Project 3: Pulse Width Modulation with an ADC source

Jacob Hollenbeck, Boise State University 03/12/2016

Contents

1 Overview **1**

2 Supplies **2**

3 Analog to Digital Converter **2**

3.1 ADC Initialization 3

3.2 ADC Read 3

4 Pulse Width Modulation **4**

5 Compilation **5**

1 Overview

This document demonstrates how pulse width modulation signals can be controlled via analog signals. This is tested and verified using the built in functionalities on an 8-bit microcontroller. Three analog signals are read using the ADC of the microcontroller and control the duty cycle of the PWM signals. The result is the controllable brightness of individual LEDs inside of a Tricolor LED.

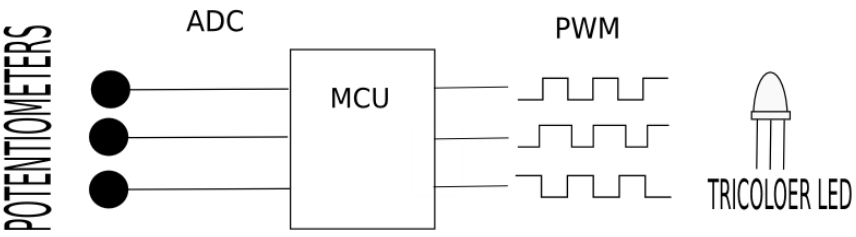

Figure 1: System Overview

An analog signal is created by applying voltage across variable resistance resistors (potentiometers). This signal is read by the MCU and used to adjust the width of the output PWM waves. Theses waves control the brightness of the tricolor LED. This project also displays the results on a VT100 terminal. This is done with the use of the UART component discussed in project 2. The duty cycle percentage is displayed for each PWM signal as well as the analog input signal. An abstract view of the project components can be seen in Figure 2.

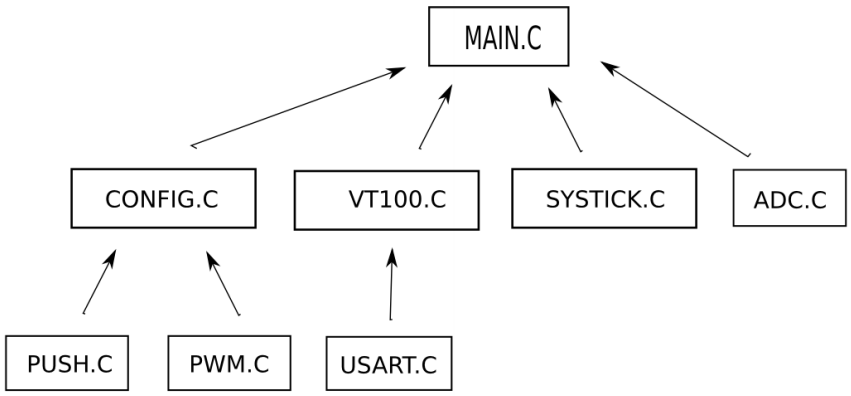

Figure 2: Abstract Overview

From Figure 2 Most of the components in this project have been used or mentioned in previous projects. The Timer has been renamed to systick.c and the UART has been renamed to usart.c. Also a VT100 file has been introduced. This VT100 file contains definitions for vt100 codes, similar to a vt100 library file in project 2. The three new files in this are the adc.c, pwm.c, and config.c. The file config.c is a wrapper for the other files. This file initializes the other components and prints out various labels and other statements to the terminal. The other two files, adc.c and pwm.c are the most important files and will be discussed in depth in this report. Note systick.c, VT100.c, and usart.c are from the code given to us by Professor Planting.

2 Supplies

- Atmega324p Microprocessor PDIP
- Four-prong button
- 220 ohm Resistor (x4)
- Atmel-ICE programmer with ISP cable
- Breadboard
- FTDI Cable
- JTAG Header
- Tri-color LED
- Potentiometer (x3)

3 Analog to Digital Converter

The 8 bit MCU featured has an on-chip, analog to digital converter. This ADC can read up to 8 individual inputs at 10 bit resolution. To find the output reading from an analog signal, the

following equation from sparkfun [1] will be used:

$$\frac{2^{Resolution}}{SystemVoltage} = \frac{ADCReading}{AnalogVoltage} \tag{1}$$

Whereas the system voltage is equal to the specified voltage for this project (5 V), the resolution is 10 and the analog voltage is the input voltage level. So, to use the same example from sparkfun:

$$\frac{1023}{5.0}2.12 = ADCReading = 434 \tag{2}$$

3.1 ADC Initialization

To initialize the ADC, several steps need to be taken. First the ADC is initialized by setting the ADEN bit high. Next, interrupts are enabled as well as auto triggering. By using auto triggering, the adc is read whenever a particular event happens. We will set this to trigger whenever there is a successful compare match for the timer0 unit. Last, the reference voltage is selected. This is used to determine the voltage to be compared. By setting the REFS[1:0] bits = 01, the default voltage is used for the reference. The ADC cannot function when the cpu frequency is high. To help resolve this, a prescaler is used. The prescaler used divides the current clock value by 128. This is shown in the following code:

```
1    // Enable ADC, set autotrig, and enable interrupts
2    ADCSRA |= (1<<ADEN)|(1<<ADATE)|(1<<ADIE);
3    ADMUX  |= (0<<REFS1)|(1<<REFS0);                  // Set AREF
4    ADCSRA |= (1<<ADPS2)|(1<<ADPS1)|(1<<ADPS0);       // Set prescaler to 128
5    ADCSRB |= (0<<ADTS2)|(1<<ADTS1)|(1<<ADTS0);       // Set trig to compare match
```

3.2 ADC Read

To read the adc channels, a function was used called adc_read(uint8_t pin). The user selects the particular channel to read as the parameter. The ADC uses a 5 bit mux to select the input. To select the correct channel, the first 5 bits of the mux register are cleared by ANDing it with the value 0xE0. This leaves the configuration bits alone. The new ADMUX is then OR'd with the input channel selected by the user. This is done as follows:

```
1    // Keep first 3 bits of ADMUX OR with the input channel
2    ADMUX = (ADMUX & 0xE0)|pin;
```

After the correct channel has been selected, the read can take place. This can be done two different ways. The first way is to set the 'start read' bit high and waiting for it to go low. The second way is to set the 'start read' bit high and waiting for the interrupt flag to go low. Because autotriggering is being used, we will wait for the flag to go low. This is done as follows:

```
1    ADCSRA |= (1 << ADSC);              // Set start bit
2    while(ADCSRA & (1 << ADIF));        // Wait for flag bit to clear
```

4 Pulse Width Modulation

A pulse width modulation signal is a square wave with a controllable width. This controllable width is often referred to as the duty cycle. The duty cycle is given in percentage as follows:

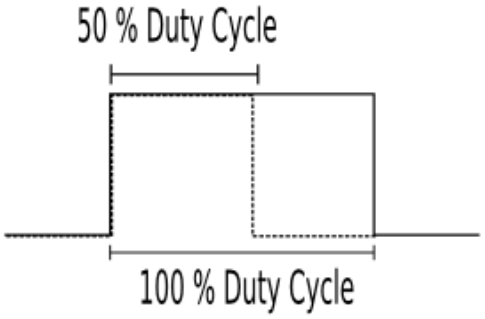

Figure 3: Duty Cycle

The selected MCU has PWM functionalities built into the three available timers. Timer0 and Timer2 are both 8-bit timers with PWM functionalities. Timer1 is a 16-bit timer with pwm functionalities. To use them, the correct mode needs be selected. There are several different PWM modes allowed. Phase correct PWM is dual slope, high resolution. Fast PWM is single slope, high frequency. The type of slope indicates when the timer resets its compare value. For this module, fast PWM is enabled. In C, this is done as follows:

```
    // Set PWM to non-inverting mode
    TCCR0A  |= (1 << COM0A1)|(0 << COM0A0);
    // Set WGM mode to fast PWM mode
    TCCR0B |= (0 << WGM02);
    TCCR0A |= (1 << WGM01)|(1 << WGM00);
```

Now that PWM mode has been enabled, the OCR value controls the duty cycle. By setting the OCR register equal to the ADC signal read from a potentiometer, the duty cycle is controllable by the user. Note that the OCR register is 3 bits wide. The ADC register is 4 bits wide. To prevent rollover, a scalar is applied to the read ADC value. This is done as follows:

```
    // scalar = 4
    OCR0A = adc_read(0)/scalar;
```

Now that the OCR has been enabled and is controllable, the output pin can be wired to input resistor on the tricolor LED. To use PWM for multiple signals, more OCR registers must be enabled. A function to enable an individual compare value would look as follows:

```
void enable_OCR0A()
{
    // Enable outout to pin PB3
    DDRB    |= (1 << PB3);
    // Set PWM to non-inverting phase correct mode
    TCCR0A  |= (1 << COM0A1)|(0 << COM0A0);
}
```

Each timer has only two OCR registers, so a second timer must be initialized as well. In this project, Timer1 is enabled as follows:

```
void pwm1_init()
{
    // Set WGM mode
    TCCR1B |= (0 << WGM13)|(1 << WGM12);
    TCCR1A |= (0 << WGM11)|(1 << WGM10);
    // Prescaler = 64
    TCCR1B |= (0 << CS12)|(1 << CS11)|(1 << CS10);
    // Enable OCR val
    enable_OCR1A();
}
```

5 Compilation

Because multiple files are used in this program, a Makefile is used to compile the .elf file. The linker process was covered in the previous report. To use the Makefile included, change to the project directory and use the following commands:

```
$ make
$ make write
```

The make write command will write the compiled .elf file to the MCU.

References

[1] *Analog to Digital Conversion*. URL: https://learn.sparkfun.com/tutorials/analog-to-digital-conversion.