# MX1 Assignment 2 Part C – RC Omniwheel Chassis – Lucien Morey (12904090)

## Overview

My project for the Mechatronics 1 unleashed program was to design then implement a closed loop velocity controller on a rectangular omniwheel chassis. The system would receive inputs from three remote control joysticks; velocity x, velocity y and velocity z (yaw input). These inputs would then be passed through a PID controller with wheel encoders as the sole observer to control motor output to meet specification. Finally, the resulting velocity x, velocity y and velocity z commands would be parsed to the motors to create planar movement as desired. I also chose to create a serial terminal console that would display system state variables in real time to help with debugging.

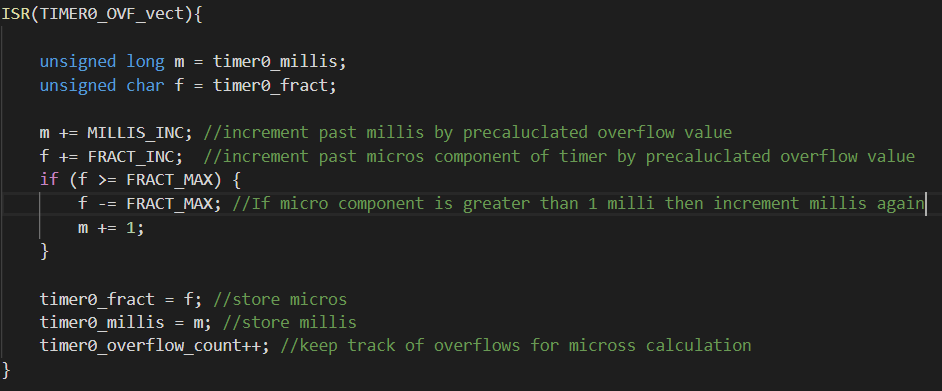
I completed the serial monitor and got as far as implementing an open loop percentage output controller on the chassis. My failure to create a velocity controller was due to difficulty in reading values from encoders. In the following sections, I will break down the system as it is and include my plan to transition to a velocity controller then reflect on lessons learned.

## A close up of a map Description automatically generatedSystem Overview

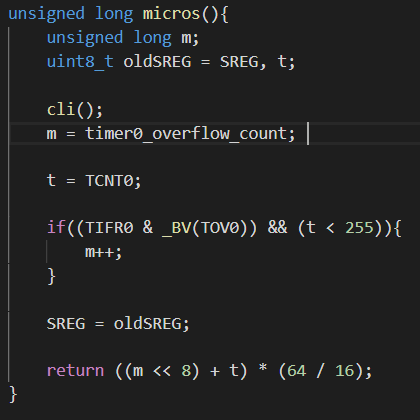
To create the system, I utilised as many of the internal resources on my MCU (Arduino Mega 2560) as possible. This was with the hope of creating a high fidelity system where no blocking would occur and get as close as possible to running calculations in parallel. Interrupt service routines were used to read all incoming PWM signals and to estimate instantaneous velocity at each wheel through photo interrupter wheel encoders. My main loop would then process the data obtained from these interrupts to calculate desired motor outputs and write to output compare registers to generate the required PWM signal for my motor controllers.

### Millis/Micros timer

My millis timer is based on the timer contained within the AVR core wiring library. Timer 0 is set up with a prescaler of 1/64; this means every tick of the clock will take 4us, and every overflow will take 1024 us. Once an interrupt occurs, an overflow counter is incremented to help keep time.

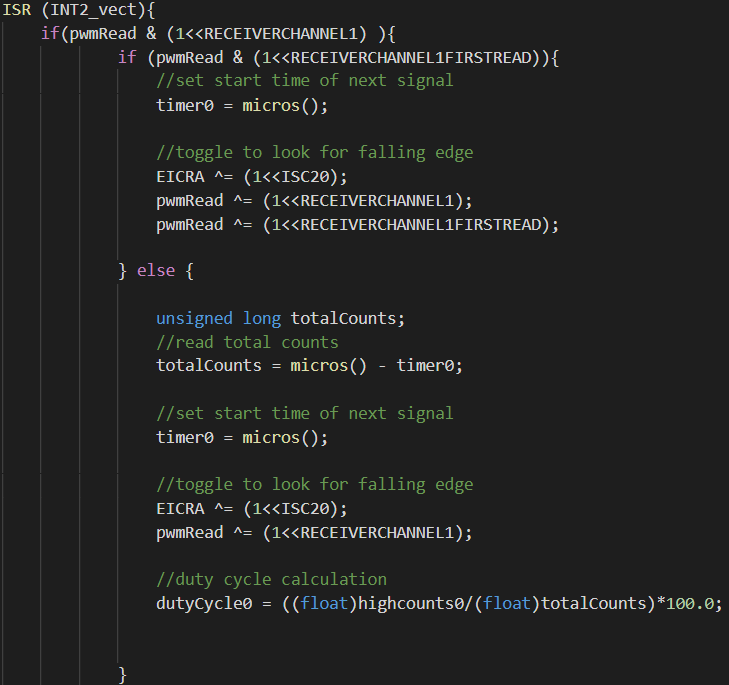


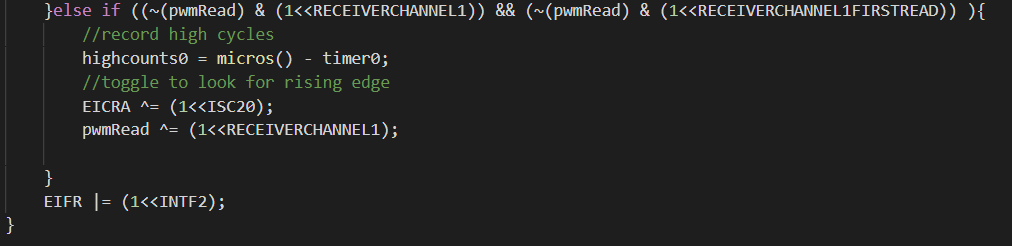
With the ratio of microseconds per overflow, I was able to create a millis and micros function that would calculate current system time based on how many interrupts had already occurred and the current value of the timer counter register. The code excerpt below reflects this calculation and also shows the global interrupt status flag being cleared to avoid errors during the process.



### PWM Input Processing

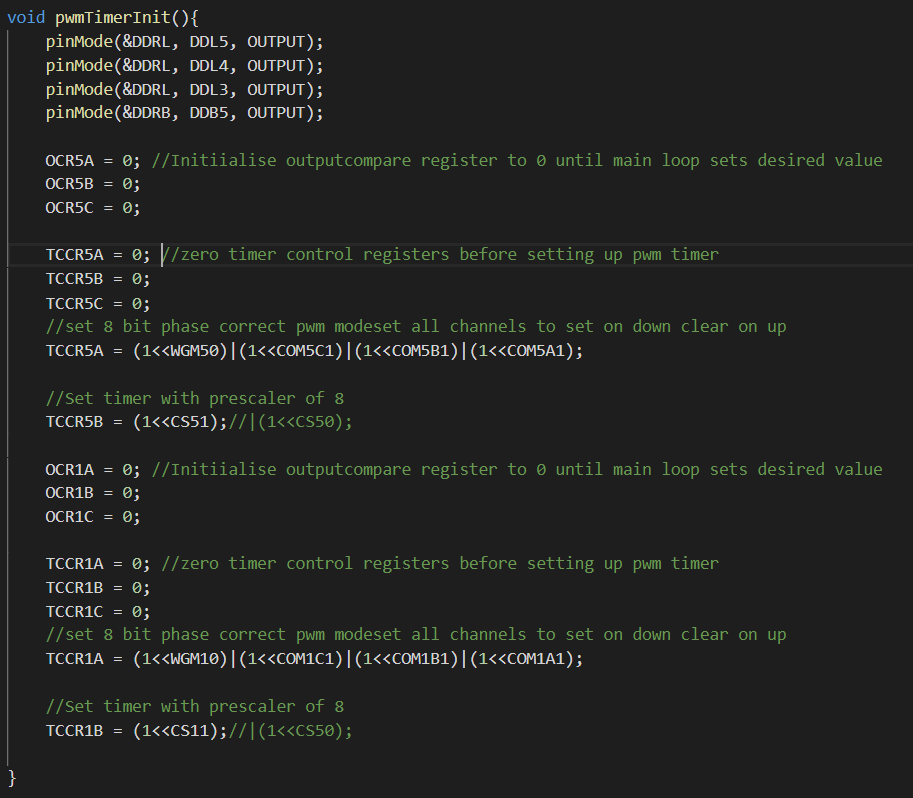
All PWM input duty calculation relies upon on the waveform period and the edges of the wave. All inputs are attached to a unique external interrupt pin to assist with edge triggering. The first rising edge detected is timestamped and the interrupt mask is set to look for a falling edge. Once a falling edge has been detected, the total time high for the signal calculated by recording the current timestamp then finding the difference between the current time and the time of the rising edge. The interrupt mask is then set to look for a rising edge again. Finally, once the next rising edge occurs, the total period can be found by taking a new timestamp and calculating the difference between the original timestamp. The start time is now updated to the current timestamp. This order of operations is important to avoid a seg fault from division by 0. The duty cycle can now be described as is the time high divided by total time.





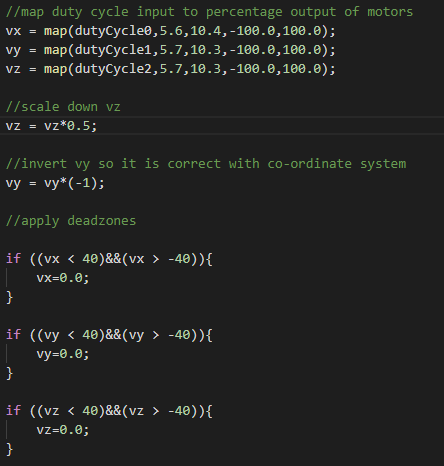
### PWM Output

PWM output to the motor controllers is generated by timers 1 and 5 in phase correct PWM mode. Each timer is in 8-bit mode, so every timer tick corresponds to a signal tick. In this mode, the output compare registers will automatically write special pins high when their value is reached on the timer. The registers are written to 0 until the main loop gives them a value to avoid dangerous behaviour upon startup.

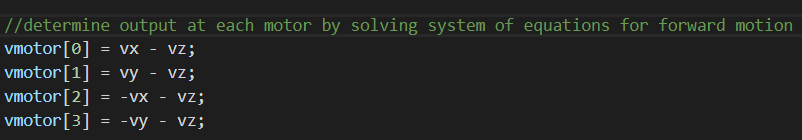


### Main Loop

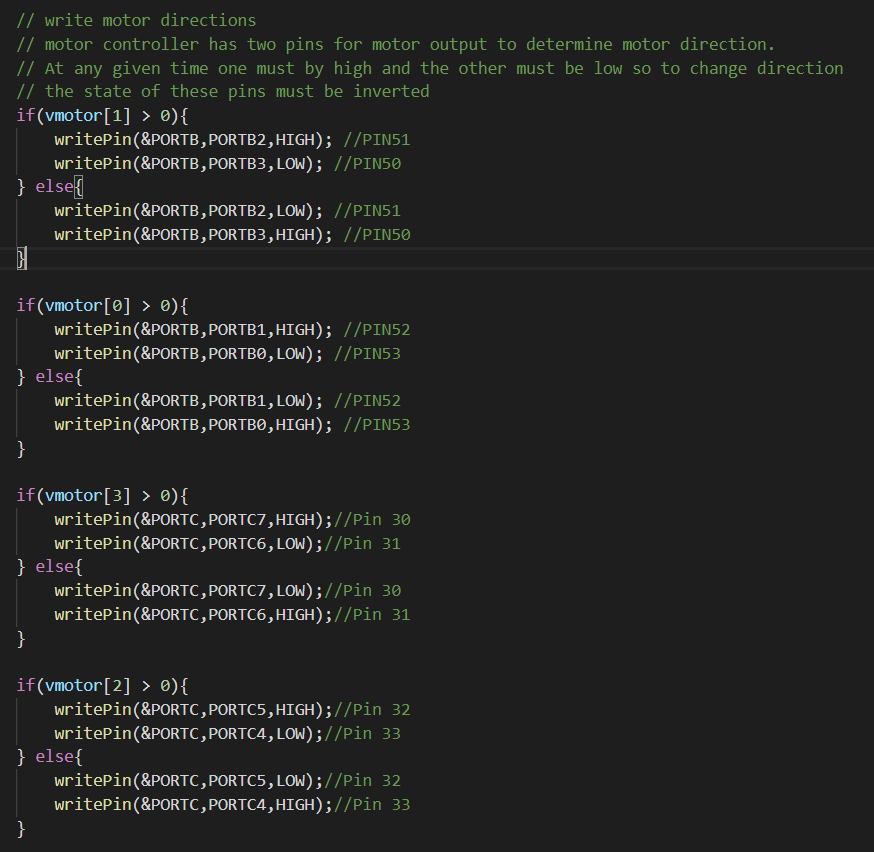
My main handles all motor velocity calculation and in future will handle all PID control as well. The current system runs in percentage output mode so that will be outlined in this section. To begin all PWM signal inputs get mapped between a percentage output of -100 to 100. Each duty cycle has a unique minimum and maximum based on measured values with an oscilloscope. After this, dead zones are applied to avoid accidental inputs from each joystick.



Each motor has direction can be calculated by summing velocity values and their directions as determined by the right-hand rule where your thumb points in the direction of the motor shaft and wrist rotation determines the required sign for each velocity value.



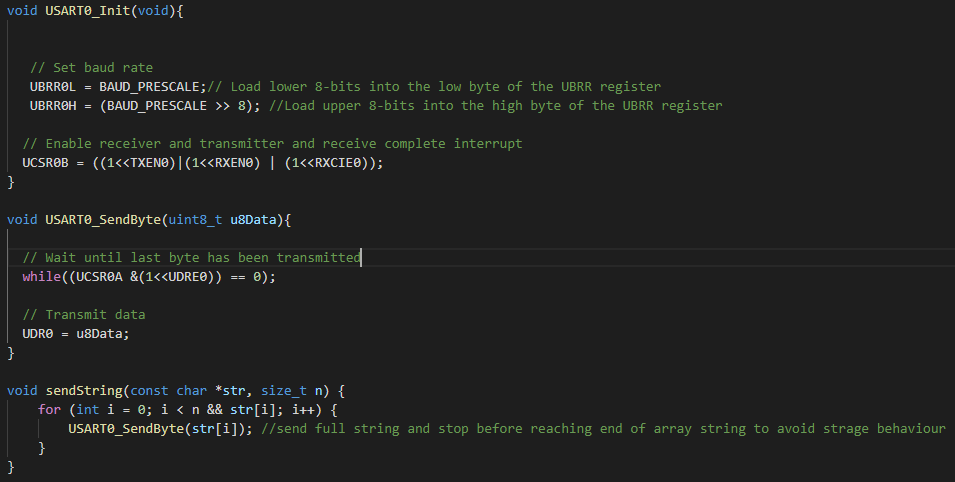
My chosen motor controllers take two inputs to determine the motor direction where at any given time, one must be high, and the other must be low for rotation to occur. The third image in this section depicts output pins written high and low to determine direction on the motor controller.



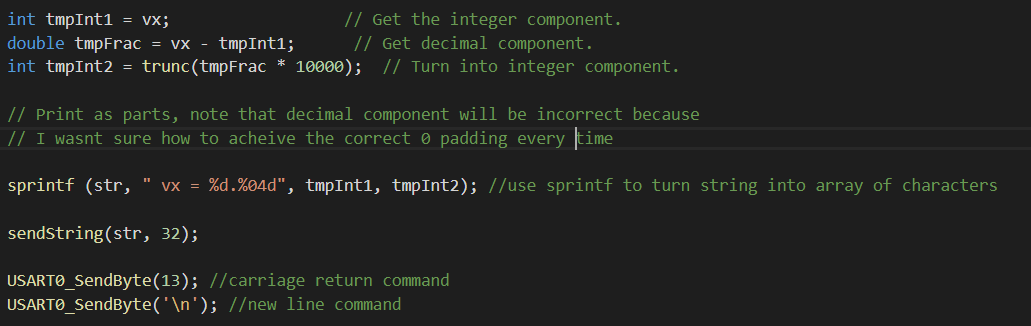
After the direction has been handled the main loop finally writes to the output compare registers and generates the desired PWM output.

### Serial Terminal Debugging

During system development, I struggled to understand what was occurring in my stem because of the required type casting during processing. To solve the problem, I incorporated a serial port sniffer into my workflow. This required me to open one of the USART ports on my MCU and set up functions to send single bytes containing ASCII characters that the port sniffer would recognise and display.



The code excerpt featured below shows the required process for printing a floating point number in the serial monitor. Floating point numbers need to be loaded into two temp integer values to avoid a seg fault when trying to string print into an array of chars. Strangely this fault would even occur when using the float formatting flag.



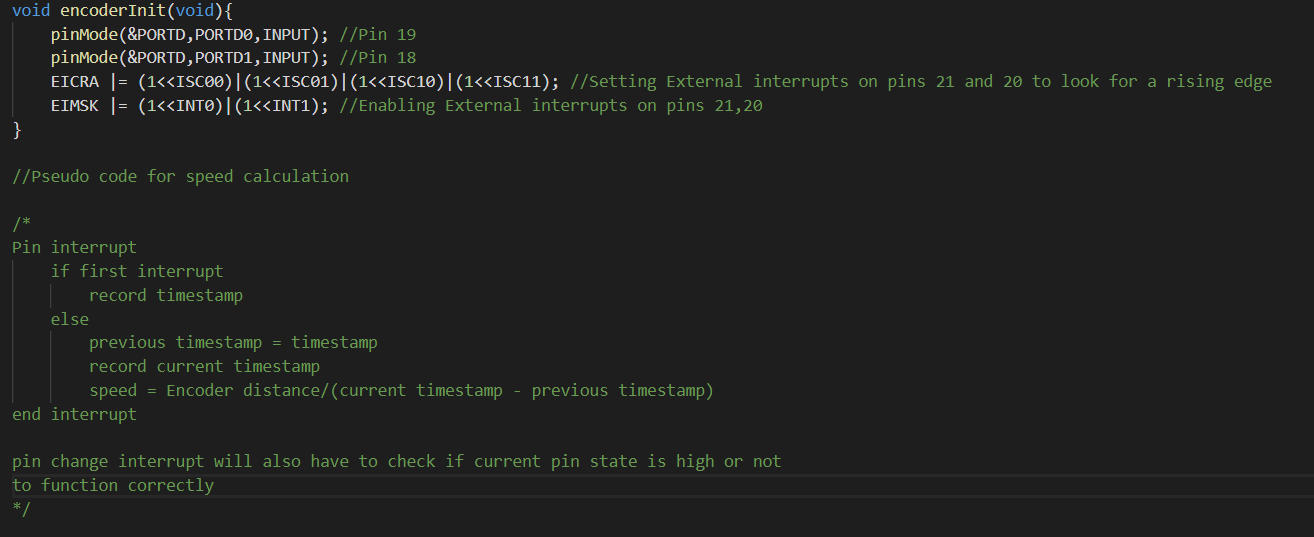
The code to send strings was based on this snippet of code that found on Stack Overflow. It seems that a fault when trying to string print a floating point number is not uncommon, and I was lucky to find this example because I still don’t understand why sending a float might cause a fault.

A screenshot of a social media post

Description automatically generated

## Transition to Velocity Control

By the time of assessment, I had not implemented velocity control because I was unable to get my encoders to read on an Arduino. I suspect this is because the voltage drop over the photo interrupter was too high and as a consequence, the Arduino wouldn’t pull up the signal to a high state. Ideally, these sensors would detect a rising edge and take a timestamp. Then the time difference between this pulse and the previous pulse would be calculated and be the divisor for a constant distance between encoder slots (approx. 2.182mm). The code featured below shows the pseudo code for implementing velocity measurement in my system. Modifications to the encoder init function would be made to account for pin change interrupts such that a change in value is also and gated with the current value. This way, calculations only take place when a rising edge occurs.



PID also needs to be implemented to complete this transition. Mapped duty cycle values will be altered to velocity values instead of percentage output values. These values become my setpoint values to be parsed into the PID controller along with the measured values to calculate system error. P,I and D values would need to be calculated by trial and error on the system. The controller would output a value to increment or decrement the current output compare register value with to hunt for the desired velocity.

## Reflection on Lessons Learned

### System Planning

A close up of a map

Description automatically generatedA close up of text on a black background

Description automatically generatedSystem planning became one of the most important steps very quickly in my design process. Making sure I understood what I was trying to achieve by creating diagrams or listing resources was important at every step. Originally all PWM inputs were going to be sampled on an individual timer, but I soon realised that I didn’t have the resources to complete the task in this way. The problem solving that followed is what encouraged this belief as I had to spend a few hours reading my datasheet t understand what kind of solution would fit best for my system. This is how I arrived at using the external interrupt pins on the board to only processes data at important timing events. Once I had this idea in my head, I remembered Jason talking about the use of a millis timer to get relative timestamps for calculation. The image to the right shows the order of operations for calculating duty cycle on a timer and is the basis for my external interrupt implementation as well. Even using a diagram designed for a slightly different system was helpful for visualising what was required.

### The Embedded C Rabbit Hole

I really enjoyed the MCU section of this course because it was my first real programming subject in my degree. I spent a lot of time thinking about how events would run in C compared to the object-oriented paradigm I have experience with. One thing that I found particularly interesting was the use of pointers. Yujun was a great teacher for this and helped me to understand this part of C programming and just how deep embedded programming rabbit hole can go.

The image on the right shows the flow breakdown of pointers given to me by Yujun. This explanation helped me to understand what was going on and how each of the operators could be used. Currently, I use pointers in agnostic pin functions similar to those in the Arduino language. Going forward, I would like to learn how to use pointers to point to functions so that I can store a planned path as I had originally intended for this assignment.