**Digital Stimulus Monitor Unit**

ECE 682Y

Spring 2008

**ABSTRACT**

The proposed Digital Stimulus Monitor Unit (DSMU) aims to support Dr. DeGroat’s embedded system lab with portable hardware and user friendly software.  Eight input/output programmable channels can accommodate up to eight digital signals simultaneously.  The specifications state the DSMU is to generate and handle signals up to 5MHz.  This includes generated signals and input signals.  In addition to generating clock signals, the device is also capable of generating complex data signals that are specified by the user.  The user interacts with the device by using a Graphical User Interface (GUI) on a Personal Computer (PC.)  Furthermore, all communication with the device takes place over a USB connection.  The data is buffered and displayed on the GUI with a maximum delay of 15-20 seconds.  Upon completion, the DSMU was able to successfully generate and accept signals up to 12 kHz.  These signals were tested using an oscilloscope.  However, issues arose when trying to display these signals on the GUI.  Thus, the DSMU behaves properly but is not able to display the correct data to the user.

**Table of Contents**

**[1] Introduction**

**[2]** **Requirements and Constraints**

**[3]** **Preliminary Design**

3.1- Preliminary Work

3.2- Communications

3.3- Software

3.3.1) Graphical User Interface

3.3.2) USB Interface

3.4- Hardware

3.5- Firmware

3.6- Transitional Design Considerations

3.7- Project Cost Estimation

3.7.1- Component Costs

3.7.2- Additional Resources and Facilities

3.7.3- Personnel Salaries and Totals

**[4]** **Final Design**

4.1- Software

4.1.1) Software Developers Guide

4.1.1a- Overview

4.1.1b- Development Method

4.2- Hardware

4.2.1) Development Board

4.2.2) Microcontroller

4.2.3) External Components

4.3- Firmware

4.3.1) Introduction

4.3.2) Code Structure

4.3.2a- Overview

4.3.2b- Configuration

4.3.2c- Input /Output

4.3.3) USB Protocol

4.3.3a- The Header

4.3.3b- Control Packets – Start and Stop

4.3.3c- Configuration Packets

4.3.3d- DDR Packets

4.3.3e- Complex Data Table Packets (CDT1 and CDT2)

4.3.4) USB Code

4.3.4a- USB Specification and the Microchip Firmware

4.3.4b- Structure of the Generic Class Firmware Example

4.3.4c- Modifications of the Example Firmware

4.3.4d- Testing Process and Driver Usage

4.3.5) Timers & Interrupts

4.4- Project Costs

**[5]** **Results**

5.1- Achieved Specification

5.2- Current Issues

5.3- Possible Future Improvements

**Appendix**

A1- Software Source Code

A1.a) Application Form Code

A1.b) Data Display Code

A1.c) USB Interaction Protocol Code

A1.d) Device Encapsulation Classes

A2- Firmware Source Code

B1.a) Main.c

B2.a) User.c

C3.a) User.h

**[1] INTRODUCTION**

A proposed prototype of a Digital Stimulus Monitor Unit (DSMU) requested by Prof. DeGroat is presented in this document. Included are the introduction, design strategies, resources, project cost, schedule, and appendix. The introduction consists of a concise outline of the report and a brief description of the device with the finalized specification. In the design strategies section a thorough explanation of the design is discussed in terms of hardware, software, and firmware, and an overall view of how these components fit together is clarified, as well as a description of transitional design elements and revised constraints that were considered. The project cost section includes an initial estimate of the overall project cost in terms of personnel compensation and parts cost for mass production along with their justifications, as well as a breakdown of the actual costs incurred for the prototype. A detailed breakdown of the results can be found in the results section. All necessary appendices are included at the end of this document.

**[2] REQUIREMENTS AND CONSTRAINTS**

The proposed Digital Stimulus Monitor Unit (DSMU) aims to support Dr. DeGroat’s embedded system lab with portable hardware and user friendly software. Eight input/output programmable channels can accommodate up to eight digital circuits simultaneously. Signals in the range of 0 to 5 volts or 0 to 3.3 volts can be sent or received at a maximum frequency of 5MHz. Meeting the 5MHz maximum frequency is the main challenge for the design due to the transfer rate limitation of the Universal Serial Bus (USB) connection and capability of affordable hardware. The device is able to receive data using a real time or buffered method. Real time input is able to run for an arbitrary period of time without delay but with a cost of lower input frequency. The buffered input induces a delay and can only operate while the onboard random access memory (RAM) is not full, but greatly improves maximum input frequency. Pulse Width Modulation (PWM) can be used for data output. These output signals can be constructed within a configuration application using user defined parameters, and the maximum output frequency is solely dependent on the microprocessor clock speed. It is the team’s overall goal to deliver the best product possible within the allowed budget and time constraint.

Physical Properties Aims:

1. Device dimensions should be approximately 2 to 3 inches by 3 to 5 inches by 1 to 2 inches.
2. The enclosure of the device should be plastic or metal.

Communication Aims:

1. To be able to receive data in real time or buffered mode
2. To be able to output with Pulse Width Modulation (PWM) mode
3. The device should communicate with software application by means of a Universal Serial Bus (USB) port which must be connected to and configured correctly.
4. The device should have eight programmable input / output channels.

Software Aims:

1. Create a single easy to use Windows application that allows the user to both configure and operate the device.
2. The application should communicate with the device by means of a USB port which must be connected to and configured correctly.
3. Provide the ability for the user to select either a buffered input mode with the fastest possible operating speed but a limited time for data collection, or a real time input mode with an arbitrary data collection time and a slower collection rate.
4. Visually display the collected data in an easy to understand format.

Hardware Aims:

1. Power should be supplied through USB or small DC wall adaptor
2. LED to indicate power on/off on the surface of the device
3. Device should have banana jack receptacles on the surface.
4. Development using the PIC18F87J50 with additional 4Mbits additional RAM

Cost aims:

1. Target price <$80, development <$120

Environmental aims:

1. RoHS compliance

**[3] Preliminary Design**

The design of the DMSU is divided into two major blocks, software and hardware/firmware. The software block is encapsulated by the host PC and the hardware/firmware block is encapsulated by the DSMU device. Each design block has individual subcomponents that serve various purposes as shown in the system diagram in Figure 3.1. Communication between the two blocks is achieved through USB protocol available on both blocks.

The main function of the software block is to provide interaction with the user through a host PC. This is made possible by a graphical user interface (GUI) that enables the user to configure outgoing data through the user input devices (keyboard & mouse) and receive visuals of incoming data via the graphical output device (computer monitor). The software determines the data transmission mode and input/output direction based on the user input, and initializes the transmission sequence.

The major subcomponents of the hardware block consist of the microcontroller and additional RAM. The PIC18F87J50 microcontroller is selected for its convenient onboard USB port, dedicated addressing bus to external RAM, its superiority in clock speed, and onboard memory size over other models of its class. The additional RAM supports the buffer input mode by serving as a temporary storage for the input data before being displayed on the GUI. As shown in Figure 3.1, the eight input/output programmable channels are connected to eight input/output ports of the microcontroller, and the information on direction of data transfer is provided by software through USB. The microcontroller is driven by the firmware stored in the onboard memory. The firmware configures the microcontroller based on the information it receives from the software block. Based on the information it receives, the firmware configures the input/output LED indicator and manage the data transfer accordingly.



*Figure 3.1: System Diagram*

The operation of the DSMU can be summarized in the Figure 3.2 state diagram. The initial state of the device is POWER OFF and the device is not operational. As soon as the device is connected to the host PC via USB, the device is in now in IDLE state. In this state, the device is ready to take user input and all hardware is powered on. As soon as the user presses the start button, the device advances to the INITIAL SETUP state where data input/output direction and LED indicators are set up. When this is completed, the device proceeds to the DATA INPUT/ OUTPUT state where the data is received or transmitted based the user configuration. In this state, the user can view the data if output mode or real time transmission mode is being used. When the data transfer is completed, the device returns to IDLE state. User using the buffered input mode is now able to view the data on the GUI. Disconnecting the USB connection during any of the state returns the device to POWER OFF state and may result in loss of data and unsuccessful data acquisition. The device returns to IDLE State if the reset button is pressed in the INITIAL SETUP state and the DATA INPUT/OUTPUT state.



**Figure 3.2:** Operation State Diagram

**3.1) Preliminary Work**

Most of the preliminary work for this project is based on the two input and output modes, buffered and real time. First research was conducted to determine the microprocessor used in the Digital Stimulus and Monitor Unit. Speed and communication ports were the main deciding factors for this decision. The PIC18F87J50 was chosen because it had a USB controller built on, a dedicated RAM bus and it is one of the fastest microprocessors with these options.

It is estimated that in the real time mode 120 instructions are needed to input and output data, build an output packet to be sent via USB and send the packet through USB. Since the processor runs at 12 mega instructions per second (MIPS), this allows for a sampling frequency of 100 kHz. Also, for buffered input, it is estimated that 60 instructions are needed to input the data, output the data, and store the data to the onboard RAM. This allows for a sampling frequency of 200 kHz.

A preliminary graphical user interface (GUI) has been drawn up and is shown in Figure 3.3. An overall design and flow of the source code for the PC software and microcontroller firmware has been created. A more detailed structure and program flow is outlined in the Software and Firmware sections.

**3.2) Communications**

The input of data can be handled in one of two modes, either a real time mode or a buffered mode. Each offers a variety of pros and cons which are discussed in more detail in the following descriptions.  
 The real time mode operates by getting data from the input ports and transferring it to the host PC to be processed all in the same step. This type of operation allows for near instant display of incoming data which gives the user the ability to monitor a signal as it fluctuates between states. The downside of this mode is that it operates at a much slower rate than other option as it is bottlenecked by the transfer speed limitations of the USB communication line. The maximum speed for USB communication using our selected PIC microprocessor is 12Mbit/sec. This corresponds to an ideal maximum transfer rate of about 1.5MHz, which itself is well below the desired 5MHz transfer rate. In addition, keep in mind that 1.5MHz is an ideal rate assuming that it’s possible to pump all the input channel’s current contents out of the USB line at the absolute fastest speed possible. In real application, though, this is not truly attainable. This results in a realistic performance expectation well below even only 1.5MHz.   
 On the opposite end of the spectrum from real time mode is buffered input mode. The basic operation of buffered mode is to take the contents from the input ports as fast as possible and store them into the off board RAM attached to the PIC Microcontroller. Because the memory modules the group has selected have a read/write speed of approximately 10 nanoseconds it is expected that data is able to be written at speeds of up to a theoretical maximum of 100Mhz. This number is again an ideal maximum, but it provides a much greater potential for high speed data collection compared to real time mode. The downside to this mode is that there is a delay between collecting data and reporting it to the host PC. In addition to this data can only be collected until the off board RAM is full at which point the collected data must be transferred to the host PC. This limitation means that buffered input mode can only operate for short bursts of time, but can work much faster than real time mode.   
 All output is handled using PWM. This means that instead of setting up a string of raw data to output, all output data in set up in time intervals. For example, a set of output data could be a simulated clock signal with the output port being held in the high state for a set period of time followed by a period of low state for a different amount of time. This avoids the reliance on operating frequency (which is dependent on which input mode is selected), and makes it possible to create output data for arbitrary lengths of time. This would not be feasible if output was being handled on a bit-by-bit basis.

**3.3) Software**

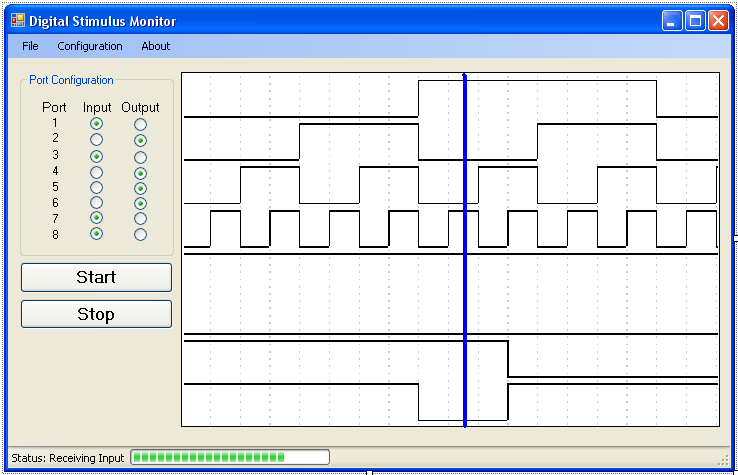
The group has chosen to develop the software for the host computer using the Visual C# language in the Microsoft Visual Studio 2005 design environment. C# was chosen as the language because of its ability to provide ease of use in creating robust Windows applications while still maintaining fast, efficient performance. By eliminating the construct of pointers from the C++ language, C# greatly simplifies the ideas of polymorphism, data input and output, and especially large data structures. Instead of having to use an external library such as the Standard Template Library (STL) or Microsoft Foundation Classes (MFC) to get data structures such as lists and dynamic arrays they are built directly into C# as base data elements.

The group has several members with a large amount of experience in the Microsoft Visual Studio design environment which helps streamline the software design and implementation process. Visual Studio provides the easiest method available for creating Windows applications in its “WinForm Designer”. This tool makes creating an application a matter of just dragging and dropping buttons, menu items, text boxes, etc. onto a form, and then writing the code that describes the functions of these elements relevant to the application. This takes the task of creating the graphical side of the user interface out of the group’s hands and eliminates the need to deal with difficult programming interfaces such as Win32 Dynamic Link Libraries (DLL’s) and COM objects, which in the past had made creating Windows applications a messy, arduous task.

**3.3.1) Graphical User Interface**

One of the main obstacles in the software implementation is communicating over the Universal Serial Bus (USB) port of the host computer. This normally involves low-level calls to Windows DLL’s and very application specific, non-generic code. To get around this a C# wrapper for those Windows DLL calls is being used. This library has been created previously by a third party and is freely available online under the GNU Public License (GPL). It is known as USBSharp and it makes it possible to connect directly to a USB device and talk to it much as one would talk to a serial device. This USB communication library makes it possible to accomplish Software Aim 2 in the time allowed for the design of this device.

All input/output channels of the device are software configurable to behave as either inputs or outputs. This is handled by providing a radio-button interface in which each of the eight ports is enumerated with a choice of either input or output. Upon this selection being made a configuration message packet is sent to the microcontroller to set up the communication port in the desired manner. Output is set up using either a predetermined signal pattern, such as a clock signal, or by using a data file containing a stream of information to be placed on the output ports. These controls are accessible via a graphical user interface (GUI), a preliminary design of which can be seen below in Figure 3.3. This design of a Windows application satisfies Software Aims 1 and 4 by combining both the configuration and operation of the device into one easy to use application, and also by formatting the output in a simple visual style that is easy to interpret.



*Figure 3.3: Preliminary Digital Stimulus Monitor GUI*

**3.3.2) USB Interface**

Software Aim 3 requires the device to perform in both a high speed buffered input mode and lower performance but quicker throughput real time mode. To accomplish this, the operation of the host PC software with regards to controlling the input from the microcontroller works in two distinct ways. First, there is a real time mode in which the data retrieved from the microcontroller is communicated to the host PC over the USB connection at the same time that it is received. The basic outline of this operation can be seen below in Figure 3.4.



*Figure 3.4: Flowchart for the Host PC software operating in real time input mode.*

As can be seen in the figure above the initial step is to first connect to the USB device. This is accomplished by either having the user select the device from a list which enumerates all available devices, or by automatically acquiring a connection to the device. The user then must select which ports he or she would like to behave as inputs and which should behave as outputs. Once this is determined, the user presses a start button which sends a configuration packet over the USB cable to the microcontroller.

Upon receiving this packet of data, the microcontroller proceeds to enter a data acquisition / transmission loop in which it collects a byte of data from the eight data ports, then transmit this data back to the host PC. The host PC waits for a new packet of data, and on receiving it displays it on the graphical user interface. This operation continues until the user presses a stop button which in turn sends the microcontroller a packet which informs it to stop collecting and sending data.

Once the microcontroller returns to its idle state, the application asks the user if he or she would like to create a log file of the preceding events so that they may save the results for later usage and analysis. If the user selects yes then a log file is created. Once this is accomplished successfully the application is ready to either be run again or to be closed.

Operation in buffered input mode behaves similarly as above until the microcontroller is configured. The flowchart for buffered input mode operation can be found below in Figure 3.5.



*Figure 3.5: Flowchart for the host PC software operating in buffered input mode.*

Once the microcontroller is configured it begins collecting data. It continues to collect and output data until its memory buffer (which is around four megabytes) is full. Once the buffer is full the microcontroller sends a packet back to the host PC notifying it of this event. Upon receiving this event, the host PC then is transferred the contents of the buffered input which are then displayed on the graphical user interface. Similar to real time mode, the user is again asked if they would like to create a log file. If he or she selects yes then a log file is created for later analysis, and the application is ready to either be run again or closed.

**3.4) Hardware**

The microcontroller is the most important piece of hardware for the correct implementation of the device. To meet Hardware Aim 4 the microcontroller has been heavily researched to assure the fastest and cheapest final product. The microcontroller that is chosen for the product is the PIC18F87J50. The PIC18F87J50 microcontroller is a 16 bit 80 pin high performance 1Mbit flash USB microcontroller. The on board RAM contains 4kbytes and the program memory size is 128kbytes. The microcontroller can handle up to 48MHz which is much higher than the project specifications of 5MHz. The PIC18F87J50 microcontroller can handle the specified 0 to 5V and 0 to 3.3V input which helps reduce the need for extra hardware or circuitry. The microcontroller can handle a full speed of 12Mbits/s USB data transfer rate. One of the most important features of the microcontroller is the option to add external memory for up to 2Mbyte address capability. This extra external memory helps when trying to buffer data over an extended amount of time. Another important aspect of the PIC18F87J50 microcontroller is the price. The volume price of this microcontroller is $3.62.

For development purposes the PIC18F87J50 microcontroller comes in a kit with a development board which has many features of its own that are similar to the expected final design. The inexpensive kit comes with development software which allows development of the device without an additional cost for the software. All of the components that are used for the implementation of the final product are included on the development board except for the external RAM. These same parts are used for the final design because we are assured that all of the parts are compatible with each other since they already come on a functional development board.

The development board has a many features that are similar to our final project. The similarities reduce development cost and help evaluate the final product. Although there are only two LEDs on the development board they provide insight to the implementation of the final product which has two LEDs per I/O port. There are two push buttons on the development board. One of these buttons is a reset button which is needed on the final design. The other button is a general purpose button which is used as a data interrupt. Since the device works with both 5 and 3.3V the development board comes with a linear voltage regulator. This reduces the need for addition circuitry that many be needed on the final product if too much power is consumed. The development board has an attached USB mini-B connector. This connector makes it easier to change the code of the microcontroller and it is similar to the USB output that is needed in the final design. The connector also adds power to the device in the final product. The development board offers convenient test points for attaching current meters to test for correct power use. This development board is similar to the final product in size and helps estimate the overall size of the final product.

Additional RAM is used to increase the amount of data that can be buffered. The additional RAM that is used is the CY7C1049DV33-10VXI. This is a 4Mbit asynchronous SRAM. The SRAM type is used because it is compatible with the PIC18F87J50 microcontroller. SRAM is also found to be the fastest type of RAM that is compatible with the PIC18F87J50 microcontroller. The voltage that is used with this RAM is 3.3V. With the use of the voltage regulator that is already used on the development board no additional components of circuitry is needed to correctly power the external RAM. Also the USB connector has enough power to supply the RAM with the required voltage when implemented in the final product to meet Hardware Aim 1. The selected RAM works with the 20 bit addressing mode which is also compatible with the PIC18F87J50 microcontroller.

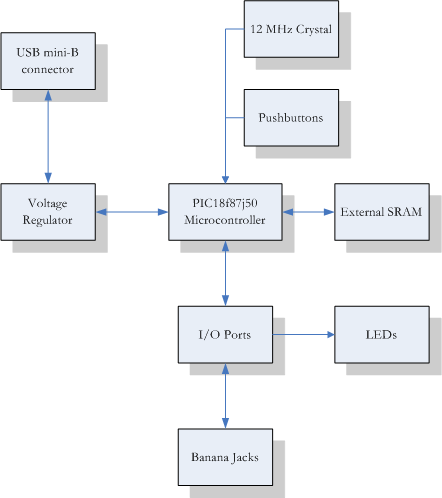
The LEDs that are used for the device need some additional wiring to function correctly. In order to meet Hardware Aim 2 each I/O port has two LEDs. One colored LED is used to determine if the device is receiving data and an LED of a different color is used to determine when the device is sending data. The direction of the current determines which LED is turned on. The circuitry for this is shown in Figure 3.6.



***Figure 3.6:*** *LED Wiring*

The 12MHz clock frequency that is used on the development board is needed to correctly use the USB module. The module requires a 6 or 48MHz clock signal to work in either low speed or high speed. There is an internal clock that can control the 6 or 48MHz clock, but in order for this to be done, an external clock of 4MHz or an integer multiple of 4MHz is required. The 12MHz crystal is used as an overall timer which all other timers use to function.

The final product includes many similar parts that the development board has. These products are purchased separately from the development board to reduce production cost for the final product. In addition to the parts mentioned earlier some addition parts are also used. To meet Hardware Aim 3 banana jacks are used to connect to the I/O ports which is a required project specifications. The overall hardware design is shown in Figure 3.7. Some additional wire is also needed to connect various components of the device.

****

***Figure 3.7:*** *Overall Hardware Design*

**3.5) firmware**

Since the PIC18F87J50 microcontroller is supported by a C to Assembly compiler, the firmware for the microcontroller is written in C. In this case, the C programming language has been chosen over Assembly due to team member familiarity and its user friendly nature. Since the firmware main task is to deal with the communication between the software and the I/O ports, the firmware implementation must help satisfy nearly all the communication and software aims.

To satisfy Communication Aim 3 and Software Aim 2, the firmware communicates with the software using the USB cable connected to the host PC. Using this connection the firmware also receives and deals with a configuration packet sent by the software, which allows the firmware to fulfill Software Aim 1 and 3 by enabling user configuration and operation of the unit. Based on that data the firmware also determines what I/O mode to operate under, and if necessary how to configure the required mode. To accomplish Communication Aims 1 and 2, the operating modes include real time or buffered input and PWM output. Once the configuration is complete the firmware begins receiving/transmitting data via the eight programmable ports and dealing with the data according to the operating I/O mode. Lastly, the firmware is in charge of returning the gathered input data to the host PC so that the software application can satisfy the Software Aim 4 by graphically displaying the gathered data. While this is a basic overview of the firmware, a more detailed version of the implementation is explained next and can be viewed in Figure 3.8.



***Figure 3.8:*** *Firmware Process Flowchart*

To control input and output from the Digital Stimulus Monitor Unit the microcontroller’s firmware operates in a variety of different states. The flow to and from those different states can be seen in flowchart in Figure 3.8. The flow always begins when the unit is connected to the host PC via the USB cable, then the microcontroller powers on and immediately enters the idle state. While in the idle state the firmware waits to receive an interrupt through the USB that says the user has clicked the Start button on the GUI. After receiving the interrupt the firmware begins setting up the I/O directions on the ports, using the Data Direction Register, based on the configuration packet that was sent through the USB right after the Start interrupt. Also, the correct LED is lit up on each port depending on if the port is inputting (Red) or outputting (Green). When the unit is outputting, the configuration packet also contains values for the output pulse signals. These values are used to set up the microcontroller’s timers to help determine when the output signals should be toggled.

After configuration the firmware enters the I/O loop. The next series of actions depends on what exactly the unit is trying to accomplish. The unit is capable of five different operating modes: real-time input and output, buffered input and output, only real time input, only buffered input, or only output. The firmware determines the operating mode through a series of if-statements demonstrated in Figure 9.

First the firmware determines if any of the ports and set to input by checking the Data Direction register for any bits set to 1. If a bit is set to 1 then that means the corresponding I/O port is set to input. If it is determined that the unit is in charge of input then the next step is reading the port values. Now the firmware needs to decide if the unit is in charge of real time or buffered input, which is determined by checking the configuration settings that came across in the configuration packet.

If the unit is in real time input mode the data packet is built and sent through the USB to be handled by the host PC. However, if the unit is in buffered input mode then the data is saved in the external memory. After the data have been saved, save address is checked to determine if the external memory is full. When the save address equals the last address in the memory then the memory is considered full. After filling the external memory, the firmware enters a loop that sends the buffered data to the host PC through the USB. Once all the buffered data has been sent the firmware returns to the idle state to wait from another Start signal.

If the unit is in real time input mode or buffered mode but the memory is not full, then the firmware decides if the unit is also in charge of output. This is determined by checking the Data Direction register for any bits set to 0, because a 0 indicates that the corresponding port is set to output. If the unit is in charge of output then the next step is to check and see if any of the timers have overflowed. If a timer overflows then the corresponding output signal is toggled. Once the output signals have been setup the signals can be sent to the ports.

Since the real time input mode is terminated by a Stop signal from the host PC the last step in the I/O loop is to check to see if a Stop signal has been received. If a Stop signal was received then the firmware returns to the idle state to wait for another Start signal, otherwise the I/O loop restarts. Also, if the unit’s USB cable is disconnected from the host PC then the unit immediately powers down, no matter the state of the firmware.

**3.6) TRANSITIONAL DESIGN**

During construction, several of the design constraints were revisited. The software team determined that initial design using the RAM for buffering would not significantly increase speed, and was therefore omitted from the final design. Also, some constraints were loosened as ports could only output at 3.3V. Therefore, all operation was reduced to 3.3V on the I/O lines. Also, the reset switch was deemed unnecessary by both the software and hardware teams and was removed from the final design and case.

**3.7) PROJECT COST ESTIMATION**

**3.7.1) Component Costs**

One of the most important aspects of any project is to consistently review developmental and production costs. The team has tried to use available financial resources as efficiently as possible, thus cutting costs while satisfying the customer’s requirement. One of the most important components of this project is the microcontroller. The PIC18F87J50 processor, which has been selected by the group, costs about $4.13 per unit for quantities less than 25. The processor is cost effective as it meets the customer’s needs and is economically priced. Depending on the quantities ordered this price can go down when considering mass-production costs. The most expensive item in the cost table (shown in Table 1) is the PIC18F87J50 FS USB Plug-In Module, which is a full-scale demonstration and development for the PIC microprocessor. It costs around $40.00. The development board, as the name suggests is required initially to program and test a prototype using the onboard PIC processor. Also, required is an external 4M Ram (SRAM – 512K x 8, asynchronous) which costs $5.10 for quantities less than 100. Various other expenses associated with the project such as costs of banana plugs, wires, etc. have also been noted in Table 1. Thus, the total hardware prototyping costs for the project have been estimated at $63.55. This amount is significantly lower than the aimed component cost of $120.00. However, the group was still able to utilize quality components that met and exceeded customer requirements.

**3.7.2) Additional Resources and Facilities**

Additional resources that are required by the team are a PIC microprocessor development kit (to add to our USB Plug-in module development kit), programming software for the microprocessor and for the computer GUI software; PC’s to assist with the programming, oscilloscopes to test output signals from the unit and soldering equipment and accessories. These items are available for the team in Dr. DeGroat’s Lab. The lab room is also to be used as lab space when needed by the team. Dr. DeGroat has kindly agreed to let us use her facility for the project. Thus, costs associated with such equipment and floor space are not included in the projected costs.

**3.7.3) Personnel Salaries and Totals**

The total project cost is however incomplete without calculating and including personnel salaries. According to the Engineering Career Services (ECS) website, annual basic pay for engineering graduates with a bachelor’s degree currently is about $56,288.00. This approximates to $28.14 per hour, assuming normal working hours and holidays. Also added to the basic pay are company benefits, paid leaves and healthcare compensation, thus bringing the hourly income to about $75.00. Assuming personnel salaries at $75/hour per person, and working hours for prototype development and testing to be around 72 per person (see Tables 2 and 3 for calculations), the total personnel costs for our six-person team amount to $32,400.00. The team will work on the project for a total of 432 hours. Thus, a total of $32,481.89 will be required by our team to implement, build and test the required prototype unit. Also shown in Figure 3.9 are costs associated with mass-production of this unit for future purposes. The cost per unit for mass-production of this unit as calculated in Figure 3.10 is $22.48, which is less than 50% of the production cost of $80.00 aimed at by the group. Two additional components under mass-production costs are the PIC18F87J50 processor and the Linear Voltage Regulator (LDO – 3.3V, 100mA). Since both of these parts are built-in on the prototype development board (PIC18F87J50 FS USB Plug-In Module), their costs are not included in the developmental costs’ table.

*Figure 3.9 Parts Cost*

*Figure 3.10 Mass Production Cost*

**[4] FINAL DESIGN**

**4.1) SOFTWARE**

**4.1.1) Software Developers Guide**

4.1.1a- Overview

The host computer software for the Digital Stimulus Monitor Unit (DSMU) was written in Visual Studio 2005 using Microsoft Visual C# and the .Net 2.0 Framework. To create the data display graph the ZedGraph (v. 5.1.2.878) free library was used. This combination of tools allowed for rapid, user friendly development of the graphical portions of the user interface. The reasons behind choosing these particular technologies are outlined below.

The most challenging part of the whole design was most certainly establishing a working Universal Serial Bus (USB) connection between the software and firmware. This was accomplished using the MPUSBAPI.dll driver library supplied by the manufacturer of the hardware used for the DSMU. Details of its implementation will be discussed below.

*Why Visual Studio:*

Through the past decade Microsoft has been creating various version of their development environment known as Visual Studio. Each version has made Windows application design more and more user-friendly. For development of the DSMU the most widely used version to date, Visual Studio 2005, was chosen.

Visual Studio 2005 provides many features which are handy to the developer. Merely by choosing to use this tool development time can be greatly sped up. This is accomplished by providing the developer with a graphical user interface to create windows applications known as the “Winform Designer”. By using this tool it becomes possible to create the appearance of Windows applications by merely dragging and dropping buttons, check boxes, text boxes, and even custom user controls onto a blank window. Once this is done, simple code can be written that details the interaction between these elements. All of this is layered on an event-driven protocol which allows button clicks and menu choices to be handled by simply “wiring up” the desired results with user interactions.

When writing code Visual Studio provides “Intellisense” that allows function, class, and variable names and keywords to be selected from a menu that conveniently pops up whenever the first few letters of the desired word are typed. This is perhaps the most effective addition in Visual Studio that allows rapid development. Instead of having to look up variable names from various files and classes they are displayed in an easy to find list anytime they are needed.

During debugging Visual Studio again excels. Breakpoints are easily set at the click of a mouse, and then once the code is stepped into it can be easily run through by using various keyboard shortcuts. Variables can be highlighted by the mouse, and their current value is displayed in a pop-up next to the mouse cursor. CPU usage can be observed in a simple graph which allows for quick elimination of memory leaks and other major bugs.

Due to all of these reasons it is easy to see why the Visual Studio design environment was chosen over the various other competitors. This decision was not regretted, as software development progressed easily and quickly.

*Why C#:*

The main concern when choosing a development language for the DSMU was one that would allow user-friendly, relatively bug free code to be written as quickly as possible. Code efficiency from the point of view of the host computer was not a pressing issue, as the DSMU would be operating at much lower speeds than that of a typical desktop computer. Because of these given constraints the Microsoft Visual C# language was chosen. It is the latest major contender in the circle of widely used development languages, and combines the best elements of C++ and Java to create a managed, pointer-free object oriented language that is still very efficient. C# uses a development model in which all parameters are handled by reference, unless otherwise noted. This effectively abstracts the archaic, bug-ridden idea of memory address pointers away and makes it much easier to create a large, robust data structure. During development C# proved to be a good choice for the tasks at hand. Development on the user interface progressed quickly and without any major hang-ups. The language was easily picked up by team members that had no previous experience with it, but still provided a detailed, fully functional end result.

4.1.1b- Development Method

*Data Display Graph:*

The majority of the development time on the graphical user interface was spent on creating the data display graph. This was accomplished by means of a custom user-control. An instance of the ZedGraph control was added to a blank form, and then modified to be more specific to the DSMU’s needs. Each IO channel is represented on the graph by creating a data list and giving it a Y-value corresponding to the channel number (i.e. Channel 3’s data is displayed at a base Y-value of 3 on the graph). This effectively created the look of having a display line for each of the eight IO channels. This user control (known as an EightChannelDisplay in the code) contained various methods to add blocks of data and set up which channels are inputs and which are outputs.

Listed below are some of the methods created for the EightChannelDisplay class, and descriptions of their operation:

* public void ResetDisplay()

-Resets the graph to its original state.

* public void UpdateChannelStatusBytes(byte dataDirectionByte, byte clockedOutputByte)

-Sets which channels are inputs and outputs based on the current data direction byte.

* public void AddDataBlock(byte[] dataPts, double elapsedTime)

-Adds a block of data points to the graph

* private void AddDataCapture(byte data, double time, bool forceDisplay)

-Adds a single data point to the graph. Uses an algorithm which only adds a point to the graph if the value is different than the previous value. This lowers the amount of data actually needed by the graph.

* public void SetupClockedOutputChannel(int channelNum, int frequency, double endTime)
* Tells the graph that a channel is a clocked output, and sets up the preset waveform on the display.
* public void SaveGraphToFile(string filename)

-Saves the graph as a .dsmu file to the desired filename.

* public void LoadGraphFromFile(string filename)

-Loads a previously saved graph from a .dsmu file at the desired path.

**Note**: Full code can be found in the EightChannelDisplay.cs code file

Once this user control was constructed it was added to the main user interface. When the start button is clicked, the user interface starts a background thread that performs a data collection loop. When a certain number of data points are collected the main thread of the interface is notified and the collected data block is given to the graph to be appended to the existing data.

***USB Interaction:***

The USB interaction for the DSMU GUI and device was accomplished using the supplied driver from Microchip (the manufacturer of the chip used in the device). These functionalities were then exported from the driver, which was not written in C# and therefore could not be used directly in the GUI code, using interop methods. Interop methods are written such that variable types from the driver’s native language such as pointers, unsigned integers, and DWords are converted into C# friendly data types such as byte arrays and booleans. A code-wrapper was constructed around the driver’s read, write, open, close, and device-count methods so that they could be easily called from the C# code behind the GUI. The functions imported were defined in the driver as described in the tables below:

HANDLE \*MPUSBOpen(instance, pVID\_PID,pEP,dwDir,dwReserved);   
Returns the handle to the endpoint pipe with matching VID & PID.

|  |  |  |
| --- | --- | --- |
| DWORD instance | Input | An instance number of the device to open, more detailed information in \_mpusbapi.cpp (see above) |
| PCHAR pVID\_PID | Input | A string containing the PID & VID value of the target device. The format is "vid\_xxxx&pid\_yyyy", where xxxx is the VID value in hex and yyyy is the PID value in hex. |
| PCHAR pEP | Input | A string of the endpoint number on the target endpoint to open. The format is "\\MCHP\_EPz", where z is the endpoint number in decimal (e.g. "\\MCHP\_EP1" |
| DWORD | dwDir | Specifies the direction of the endpoint,  use MP\_READ for MPUSBRead and MPUSBReadInt use MP\_WRITE for MPUSBWrite |
| DWORD | dwReserved |  |

DWORD \*MPUSBRead(handle,pData,dwLen,pLength,dwMilliseconds);

|  |  |  |
| --- | --- | --- |
| HANDLE handle | Input | Identifies the endpoint pipe to be read, the pipe handle must have been created with MP\_READ access attributes. |
| PVOID pData | Output | points to the buffer that receives the data read from the pipe |
| DWORD dwLen | Input | specifies the number of bytes to be read from the pipe |
| PDWORD pLength | Output | Points to the number of bytes read. MPUSBRead sets this value to zero before doing any work or error checking. |
| DWORD dwMilliseconds | Input | Specifies the time-out interval, in milliseconds. The function returns if the interval elapses, even if the operation is incomplete. If dwMilliseconds is zero, the function tests the data pipe and returns immediately. If dwMilliseconds is INFINITE, the function's time-out interval never elapses |

DWORD \*MPUSBWrite(handle,pData,dwLen,pLength,dwMilliseconds);

|  |  |  |
| --- | --- | --- |
| HANDLE handle | Input | Identifies the endpoint pipe to be written to. The pipe handle must have been created with MP\_WRITE access attribute. |
| PVOID pData | Input | points to the buffer containing the data to be written to the pipe |
| DWORD dwLen | Input | specifies the number of bytes to write to the pipe |
| PDWORD pLength | Output | Points to the number of bytes written by this function call. MPUSBWrite sets this value to zero before doing any work or error checking |
| DWORD dwMilliseconds | Input | Specifies the time-out interval, in milliseconds. The function returns if the interval elapses, even if the operation is incomplete. If dwMilliseconds is zero, the function tests the data pipe and returns immediately. If dwMilliseconds is INFINITE, the function's time-out interval never elapses. |

HANDLE \*MPUSBClose(instance);   
Returns the handle to the endpoint pipe with matching VID & PID.

|  |  |  |
| --- | --- | --- |
| DWORD instance | Input | An instance number of the device to close |

Note: These tables are from [**http://picmania.garcia-cuervo.com/USB\_MPUSBAPI\_DLL.php**](http://picmania.garcia-cuervo.com/USB_MPUSBAPI_DLL.php)

C# wrappers for all of the above functions were written, and then were used in the GUI to perform the USB interaction.

*DSMU Input / Output Channel Encapsulation:*

The easiest way to keep track of which of the eight channels on the DSMU device was set to be a input, clocked output, or complex output was to create a class to describe each of these types of channels, and then maintain a list that contained one of the three classes for each of the physical channels.

This list was implemented as an inherited instance of a standard C# ArrayList data type. By inheriting from the ArrayList type all the base functionalities required (such as adding, removing, and iterating) were already constructed. From that point it was only necessary to add methods to get certain information from the list pertaining more specifically to the DSMU device’s current state.

Some of the methods added to the inherited ArrayList are described below:

* public ArrayList GetComplexOutputChannels()
  + Returns a sub-list containing all the instances of ComplexOutputChannel classes.
* public ArrayList GetClockedOutputChannels()
  + Returns a sub-list containing all the instances of ClockedOutputChannel classes.
* public byte GetDataDirectionByte()
  + Returns a byte containing 1’s for channels that are inputs and 0’s for channels that are outputs.
* public byte GetComplexOutputByte()
  + Returns a byte containing 1’s for channels that are complex outputs and 0’s for all other channel types.
* public byte GetClockedOutputByte()
  + Returns a byte containing 1’s for channels that are clocked outputs and 0’s for all other channel types.
* public byte[] GetComplexOutputTable()
  + Returns a 64-byte table containing the bit stream information for all ComplexOutputChannel classes. For more information about this table, see the USB protocol section of this document.

**Note:** Full code can be found in the IoChannelList.cs code file

When the DSMU GUI software is loaded a new instance of the list is instantiated with the basic all-input channel configuration. Any time the user modifies a channel, though, the list is updated to hold instances of the required output channels with each of these instances holding the pertinent information (such as frequency for a clocked output, or bit stream for a complex output).

*USB Protocol and Packet Construction:*

*Clocked Output Configuration:*

*Overall Graphical User Interface (GUI) Operation:*

The basic flow of operation for the GUI is outlined in the flowchart below:



*Figure 4.1: Flowchart of overall GUI operation*

Upon loading the application the DSMU is searched for by the software. If the device is found to be connected to the computer the GUI starts up as normal. If the device is not found an error message is displayed and the application will not be able to perform any device interaction.

Once loaded successfully the user must configure the ports as desired. This involves specifying which ports are inputs, which are complex outputs, and which are clocked outputs. When these settings are selected by the user configuration packets are sent to the DSMU device by way of the USB connection.

When the channels are configured as desired by the user the program then waits for the start button to be pressed. Once pressed, a packet is sent over the USB line to tell the device to go into its Input / Output loop. During this loop the USB line is read continuously for new data, and any new data that is received is added to a list. When new data is added to the list the size of the list is compared to a set maximum number of new data points. If the number of new data points exceeds this limit, then the data is added to the graph and the list is cleared.

When the user is satisfied with the Input / Output collection run time they must hit the stop button on the GUI. When this button is pressed a packet is sent to the device over the USB line indicating to the device that a halt is desired. The DSMU device will then stop its Input / Output collection cycle. The GUI then collects any remaining data points from the USB buffer, and adds them to the graph. Once this process is complete the GUI is ready to either be shut down or to start another collection cycle.

**4.2) HARDWARE**

**4.2.1) Development Board**

The development board is the most important piece of hardware for this project. The development board has many key features that allow easy operation and test. The development board that is being used for this project is the MA180021. With this development board testing and implementation is done almost immediately due to all the features of this board.

Another feature of the development board is the 12 MHz miniature surface mounted crystal. The USB module for the microcontroller requires a fixed 6MHz or 48MHz clock to work in low speed or high speed mode, respectively. An internal 96MHz clock frequency is generated by the microcontroller. Since the microcontroller and the USB module do not have to run at the same frequency the surface mounted crystal can generate a different clock speed in integer multiples from the internal 96MHz signal.

The next feature of the development board is the jumpers. These jumpers can help with development and also to protect other devices. A self powered peripheral could source small amounts of power to the host device even when the host device is not powered. The small amount of power could damage the host device by not allowing it to become depowered. If the host device does not depower fully there could be problems at power up. The jumpers can prevent this by not allowing the USB peripheral device to power the 5V bus line when the host device has not powered the 5V bus line itself.

The development board also has a USB mini-B connector. The USB mini-B connector allows for the device to easily communicate with a computer for testing and also to graphically represent the data that the microcontroller has obtained. The mini-B connector allows for easy communication between the microcontroller and a computer, but the connector does not compromise the size of the development board.

The development board features an ICSP programming and debugging 6-pin header. Due to the size of this development board a 6-pin header was used instead of a full sized RJ11 jack which is normally used by Microchip on development boards. The 6-pin header allows for the microcontroller to be easily programmed, debugged, and reprogrammed in excess of 10,000 cycles.

The next features of the development board are not as important but still allow for easy testing and implementation of the device. The first is the 3.3V low IQ LDO linear regulator. The USB interface can provide 5V for USB peripheral devices to use. The 3.3V low drop out quiescent voltage linear regulator was used to ensure the development board can operate from only the power supplied by the USB interface. Next, are the two LEDs that are included on the development board. The LEDs are efficient with power consumption and can be used by the software for general test purposes. Next, are the two miniature pushbuttons. One pushbutton can be used as a reset for the microcontroller and the other can be used as a general purpose switch. Finally, the microcontroller has convenient test points which allow for easy testing of ground or power.

**4.2.2) Microcontroller**

One of the most important components of the development board is the microcontroller. The microcontroller that is included on this board is the PIC18F87J50. This microcontroller is desirable for this project because of some important features. The microcontroller is USB 2.0 with a low speed of 1.5Mbits/s and a full speed of up to 12Mbits/s and supports control, interrupt, and bulk transfers. The USB also has a 3.9Kbyte dual access RAM. The microcontroller has a flexible oscillator structure which allows for two external clock modes which work at up to 48MHz and an internal oscillator tunable from 31KHz to 8MHz. The microcontroller features an external memory bus. This external memory bus allows for access of up to 2Mbytes of RAM. The external ram can easily be accessed from 12-bit, 16-bit, and 20-bit addressing modes. The microcontroller also has some special features. The microcontroller can accept voltages from 0 to 3.3V and also 0 to 5V. The microcontroller also has a C compiler optimized architecture to allow for easy programming. The microcontroller also features high speed, low power CMOS flash technology.

**4.2.3) External Components**

External components for the finished project included low current red LEDs from Kingbright, and low current green LEDs from Osram Opto. The LEDs were low current due to the fact that USB can only supply 500mA maximum, and the onboard regulator can only supply 300mA. Also, 9 banana jacks were used for I/O ports 1-8 and ground. The case used was a generic project box from Radioshack. The jacks were washer mounted to the project box, and all other parts were secured into the box using hot glue.

**4.3) FIRMWARE**

**4.3.1) Introduction**

This document provides an overview of the entire firmware operations for the Digital Stimulus Monitor Unit (DSMU). Included are sections on input/output loop, timer and interrupts, USB configuration and communication protocol. Detailed code and relevant diagrams can be found at the end of this document.

The main task of the firmware is to coordinate data acquisition and transmission for the DSMU. The firmware is written in C programming language since the PIC18F85J50 microcontroller is supported by a C to Assembly compiler. The C programming language has been chosen over Assembly due to team members’ familiarity and its user friendly nature. The firmware code is compiled, debugged and tested using the MPLAB Integrated Development Environment (IDE) and the PIC In-Circuit Debugger (ICD).

**4.3.2) Firmware Structure**

4.3.2a- Overview:

The firmware is responsible for controlling the microcontroller as it collects and transmits data to and from the PC. All of the firmware code is written in the C programming language and compiled with the Microchip C18 compiler before being programmed to the chip on the DSMU. The overall structure of the firmware is displayed in Figure 4.2. As the microchip is powered on, it enters the USB initialization phase, where configurations are made to establish USB communications. After USB is initialized, the program goes into the Configuration loop. This is where data packets are received from the PC and all the input and output settings are configured. The program will remain in this configuration loop until all the configurations are set up and a start signal is received.

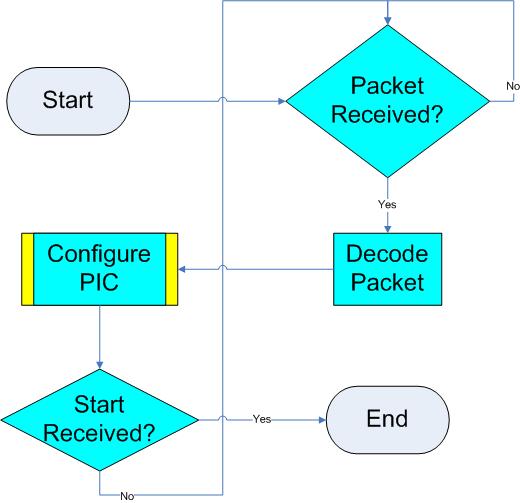


Figure 4.2 Overall Firmware Structure

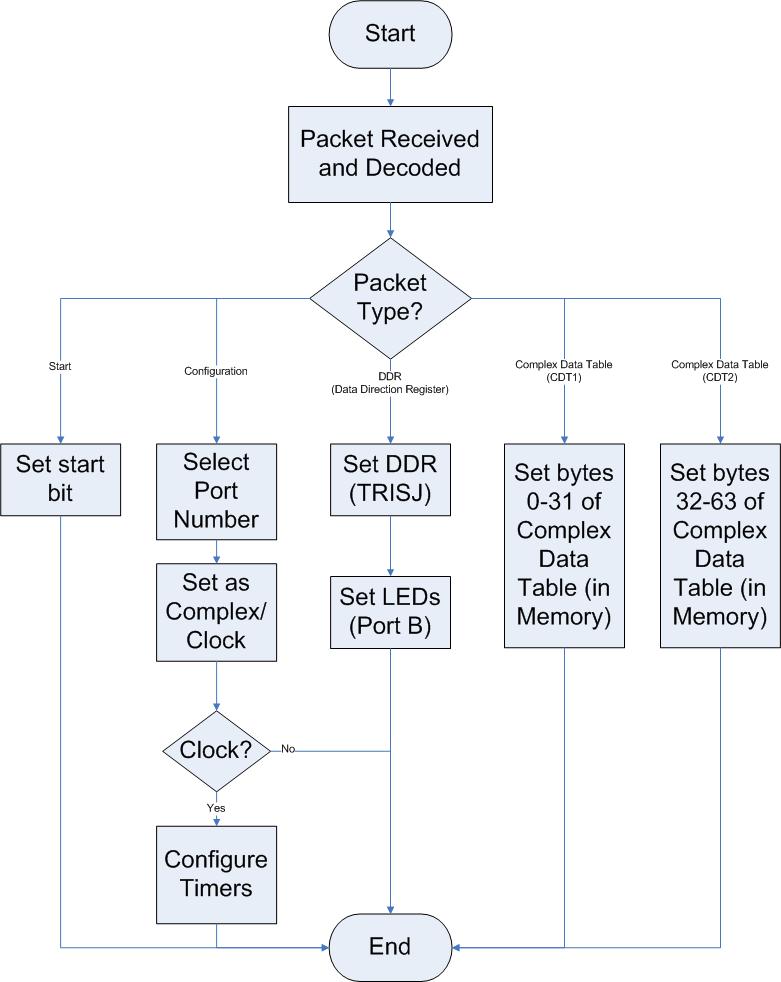
After the start signal is received, the program will jump out of the Configuration loop and enter the Input/ Output loop. In this phase, the chip collects digital signals from the input ports and transmits them back to the PC through the USB connection. It also transmits output waveforms, which have been configured by the user, to the ports designated as outputs. The device will continue to collect and transmit data until it receives a signal to stop from the GUI. At this point, the program will reenter the Configuration loop where it will await more data packets from the GUI. More complete detail of how the Configuration loop and the Input/ Output loops were implemented is described in the following sections.

4.3.2b- Configuration:

After the DSMU runs through its initialization stage, it immediately enters a configuration stage. This is the state where a majority of the communication between the PC and the DSMU takes place. During this time, all data is transferred to the DSMU. This includes the DDR, the complex data table, and any timer information required for generating a clock signal. The DSMU remains in the configuration state until the user pushes the start button. Furthermore, once the stop button is pressed, the DSMU will re-enter the configuration state. A flowchart of the configuration stage is provided in Figure 4.2(a).



*Figure 4.2(a): High level flowchart of the configuration state.*



*Figure 4.2(b): Detailed flowchart for configuration state showing how the PIC is configured*

During the configuration state, all data communication is taken place using USB packets. Furthermore, each of these packets contains different types of information. As shown in Figure 4.2(b), each of these packet types are decoded and interpreted differently. The protocol for these packet types was designed specifically for this project, and is explained in detail in the next section.

4.3.2c- Input/output:

The Input/ Output phase of the program is where all of the data is processed. The program reads data from the input ports and transmits it back to the GUI through the USB. It also writes the output signals sent to it from the GUI to the output ports. Figure 4.3 shows the breakdown of the I/O process.

Upon entering the I/O phase the program first starts the timers for the any clocked output wave signals that have been specified. These timers are set up so that every time they overflow, an interrupt is generated and the clock signal that it controls is toggled. This way, all the clock signals happen in the background and the I/O loop only needs to worry about the complex waveforms and inputs. Next the program checks to see if the device has been configured to generate any output at all. If there are no outputs, the above timers would not have been started and the program enters into input only mode. In this mode, the program simply reads from PORTJ, and sends the data back to the PC. It continues with this process until a stop signal is received, then which it reenters the configuration phase.

If the GUI has indicated to the firmware that there are output waveforms to be generated, the program enters the input and output mode. In this mode, the program takes bytes of data from the complex output waveform table and outputs it to PORTJ. The chip circulates through the 64 byte long table, outputting one at a time until it reaches the end. It then goes back to the beginning and continues this cycle. At the same time it is performing this task, it also reads the inputs from PORTJ and sends the data back to the PC. This process continues until a stop signal is received, then which it reenters the configuration phase. The complete code of the I/O mode can be viewed in the ProcessIO method in section A1 of the Appendix.



Figure 4.3 Input / Output Process

**4.3.3) USB Protocol**

In order for the PIC to communicate with the personal computer (PC) they must both follow an organized set of rules that governs how the data is sent. Thus, the team developed a protocol that stated how all data was to be transmitted. The details of this protocol went through many stages, though the components remained the same throughout the design process. Furthermore, only the final version of the protocol will be discussed in detail in this report.

It was decided to organize the data into smaller segments, known as packets. Each packet contains a different type of data. This is a much more modular design (and thus easier to debug) than sending all necessary data at once. Each packet is further divided into smaller segments (see Figure 4.4) Each of these smaller segments is dependent on the data the packet is carrying. They will be discussed in detail in the following sections.

Header

Data and Other Information\*

\*The other information is dependent on the packet type denoted in the header. See Table 1.

*Figure 4.4: Picture of packet divided into smaller segments*

4.3.3a- The Header:

Each data packet contains a one byte header. The value of this header is encoded by the sending device and decoded by the receiver. The header provides information about what the ata in the remainder of the packet is. The header byte is enumerated in Table 4.1.

|  |  |  |  |
| --- | --- | --- | --- |
| Title | Hexadecimal Value | Direction | Description |
| Start | 0x05 | PC -> DSMU | Alerts the DSMU to begin operation at users’ request. |
| Stop | 0x06 | PC -> DSMU | Alerts the DSMU to halt operation at users’ request. |
| Configuration | 0x07 | PC -> DSMU | Configures an individual port as generating clock or complex data. |
| DDR | 0x08 | PC -> DSMU | Configures DDR on the DSMU. This sets the direction of all the ports. |
| Complex Data Table (CDT) | 0x09 | PC -> DSMU | Sends the first 32 bytes of the complex data table. |
| CDT2 | 0x0A | PC -> DSMU | Sends the last 32 bytes of the complex data table. |
| Data | 0x0B | DSMU -> PC | Sends data acquired from input ports to the PC to be displayed on the graph. |
| DEBUG | 0x00  0x01  0x02  0x03 | PC -> DSMU or  DSMU -> PC | Used for debug purposes only. Not used in final implementation of the DSMU. |

*Table 4.1: Enumerated values of the header byte*

As previously stated, the header is simply one byte of data. However, not all possibilities for this byte are used. Therefore, all illegal header values are thrown out. This was considered acceptable because all errors that occur in the transmission of the data were automatically checked by the USB libraries being used. Thus, any illegal headers found must be due to erroneous code and would be caught during testing. This handling of illegal headers can be seen in the firmware code in section A1 of the Appendix section.

4.3.3b- Control Packets – Start and Stop:

Start and stop packets are the simplest types of packets used in this protocol. They are 1 byte long and contain only a header. No data is used in for any of them. When the user clicks the start button, a start packet is sent and the firmware exits the configuration stage and proceeds to the I/O stage. Furthermore, when a stop packet is received, the firmware exits the I/O stage and re-enters the configuration stage.

4.3.3c- Configuration Packets:

The configuration packet is a packet which configures one output port. It states whether the port is to generate a clock or complex signal, and also carries information for the timer registers (if applicable.) The general format is shown in Figure 4.5.

Header

Port Number

Clock/Complex

Timer Register Information\*

\*Only required if Clock signal specified

*Figure 4.5: Format of configuration packet*

The port number is one byte of information which corresponds to the port number that will be configured. This will be a value between 0x00 and 0x07, inclusive. The clock/complex segment is also one byte, but only the least significant bit is used. If the value is 0x00, a complex signal is to be generated. If the value is 0x01, a clock signal is to be generated. Since the clock signals will be generated using timers as interrupts, the timer control information will follow in the Timer Register Information segment. This timer information includes a starting value for the timer and a configuration register. These are both calculated and built by the GUI.

4.3.3d- DDR Packets:

The DDR packet will always be two bytes. The first byte is the header, as described earlier, and the second byte is the DDR contents. This controls the direction of the ports themselves, whether they are inputs or outputs. The value that is stored in the packet will simply be stored in the data direction register for the I/O port on the PIC (TRISJ.) Each bit has a corresponding port, and 0 corresponds to an output and 1 corresponds to an input. For example, if the DDR packet read 0x0F, ports 4-7 would be output ports and ports 0-3 would be input ports.

In addition, the DDR packet also controls the LEDs on the device. The LEDs are controlled by Port B on the PIC. Thus, in addition to setting TRISJ to the DDR, Port B also takes on the same values. This allows the proper LEDs to light up, indicating which ports are inputs and which are outputs.

4.3.3e- Complex Data Table Packets (CDT1 and CDT2):

The complex data table contains the waveforms of all complex output ports. These are custom waveforms as described by the user in a series of binary values (up to 64 bits.) The complex data table contains this information for all complex ports. Thus, the complex data table has a constant size of 64 bytes (8 ports each using 64 bits.) Since the header byte would make this over the maximum allowable size of a USB packet, it was decided to split this into two separate packets called CDT1 and CDT2. The packets are illustrated in Figure 4.6.

Header Byte = CDT1

.

.

.

Header Byte = CDT2

.

.

.

Port

Port

7 6 5 4 3 2 1 0

7 6 5 4 3 2 1 0

Table Row

0

1

2

3

31

32

33

34

35

63

*Figure 4.6: An illustration of how the complex data table packets are constructed*

As shown in Figure 4.6, the table has a constant size of 64 bytes regardless of how many ports are selected as complex output ports. This is because the firmware uses the DDR Packet to determine if the port is an input or an output (see Section 4.3.3d), and Configuration Packets to determine if the port is to output a clock signal or a complex signal (see Section 4.3.3c.) Thus, the firmware automatically selects the columns in the table to output depending on the DDR and Configuration packets sent.

**4.3.4) USB Code**

4.3.4a- USB Specification and the Microchip Firmware:

In the development of the 18F87J50 demo board, a base firmware had to be established that would interact with the onboard USB hardware to communicate with a target PC. The extensive requirements of the official USB specification led to the decision that existing firmware should be used rather than attempting to code from scratch. Microchip, the manufacturer of the 18F87J60 demo board, had previously developed several examples of USB usage that were contained within its “MCHFSUSB” package. These examples were available for free on their website.

Each of the Microchip examples contained different classes of USB firmware. In the official USB specification, several device classes are defined; audio, communications, HID (Human Interface Device), mass storage, generic, printer, and others. The Microchip examples contained code for HID, communication, and generic devices. The communication device class was eliminated as an option because of the complexity of the example, and because the intended application of the device was not in a communications role. The benefit of using the HID device class was that Microsoft Windows provides pre-installed drivers and no installation would be required in order to communicate with the GUI. Due to difficulty in modifying the example code to work with both transmission as well as reception, the generic class was eventually chosen. This choice also meant that a suitable driver had to be located and modified to suit the device. The driver that was utilized in communication was also taken directly from Microchip and interfaced with the C# GUI.

4.3.4b- Structure of the Generic Class Firmware Example:

The generic firmware example was structured in a way such that all of the main background USB functions were separated from the user’s code. From a very high level, the example is structured as shown in figure 4.7.

main.c

usbdrv.c

USB Tasks

Initialize USB

Check Bus Status

Service USB

user.c

User code

USB Hardware Interaction

Figure 4.7: High Level Layout of Example Firmware

As shown in this diagram, the main loop of the program involves initializing the USB hardware, then continuously servicing the USB hardware and executing the user’s code. In this case, servicing the USB hardware means that the program checks on the device’s status, determines whether the state has changed (attached, detached, suspended), and handles any hardware interrupts that may have been generated. In this way, most of the hardware-level requirements of the USB protocol are hidden from the code that is developed by users that wish to utilize the framework.

However, the user’s code still must utilize the transmission and reception capabilities of the USB to have any useful effect. The main transmission and reception methods are shown in figure 4.8.

usbgen.c

user.c

Data Packet

size

USBGenWrite

mUSBGenTxIsBusy()

USB Busy?

No

Figure 4.8: USB Transmission

usbgen.c

user.c

Empty Data Packet

size

USBGenRead

Figure 4.9: USB Reception

As shown in figure 4.9, transmission and reception of data happens using functions contained in the usbgen.c file. In transmission, the user first creates a “data packet” that contains the information they wish to send. This data packet is actually a simple byte array that has been named using structures, and is defined in user.h. Once the data has been formed, the user checks to see whether the bus is busy, and then passes a pointer to the data packet and its size through to the “USBGenWrite” function.

In reception of data, the “USBGenRead” function is given the pointer to an empty data packet, and the maximum size of this structure. If data is waiting on the bus, then the function will fill each byte of the data packet up to the specified size. If data is not waiting, then the function will not block, but will return a 0 to signify that nothing was received. This data can then be accessed either by the name given in the structure or by simply using the “datapacket.\_byte[n]” interface.

4.3.4 -cModifications to the Example Firmware:

In order to suit our device, many modifications to this existing firmware were made. First, the demo was originally developed for the pic18F4550, and many of the definitions were changed from that device to those that match the 18F87J50. This change was made inside of io\_cfg.h. A different device was selected through the MPLAB configuration, and a new linker script was installed in the project.

Secondly, all of the specialized demo code was removed. All of the functions inside of user.c were emptied, and temperature.c was removed entirely. The structure definition of a data packet inside of user.h was heavily modified in order to properly match the USB protocol that was developed for the DSMU.

Finally, the USB descriptors that identify a device were modified from their originals to match the DSMU. Inside of usbdsc.c, the description strings were changed to suit our product so that the correct device name would be displayed when it is first plugged into the PC. Additionally, the Vendor ID and Product ID had to be changed from the default Microchip setting. These two IDs are critical in defining what device is connected to the PC. Under professional circumstances, these numbers would need to be bought and registered from an official USB authority. In the case of the DSMU, a Vendor ID of 0x04DB and Product ID of 0x000E were arbitrarily chosen.

As the Vendor ID and Product ID were chosen differently from their defaults, the Microchip device driver for the PC had to be modified to match their values. The Vendor ID and Product ID are the defining mechanism for which driver is used when a device is plugged in to the USB ports, so by changing the driver correctly, Windows will accept it as a choice during installation.

4.3.4d- Testing Process and Driver Usage:

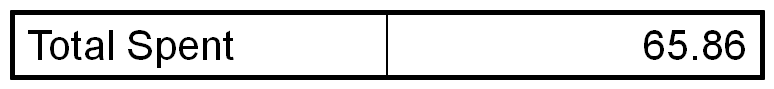
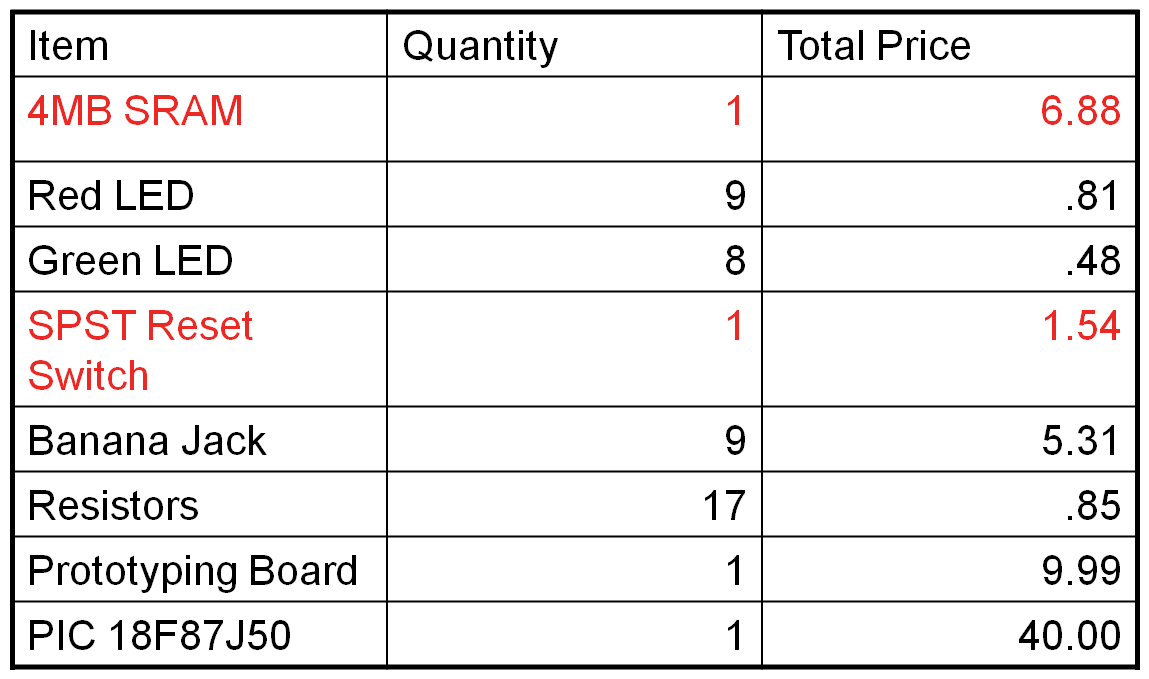
In order to test that the base USB firmware was operating properly, a C# application was developed that would execute three different test cases. This application communicated with the Microchip USB driver by explicitly registering functions with a DllImport of “mpusbapi.dll”. Through these functions, pipes can be opened that send and receives byte arrays of data.

The first test performed was to verify that data was being sent properly from the PC to the device. Once received by the device, the onboard LED would be lit. After this test was verified, a second test involved both sending and receiving data. An integer was sent from the PC to the device, where the device doubled the number and sent it back for display. After this test was run, a third test sent a byte array of sequential numbers to the device. The device then lit when certain values were matched to expected parts of the data packet. The purpose of this test was to ensure that the packet was being filled properly and that no overlap was occurring.

**4.3.5) Timers & Interrupts**

Timers 0, 1 and 3 will be used to implement the clocked output for the DSMU project. The over-flow flag of each timer will trigger an event that toggles the output clock signal. The user can specify any of ports one, two and three on the digital stimulus monitoring unit to be clocked outputs. Each port maps to the three timers respectively. Timer 0 has the lowest output frequency, while timers 1 and 3 have a slightly higher output frequency. This restriction is due to the fact that timers 1 and 3 only contain prescale values from 1 to 8 and timer 0 can obtain values in the range of 1 to 256. Therefore, timer 0s operating frequency is from 10 MHz to .60 Hz compared to timers 1 and 3 which operate from 10 MHz to 16 Hz. All timers contain a 16-bit counter which is preloaded with values sent to the PIC from the PC. The PC contains an algorithm which determines what the value of the counter and prescale values need to be to obtain the specified frequency. The algorithm determines these values by dividing the requested frequency into the operating frequency of the PIC and loading this value into the counter. If the value does not fit within the counter bounds then the prescale is incremented, the value is divided by the prescale value and again checked to see if it fits in the timer. This operation is done until the final counter value is valid for a 16-bit counter. The value of the counter and a byte containing the timer configuration register information, including prescale, is sent to the PIC.

**4.4) PROJECT COSTS**



Items listed in red were initially included in the design and were removed during construction. By removing those items, $8.42 is saved in the development costs. Furthermore, when looking at the mass production costs, this reduces the cost per 1000 units approximately $4,600.00. This makes the cost per unit only $17.88 a unit.

**[5] RESULTS**

**5.1) Specification Achieved**

1. The DSMU has 8 configurable I/O ports.
   * The specification required the DSMU to have 8 ports. However, that specification was exceeded by having all 8 ports individually configured as either input or output. In addition, these ports are marked as input or output by lighting LEDs to indicate to the user that the port was properly configured.
2. Output signals at 3.3 V.
   * The DSMU is able to output its signals with a VOL of 0V and a VOH of 3.3V. These values are acceptable by today’s standards of digital circuits.
3. The DSMU will output a complex data table.
   * The DSMU will output a user-specified sequence up to 64 bits long. If the user configures this bit sequence to be a clock signal, it can achieve a maximum frequency of 12.25 kHz.
4. The DSMU can read custom signals on its input ports.
   * The DSMU can read any signal on its input ports, as long as they are configured to accept an input signal. Additionally, the maximum frequency the DSMU can read is 12.25 kHz.
5. USB Communication
   * USB Communication was properly established between the PC and the DSMU. The PC will properly take user specified characteristics and configure the DSMU accordingly. Furthermore, the DSMU is able to send data packets back to the PC for display.
6. A Graphical User Interface is properly written.
   * The GUI is complete and will properly communicate and configure the device. Furthermore, it will graph all received data onto its display. However, due to limitations in the USB driver being used, data is often lost. This causes skewed graphs to be displayed.
7. The Graphical User Interface is able to save data for future reference.
   * The GUI is able to save a copy of the gathered data as a JPEG picture. This can then be used by the user in the future for a reference.
8. The project was completed under budget.

**5.2) Issues Encountered**

1. Issues related to lab equipments

* Did not have necessary administrative privileges to install software necessary for the project developments.
* Did not have working devices such as the circuit programmer until the 8th week.
* Lacked necessary equipments such as signal generator, working multimeter.

1. Getting host PC to recognize the device

* Initially it was thought that the device was programmed as a Human Interface Device (HID).
* The first version of the USB communication code on the host PC was written to connect to such a device, but it was later found that the device had been programmed to use a proprietary driver.
* USB devices use a unique identifier known as a vendor and product ID (VID and PID) to distinguish one another. This number must be hard-coded into the host PC software, and if it is not exactly correct no connection will be established. This caused a number of problems as the current VID and PID was changed numerous times, but not communicated clearly between the development groups.

**5.3) Possible Future Improvements**

1. Use two PIC microcontrollers and shared RAM

* Since the current DSMU design’s speed was limited by USB transmission, the best idea would be to isolate USB tasks from the input/output tasks. This can be done by using one PIC for USB and another PIC for input/output. Any data that needs to be transferred between the two PICs will be stored in a shared, dual-access RAM. This should improve the overall speed and decrease the error rate of the DSMU.

1. Configure the device to be a human interface device (HID)

* The microcontroller chosen has the capability of being programmed to operate as a Human Interface Device. Devices that meet these specifications are able to operate using drivers built into all installations of Windows. This would eliminate the need to create, debug, and install an extra driver along with the device. HID devices also work using a very widely documented protocol, and are guaranteed to operate at 64 kilobytes per second at all times. This would ensure capability of up to 32 kilohertz operation by the DSMU.

1. Mount PIC microcontroller to a circuit board

* Actually mount the PIC demo board on a circuit board so that the PIC microcontroller pins can be easily accessed during construction and testing. The current hardware configuration involved more hot glue than solder, which is unstable and unprofessional.

1. Determine an accurate method to find input/output frequency

* The current frequency was derived through testing and was then hardcoded into the GUI in order to output data capture times onto the graph, but this method is not completely accurate. A new method needs to be derived so that the frequency can be determined dynamically to ensure greater accuracy.

**APPENDIX**

**A1- Software Source Code**

*A1.a) Application Form Code*

*A1.b) Data Display Code*

*A1.c) USB Interaction Protocol Code*

*A1.d) Device Encapsulation Classes*

**B2- Firmware Source Code**

*B1.a) Main.c*

*B2.a) User.c*

*C3.a) User.h*