Microavionics Lab 4 (5067)

Sage Herrin
University of Colorado Boulder
Due 10/13/20

This lab served as an introduction to timers and interfacing with the LCD provided with the EasyPIC board, as well as to solidify and build upon the current level of understanding of logic and operations in assembly. The final outcome was an assembly program that controlled a PWM output and varied it from a 5% to a 10% duty cycle in increments of 1%, controlled via push button, while writing the current output duration of the high portion of the PWM to the LCD.

A. Lab Questions

1

The compiler directive 'DB' serves to both reserve and assign places in program memory that are used to write to the LCD. In this lab it is used as part of the initialization of the LCD, as well as to write initial phrases to the LCD, such as "Hello" and "World", and later on "ASEN 5067" as well as a PW variable.

2

When the lab4_example.asm code was compiled and run, the value of 0x33 is given to the label LCDstr, determined using the both the program memory view window to obtain the opcode and the disassembly listing to determine the value given to the label by finding the corresponding opcode.

3

The nonsense instructions in the disassembly window pertaining to the lines with the db directives are created essentially from two different aspects. One being that due to the endian-ness of our board (little endian), the hex db values are interpreted not only as binary values that are listed as commands in the disassembly listing, but are also listed backwards. For example, in the first db line, where there is written first 0x33 and then 0x32, in the corresponding command to that line in the disassembly listing there is a 3233, because the db values were interpreted as binary commands, and were also flipped due to the endian-ness. This also shows why db lines are put at the very end of a file, because the nonsense commands that are generated from them are not used and hence don't effect anything in the script.

4

The general structure of the code pertaining to the timing of the main loop events, i.e. RD4 and PWM timing, is all controlled by timers in this lab. The layout of the timers/main loop timing on a high level consists of four timers, which are Timer0, Timer1, Timer3, and Timer5. Timer0 is reconfigured several times during initialization to work as a 1 second timer, a 10 millisecond timer, and a 250 millisecond timer, in that order. Once configured as a 250 millisecond timer, Timer0 is left alone for the rest of the duration of the code and is used in the main loop to control the "alive" led, RD4.By repeatedly resetting this timer, configured as a 250 millisecond timer, the RD4 LED will toggle on for 250 milliseconds, then off for 250 milliseconds, and will do this for as long as the main loop is running.

After the 250 millisecond timer (Timer0) is set, Timer1 is set via a call to a subroutine called "lowloop". The lowloop timer controls the low portion of the PWM signal generated on the board, and is initially set as a 19 millisecond timer. After Timer1 is set, there is a call to another subroutine within lowloop called "highloop". The Highloop subroutine sets a timer (Timer3) which controls the high portion of the generated PWM signal. Unlike the 250 millisecond Timer0 and the 19 millisecond Timer1, which are set with a call to a subroutine the program does not idle until the timer goes off, the 1 millisecond highloop timer (Timer3) DOES idle until the timer goes off. Once this timer completes, the PWM has gone through one full period (19ms + 1ms = 20ms). This triggers a reset of the lowloop timer (Timer1), and returns to where lowloop was called in the main loop. The next and final step of the main loop is a call to a switch checking subroutine, similar to that used in lab 3. This subroutine also uses a timer, Timer5, which is simply used to implement

a switch debounce by waiting 15 milliseconds after a button is initially toggled before it does anything. Toggling a switch/button (RD3) leads further into the switch checking subroutine. The logic implemented in this subroutine essentially determines if a button has been pushed and released, and if it has, then it increases the duration that the RC2 pin is high and decreases the duration that it is low, thus increasing the duty cycle of the PWM signal while keeping the period at a constant 20 milliseconds. This is a high-level synopsis of how timers in the code are used to control the alive LED and pwm signal, but full details can be seen in assembly code/comments.

All timer duration calculations took the same form and following the same process. Starting with the desired timer duration, for example 250 milliseconds, the desired duration is divided by the time per instruction cycle. For our board, using a 16 MHz clock, with clock cycles per instruction cycle ⇒ 4 MHz instruction cycle ⇒ 250e-9 seconds per instruction. Dividing the desired timer duration by the instruction frequency gives the number of instructions necessary for said timer duration, e.g. (250e-3)/(250e-9) = 1e6 instructions. Since 1e6 > 65563, a prescalar value must be determined. Dividing the necessary number of instructions by the maximum number of bits in two bytes, 65536, gives the prescalar. In this example, (1e6)/(65536) = 15.2588. This number is rounded up to be sure there is sufficient room for the full timer duration to be stored. Rounding this up gives 16 as the prescalar. Dividing the number of necessary instructions by the prescalar ⇒ (1e6)/(16) = 62500. This values is then subtracted from 65536 to give the total number of "counts" the timer needs to make before it goes off, incorporating the effect of the prescalar. Thus the value 65536-62500 = 3036 is saved in the TMR0H and TMR0L bytes, which tells Timer0 how long to run for. All timers written in this lab use this same calculation to determine their initial duration.

5

Figures 1 through 6 below show the generated PWM signal cycling from a 5% to a 10% duty cycle while maintaining a period of 20 milliseconds.

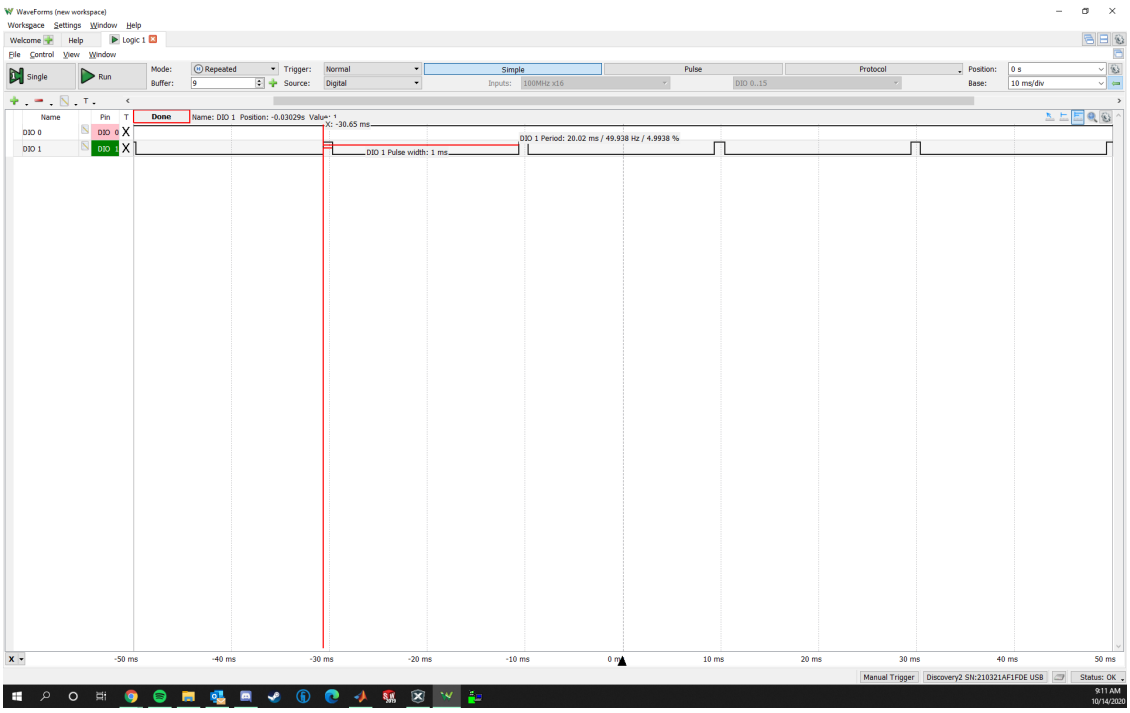

Fig. 1 1ms high PWM signal

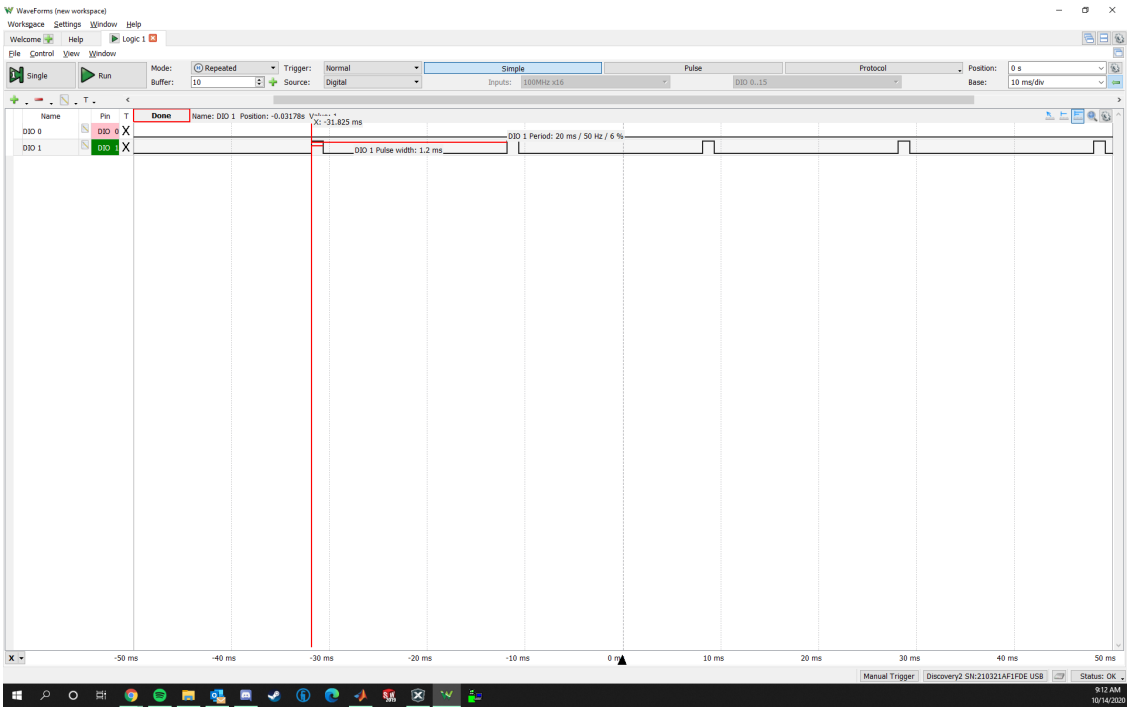

Fig. 2 1.2ms high PWM signal

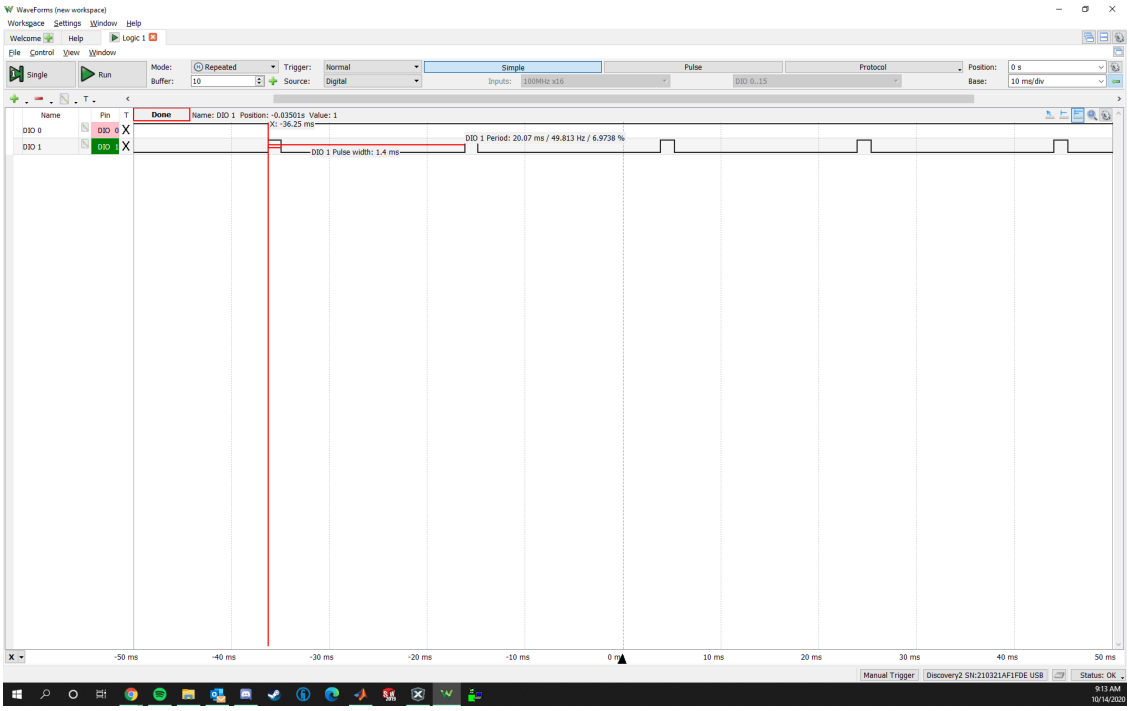

Fig. 3 1.4ms high PWM signal

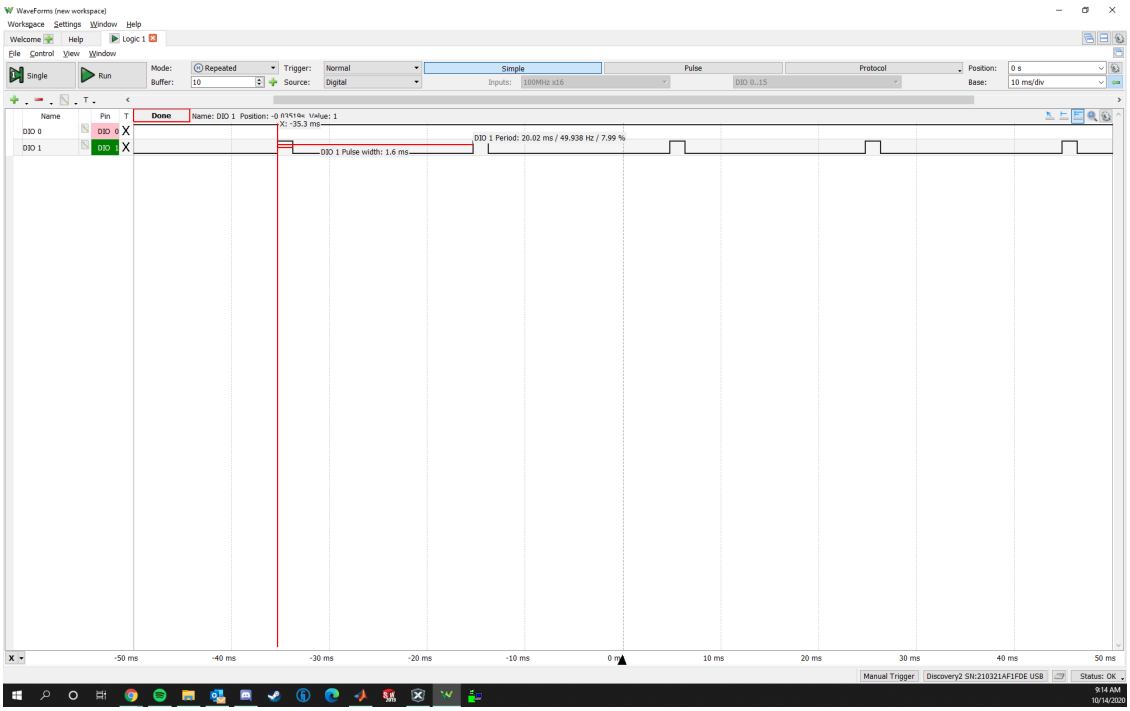

Fig. 4 1.6ms high PWM signal

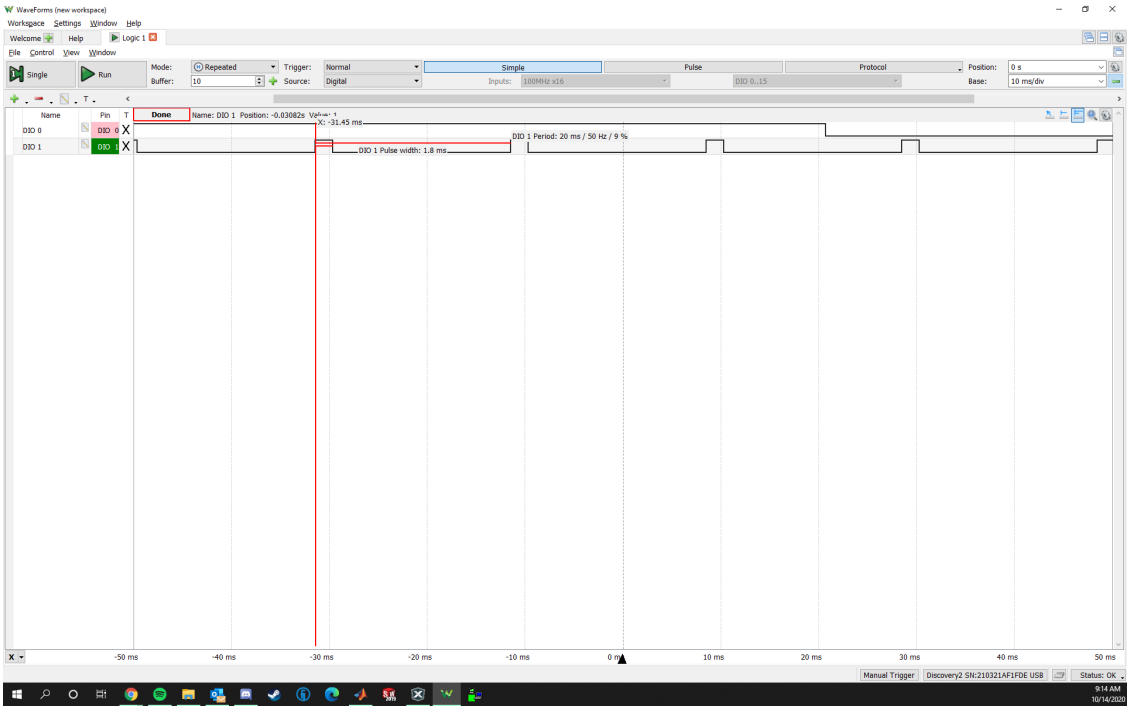

Fig. 5 1.8ms high PWM signal

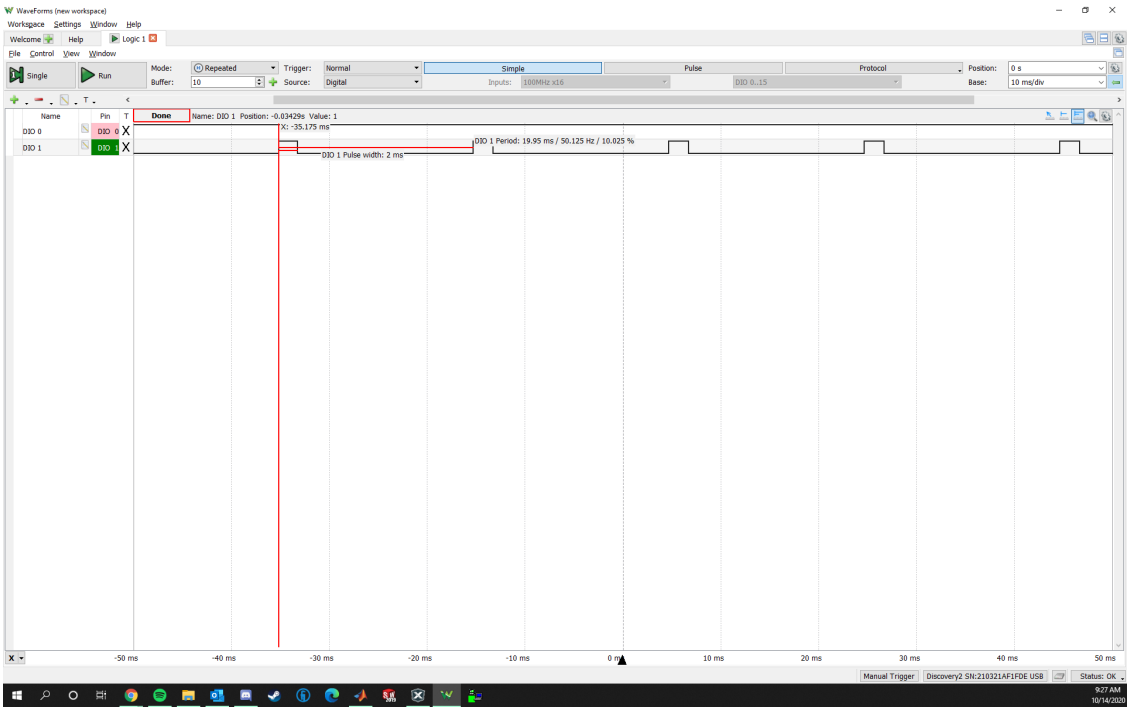

Fig. 6 2ms high PWM signal

6

For the 5067 requirements of incrementing the PWM signal from 5% duty cycle to 10% duty cycle, once the servo is connected to the board and the PWM output, increasing the duty cycle actuates the servo by a small amount, slowly increasing in total rotation, until it reaches some maximum rotation at when the duty cycle is 10%. Changing the duty cycle again at this point results in setting the duty cycle back to 5%, which rotates the servo back the other way.

7

This type of signal, i.e. a PWM with a 5-10% duty cycle and a period of 20 milliseconds, provides the desired output from the servo because the servo expects to be updated with a high value every 20 milliseconds, and the range in duty cycle covers the range of actuation of the servo very well. This combination of parameters of the PWM signal is what provides the expected/desired output to the motor.