ECE 341 - Lab #10

Collin Heist

August 20, 2019

Contents

Listings

1 Introduction

The purpose of this lab is to use the PIC32 to measure the frequency of events. For this particular lab, we'll be spinning a motor connected to the board and using outputs from hall effect sensors, as well as the input capture peripheral, to measure the rotational frequency of the motor.

In this lab, we'll be using two 'new' peripherals. One is the output compare (PWM) peripheral we'll be using to power the motor at various speeds; and the second (actually new) peripheral is the input capture peripheral. This peripheral will trigger on a variable number of pulse edges on the input pin. This will be used so that when the hall effect sensors on the motor turn on / off, the rise or fall of that signal will trigger the input capture interrupt, allowing us to analyze the timing of the event.

2 Implementation

Just like all the other labs, the first and most important function is the system initialization. This is shown below in Listing 1.

Listing 1: System Initialization

```
void system_init() {
  Cerebot_mx7cK_setup();
  PORTSetPinsDigitalOut(IOPORT_B, SM_LEDS);
  PORTSetPinsDigitalOut(IOPORT_D, BIT_1);
  LATBCLR = SM_LEDS;
  LATDCLR = BIT_1;

  init_LCD();
  reset_clear_LCD();

  PORTSetPinsDigitalIn(IOPORT_D, BIT_3 | BIT_12);
  mIC5ClearIntFlag();
  OpenCapture5(IC_ON | IC_CAP_16BIT | IC_IDLE_STOP |
    IC_FEDGE_FALL | IC_TIMER3_SRC |
    IC_INT_1CAPTURE | IC_EVERY_FALL_EDGE);
  ConfigIntCapture(IC_INT_ON | IC_INT_PRIOR_3 |
    IC_INT_SUB_PRIOR_0);

  mCNOpen(CN_ON, (CN8_ENABLE | CN9_ENABLE), 0);
  mCNSetIntPriority(1);
  mCNSetIntSubPriority(0);
  unsigned int x = PORTReadBits(IOPORT_G, BTN1 | BTN2);
  mCNClearIntFlag();
  mCNIntEnable(1);

  OpenTimer3(T3_ON | T3_SOURCE_INT | T3_PS_1_256, 0xFFFF);
  mT3SetIntPriority(2);
```

```
    mT3SetIntSubPriority(2);
    mT3IntEnable(1);

    init_pwm(40, 1000);

    INTEnableSystemMultiVectoredInt();
}
```

Once again, I will be using the LCD to output the current status of the speed readings, as well as the PWM output based off the button status'. The next section of code initializes the input capture module. Bit 3 and 12 are the two tachometer inputs (even though we'll only be using one). For this program, I'll be using input capture #5, with an associated 16-bit timer that triggers on falling edges, is tied to timer 3, and has no prescale counter. Like previous labs, the change notice peripheral is also configured for buttons 1 and 2 so that I can change the PWM output in real-time. Timer 3 is used with the input capture peripheral, and is set to the highest possible value in the period register, with a prescale value of 256. I will not show the timer 3 interrupt routine, but it simply toggled **LEDC** and clears the interrupt flag. Finally, the PWM is initialized for a 40% duty cycle and 1,000 Hz frequency.

The specific setup for the input capture is quite complicated but very important. We use a 16-bit timer because timer 3 is only 16-bits, it triggers on falling edge but this is structurally equivalent to triggering on rising edges (just with a phase shift for the first reading). The peripheral can be configured to trigger the interrupt on 1, 2, 4, or 16 of these edges and for purposes of this lab it is triggered every edge. The interrupt is given a priority level of three so that when the motor completes a rotation, this timing evaluation happens immediately. This might have additional consequences because the ISR for this peripheral is quite long compared to the others, meaning some timing interrupts could be less accurate in this current scheme.

Button presses in this program are structurally the same as the previous lab. Each different button combination corresponds to a different duty cycle for the motor. The only difference is that this lab just uses the top line of the LCD to display the current duty cycle. That code is shown below in Listing 2.

Listing 2: Change Notice ISR

```
void __ISR(_CHANGE_NOTICE_VECTOR, IPL1) CNIntHandler(void) {
  LATBINV = LEDB;
  hw_ms_delay(20);
  unsigned int buttons = read_buttons();
  unsigned int pwm_cycle = decode_buttons(buttons);
  set_pwm(pwm_cycle);

  char pwm_str[8] = {'\0'};
  sprintf(pwm_str, "%u\%", pwm_cycle);

  char lcd_str[50] = "PWM Cycle: ";
  strcat(lcd_str, pwm_str);
```

```
    set_cursor_LCD(FIRST_LINE_START);
    put_string_LCD("                ");
    set_cursor_LCD(FIRST_LINE_START);
    put_string_LCD(lcd_str);

    LATBINV = LEDB;
    mCNClearIntFlag();
}
```

After reading the buttons and obtaining the corresponding duty cycle, a string is generated with the updated value. Since the LCD is shared between the background and foreground tasks, I ensure the location of the cursor on the LCD is correct by setting it to the beginning of the first line, clearing that line by writing 16 spaces (as the LCD is 16 characters wide), resetting the location and *finally* writing the formatted string to the LCD. **LEDB** is then toggled at the end of the interrupt for debugging / monitoring purposes, just before the interrupt flag is cleared.

The newest foreground task is the input capture module. This interrupt routine is triggered when the motor sensor's are triggered on / off. Therefore, this following code (Listing 3) is what actually performs the speed calculation:

Listing 3: Input Capture ISR

```
void __ISR(_INPUT_CAPTURE_5_VECTOR, IPL3) IC5Handler (void) {
    LATBINV = LEDD;
    unsigned int input_buffer[4] = {0};
    static unsigned short int new_capture = 0;
    static unsigned short int old_capture = 0;
    unsigned short int delta_t = 0;

    ReadCapture5(input_buffer);
    new_capture = input_buffer[0];
    delta_t = new_capture - old_capture;
    old_capture = new_capture;

    float rev_per_sec = 10.0E6 / 256.0 / (float) delta_t;
    unsigned int i = 0;
    for (i = BUFFER_LEN-1; i > 0; i--)
        rps_buffer[i] = rps_buffer[i-1];
    rps_buffer[0] = rev_per_sec;

    mIC5ClearIntFlag();
}
```

The variable **input_buffer** is an array of unsigned integers, despite us only caring about a single variable from the timer on the input capture module. This is because there can be at most four values read from the input capture peripheral at once, and so an entire buffer needs to be passed in at once. The variables for the new and old capture times are **static** because their values need to persist between instances of the interrupt triggering. Because the time difference between the tachometer outputs correspond to a full rotation of the motor, two triggers of the interrupt are required to obtain a reasonable reading for the RPS.

ReadCapture5() fills the buffer with the value of the timer at the time of the interrupt trigger. That value is stored as the new capture time, and the corresponding δt is then calculated. This equates to how long between a full rotation of the motor. The next section of code converts that raw difference in the timer register to the current revolutions per second, and then stores that inside the buffer at position 0. To keep this buffer changing in order to calculate a rolling average, all values of the buffer are shifted down an element before this addition occurs. This buffer is declared as a global variable (with values of zero) and a length defined in the header file, which I used 32 values for this particular lab.

The calculation for the RPS from a given timer count difference is shown below:

$$RPS = \frac{f_{PB}}{t_{ps}} \cdot \frac{1}{t_{diff}}$$

The code in Listing 3 exactly matches this above function, with the only difference being I used pre-defined values for f_{pb} and t_{ps}, as they are constant throughout the program's operation.

With all of these functions written, the **main()** function is implemented as follows:

Listing 4: Main Program Loop

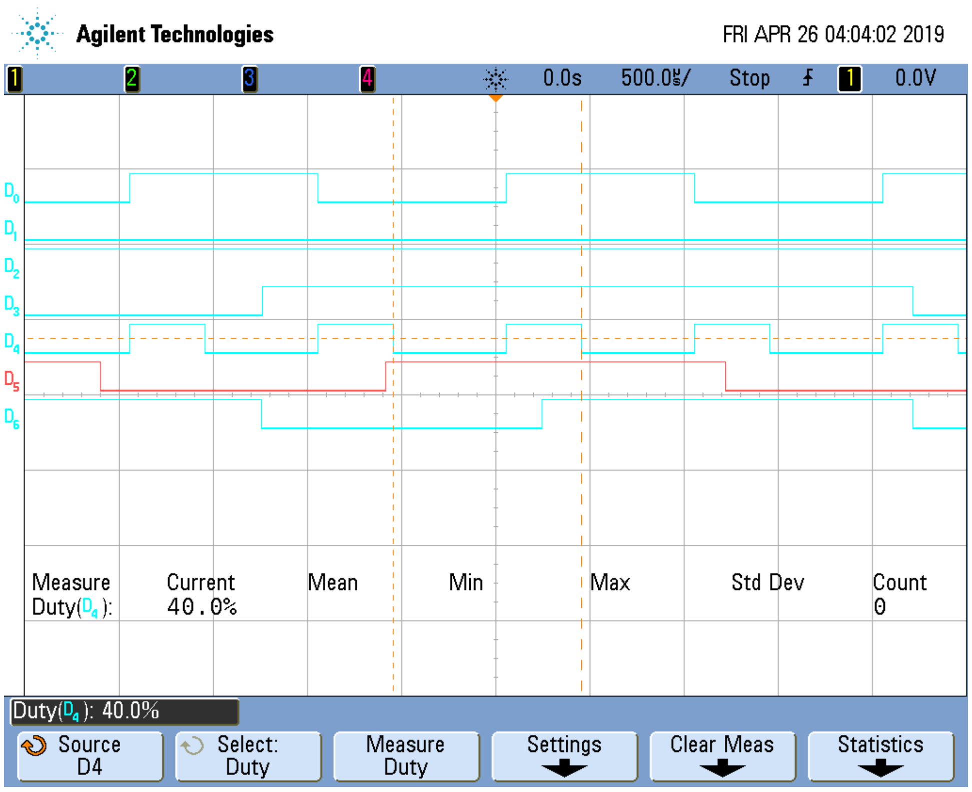
```c
int main() {
  system_init();

  while (1) {
    hw_ms_delay(100);
    mCNIntEnable(FALSE);

    float avg = 0;
    unsigned int i = 0;
    for (i = 0; i < BUFFER_LEN; i++)
      avg += rps_buffer[i];
    avg /= BUFFER_LEN;

    char lcd_str[16] = {'\0'};
    sprintf(lcd_str, "Average: %06.2f", avg);
    set_cursor_LCD(SECOND_LINE_START);
    put_string_LCD("                ");
    set_cursor_LCD(SECOND_LINE_START);
    put_string_LCD(lcd_str);

    mCNIntEnable(TRUE);
  }

  return 1;
}
```

Every one hundred milliseconds, the change notice interrupts are disabled (to prevent interference with the LCD), the current average value of the previous **BUFFER_LEN** values in the RPS buffer are calculated. A string is then constructed that displays the average RPS value inside this buffer. This is placed on the second line of the LCD (in a similar way as the previous duty cycle).

Obviously the way I calculated the moving average is a bit slow for larger RPS buffers, as the entire buffer is iterated over each loop of *main*; however this is a very simple and easy to implement average for the purposes of this lab. I chose to use a rolling buffer (whose values are shifted down each time a new value is added) because this is also very easy to implement, and does not deal with any pointer logic, or modulating around the buffer's length with index values.

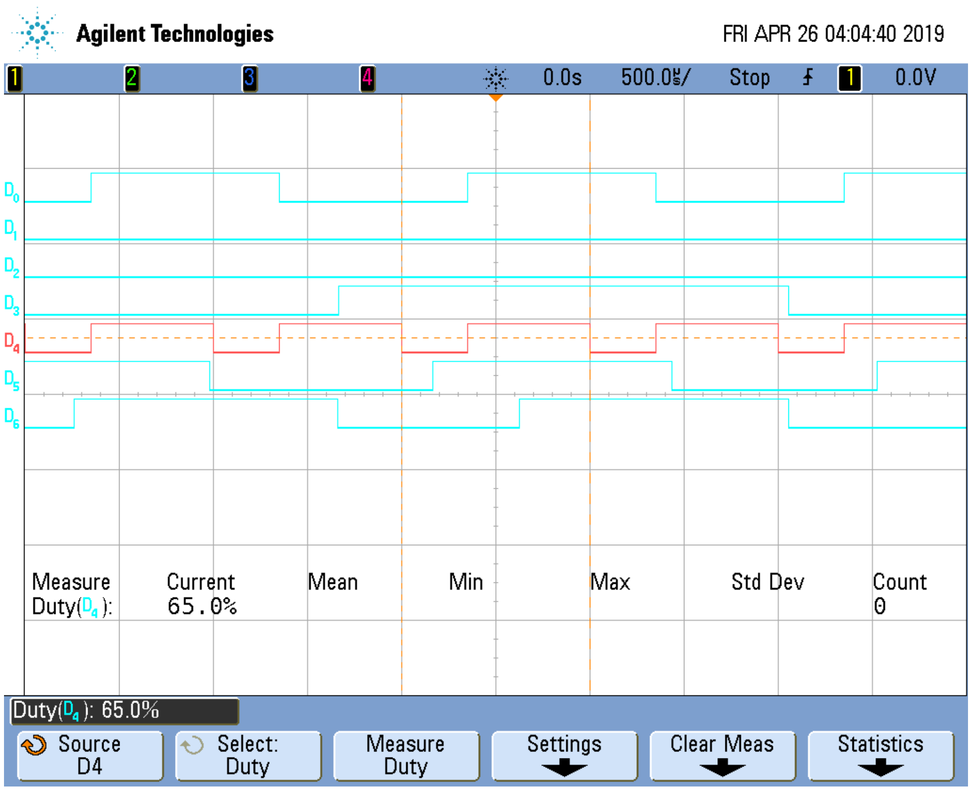
3 Testing and Verification

Testing for this lab was very straightforward. Each separate task utilizes a different LED (such as D in the input capture ISR, etc.), so hooking each of these probes onto the oscilloscope verified all of these values at the same time. The captures for the four button combinations are shown in Figures 1, 2, 3, and 4. In all of the following captures: D_0 is **LEDA** (the timer 2 ISR), D_1 is **LEDB** (the change notice ISR), D_2 is **LEDC** (the timer 3 ISR), and D_3 is **LEDD** (the input capture ISR).



```
Measure    Current  Mean  Min  Max  Std Dev  Count
Duty(D4):  40.0%                             0
Duty(D4): 40.0%
```

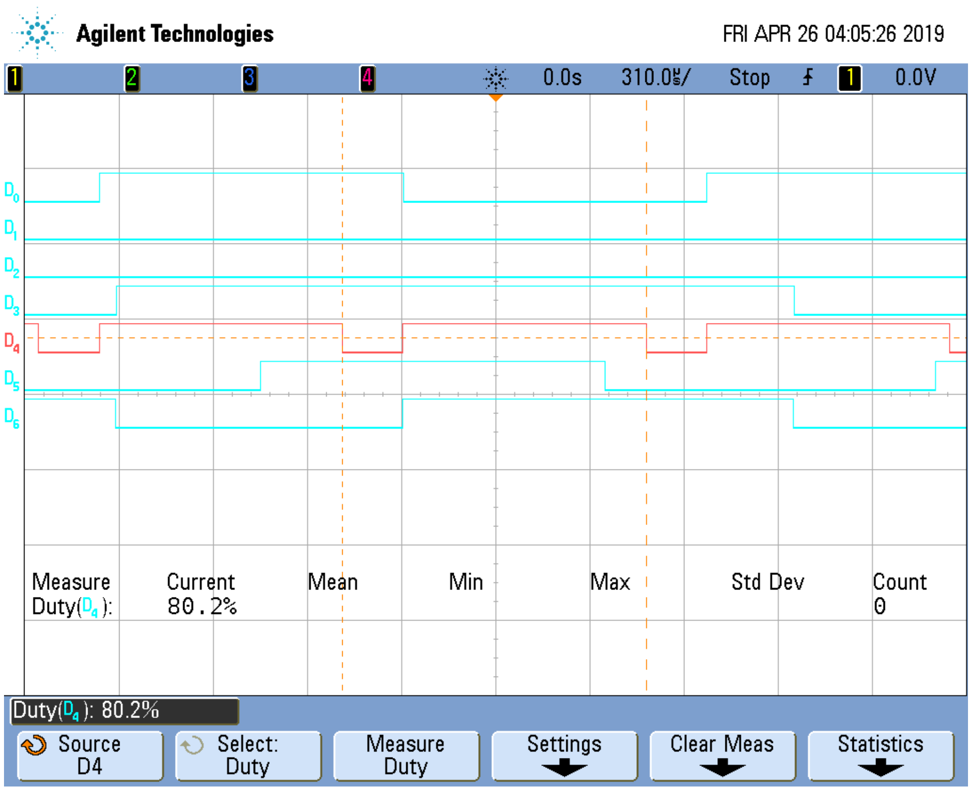
Figure 1: 40% Duty Cycle



Measure	Current	Mean	Min	Max	Std Dev	Count
Duty(D4):	65.0%					0

Duty(D4): 65.0%

Figure 2: 65% Duty Cycle



Measure	Current	Mean	Min	Max	Std Dev	Count
Duty(D4):	80.2%					0

Duty(D4): 80.2%

Figure 3: 80% Duty Cycle

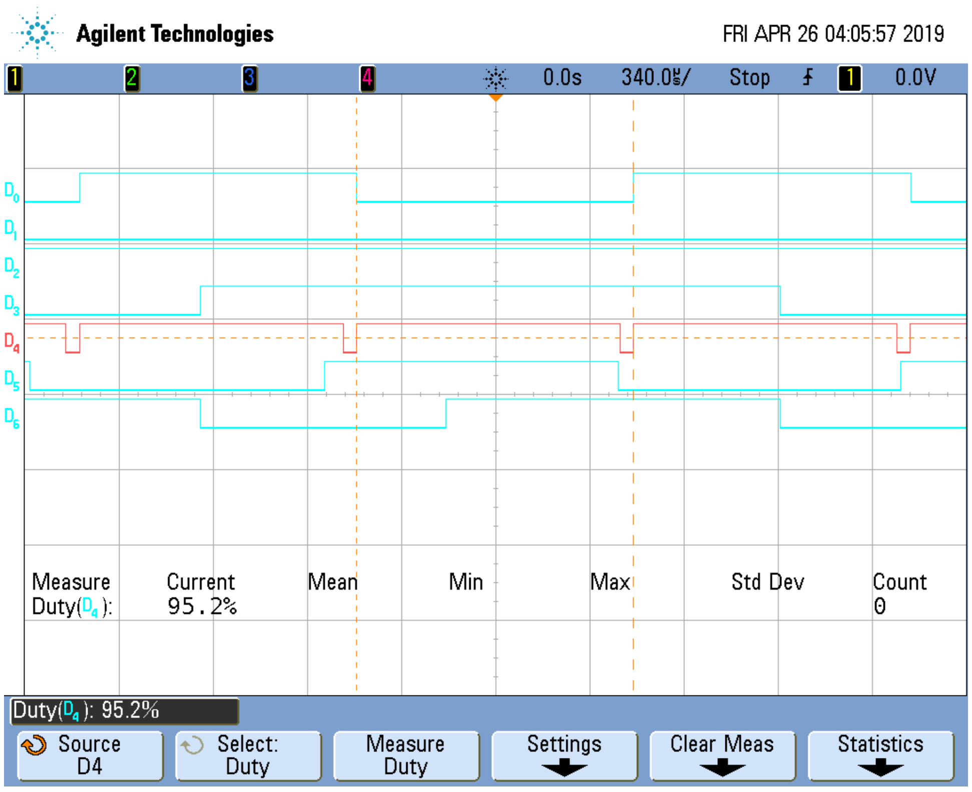
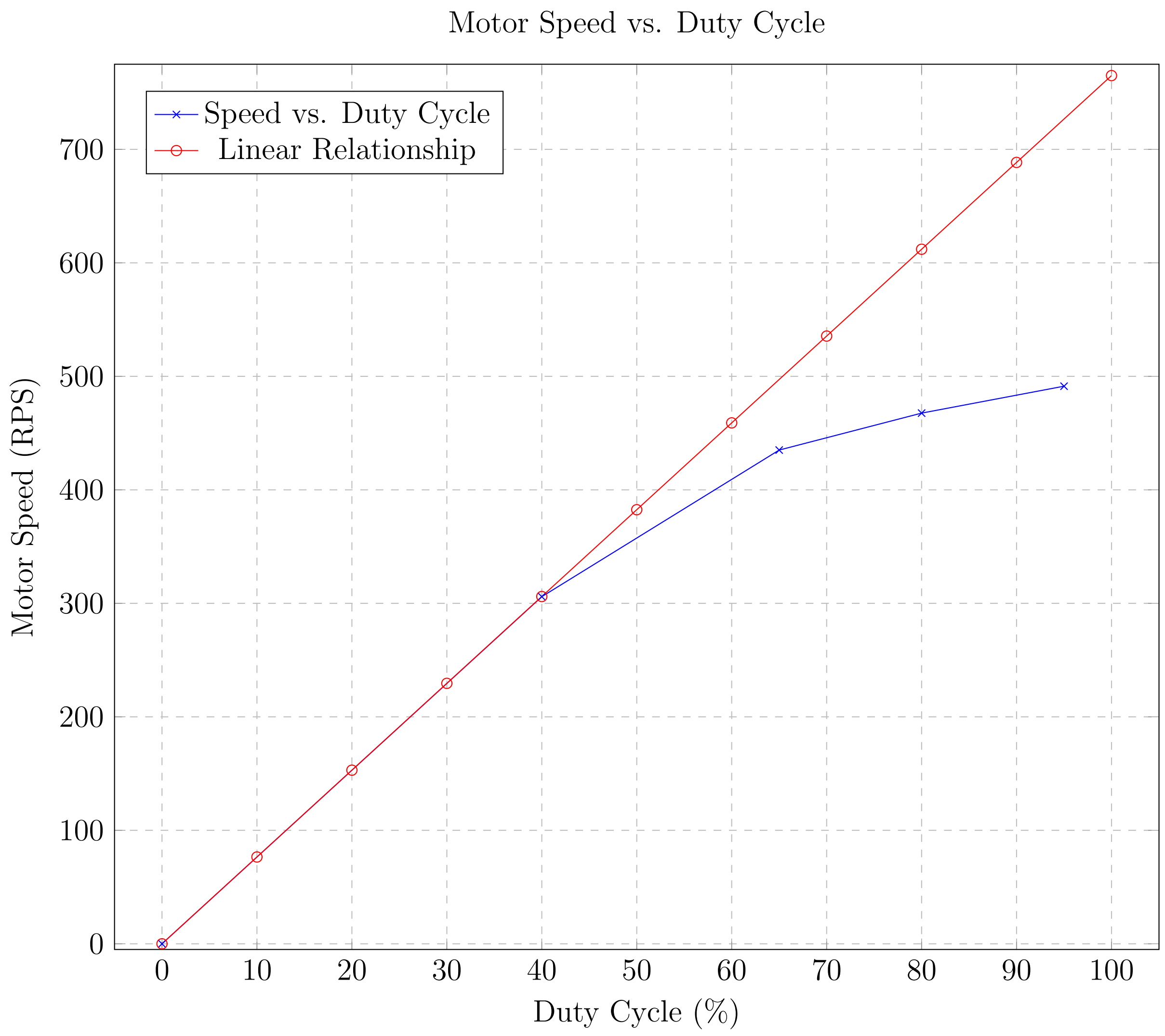


Figure 4: 95% Duty Cycle

Duty Cycle (%)	V_M **(V)**	f_{SB} **(Hz)**	**Speed (RPS)**
0	7.915	0	0
40	7.895	307	305.85
65	7.890	417	435.05
80	7.884	483	467.6
95	7.882	485	491.32

The captures of all of this data are shown in Figures 5, 6, 7, 8, and 9 at the end of the report.

The supply voltage clearly drops as the duty cycle increases, and this is a result of the increased *load* on the power supply. Due to the increasing current draw from the motor, the power supply is unable to maintain the desired potential difference across the terminals.



As the above plot shows, the relationship between motor speed and duty cycle is not exactly linear (although very close). This might be in part due to the limited amounts of data collected, or (more likely) the fact that the motor has internal resistances and friction that prevent it from reaching the higher speeds we'd expect at higher duty cycles (if it were linear).

4 Conclusion

The purpose of the lab, to use the input capture peripheral to measure a motor's speed, was certainly accomplished. Although the specific mathematical calculation that converts the different timer measurements to an RPS value is slightly non intuitive (in my opinion), the results are in-line with the physical expectations of the motor.

This type of system shows how easy it would be to implement feedback for controlling the motor speed. Pseudo-code is attached below that demonstrates an example of this system:

Listing 5: Feedback Pseudo-Code

```
func input_capture() {
  delta_time = time_new - time_old
  curr_rps = delta_time * conversion_factor

  if (curr_rps > desired_rps)
    new_pwm = curr_pwm > 1 ? curr_pwm - 1 : 0
  else if (curr_rps < desired_rps)
    new_pwm = curr_pwm < 99 ? curr_pwm + 1 : 100

  set_pwm(new_pwm)
}
```

The general idea is to take the current reading of the motor's RPS, and depending on if that is above or below a desired speed, adjust the PWM output accordingly.

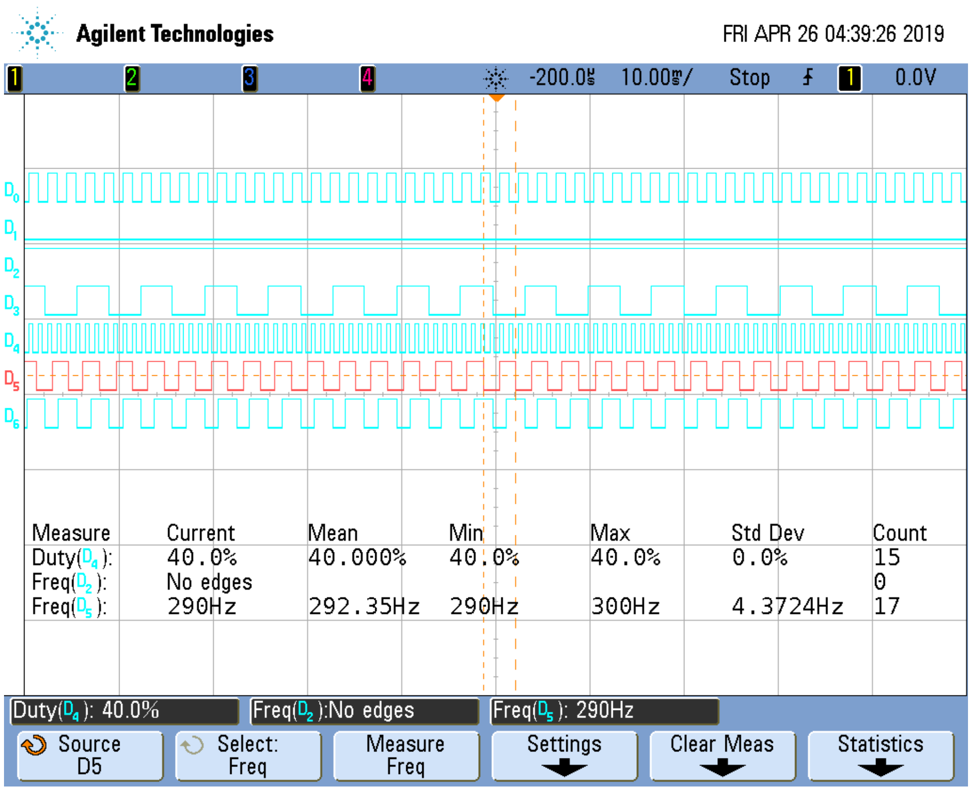
The limiting factors for measuring a signal's frequency are primarily the timer used for the input capture peripheral. We used timer 3 with a prescale value of 256 and a maximum possible period register. This means that the *fastest* possible frequency we could measure would be:

$$f_{max} = \frac{f_{PB}}{t_{PS}}$$

Which corresponds to a frequency of about 39,062 Hz for our current timer configuration. This would be a signal that only allows a single increment of the timer's period register before triggering an interrupt. On the other hand, the slowest possible signal that could be detected by this system is when the period register reaches **0xFFFF**, or:

$$f_{min} = \frac{f_{PB}}{t_{PS}} \cdot \frac{1}{65,535}$$

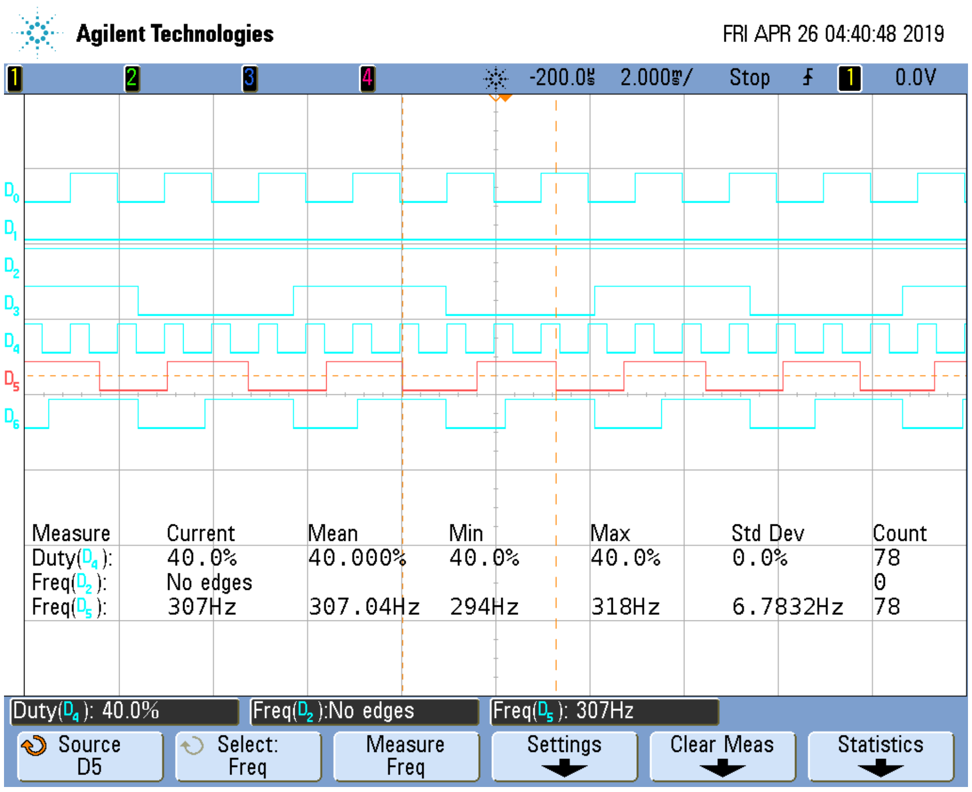
This is approximately 0.596 Hz for the current setup. This is obviously drastically affected by the prescale value used for the associated timer. We are using a very large prescale value which greatly reduces the maximum frequency detectable, but gives a very large range of detectable frequencies.



Measure	Current	Mean	Min	Max	Std Dev	Count
Duty(D_4):	40.0%	40.000%	40.0%	40.0%	0.0%	15
Freq(D_2):	No edges					0
Freq(D_5):	290Hz	292.35Hz	290Hz	300Hz	4.3724Hz	17

Duty(D_4): 40.0% Freq(D_2):No edges Freq(D_5): 290Hz

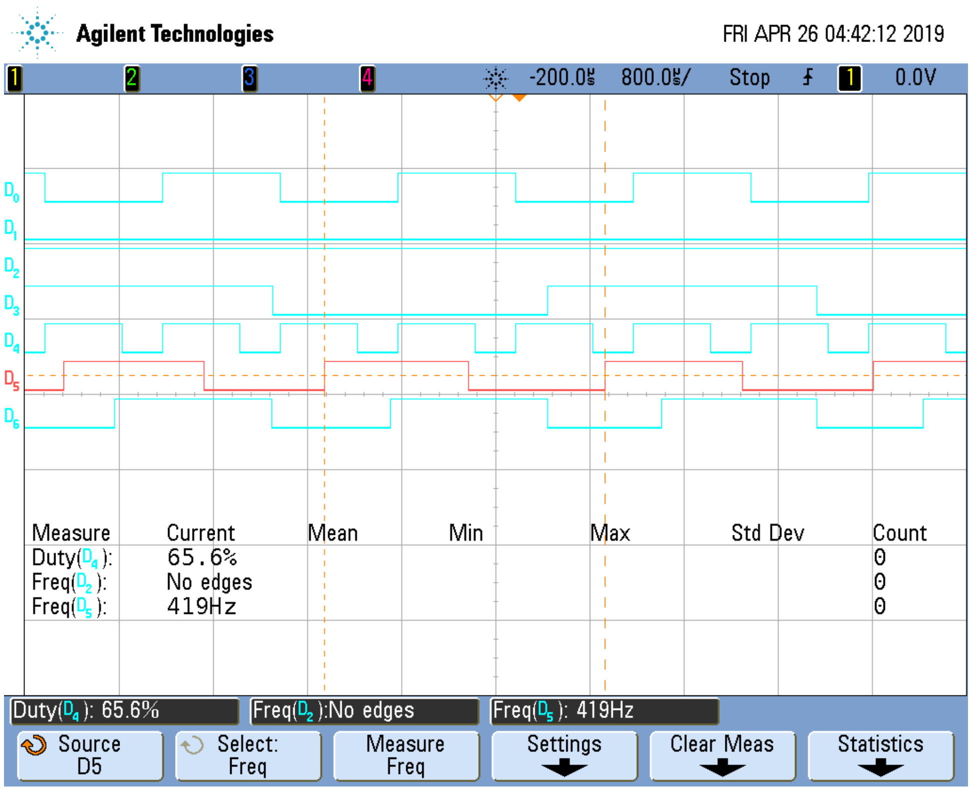
Figure 5: 0% Duty Cycle Data



Measure	Current	Mean	Min	Max	Std Dev	Count
Duty(D_4):	40.0%	40.000%	40.0%	40.0%	0.0%	78
Freq(D_2):	No edges					0
Freq(D_5):	307Hz	307.04Hz	294Hz	318Hz	6.7832Hz	78

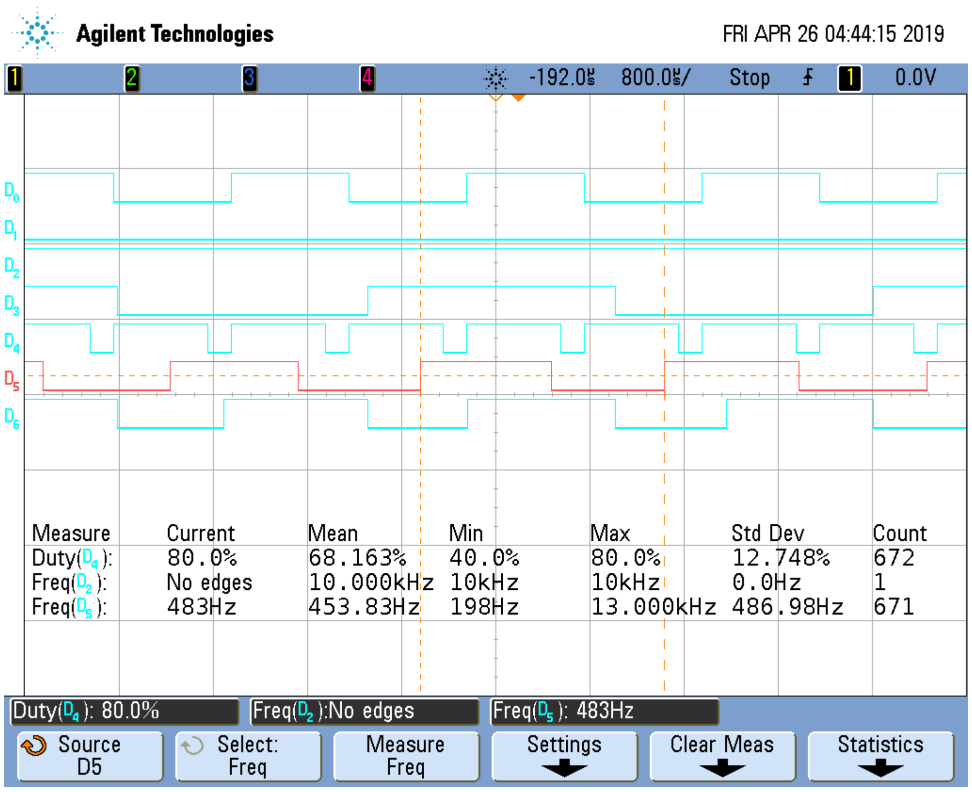
Duty(D_4): 40.0% Freq(D_2):No edges Freq(D_5): 307Hz

Figure 6: 40% Duty Cycle Data


Agilent Technologies FRI APR 26 04:42:12 2019

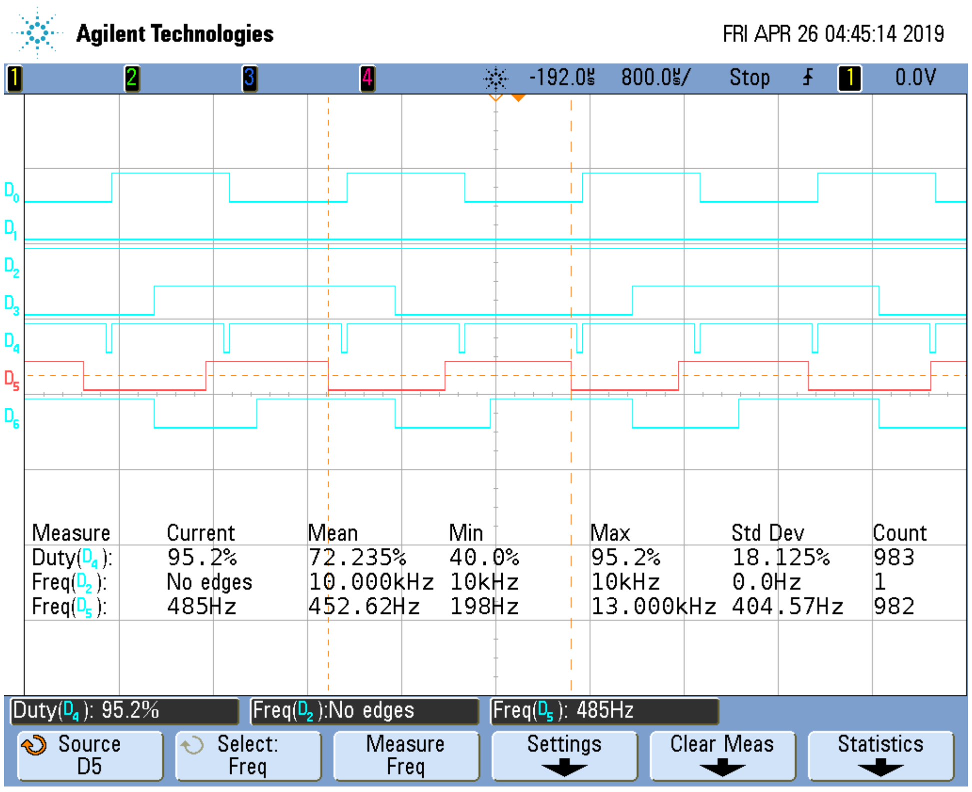
```
1  2  3  4   -200.0µs  800.0µs/  Stop  1  0.0V

Measure      Current     Mean       Min      Max        Std Dev    Count
Duty(D4):    65.6%                                                 0
Freq(D2):    No edges                                              0
Freq(D5):    419Hz                                                 0

Duty(D4): 65.6%    Freq(D2):No edges    Freq(D5): 419Hz
```

Figure 7: 65% Duty Cycle Data


Agilent Technologies FRI APR 26 04:44:15 2019

```
1  2  3  4   -192.0µs  800.0µs/  Stop  1  0.0V

Measure      Current     Mean        Min      Max        Std Dev    Count
Duty(D4):    80.0%       68.163%     40.0%    80.0%      12.748%    672
Freq(D2):    No edges    10.000kHz   10kHz    10kHz      0.0Hz      1
Freq(D5):    483Hz       453.83Hz    198Hz    13.000kHz  486.98Hz   671

Duty(D4): 80.0%    Freq(D2):No edges    Freq(D5): 483Hz
```

Figure 8: 80% Duty Cycle Data



Measure	Current	Mean	Min	Max	Std Dev	Count
Duty(D4):	95.2%	72.235%	40.0%	95.2%	18.125%	983
Freq(D2):	No edges	10.000kHz	10kHz	10kHz	0.0Hz	1
Freq(D5):	485Hz	452.62Hz	198Hz	13.000kHz	404.57Hz	982

Duty(D4): 95.2% Freq(D2):No edges Freq(D5): 485Hz

Figure 9: 95% Duty Cycle Data