2 Data Bus Width Selection

As with setting the Peripheral Mode, there are several ways to implement an eight bit wide data bus. The SETTING_WORDWIDE bit 0 must be set to 0 for eight bit operation. The QuickUSB Library can be used to do this by implementing PseudoCode1 with Address = SETTING_WORDWIDE or Hex 01 and BitClearMask = Hex FFFE. Additionally, an eight bit bus can be implemented by unchecking the WORDWIDE bit 0 checkbox on the QuickUSB

Diagnostics tool Defaults Tab. Finally, an eight bit bus can be implemented by unchecking the Word Wide check box on the Cmd/Data tab on the QuickUSB Diagnostics tool. Like any changes made in the Settings tab however, any changes to the Word Wide check box on the Cmd/Data tab will be overridden by the settings in the Defaults Tab after a power cycle.

4.2.3.1.3.3 Waveform Specification

Unlike a Cypress platform implementation, waveforms in a QuickUSB implementation are predefined in compiled 8051 code that is loaded into the QuickUSB EEPROM. There are five predefined waveforms applicable to GPIF master mode: Simple I/O Model, FIFO Handshake I/O Model, Full Handshake I/O model, Block Handshake I/O model, and Pipeline I/O Model. The Simple I/O Model was selected as the most appropriate waveform for the completed scope; while it does not read data in continuously as suggested in the design rationale, it most closely matched the intended design. Its use is based on the assumption that the A/D chip external to the Cypress chip has a setup time less than half of an IFCLK clock cycle. The Simple I/O Model waveform is shown below in Figure 8.

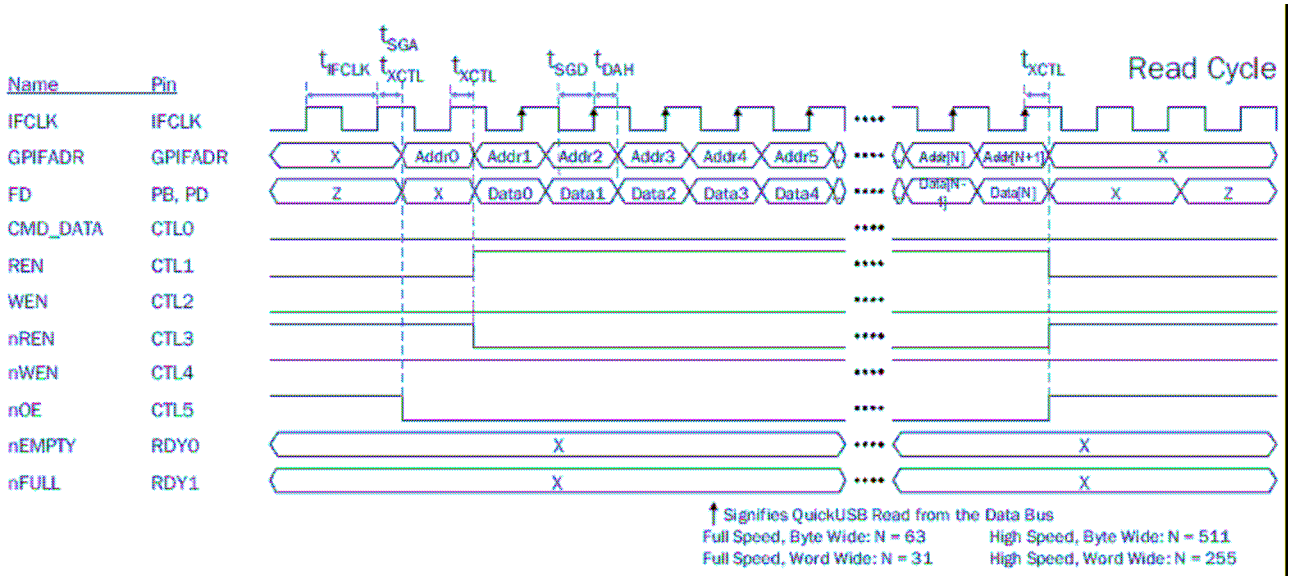


Figure 8: Simple I/O Model Waveform Diagram

Although the Simple I/O Model was selected for the final design due to its speed, a handshaking protocol was selected for development. This made it unnecessary to break out the IFCLK signal and relied on asynchronous transfers – a situation appropriate for bread boarding during testing. Of the five predefined waveforms available with the QuickUSB package, the Full Handshake I/O Model was selected for development testing. The Full Handshake I/O Model waveform is shown below in Figure 9.

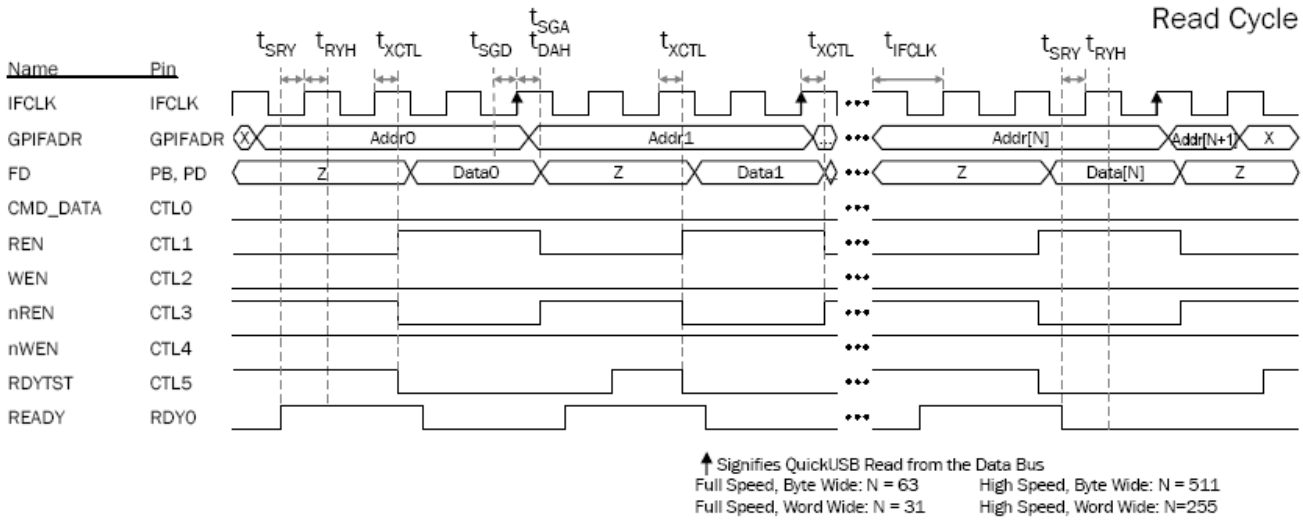


Figure 9: Full Handshaking I/O Model Waveform Diagram

In order to use one of the waveform models in the QuickUSB hardware, the corresponding pre-compiled file must be loaded into the QuickUSB EEPROM. This was done using the QuickUSB Programmer executable included in with the QuickUSB product package. For the Simple I/O Model the quickusb-simple.v2.11.qusb file was loaded, and for the Full Handshake I/O Model the quickusb-fhs.v2.11rc9.qusb file was loaded. If the default file location settings are selected during installation of the QuickUSB software, these files are located in the directory C:\Program Files\Bitwise Systems\QuickUSB\QuickUsbProgrammer.

4.2.3.1.3.4 GPIF Clock Selection

As with setting the Peripheral Mode, there are two ways to implement an internally supplied GPIF clock. The SETTING_FIFO_CONFIG LSB bit 7 must be set to 1 to enable an internally generated signal; the QuickUSB Library can be used to do this by implementing PseudoCode1 with Address = SETTING_FIFO_CONFIG or Hex 03 and BitSetMask = Hex 0080. Once the internal signal is specified, the SETTING_FIFO_CONFIG LSB bit 6 can be set to 1 for an internal 48 MHz clock, or set to 0 for an internal 30 MHz clock. A 48 MHz clock is the default setting and this was not changed.

Additionally, an internal clock can be implemented by checking the IFCONFIG bit 7 checkbox on the QuickUSB Diagnostics tool Defaults Tab. The internal clock speed is set to 48 MHz or 30 MHz respectively by checking or unchecking the IFCONFIG bit 7 checkbox on the Defaults tab on the QuickUSB Diagnostics tool as shown in Figure 7 above.

4.2.3.1.4 Testing Plan and Results

The Peripheral Domain cannot be tested independently with the hardware available. The Cypress FX2 development board allows access directly to the onboard 8051 CPU which would likely allow this independent testing. For this project the Peripheral Domain was only tested as part of overall firmware testing.

4.2.3.1.5 System Integration

The Peripheral Domain is a subsystem of the Firmware level of the Cypress FX2 chip. The Peripheral Domain does not encapsulate any lower level systems that are within the scope of this project and it is considered to be at the lowest level of the oscilloscope system. It is in parallel with the USB Domain system, and both are encapsulated within the Cypress chip level hardware and firmware.

4.2.3.1.6 Conclusions

The Peripheral Domain system was configured for its purpose in the oscilloscope system using QuickUSB firmware and PC applications. A Cypress firmware platform implementation is possible, but was not completed past an initial design as part of this project. No testing at this level was done; all testing was done at the Firmware level of the Cypress chip.

4.2.3.1.7 Recommendations

As a result of this work the following recommendations could be made:

- The QuickUSB firmware implementation should be retained for the next phase of development
- The procedure for and implications of erasing the QuickUSB EEPROM to allow a Cypress platform implementation should be investigated

4.2.3.2 USB Domain

4.2.3.2.1 Overview

4.2.3.2.1.1 Introduction

The USB Domain firmware defines the functionality required to transfer data between the Cypress onboard FIFO storage and the host PC through the USB cable.

4.2.3.2.1.2 Objectives

- Implement firmware capable of transferring acquired data across the USB cable to the host PC

4.2.3.2.1.3 Theory

The following information is a partial summary of information contained in Chapter 1 of Cypress document number 001-13670, *EZ-USB Technical Reference Manual.* See Appendix B 4. This information details a very small portion of the USB 2.0 specification; it discusses only the features of the specification that are applicable within the scope of this project, and that are accessible to a developer. The majority of the functionality to transfer data across the USB cable is performed by the Cypress on chip Serial Interface Engine without assistance from a developer. For more information on the USB 2.0 specification please refer to the *Universal Serial Bus Specification Version 2.0* available at www.usb.org.

The USB system is a host mastered protocol meaning all USB peripherals act as slave devices and do not initiate data transfers but instead respond to requests from the host PC. These transfers are

logically implemented with a host either requesting or sending end points. End points are implemented as onboard storage FIFOs on the Cypress chip. There are four types of data transfer available: interrupt, bulk, isochronous, and control. The type of transfer used is defined by defining the type of endpoint used to implement it.

Bulk transfer is a bursty method of data transfer, transferring 512byte sized packets under USB High Speed. Bulk transfer employs error checking so packets are sent with guaranteed accuracy. Additionally, bulk transfer implements flow control through the use of a hand shaking protocol. Interrupt transfers are similar to bulk transfers except they have a maximum packet size of 1024 bytes. Additionally, interrupt transfers feature a polling interval parameter that ensures that the host polls the interrupt endpoints at specified intervals.

Isochronous transfers are time guaranteed but not content guaranteed, transferring packets of up 1024 bytes. Space is reserved in each USB transfer for isochronous packets as they are time critical. To facilitate the faster transfer, only a simplified error checking scheme is employed. These types of transfers are generally used for streaming data.

Similar to Isochronous transfers, control transfers have reserved space in all USB transactions. These transfers have a maximum packet size of 64bytes. Of all the transfer types, control transfers employ the highest level of error checking to guarantee content.

The Cypress FX2 dedicates two endpoints, Endpoint 0 and Endpoint 1, exclusively to control transfers. These endpoints are accessible by the Cypress firmware only and are not used for high speed data transfer through the chip. There are six other available endpoints, Endpoints 2 through Endpoint 8, that can be configured for various sized packets and various types of transfers; these large endpoints are used for high speed transfer through the Cypress chip. Each of the larger endpoints also has multiple buffering options. These endpoints are double buffered by default but can be triple or quad buffered also. Buffering allows the Peripheral Domain to be simultaneously processing one of the buffer copies of the endpoint, while another buffer copy is being transferred between the Cypress chip and the host.

The Cypress documentation describes their large data transfer endpoints as “Quantum Endpoints” due to their fast intra-chip transfers. As discussed in section 4.2.1.3 above, the on chip FIFO storage where end points are implemented is shared by both the Peripheral Domain and USB Domain. This means that both domains have access to the common ram that makes up the storage FIFOs. A transfer of an end point between domains does not involve any transfer of data within ram, only the allowed access is changed. This process of moving an end point from one domain to the other is referred to as committing or arming the endpoint. These intra-chip transfers can be configured either for automatic or manual operation; this is done by configuring the data transfer end point for either automatic or manual arming. In manual mode the 8051 CPU is directly in the data path and initiates all intra-chip transfers. In automatic mode, the 8051 CPU is removed from the data path. The differences between automatic and manual mode are shown in Figure 10 below.

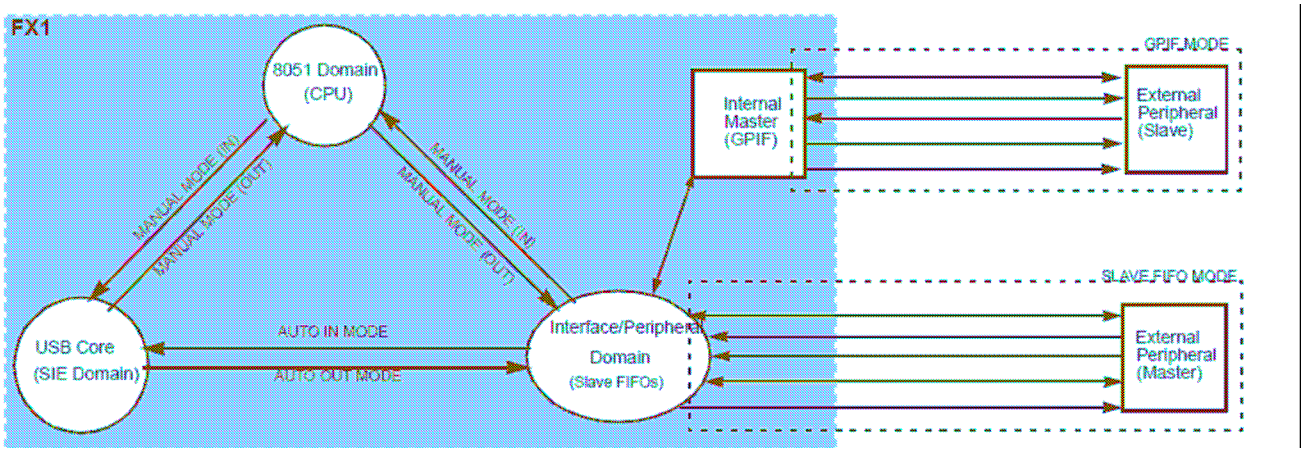


Figure 10: Automatic vs Manual Mode Diagram

4.2.3.2.2 Design Rationale

4.2.3.2.2.1 Type of Transfer

It was decided to use bulk transfers at this stage in development due to their error checking features. This meant that additional checking in the PC host software would not have to be done to determine if data was lost in the transfer. Isochronous transfers are likely faster than bulk transfers as they have less overhead, and deliver a larger packet. It was felt however that the guarantee of data integrity provided by bulk transfers outweighed any negative effect they would have on performance.

4.2.3.2.2.2 Endpoint Buffering

It was decided to use the maximum buffering possible on end points; this meant that a quad buffering scheme was chosen. Buffering is implemented to smooth out bursty data transfers and there were concerns over the scheduling of transfers by the PC host. At this point, there is very little control exerted over PC host side USB transfers aside from initiation; the processor level scheduling is implemented by the PC operating system. Quad buffering enabled the largest degree of protection against a mismatch in the hardware level data acquisition system speed and the host PC software.

4.2.3.2.2.3 Method of Intra-chip Transfer

It was decided to use automatic arming of endpoints to ensure the fastest intra-chip transfer possible. This was done to meet performance specifications. In addition, this promotes easier development as the 8051 CPU merely configures intra-chip transfers but no code is needed to implement them.

4.2.3.2.3 Design Implementation

As with design implementation in the Peripheral Domain (see section 4.2.3.1.3 above), the QuickUSB platform provides two ways to configure endpoints: through a compiled code library

shipped with the product (See Section Firmware to Software Interface)and through an executable, QuickUSB Diagnostics; both products are shipped with the QuickUSB package.

It was decided to implement bulk transfer and quad buffering for data acquisition incoming data transfers by configuring Endpoint 6; Endpoint 6 is not unique, and one of the other high speed endpoints could have been used instead. In order to use Endpoint 6 in this way, it would have to be activated and set as an IN endpoint. Note: the direction of endpoints is always given with respect to the PC host; for an IN endpoint, data travels from an external peripheral to the PC.

In order to activate Endpoint 6 the QuickUSB setting SETTING_EP26CONFIG, located at QuickUSB setting address 1 must be modified. Specifically bit 15 of the Least Significant Byte must be set to 1; it should be noted that although the QuickUSB documentation refers to the Endpoint 6 portion of the SETTING_EP26CONFIG address as the Least Significant Byte, the entire address is 32 bits long; the Endpoint 6 portion is the Least Significant two byte word, and any software that manipulates the SETTING_EP26CONFIG address must specify a 32 bit integer to hold its value.

To set activate Endpoint 6 Pseudocode 1 could be implemented with Address = SETTING_EP26CONFIG or Hex 01, BitSetMask = Hex 0000 8000. Additionally, the endpoint can be activated by bus can be implemented by checking the EP6CFG bit 7 on the QuickUSB Diagnostics tool Defaults Tab as shown in Figure 7 above.

In order to specify Endpoint 6 as an IN endpoint, the QuickUSB setting bit 14 of the Least Significant Byte of SETTING_EP26CONFIG must be set to 1. To do this Pseudocode 1 could be implemented with Address = SETTING_EP26CONFIG or Hex 01, BitSetMask = Hex 0000 4000. Additionally, the endpoint can be activated by bus can be implemented by checking the EP6CFG bit 6 on the QuickUSB Diagnostics tool Defaults Tab as shown in Figure 7 above.

4.2.3.2.3.1 Type of Transfer

In order to select Bulk transfer the QuickUSB setting SETTING_EP26CONFIG, located at QuickUSB setting address 1 must be modified. Specifically bits 13 and 12 of the Least Significant Byte must be set to 1 and 0 respectively.

To set Bulk transfer Pseudocode 1 could be implemented with Address = SETTING_EP26CONFIG or Hex 01, BitSetMask = Hex 0000 2000, and BitClearMask = Hex FFFF EFFF. Additionally, Bulk Transfer can be implemented by checking and unchecking the EP6CFG bits 5 and 4 respectively on the QuickUSB Diagnostics tool Defaults Tab as shown in Figure 7 above.

4.2.3.2.3.2 Endpoint Buffering

In order to select Quad buffering the QuickUSB setting SETTING_EP26CONFIG, located at QuickUSB setting address 1 must be modified. Specifically bits 9 and 8 of the Least Significant Byte must be set 0.

To select Quad buffering Pseudocode 1 could be implemented with Address = SETTING_EP26CONFIG or Hex 01, BitClearMask = Hex FFFF FBFF. Additionally, Quad

buffering can be implemented by unchecking the EP6CFG bits 1 and 0 on the QuickUSB Diagnostics tool Defaults Tab as shown in Figure 7 above.

4.2.3.2.3.3 Method of Intra-chip Transfer

The QuickUSB implementation does not expose the register required to specify either manual or automatic inter-domain transfer; it is assumed that a combination of modes is used depending on the type of transfer requested.

For a Cypress platform implementation, the method of intra-chip transfers would have to be specified in firmware code.

4.2.3.2.4 Testing Plan and Results

The project specific configuration of the USB Domain was not able to be tested independently with the hardware available. The manufacturer encapsulated USB functionality however, was tested using the QuickUSB Programmer executable. The QuickUSB board was plugged into a PC, and a waveform was downloaded to the QuickUSB EEPROM. Since EEPROM programming is done through the USB connection, the success of this step confirmed that the USB functionality was working properly. In addition, by using both the program feature, and the verify feature of the programming application, data transfer in both directions was tested. Additional testing of the USB Domain was completed as part of overall firmware testing.

4.2.3.2.5 System Integration

The USB Domain is a subsystem of the Firmware level of the Cypress FX2 chip. The USB domain does not encapsulate any lower level systems that are within the scope of this project and it is considered to be at the lowest level of the oscilloscope system. It is in parallel with the Peripheral Domain system, and both are encapsulated within the chip level hardware and firmware.

4.2.3.2.6 Conclusions

The USB Domain functionality was implemented using pre-compiled software supplied with the QuickUSB board. While a Cypress platform implementation was possible, this was not done. The USB functionality was partially tested and it was confirmed that data transfer through the USB cable, between the PC host and Cypress chip functioned properly.

4.2.3.2.7 Recommendations

Based on the results of this work, the following USB Domain recommendations can be made:

- The QuickUSB firmware implementation should be retained during the next development phase
- Benchmarking should be done with a functioning test system to determine the data transfer speed and reliability differences between bulk and isochronous endpoint

configurations

4.2.4 Testing Plan and Results Summary of Firmware

Throughout the project, bottom up testing was implemented wherever possible. At the firmware subsystem level, part of the USB functionality was tested independently and served as a known good component; none of the Peripheral Domain functionality was tested independently. Since functionality at the firmware level could not be tested independently, an additional layer of integration that allows communication between the PC host and the USB device was required. The QuickUSB Diagnostics utility program provided a known good software bridge between the PC host and the USB device; this diagnostics program was used to conduct firmware testing.

The objective of Firmware testing was to verify acquisition of varying data placed on the Cypress eight bit data bus. A testing plan was developed to verify functionality in a step wise fashion, minimizing the variables in each step. The plan was outlined as follows:

31. Load the QuickUSB EEPROM with the quickusb_simpleio.qusb file
2. Set default settings as discussed in Peripheral Domain and USB Domain sections
3. Read in 512 bytes of data with no data on data bus
4. Read in 512 bytes of data with constant data on data bus
5. Read in 2 bytes of data with constant data on data bus
6. Load the QuickUSB EEPROM with the quickusb_fhs.qusb file
7. Verify correct default settings and change if required
8. Read in 2 bytes of static data with manual handshaking
9. Read in 512 bytes of varying data with automated handshaking

This testing was done using a bread board system, a custom data generation system (See Appendix D), and the QuickUSB development platform with USB implementation and adapter board. The schematic for the testing setup is shown in Figure 11 below. A picture of the testing setup is shown in Figure 12 below.

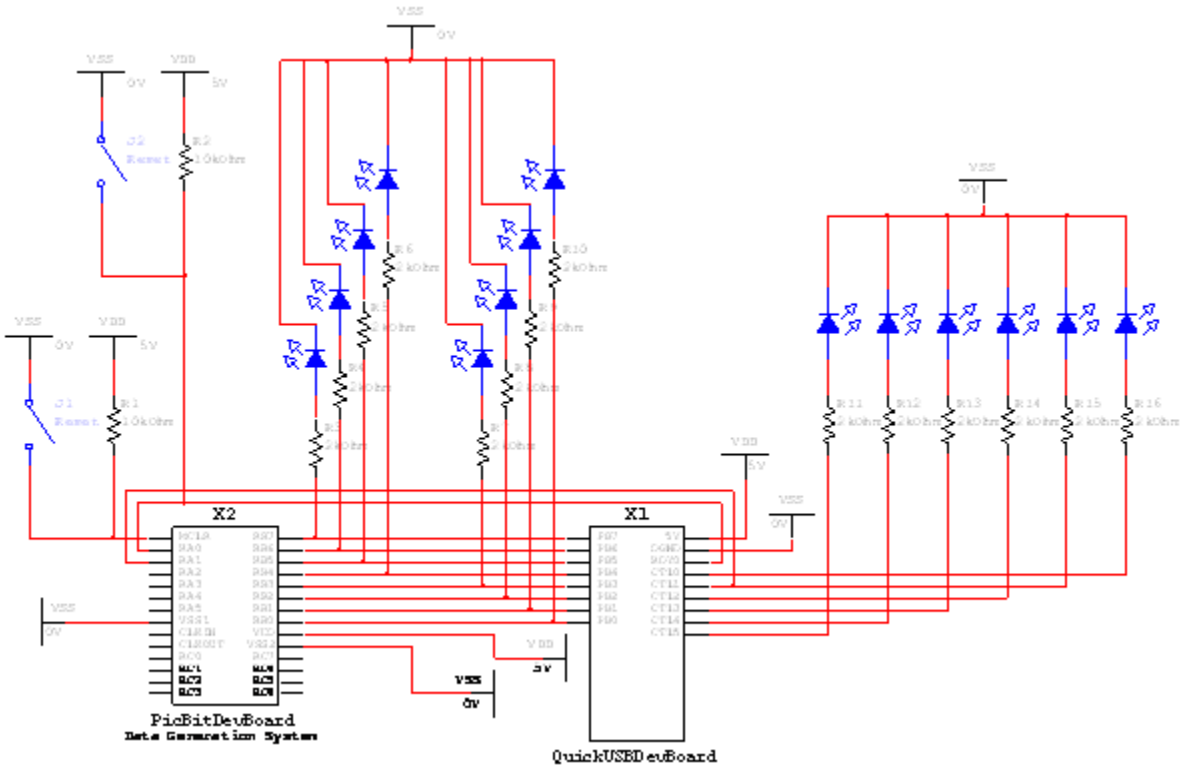


Figure 11: QuickUSB Testing Set Up Schematic

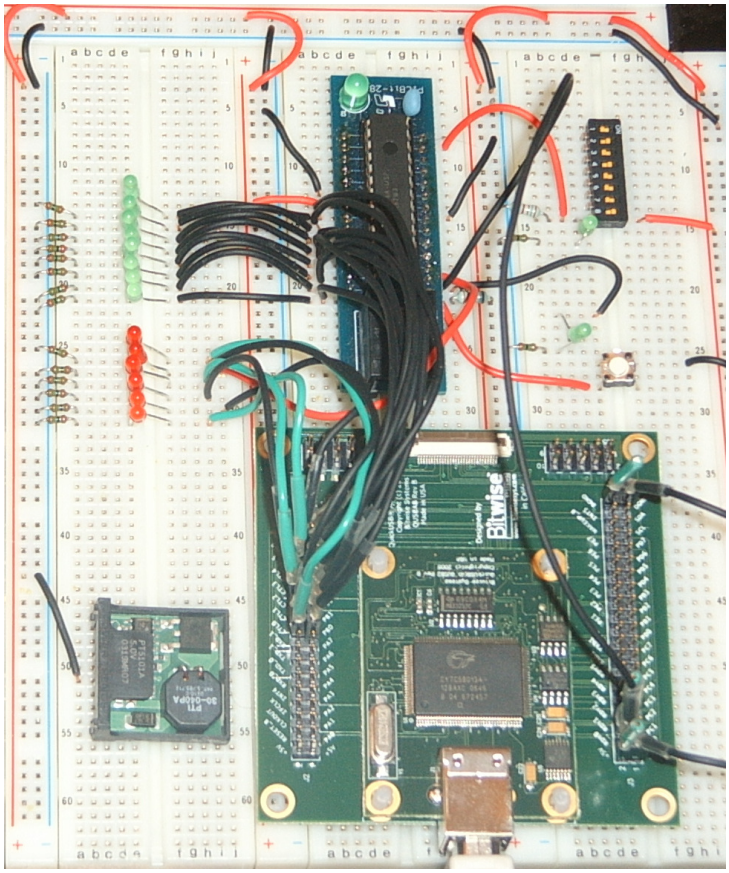


Figure 12: QuickUSB Testing Set Up Picture

Steps 1 & 2:
The Simple I/O Model was used initially to ensure that no handshaking was required, and that the device would begin reading data from the bus immediately; this model was loaded using the QuickUSB Programmer utility included with the QuickUSB board. The default settings to implement GPIF mode data acquisition as discussed in the Peripheral Domain and USB Domain were entered in the Defaults tab of the QuickUSB Diagnostics utility. These settings are shown in Figure 13 below. The QuickUSB board power was then cycled to ensure the default settings were loaded.

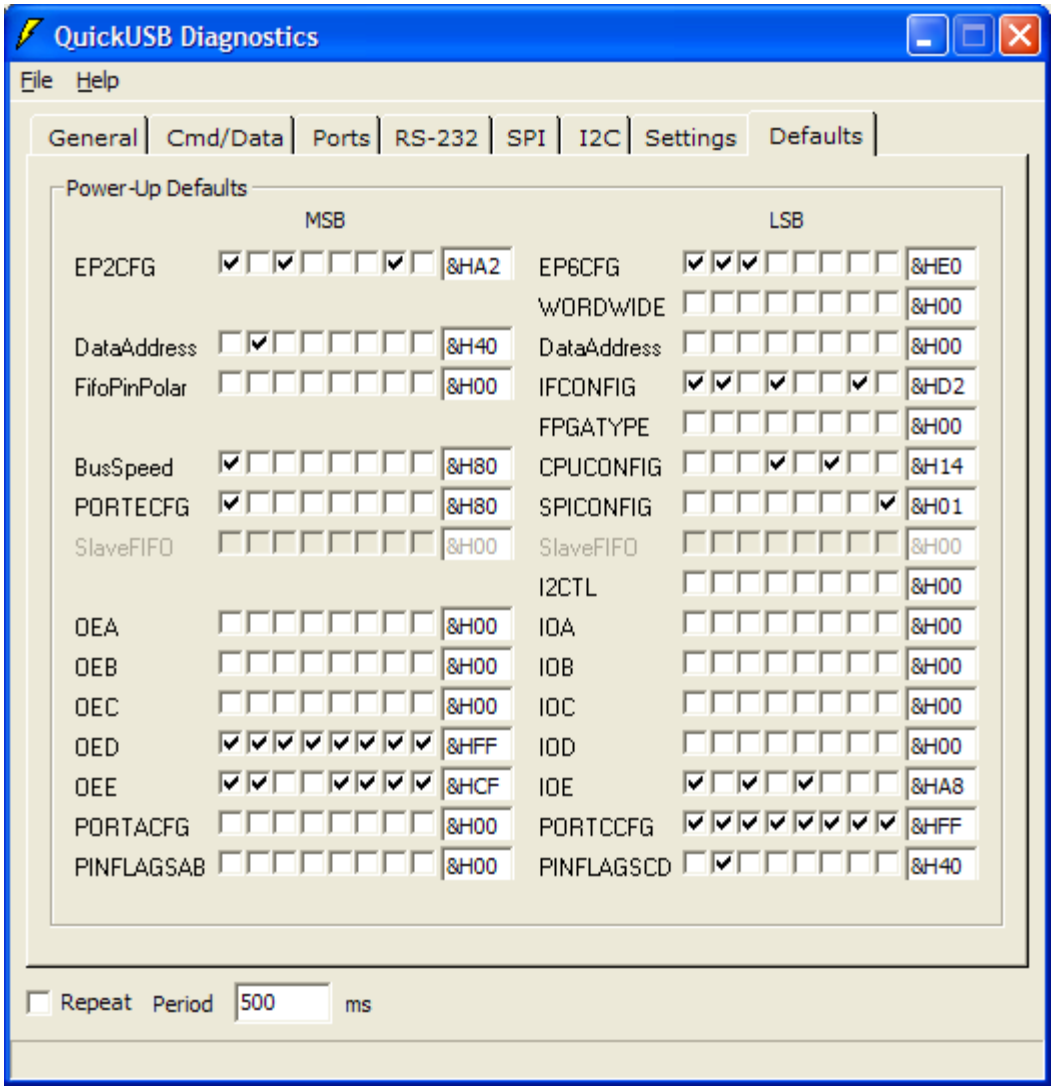


Figure 13: Default Settings Used for Firmware Testing

It was noted that although a setting is present in the QuickUSB Programmer utility to preserve default settings, the default settings were altered after programming was finished. To ensure consistent settings throughout all tests, a screen shot of the desired default settings was taken and used as reference during subsequent testing.

Step 3
A 512 byte read was conducted using the QuickUSB Diagnostics utility. The number of bytes read, 512, is the default setting for any data transfer using the diagnostics utility, and is the number of bytes in a High Speed USB packet. The transfer was initiated by creating an empty .bin file on the PC host computer, directing the diagnostics utility to the file through the browse button on the Cmd/Data tab, and pressing the Read button. The Cmd/Data tab is shown in Figure 14 below. At this point, there was no external device connected to the Port B pins of the QuickUSB board.

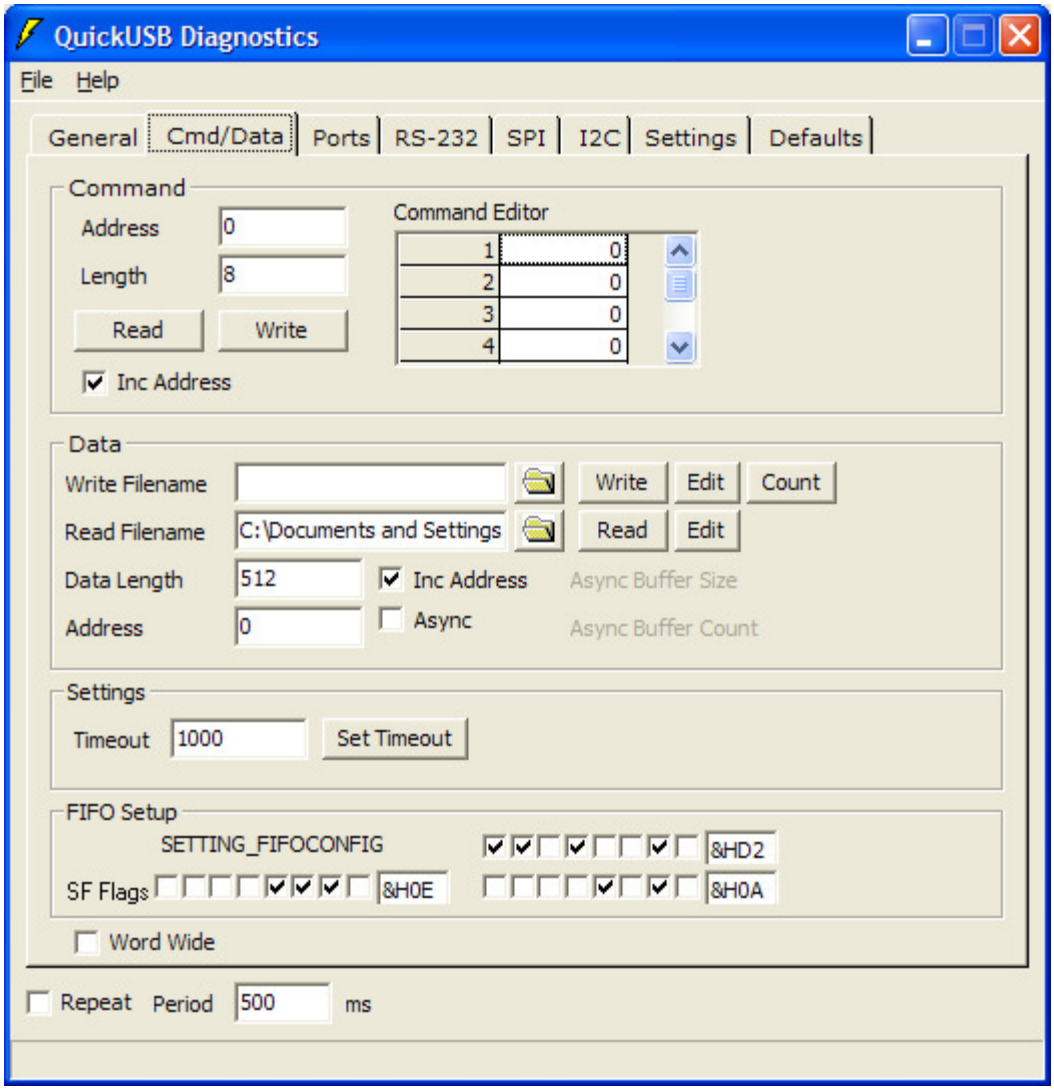


Figure 14: QuickUSB Diagnostics Program Cmd/Data Tab

Opening the empty test .bin file showed that data had been placed in the file, a series of ý characters; it is assumed that this is the value read for an empty bus. No errors were reported during this test.

Step 4
An eight bit number 00010001 was placed on the data bus by the PicBit data generation system. A read of 512 bytes was conducted as in Step 3. Opening the test .bin file confirmed that the static data had been read in.

Step 5
With the static data still on the data bus, 2 bytes of data were read in using the diagnostics utility. This was done in anticipation of subsequent steps involving a manually generated handshaking signal. It would have been undesirable to generate the handshake signal for a 512 byte read. The contents of the test file confirmed that it was possible to read only 2 bytes.

Step 6 & 7
The handshaking protocol was required to ensure that a synchronizing clock signal was not required for the data generation device. All synchronization clock signals are required to be in excess of 5 MHz and so were not suitable for a bread board testing system. In a final product, the data would be generated by an external A/D chip that would take samples based on the clock signal. For testing, the data generation device placed samples on the bus based on the handshaking protocol. The handshaking file, quickusb_fhs.qusb, was loaded onto the QuickUSB EEPROM, the default settings were modified to match those of Step 2, and the QuickUSB board power was cycled to ensure the default settings were loaded.

Step 8
The QuickUSB RDY0 line was connected to a manual pushbutton separate from the data generation device. The manual pushbutton was used to implement the slave device handshaking signal, and 2 bytes of static data was read in. The data generating device was set to output a static value to the bus; it did not vary its data based on the QuickUSB control signal. On the first attempt, the read operation did not complete; the contents of the test .bin file confirmed the error. It was believed that there had been a timeout error, and the timeout value was reset from its default value of 1000 ms to 10000 ms using the Set Timeout button on the Cmd/Data tab of the diagnostics utility. The second read attempt with the modified timeout was successful.

Step 9
The QuickUSB RDYO line was connected to the data generating device. The data generating device was set to vary its output data based on the QuickUSB CTL1 signal. A 512 byte read was initiated from the Cmd/Data tab of the diagnostics utility. As in Step 8, the initial attempt generated an error; resetting the timeout value as before corrected the problem and the contents of the test .bin file confirmed that the varying data had been read. The data generating device was programmed to cycle through a 256 element array from a sine value table, and the test .bin file contained two copies of this table as expected.

Once Step 9 was completed, the firmware system was considered a known good component for further testing of higher level software components; the modified timeout of value of 10000 ms was accepted as a requirement and integrated into the design of the firmware.

4.2.5 System Integration Summary of Firmware

The firmware encapsulates the functionality of the underlying subsystems present on the Cypress FX2 chip. In conjunction with the Cypress silicon, the firmware is able to implement data transfer between the oscilloscope data acquisition hardware and the host PC data acquisition software. The firmware layer is encapsulated by the Firmware to Software Interface component of the host PC oscilloscope application. The system integration of the subsystems into the Firmware level is shown in Figure 15 below.

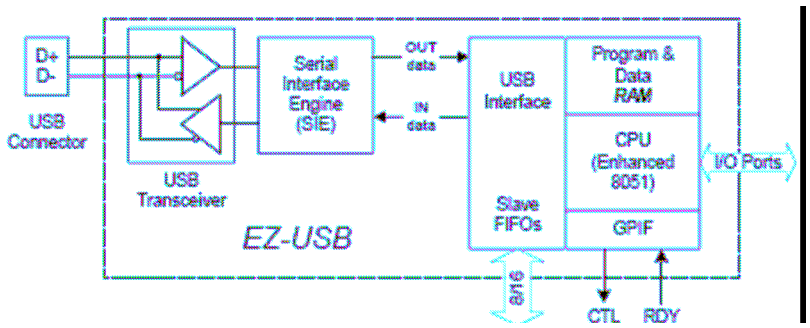


Figure 15: Firmware System Integration Diagram

The firmware system is integrated into a software project through the use of a manufacturer supplied code library. These libraries are discussed in section 4.3.3.1.1.3 below.

4.2.6 Conclusions Summary of Firmware

The firmware system encapsulating data transfer through the Cypress FX2 chip was implemented with pre-compiled QuickUSB software. The system was tested and functioned as anticipated.

4.2.7 Recommendations Summary of Firmware

As mentioned in the Firmware subsystems sections:

- The QuickUSB firmware should be retained for the next phase of development, but replacing it with a Cypress platform should be investigated.
- The strategy of using Bulk transfer methods should be evaluated further for performance, and the option of using Isochronous transfers investigated.

4.3 Software

4.3.1 Overview

4.3.1.1 Introduction

The software portion of the project covers everything in the PC, from the USB port right up to the screen display. All of the previous sections were focused on getting data samples into the USB as fast as possible, and the software is responsible for processing that data and displaying it to the user.

4.3.1.2 Objectives

- To isolate USB platform dependence to low level module
- To develop with an easy to use inexpensive IDE for the chosen software language
- To create maintainable code through object oriented design techniques

4.3.1.3 Theory

All of the software design in this project was approached using the top-down design principles stressed in OOP design. OOP is a software design method of organizing code into logical blocks, or classes. Each of these object classes contains all of the properties required to completely describe the object, as well as methods required to describe its behaviour. This method inherently stresses modularity and encapsulation, which are key components for code reuse and maintainability. OOP design also provides an implementation for polymorphism, or object inheritance, allowing for minimization of code duplication between classes of similar types.

Class libraries are a major component of OOP. They consist of a collection of related class descriptions placed into a separate software project to be accessed from the main program. This allows for common classes to be reused many times across different applications without duplicating them and keeping the program code base as small as possible.

Software data processing is often done as a modular unit, taking in a data stream, and outputting a processed data stream. This 'black box' solution allows for the insertion of a data processing code block into the software data stream without interrupting any of the logical program flow. The data processing block will often involve the creation of an intermediate buffer to separate incoming and outgoing data, minimizing errors due to data processing time.

4.3.2 Design Rationale

The first, and possibly most important decision made for the software component of the project was the selection of programming language. Possible choices included C++, C#, and Visual Basic. The pros and cons of each were weighed carefully.

C++ has more low level access than the other two choices, which used to mean faster program execution at run-time, but with the advancement of compilers, that advantage has been minimized. Unfortunately, C++ has some substantial drawbacks to the developer. There is no automatic garbage collection or memory management. Aside from the code writing difficulties presented by C++, the MFC graphical form designer leaves much to be desired. The largest advantage of using C++ is that many development tools are written in C++, minimizing development integration difficulty.

Between C# and Visual Basic, the choice mainly came down to programmer's preference. They are similar in that they both have automatic garbage collection and memory management. They also share the same form designer interface, which surpasses the C++ form designer by far. The ability to create a wrapper around any C++ development tools for porting into C# or Visual Studio meant that either of the latter languages would satisfy the ease of development objective. C# was chosen simply because of previous experience with the language, minimizing the learning curve for development on this project.

For the software development platform, Visual C# Express Edition was chosen. It satisfies the cost design objective as it is available for free from the Microsoft web site and has been included on the software CD with this report. The choice of Visual C# was a result of the seamless IDE developed by Microsoft, and all of the available tools, satisfying the ease of development design objective.

All data processing aside from sample value scaling was considered beyond the scope of this project. Initially the possibility of including a spectrum analyzer was discussed, but due to time considerations was not included.

Each level of software abstraction was implemented as a separate class library project. This was done to facilitate replacement of modules at any level of abstraction either to switch to a new USB platform or to implement a different type of scope such as a spectrum or network analyzer, satisfying the ease of development and accommodation of future considerations objectives.

It should be noted that implementation outlines in this section of the report describe a high level of abstraction to maintain clarity of design principles. For a more detailed description of the code implementation please see the appropriate code class appendix referenced in each section.

4.3.3 Design Implementation

4.3.3.1 Firmware to Software Interface

4.3.3.1.1 Overview

4.3.3.1.1.1 Introduction

The Firmware to Software Interface is a manufacturer supplied code library combined with a high level wrapper structure; this combination exposes high level functions to host PC software that are capable of interfacing with on chip firmware code, bridging the two systems.

4.3.3.1.1.2 Objectives

- Expose functionality to a host PC buffer system to open a USB device
- Expose functionality to a host PC buffer system to manipulate settings on a USB device
- Expose functionality to a host PC buffer system to read data from a USB device
- Expose functionality to a host PC buffer system to close a USB device

4.3.3.1.1.3 Theory

4.3.3.1.1.3.1 QuickUSB

Bitwise Systems provides developers of a QuickUSB application with a Visual Studio 6 compliant code library, QuickUSB.dll. See Appendix C 3.3 This library is precompiled so no source code is available, but the library can be added to a Visual Studio project. The QuickUSB library encapsulates the majority of the functionality required for a PC host to communicate with the firmware on a Cypress chip, but this functionality is defined at a lower level than would be required for a specific application. Common programming practice would require the development of a wrapper class to encapsulate the Cypress functions into a representation that more closely matched the needs of the PC host application. In addition, since the code library is Visual Studio 6 compliant, but not Visual Studio .Net compliant, this project requires that the wrapper class contain

special functionality to allow communication between the newer development environment and the legacy developed code.

4.3.3.1.1.3.2 Cypress .NET dll

Cypress provides developers with a Visual Studio .Net 2.0 compliant code library, CyUSB.dll. See Appendix C 3.2. This library is precompiled so no source code is available, but the library can be added to a Visual Studio project. The Cypress library encapsulates the majority of the functionality required for a PC host to communicate with the firmware on a Cypress chip, but this functionality is defined at a lower level than would be required for a specific application. Common programming practice would require the development of a wrapper class to encapsulate the Cypress functions into a representation that more closely matched the needs of the PC host application.

4.3.3.1.2 Design rationale

It was decided to utilize the QuickUSB supplied software library to interface with the USB chip level firmware. Since the QuickUSB supplied firmware was being utilized, use of the QuickUSB software library mandatory; in addition, the combination of these two products allowed rapid development even if sacrificing some design flexibility.

It was decided that the interface provided to the next layer of software on the host PC would be as simple as possible, and would expose only device open, read and close methods. This simplified interface promoted faster design, and was consistent with the software principle of encapsulation. Although a fully functioning oscilloscope would require □i-directional data transfer, in keeping with a simplified design, no write method was designed. Additionally, there are numerous functions available through the QuickUSB library that were beyond the scope of this project and so were not implemented.

4.3.3.1.3 Design implementation

In order to provide a simple interface for client software, the QuickUSB software library was encapsulated in a C# class, QuickUSB.cs. See Appendix C2.1.7.

As the QuickUSB supplied library could not be directly incorporated into a Visual Studio .Net 2.0 project, Visual Studio interoperability features were used. To allow utilization of the QuickUSB library functions each required an interoperability declaration in the encapsulating class. These statements can be seen in the code prefaced with the statement DLLImport.

During testing it was noticed that a call to the QuickUSB device open function would return an error unless a call to the QuickUSB get devices function had been previously called. The QuickUSB get device function was encapsulated into the OpenDevice function exposed to client software. Additionally, firmware testing had revealed the need to reset the QuickUSB device's timeout value for data transfer from the default value of 1000 ms to 10000 ms. This functionality was also encapsulated into the OpenDevice function exposed to client software allowing the client software to initiate a Read command as soon as a device was opened.

As can be seen the the QuickUSB.cs class code, both the Read and CloseDevice functions are direct implementations of the underlying QuickUSB library functions; no additional

functionality is encapsulated in these class functions.

4.3.3.1.4 Testing Plan and Results

The hardware was set up as described in 4.2.3.1.4 above, with the custom data generator supplying dynamic data to the Cypress chip through the Full Handshaking I/O Model. The QuickUSB.cs class was instantiated on a testing application form, and the OpenDevice, Read, and CloseDevice methods were called in succession. The Read method was called with a length parameter of 512 bytes. A review of the Read destination byte array showed that the transfer of data was successful.

4.3.3.1.5 System Integration

The Firmware to Software interface class encapsulates the lower firmware subsystem and allows it to be used by higher level PC host software applications. This relationship is shown in Figure 16 below.

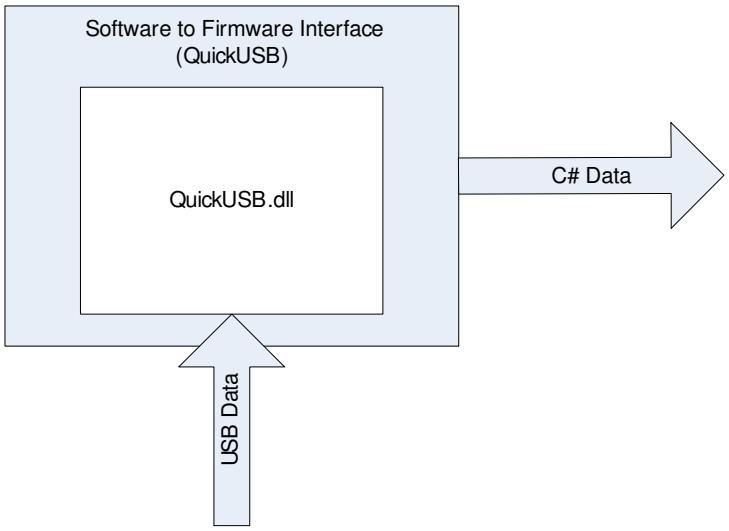


Figure 16: Software to Firmware Interface System Integration Diagram

4.3.3.1.6 Conclusions

The manufacturer supplied QuickUSB.dll library was integrated into the Visual Studio.Net 2.0 environment through the use of an encapsulating class. This allowed higher level software to initiate data acquisition transfers from the host PC, and data on the Cypress chip peripheral data bus could be read into host PC software. The software was tested and performed as expected.

4.3.3.1.7 Recommendations

- Additional methods available in the QuickUSB.dll library should be exposed to higher level software as required to implement additional scope functionality

4.3.3.2 Data Acquisition System

4.3.3.2.1 Overview

4.3.3.2.1.1 Introduction

The data acquisition system consists of the firmware interface as well as an event generator for communication to the controlling application. Its class definition can be found in C2.1.3: CyAPIInterface.cs. The data acquisition system is responsible for continuously polling the interface to obtain data and condition it in preparation for insertion into a buffer, then raising an event so that the receiving system knows to load the data.

4.3.3.2.1.2 Objectives

- To utilize and provide a standardized wrapper interface
- To supply uninterrupted USB transfer operation

4.3.3.2.1.3 Theory

Software interrupt service routines are special sections of code that is executed when a certain trigger is fired. Interrupt handlers halt the current process, execute the interrupt service routine, and then resume regular program execution. Events perform in a similar manner to interrupt service routines with one exception. They do not interrupt program flow when their trigger is fired. Instead the event is not executed until after the regular program flow is completed and the application returns to an idle state, or an explicit DoEvents() function call is made.

4.3.3.2.2 Design & Design rationale

Data acquisition is a simple procedure at its highest level of abstraction. It is a continuous loop of opening the USB device using the firmware interface, reading a packet of data, conditioning data, and generating an event. This is an application level procedure, and it acts a controller whose sole purpose is to get USB data into the PC as fast as possible. The data acquisition system is also the highest level of software possible to be dependent upon the USB platform implemented.

To enable the fastest data transfer rates possible, the data acquisition system does not do any data processing. It essentially converts the polling of the USB bus into event generation for any host programs that may be using the data acquisition system. In this way the performances of both the data acquisition system and the host application are improved, by cutting out the polling overhead from the host application, and by allowing the data acquisition system to constantly poll the bus and be ready for a response.

During the scope of this project, the data acquisition system was given simple operation with the firmware interface. The functions were mainly limited to opening, closing, and reading from a device. Partially implemented in the data acquisition system is the ability to send commands to the device. Sending commands was deemed a complexity beyond the scope of this project.

4.3.3.2.3 Testing Plan and Results

To test the data acquisition system, a simulated firmware interface system was developed, which would generate a known signal and return it to the data acquisition system. The data acquisition system then displayed the signal values to the command window from its event

handler to ensure correct operation.

4.3.3.2.4 System Integration

The data acquisition system was a higher level of abstraction for the firmware interface wrapper, and so its integration consisted of adding the wrapper class as shown in Figure 17: Data Acquisition System Integration, and implementing the algorithm described above.

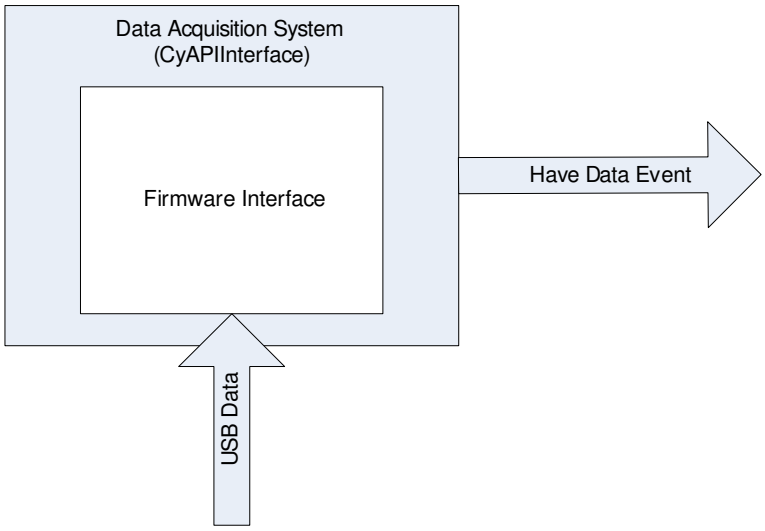


Figure 17: Data Acquisition System Integration

4.3.3.2.5 Conclusions

As one of the most concentrated performance efforts in this project, the data acquisition system effectively performs its required functions. Any changes of USB platform may require changes to the data acquisition system depending on whether an interface wrapper can be written to offer correct functionality with the same function calls.

4.3.3.2.6 Recommendations

For further development toward a future design, changes to the data acquisition system are contingent upon changes in the firmware interface system and whether upgrades are necessary to ensure correct operation.

4.3.3.3 Buffer System

4.3.3.3.1 Overview

4.3.3.3.1.1 Introduction

The buffer system in this project consists of a single data buffer. It is a custom first-in-first-out queue in between the data acquisition system and the GUI. Its class definition can be found in C2.1.2: CyAPIBuffer.cs. It is responsible for buffering the data packets coming in from the USB interface and smoothing out the delays caused by the overhead of the USB transfer protocol.

4.3.3.3.1.2 Objectives

- To be continually available from both ends to receive and supply data
- To have sufficient performance to support maximum throughput of USB transfer

4.3.3.3.1.3 Theory

Multithreading is the term used in computing to describe multiple processes or applications operating simultaneously on a single system. This enables an application to be split into several processes to ensure that any speed critical tasks can be carried out efficiently while allowing any other processing to be carried out when there is time. In all but the most high-end computers at the time of this report, there are no true multithreading capabilities; instead, computers simulate multithreading by allowing the operating system to execute several instructions from one process, then several from the next, and so on, until program execution is returned to the first process in line again.

In USB data transfer, the data is sent in packets, or data chunks. These packets are discussed in the firmware integration system section. Due to the USB transfer protocol overhead associated with each packet transfer, the incoming data needs to be buffered to smooth out delays between packets.

4.3.3.3.2 Design & Design rationale

To ensure correct operation, it was required that the data acquisition system be able to run continuously uninterrupted, and that it and the GUI both have simultaneous access to the data buffer system. To accommodate this, it was determined that the software needed to be a multithreaded application. One thread handles the data acquisition system and feeds data into the buffer, and the other thread manages the GUI and retrieves data from the buffer.

To handle communication across threads, an event is generated by the data acquisition system when it has data to load into the buffer. This event is used to add the packet of data received to the buffer and then immediately control is resumed to the data acquisition system for polling the USB bus for more data.

4.3.3.3.3 Testing Plan and Results

Testing of the buffer system was done in three stages during development. As soon as the multithreading capability was implemented, each thread was set to run in a separate loop, each outputting values to the console window. In this way it was verified that both threads were running simultaneously and could both access a shared workspace.

Once the data received event handling was set up, the buffer system was again extensively tested for correct operation. This testing was done by putting a break point in the data received event handler. When the program paused on the break point in the event handler, it was verified that not only was the event handler being executed correctly, but the other thread continued to operate while the data acquisition thread was paused.

The final stage of testing involved a test application written to run the buffer class. The test application supplied a known signal to the data acquisition system, and it was confirmed that the signal was received into the buffer, and was simultaneously available for retrieval and output to the console window.

4.3.3.3.4 System Integration

The buffer system integration is shown by Figure 18: Buffer System Integration. The actual buffer data and the data acquisition class are included as properties of the interface buffer class.

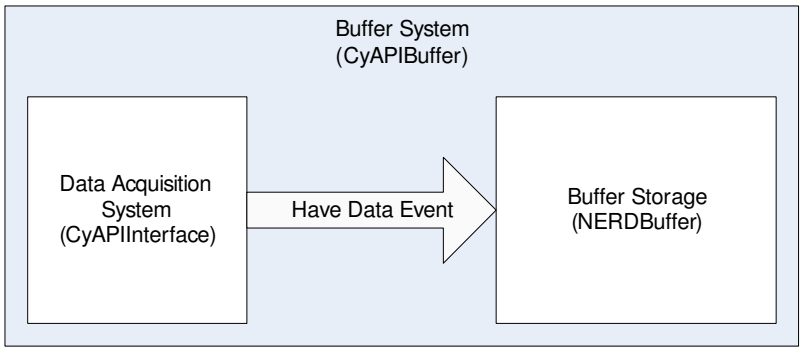


Figure 18: Buffer System Integration

4.3.3.3.5 Conclusions

The buffer system works well to create a solid repository of data for the GUI to draw from. It is very efficient at moving data from the USB domain into the buffer quickly so that program control can return to firmware interface.

4.3.3.3.6 Recommendations

There is a minor performance consideration with the interface buffer class in its inclusion of the generic buffer class as a property. This adds an extra call stack reference during the call of each buffer function, and could be avoided by modifying the interface buffer to inherit from the generic buffer instead. This was not done in this project due to time considerations.

It should also be noted here that if any data processing is to be done in the future, it should be done by stacking another buffer on top of this one, with a data processing module between them, this would ensure that data integrity is retained at all times and processing delays do not affect program execution.

4.3.3.4 GUI

4.3.3.4.1 Overview

4.3.3.4.1.1 Introduction

The GUI is essentially a collection of controls organized onto a Windows form. Its class definition can be found in C2.1.1: ScopeForm.cs. It is responsible for receiving input from the user and displaying results back on the screen.

4.3.3.4.1.2 Objectives

- To be within a fully resizable viewing area
- To be constructed in a modular fashion
- To be written with a code base that is scalable and maintainable

4.3.3.4.1.3 Theory

GUI design is a classification of software design that has seen much growth in recent years. One

of the most common forms of GUI is the standard Microsoft Windows form. GUI design is thought of by many software designers as an unnecessary inconvenience, merely making the software package appear more visually appealing. Although the visual appeal is a pleasant side effect, GUI design plays a much more important role in limiting user input to specific ranges or sets of values, minimizing the need for error checking on input values.

Windows forms not only encourage good GUI design, but through their use of modular form controls they also encourage the encapsulation techniques stressed by OOP. This makes form based programs much easier to develop, as well as cleaner and easier to maintain.

4.3.3.4.2 Design rationale of GUI

Since the project objective was to produce a USB based oscilloscope, the GUI was designed to reasonably imitate the appearance of a scope. A large scope display occupies most of the left side of the form, while a few adjustment knobs and buttons sit on the right.

One important aspect of the GUI is the requirement to handle resizing of the interface to support different resolutions of different computer screens. The form controls were anchored appropriately to enable graceful resizing of the application, moving the knobs on the right and resizing the graph control.

4.3.3.4.3 Design Implementation of GUI

4.3.3.4.3.1 Graph Control

4.3.3.4.3.1.1 Overview

4.3.3.4.3.1.1.1 Introduction

The graph control is a custom Windows form control class to be placed onto the program GUI. The control is used for graphing output to the PC screen. Its class definition can be found in C2.1.5: GraphBox.cs. It is responsible for showing the data in an easily recognizable format, as well as scaling the view window range and erasing previous plots when it plots over itself.

4.3.3.4.3.1.1.2 Objectives

- To operate as a generic control used for many scope types including logic and spectrum analyzers
- To implement high speed graphing to keep up with firmware interface
- To be encapsulated into independent class with simple program interface

4.3.3.4.3.1.1.3 Theory

In C#, all basic drawing functions are done using the System.Graphics class. This class draws on bitmap images created from either a stored image or a temporary screenshot of the form. The graph control mainly utilizes the line drawing functions to use linear interpolation between points, as well as to draw the grid lines on the graph. It should be noted that in computer graphics the y axis is positive pointing downwards and the origin is the top left corner of an image, which is opposite to standard y axis convention.

The graph control was designed using the Visual C# form control designer, which allows the control to be graphically designed on a piece of Windows form background. Other controls and

graphics are placed onto the background rectangle to form a composite custom control, with user defined code added to determine control behavior. In this way a control is kept modular and can be placed onto a form as a single unit.

Scope display ranges are adjustable, to zoom in or out on different portions of a signal. The x and y axis are separately controlled, so that for example in the case of an oscilloscope, voltage and time scales can be adjusted independently.

4.3.3.4.3.1.2 Design & Design rationale

Using the solution manager available in Visual C# Express, the graph control was created as a separate class library project. This ensured encapsulation of the class, and made its insertion into subsequent projects a simple matter, satisfying the objective of ease of development. The control was split up into four sections: x axis label, y axis label, graph area, and data display.

Axis labeling is done separately from graphing for performance reasons. With independent labeling, the labels only need to be redrawn when the view range is changed or the control is resized on the screen. There are several methods to set the labels and axis range, but the one used the most in this project sets the label string, axis range, and number of gridlines along that axis all at once, simplifying use. The x and y graph axis labels are docked on the bottom and left of the control respectively, so that they will resize with the control.

All of the main graphing methods interact with the graph area of the graph control. These are designed to be as simple and generic as possible. The main plot function simply takes two points with x and y coordinates in the same scale as the axis labels, to avoid any manual conversion. These two points are plotted using simple linear interpolation to increase performance as much as possible. The plot function will erase any previous plots covering the same x scale range and overwrite them with the new plot line.

A provision was provided within the graph control for the display of calculated information, such as voltages and periods. The data display was inserted as an item list control. The item list control is a standard Windows control designed to show collections of objects, making it easy to maintain an organized visual format while also remaining simple to access and modify each item.

As the graph control fills most of the application form area, it needs to resize with the form. Image scaling and value recalculations were all done within the sizeChanged() event. To facilitate resizing, the dock property was used extensively on all components of the graph control.

4.3.3.4.3.1.3 Testing Plan and Results

A small test application was written for graph control testing. The testing was done in several stages. First each of the label display and range functions was tested extensively for correct operation. Once the label ranges were verified to display correctly, the graph area was tested by manually generating samples equivalent to a 50Hz 4V sine wave. The graph plot functions were verified to display the correct signal values.

There was an issue with the plot performance of the graph control. Due to the way the draw events work, the application had to constantly invoke calls to the DoEvents() function to update the display. This incurred a large processing overhead, making it impossible for the graphic

system to keep up with the data input.

4.3.3.4.3.1.4 System Integration

The graph control system was integrated as shown in Figure 19: Graph Control Integration. It was packaged as a control to be integrated into a controlling form as a single module.

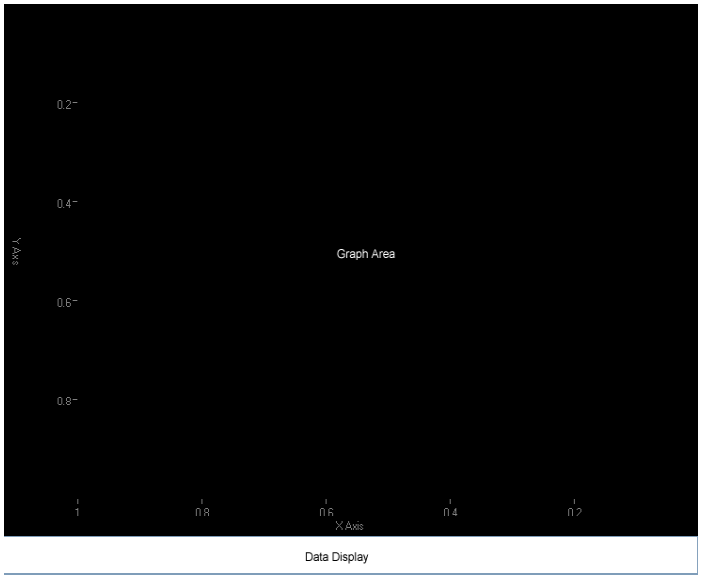


Figure 19: Graph Control Integration

4.3.3.4.3.1.5 Conclusions

The graph control satisfied the correct operation procedure, but suffered from severe performance issues. The performance of the graph control is likely to be the biggest performance bottleneck of the entire scope system.

4.3.3.4.3.1.6 Recommendations

In future developments on this project the performance of the graph control would need to be improved. This could be done by investigating into the draw events to find more options for updating the display. A more proficient solution would be to utilize a more powerful graphics engine, possibly building a Microsoft DirectX component. Improving the graphics performance would be one of the most significant enhancements for the overall system.

4.3.3.4.3.2 Knob Control

4.3.3.4.3.2.1 Overview

4.3.3.4.3.2.1.1 Introduction

Like the graph control, the knob control is a composite Windows form control. Its class definition can be found in C2.1.6: Knob.cs. Unlike the graph control, the knob control is an input to read value adjustments from the user.

4.3.3.4.3.2.1.2 Objectives

- To provide a familiar scope interface
- To implement easy to use functionality

- To present a visually appealing style
- To support logarithmic scaling on adjustments

4.3.3.4.3.2.1.3 Theory

In a Windows form GUI, mouse input is tracked through events. This means that during long computation processes, the mouse input is often not registered until an explicit call to DoEvents() is made. It also means that procedures reading the mouse should be as short as possible, to allow program flow to resume normal operation.

Knob controls utilize the graphics class in a much more involved and complex manner than the graph control. The main draw functions used by the knob class are the ellipse functions. There is also much more computation required during the know draw functions due to necessary calculation of ellipse parameters as well as trigonometry and possibly logarithmic functions used to calculate the value of the knob.

4.3.3.4.3.2.2 Design & Design rationale

The knob control consists of a Windows form picture box with enhanced event handling and display functionality. It is shown in Figure 20: Knob Control.

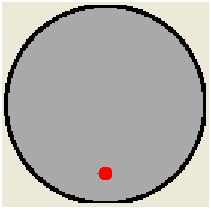


Figure 20: Knob Control

Due to performance considerations, the mouse event used to calculate the user input was the mouse move event, rather than the mouse down event. While using the mouse move event means that it would have to include an explicit check to see if the mouse button was down, it would also mean that the event handler did not have to remain running until the user released the mouse button, so that regular program flow interruption will be minimized.

All of the graphic functionality on the knob control was calculated setting the □i-dir of the control as the origin. Most positioning calculations were done from there using a polar coordinate system. This way the value of the knob can easily be determined by negating the angle to compensate for the negative y axis of computer graphic systems.

To support logarithmic as well as linear scaling on knob values, the stored value of the knob is simply a linear value between zero and one. This way the actual value can be calculated using either method upon retrieval, using one of two algorithms to convert the zero to one scale to a minimum to maximum value scale either linearly or logarithmically depending on the application requirement.

4.3.3.4.3.2.3 Testing Plan and Results

A simple test application was devised for the knob control, using a text box on a form to verify correct operation of the form control. Performance benchmarking was not considered for the knob control as it was assumed that it will not be adjusted continuously.

4.3.3.4.3.2.4 System Integration

As the knob control consists of a single picture box with added functionality, there was no integration step required to assemble the unit. Like the graph control, it was packaged as a control to be integrated into a controlling form.

4.3.3.4.3.2.5 Conclusions

All functionality of the knob control meets the objectives. It is a simple drop in control module easily placed onto any Windows form and should be acceptable for all levels of scope development. Another possible future addition to the knob class would include fine tuning, possibly through a right click menu on the control.

4.3.3.4.3.2.6 Recommendations

While the functionality of the knob control is ideal, the graphics remain simplistic and unrefined. For a finished product, a more detailed suitable background image should be used.

4.3.3.4.4 Testing Plan and Results Summary of GUI

Testing the GUI was done by manually generating a 50Hz 2V sine wave and displaying it on the screen. The view range was adjusted using the knob controls and it was verified that the graph display correctly adjusted accordingly. All functionality operated as expected, including the performance bottleneck introduced by the graph control. The draw events of the graph class had to be modified to enable adjustment of the knobs during program execution.

4.3.3.4.5 System Integration Summary of GUI

Integration of the GUI components mainly consisted of taking the custom controls explained above and placing them onto a Windows form as shown in Figure 21: GUI.

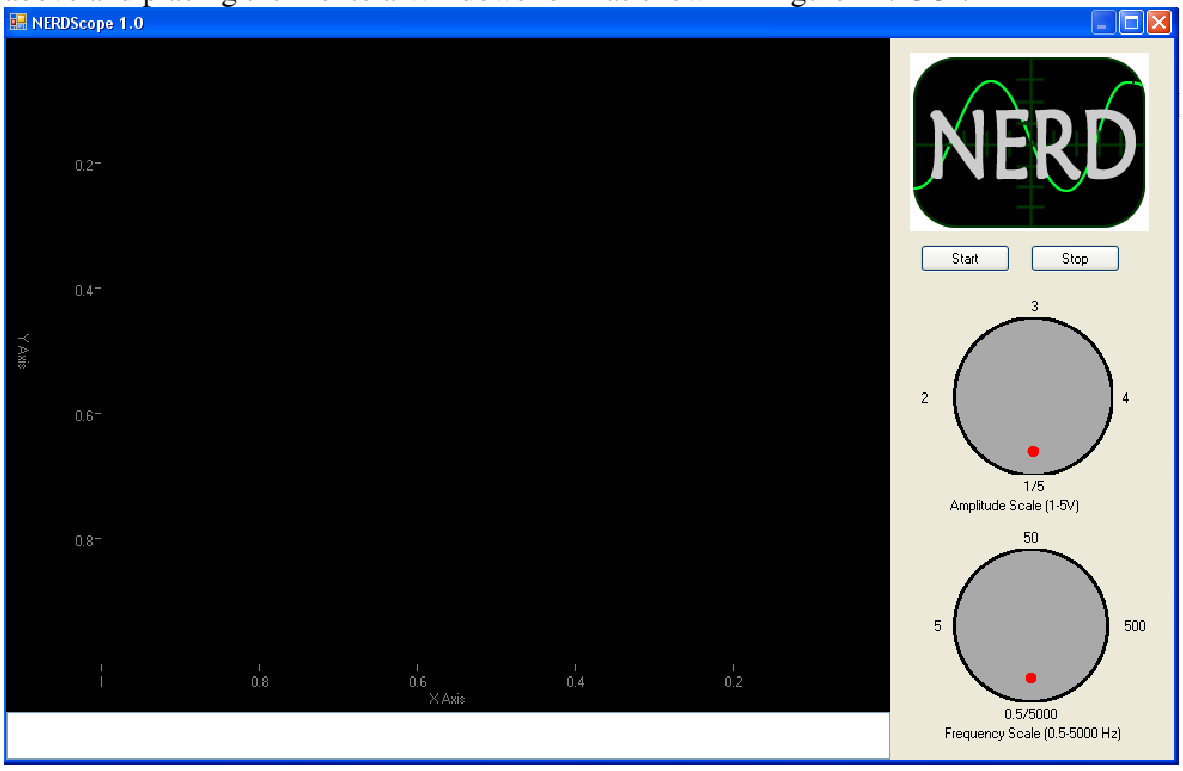


Figure 21: GUI

4.3.3.4.6 Conclusions Summary of GUI

The GUI developed for this project is useable for development as it maintains reliable execution throughout the scope of program operation. In a finished product it would be considered inadequate due to the performance downfalls, as well as the level of visual appeal present in the graphical design.

4.3.3.4.7 Recommendations Summary of GUI

To improve the performance of the GUI, the graph control would need to be upgraded as stated in the graph control recommendations section. This bottleneck is so significant that all others are negligible beside it, and a performance reevaluation would have to be done to see where the next largest performance bottleneck exists.

Aside from performance bottlenecks, the only other functionality recommendation for the GUI is to add fine tuning to the knobs as mentioned, and to add a time delay offset control knob. A voltage offset knob is not deemed to be required, as the scope specification currently only operates in AC coupling mode. If DC coupling is added in the future, a DC offset knob control should be added as well.

Functionality aside, the GUI leaves much to be desired as far as visual appeal and graphics, as mentioned in each section. A suitable background image to emulate the appearance of a scope would be a major factor in improving the marketability and visual appeal of the GUI.

4.3.4 Testing Plan and Results Summary of Software

Several different known signals were manually generated for testing of the software system. These were injected into the data acquisition system at compile-time, and the program was tested for correct output along all of its operating ranges. The entire software system performed remarkably well for an initial design platform, with the exception of the display performance bottleneck mentioned in the GUI section.

4.3.5 System Integration Summary of Software

The software system was integrated as shown in Figure 22: Software System Integration below. The application runs as a Windows form containing the GUI, with the buffer and data acquisition systems added as class properties.

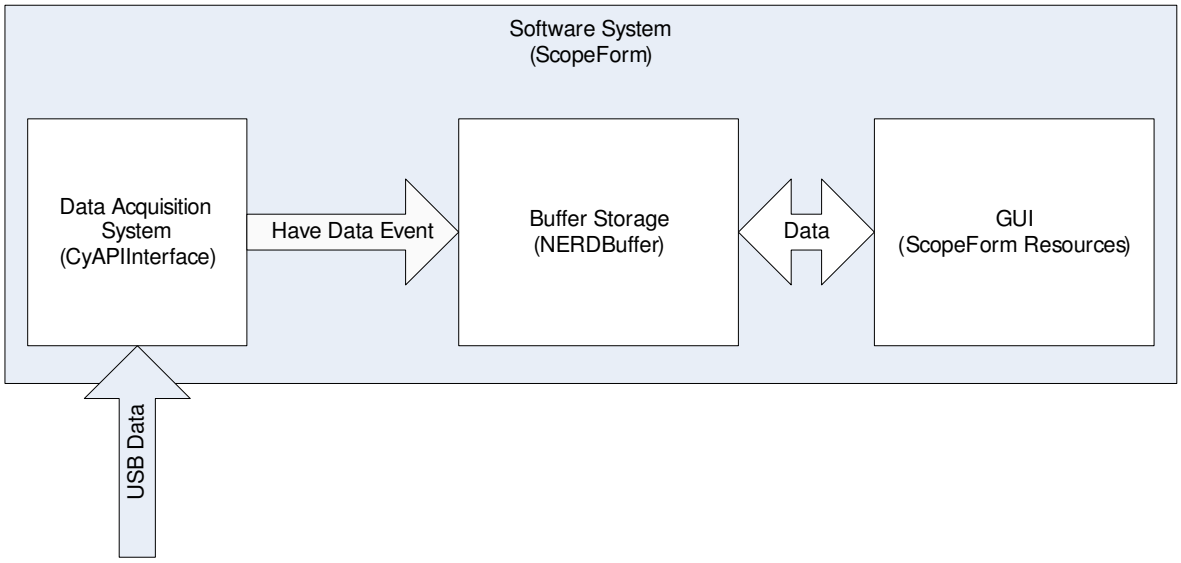


Figure 22: Software System Integration

4.3.6 Conclusions Summary of Software

Functionally, the software system provides a good base for developing future versions of the scope. Its limitations include some performance issues as mentioned in the appropriate sections, and it has yet to fully implement sending commands to the device.

4.3.7 Recommendations Summary of Software

While the limitations of the software prevent it from being a viable finished product, it is a stable platform and would be useful for further development of the firmware and hardware systems.

To improve the software system to the point where it would be a finished product, the first step would be to correct the performance bottlenecks mentioned above, allowing for smoother operation of the entire system. Second would require completing the implementation of □i-directional communications, allowing command and data to be sent to the device for adjusting input voltage ranges, sampling frequency, and other input parameters. Lastly, the GUI would have to be visually improved in a finished product, to increase familiarity and marketability

5 Testing Plan and Results Summary

In order to test the oscilloscope system, both the data acquisition system and the GUI were run individually and measured for performance. Then, both parts of the system were run simultaneously and measured independently to determine the effect of each on the other. Of particular concern was the effect of the GUI on the data acquisition system, as the operating system allows not control of USB transfer scheduling. During this testing, the QuickUSB Simple I/O Model was used and the data bus was sampled once per IFCLK cycle at 48MHz.

- Benchmark PC Vitals
    - Dual Core Pentium D 2.80 GHz

- 1.00 GB RAM
- NVIDIA GeForce 7600 GT (possibly irrelevant)

- Benchmark graphing

To benchmark the graphing portion of the program, a constant set of data was continuously graphed; no data was acquired from the USB device. Each point shown in the table below represents one eight bit sample. During this testing, only the operating system and the graphing portion within the C# development system were running.

Points	50	500	5,000	50,000
Time (ms)	460	4,562	44,516	453,047

- Benchmark data transfer

To benchmark the data transfer portion of the program, a preset of quantity of data was requested from the USB device; no data was graphed. Each packet shown in the table below represents one eight bit sample. During this testing, only the operating system and the data acquisition portion within the C# development system were running.

Packet (Bytes)	51200
Time (ms)	15.6
Data transfer rate	26Mb/s

- Benchmark complete system

To benchmark the complete system, a preset of quantity of data was requested from the USB device while a constant set of data was being graphed. Next, a preset set of data was graphed while data was continuously requested from the USB device. During this testing, only the operating system and the scope application within the C# development system were running.

Points	50	500	5,000
Time (ms)	500	5,109	52,124

Packet (kB)	51200
Time (ms)	31.25
Data transfer rate	13Mb/s

6 System Integration

The system was integrated according to Figure 23: Scope System Integration.

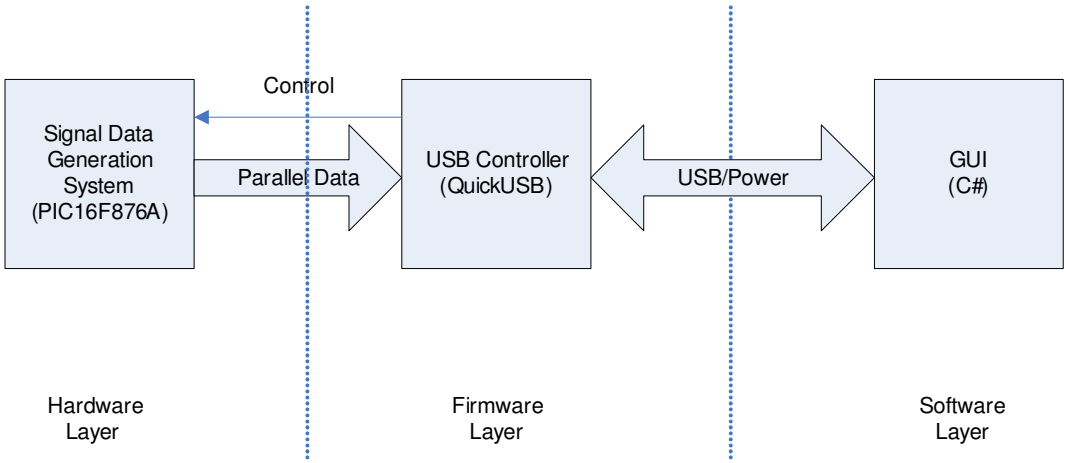


Figure 23: Scope System Integration

To reproduce the project to the level completed as of the writing of this report, there are several hardware steps and several software steps required. Setting up the hardware requires the wiring of the system as shown in the firmware testing section. The picture shown in that section illustrates an example of the recommended breadboard layout scheme. The data generation system is documented in Appendix D: Testing and Integration Hardware.

Setting up the software requires the installation of the QuickUSB software contained on the QuickUSB disk. Alternatively this software can be installed from appendix B6 located on the project software CD. It should be noted that the version on the project CD is password protected and the password can be found in the accompanying readme file. Running the scope also requires Visual C# Express available on the Microsoft website or on the software CD included with this report. It should be noted that the installation of Visual C# Express requires Windows Service Pack 2 to be installed.

Once the drivers and software is installed, and the hardware is plugged in, the oscilloscope project can be opened by opening the NERDScope solution found on the project CD, and run from the Visual C# development environment. As the project is still in development, a published executable was not completed.

7 Conclusions Summary

The first phase in the development of a USB 2.0 based PC oscilloscope was completed. Although some components of the system were simulated, the primary objective of implementing data transfer from a USB device was met. In addition, the completed system is easy to use as hoped for.

The average throughput achieved without the overhead of a running GUI was 26Mb/s, substantially less than the maximum transfer rate supported by High Speed USB. There are several possible sources for this discrepancy, none of which have been quantified or verified. The most

likely major factor is the additional overhead incurred by using the QuickUSB drivers instead of the Cypress firmware. This rate dropped substantially when the developed GUI was running simultaneously with data acquisition. These performance measures highlighted several drawbacks of the current design:

- The GUI software in its current state may have to much overhead for a marketable product
- A continuous oscilloscope capable of processing a 5Mhz signal is not possible with this technology; the data throughput is inadequate and data would be lost. To meet the Nyquist sampling criteria, a continuous operation scope would require at least an 80Mb/s transfer rate to reliably reconstruct a 5MHz signal
- The onboard storage incorporated into the Cypress chip is inadequate for a sampling oscilloscope at only 4kb maximum of storage.

The project as completed to this point allows a stable platform for further development. Furthermore, the current design is a result of rigid adherence to modularity and encapsulation resulting in a system that can be refined subsystem by subsystem while retaining a complete unit for testing and integration.

8 Recommendations Summary

Each section discussed in this report includes recommendations for improvement where applicable. In addition to these subsystems level recommendations it is also recommended that for future development:

- The next phase of development should focus on reducing the overhead required by the GUI, specifically the graphing control
- Either after, or in parallel with a GUI upgrade, the data acquisition daughterboard should be implemented, replacing the current adapter board; its design should include onboard storage at least two orders of magnitude larger than the Cypress on chip storage
- After completing the first two recommendations, using a Cypress firmware platform instead of the QuickUSB platform should be investigated for performance increases
- The custom USB PCB, or a similar design, should be implemented to replace the QuickUSB board to allow production of a reasonably priced unit.
- Upgrade each section independently. Parallel development possible with stable platforms for each section.

The following recommendations are suggested for improved project management.

- A project leader be assigned or voted on.
- A project plan be written in the first week for each section
- At the weekly meetings, each member bring a short written progress plan for their assigned sections of the project, and be able to defend schedule variations

9 Glossary of Terms

Table 2: Glossary of Terms

Acronym	Definition
A/D	Analog to Digital Converter
AGND	Analog ground line
Alias	A frequency created by the combination of the sampling frequency and the actual frequency in signal processing
Anti-alias	A filter designed to remove the alias frequency
DGND	Digital ground line
DSP	Digital Signal Processing
DLL	Dynamic Link Library. Precompiled software code
DPLUS	A signaling line on the USB bus
DMINUS	A signaling line on the USB bus
Endpoint	A data structure used to define USB data transfers
EPROM	Electronically Erasable Programmable Read Only Memory
FIFO	First In First Out. A data structure implemented in computer memory
GUI	Graphical User Interface
GPIF	GPIF
IDE	Integrated Development Environment
I/O	Input/Output
MFC	Microsoft Foundation Classes
NERD	Novice Engineering Research Devices
Nyquist	DSP sampling theory
OOP	Object Orientated Programming
PC	Personal Computer
Ppm	Parts Per Million
USB	Universal Signaling Bus
USD	United Stated Dollars
Via	Hole that goes through all levels of a PCB board
VGA	Variable Gain Amplifier
Wrapper Class	An OOP structure used to define and simplify an interface between OOP structures

10 Appendices

Appendix A: Hardware Appendices

Appendix A2. VGA System Schematic

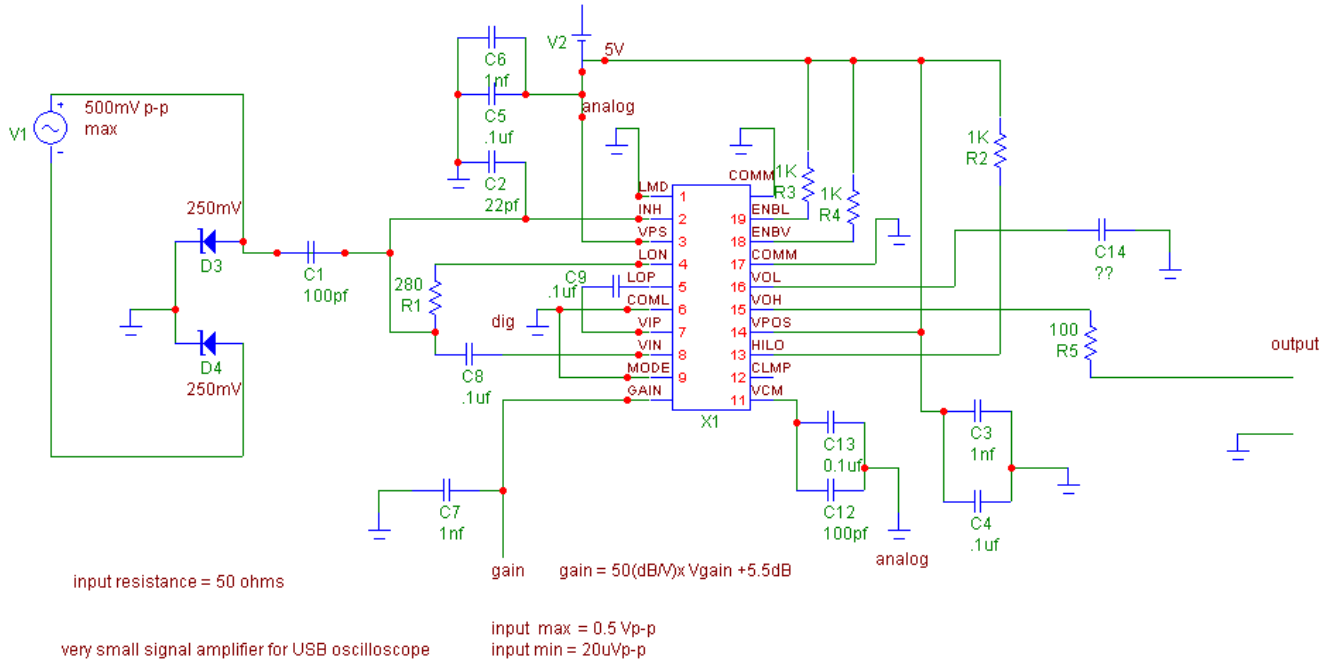


Figure 24: VGA System Schematic

Appendix A3.1 Four-Layer Board PCB Schematic

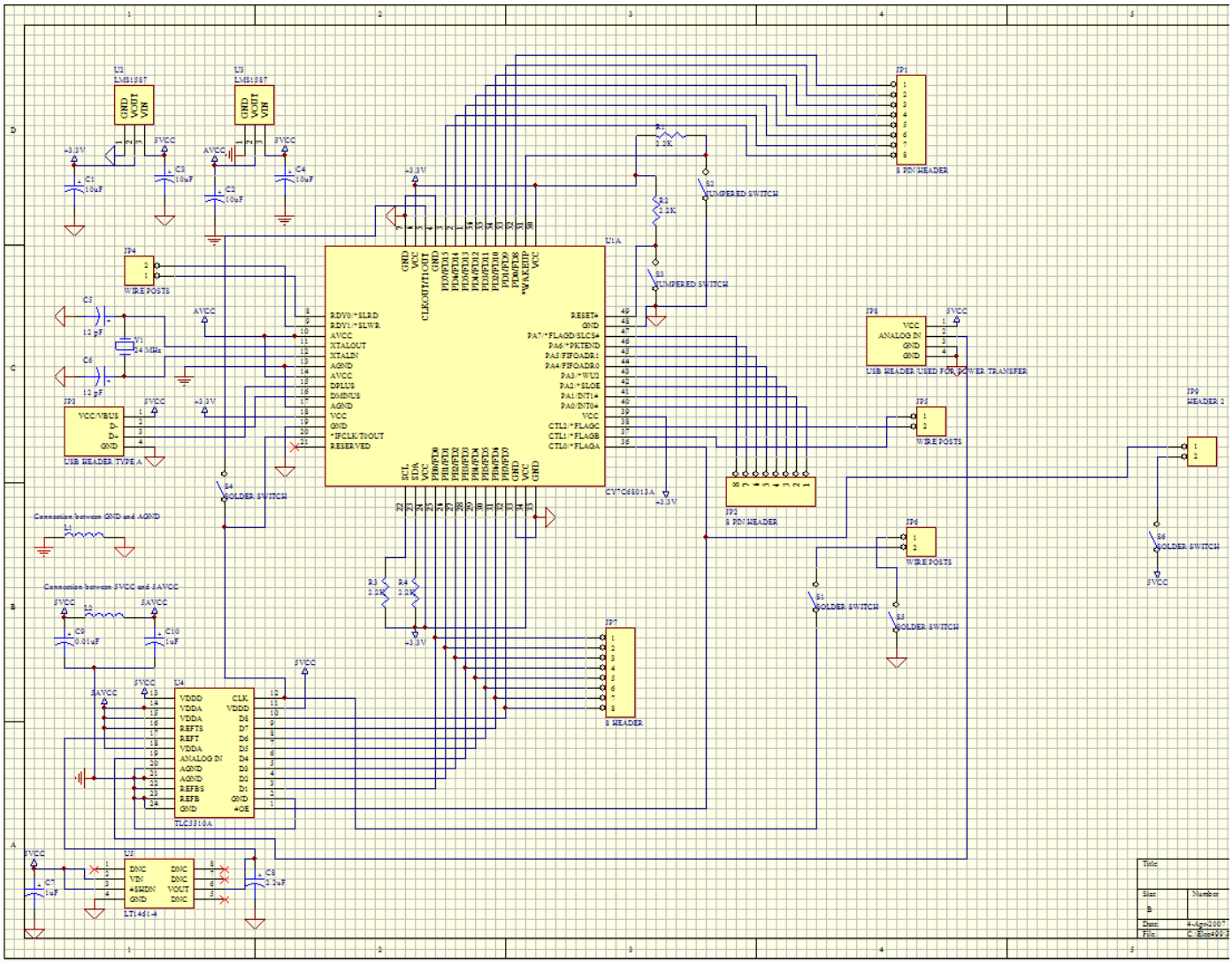


Figure 25: Four-Layer Board PCB Schematic

Appendix A3.2 Four-Layer Board PCB Lay-out

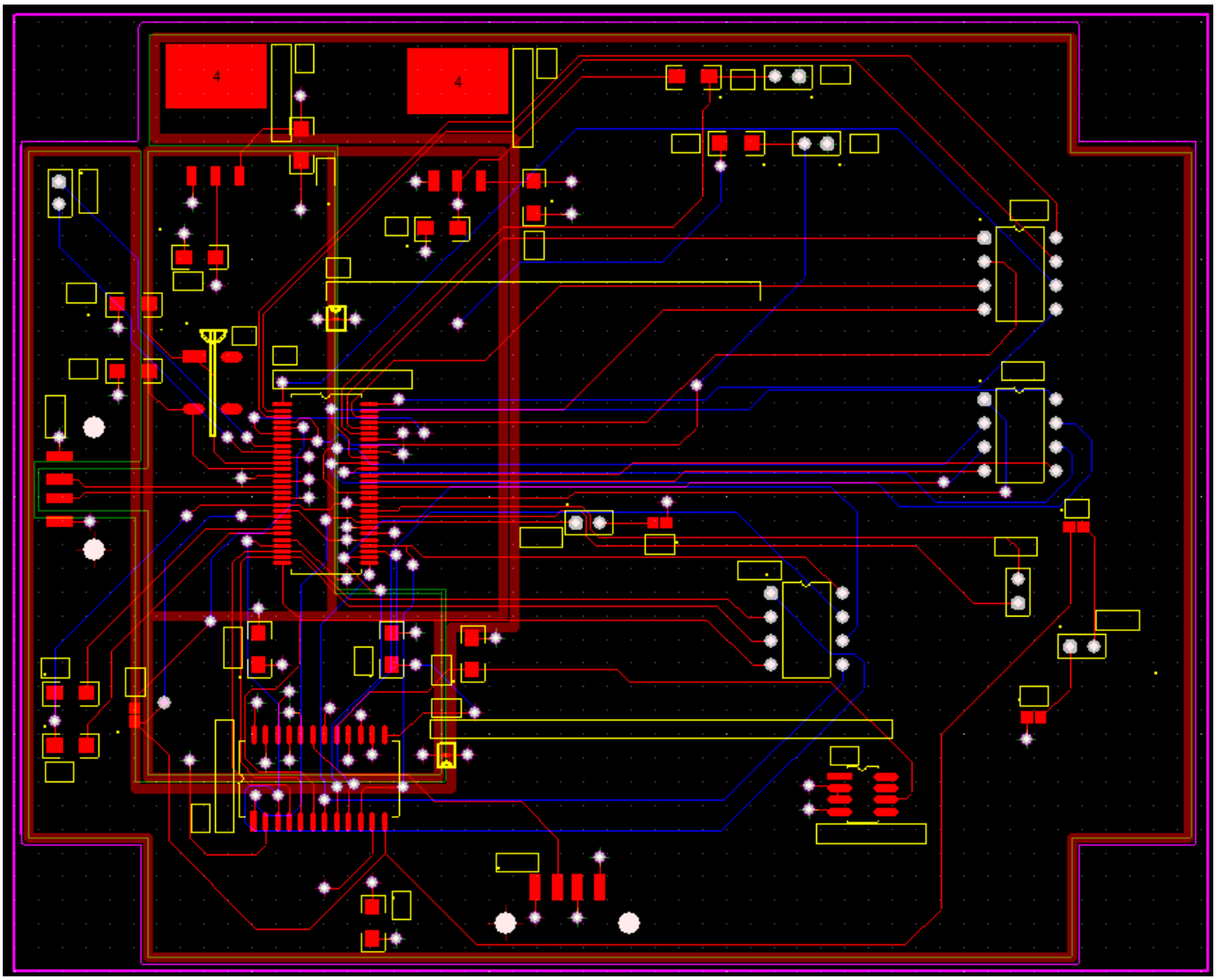


Figure 26: Four-Layer Board PCB Lay-out

Appendix A4. Parts List for PCB

Costs for 4 layer board parts				
Description	Part Number	Quantity	Quantity Purchased	Price (CDN)
Capacitor 12 pF	311-1213-1-ND	2	10	1.65
Oscilator 24 MHz	ASV-24.000MHZ-EJ-T	1	1	3.72
USB Header type A	AE1084-ND	2	2	3.1
Resistor 2.2K	311-2.2KACT-ND	4	10	0.94
Jumpered Switch/ can just wire wrap it so no cost	ALREADY HAVE	2	Not to be purchased	0
3.3 V regulator	LMS1587IS-3.3-ND	2	2	6.34
Capacitor 10 uF	399-4937-1-ND	4	10	8.7
8 PIN HEADER	08-600-10 / A101-ND	3	3	3.93
4 Volt Ref	LT1461CCS8-4	1	1	3.72
Capacitor 1 uf	311-1181-1-ND	1	10	3.04
Capacitor 2.2 uf	399-1260-1-ND	1	10	3.25
8 Wire Wrap Posts + 4 Used as switches	ALREADY HAVE / Can't find exact part however price indicated appears to be the average	12	20	2
Capacitor 0.01 uF	311-1174-1-ND	1	10	1.04
USB Chip	CY7C68013A/ 428-1627-ND	1	1	17.87
A/D Chip	TLC5510A	1	1	6.23
	total price of prototype parts			65.53
Anticipated costs for variable gain board parts				
Variable gain amp	AD605	1	1	19.78
PIC chip used for pricing	16F627	1	1	4.41
D/A used for pricing	TLC7524CD	1	1	3.86
Anticipated costs of additional circuitry		1	1	10

Appendix B: Firmware Apendices

Appendix C: Software Appendices

Appendix C2.1.1 ScopeForm.cs

```
using System;
using System.Collections.Generic;
using System.ComponentModel;
using System.Data;
using System.Drawing;
using System.Text;
using System.Windows.Forms;

using nsCypressBuffer;

namespace NERDScope
{
    public partial class NERDScope : Form
    {
        private CypressBuffer buff;
        private Boolean running = false;

        const double fSample = 25600; // sampling frequency (should be set in
hardware design)
        const double fSig = 50; // signal frequency (only for generated signals)

        /// <summary>
        /// Default constructor
        /// </summary>
        public NERDScope()
        {
            InitializeComponent();

            buff = new CypressBuffer();
            AmplitudeKnob.Range(5, 1);
            FrequencyKnob.Range(5000, 0.5);
        }

        /// <summary>
        /// Repeatedly collects and graphs samples
        /// </summary>
        private void Run()
        {
            running = true;
            while(true)
            {
                double fDisp = FrequencyKnob.Value(true);
                int numSamples = (int)(fSample / fDisp);
                graphBox1.LabelX("Time (s)", 0, 1 / fDisp, 8);
                graphBox1.LabelY("Voltage (V)", -AmplitudeKnob.Value(),
AmplitudeKnob.Value(), 10);

                double x1 = 0;
                double y1 = 0;
                double x2, y2;

                for (int i = 0; i <= numSamples; i++)
                {
                    if (!running) return;
                    x2 = i / (fDisp * numSamples);
                    object val = buff.next();
                    while (val == null) val = buff.next();
```

```
                    y2 = ((int)val-128.0)/64.0; // generated signal, should be
value from sample
                    graphBox1.Plot(x1, y1, x2, y2);
                    x1 = x2;
                    y1 = y2;
                    Application.DoEvents();
                }
            }
        }

        /// <summary>
        /// Event handler for form closing.
        /// Cleans up resources.
        /// </summary>
        /// <param name="sender">object generating event, usually this form</param>
        /// <param name="e">arguments for event</param>
        private void NERDScope_FormClosing(object sender, FormClosingEventArgs e)
        {
            buff.Dispose();
        }

        /// <summary>
        /// Event handler for  Start button clicked.
        /// starts sampling and graphing samples.
        /// </summary>
        /// <param name="sender">object generating event</param>
        /// <param name="e">arguments for event</param>
        private void StartButton_Click(object sender, EventArgs e)
        {
            Run();
        }

        /// <summary>
        /// Event handler for Stop button clicked.
        /// stops sampling.
        /// </summary>
        /// <param name="sender">object sending event</param>
        /// <param name="e">arguments for event</param>
        private void StopButton_Click(object sender, EventArgs e)
        {
            running = false;
        }
    }
}
```

Appendix C2.1.2: CyAPIBuffer.cs

```
using System;
using System.Threading;
using CyUSB;
using nsNERDBuffer;

namespace nsCypressBuffer
{
    /// <summary>
    /// Buffer class for interfacing to Cypress USB chips using the .NET 2.0 API.
    /// </summary>
    public class CypressBuffer
    {
        public NERDBuffer myBuffer;
        private CyAPIInterface myAPI;
        private Thread backGroundThread;

        /// <summary>
        /// Constructor for API Buffer, sets up interface events and processes.
        /// Starts background thread polling USB bus for data.
        /// Needs to be cleaned up with Dispose()
        /// </summary>
        public CypressBuffer()
        {
            myBuffer = new NERDBuffer();
            myAPI = new CyAPIInterface();
            myAPI.haveData += new haveDataDelegate(haveDataEvent);

            backGroundThread = new Thread(new ThreadStart(myAPI.Run));
            backGroundThread.Start();
        }

        /// <summary>
        /// Destructor
        /// Needs to stop background thread so it doesn't continue to run after
program exits.
        /// </summary>
        public void Dispose()
        {
            while(backGroundThread.IsAlive)
            backGroundThread.Abort();
        }

        /// <summary>
        /// Event triggered when USB bus polling receives data.
        /// Pushes received data onto buffer to be processed and clears Interface.
        /// </summary>
        private void haveDataEvent()
        {
            myBuffer.push(myAPI.dataList);
            myAPI.dataList.Clear();
        }

        /// <summary>
        /// Pops the first element off the buffer
        /// </summary>
        /// <returns>object popped, generic class</returns>
        public object next()
        {
```

```
            object val = myBuffer.next();
            if (myBuffer.Empty()) myAPI.gettingData = true;
            return val;
        }

        /// <summary>
        /// Sends a command to the device
        /// </summary>
        /// <param name="cmd">Command Data structure with intruction code and
data</param>
        public void send(Command cmd)
        {
            myAPI.commands.Add(cmd);
        }
    }
}
```

Appendix C2.1.3: CyAPIInterface.cs

```
using System;
using System.Collections;
using CyUSB;
using NerdQuickUSB;

namespace nsCypressBuffer
{
    public delegate void haveDataDelegate();

    /// <summary>
    /// USB interface thread class for polling USB port for data and forwarding it
to the buffer.
    /// </summary>
    public class CyAPIInterface
    {
        public event haveDataDelegate haveData;
        public ArrayList dataList;
        public ArrayList commands;
        private USBDeviceList usbDevices;
        private USBDevice device;
        public bool gettingData;

        /// <summary>
        /// Default constructor
        /// </summary>
        public CyAPIInterface()
        {
            commands = new ArrayList();
            dataList = new ArrayList();

            App_PnP_Callback evHandler = new App_PnP_Callback(PnP_Event_Handler);
            usbDevices = new USBDeviceList(CyConst.DEVICES_MSC, evHandler);

            foreach (USBDevice curDevice in usbDevices) // Choose which device to
use & set up
            {
                Console.Write("VendorID=" + curDevice.VendorID.ToString() + " ");
                Console.Write("USBAdress=" + curDevice.USBAddress.ToString() + "
");
                Console.Write("Product=" + curDevice.Product + " ");
                Console.WriteLine("FriendlyName=" + curDevice.FriendlyName + " ");
                Console.WriteLine("Name=" + curDevice.Name + " ");
                Console.WriteLine(" ");
            }

            device = usbDevices[0] as CyUSBDevice;
        }

        /// <summary>
        /// Continuously polls USB port for data, and generates data event flags.
        /// </summary>
        public void Run()
        {
            QuickUSB myUSB = new QuickUSB();
            while (true)
            {
                while(!gettingData) {}
                bool opened = myUSB.OpenDevice();
```

```
                opened = myUSB.OpenDevice();
                int length = 512;
                byte[] data = new byte[length];
                bool read = myUSB.Read(out data, length);

                for (int i = 0; i < length; i++)
                {
                    int newVal = (int)(data[i]);
                    dataList.Add(newVal);
                }

                closed = myUSB.CloseDevice();

                if (haveData != null) // generate event for buffer
                {
                    haveData();
                }

                //check if there are commands to send
                while(commands.Count > 0)
                {
                    //issue command
                }

                gettingData = false;
            }
        }

        /// <summary>
        /// Handles events associated with hot swapping of USB devices by adding or
removing them from stored array of available devices.
        /// </summary>
        /// <param name="pnpEvent">enum representing type of event</param>
        /// <param name="hRemovedDevice">handle of device if removed</param>
        public void PnP_Event_Handler(IntPtr pnpEvent, IntPtr hRemovedDevice)
        {
            if (pnpEvent.Equals(CyConst.DBT_DEVICEREMOVECOMPLETE))
            {
                usbDevices.Remove(hRemovedDevice);
            }

            if (pnpEvent.Equals(CyConst.DBT_DEVICEARRIVAL))
            {
                usbDevices.Add();
            }
        }
    }
}
```

Appendix C2.1.4: NERDBuffer.cs

```
using System;
using System.Collections;

namespace nsNERDBuffer
{

    /// <summary>
    /// A simple FIFO data buffer with support for pushing arrays
    /// </summary>
    public class NERDBuffer
    {
        private Queue mQueue;

        /// <summary>
        /// Default Constructor, initializes queue
        /// </summary>
        public NERDBuffer()
        {
            mQueue = new Queue();
        }

        /// <summary>
        /// Pops the first element off the queue
        /// </summary>
        /// <returns>object popped, generic class</returns>
        public object next()
        {
            if (mQueue.Count == 0) return null;
            return mQueue.Dequeue();
        }

        /// <summary>
        /// Pushes each object in array in order it appears
        /// </summary>
        /// <param name="dataList">ArrayList of objects to be pushed</param>
        public void push(ArrayList dataList)
        {
            foreach (object temp in dataList)
            {
                push(temp);
            }
        }

        public bool Empty()
        {
            return (mQueue.Count == 0);
        }

        /// <summary>
        /// Pushes a single object onto the back of the queue
        /// </summary>
        /// <param name="data">Object to be pushed</param>
        public void push(object data)
        {
            mQueue.Enqueue(data);
        }
    }
}
```

Appendix C2.1.5: Graphbox.cs

```
using System;
using System.Collections.Generic;
using System.ComponentModel;
using System.Drawing;
using System.Data;
using System.Text;
using System.Windows.Forms;

namespace GraphBox
{
    public partial class GraphBox : UserControl
    {
        public int YDivisions = 5;
        public double YAxisMax = 1;
        public double YAxisMin = 0;
        public String YAxis = "Y Axis";

        public int XDivisions = 5;
        public double XAxisMax = 1;
        public double XAxisMin = 0;
        public String XAxis = "X Axis";

        private Pen graphPen = new Pen(Brushes.Green);
        private Pen gridPen = new Pen(Brushes.Gray);

        private SolidBrush clearPen = new SolidBrush(Color.Black);
        private Brush gridBrush = Brushes.Gray;

        private StringFormat strYLabel = new StringFormat();
        private StringFormat strHorz = new StringFormat();
        private StringFormat strVert = new
StringFormat(StringFormatFlags.DirectionVertical);

        /// <summary>
        /// Default Constructor
        /// </summary>
        public GraphBox()
        {

            InitializeComponent();
            if (DesignMode) return;

            graphPen.Width = 3.0F;

            strYLabel.Alignment = StringAlignment.Far;
            strHorz.Alignment = StringAlignment.Center;
            strVert.Alignment = StringAlignment.Center;

            Graph.Image = new Bitmap(Graph.ClientSize.Width,
Graph.ClientSize.Height);
            XLabel.Image = new Bitmap(XLabel.ClientSize.Width,
XLabel.ClientSize.Height);
            YLabel.Image = new Bitmap(YLabel.ClientSize.Width,
YLabel.ClientSize.Height);

        }

        /// <summary>
```

```
/// Sets parameters for Y-axis display
/// </summary>
/// <param name="min">Minimum value of Y-axis</param>
/// <param name="max">Maximum value of Y-axis</param>
/// <param name="div">Number of grid divisions in Y-axis</param>
public void setYBounds(double min, double max, int div)
{
    YDivisions = div;
    setYBounds(min, max);
}

/// <summary>
/// Sets parameters for Y-axis display
/// </summary>
/// <param name="min">Minimum value of Y-axis</param>
/// <param name="max">Maximum value of Y-axis</param>
public void setYBounds(double min, double max)
{
    YAxisMin = min;
    YAxisMax = max;
    LabelY();
}

/// <summary>
/// Sets parameters for X-axis display
/// </summary>
/// <param name="min">Minimum value of X-axis</param>
/// <param name="max">Maximum value of X-axis</param>
/// <param name="div">Number of grid divisions in X-axis</param>
public void setXBounds(double min, double max, int div)
{
    XDivisions = div;
    setXBounds(min, max);
}

/// <summary>
/// Sets parameters for X-axis display
/// </summary>
/// <param name="min">Minimum value of X-axis</param>
/// <param name="max">Maximum value of X-axis</param>
public void setXBounds(double min, double max)
{
    XAxisMin = min;
    XAxisMax = max;
    LabelX();
}

/// <summary>
/// Plots line on graph using actual values
/// </summary>
/// <param name="x1">X value for point 1</param>
/// <param name="y1">Y value for point 1</param>
/// <param name="x2">X value for point 2</param>
/// <param name="y2">Y value for point 2</param>
public void Plot(double x1, double y1, double x2, double y2)
{
    double xScale = Graph.Width / (XAxisMax - XAxisMin);
    double yScale = Graph.Height / (YAxisMax - YAxisMin);
    int iX1 = (int)((x1 - XAxisMin) * xScale);
```

```
            int iY1 = (int)((y1 - YAxisMin) * yScale);
            int iX2 = (int)((x2 - XAxisMin) * xScale);
            int iY2 = (int)((y2 - YAxisMin) * yScale);
            Plot(iX1, iY1, iX2, iY2);
        }

        /// <summary>
        /// Plots line on graph using pixel values
        /// </summary>
        /// <param name="x1">X value for point 1</param>
        /// <param name="y1">Y value for point 1</param>
        /// <param name="x2">X value for point 2</param>
        /// <param name="y2">Y value for point 2</param>
        public void Plot(int x1, int y1, int x2, int y2)
        {
            Graphics g = Graphics.FromImage(Graph.Image);
            g.FillRectangle(clearPen, Graph.Width - x1+1, 0, x1 - x2,
Graph.Height);
            g.DrawLine(graphPen, Graph.Width - x1, y1, Graph.Width - x2, y2);
            g.Dispose();
            DrawGrid();
        }

        /// <summary>
        /// Labels X-axis
        /// </summary>
        /// <param name="Label">new label</param>
        public void LabelX(String Label)
        { XAxis = Label; LabelX(); }
        /// <summary>
        /// Labels Y-axis
        /// </summary>
        /// <param name="Label">new label</param>
        public void LabelY(String Label)
        { YAxis = Label; LabelY(); }

        /// <summary>
        /// Labels X-axis
        /// </summary>
        /// <param name="xMin">Minimum Value</param>
        /// <param name="xMax">Maximum Value</param>
        public void LabelX(double xMin, double xMax)
        { XAxisMax = xMin; XAxisMin = xMax; LabelX(); }
        /// <summary>
        /// Labels Y-axis
        /// </summary>
        /// <param name="xMin">Minimum Value</param>
        /// <param name="xMax">Maximum Value</param>
        public void LabelY(double yMin, double yMax)
        { YAxisMax = yMin; YAxisMin = yMax; LabelY(); }

        /// <summary>
        /// Labels X-axis
        /// </summary>
        /// <param name="Label">New Label</param>
        /// <param name="xMin">Minimum Value</param>
        /// <param name="xMax">Maximum Value</param>
        public void LabelX(String Label, double xMin, double xMax)
        { XAxis = Label; LabelX(xMin, xMax); }
```

```
        /// <summary>
        /// Labels Y-axis
        /// </summary>
        /// <param name="Label">New Label</param>
        /// <param name="xMin">Minimum Value</param>
        /// <param name="xMax">Maximum Value</param>
        public void LabelY(String Label, double yMin, double yMax)
        { YAxis = Label; LabelY(yMin, yMax); }
        /// <summary>
        /// Labels X-axis
        /// </summary>
        /// <param name="Label">New Label</param>
        /// <param name="xMin">Minimum Value</param>
        /// <param name="xMax">Maximum Value</param>
        /// <param name="div">Number of grid divisions</param>
        public void LabelX(String Label, double xMin, double xMax, int div)
        { XDivisions = div; LabelX(Label, xMin, xMax); }
        /// <summary>
        /// Labels Y-axis
        /// </summary>
        /// <param name="Label">New Label</param>
        /// <param name="xMin">Minimum Value</param>
        /// <param name="xMax">Maximum Value</param>
        /// <param name="div">Number of grid divisions</param>
        public void LabelY(String Label, double yMin, double yMax, int div)
        { YDivisions = div; LabelY(Label, yMin, yMax); }
        /// <summary>
        /// Labels X-axis
        /// </summary>
        public void LabelX()
        {
            Graphics g = Graphics.FromImage(XLabel.Image);
            g.Clear(XLabel.BackColor);
            double xStep = (XAxisMax - XAxisMin) / XDivisions;
            g.DrawString(XAxis, Font, gridBrush, XLabel.Width / 2, 22, strHorz);
            for (int i = 0; i < XDivisions; i++)
            {
                int xPos = i * (XLabel.Width - YLabel.Width) / XDivisions +
YLabel.Width;
                g.DrawLine(gridPen, xPos, 0, xPos, 5);
                g.DrawString(String.Format("{0:G4}", XAxisMax - i * xStep), Font,
gridBrush, new Rectangle(xPos - 30, 8, 60, 15), strHorz);
            }
            g.Dispose();
            XLabel.Invalidate();
        }

        /// <summary>
        /// Labels Y-axis
        /// </summary>
        public void LabelY()
        {
            Graphics g = Graphics.FromImage(YLabel.Image);
            g.Clear(YLabel.BackColor);
            double yStep = (YAxisMax - YAxisMin) / YDivisions;
            g.DrawString(YAxis, Font, gridBrush, 5, YLabel.Height / 2, strVert);
            for (int i = 1; i < YDivisions; i++)
            {
                int yPos = i * YLabel.Height / YDivisions;
```

```
                g.DrawLine(gridPen, YLabel.Width - 5, yPos, YLabel.Width, yPos);
                g.DrawString(String.Format("{0:G4}", YAxisMin + i * yStep), Font,
gridBrush, new Rectangle(15, yPos - (int)(Font.Size / 2), YLabel.Width-20, 20),
strYLabel);
            }
            g.Dispose();
            YLabel.Invalidate();
        }

        /// <summary>
        /// Clears Graph Area
        /// </summary>
        public void ClearGraph()
        {
            Graphics g = Graphics.FromImage(Graph.Image);
            g.Clear(Color.Black);
            g.Dispose();
            DrawGrid();
        }

        /// <summary>
        /// Draws the grid on the graphing area
        /// </summary>
        public void DrawGrid()
        {
            Graphics g = Graphics.FromImage(Graph.Image);
            for (int i = 1; i < YDivisions; i++)
            {
                int yPos = i * Graph.Height / YDivisions;
                g.DrawLine(gridPen, 0, yPos, Graph.Width, yPos);
            }
            g.DrawLine(gridPen, 0, Graph.Height - 1, Graph.Width, Graph.Height -
1);
            for (int i = 0; i < XDivisions; i++)
            {
                int xPos = i * Graph.Width / XDivisions;
                g.DrawLine(gridPen, xPos, 0, xPos, Graph.Height);
            }
            g.Dispose();
            Graph.Invalidate();
        }

        /// <summary>
        /// Event handler for control resize
        /// </summary>
        /// <param name="sender">object generating event</param>
        /// <param name="e">event arguments</param>
        private void Graph_Resize(object sender, EventArgs e)
        {
            Graph.Image = new Bitmap(Graph.Image, new Size(Graph.Width,
Graph.Height));
            XLabel.Image = new Bitmap(XLabel.Width, XLabel.Height);
            YLabel.Image = new Bitmap(YLabel.Width, YLabel.Height);
            LabelX();
            LabelY();
        }
    }
}
```

Appendix C2.1.6 Knob.cs

```
using System;
using System.Collections.Generic;
using System.ComponentModel;
using System.Drawing;
using System.Data;
using System.Text;
using System.Windows.Forms;

namespace nsKnob
{
    public partial class Knob : UserControl
    {
        private double value = 0;
        private double maxValue = 1;
        private double minValue = 0;

        /// <summary>
        /// Calculates Value of control
        /// </summary>
        /// <returns>Value control is set at</returns>
        public double Value() { return value * (maxValue - minValue) + minValue; }

        /// <summary>
        /// Calculates Value of control
        /// </summary>
        /// <param name="logBase">true if control increases on logarithmic scale,
false if on linear scale</param>
        /// <returns>Value control is set at</returns>
        public double Value(bool logBase) {
            if (!logBase) return Value();
            double powBase = Math.Log10(maxValue / minValue);
            return Math.Pow(10, powBase * (value - 1)) * maxValue;
        }

        /// <summary>
        /// Sets range of values for control
        /// </summary>
        /// <param name="newMaxValue">new maximum value of range</param>
        /// <param name="newMinValue">new minimum value of range</param>
        public void Range(double newMaxValue, double newMinValue) { maxValue =
newMaxValue; minValue = newMinValue; }

        /// <summary>
        /// Gets maximum value of control
        /// </summary>
        /// <returns>maximum value of control</returns>
        public double MaxValue() { return maxValue; }

        /// <summary>
        /// Gets minimum value of control
        /// </summary>
        /// <returns>minimum value of control</returns>
        public double MinValue() { return minValue; }

        /// <summary>
        /// Default Constructor
        /// </summary>
        public Knob()
```

```
        {
            InitializeComponent();
            KnobImage.Image = new Bitmap(KnobImage.ClientSize.Width,
KnobImage.ClientSize.Height);
            drawKnob();
        }

        /// <summary>
        /// Event of mouse moving over control.
        /// Used to set value
        /// </summary>
        /// <param name="sender">object generating event</param>
        /// <param name="e">event arguments</param>
        private void KnobImage_MouseMove(object sender, MouseEventArgs e)
        {
            if (e.Button == MouseButtons.Left)
            {
                int height = KnobImage.ClientSize.Height;
                int width = KnobImage.ClientSize.Width;
                int radius;
                if (width > height) radius = height;
                else radius = width;
                int Cx = width / 2;
                int Cy = height / 2;
                if (width > height) radius = height / 2;
                else radius = (int)(0.4 * width);

                double alpha = ((Math.Atan2(e.Y - Cy, e.X - Cx)));
                double alpha2 = alpha - (Math.PI / 2);
                if (alpha2 < 0)
                {
                    alpha2 = alpha2 + (2 * Math.PI);
                }

                value = ((alpha2) / (2 * Math.PI));

                drawKnob();
            }
        }

        /// <summary>
        /// draws knob on form
        /// </summary>
        public void drawKnob()
        {
            int height = KnobImage.ClientSize.Height;
            int width = KnobImage.ClientSize.Width;
            int Cx = width / 2;
            int Cy = height / 2;
            int radius;
            if (width > height) radius = height;
            else radius = width;
            if (width > height) radius = height / 2;
            else radius = (int)(0.4 * width);

            double alpha = value * 2 * Math.PI + Math.PI/2;

            int Ex = (int)(Math.Cos(alpha) * 0.7 * radius + Cx);
            int Ey = (int)(Math.Sin(alpha) * 0.7 * radius + Cy);
```

```
            System.Drawing.SolidBrush brush1 = new
System.Drawing.SolidBrush(System.Drawing.Color.Red);
            System.Drawing.SolidBrush brush2 = new
System.Drawing.SolidBrush(System.Drawing.Color.DarkGray);
            System.Drawing.Graphics formGraphics =
Graphics.FromImage(KnobImage.Image);

            // Create a new pen.
            Pen skyBluePen = new Pen(Brushes.Black);

            // Set the pen's width.
            skyBluePen.Width = 3.0F;

            // Set the LineJoin property.
            skyBluePen.LineJoin = System.Drawing.Drawing2D.LineJoin.Bevel;

            formGraphics.FillEllipse(brush2, new Rectangle(Cx - radius, Cy -
radius, 2 * radius, 2 * radius));
            formGraphics.FillEllipse(brush1, new Rectangle(Ex - 5, Ey - 5, 10,
10));
            formGraphics.DrawEllipse(skyBluePen, new Rectangle(Cx - radius, Cy -
radius, 2 * radius, 2 * radius));

            brush1.Dispose();
            skyBluePen.Dispose();
            formGraphics.Dispose();

            KnobImage.Invalidate();
        }

        /// <summary>
        /// Event handling resize of control
        /// </summary>
        /// <param name="sender">object generating event</param>
        /// <param name="e">event arguments</param>
        private void KnobImage_ClientSizeChanged(object sender, EventArgs e)
        {
            KnobImage.Image = new Bitmap(KnobImage.ClientSize.Width,
KnobImage.ClientSize.Height);
            drawKnob();
        }
    }
}
```

C2.1.7 QuickUSB.cs

```
using System;
using System.Collections.Generic;
using System.Text;
using System.Runtime.InteropServices;

namespace NerdQuickUSB
{
    public class QuickUSB
    {
        private bool mdeviceOpened = false;
        private string mdeviceName = "QUSB-0";
        private IntPtr mdeviceHandle = new IntPtr();
        public string LastError = "";

        //QuickUSB manufacturer DLL declarations
        [DllImport("quickusb.dll", CharSet = CharSet.Ansi)]
        private static extern int QuickUsbFindModules(StringBuilder nameList, int
bufferLength);
        [DllImport("quickusb.dll", CharSet = CharSet.Ansi)]
        static extern int QuickUsbOpen(out IntPtr handle, string devName);
        [DllImport("quickusb.dll", CharSet = CharSet.Ansi)]
        private static extern int QuickUsbClose(IntPtr handle);
        [DllImport("quickusb.dll", CharSet = CharSet.Ansi)]
        static extern int QuickUsbReadData(IntPtr Handle, byte[] outData, out int
length);
        [DllImport("quickusb.dll", CharSet = CharSet.Ansi)]
        static extern int QuickUsbSetTimeout(IntPtr handle, int length);

        /// <summary>
        /// opens the usb device for reead/write
        /// </summary>
        /// <returns>true if opened, false otherwise</returns>
        public bool OpenDevice()
        {
            StringBuilder sb = new StringBuilder(128);
            if(QuickUsbFindModules(sb, sb.Capacity) == 0) return false;
            if (QuickUsbOpen(out mdeviceHandle, mdeviceName) == 0) return false;
            if (!SetTimeout()) return false;
            return QuickUsbOpen(out mdeviceHandle, mdeviceName) != 0;
        }

        /// <summary>
        /// sets the timeout setting of the QuickUSB module
        /// </summary>
        /// <returns>true if successful, false otherwise</returns>
        private bool SetTimeout()
        {
            int returnValue = 0;
            returnValue = QuickUsbSetTimeout(mdeviceHandle, 10000);
            if (returnValue != 0)
            {
                mdeviceOpened = false;
                return true;
            }
            return false;

        }
```

```
        /// <summary>
        /// closes the device when read/write is completed
        /// </summary>
        /// <returns>true if successful, false otherwise</returns>
        public bool CloseDevice()
        {
            int returnValue = 0;
            returnValue = QuickUsbClose(mdeviceHandle);
            if (returnValue != 0)
            {
                mdeviceOpened = false;
                return true;
            }
            return false;
        }

        /// <summary>
        /// reads a number of bytes and places into an array
        /// </summary>
        /// <param name="data">buffer array of bytes to load with data</param>
        /// <param name="length">number of bytes in data array</param>
        /// <returns>true if successful, false otherwise</returns>
        public bool Read(out byte[] data, int length)
        {
            try
            {
                if (mdeviceOpened)
                {

                    byte[] readData = new byte[length];



                    int len = readData.Length;

                    int result = QuickUsbReadData(mdeviceHandle, readData, out
len);
                    if (result == 0)
                    {
                        LastError = "USB Connection is not open";
                        //IsOpen = false;
                        CloseDevice();
                        data = readData;
                        return false;
                    }

                    data = readData;
                    return true;
                }
                else
                {
                    LastError = "Device not open";
                    data = new byte[length];
                    return false;
                }
            }
            catch (DllNotFoundException)
            {
```

```
                LastError = "Cannot find the QuickUSB dll library. Please install
QuickUsb Drivers.";
                data = new byte[length];
                return false;
            }

        }

    }
}
```

Appendix D: Testing and Integration Hardware Apendices

Appendix D1: Hardware Data Stream Generation System

D1.1 Overview

D1.1.1 Introduction

The data stream generation hardware is the system used to simulate data from and A/D system during the project presentation. It was responsible for simulating a digital sinusoidal wave on the 8-bit data bus. While not a part of the project itself, it is a valuable testing tool for generating a known signal.

D1.1.2 Objectives

- To program a PIC microcontroller to communicate data to the QuickUSB board
- To implement low-level hardware testing of waveforms and handshake communications

D1.1.3 Theory

The hardware data stream generation system is designed to implement the QuickUSB Full Handshake I/O Model as described in the main report. It utilizes a PIC16F876A microcontroller to communicate with the Quick USB module and to compute the 8-bit digital data value for loading into the USB chip.

The PicBit is a development tool designed for PIC microcontrollers. It consists of an electronics kit with a circuit board, IC socket, RJ11 jack, resonator, and some minor components. It assembles into a circuit which will easily plug into a common bread board. The PicBit unit breaks out all of the PIC pins onto the breadboard, but has the high frequency resonator and programmer components soldered in, allowing for reliable rapid prototyping.

D1.2 Design Rationale

The PIC16F876A chip was chosen by process of elimination. Of all the PIC microchips on hand, the PIC16F876A was the only 28-pin model, which would fit the PicBit bread boarding socket on hand.

To simplify and expedite chip firmware design, the CCS PCWH development environment and C compiler was used to generate the firmware hex code for the PIC16F876A. The firmware was loaded onto the chip using an ICD-U40 programmer because that was the programmer module available.

D1.3 Design Implementation

D1.3.1 Hardware

The test circuit was wired using the schematic shown in Figure 27: Test System Schematic below

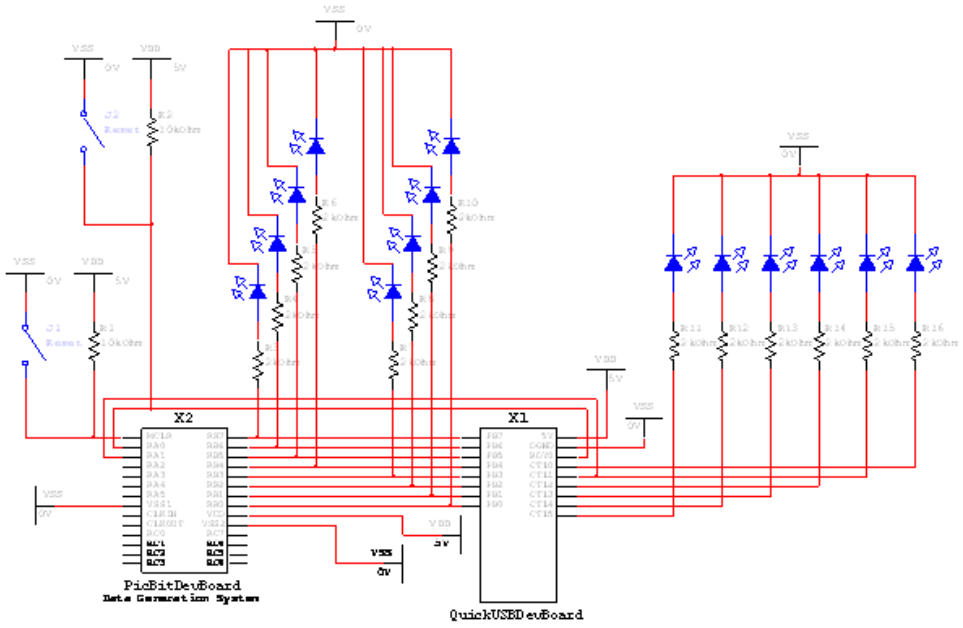


Figure 27: Test System Schematic

A photograph of the fully assembled unit can be seen in Figure 28: Testing Circuit Board.

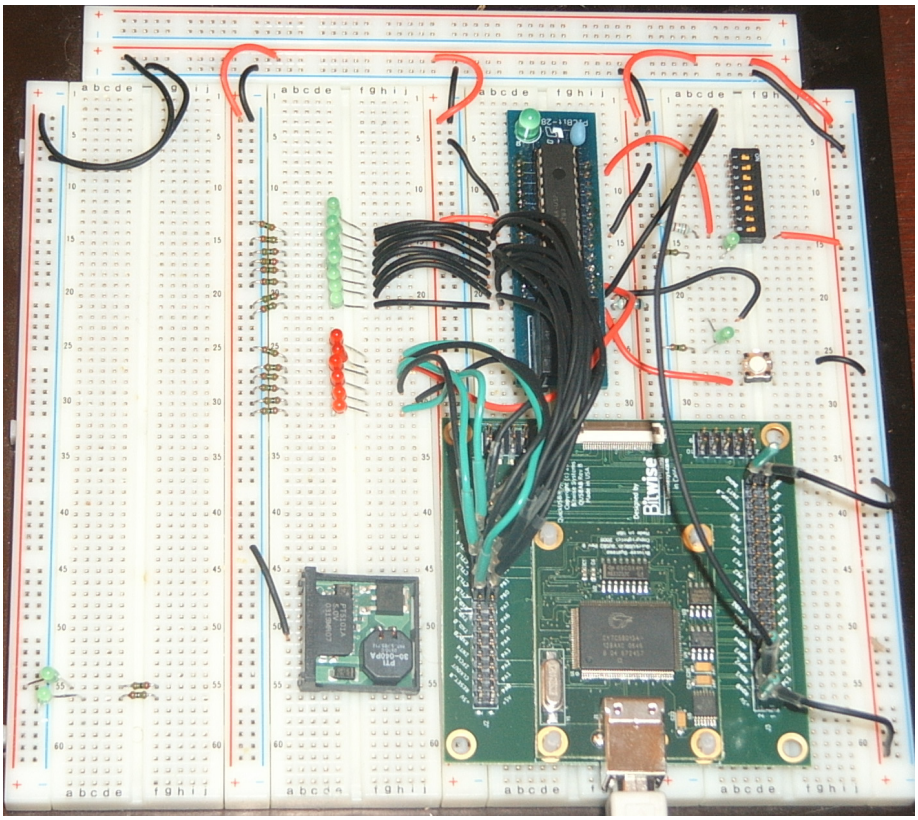


Figure 28: Testing Circuit Board

D1.3.2 Firmware

The firmware code is outlined in the files wav.h and wav.c shown below:

wav.h

```
#include <16F876A.h>
#device adc=8
#use delay(clock=20000000)
#fuses NOWDT,HS, PUT, NOPROTECT, NODEBUG, BROWNOUT, NOLVP, NOCPD,
NOWRT
```

wav.c

```
#include "E:\electronics\programs\test\led2\wav.h"

#define SINE_TABLE_SIZE 256
#define TRIANGLE_TABLE_SIZE 256
const char sine_table[SINE_TABLE_SIZE]={
  127, 130, 133, 136, 139, 142, 145, 148, 151, 154, 157, 160, 163, 166, 169,
  172, 175, 178, 181, 184, 186, 189, 192, 195, 197, 200, 202, 205, 207, 210,
  212, 214, 216, 219, 221, 223, 225, 227, 229, 230, 232, 234, 236, 237, 239,
  240, 241, 243, 244, 245, 246, 247, 248, 249, 250, 250, 251, 252, 252, 253,
  253, 253, 253, 253, 253, 253, 253, 253, 253, 253, 252, 252, 251, 250, 250,
  249, 248, 247, 246, 245, 244, 242, 241, 240, 238, 237, 235, 234, 232, 230,
  228, 226, 224, 222, 220, 218, 216, 214, 211, 209, 207, 204, 202, 199, 197,
  194, 191, 189, 186, 183, 180, 178, 175, 172, 169, 166, 163, 160, 157, 154,
  151, 148, 145, 142, 138, 135, 132, 129, 127, 124, 121, 118, 114, 111, 108,
  105, 102, 99, 96, 93, 90, 87, 84, 81, 78, 75, 73, 70, 67, 64,
  62, 59, 56, 54, 51, 49, 46, 44, 42, 39, 37, 35, 33, 31, 29,
  27, 25, 23, 22, 20, 18, 17, 15, 14, 12, 11, 10, 9, 8, 7,
  6, 5, 4, 3, 3, 2, 2, 1, 1, 1, 1, 1, 1, 1, 1,
  1, 1, 2, 2, 3, 3, 4, 5, 5, 6, 7, 8, 9, 11, 12,
  13, 14, 16, 17, 19, 21, 22, 24, 26, 28, 30, 32, 34, 36, 38,
  41, 43, 45, 48, 50, 53, 55, 58, 60, 63, 66, 69, 71, 74, 77,
  80, 83, 86, 89, 92, 95, 98, 101, 104, 107, 110, 113, 116, 119, 122, 125
};

void main()
{
  int i;

  setup_adc_ports(NO_ANALOGS);
  setup_adc(ADC_OFF);
  setup_spi(FALSE);
  setup_counters(RTCC_INTERNAL,RTCC_DIV_1);
  setup_timer_1(T1_DISABLED);
  setup_timer_2(T2_DISABLED,0,1);
  setup_comparator(NC_NC_NC_NC);
  setup_vref(FALSE);

  while(true)
  {
```

```
    for(i=0; i<SINE_TABLE_SIZE; i++)
    {
      output_b(sine_table[i]);
      output_high(PIN_A0);
      //delay_ms(500);
      while(!input(PIN_A1))
      {
        output_b(sine_table[i]);
      }
      output_low(PIN_A0);
      //delay_ms(500);
      while(input(PIN_A1))
      {
        output_b(sine_table[i]);
      }
    }
  }
}
```

D1.4 Testing Plan and Results

After the data acquisition system was verified to work correctly as stated in the report, the hardware data stream generation system was tested by sending data through the USB and verifying that the correct signal was received.

D1.5 Conclusions

The Hardware Data Stream Generation System worked perfectly and was an invaluable testing and integration tool, simulating the entire hardware system from the A/D converter down.

D1.6 Recommendations

Due to the solid success of digital data generation using this system. It is recommended that this or a similar setup be used during development of future systems, allowing for a stable data input platform to be used while a final version is being developed.

Appendix E: Progress Reports

Appendix E2.1 Progress Report #1

1 Introduction

With the availability of the USB 2 interface chips, and high-speed A-D converters, there is a greater availability of standard engineering instruments implemented as USB based computer peripherals. This means that the hobbiests and engineering students will want such tools for their home computers to enable simple waveform evaluation. There are some inexpensive oscilloscopes available to be used as a computer peripheral, most of what is available now is around 150.00-200.00, but only operate in the 200kHz range. A small group of engineering students is designing a higher frequency oscilloscope. This oscilloscope would be connected to the desktop, or laptop computer, and use the computer screen to display signal information electronics students and hobbiests working on home projects would use it. The oscilloscope would be able to process signals up to the 500 kHz range, while maintaining a low cost. It would use the high speed USB port to connect to the computer. Most of the functionality will reside in the software, so future feature upgrades would only involve the installation of new software modules. This document describes the plan of action for the realization of the proposed oscilloscope. The separate layers of the design are described, followed by the design decisions made up to this date. The proposed development environments are listed and the project specifications are defined. An anticipated time-line for the project is presented, and the division of labour is described.

2 Layers

The project will be divided into three main layers: hardware layer, firmware layer, and software layer. The hardware layer must condition the input signals, and convert them into a parallel data stream for use by the USB controller chip. The controller chip is in the Firmware layer. The USB chip communicates with a PIC that will control the gain and frequency of the oscilloscope. It was decided to use a PIC not a Programmable Gate Array chip to limit the boards to two layers as we plan to manufacture the boards ourselves. The data will be converted by the USB chip into a USB data steam and sent to the PC. The PC has the software layer. The data will be displayed using signal-processing routines in MATLAB. The software layer also includes GUI interface to set the controls for the gain and frequency. The following diagram shows the layers and connections of the proposed oscilloscope. The second diagram shows the proposed software architecture. And a chronological development plan.

Figure 29 Oscilloscope Architecture Layers

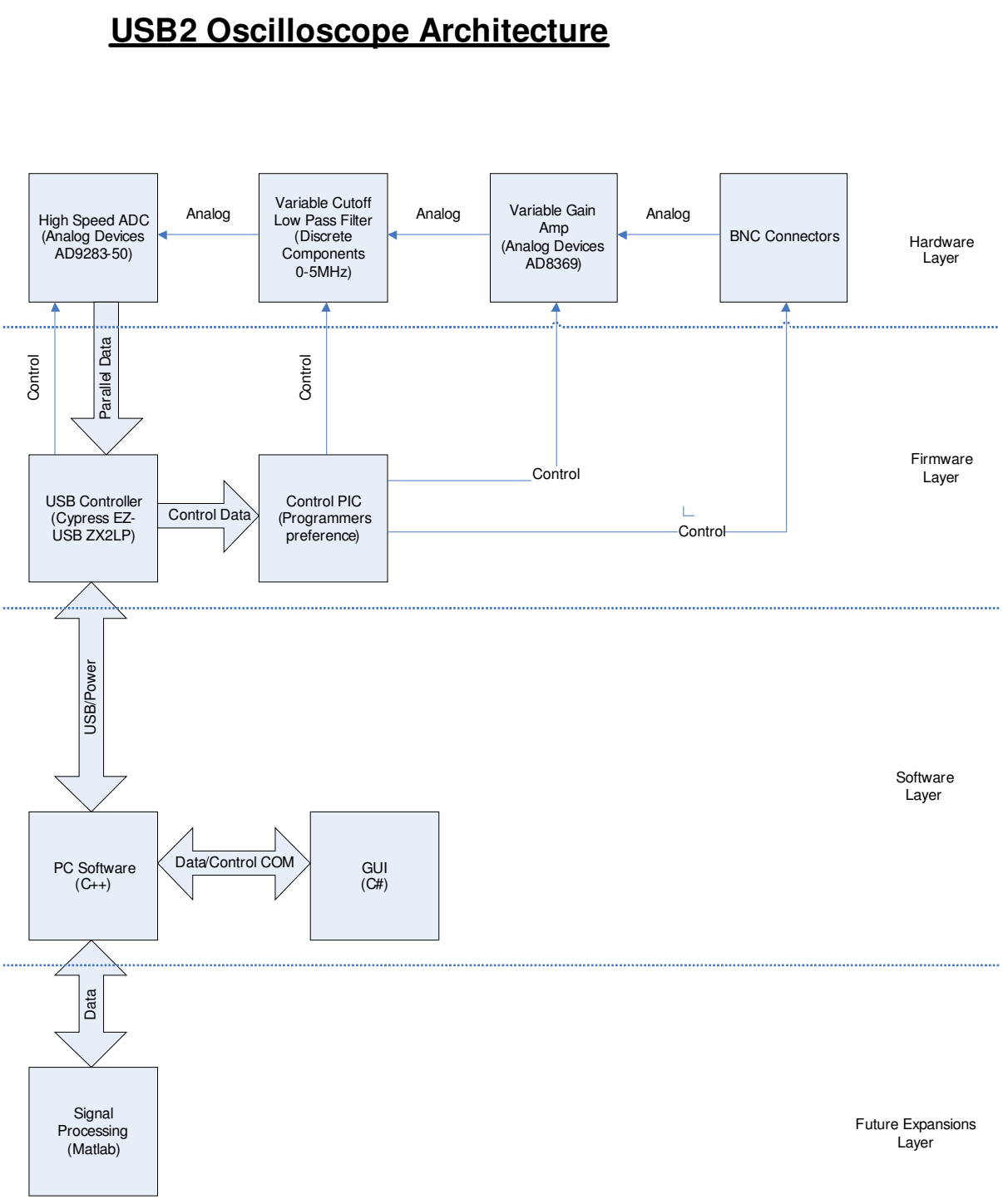
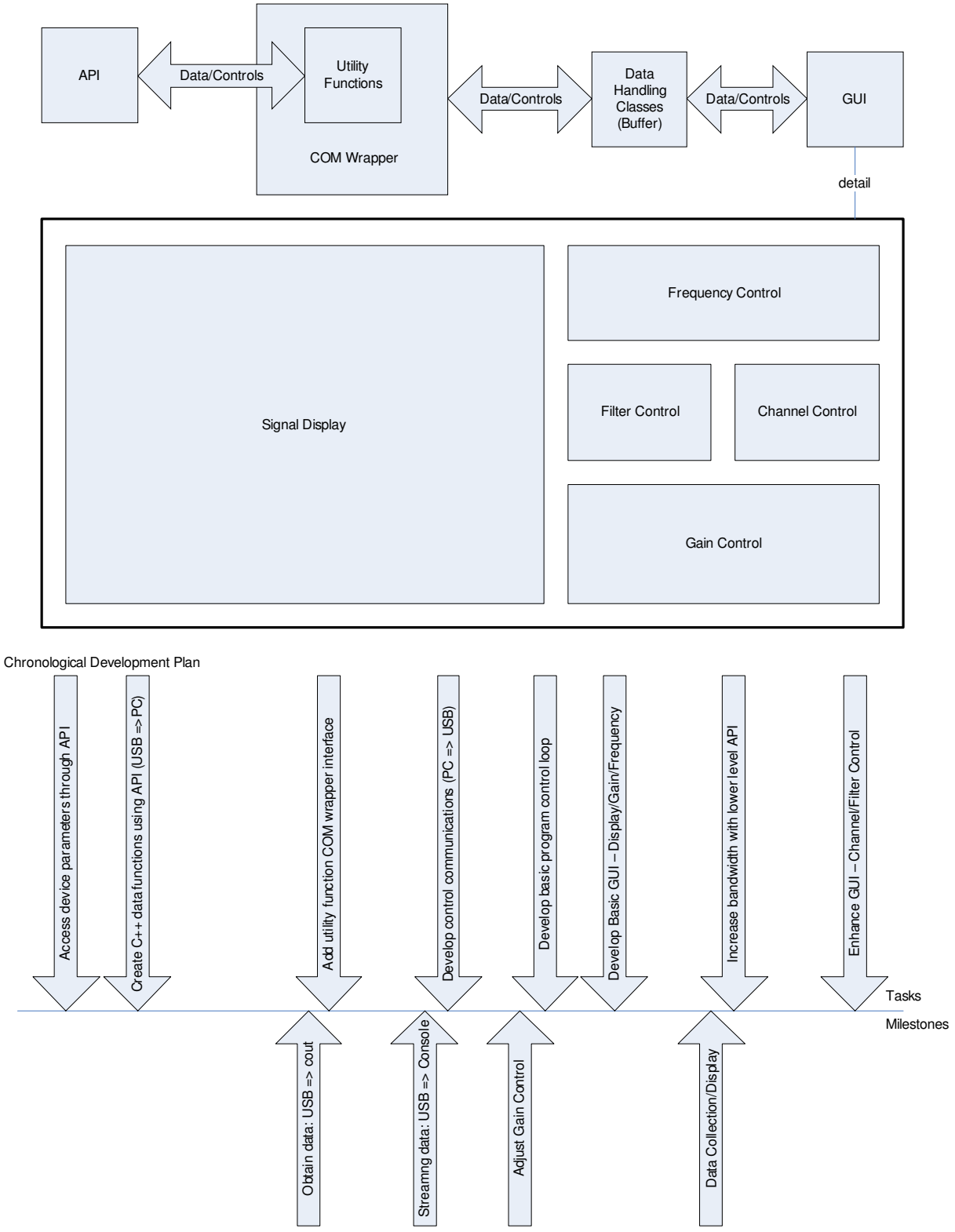


Figure 30 Software Architecture



3 Design decisions

As previously stated, the controls for the oscilloscope will be handled with a PIC device so the board can be a two layer board, and will be manufactured by one of the team members. This will eliminate

the need to send the work to a third party, and eliminate possible delays from that source. This decision may be revised if deemed necessary for the peak performance of the hardware chips. Since the scope is to work in the 500kHz range, the amplifier circuit will require a faster than average op-amp. After some investigations it was decided to choose one in the 5-20 dollar price range with a fairly high gain bandwidth rating for the high frequency filter. The other filters can use cheaper op-amps. The documentation on the A/D chip claim operation can be achieved with a single ended source, but concede there are benefits to driving it with a well-balanced input. A dual op-amp chip should provide a closer match for each input than using op-amps on separate chips. A butterworth filter should provide the required characteristic of the LP filter. Only two or three poles should be required. It was decided to have at least two boards manufactured. This provides a back-up in case of any problems, and two samples for the team members to work with. It was decided to take advantage of the power supplied on the USB line and a switched regulator for voltage supply to the hardware chips.

4 Platforms/Development environments

4.1 Hardware Layer

The hardware layer development will primarily include the circuit design using discrete components and integrated circuits, as well as a circuit board design. Being an industry standard, Protel will be used for design of the hardware layer, allowing the circuit and PCB to be designed in a single integrated software suite.

4.2 Firmware Layer

Based around the Cypress CY7C68013 USB 2.0 transceiver chip and a PIC controller, the firmware layer will have two development platforms. The USB transceiver chip will be developed using the available development drivers through the USB port PIC development will be done using the MPLAB free student edition available from MICROCHIP. MPLAB is a well-known tool for programming PIC controllers and as it is available at no cost to students so it was decided to take advantage of this powerful tool set.

4.3 Software Layer

Software for the USB oscilloscope will also be developed in two parts, although both parts will be developed in Microsoft's Visual Studio. The first part will be the C++ API interface to the USB driver and COM wrappers. These will be developed in C++, to be compatible with the API, as well as to offer COM access to the data stream buffer, giving future expansion and upgrade modules an easy tie-in point. The part of Software development will be the C# oscilloscope GUI. C# was chosen because it's graphic form designer tools far outstrip the C++ MFC libraries in terms of usability, scalability, and development time frame.

5 Performance Specifications

5.1 Input Voltage:

Max input voltage +\- 2.2V.

5.2 Input Frequency:

Max input frequency: 500 kHz.

Min input frequency: 16 kHz

5.3 Input Sample Rate:

1 MHz

6 Anticipated time-line

When the team met for the first time a preliminary time-line was constructed:
Jan 26: All parts ordered.
Feb 02: Software development started.
Feb 23: Prototype assembled.
Mar 02: Prototype testing started.
Mar 09: Prototype adjustments and testing completed.
Mar 16: Prototype completed and debugged. Final document started.
This schedule will leave time for adjustments if needed and should have the project completed well within the required time for the 499 project course.

7 Team members/division of labour

The project team is comprised of four 4th year electronics-engineering students:
Aaron Willis
Darcy Metz
Brian Walts
Leah Dickinson
The labour has been roughly divided according to the layers. Leah and Brian are working on the hardware. Brian will be manufacturing the boards. Aaron and Darcy are working on the software. Brian with Leah contributing some parts will head the firmware. Leah will be doing most of the documentation. Darcy will construct the web page with contributions from all team members.

8 Progress

At this early date into the project the basic architecture has been decided upon. Some of the initial design decisions have been made, including the development environments. Some initial decisions may be revised at a later date. Most of the labour has been divided between the team members. The main hardware components have been chosen and the schematics diagrams have been started. There has been some further discussions on the issue of board layers and it will probably take another meeting until this is finalized. A project schedule has been constructed, and weekly project meetings have been arranged. Due to the time commitments, there has been no personal meeting with the project supervisor yet, but it is hoped that one will be possible in the near future. During the meeting this week Brian and Leah should be able to finalize the parts list for order, and the team should make a final decision about the boards.

9 Conclusion

This engineering tool will be designed to be inexpensive, and work in the higher non-RF frequency range. It will be designed to be used by the engineering students, and by the interested hobbiest who cannot afford to purchase a separate oscilloscope for waveform evaluation purposes. It will take to three months to realize. The realization will be divided into three layers: hardware, firmware, and software. Most of the initial design decisions have been made and the work has been started. A project schedule has been constructed with some time allowed for errors and debugging. The labour will be

divided between the four-member team with Brian and Leah working on the hardware and firmware, Darcy and Aaron working on the software. Leah will be in charge of documentation, and Darcy will head the web-page efforts. At the end on March the team will demonstrate an inexpensive high frequency oscilloscope to be used as a USB peripheral device.

10 PROJECT SUMMARY

10.1.1 Project #3 USB based engineering instrumentation

Novice Engineering Research Devices
Contact: Darcy Metz dmetz@uvic.ca

Team Members:
Aaron Willis: awillis@uvic.ca
Darcy Metz: dmetz@uvic.ca
Brian Walts: bwalts@uvic.ca
Leah Dickinson: ldickins@uvic.ca

Faculty Supervisor: Paul Kraetner: pkraeutner@shaw.ca

The team is proposing to design and build an oscilloscope that will plug into the USB 2 port of any computer, and use the computer screen to display the relevant signal information. This is to be an inexpensive oscilloscope that will work up to the 500 kHz range. It is meant for small signal evaluation. The preliminary model will display a picture of the input waveform. Once the hardware is in place, future expansion will allow spectrum content to be displayed. The project has been divided into three specific layers to be worked on now, and a future software layer to be worked on later. The labour has been divided roughly into the three layers. The hardware, or physical layer will take advantage of the available power line on the USB port, so no external power source will be necessary. The interface will use a USB interface chip to keep hardware component count down, and consequently the cost of the oscilloscope will be dept low. Most of the functionality will reside in the software; future feature upgrades would only involve the installation of a new software module. Future upgrades will include spectrum analysis modules.

Appendix E2.2 Progress Report #2

1 Progress report #2

1.1 Hardware:

The Hardware side has had a slow start, the variable Digital to analog chip chosen was deemed inappropriate due to the band pass filter on the input. The original A/D was also discarded due to the low input voltage range, there was concern that noise would be come a factor. Installing the devices in the protel library is taking considerably longer than anticipated.

It was decided to use the suggestion to build two separate boards: one with the USB chip, and an A/D chip, called the high speed board. This will be a 4 layer board, and must be ordered so it is the one with more critical timing. At the meeting on Wednesday the progress was presented to all team members, and after some discussions it was decided that some changes were required. The board lay-out is almost complete for this board, and it should be ordered within the week. A theory of operation document for this first board will also be started within the week. Depending on the turnaround for the board and the parts, it is hoped that the board will be assembled on the weekend of March 17/18, with troubleshooting finished by mid-week. This should allow software testing with the USB board to begin on March 22. The second board will have the VGA, the anti-aliasing filter, and line conditioning. It will be two layers, so a team member can manufacture it. Work on this board will be placed on a lower priority, as the USB board is needed for the software to be implemented.

The second board may not be completed before the deadline however the design will be completed before that time, but the high speed board will be available and may be connected to a signal generator for demonstration purposes.

1.2 Software:

The software portion of the project has been broken down into several distinct modules. These include the USB interface/buffer library, graphing class library, custom controls library, and graphical user interface. Also in the software section is the GPIF setup for the Cypress USB chip and the web site implementation.

The USB interface/buffer class library is has been completed to a stage where it is operational, pending testing of the USB communication. Once the A/D and

USB sampler circuit board is complete this testing will commence. There is a known performance issue that has been noted with this class library. Right now the APIBuffer class contains a generic NERDBuffer class as its buffer with interfacing functions. This will be changed to the APIBuffer extending the

NERDBuffer, saving extra stack and pointer reference for every sample obtained. This is a medium priority, and will be tackled once the other aspects of the system are in working order.

The graphing class library is essentially a collection of classes for high speed graphing of points and functions in a picture box on a Windows form. The preliminary technological hurdles have been overcome in this area, as a system of graphing a line between two points has been established, as well as a method of erasing them for overwriting. Darcy is currently working on finishing this library as a modular class with a simple interface.

Our library of custom controls will mainly consist of rotary dials to imitate the interface of a standard oscilloscope. Aaron has made some progress in this respect, interpreting the mouse down and mouse up events to allow the user to set a value by clicking and dragging. There will also be a right click component to the control, bringing up a second Windows form for precision adjustments. Aaron is working on finishing the custom controls library.

The graphical user interface is the actual application to run for the oscilloscope. It will tie all of the components together. Almost all of the design work has been done for the main interface, pending completion of the graphing and controls libraries. It consists of a form showing the controls as well as a graphical display of the signal read in by the USB connection. There will also be a full-screen mode to maximize the signal display for easier viewing.

The basic system requirements for the GPIF and their effects on the hardware design have been determined and passed on to the hardware design team. As the documentation is somewhat limited and unclear, the details of the GPIF set up, as well as the protocol, will be somewhat of a trial and error process, and so is pending the completion of the A/D and USB circuit board.

The web site of the company is almost complete. The backbone structures, as well as the company information pages are all complete. The oscilloscope section is merely awaiting final design specs and testing benchmark results.

In short, all sections of the software design and implementation are well on their way to being finished. All major foreseen hurdles overcome with the exception of the GPIF setup. There are a couple of areas awaiting hardware for final testing and completion, but most are making steady progress.