Timer/Counter Project 3

ECEN 220: Introduction to Embedded Systems
University of Nebraska–Lincoln
March 6, 2021

Name: David Perez

Contents

1 Introduction

For this project we were tasked to experiment with Pulse Width Modulations(PWM) that are made possible with the Arduino nanos. For the specific programs in this report, the fast PWM was used rather than the several others capable on the Arduino nanos. Using the fast PWM we were able to create a waveform of our choice and change the duty cycle as well. We were able to do this by simply changing the high time value in the OCR1A registers which changed how long the pin would be high for.

Another useful opportunity that the PWM creates is it allows us to change the direction of servo motors. By adjusting the high time on our pulse wave we were able to rotate the servo clockwise or counterclockwise. On top of this, we used delays as well as the push botton to allow us to manualy change the motor direction.

2 Program Description

As stated in the introduction, both of the programs in this project use the fast PWM mode on the Arduino nanos. The period that we were aiming for is 20ms (that is 20ms from rising edge to rising edge). In program 1 we first initialized all of the register values and their corresponding addresses. Unfortunatley, the system clock runs far too fast for use to create the the high times that would later be needed in program 2. To get around this, I divided the system clock by a prescalar value of 8. This was derived by the following formula : Prescalar >= F(MCU)/(F(desired) * 40000)

We were also tasked to selectively change the duty cycle which, for convenience, we used the ICR1H and ICR1L registers to change the TOP value of our PWM. we were able to achieve a 20ms period by also changing OCR1AH and OCR1AL registers which determined the high time our PWM. For my project I selected a TOP value of 40,000 and a high time of 10,000 (for 25% duty cycle). The equation that was used to derive this is as follows: 100% * (OCRIA / ICR1)

Moving onto program 2, our objective was to use this wave form and adjust it to suit our servo motors. Using the a good portion of the information from program 1, we had to then create a wave form that had a high time in the range of 700-1500 microseconds for a clockwise rotation and a high time in the range of 1500-2300 microseconds for a counterclockwise rotation. On top of that, we then attached a push button to pin PB0 and when that value was debounced to 0 we would rotate the servo in one of the directions. A side note on that is we also had to make the motor stop and return to its original position after another press had occured and to make sure the motor woldn't prematurely rotate until the rotation had been complete.

2.1 Program 1

When creating this program there were a few important things to keep in mind. The first of which is the prescalar value that we chose. By using the formula mentioned above, I determined that if i wanted a frequency of 50Hz (period of 20ms) then I wouldn't be able to have a prescalar higher than 256 (as well as a prescalar lower that 8). The prescalar I chose was 8 which resulted in me having to assign a value of 40,000 for my ICR1 register. For the TCCR1A and TCCR1B registers, I assigned a value of 0b10 for COMIA to put PB1 in non-inverting mode, a value of 0b1110 for fast PWM mode with a TOP value dependent on ICR1A and a value of 0b010 for CS1 to divide my clock down by a prescalar of 8. With these values in place I was thus able to obtain a duty cycle of 25%, 50% and 75%. Below are images that display the different duty cycles that were generated and thus read using a logic analyzer.

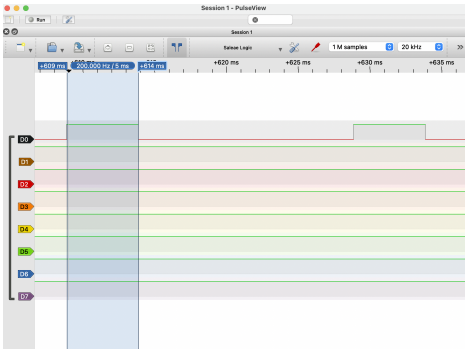

Figure 1: Screen Capture a 25% duty cycle

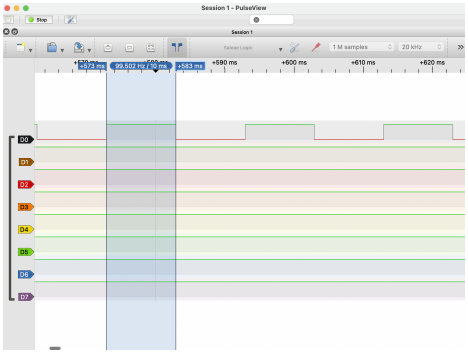

Figure 2: Screen Capture a 50% duty cycle

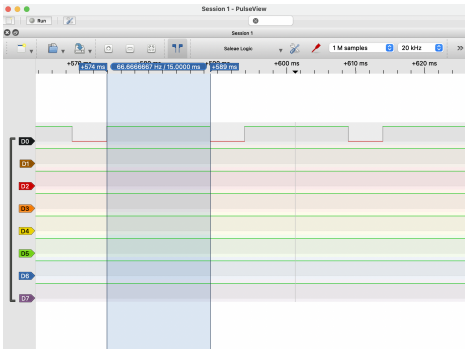

Figure 3: Screen Capture a 75% duty cycle

2.2 Program 2

For program 2, we are using the waveform from program 1 but trimming down our high time to suit our servo motor direction specifications that would be controlled by a pushbutton. To construct this program there was several things to keep in mind. After establishing all the initial register values I then had to construct the logic that would enable the servo to respond to the button presses. First off, I had to debounce the button to make sure I wasn't getting garbage values when reading the pin (PB0) that the button was connected to. Once I did this and detected a state change I then had to decide if the button was being pressed or not. If all these came out to be true and a button had been pressed I could then decide which direction to turn the motor. I accomplished this by using a flag to remember the previous direction the motor had rotated. Based on this value I rotated the servo in the desired direction. A value of 2250 was assigned to OCR1A to rotate the servo clockwise and 3800 to rotate the motor counterclockwise.

The period for this program remained the same throughout but the high time and duty cycle changed between based on one of the two values assigned to OCR1A. When rotating clockwise, the duty cycle came out to 5.625% which is pictured in Figure 4 below.

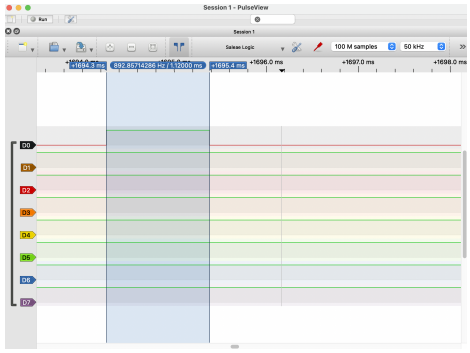

Figure 4: Servo Rotating Clockwise with a 5.625%duty cycle

When rotating the servo counterclockwise the duty cyle came out to be 9.5%. This is pictured below in Figure 5.

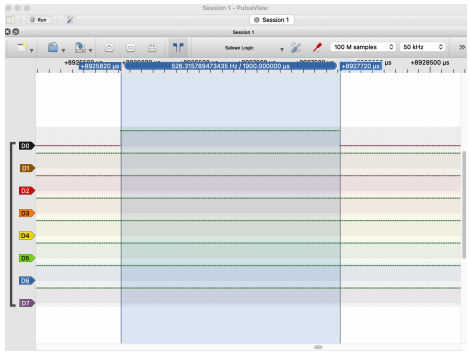

Figure 5: Servo Rotating Clockwise with a 9.5%duty cycle

When the motor was not turning the duty cycle would thus be %. The reason behind this was because in order to stop the motor from rotating I had assigned a 0 as the high time in the OCR1A register which meant that the pin would never be high when it wasn't rotating. This is visually depicted in Figure 6 below.

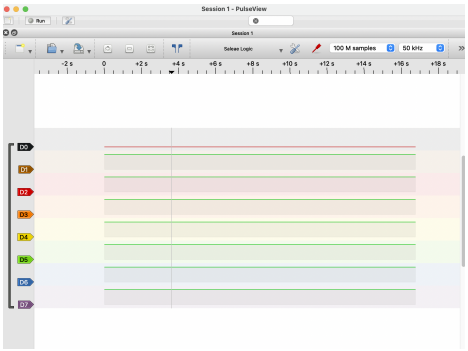

Figure 6: Servo idle with 0% duty cycle

Lastly I would like to talk about the power that is being used by the servo motor when the motor is in motion and when it is not. To calculate this I mutliplied the meausured Arduino nano voltage of 4.74 by the current being drawn from the motor. When the motor is idle it draws roughly 18 milli-watts(mW) of power. When in motion, the motor draws significantly more power in the range of 40mW to about 150mW. I included a table below because the power drawn can be very inconsistent.

Measured Current of Moving Motor(mA)	**Calculated Power(mW)**
140.4	665.5
52.5	248.85
88.5	419.49
103	488.22
142.5	675.45
133.4	632.316
129.5	613.83
93.2	441.77

Table 1: Program 2 Current and Power Consumed by Motor in Motion

3 Conclusion

Both programs in this project offered some very important information on waveforms. With program 1 we were able to grasp an understanding on how to create a PWM. By using the timer of the Arduino nano we were able to create these waveforms. From there we were able to use that information and generate waveforms of our choosing and even adjust the duty cycle of these waveforms. With that we were able to use this information and apply it to a very important component in electrical engineering and that is the electric motor. With our version of an electric motor, the servo motor, we were able to change its direction based on the high times of the wave form that we gave it. Then, with the help of a push button we could manually change the servo motors direction. All in all, we learned how to generate wave forms, change their duty cycles, and even apply this to an electric motor to get it to respond in the manner that we desired.

4 Appendix

```
1  /** Includes **/
2  #include <stdint.h>
3
4
5
6  /** Memory mapped register defines **/
7  #define REG_DDRB (*((volatile uint8_t*) 0x24))
8
9  #define REG_TCCR1A (*((volatile uint8_t*) 0x80))
10 #define REG_TCCR1B (*((volatile uint8_t*) 0x81))
11 #define REG_OCR1AH (*((volatile uint8_t*) 0x89))
12 #define REG_OCR1AL (*((volatile uint8_t*) 0x88))
13
14 #define REG_ICR1H (*((volatile uint8_t*) 0x87))
15 #define REG_ICR1L (*((volatile uint8_t*) 0x86))
16
17 //Registers for debouncing function
18 volatile uint8_t* pPINB;
19 volatile uint8_t* pDDRB;
20 volatile uint8_t* pPORTB;
21
22 /** Function declarations **/
23 int8_t debouncePin(volatile uint8_t* mmrPINx, uint8_t bitToRead);
24 void myHardDelay(uint32_t delayInMsec);
25
26
27 int main()
28 {
29   //assign register values to variables
30   pPINB = 0x23;
31   pDDRB = 0x24;
32   pPORTB = 0x25;
33
34   //Seting up registers for debouncing function
35   // set PB0 as an input
36   *pDDRB = 0x00;  //0b00000000
37
38   // Set PB0 as having a internal pullup resistor enabled
39   *pPORTB = (*pPORTB) | 0x01; //0b00000001
40   /*
41   Specifications:
42   1.Output a square wave on pin OC1A, which is PB1
43   2. Make the duty cycle customizable, and make it proportional to OCR1A / TOP (aka non-
         inverting mode)
44   3. Use the fast PWM mode that sets the TOP value as ICR1A
45   4. Use a Timer/counter 1 clock source of f_MCU / 1.
```

```
46      Note: this will give a period of T = TOP * prescalar / f_MCU
47      */
48
49      /*
50      Determine bit fields
51      WGM1[3:0] = 0B1110 for fast PWM mode where TOP = ICR1A
52      COM1A[1:0] = 0b10 for non-inverting mode on pin OCR1A / PB1
53
54      // determine WHAT PRESCALAR AND ICR1A VALUE TO USE
55      CS1[2:0] = 0b010 for a clock prescalar of / 8
56      Now our timer will increment every 8/16MHz = 5x10^(-7) 50 micosecond??
57      */
58
59      //Set the registers for Fast PWM
60
61      //Configure PB1 as an output
62      REG_DDRB = 0x02;
63
64      //TCCR1A contains COM1A[1:0] bit field and the WGM1[1:0] partial bit field.
65      // COM1A[1:0] cat 0b0000 cat WGM1[1:0]= 0b10 == 0b10000010 == 0x82
66      REG_TCCR1A = 0x82;
67
68      //Configure the TOP value aka ICR1AH and ICR1AL
69      uint16_t top_value = 40000;
70      REG_ICR1H = top_value >> 8;
71      REG_ICR1L = top_value & 0x00FF;
72
73      //set a small high time to prevent the servo from moving once the TCCR1B register is
            initialized
74      uint16_t high_time = 1;
75      REG_OCR1AH = high_time >> 8;
76      REG_OCR1AL = high_time & 0x00FF;
77
78      //TCCR1B = 0b000 cat WGM1[3:2] cat CS1[2:0] == 0b000 cat 0b11 cat 0b010 == 00011010 == 0x1A
79      REG_TCCR1B = 0x1A;
80
81      //initialize flag for the servo direction where a 0 rotates clockwise and a 1 rotates counter
            clockwise
82      int8_t servoDirection = 0;
83      while(1)
84      {
85        //debounce the pushbutton
86        int8_t state1 = debouncePin(0x23, 0x01);
87
88        //check if the button is done bouncing
89        if(state1 == (-1) | state1 == 1)
90        {
91          int8_t state2 = *pPINB & 0x01;
92          //check for a state change on PB0
93          if(state1 != state2)
94          {
95            //check if the button is being pushed
96            if(state1 == (-1))
97            {
98              //rotate the servo in a clockwise directoins
99              if(servoDirection == 0)
100             {
101
102               for(volatile uint32_t z=0; z < 100000; z++)
103               {
104                 high_time = 2250;
105                 REG_OCR1AH = high_time >> 8;
106                 REG_OCR1AL = high_time & 0x00FF;
107               }
108               high_time = 0;
109               REG_OCR1AH = high_time >> 8;
110               REG_OCR1AL = high_time & 0x00FF;
111               //change the servo direction next time loop occurs
```

```
112               servoDirection = 1;
113             }
114
115             //rotate the servo counterclockwise
116             else if (servoDirection == 1)
117             {
118               for(volatile uint32_t j=0; j < 100000; j++)
119               {
120                 high_time = 3800;
121                 REG_OCR1AH = high_time >> 8;
122                 REG_OCR1AL = high_time & 0x00FF;
123               }
124               high_time = 0;
125               REG_OCR1AH = high_time >> 8;
126               REG_OCR1AL = high_time & 0x00FF;
127
128               ///change the servo direction next time loop occurs
129               servoDirection = 0;
130             }
131           }
132         }
133       }
134     }
135   }
136
137
138   int8_t debouncePin(volatile uint8_t* mmrPINx, uint8_t bitToRead)
139   {
140     //read PB0
141     uint8_t firstDebouceSample = *mmrPINx & bitToRead;
142
143     //Delay for 30 milliseconds
144     myHardDelay(30);
145
146     uint8_t secondDebouceSample = *mmrPINx & bitToRead;
147
148     if(firstDebouceSample == secondDebouceSample)
149     {
150       //The pin is successfully debounced
151       if(firstDebouceSample == 0x00)
152       {
153         //The pin is being pushed (debounced and low)
154         return -1;
155       }
156       else
157       {
158         // The pin is not being pushed (debounced and high)
159         return 1;
160       }
161     }
162     else{
163       // The pin may be bouncing
164       return 0;
165     }
166   }
167
168   void myHardDelay(uint32_t delayInMsec)
169   {
170     volatile uint16_t delayCount;
171     volatile uint16_t i;
172     delayCount = delayInMsec * 373;
173     for(i=0;i <delayCount; i++){
174
175     }
176   }
```

Listing 1: Program 1

```
/** Includes **/
#include <stdint.h>



/** Memory mapped register defines **/
#define REG_DDRB (*((volatile uint8_t*) 0x24))

#define REG_TCCR1A (*((volatile uint8_t*) 0x80))
#define REG_TCCR1B (*((volatile uint8_t*) 0x81))
#define REG_OCR1AH (*((volatile uint8_t*) 0x89))
#define REG_OCR1AL (*((volatile uint8_t*) 0x88))

#define REG_ICR1H (*((volatile uint8_t*) 0x87))
#define REG_ICR1L (*((volatile uint8_t*) 0x86))

/** Function declarations **/
void myHardDelay(uint32_t delayInMsec);


int main()
{
  /*
  Specifications:
  1.Output a square wave on pin OC1A, which is PB1
  2. Make the duty cycle customizable, and make it proportional to OCR1A / TOP (aka non-
      inverting mode)
  3. Use the fast PWM mode that sets the TOP value as ICR1A
  4. Use a Timer/counter 1/8 clock source of f_MCU / 8.
  Note: this will give a period of T = TOP * prescalar / f_MCU
  */


  /*
  Determine bit fields
  WGM1[3:0] = 0B1110 for fast PWM mode where TOP = ICR1A
  COM1A[1:0] = 0b10 for non-inverting mode on pin OCR1A / PB1

  // determine WHAT PRESCALAR AND ICR1A VALUE TO USE
  CS1[2:0] = 0b010 for a clock prescalar of / 8
  Now our timer will increment every 8/16MHz = 5x10^(-7) 50 micosecond??
  */

  //Set the registers

  //Configure PB1 as an output
  REG_DDRB = 02;

  //TCCR1A contains COM1A[1:0] bit field and the WGM1[1:0] partial bit field.
  // COM1A[1:0] cat 0b0000 cat WGM1[1:0]= 0b10 == 0b10000010 == 0x82
  REG_TCCR1A = 0x82;

  //Configure the TOP value aka ICR1AH and ICR1AL
  uint16_t top_value = 40000;
  REG_ICR1H = top_value >> 8;
  REG_ICR1L = top_value & 0x00FF;

  // Set the initial duty cycle to 25%. Do this before the timer is turned on,
  //which happens when CS1 is no longer 0b000 (this is
  //done when TCCR1B is configured, so do it before then!)

  uint16_t high_time = 10000; //to create a 25% duty cycle
  /* uint16_t high_time = 20000; //to create a 50% duty cycle
  uint16_t high_time = 30000; // to create a 75% duty cycle */

  REG_OCR1AH = high_time >> 8;
  REG_OCR1AL = high_time & 0x00FF;
```

```
67
68      //TCCR1B = 0b000 cat WGM1[3:2] cat CS1[2:0] == 0b000 cat 0b11 cat 0b010 == 00011010 == 0x1A
69      REG_TCCR1B = 0x1A;
70
71
72      //continuously cycle the fast PWM
73      while(1)
74      {
75        high_time = 10000;
76        REG_OCR1AH = high_time >> 8;
77        REG_OCR1AL = high_time & 0x00FF;
78      }
79    }
80
81    void myHardDelay(uint32_t delayInMsec)
82    {
83      volatile uint16_t delayCount;
84      volatile uint16_t i;
85      delayCount = delayInMsec * 373;
86      for(i=0;i <delayCount; i++){
87      }
88    }
```

Listing 2: Program 2