# logo_bw.eps *Worcester Polytechnic Institute*

# *Robotics Engineering Program*

# Introduction to AVR

# Submitted By Joel Sutherland Adam Panzica

## **Date Submitted : 1/28/2010**

## **Date Completed : 1/20/2010**

## **Course Instructor : Profs. Fischer & Ciaraldi**

## **Lab Section : C02**

# Abstract

We developed a demonstration of techniques for configuring the STK500 for Analog-to-Digital conversion, real-time square-wave generation, serial communication, and waveform analysis.

# Introduction

In this lab, we demonstrated several capabilities of the STK500 development system with the Atmel AVR microcontroller:

1. We wrote functions to configure the STK500 for communication with a Hyperterminal and to send and receive information.
2. We configured the Analog-to-Digital Converter (ADC) and developed software to read, interpret, and log potentiometer readings.
3. We used timers and interrupts to develop a square-wave generator which takes input from digital switches and and the ADC-interpreted potentiometer to control frequency and duty cycle, respectively.
4. We used the AVR to take samples of input waveforms and log them.

By completing this lab, we became familiar with configuring timers, interrupt registers, and analog/digital input/output. This will serve as a foundation for further development on this platform.

# Methodology

### Configuring Serial Communication

We first set up basic functions to establish a serial connection from the STK500 to a PC serial port, to be read from the PuTTY hyperterminal. We wrote a parameterized initialization function, Initialize\_Serial\_P(), that allows the serial port to be configured to any of the modes supported by the AVR’s USART, giving it great flexibility for future use without having to alter code. All valid values that can be passed to the function are defined in symbolic constants, easing the demands on the programmer and making the code easier to read and understand. We also developed and custom print functions, Serial\_Print\_String() and Serial\_Print\_Int(). These functions send strings of arbitrary length as well as 16bit integers, which can be represented as any base between 2 and 32, respectively. These functions provide a powerful tool for debugging, and also provide a means of datalogging any inputs to the AVR. The code for the serial library can be seen in the Simple\_Serial.h/.c files in appendix A.

### Configuring the ADC

For our ADC configuration, we defined a “sensor” struct, ADC\_Sensor, used by the rest of our functions to define a sensor by name, channel, and minimum and maximum values in engineering units. This provides a convenient interface for the programmer as it is often desirable to output data in engineering units rather than internal units when datalogging, and the struct, when combines with the functions which utilize it, automates this process, as well as keeping the name and channel of the sensor in one conveniently referenceable variable. The ADC library also contains an initialization function, ADC\_Initialize\_P(), to enable the ADC for use, and a parameterized ADC value retrieval function, ADC\_Get\_Value(), which can take a single sample from a specified channel. This library of functions provides a solid framework for utilizing the ADC for both this lab and whenever it might be needed in the future. The code for the ADC library can be seen in ADC.h/c.

### Square-Wave Generator and Timer functions

To generate a square wave, we needed a system which could both count to a set value and reset, and at the same time trigger a response when the counter reaches an intermediate value. These two points provide a frequency (the reset point) and a duty cycle (the intermediate value). All of the hardware timers on the AVR provide just such facilities, but in order to reach the low frequencies required in this lab, the 16bit timer must be used, as the divider from system clock is limited to 1024. Utilizing the equation, the 16bit timer can count to 3.64 seconds without rolling over, giving it a minimum frequency of less than 1/3Hz. The timer functions utilize the same hardware timer, but omit the intermediate trigger.

Our square-wave generator consists of several elements:

1. A timer which triggers interrupts
2. Interrupt Service Routines to toggle the output value
3. Switches which change the frequency of the signal
4. A potentiometer which sends a signal to the ADC to change the duty cycle

We set timer to FastPWM mode, and chose to toggle the output value in the ISRs instead of configuring the timer to do so automatically, as we had difficulty getting the pins which are supposed to toggle automatically in FastPWM mode to do so, and it proved reliable. The square wave generator initialization function can be seen in Sqaure\_Wave16\_Initialize(), which provides a parameter for setting the initial frequency. There are also functions to adjust the frequency and duty cycle, in Square\_Wave16\_Set\_Frequency() and Square\_Wave16\_Set\_Duty(). The ISR’s which handle the triggers includes a switch statement which looks at a global flag to see if they should handle events for square wave generation or for timer events. Full code can be seen in Timer16\_Functions.h/.c.

For the square-wave generator and the external waveform logging, we configured PuTTY to automatically log the data from the STK500 in Comma-Separated Values (CSV) format. We then imported this data into Matlab and generated the graphs there.

# Results

### Square-Wave Generator

After the square-wave generator was complete, we ran a demonstration of its capabilities. We stepped through 1Hz, 50Hz, 75Hz, and 100Hz frequencies and altered the duty cycle. The results were consistent with our expectations.

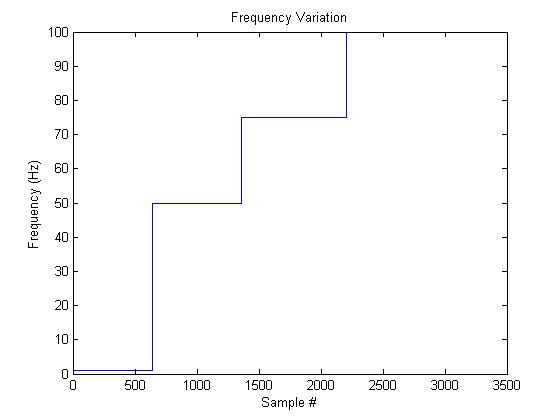


Figure 1: Frequency Variation

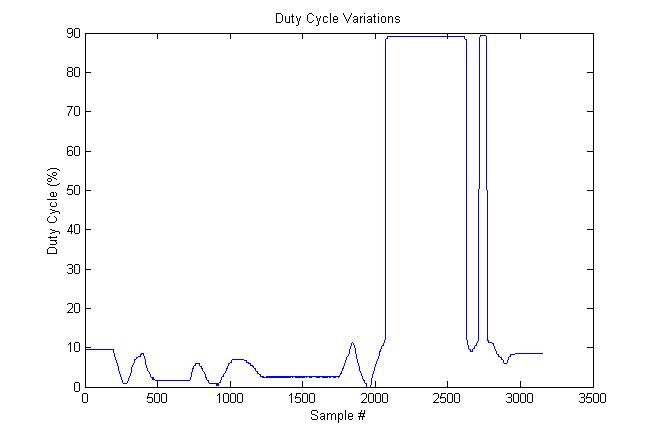


Figure 2: Duty Cycle Variations

### Waveform Logging

For the final section of the lab, we had to set up the AVR’s ADC to take samples at exactly 200Hz using the timer. We then sampled three different wave forms generated by the lab bench functions generators: Square, Saw, and Sine. Each waveform was sampled at 5 different frequencies: 25Hz, 50Hz, 75Hz, 100Hz and 125Hz. The data was logged using the serial output to puTTY, and then imported into MATLab and parsed into graph form. The graphs of the various waveforms can be seen in figures 3-18 below.

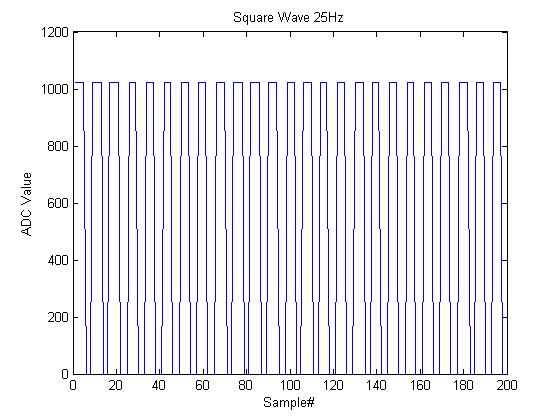


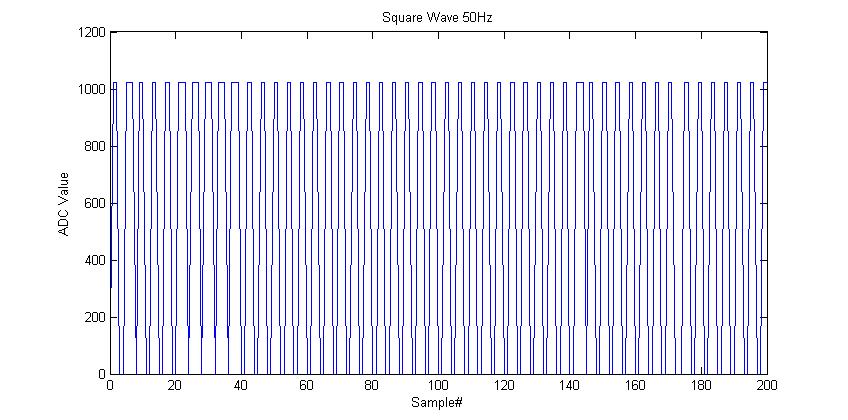
Figure 3: Square Wave, 25Hz

Figure 4: Square Wave, 50Hz

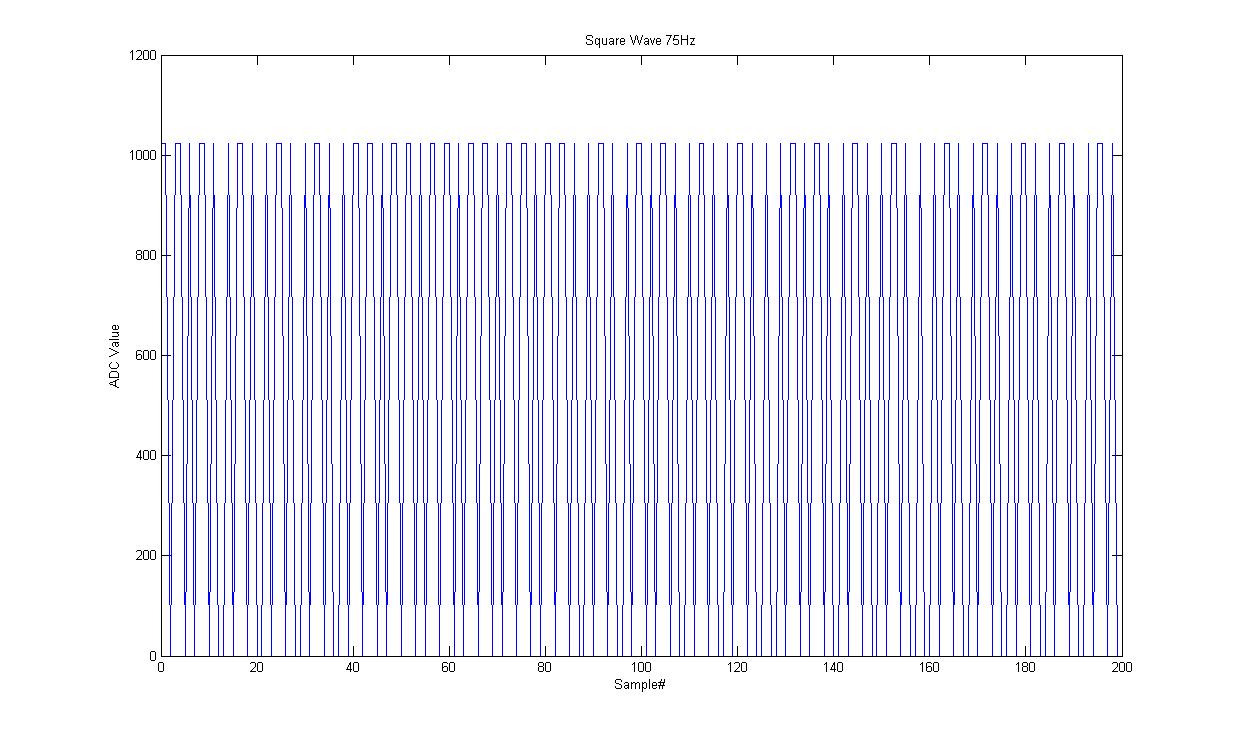


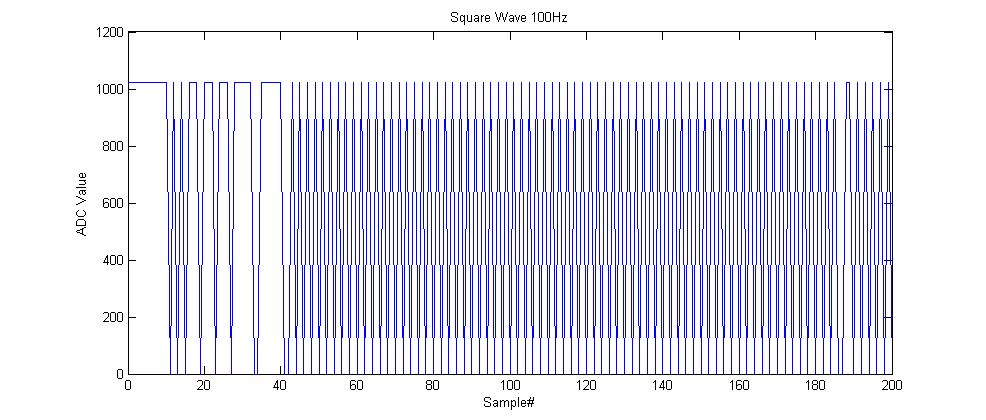
Figure 5: Square Wave, 75Hz

Figure 6: Square Wave, 100Hz

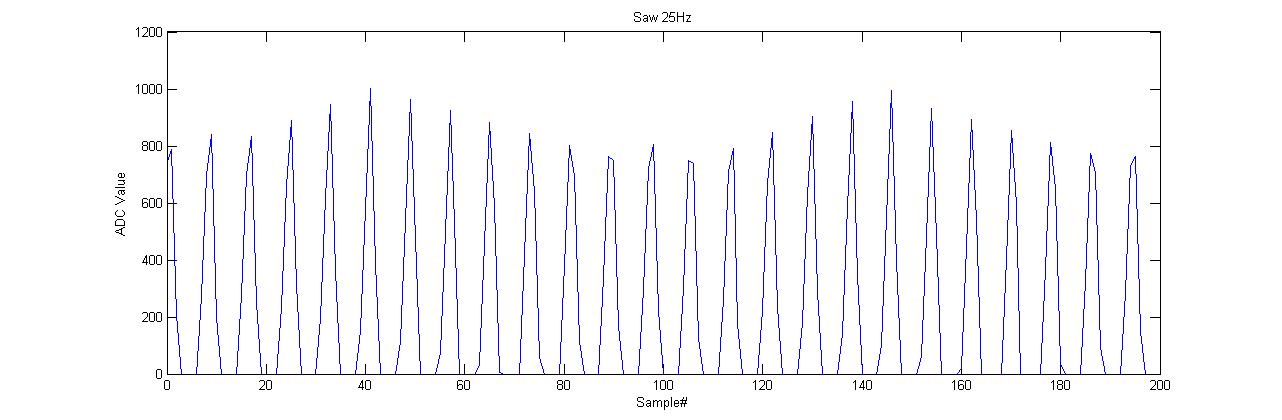


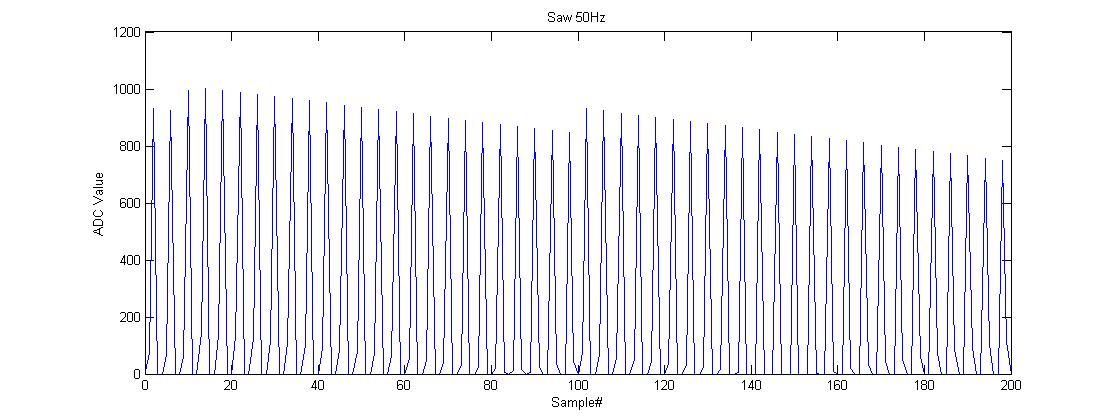
Figure 7: Saw Wave, 25Hz 

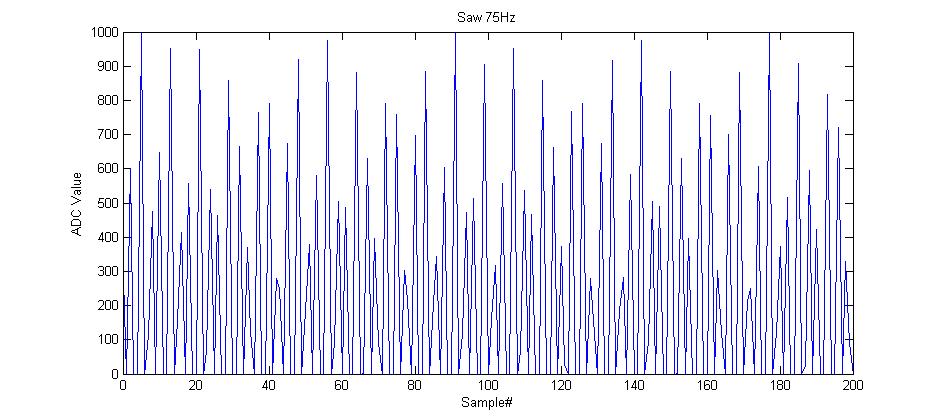
Figure 8: Saw Wave, 50Hz

Figure 9: Saw Wave, 75Hz

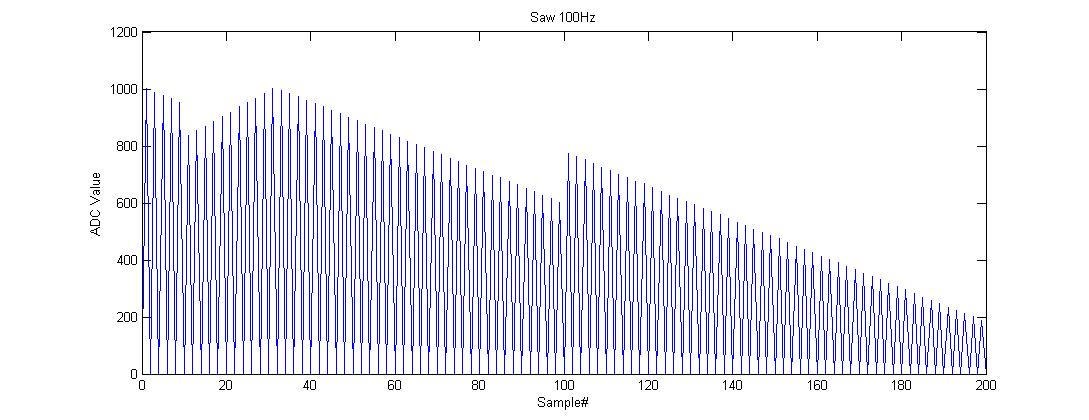


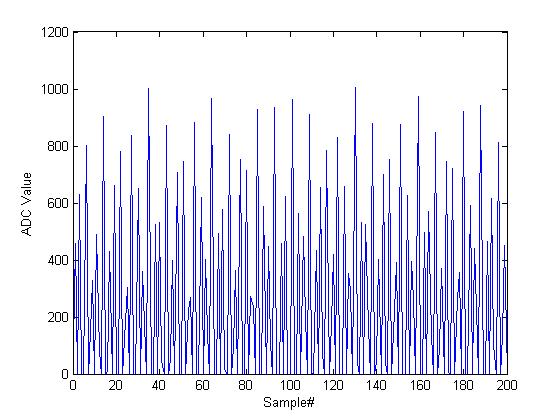
Figure 10: Saw Wave, 100Hz

Figure 11: Saw Wave, 125Hz

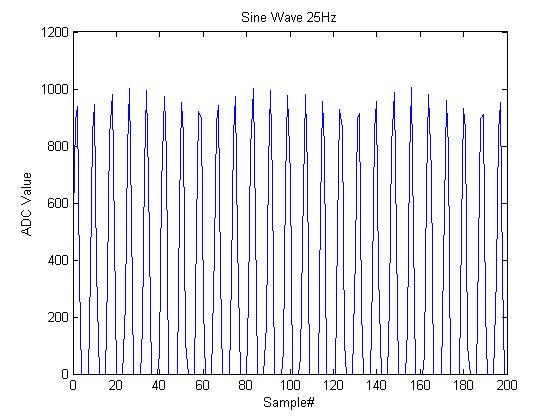


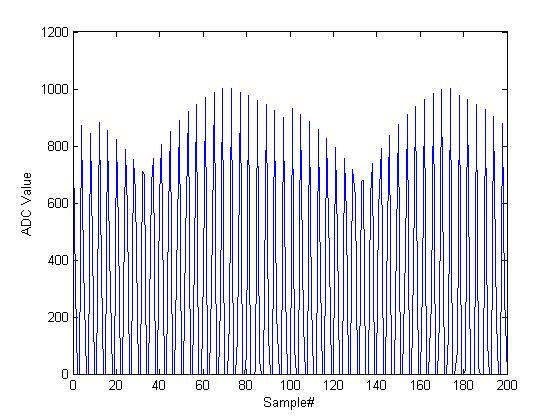
Figure 12: Sine Wave, 25Hz

Figure 13: Sine Wave, 50Hz

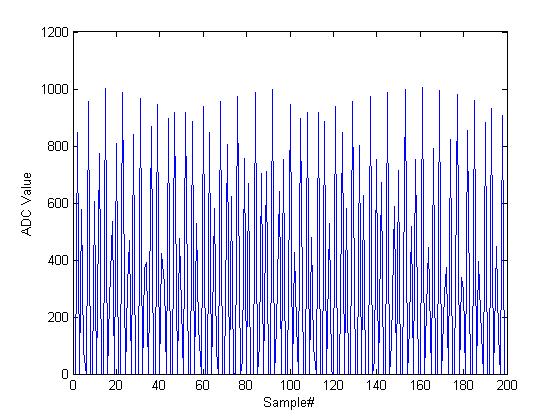


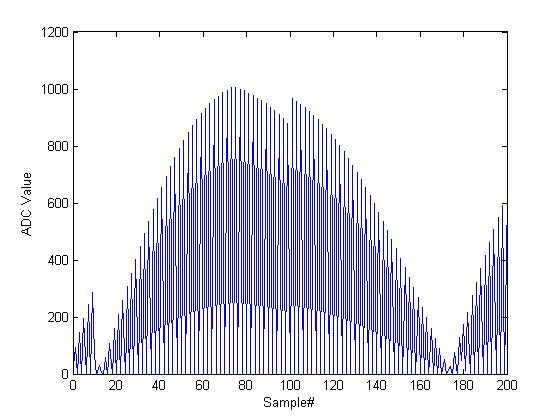
Figure 14: Sine Wave, 75Hz

Figure 15: Sine Wave, 100Hz

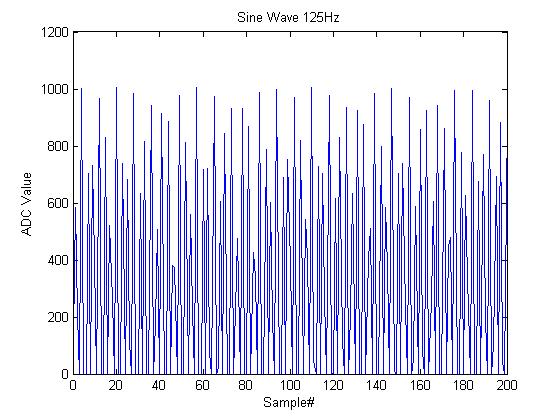


Figure 16 : Sine Wave, 125Hz

Due to the relatively low sample rate compared to wave frequency, the actual wave shape is very difficult to discern past 25Hz. However, the general wave form can be seen up to 100Hz by examining the whole graph, where the pattern becomes immediately evident. However, at 125Hz, and interesting phenomenon occurs: the graph of the waveform seems to repeat that of 75Hz. This is a prime example of frequency ‘folding’ around the Nyquest point. The Nyquest frequency is defined as being ½ that of the sample frequency, and is the maximum frequency of a waveform that can be measured. Past the Nyquest frequency, the waveform will fold around the point, following the equation. With our 200Hz sample rate, the Nyquest frequency is 100Hz, so, for example, 125Hz folds back on itself to become 75Hz. When the wave frequency folds back past 0Hz (, It then begins to fold back up towards the Nyquest point, so again using our 200Hz sample rate, a 225Hz waveform would appear to be a 25Hz wave form. While the Nyquest frequency is the theoretical maximum frequency of a waveform that can be measured, in practice the sample rate should be at least 6-10 times the maximum frequency expected to be measured. At the Nyquest point, there are only two sample per period, which makes it highly unlikely that an accurate waveform analysis can be performed. It takes at least 3-5 points (or preferably more) in order to get some idea of the shape and properties of the waveform being studied.

# Discussion

Through this laboratory experience, we established many of the capabilities and limitations of the STK500. We also gained valuable experience in the reading and interpretation of technical documents for microprocessors and peripheral devices useful in Robotics development.

### Serial Configuration

In the development of our serial configuration code, we strove to write reusable code and practiced creating modular, parameterized functions. This proved quite useful as we were able to modify the baudrate and channel settings quickly to adapt to different sampling requirements and computer setups. We also renewed our appreciation for the serial port as a debugging tool, as it helped us track down errors in the code on many occasions.

### Configuring the ADC

In the ADC configuration, we further built a library for convenient future use. We defined a sensor struct to represent different definitions of the digital signals being processed so we could quickly modify the code to read from a source other than the potentiometer we used for this lab.

### Square-Wave Generator and Waveform Logging

Our square-wave generator utilized the 16-bit timer, the ADC, Serial Communication, and basic I/O form the STK500. Each element we had previously implemented in a generalized form, so our focus could be on the task of accurately interpreting the desired waveforms.

In the waveforms we studied, we observed the folding which was expected when analyzing a signal with a frequency close to the sampling frequency. We also observed imperfections in uniformity of the folding as a result of the small amount of time lost in printing and calculation.

# Appendix A: Serial Configuration

/\*\*

\* @file Simple\_Serial.h

\*

\*@date Jan 26, 2010

\*@author Adam Panzica

\*/

**#ifndef** SIMPLE\_SERIAL\_H\_

**#define** ***SIMPLE\_SERIAL\_H\_***

**char** ***Init\_Serial\_P***( **unsigned** **int** **baud**, **unsigned** **char** **frameSize**, **unsigned** **char** **stopBits**, **unsigned** **char** **parity** )**;**

**char** ***Init\_Serial\_I***( **unsigned** **int** **baud**, **unsigned** **char** **frameSize**, **unsigned** **char** **stopBits**, **unsigned** **char** **parity** )**;**

**char** ***Serial\_Print\_Char***( **unsigned** **char** **data**)**;**

**char** ***Serial\_Read\_Char***(**unsigned** **char\*** **data**)**;**

**char** ***Serial\_Print\_String***(**char\*** **string**)**;**

**char** ***Serial\_Print\_Int***( **int** **data**, **int** **base**)**;**

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* SYMBOLIC CONSTANTS \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

/\*\*@def BAUD24

\* @brief Represents a baudrate of 2400bps\*/

**#define** ***BAUD24*** 2400

/\*\*@def BAUD48

\* @brief Represents a baudrate of 4800bps\*/

**#define** ***BAUD48*** 4800

/\*\*@def BAUD96

\* @brief Represents a baudrate of 9600bps\*/

**#define** ***BAUD96*** 9600

/\*\*@def BAUD144

\* @brief Represents a baudrate of 14.4kbps\*/

**#define** ***BAUD144*** 14400

/\*\*@def BAUD192

\* @brief Represents a baudrate of 19.2kbps\*/

**#define** ***BAUD192*** 19200

/\*\*@def BAUD288

\* @brief Represents a baudrate of 28.8kbps\*/

**#define** ***BAUD288*** 28800

/\*\*@def BAUD384

\* @brief Represents a baudrate of 38.4kbps\*/

**#define** ***BAUD384*** 38400

/\*\*@def BAUD576

\* @brief Represents a baudrate of 57.6kbps\*/

**#define** ***BAUD576*** 57600

/\*\*@def BAUD768

\* @brief Represents a baudrate of 76.8kbps\*/

**#define** ***BAUD768*** 76800

/\*\*@def BAUD1152

\* @brief Represents a baudrate of 115.2kbps\*/

**#define** ***BAUD1152*** 115200

/\*\*@def BAUD2304

\* @brief Represents a baudrate of 230.4kbps\*/

**#define** ***BAUD2304*** 230400

/\*\*@def BAUD2500

\* @brief Represents a baudrate of 250.0kbps\*/

**#define** ***BAUD2500*** 250000

/\*\*@def FRM9

\* @brief Represents a frame size of 9 bits\*/

**#define** ***FRM9*** 9

/\*\*@def FRM8

\* @brief Represents a frame size of 8 bits\*/

**#define** ***FRM8*** 8

/\*\*@def FRM7

\* @brief Represents a frame size of 7 bits\*/

**#define** ***FRM7*** 7

/\*\*@def FRM6

\* @brief Represents a frame size of 6 bits\*/

**#define** ***FRM6*** 6

/\*\*@def FRM5

\* @brief Represents a frame size of 5 bits\*/

**#define** ***FRM5*** 5

/\*\*@def STOP1

\* @brief Represents 1 stop bit\*/

**#define** ***STOP1*** 1

/\*\*@def STOP2

\* @brief Represents 2 stop bits\*/

**#define** ***STOP2*** 2

/\*\*@def NOPAR

\* @brief Represents no parity\*/

**#define** ***NOPAR*** 0

/\*\*@def ODDPAR

\* @brief Represents odd parity\*/

**#define** ***ODDPAR*** 2

/\*\*@def EVPAR

\* @brief Represents even parity\*/

**#define** ***EVPAR*** 1

**#endif** /\* SIMPLE\_SERIAL\_H\_ \*/

/\*\* @brief Provide a simple serial protocol utilizing USART0

\*

\* @file Simple\_Serial.c

\*

\* This source file contains a library of functions to provide simple

\* serial communications (Char/String out, Char/String in), utilizing

\* either polling or interrupt based transmission handling. It uses

\* a single USART for communications, USART0

\*

\* @author Adam Panzica

\* @date 19-Jan-2009

\* @version 1.0 Initial version with fixed parameters for initialization functions, Print\_Char(), Print\_String(), Read\_Char()

\* @version 1.1 Modified initialization functions to allow for adjustable setup parameters.

\* @version 1.2 Added Print\_Int()

\*/

**#include** <avr/io.h>

**#include** "Simple\_Serial.h"

**#include** <avr/delay.h>

**#include** <stdlib.h>

/\*\* clock frequency of the AVR in MHz/100\*/

**#define** ***FCLK*** 18432

/\*\*@fn char Init\_Serial\_P( unsigned int baud, unsigned char frameSize, unsigned char stopBits, unsigned char parity )

\*@brief Initialize USART0 for asynchronous serial communications, polling

\*@param [in] baud Baudrate for serial communications

\*@param [in] frameSize number of data bits (5-9), defaults to 8 if parameter is out of bounds

\*@param [in] stopBits number of stop bits (1-2), defaults to 1 if parameter is out of bounds

\*@param [in] parity 1 if even parity is desired, 2 if odd parity is desired, 0 for no parity, defaults to no parity if parameter is out of bounds

\*@return 1 if success, else 0 if there was still data in RXX/TX buffers, preventing a change to USART parameters

\*

\* This function initializes USART0 for asynchronous serial communications by setting the appropriate enable and control registers

\* In this mode, the RX and TX buffers do not generate interrupts, and thus must be polled to receive information

\*/

**char** ***Init\_Serial\_P***( **unsigned** **int** **baud**, **unsigned** **char** **frameSize**, **unsigned** **char** **stopBits**, **unsigned** **char** **parity** )

{

**unsigned** **char** ***tempUCSRB*** **=** 0b0**;**

**unsigned** **char** ***tempUCSRC*** **=** 0b0**;**

**int** ***tempBaud*;**

//Check to ensure that there is no data in the RX/TX buffers before changing frame size

**if**((UCSR0A **&** (1**<<**RXC0)) **|** (UCSR0A **&** (1**<<**TXC0))) **return** 0**;**

//Set Parity bits in UCSRC

**switch** (**parity**)

{

**case** 0**:**

**tempUCSRC|=** (0**<<**UPM01) **|** (0**<<**UPM00)**;**

**break;**

**case** 1**:**

**tempUCSRC|=** (1**<<**UPM01) **|** (0**<<**UPM00)**;**

**break;**

**case** 2**:**

**tempUCSRC|=** (1**<<**UPM01) **|** (1**<<**UPM00)**;**

**break;**

**default:**

**tempUCSRC|=** (0**<<**UPM01) **|** (0**<<**UPM00)**;**

**break;**

}

//Set stop bits in UCSRC

**switch**(**stopBits**)

{

**case** 1**:**

**tempUCSRC** **|=** 0**<<**USBS0**;**

**break;**

**case** 2**:**

**tempUCSRC** **|=** 1**<<**USBS0**;**

**break;**

**default:**

**tempUCSRC** **|=** 0**<<**USBS0**;**

**break;**

}

//Set frame size in UCSR and UCSRB

**switch**(**frameSize**)

{

**case** 9**:**

**tempUCSRC|=** (1**<<**UCSZ00) **|** (1**<<**UCSZ01)**;**

**tempUCSRB|=** 1**<<**UCSZ02**;**

**break;**

**case** 8**:**

**tempUCSRC|=** (1**<<**UCSZ00) **|** (1**<<**UCSZ01)**;**

**tempUCSRB|=** 0**<<**UCSZ02**;**

**break;**

**case** 7**:**

**tempUCSRC|=** (0**<<**UCSZ00) **|** (1**<<**UCSZ01)**;**

**tempUCSRB|=** 0**<<**UCSZ02**;**

**break;**

**case** 6**:**

**tempUCSRC|=** (1**<<**UCSZ00) **|** (0**<<**UCSZ01)**;**

**tempUCSRB|=** 0**<<**UCSZ02**;**

**break;**

**case** 5**:**

**tempUCSRC|=** (0**<<**UCSZ00) **|** (0**<<**UCSZ01)**;**

**tempUCSRB|=** 0**<<**UCSZ02**;**

**break;**

**default:**

**tempUCSRC|=** (1**<<**UCSZ00) **|** (1**<<**UCSZ01)**;**

**tempUCSRB|=** 0**<<**UCSZ02**;**

**break;**

}

//Set enable flags

**tempUCSRB|=** (1**<<**RXEN0) **|** (1**<<**TXEN0)**;**

//Set baud rate

**tempBaud** **=** **FCLK/**((**baud/**100)**\***16)**\***10-1**;**

UBRR0H **=** (**unsigned** **char**)(**tempBaud>>**8)**;**

UBRR0L **=** (**unsigned** **char**)**tempBaud;**

//Set control registers

UCSR0B **=** **tempUCSRB;**

UCSR0C **=** **tempUCSRC;**

**return** 1**;**

}

/\*\*@fn char Init\_Serial\_I( unsigned int baud, unsigned char frameSize, unsigned char stopBits, unsigned char parity )

\*@brief Initialize USART0 for asynchronous serial communications, interrupts

\*@param [in] baud Baudrate for serial communications

\*@param [in] frameSize number of data bits (5-9), defaults to 8 if parameter is out of bounds

\*@param [in] stopBits number of stop bits (1-2), defaults to 1 if parameter is out of bounds

\*@param [in] parity 1 if even parity is desired, 2 if odd parity is desired, 0 for no parity, defaults to no parity if parameter is out of bounds

\*@return 1 if success, else 0 if there was still data in RXX/TX buffers, preventing a change to USART parameters

\*

\* This function initializes USART0 for asynchronous serial communications by setting the appropriate enable and control registers

\* In this mode, the RX and TX buffers generate interrupts

\*/

**char** ***Init\_Serial\_I***( **unsigned** **int** **baud**, **unsigned** **char** **frameSize**, **unsigned** **char** **stopBits**, **unsigned** **char** **parity** )

{

**unsigned** **char** ***tempUCSRB*** **=** 0b0**;**

**unsigned** **char** ***tempUCSRC*** **=** 0b0**;**

**int** ***tempBaud*;**

//Check to ensure that there is no data in the RX/TX buffers before changing frame size

**if**((UCSR0A **&** (1**<<**RXC0)) **|** (UCSR0A **&** (1**<<**TXC0))) **return** 0**;**

//enable RX/TX interrupt flags

**tempUCSRB** **|=** (1**<<**RXCIE0) **|** (1**<<**TXCIE0) **|** (1**<<**UDRIE0)**;**

//Set Parity bits in UCSRC

**switch** (**parity**)

{

**case** 0**:**

**tempUCSRC|=** (0**<<**UPM01) **|** (0**<<**UPM00)**;**

**break;**

**case** 1**:**

**tempUCSRC|=** (1**<<**UPM01) **|** (0**<<**UPM00)**;**

**break;**

**case** 2**:**

**tempUCSRC|=** (1**<<**UPM01) **|** (1**<<**UPM00)**;**

**break;**

**default:**

**tempUCSRC|=** (0**<<**UPM01) **|** (0**<<**UPM00)**;**

**break;**

}

//Set stop bits in UCSRC

**switch**(**stopBits**)

{

**case** 1**:**

**tempUCSRC** **|=** 0**<<**USBS0**;**

**break;**

**case** 2**:**

**tempUCSRC** **|=** 1**<<**USBS0**;**

**break;**

**default:**

**tempUCSRC** **|=** 0**<<**USBS0**;**

**break;**

}

//Set frame size in UCSR and UCSRB

**switch**(**frameSize**)

{

**case** 9**:**

**tempUCSRC|=** (1**<<**UCSZ00) **|** (1**<<**UCSZ01)**;**

**tempUCSRB|=** 1**<<**UCSZ02**;**

**break;**

**case** 8**:**

**tempUCSRC|=** (1**<<**UCSZ00) **|** (1**<<**UCSZ01)**;**

**tempUCSRB|=** 0**<<**UCSZ02**;**

**break;**

**case** 7**:**

**tempUCSRC|=** (0**<<**UCSZ00) **|** (1**<<**UCSZ01)**;**

**tempUCSRB|=** 0**<<**UCSZ02**;**

**break;**

**case** 6**:**

**tempUCSRC|=** (1**<<**UCSZ00) **|** (0**<<**UCSZ01)**;**

**tempUCSRB|=** 0**<<**UCSZ02**;**

**break;**

**case** 5**:**

**tempUCSRC|=** (0**<<**UCSZ00) **|** (0**<<**UCSZ01)**;**

**tempUCSRB|=** 0**<<**UCSZ02**;**

**break;**

**default:**

**tempUCSRC|=** (1**<<**UCSZ00) **|** (1**<<**UCSZ01)**;**

**tempUCSRB|=** 0**<<**UCSZ02**;**

**break;**

}

//Set enable flags

**tempUCSRB|=** (1**<<**RXEN0) **|** (1**<<**TXEN0)**;**

//Set baud rate

**tempBaud** **=** **FCLK/**((**baud/**100)**\***16)**\***10-1**;**

UBRR0H **=** (**unsigned** **char**)(**tempBaud>>**8)**;**

UBRR0L **=** (**unsigned** **char**)**tempBaud;**

//Set control registers

UCSR0B **=** **tempUCSRB;**

UCSR0C **=** **tempUCSRC;**

**return** 1**;**

}

/\*\*@fn char Serial\_Print\_Char( unsigned char data)

\* @brief Print a single character to the serial port

\* @param [in] data char to be sent to the serial port

\* @return 1 if success, else 0

\*

\* This function prints a single character to

\*the serial port utilizing USART0

\*/

**char** ***Serial\_Print\_Char***( **unsigned** **char** **data**)

{

// check to make sure serial port is enabled

**if**(**!**(UCSR0B **&** (1**<<**TXEN0))) **return** 0**;**

// Wait for empty transmit buffer

**while** ( **!**( UCSR0A **&** (1**<<**UDRE0)) )**;**

// Put data into buffer, sends the data

UDR0 **=** **data;**

**return** 1**;**

}

/\*\*@fn char Serial\_Read\_Char( unsigned char\* data)

\* @brief Read a single character from the serial port

\* @param [in] data\* pointer to the location to score the red byte from the serial port

\* @return 1 if success, else 0

\*

\* This function reads a single character from

\*the serial port utilizing USART0

\*/

**char** ***Serial\_Read\_Char***(**unsigned** **char\*** **data**)

{

// check to make sure serial port is enabled

**if**(**!**(UCSR0B **&** (1**<<**RXEN0))) **return** 0**;**

// Wait for data to be received

**while** ( **!**(UCSR0A **&** (1**<<**RXC0)) )**;**

//Get and return received data from buffer

**\*data** **=** UDR0**;**

**return** 1**;**

}

/\*\*@fn char Serial\_Print\_String( char\* string)

\* @brief Print a string of chars to the serial port

\* @param [in] string\* pointer to the location of a null terminated string

\* @return 1 if success, else 0

\*

\* This function reads a single character from

\*the serial port utilizing USART0

\*/

**char** ***Serial\_Print\_String***(**char\*** **string**)

{

**int** ***i*=**0**;**

// check to make sure serial port is enabled

**if**(**!**(UCSR0B **&** (1**<<**TXEN0))) **return** 0**;**

//print the string to the serial port

**while**(**string**[**i**] **!=** '\0')

{

**Serial\_Print\_Char**(**string**[**i**])**;**

**i++;**

}

**return** 1**;**

}

/\*\*@fn char Serial\_Print\_Int( int data, int base)

\* @brief Print an int to the serial port

\* @param [in] data integer to be printed to the serial port

\* @param [in] base base to use for int to string conversion (2,10,16, etc)

\* @return 1 if success, else 0

\*

\* This function prints a single int to

\*the serial port utilizing USART0

\*/

**char** ***Serial\_Print\_Int***( **int** **data**, **int** **base**)

{

**char** ***buffer***[17]**;**

//check to make sure the serial port is enabled

**if**(**!**(UCSR0B **&** (1**<<**TXEN0))) **return** 0**;**

//perform data conversion from int to string

**itoa**(**data**, **buffer**, **base**)**;**

//print the resultant string to the serial port

**Serial\_Print\_String**(**buffer**)**;**

**return** 1**;**

}

# Appendix B: ADC Configuration

/\*\*

\* @file ADC.h

\*

\*@date Jan 22, 2010

\*@author Adam Panzica

\*/

**#ifndef** ADC\_H\_

**#define** ***ADC\_H\_***

/\*\*

\*@struct ADC\_Sensor ADC.h "ADC.h"

\*@brief this strcut defines an ADC Sensor object

\*/

**struct** ADC\_Sensor

{

**char** name[15]**;** /\*\*< name array of chars for storing the sensor name\*/

**unsigned** **char** channel**;** /\*\*< channel number the ADC sensor is on\*/

**int** valueAtMax**;** /\*\*< valueAtMax value of the ADC sensor in engineering units at Vin=VRef\*/

**int** valueAtMin**;** /\*\*< valueAtMin value of the ADC sensor in engineering units at Vin=Ground\*/

**int** scaleFactor**;** /\*\*< scaleFactor scale factor for converting ADC units to engineering units for the sensor\*/

}**;**

**void** ***ADC\_Init\_P***()**;**

**int** ***ADC\_Get\_Value***(**unsigned** **char** **channel**)**;**

**int** ***ADC\_Calc\_Volts***(**int** **ADCValue**, **int** **VRef**)**;**

**struct** ADC\_Sensor ***ADC\_Sensor\_Construct*** ( **char** **\*name**, **unsigned** **char** **channel**, **int** **valueAtMax**, **int** **valueAtMin**)**;**

**int** ***ADC\_Sensor\_Calc\_Value*** (**struct** ADC\_Sensor **\*sensor**, **int** **ADCValue**)**;**

**#endif** /\* ADC\_H\_ \*/

/\*\* @brief Provides functions for utilizing the ADC converter

\*

\* @file ADC.c

\*

\* This source file contains a library of functions to utilize the ADC in both polling

\* and interrupt mode

\*

\* @author Adam Panzica

\* @author Joel Sotherland

\* @date 22-Jan-2009

\*@version 1.0 Initial version with ADC\_Init\_P() and ADC\_Get\_Value()

\*@version 1.1 Updated ADC code, fixed prescaler issue, added ADC\_Sensor struct and its constructor ADC\_Sensor\_Construct() for conversions.

\*@version 1.2 Added ADC\_Sensor\_Calc\_Value() and ADC\_Calc\_Volts()

\*

\*/

**#include** <avr/io.h>

**#include** "ADC.h"

/\*\*@fn void ADC\_Init\_P()

\*@brief Initialize ADC for polling use

\*@return none

\*

\* This function initializes the ADC to be read using a polling system

\*/

**void** ***ADC\_Init\_P***()

{

ADCSRA **=** 1**<<**ADEN **|** 1**<<**ADPS2 **|** 1**<<**ADPS1 **|** 1**<<**ADPS0**;**

}

/\*\*@fn int ADC\_Get\_Value(unsigned char channel)

\*@brief Take a single sample from the ADC

\*@param [in] channel channel number (0-7) to read the ADC value from. Defaults to 0 if out of bounds

\*@return Value on the specified ADC channel

\*

\* This function takes a single sample from a specified ADC channel using a polling mechanism to determine when the sample is ready

\*/

**int** ***ADC\_Get\_Value***(**unsigned** **char** **channel**)

{

**int** ***temp*;**

**char** ***low*;**

//check channel bounds

**if**((**channel** **<=** 7) **&&** (**channel** **>=** 0)) ADMUX **=** **channel;**

**else** ADMUX **=** 0**;**

//start a sample

ADCSRA **|=** 1**<<**ADSC**;**

//wait till sample is complete

**while**(ADCSRA**&**(1**<<**ADSC))**;**

//read sample value

**low** **=** ADCL**;**

**temp** **=** ADCH**;**

**temp** **=** **temp<<**8**;**

**temp** **|=** **low;**

**return** **temp;**

}

/\*\*@fn struct ADC\_Sensor ADC\_Sensor\_Construct ( char \*name, unsigned char channel, int valueAtMax, int valueAtMin)

\* @brief this function constructs an ADC\_Sensor struct

\* @param [in] \*name pointer to an array of chars containing the name of the sensor. Max length is 15 including null terminator

\* @param [in] channel channel number that the sensor is on (0-7)

\* @param [in] valueAtMax the value (in engineering units, E.G. PSI) of the sensor at Vin=Vref

\* @param [in] valueAtMin the value (in engineering units, E.G. PSI) of the sensor at Vin=Ground

\*

\* This function constructs an ADC\_Sensor using the passed values. It returns an ADC\_Sensor struct which is used in other ADC

\* functions to calculate engineering values from raw ADC units

\*/

**struct** ADC\_Sensor ***ADC\_Sensor\_Construct*** ( **char** **\*name**, **unsigned** **char** **channel**, **int** **valueAtMax**, **int** **valueAtMin**)

{

**int** ***i*;**

**struct** ADC\_Sensor ***temp*;**

**temp.**scaleFactor **=** ((**valueAtMax-valueAtMin**)**\***100)**/**1024**;**

**temp.**channel **=** **channel;**

**temp.**valueAtMax **=** **valueAtMax;**

**temp.**valueAtMin **=** **valueAtMin;**

**for**(**i=**0**;** **i<**15**;** **i++**)

{

**temp.**name[**i**]**=name**[**i**]**;**

}

**return** **temp;**

}

/\*\*@fn int ADC\_Sensor\_Calc\_Value (struct ADC\_Sensor \*sensor, int ADCValue)

\* @brief this function calculates the engineering unit value of a sensor reading in ADC values

\* @param [in] \*sensor pointer to an ADC\_Sensor

\* @param [in] ADCValue returned by the ADC (0-1024)

\* @returns the value of the sensor in engineering units (multiplied by a scale factor of 100 to allow for decimal values)

\*

\* This function constructs an ADC\_Sensor using the passed values. It returns an ADC\_Sensor struct which is used in other ADC

\* functions to calculate engineering values from raw ADC units

\*/

**int** ***ADC\_Sensor\_Calc\_Value*** (**struct** ADC\_Sensor **\*sensor**, **int** **ADCValue**)

{

**return** (**sensor->**scaleFactor)**\*ADCValue+**(**sensor->**valueAtMin)**\***100**;**

}

/\*\*@fn int ADC\_Calc\_Volts(int ADCValue, int VRef)

\* @brief this function converts raw ADC values into millivolts for a given Vref.

\* @param [in] ADCValue the 10-bit value returned by the ADC

\* @param [in] VRef the reference voltage (in mV) used by the ADC

\* @returns the input voltage in millivolts

\*

\* This function calculates the input voltage based on ADC value and Vref.

\*/

**int** ***ADC\_Calc\_Volts***(**int** **ADCValue**, **int** **VRef**)

{

**long** ***temp1*;**

**long** ***temp2*;**

**long** ***temp3*;**

**temp1** **=** (**long**)**ADCValue;**

**temp2** **=** (**long**)**VRef;**

**temp3** **=** **temp1\*temp2/**1024**;**

**return** (**int**)**temp3;**

}

# Appendix C: Wave Generator and Logging

/\*\*

\* @file Timer16\_Functions.h

\*

\*@date Jan 26, 2010

\*@author Adam Panzica

\*@author Joel Sutherland

\*/

**#ifndef** SQUARE\_WAVE\_H\_

**#define** ***SQUARE\_WAVE\_H\_***

**void** ***Square\_Wave16\_Initialize***(**unsigned** **int** **frequency**, **unsigned** **int** **fclk**)**;**

**void** ***Square\_Wave16\_Set\_Duty\_Cycle***(**char** **dutyCycle**)**;**

**void** ***Square\_Wave16\_Set\_Freq***(**unsigned** **int** **frequency**, **unsigned** **int** **fclk**)**;**

**void** ***Timer16\_Initialize***(**unsigned** **int** **frequency**, **unsigned** **int** **fclk**)**;**

**void** ***Timer16\_Set\_Freq***(**unsigned** **int** **frequency**, **unsigned** **int** **fclk**)**;**

**unsigned** **char** ***ADC\_To\_Duty***(**int** **ADCVal**)**;**

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* SYMBOLIC CONSTANTS \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

/\*\*@def FPWM

\* @brief selection for fast PWM mode

\*/

**#define** ***FPWM*** 0

/\*\*@def CTC

\* @brief selection for CTC mode

\*/

**#define** ***CTC*** 0

**#endif** /\* SQUARE\_WAVE\_H\_ \*/

/\*\*

\* @file Timer16\_Functions.c

\* @brief Functions for generating square waves

\*

\*This source file contains functions for generating square waves using the 16bit timer on the AVR and interrupts

\*

\*@date Jan 25, 2010

\*@author Adam Panzica

\*@author Joel Sutherland

\*@version 1.0 Initial version with Timer\_Initilize() and Timer\_Set\_Duty\_Cycle(), and ISR(TIMER1\_COMPA) and ISR(TIMER1\_CAPT)

\*@version 1.1 Reworked Timer\_Initilize() and Timer\_Set\_Duty\_Cycle() to Square\_Wave16\_Initialize() and Square\_Wave16\_Set\_Duty\_Cycle(). Added Square\_Wave16\_Set\_Duty\_Cycle() and Square\_Wave16\_Set\_Freq()

\*@version 2.0 Changed file name to Timer16\_Functions, added Timer16\_Initialize() and Timer16\_Set\_Freq(), and adjusted ISR's to allow for timer specific code as well as PWM code

\*/

/\*\*@def PRESCALE

\* @brief prescale value for timer calculations\*/

**#define** **PRESCALE** 1024

**#include** <avr/io.h>

**#include** <avr/interrupt.h>

**#include** "Simple\_Serial.h"

**#include** "Timer16\_Functions.h"

**#include** "ADC.h"

/\*\*@var mFlag

\* @brief flag for ISR's to select between Timer and PWM operation

\*/

**static** **char** ***mFlag*** **=** 0**;**

//lab1 specific counting flag

**static** **int** ***timestamp*** **=** 0**;**

/\*\*@fn ISR(TIMER1\_COMPA\_vect)

\* @brief ISR handler for Timer1 compare A match

\*

\* This ISR is run when the value in OCR1A matches TCNT1. It blinks an LED on PORTB and toggles a pin on PORTD

\*/

**ISR**(***TIMER1\_COMPA\_vect***){

**switch**(***mFlag***)

{

**case** 0**:**

//Insert Timer mode handling code here

**if**(***timestamp*** **<=** 200)

{

**Serial\_Print\_Int**(***timestamp***, 10)**;**

**Serial\_Print\_String**(", ")**;**

**Serial\_Print\_Int**(**ADC\_Get\_Value**(1), 10)**;**

**Serial\_Print\_String**("\r\n")**;**

***timestamp*++;**

}

**break;**

**case** 1**:**

//Insert PWM mode handling code here

PORTB **=** PORTB **&**(**~**(1**<<**PB3))**;**

PORTD **=** PORTD **&** **~**(1**<<**PD6)**;**

**break;**

**default:**

**break;**

}

}

/\*\*@fn ISR(TIMER1\_OVF\_vect)

\* @brief ISR handler for Timer1 overflow

\*

\* This ISR is run when the value TCNT1 matches ICR1. It blinks an LED on PORTB and toggles a pin on PORTD

\*/

**ISR**(***TIMER1\_OVF\_vect***){

PORTB **=** PORTB**|**(1**<<**PB3)**;**

PORTD **=** PORTD **|** (1**<<**PD6)**;**

}

/\*\*@fn void Square\_Wave16\_Initialize(unsigned int frequency, unsigned int fclk)

\*@brief Initialize the square wave generator with a frequency

\*@param [in] frequency the desired frequency in Hz

\*@param [in] fclk the base clock of the processor in MHz/100

\*

\*This function initializes a square wave generator using the AVR's 16bit timer. Default duty cycle is 0%

\*/

**void** ***Square\_Wave16\_Initialize***(**unsigned** **int** **frequency**, **unsigned** **int** **fclk**)

{

//Set the mFlag to PWM mode

***mFlag*** **=** 1**;**

//Enable Interrupts, set interrupt flags in TIMSK1

**sei**()**;**

TIMSK1 **=** (1**<<**OCIE1A)**|**(1**<<**TOIE1)**;**

//calculate value of TOP from desired frequency and base clock

**Square\_Wave16\_Set\_Freq**(**frequency**, **fclk**)**;**

OCR1A **=** 0**;**

//Set up TCCR1A/B/C for FastPWM operation with ICRn as the TOP value

TCCR1A**|=** (1**<<**WGM11)**|**(0**<<**WGM10)**;**

TCCR1B **=** (1**<<**WGM13)**|**(1**<<**WGM12)**;**

TCCR1C **=** 0**;**

//Set output pin states

TCCR1A**|=** (0**<<**COM1A1)**|**(1**<<**COM1A0) **|** (0**<<**COM1B1) **|** (1**<<**COM1B0)**;**

//Set Prescaler

TCCR1B**|=** (1**<<**CS12)**|**(0**<<**CS11)**|**(1**<<**CS10)**;**

}

/\*\*@fn void Timer16\_Initialize(unsigned int frequency, unsigned int fclk)

\*@brief Initialize a 16bit timer

\*@param [in] frequency the desired frequency in Hz

\*@param [in] fclk the base clock of the processor in MHz/100

\*

\*This function initializes a 16bit timer utilizing OCR1A for the TOP value

\*/

**void** ***Timer16\_Initialize***(**unsigned** **int** **frequency**, **unsigned** **int** **fclk**)

{

//Set the mFlag to Timer mode

***mFlag*** **=** 0**;**

//Enable Interrupts, set interrupt flags in TIMSK1

**sei**()**;**

TIMSK1 **=** (1**<<**OCIE1A)**|**(1**<<**TOIE1)**;**

//calculate value of TOP from desired frequency and base clock

**Square\_Wave16\_Set\_Freq**(**frequency**, **fclk**)**;**

OCR1A **=** 0**;**

//Set up TCCR1A/B/C for CTC operation depending on mode flag with ICRn as the TOP value

TCCR1A**|=** (0**<<**WGM11)**|**(0**<<**WGM10)**;**

TCCR1B **=** (1**<<**WGM13)**|**(1**<<**WGM12)**;**

TCCR1C **=** 0**;**

//Set output pin states

TCCR1A**|=** (0**<<**COM1A1)**|**(1**<<**COM1A0) **|** (0**<<**COM1B1) **|** (1**<<**COM1B0)**;**

//Set Prescaler

TCCR1B**|=** (1**<<**CS12)**|**(0**<<**CS11)**|**(1**<<**CS10)**;**

}

/\*\*@fn void Square\_Wave16\_Set\_Duty\_Cycle(char dutyCycle)

\*@brief Set the duty cycle of the square wave

\*@param [in] dutyCycle the desired duty cycle to be set in %, 0-100

\*

\*This function sets the duty cycle for the square wave being generated by the 16bit timer.

\*/

**void** ***Square\_Wave16\_Set\_Duty\_Cycle***(**char** **dutyCycle**)

{

**unsigned** **long** ***temp*;**

**temp** **=** ((**long**)(ICR1)**\***(**long**)**dutyCycle**)**/**100**;**

OCR1A **=** (**int**)**temp;**

}

/\*\*@fn void Square\_Wave16\_Set\_Freq(unsigned int frequency, unsigned int fclk)

\*@brief Set the frequency of the square wave

\*@param [in] frequency Desired frequency to be set

\*@param [in] fclk Base clock rate

\*

\*This function sets the frequency for the square wave being generated by the 16bit timer.

\*/

**void** ***Square\_Wave16\_Set\_Freq***(**unsigned** **int** **frequency**, **unsigned** **int** **fclk**)

{

**long** ***temp*;**

**int** ***top*;**

**temp** **=** ((**long**)**fclk/**((**long**)**frequency**))**\***1000**/PRESCALE**-1**;**

**top** **=** (**int**)**temp;**

ICR1 **=** **top;**

//reset the timer to ensure the timer doesn't miss TOP and wrap around

TCNT1 **=** 0x00**;**

}

/\*\*@fn void Timer16\_Set\_Freq(unsigned int frequency, unsigned int fclk)

\*@brief Set the frequency of the timer

\*@param [in] frequency Desired frequency to be set

\*@param [in] fclk Base clock rate

\*

\*This function sets the frequency for the timer generated by the 16bit timer.

\*/

**void** ***Timer16\_Set\_Freq***(**unsigned** **int** **frequency**, **unsigned** **int** **fclk**)

{

**long** ***temp*;**

**int** ***top*;**

**temp** **=** ((**long**)**fclk/**((**long**)**frequency**))**\***1000**/PRESCALE**-1**;**

**top** **=** (**int**)**temp;**

OCR1A **=** **top;**

//reset the timer to ensure the timer doesn't miss TOP and wrap around

TCNT1 **=** 0x00**;**

}

/\*\*@fn unsigned char ADC\_To\_Duty(int ADCVal)

\*@brief Converts an ADC value to a duty cycle

\*@param [in] ADCVal 10bit ADC value

\*@return Duty cycle (0-100%)

\*

\* This function converts a 10bit ADC value to a duty cycle from

\* 0 to 100%

\*/

**unsigned** **char** ***ADC\_To\_Duty***(**int** **ADCVal**){

**long** ***temp*;**

**temp** **=** ((**long**) (**ADCVal**)**\***100)**/**1024**;**

**return** (**unsigned** **char**) **temp;**

}

# Appendix D: Lab1\_Demos.c

/\*\* @brief Lab 1 main source file

\*

\* @file Lab1\_Demos.c

\*

\* This source file contains the main code loops for Lab 1.

\* Additional functionality is provided by various header files

\*

\* @author Adam Panzica

\* @author Joel Sotherland

\* @date 26-Jan-2009

\*/

**#include** "Simple\_Serial.h"

**#include** "portIO.h"

**#include** "ADC.h"

**#include** "Timer16\_Functions.h"

**#include** <avr\io.h>

**#include** <avr\delay.h>

**#include** <avr\interrupt.h>

**void** ***Step\_2***()**;**

**void** ***Step\_3***()**;**

**void** ***Step\_6***()**;**

**unsigned** **int** ***Switch\_To\_Frequency***(**char** **pins**)**;**

/\*\*@fn ISR(BADISR\_vect)

\* @brief Overwrites the default unhandled interrupt vector

\*

\* Overwrites the bad ISR vector to write out an error message

\* to the serial port instead of killing the processor

\*/

**ISR**(**BADISR\_vect**)

{

**Serial\_Print\_String**("Unhandled Interrupt\r\n")**;**

}

/\*\*@var sensor

\* @brief global struct containing the ADC sensor data

\*/

**struct** ADC\_Sensor ***sensor*;**

/\*\*@fn int main()

\*@brief Program entry point

\*/

**int** ***main***()

{

//\*\*\*\*Initialization\*\*\*\*//

DDRD **=** 0xFF**;**

DDRB **=** 0xFF**;** // set the LEDs port as output

PORTB **=** 0xFF**;**

**\_delay\_ms**(1000)**;**

**Init\_Serial\_P**(**BAUD192**, **FRM8**, **STOP1**, **NOPAR**)**;**

**ADC\_Init\_P**()**;**

***sensor*** **=** **ADC\_Sensor\_Construct**("Sensor1", 1, 270, 0)**;**

//\*\*\*\*Functionality\*\*\*\*//

**Step\_2**()**;**

//Step\_3();

//Step\_6();

**return** 1**;**

}

/\*\*@fn unsigned int Switch\_To\_Frequency(char pins)

\*@brief Converts a switch press into a frequency setting

\*@param [in] pins The value of the pins attached to the switches on the STK500

\*@returns A frequency value from 1-100Hz depending on the swtich pressed

\*/

**unsigned** **int** ***Switch\_To\_Frequency***(**char** **pins**){

**pins** **=** **~pins;**

**switch**(**pins**){

**case** 0b0**:**

**return** 1**;**

**break;**

**case** 0b1**:**

**return** 25**;**

**break;**

**case** 0b10**:**

**return** 50**;**

**break;**

**case** 0b100**:**

**return** 75**;**

**break;**

**case** 0b1000**:**

**return** 100**;**

**break;**

**default:**

**return** 1**;**

**break;**

}

**return** 1**;**

}

/\*\*@fn void Step\_2()

\*@brief Code for solving step 2 of the lab

\*/

**void** ***Step\_2***(){

**int** ***ADCTemp*;**

**int** ***count*=**0**;**

**while**(1)

{

**ADCTemp** **=** **ADC\_Get\_Value**(***sensor*.**channel)**;**

**Serial\_Print\_String**(***sensor*.**name)**;**

**Serial\_Print\_String**(", ")**;**

**Serial\_Print\_Int**(**count**, 10)**;**

**Serial\_Print\_String**(", ")**;**

**Serial\_Print\_Int**(**ADCTemp**, 10)**;**

**Serial\_Print\_String**(", ")**;**

**Serial\_Print\_Int**(**ADC\_Calc\_Volts**(**ADCTemp**, 5000), 10)**;**

**Serial\_Print\_String**(", ")**;**

**Serial\_Print\_Int**(**ADC\_Sensor\_Calc\_Value**(**&*sensor***, **ADCTemp**), 10)**;**

**Serial\_Print\_String**("\r\n")**;**

**count++;**

**\_delay\_ms**(100)**;**

}

}

/\*\*@fn void Step\_3()

\*@brief Code for solving step 3 of the lab

\*/

**void** ***Step\_3***(){

**int** ***ADCTemp*;**

**int** ***freq*** **=** 1**;**

**int** ***count*** **=** 0**;**

**unsigned** **char** ***dutyCycle*;**

**Square\_Wave16\_Initialize**(**freq**, 18432)**;**

**while**(1){

**count++;**

**if**(PINC **!=** 0xFF){

**freq** **=** **Switch\_To\_Frequency**(PINC)**;**

**Square\_Wave16\_Set\_Freq**(**freq**, 18432)**;**

}

**ADCTemp** **=** **ADC\_Get\_Value**(***sensor*.**channel)**;**

**dutyCycle** **=** **ADC\_To\_Duty**(**ADCTemp**)**;**

**Square\_Wave16\_Set\_Duty\_Cycle**(**dutyCycle**)**;**

**Serial\_Print\_String**("PWM, ")**;**

**Serial\_Print\_Int**(**count**,10)**;**

**Serial\_Print\_String**(", ")**;**

**Serial\_Print\_Int**((**int**)**dutyCycle**,10)**;**

**Serial\_Print\_String**(", ")**;**

**Serial\_Print\_Int**(**freq**,10)**;**

**Serial\_Print\_String**(", ")**;**

**Serial\_Print\_Int**((**int**)((PIND**&**0b01000000)**>>**PD6),2)**;**

**Serial\_Print\_String**(", ")**;**

**Serial\_Print\_Int**(**ADCTemp**,10)**;**

**Serial\_Print\_String**("\r\n")**;**

}

}

/\*\*@fn void Step\_6()

\*@brief Code for solving step 6 of the lab

\*/

**void** ***Step\_6***(){

**int** ***i*=**0xFF**;**

**while**(**i==**0xFF)

{

**i=**PINC**;**

}

**Timer16\_Initialize**(200, 18432)**;**

**while**(1)**;**}

# Appendix E: Authorship

|  |  |
| --- | --- |
| **Section** | **Author** |
| **Introduction** | Joel Sutherland and Adam Panzica |
| **Methodology** | Joel Sutherland and Adam Panzica |
| **Results** | Joel Sutherland and Adam Panzica |
| **Discussion** | Joel Sutherland and Adam Panzica |
| **Conclusion** | Joel Sutherland and Adam Panzica |