Lab#3 Servomotor – open loop control

Leonardo Fusser (1946995)

Objectives:

- C code a PID algorithm to control the position of a BDC motor in open loop mode.
- Test the control loop in simulation mode.
- Test the control loop in target mode.

Hardware: Explorer 16/32 with pic32 PIM, 1 USB cable, pmodHB5 H-bridge, BDC motor with encoder, 9V DC supply.

To hand in

- Answer to the lab's questions and submit on Teams.
- An indented and commented C code. See software requirement of the previous labs.

Not respecting the instructions will result in a refusal and a re-submission.

Structure of the template:

Table 1: multi file system template

Files	Content
Files to populate:	
`main.c`	This is where you initialise all your resources. The `pid_control` task or the `ol_control` task must run in the super loop.
`pid_control.c`	This is where you implement your PID controller.
`ol_control.c`	This is where you implement your Open Loop controller.
Hardware and software configuration:	
`initBoard.c`	Includes its own header file: `initBoard.h` Contain all functions related to initializing the board: i.e. `#pragma, initIO(),initTimer(),...`
`configuration.h`	Includes all common macros needed by many files. Includes the important macro to swap from simulation mode to target mode and vis-versa: `#define SIMULATION`
Motor simulation model:	
`DCMotor_model3.c`	This is the mathematical model representing the simulated BDC motor.
Resource files:	
`Tick_core.c`	Tick library implemented using the 32-bit core timer.
`adc32.c`	Contains useful ADC converter functions.
`console32.c`	Combined LCD and UART Drivers for PIC32.
`pv_measure.c`	Measure the position of the motor by reading the rotary encoder pulses. Implemented using an CN interrupt.
`pwm.c`	Function and ISR to implement soft PWM. Uses Timer3 ISR to implement the soft PWM.

Requirements:

- You must write an opened loop controller to control the position of a motor (AKA servomotor control).
- The code is implemented on a bare-metal microcontroller (no OS involved).
- The sampling time must be 10mS.
- To keep it real time, no blocking delays are allowed.

Motor characteristics:

- BDC type.
- Control resolution (CR): 0.888 tics/º (or 320 tics/rev).

Set point:

- In simulation mode, the DMCI plugin simulates the slider pot.
- The DMCI slider pot will be used to set the PWM.
- The DMCI slider pot is connected to the ADC converter.
- The ADC converter has a 0-1023 range (10bits).

PWM:

To set the PWM and direction, you must use the `set_pwm()` API.
Examples:

set_pwm(10000) sets the PWM to 100.00% CW direction.
set_pwm(8734) sets the PWM to 87.34% CW direction.
set_pwm(0) sets the PWM to 0.00%.
set_pwm(-5000) sets the PWM to 50.00% CCW direction.

Target mode or simulation mode:

To change the mode, you must comment in/out the following macro in the `configuration.h` file:

```
/* Macro to swap from simulation mode to target mode and vis-versa   */
/* To run target mode, comment out this line.                        */
#define SIMULATION
```

Program structure:

Your program should have the following structure:

```
int main(void){

      /* Initialize all required resources here */

      while (1){
            /* runs in Open Loop mode */
            ol_control;
      }
}
```

```
void ol_control(void){

      if(TickDiff(stamp) > TICKS_TEN_MS){

            /* Do read ADC                          */

            /* Do read re-scale the read value    */

            /* Do set the PWM                       */

            /* Display info to LCD – Target mode only  */
      }
}
```

Lab Work

GitHub:

- Make sure you are logged into GitHub. Copy-paste the following URL to accept an invitation for this lab:

  https://classroom.github.com/a/xwnJOG4w

You will get a private new repository for this lab. You will push your project to this repo at the end of this lab.

https://github.com/advanced-programming/lab3_servo_511_a21-yourname

Part 1: simulation mode

DMCI plugin Install the DMCI plugin: see appendix.

Enable the simulation mode by uncommenting the following macro inside configuration.h file.

```
#define SIMULATION
```

In simulation mode you must disable the following resources:

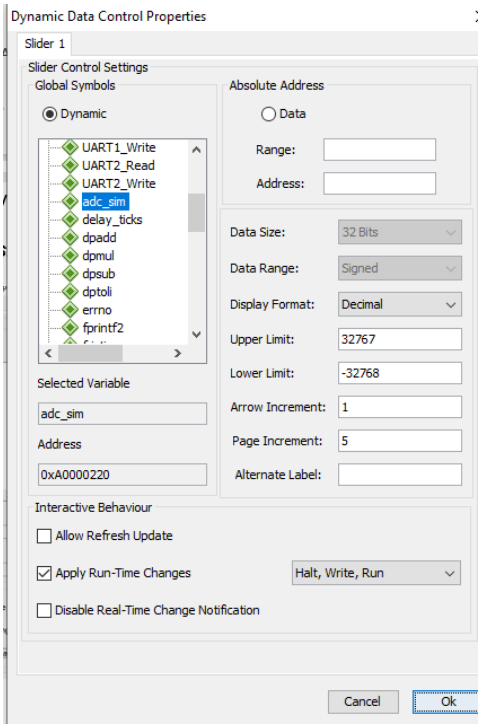
```
initPWM();
initPV_measure();
```

Also. in simulation mode, you can run the loop faster by changing the sampling time to 1mS.

In this part, you are going to control the motor in open loop mode by changing the PWM value.

Program the DMCI potentiometer to simulate the 10-bit ADC converter.

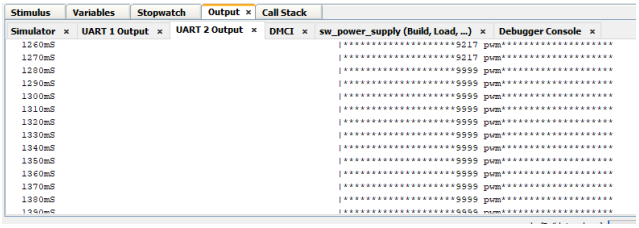
The name of the variable is `adc_sim`:



Change the scale of the ADC to obtain the following PWM ranges:

Position 0% (left) PWM = -100.00% (-10000).
Position 50% (middle) PWM = 0.00% (0).
Position 100% (right) PWM = +100.00% (+10000).

The PWM and the position (in tics) will be automatically printed on UART1 and UART2:



Run the code and then test by moving the potentiometer wiper.
The position PV should decrease or decrease depending on the direction of the motor.
A positive PWM means CW and a negative PWM means CCW.

Note: to save time, you can run the loop faster by changing the sampling time to 1mS.

Does the position change instantly when you change the PWM?

- The position does not change instantly when the PWM is changed and there is a significant delay before the actual PWM value is set. The delay is around 10mS, which is due to our sampling time also being fixed at this time interval.

Why is there a delay between the PWM command and the actual position PV of the motor?

- As mentioned above, the delay occurs between the PWM command and the position (PV) of the motor because of the set sampling time. While in simulation mode (using MPLAB X IDE simulator), the sampling time is set at 10mS. This means there will always be a delay of around 10mS between the PWM command and the position (PV) of the motor. This is not the case when target mode is used. Instead, when in target mode (using Explorer 16/32 board), the sampling time is much faster, which in turn means that the delay between the PWM command and the position (PV) of the motor is much less (this is significant for real-time operation). In target mode, the sampling time is set at 1mS.

Try to set the position PV at 160 tics exactly.
Is it feasible? Is it hard to achieve? Explain why.

- It is very difficult, and most likely, impossible to set the position (PV) of the motor at 160 tics exactly. It's simply because a human being cannot control the position of the motor (PV) very precisely the same way a computer might be able to do. This is the one major flaw with an open-loop control system. In a closed-loop system, the position (PV) of the motor can be set to a very close value because a very good and fine-tuned algorithm exists so that a computer can replace the job of the human being.

What is the PWM value when the position stabilizes near 160 tics?

- The measured PWM value when the position stabilizes near 160 tics is around 48 PWM. This is very close to our original goal (was expecting 50 PWM).

Give a demo.

Part 2: Target mode

In part2, you are going to implement your loop control task on a real motor.

Disable the simulation mode by commenting out the following macro inside configuration.h file.

```
//#define SIMULATION
```

Control resolution (CR)

In target mode you must enable the following resources:

```
initIO();
initPWM();
initPV_measure();
initADC();
LCDInit();
```

Using your open loop control mode task, set the PWM to 0.

Inside the control task and using the get_pv() API, display the tics on the LCD screen:

PV:

Hint: do not display all the time but display every 10mS.

Manually spin the motor to find out the number of tics per revolution.

- It takes around 16 ticks for the motor to rotate one full revolution. Knowing this, we can also derive the PPR of this particular motor (PPR refers to the total number of pulses-per-revolution). Since we know that our microcontroller counts a total of 16 pulses for one full revolution of the motor, we know that the PPR of our motor must be around 16. We also can also find out the CPR of this particular motor by doing the following calculation:

$$Motor\ CPR = 4 * PPR = 4 * 16 = 64$$

Since this is a quadrature encoder, the CPR of the motor can be found by multiplying the PPR by a constant of 4. In our case, the motor CPR is 64.

Looking back at previous labs, when this particular motor's characteristics was given, we can see that the above information is correct.

By observing the motor, find out the gearbox ratio of the motor.

- It takes around 310 ticks for the gearbox to rotate one full revolution. We can get a rough estimate of what the gearbox ratio of the motor is by doing the following calculation:

$$Gearbox\ ratio = \frac{Gearbox\ full\ rotation\ (in\ ticks)}{Motor\ full\ rotation\ (in\ ticks)} = \frac{\sim 310}{\sim 16} = \sim 19$$

Therefore, we can safely assume that the motor has a gearbox ratio of around 1:19 based on what we found in the above calculation.

Looking at the information given in previous labs, we can see that for the particular motor that we are using, the gearbox ratio is indeed 1:19.

Open loop control

In target mode, the slider potentiometer replaces the DMCI slider.

Now, you must display the PWM value along with the position PV on the LCD:

```
PV:
PWM:
```

Test by moving the potentiometer wiper. The position should decrease or decrease depending on the direction of the motor.

You must take three screenshots. Each screenshot contains 2 signals: **Dir** and **En**.

	Screenshot 1	**Screenshot 2**	**Screenshot 3**
PWM	About 80%	About 50%	About 0%
Direction	CCW	CW	CW

Give a demo to the teacher.

Refer to the next three screenshots on the following pages.

Screenshots:

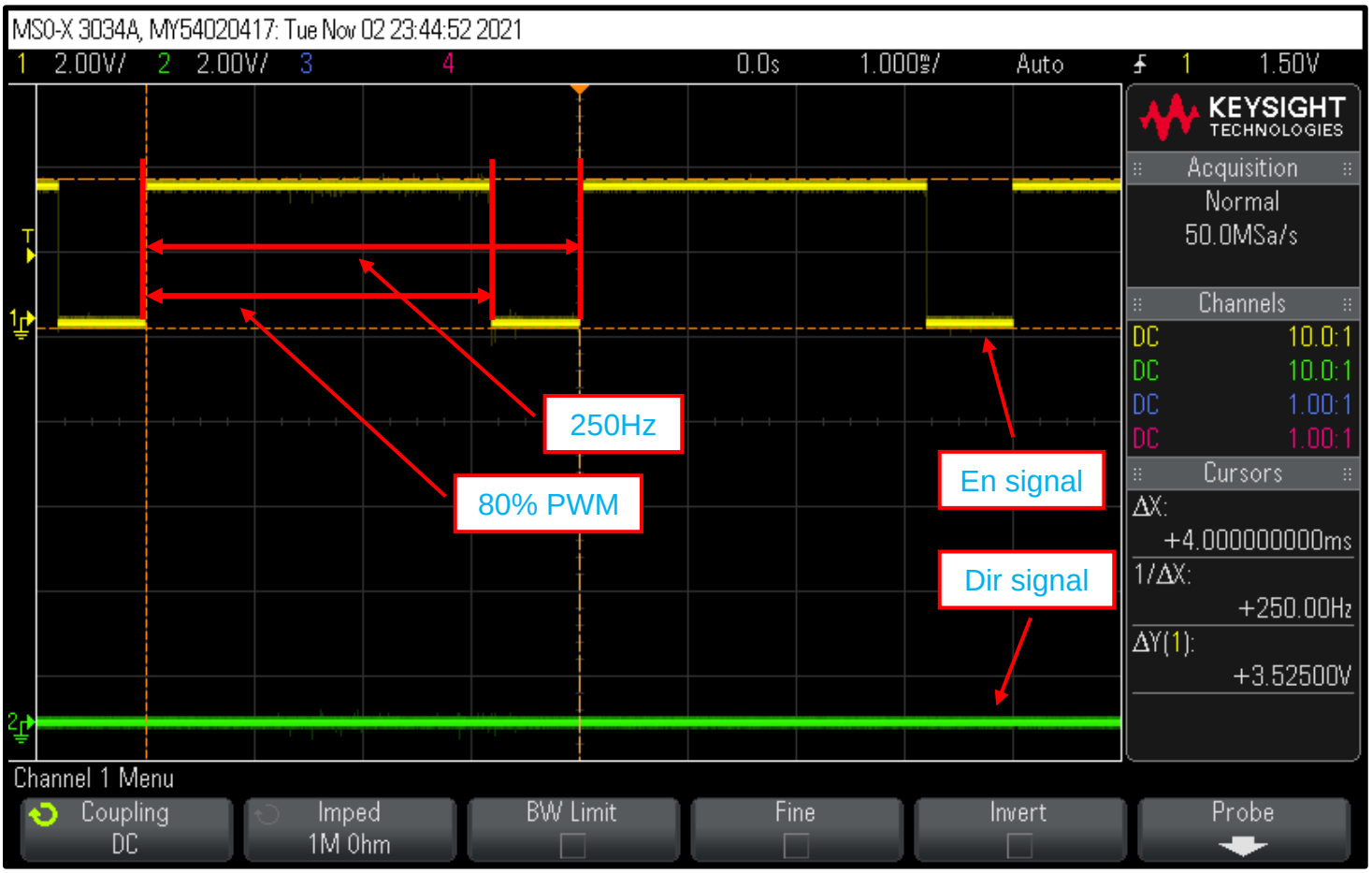


Figure 1. Screenshot above shows when the PWM value is set at 80% and when the direction of the motor is spinning in the CCW (counterclockwise) direction. Even without this information, we would still know that the PWM value is set at roughly 80% and that the direction of the motor is CCW. This is because we can see in the En signal, that around 80% of the time, the signal is high, and that the remaining 20% of the signal is low. Therefore, we know that the PWM must be set at around 80%. We also know that the motor is spinning in the CCW direction because the Dir signal is low, and in our case, this means that the motor must be spinning in the CCW direction. The 250Hz En signal is generated from the timer ISR within the pwm.c file. En signal and Dir signal applied to corresponding En and Dir pins on pmod h-bridge adapter.

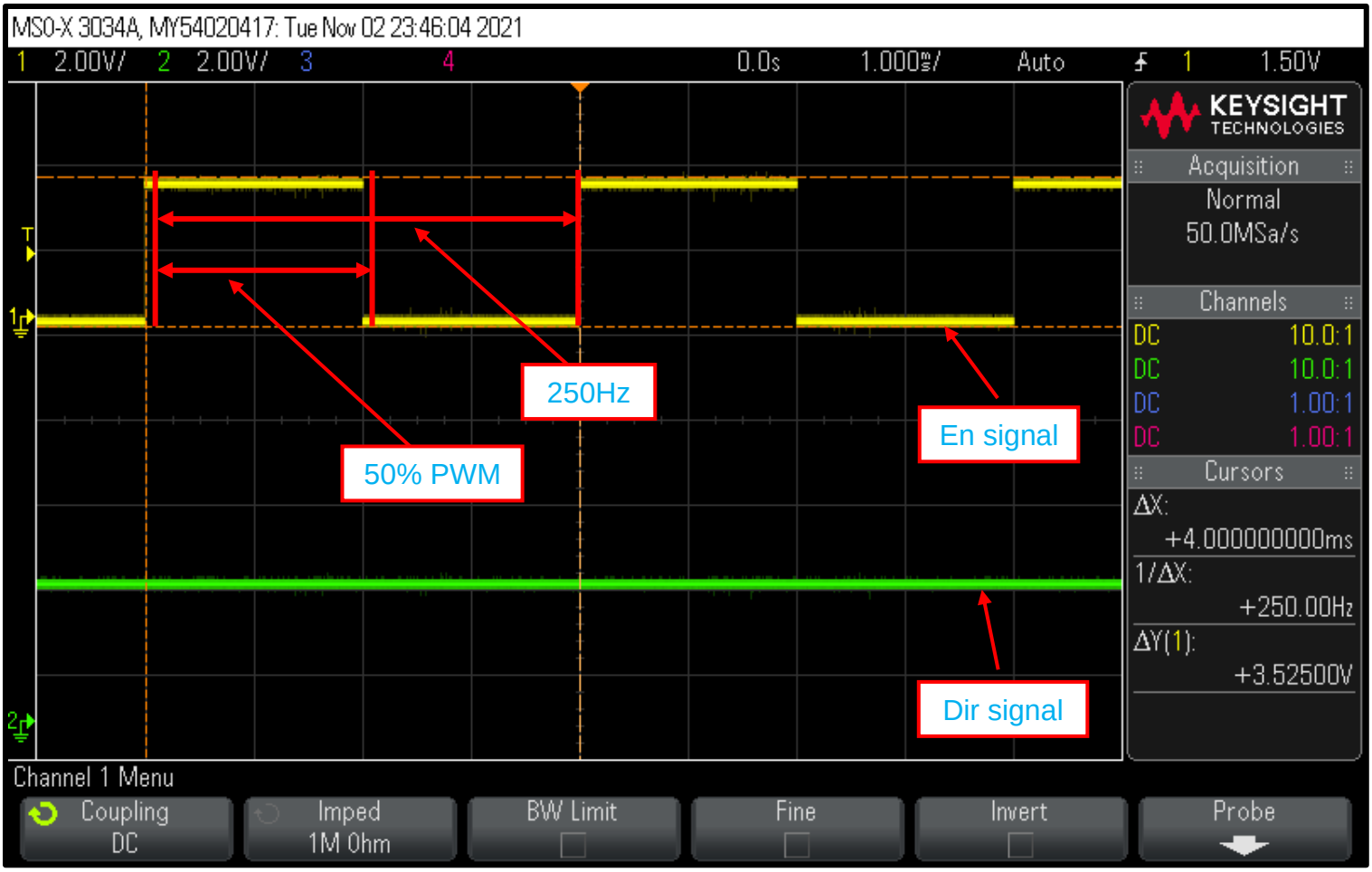


Figure 2. Screenshot above shows when the PWM value is set at 50% and when the direction of the motor is spinning in the CW (clockwise) direction. Same as before, we would still know that the PWM value is set at roughly 50% and that the direction of the motor is CW if we had not been given this information beforehand. This is because we can see in the En signal, that around 50% of the time, the signal is high, and that the remaining 50% of the signal is low. Therefore, we know that the PWM must be set at around 50%. We also know that the motor is spinning in the CW direction because the Dir signal is high, and in our case, this means that the motor must be spinning in the CW direction. The 250Hz En signal is generated from the timer ISR within the pwm.c file. En signal and Dir signal applied to corresponding En and Dir pins on pmod h-bridge adapter.

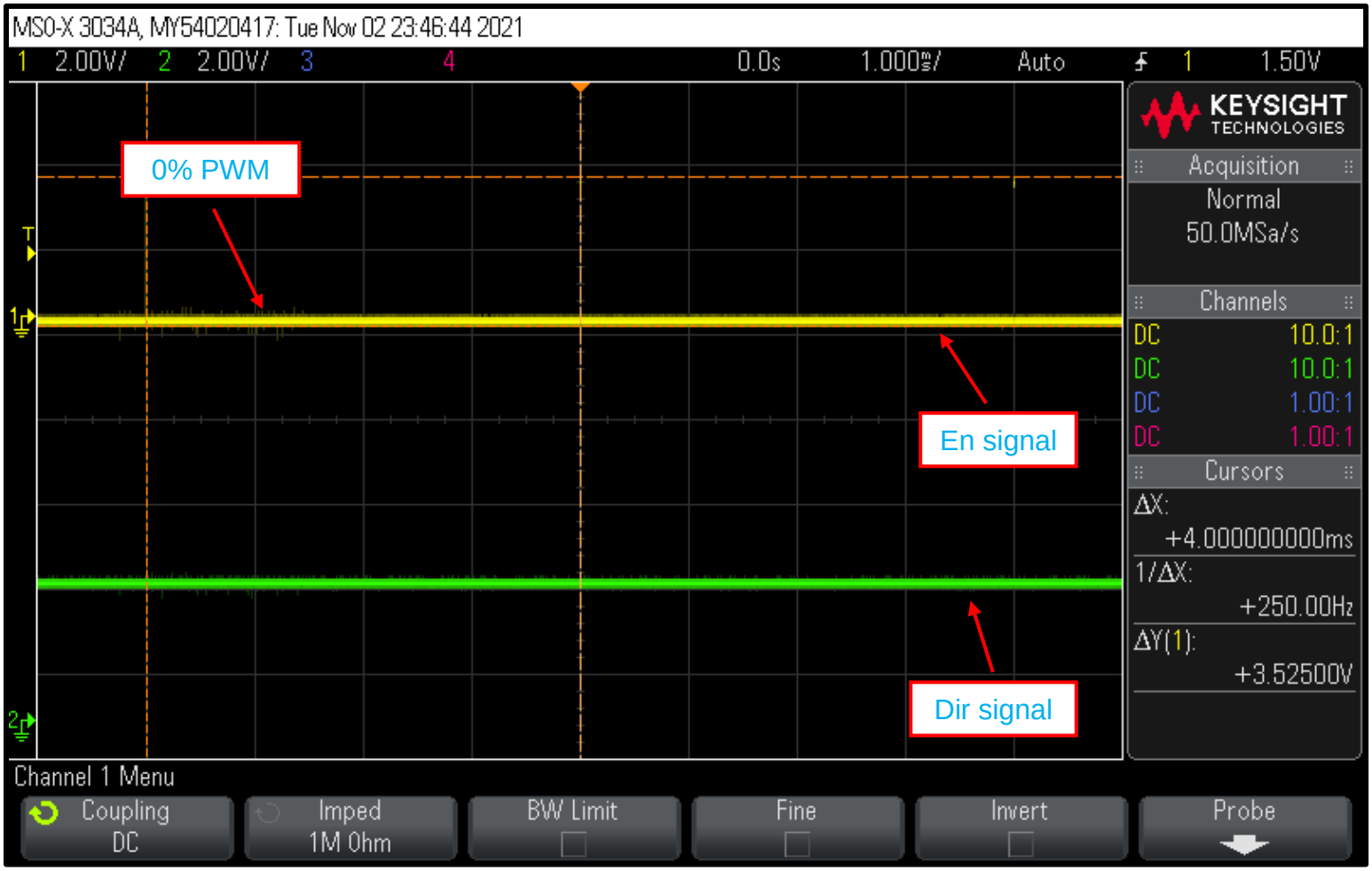


Figure 3. Screenshot above shows when the PWM value is set at 0% and when the motor is not spinning. Same as before, we would still know that the PWM value is set at roughly 0% and that the motor is not spinning in either direction (CW or CCW) if we had not been given this information beforehand. This is because we can see in the En signal, that for the entire signal duration, the signal is neither high nor low or both (it is just a flat line). Therefore, we know that the PWM must be set at around 0%. Regardless of if the Dir signal is high or low for this situation (for the screenshot above it shows high), we know that the motor is not spinning in either direction (CW or CCW) because we already determined that the PWM is set at around 0%. En signal and Dir signal applied to corresponding En and Dir pins on pmod h-bridge adapter.

After lab questions:

1- Explain what you learned in this lab. What went wrong and what did you learn from your mistakes. Give specifics please.

- Since there was not a lot of programming involved to implement the open-loop control system in MPLAB using C programming language, coding was not much of an issue. The issue that was evident with me was that I did not quite understand how an open-loop control system worked, yet alone how a closed-loop control system worked. More specifically, I was not aware how the system was able to take my input (using DMCI or slider on Explorer 16/32 board) and convert it and keep a somewhat constant PWM signal. Also, I was not quite aware how the pmod h-bridge adapter worked, and how the En and Dir pins controlled the speed and direction of the DC motor connected to the M+ and M- pins on the pmod h-bridge. Once some further research was done based on these two issues that were encountered, I understood how an open-loop systems takes a user's input and convert's it to a somewhat stable PWM output. Also, I now know how a typical h-bridge works and how the En and Dir pins in a typical h-bridge drive a DC motor.

Appendix

API definitions

/* Initializes the CN interrupt that reads the rotary encoder */
```
void initPV_measure(void)
```

/* returns the motor position in tics */
```
int get_pv(void)
```

/* Initializes Timer3 and its ISR to implement a soft PWM */
```
void initPWM(void)
```

/* Sets the PWM.
 * The parameter takes a range from -10000 to +10000.
 * A negative value represents a CCW direction
 * and a positive value represents a CW direction.
 * A value of 10000 represents a duty cycle of 100.00% CW direction.
 * A value of 0 represents a duty cycle of 0.00%.
 * A value of -10000 represents a duty cycle of 100.00% CCW direction.
 */
```
void set_pwm(int on)
```

/* initialize the ADC for single conversion in 10-bit mode, select Analog input pins */
```
void initADC(void)
```

/* Reads the ADC value of the channel specified by the parameter */
/* The channel parameter for the explorer16/32 PIC32MX795F512L is 2 */
/* In simulation mode, reads the adc_sim variable */
```
int readADC( int ch)
```

DMCI plugin

Install the DMCI plugin: Tools-> Plugins