CPE301 – SPRING 2020

MIDTERM 2

Student Name: Do Le

Student #: 2001183621

Student Email: [led2@unlv.nevada.edu](mailto:led2@unlv.nevada.edu)

Primary Github address: <https://github.com/DoVietLe>

Directory: <https://github.com/DoVietLe/assignments>

Q: Write, simulate, and demonstrate using Atmel Studio 7 a C code for the AVR ATMEGA328p microcontroller that performs the following functions:

You’ll use the ADC, and PWM/CCP Module of the ATmega328/p to set and determine the speed of the DC Motor.

1. Use the motor driver, program the ATMega to drive the geared DC motor in CW and CCW direction for a given PWM.

2. Using the Potentiometer connected to ADC0, translate the ADC value to PWM value/ speed of the motor. Verify the operation.

3. Using the CCP capture pin of PWM1, in mode 1x and 2x determine the speed of the DC Motor for a set ADC Pot value/position.

4. Using CCP capture and interrupt (mode 4x), determine the speed of the DC Motor for a set ADC Pot value/position.

1. **COMPONENTS LIST AND CONNECTION BLOCK DIAGRAM w/ PINS**

Components

• Xplained Mini Board

* + ATmega328PB
  + Programmer
  + Debugger

• Atmel Studios 7.0

* + Compiler
  + Serial Monitor

• TB6612FNG Motor Driver

• DC Motor

• Potentiometer

• Pushbutton

Block Diagram

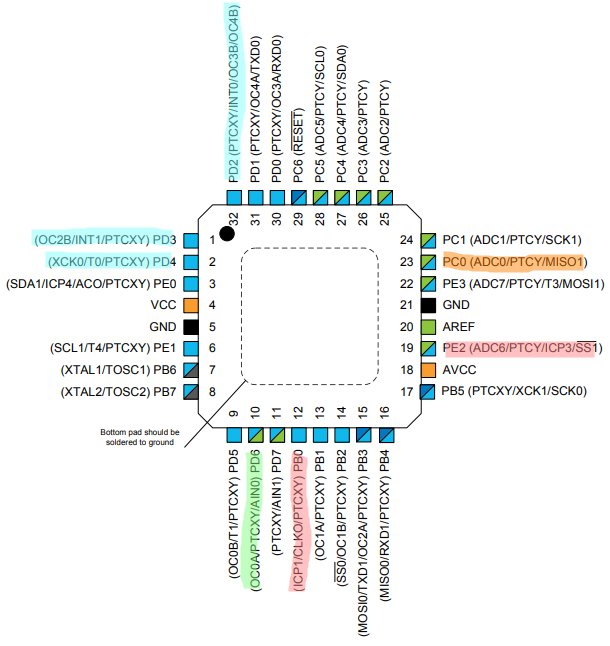


Figure : Assignment pinout

PD3 & PD4 used to control motor direction.

PD2 used for INT0 connected to button.

PC0 used for ADC connected to potentiometer.

PD6 used for PWM.

PB0 & PE2 used for capture.

1. **INITIAL/MODIFIED/DEVELOPED CODE OF TASK 1/A**

Code (1x Mode)

#include <avr/interrupt.h>

#include <avr/io.h>

#define UBBR\_VALUE 103

#define MAX\_PWM 256

volatile *uint16\_t* overflows = 0;

volatile *uint16\_t* capturedOverflows = 0;

volatile *uint16\_t* capturedVal = 0;

volatile *uint32\_t* periodTimer[20]; // Stores the period over 20 periods.

volatile *uint8\_t* arrayPos = 0;

volatile *uint64\_t* averagePeriod = 0;

volatile *uint64\_t* rpm = 0;

volatile *uint8\_t* cw = 0;

volatile char myIntString[20];

void enableDirection() {

DDRD |= (1<<PIND4)|(1<<PIND5);

setCounterClockwise();

}

void setCounterClockwise() {

cw = 0;

PORTD |= (1<<PIND4);

PORTD &= ~(1<<PIND5);

}

void setClockwise() {

cw = 1;

PORTD |= (1<<PIND5);

PORTD &= ~(1<<PIND4);

}

void setPWM() {

// Sets up port.

DDRD |= (1<<PIND6); // Uses PIND6 as output for OC0A.

// Adjusts PWM settings.

TCCR0A = 0b10000011; // Use Fast PWM on OC0A in non-inverting mode.

TCCR0B = 0b00000010; // Use Fast PWM with 8 prescaler.

}

void enableADC() {

// Sets PINC0 as input.

DDRC &= ~(1<<PINC0);

// Sets up ADC settings.

ADMUX = 0x40; // Reads from ADC0. Right justified. Internal 1.1V reference.

ADCSRB = 0x00; // Free running mode.

ADCSRA = 0b10000111; // Enables ADC with 128 pre-scaler.

}

void disableADC() {

ADCSRA &= 0x7F; // Disables the ADEN bit.

}

float analogRead() {

ADCSRA |= (1<<6); // Enables ADSC to start AD conversion.

while (!(ADCSRA&(1<<4))); // Waits until ADIF is set indicating ADC is done.

ADCSRA |= (1<<4); // Clears ADIF flag.

return ADC; // Returns value.

}

void setDutyAndCalculateSpeed(float duty) {

// Calculates the cycles for the duty cycle.

*uint16\_t* cycles = (duty/100)\*MAX\_PWM;

// Assigns the cycles if it's not the current cycle.

if (OCR0A != cycles) {

OCR0A = cycles;

// Resets everything to calculate 20 period cycle.

arrayPos = 0; // Resets array pos to restart calculation.

averagePeriod = 0; // Resets period.

while (arrayPos != 19 && OCR0A >= 35); // Waits until 20 periods are recorded.

// Calculates the average period.

for (int i = 0; i < 20; i++)

averagePeriod += periodTimer[i]/20.0;

// Calculates RPM.

rpm = (16000000\*60)/(8\*averagePeriod\*120);

sendSpeed(rpm);

}

}

// Enables the UART and it's transmitter.

void enableTransmit() {

// Sets port.

DDRD |= 0x02;

// Sets baud rate.

UBRR0 = UBBR\_VALUE;

// Sets UART settings.

UCSR0C = 0b00000110; // to async mode with no parity

// 2 stop bits, and a frame of 8 bits.

UCSR0B = 0b00001000; // Enables transmit line.

}

// Disables the transmitter on UART.

void disableTransmit() {

UCSR0B &= ~(1<<TXEN0);

}

// Sends one byte of data out the TX line.

void sendByte(*uint8\_t* b) {

if (UCSR0B & (1<<TXEN0)) { // Checks whether transmit is enabled.

while (!(UCSR0A & (1<<UDRE0))); // Waits until the UDR0 register is ready.

UDR0 = b; // Sends the byte.

}

}

void sendString(char string[]) {

int i = 0;

while (string[i] != '\0')

sendByte(string[i++]);

}

void sendSpeed(*uint64\_t* data) {

// Sends period through USART.

*snprintf*(myIntString, 20, "RPM: %u", data);

sendString(myIntString);

sendByte('\n');

sendByte('\r');

}

void setPCI() {

// Sets port direction.

DDRD &= ~(1<<PIND2); // Sets INT0 pin as input.

PORTD |= (1<<PIND2); // Enables pull-up resistor.

// Assigns settings.

EIMSK |= (1<<INT0); // Enables INT0 interrupt.

EICRA |= (1<<ISC01); // Trigger interrupt on falling edge.

}

void enableCapture() {

// Sets port directions.

DDRB &= ~(1<<PINB0); // Sets PINB0 as input for input capture.

// Sets up timer.

TCCR1A = 0b00000000; // Sets timer to normal mode.

TCCR1B = 0b01000001; // Enables input capture on rising edge. Prescaler as 8.

TIMSK1 |= (1<<ICIE1)|(1<<TOIE1);// Enables capture interrupt and overflow interrupt.

}

int main(void)

{

// Variables.

float dutyCycle = 0; // Stores the duty cycles.

// Enables settings.

setPWM();

enableTransmit();

enableADC();

enableDirection();

enableCapture();

setPCI();

sei();

// Initializes direction.

setCounterClockwise();

while (1)

{

// Gets duty cycle based on potentiometer.

dutyCycle = 100\*(analogRead()/1023); // Calculates ratio based on potentiometer value.

setDutyAndCalculateSpeed(dutyCycle); // Set the new duty cycle to the PWM.

}

}

ISR(TIMER1\_CAPT\_vect) {

// Sets capture variable and resets for next cycle.

capturedVal = ICR1; // Saves the spare value when a capture occurs.

TCNT1 = 0; // Resets timer.

capturedOverflows = overflows; // Stores the amount of overflows.

overflows = 0; // Resets the overflow counter.

// Calculates period and stores in array.

periodTimer[arrayPos] = ((*uint32\_t*)capturedVal + (*uint32\_t*)capturedOverflows\*0x10000L);

arrayPos = (arrayPos + 1) % 20; // Increments array position.

TIFR1 |= (1<<ICF1); // Clears interrupt flag.

}

ISR(TIMER1\_OVF\_vect) {

overflows++; // Increments the overflows.

TIFR1 |= (1<<TOV1); // Clears interrupt flag.

}

ISR(INT0\_vect) {

// Reverses the direction based on the current direction.

if (cw) {

setCounterClockwise();

} else {

setClockwise();

}

EIFR |= (1<<INTF0); // Clears interrupt flag.

}

1. **DEVELOPED MODIFIED CODE OF TASK 2/A from TASK 1/A**

Code (2x Mode)—Modified Parts Shown

volatile *uint16\_t* overflows0 = 0;

volatile *uint16\_t* overflows1 = 0;

volatile *uint16\_t* capturedOverflows0 = 0;

volatile *uint16\_t* capturedVal0 = 0;

volatile *uint16\_t* capturedOverflows1 = 0;

volatile *uint16\_t* capturedVal1 = 0;

volatile *uint32\_t* periodTimerRising = 0; // Stores the period in timer counts between

volatile *uint32\_t* periodTimerFalling = 0; // Stores the period in timer counts between

volatile *uint32\_t* periods[40]; // Stores period over 20 periods.

volatile *uint8\_t* arrayPos = 0;

void setDutyAndCalculateSpeed(float duty) {

// Calculates the cycles for the duty cycle.

*uint16\_t* cycles = (duty/100)\*MAX\_PWM;

// Assigns the cycles if it's not the current cycle.

if (OCR0A != cycles) {

OCR0A = cycles;

// Resets everything to calculate 20 period cycle.

arrayPos = 0; // Resets array pos to restart calculation.

averagePeriod = 0; // Resets period.

while (arrayPos != 39 && OCR0A >= 35); // Waits until 20 periods are recorded.

// Calculates the average period.

for (int i = 0; i < 40; i++)

averagePeriod += periods[i]/40.0;

// Calculates RPM.

rpm = (16000000\*60)/(8\*averagePeriod\*120);

sendSpeed(rpm);

}

}

ISR(TIMER1\_CAPT\_vect) {

if (TCCR1B & (1<<ICES1)) {

// Captures values and sets data.

capturedVal0 = ICR1; // Saves the spare value when a capture occurs.

TCNT1 = 0; // Resets timer.

capturedOverflows0 = overflows0; // Stores the amount of overflows.

overflows0 = 0; // Resets the overflow counter.

TCCR1B &= ~(1<<ICES1); // Switch to capture falling edge.

// Stores period.

periods[arrayPos] = (*uint32\_t*)capturedVal0 + (*uint32\_t*)capturedVal1 + 0x10000L\*capturedOverflows0;

} else {

// Captures values and sets data.

capturedVal1 = ICR1; // Saves the spare value when a capture occurs.

TCNT1 = 0;

capturedOverflows1 = overflows1; // Stores the amount of overflows.

overflows1 = 0; // Resets the overflow counter.

TCCR1B |= (1<<ICES1); // Switch to capture rising edge.

// Stores period.

periods[arrayPos] = (*uint32\_t*)capturedVal0 + (*uint32\_t*)capturedVal1 + 0x10000L\*capturedOverflows1;

}

// Increments array pos.

arrayPos = (arrayPos + 1) % 40;

TIFR1 |= (1<<ICF1); // Clears interrupt flag.

}

ISR(TIMER1\_OVF\_vect) {

overflows0++; // Increments the overflows.

overflows1++;

TIFR1 |= (1<<TOV1); // Clears interrupt flag.

}

Code (4x Mode)—Modified Parts Shown

volatile *uint16\_t* overflows[4];

volatile *uint16\_t* capturedOverflows[4];

volatile *uint16\_t* capturedVal[4];

volatile *uint32\_t* periodTimerRising[2]; // Stores the period in timer counts between

volatile *uint32\_t* periodTimerFalling[2]; // Stores the period in timer counts between

volatile *uint32\_t* periods[80]; // Stores period for 20 periods.

volatile *uint8\_t* arrayPos = 0;

volatile *uint32\_t* averagePeriod = 0; // Calculates average period.

volatile *uint64\_t* rpm = 0;

volatile *uint32\_t* prevPeriod = 1;

volatile *uint8\_t* cw = 0;

volatile char myIntString[20];

void setDutyAndCalculateSpeed(float duty) {

// Calculates the cycles for the duty cycle.

*uint16\_t* cycles = (duty/100)\*MAX\_PWM;

// Assigns the cycles if it's not the current cycle.

if (cycles != OCR0A) {

OCR0A = cycles;

// Resets everything to calculate 20 period cycle.

arrayPos = 0; // Resets array pos to restart calculation.

averagePeriod = 0; // Resets period.

while (arrayPos != 39 && OCR0A >= 35); // Waits until 20 periods are recorded.

// Calculates the average period.

for (int i = 0; i < 40; i++)

averagePeriod += periods[i]/40.0;

// Calculates RPM.

rpm = (16000000\*60)/(8\*averagePeriod\*120);

sendSpeed(rpm);

}

}

void enableCapture() {

// Sets port directions.

DDRB &= ~(1<<PINB0); // Sets PINB0 (ICP1) as input.

DDRE &= ~(1<<PINE2); // Sets PINE2 (ICP3) as input.

// Sets up timer 1.

TCCR1A = 0b00000000; // Sets timer to normal mode.

TCCR1B = 0b01000001; // Enables input capture on rising edge. Prescaler as 8.

TIMSK1 |= (1<<ICIE1)|(1<<TOIE1);// Enables capture interrupt and overflow interrupt.

// Sets up timer 3.

TCCR3A = 0b00000000; // Sets timer to normal mode.

TCCR3B = 0b01000001; // Enables input capture on rising edge. Prescaler as 8.

TIMSK3 |= (1<<ICIE3)|(1<<TOIE3);// Enables capture interrupt and overflow interrupt.

}

ISR(TIMER1\_CAPT\_vect) {

if (TCCR1B & (1<<ICES1)) {

capturedVal[0] = ICR1; // Saves the spare value when a capture occurs.

TCNT1 = 0; // Resets timer.

capturedOverflows[0] = overflows[0];// Stores the amount of overflows.

overflows[0] = 0; // Resets the overflow counter.

TCCR1B &= ~(1<<ICES1); // Switch to capture falling edge.

// Stores period.

periods[arrayPos] = (*uint32\_t*)capturedVal[0] + (*uint32\_t*)capturedVal[1] + 0x10000L\*capturedOverflows[0];

} else {

capturedVal[1] = ICR1; // Saves the spare value when a capture occurs.

TCNT1 = 0; // Resets timer.

capturedOverflows[1] = overflows[1];// Stores the amount of overflows.

overflows[1] = 0; // Resets the overflow counter.

TCCR1B |= (1<<ICES1); // Switch to capture rising edge.

// Stores period.

periods[arrayPos] = (*uint32\_t*)capturedVal[0] + (*uint32\_t*)capturedVal[1] + 0x10000L\*capturedOverflows[1];

}

// Increments array pos.

arrayPos = (arrayPos + 1) % 80;

TIFR1 |= (1<<ICF1); // Clears interrupt flag.

}

ISR(TIMER3\_CAPT\_vect) {

if (TCCR3B & (1<<ICES3)) {

capturedVal[2] = ICR3; // Saves the spare value when a capture occurs.

TCNT3 = 0; // Resets timer.

capturedOverflows[2] = overflows[2];// Stores the amount of overflows.

overflows[2] = 0; // Resets the overflow counter.

TCCR3B &= ~(1<<ICES3); // Switch to capture falling edge.

// Stores period.

periods[arrayPos] = (*uint32\_t*)capturedVal[2] + (*uint32\_t*)capturedVal[3] + 0x10000L\*capturedOverflows[2];

} else {

capturedVal[3] = ICR3; // Saves the spare value when a capture occurs.

TCNT3 = 0; // Resets timer.

capturedOverflows[3] = overflows[3];// Stores the amount of overflows.

overflows[3] = 0; // Resets the overflow counter.

TCCR3B |= (1<<ICES3); // Switch to capture rising edge.

// Stores period.

periods[arrayPos] = (*uint32\_t*)capturedVal[2] + (*uint32\_t*)capturedVal[3] + 0x10000L\*capturedOverflows[3];

}

// Increments array pos.

arrayPos = (arrayPos + 1) % 80;

TIFR1 |= (1<<ICF1); // Clears interrupt flag.

}

ISR(TIMER1\_OVF\_vect) {

overflows[0]++; // Increments the overflows.

overflows[1]++;

TIFR1 |= (1<<TOV1); // Clears interrupt flag.

}

ISR(TIMER3\_OVF\_vect) {

overflows[2]++; // Increments the overflows.

overflows[3]++;

TIFR3 |= (1<<TOV3); // Clears interrupt flag.

}

1. **SCHEMATICS**

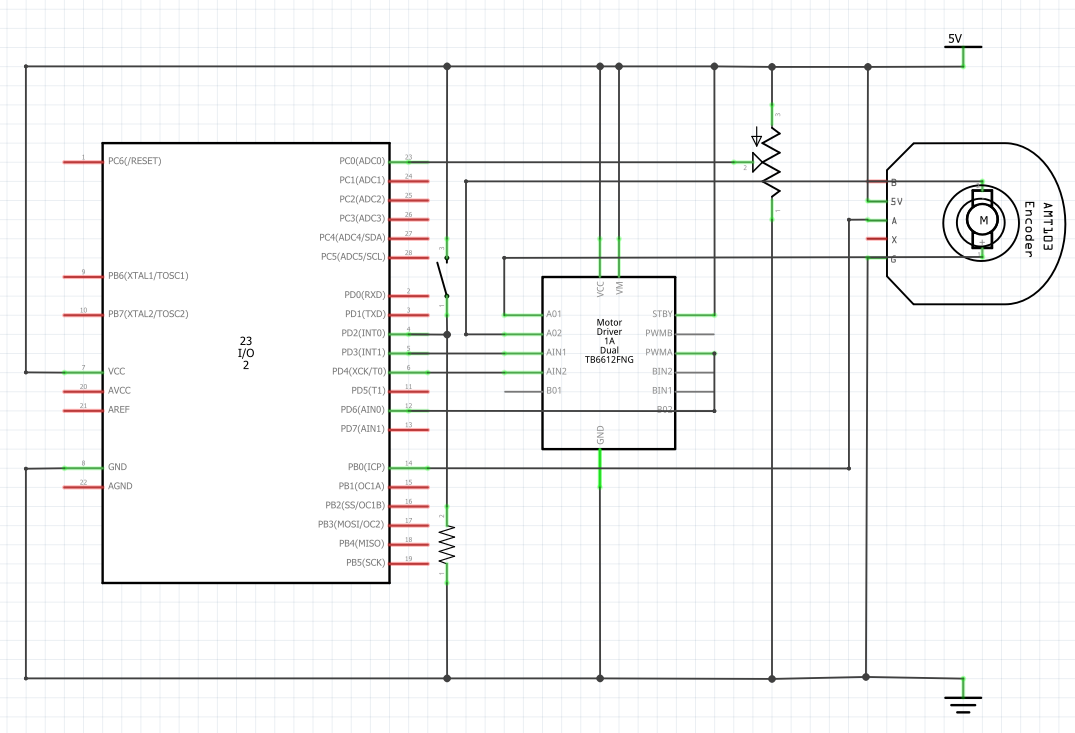


Figure : Assignment schematic

1. **SCREENSHOTS OF EACH TASK OUTPUT (ATMEL STUDIO OUTPUT)**

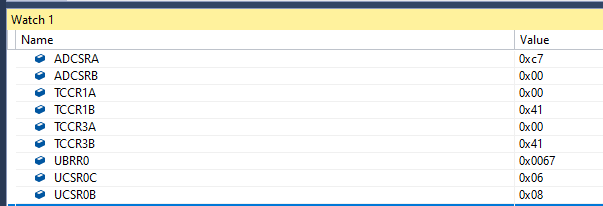


Figure : Settings on Atmel

1. **SCREENSHOT OF EACH DEMO (BOARD SETUP)**

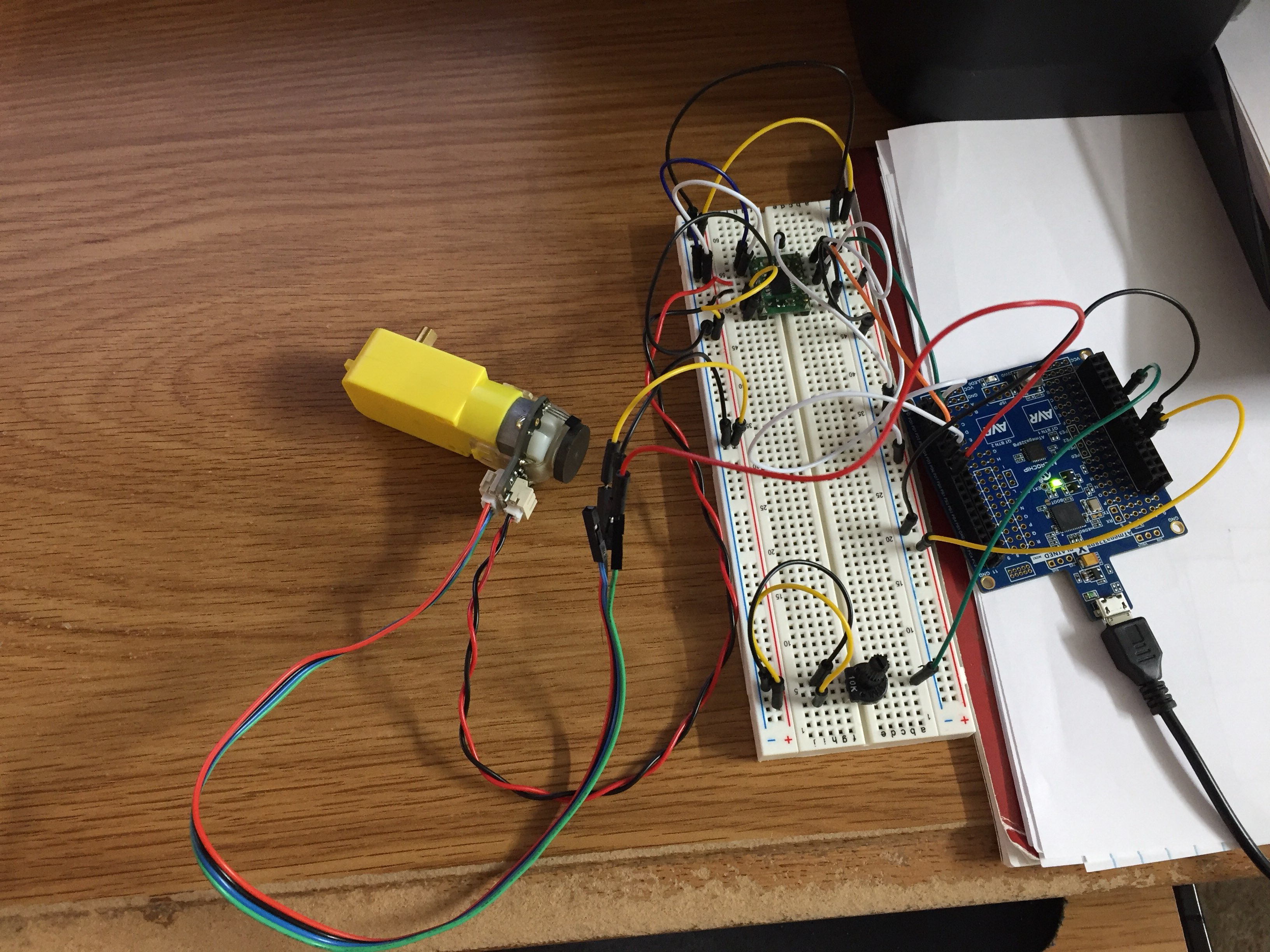


Figure : Board setup

1. **VIDEO LINKS OF EACH DEMO**

Video Link

https://www.youtube.com/watch?v=DC4lHd7k224

1. **GITHUB LINK OF THIS DA**

https://github.com/DoVietLe/assignments/tree/master/ESD301/Midterm02

**Student Academic Misconduct Policy**

<http://studentconduct.unlv.edu/misconduct/policy.html>

“This assignment submission is my own, original work”.

Do V. Le