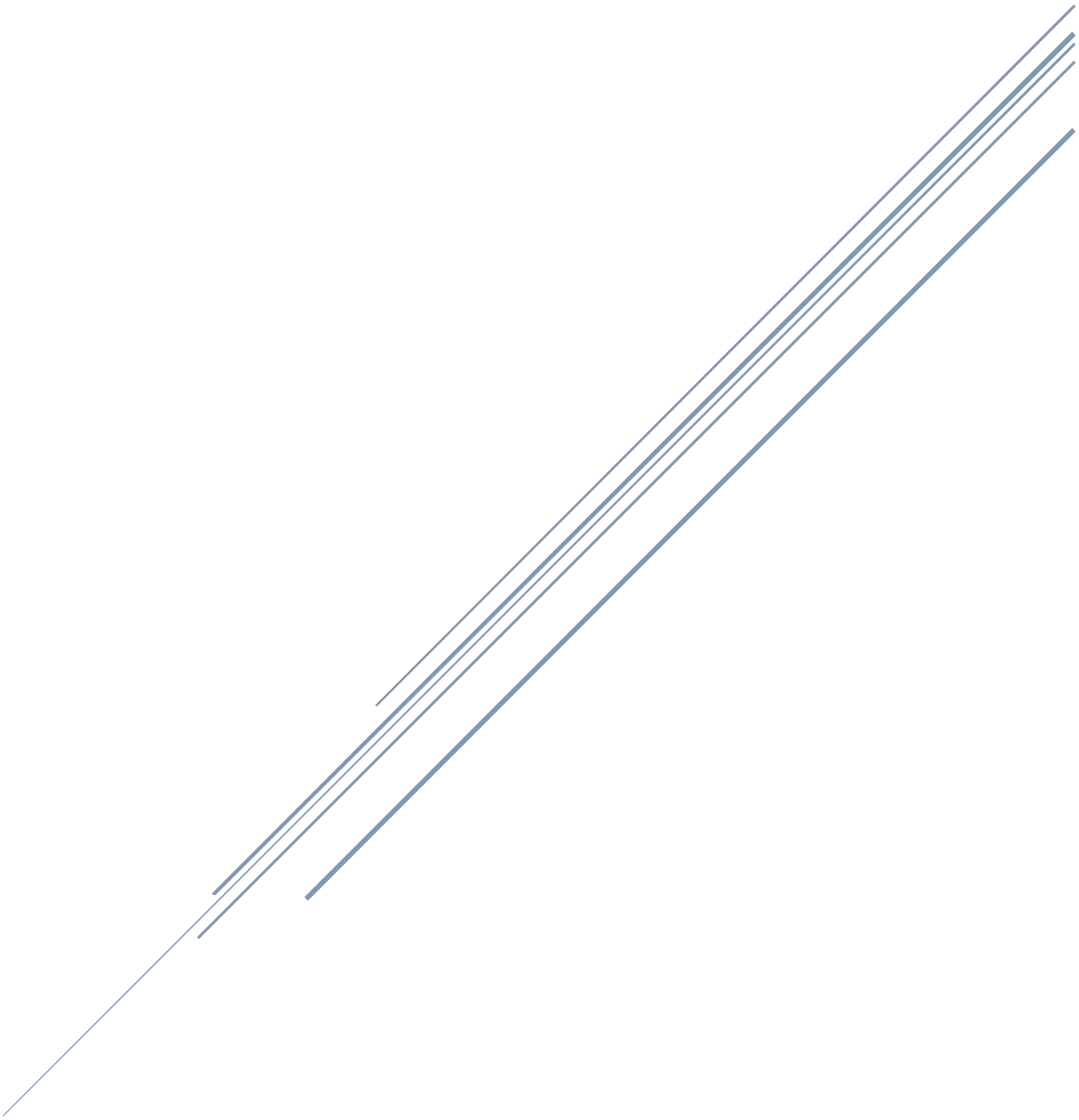
EMBEDDED SYSTEMS

Coursework 2: Brushless DC Motor Control

Patrick Foster, Eusebius Ngemera, and Zoe Williamson

Team *GreenRhino*



Imperial College London

Contents

Introduction

This project's aim is to implement precision control of a brushless DC motor using an embedded system with efficient, thread-safe firmware. The system has two compulsory functions, to spin for a defined number of rotations and stop without overshooting, and to spin at a defined angular velocity. These functions are controlled by the user serial input to the board. Optional functions to be implemented in this embedded system are, automatic tuning when the moment of inertia is changed and the motor playing a tune as it works.

Hardware

ST Nucleo Board

The board used was the NUCLEO-F303K8, together with the mbed online compiler at developer.mbed.org.

Brushless Synchronous Motor

The mechanical drawbacks of a brush motor – limited speed, lifetime and power/weight – lead to the solution of brushless motors. The brushless motor uses external components to switch the motor windings and cause the motor to rotate.

Optical disc & Motor PCB

The position of the motor is measured using photo interrupters and an optical disc. The optical disc works in two ways, absolute position feedback and relative position feedback. The absolute position of the motor is calculated using three sensors and a semi-circle slit in the optical disc. The optical disc has a reflective surface, so when the light from the photo interrupters is incident to its surface a high percentage of the light is reflected back causing a high response in the sensors. When the light from the photo interrupts passes through the gap, very little is reflected back which causes a low response in the sensor. As there are three sensors the absolute position of the motor can be calculated to intervals of 60° depending on the sensor outputs, as described in the adjacent table.

Angle(°)	Output {I3,I2,I1}
0-60	101
60-120	100
120-180	110
180-240	010
240-300	011
300-360	001

Figure 2 - Absolute position table, Lecture 5A

For finer precision on the position of the motor, the optical disc has an outer ring of small slits. Using these the position relative to previous position can be calculated. Per revolution, there are 117 slits, which converts to 468 quadrature states. Only one sensor is used for the relative position and a square wave is produced, with the period of the square wave relating to the speed and position.

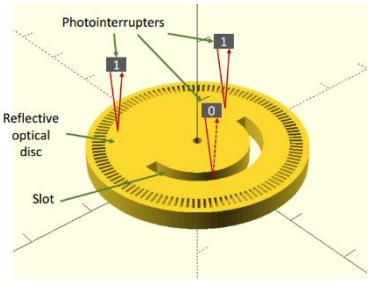


Figure 1 - Absolute position, optical disc, Lecture 5A

Control Algorithm

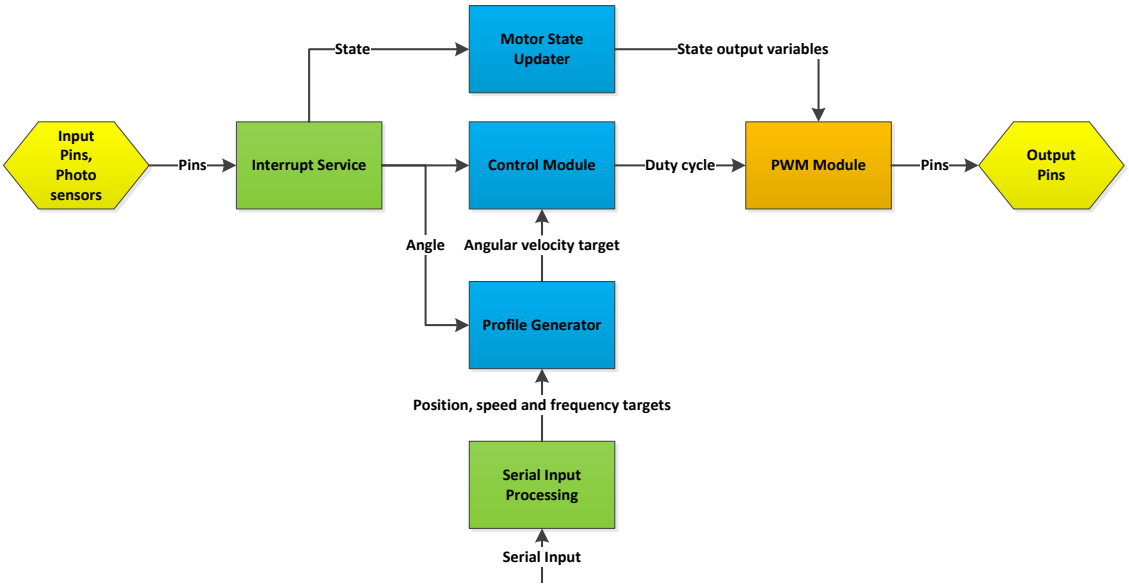


Figure 3 - Flow diagram of software program

The diagram above shows a high-level view of the different functional blocks. A serial input processing module sets the target speed and target angle. Input pins from the photo interrupters and incremental encoders are attached interrupts that update the current angle and number of turns so far. This rotation count is used when calculating the angular speed before being used by the PID control module. Given the current and target speeds, the controller outputs a duty cycle for the motor's PWM output.

The rotation count is also used in the Profile Generator (position control) to indicate when the motor needs to stay at steady state speed or start decelerating to a controlled stop by setting a new target angular speed. The duty cycle is passed to a modified `motorOut` function, whose pin outputs are now Pulse-Width Modulated (PWM).

PID Controller

The control process is based on a Proportional–Integral–Derivative (PID) controller.

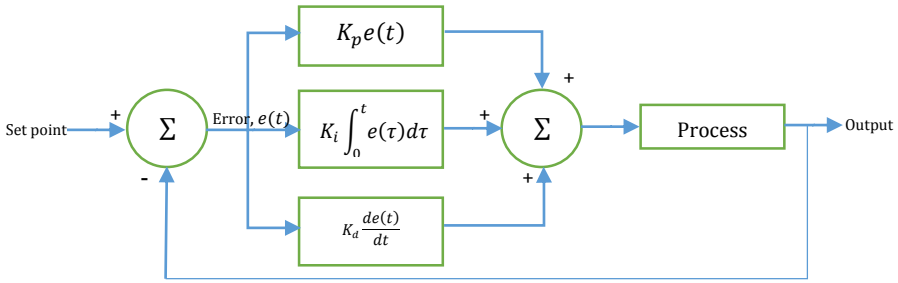


Figure 4 - Diagram of a PID loop

The 'set point' in the diagram is the target speed, which is set by the user via the serial input or set to maximum speed. The difference between the target speed and the current output is

calculated, creating the error value. Weighted values of the proportional, integrated and differentiated values of the error are summed then processed to create a PWM output duty cycle that corrects the speed of the motor towards the target speed.

In our experimental analysis, we started with a PID controller, i.e. non-zero K_p, K_i and K_d, but discovered better overall speed control with a simpler Proportional controller with zero values for K_i and K_d.

Measuring Angular Speed

Instead of measuring the duration of one rotation, our system measures the number of quadrature steps taken in a fixed amount of time. This has given us a quicker response time and thus a more robust speed controller.

A problem we had initially was fluctuations at low speeds due to the discretisation errors of small step counts. This lead us to evaluating an alternative way of measuring speed: timing how long it takes to advance by a fixed number of quadrature steps. However, this method proved unstable due to precise but inaccurate time measurements caused by shared CPU utilisation across multiple concurrent threads.

We solved the above problem by having two timing intervals for quadrature steps depending on the current speed. The additional time interval is a longer interval that would be used for speeds less than 7 rotations per second. At these low speeds, we also lowered the controller's K_p value to account for the higher raw input values of perceived speed.

Position Control

The position control thread continuously checks the angle left to reach the target and calculates a control value to pass to the PID loop accordingly, allowing the control deceleration of the rotor. The deceleration profile be incremented for the simple linear deceleration. We achieve this by comparing the speed value generated by the PID loop multiplied by scaling constant with the remaining distance to the target, setting the speed target to the distance remaining divided by the scaling constant. This results in a linear drop-in target speed from the initial value to 0, reaching zero at the same time as the rotor reaches its target position.

We had some issues with undershooting as a result of the motor stalling at a low target speeds, so we modified the deceleration profile to stop at a minimum target speed, slightly above the stalling speed of the motor to alleviate this issue. Due to the high/low speed mode of the PID loop, the scaling constant use to calculate the separation profile had to be modified to change dynamically with the mode, resulting in the constants we used in the final version of the code.

Task Analysis

Our software program has up to four threads running concurrently at any given time: PID controller, serial parser, position control and main loop writing out to the next motor state. At the beginning, we start with just the main function and serial parser. At this point, both the motor duty cycle and lead are zero therefore there is no movement by the motor. After the

command arguments are parsed, target speed and target rotations are set, and both the PID controller and the position control threads are started.

Our program makes use of global variables to pass parameters and measurements between threads. No two threads write to the same variable at the same time, assuming correct behaviour from the user, i.e. a new command is not given before the fixed number of rotations are completed. The figure below illustrates global variables, outlined in orange, which threads use.

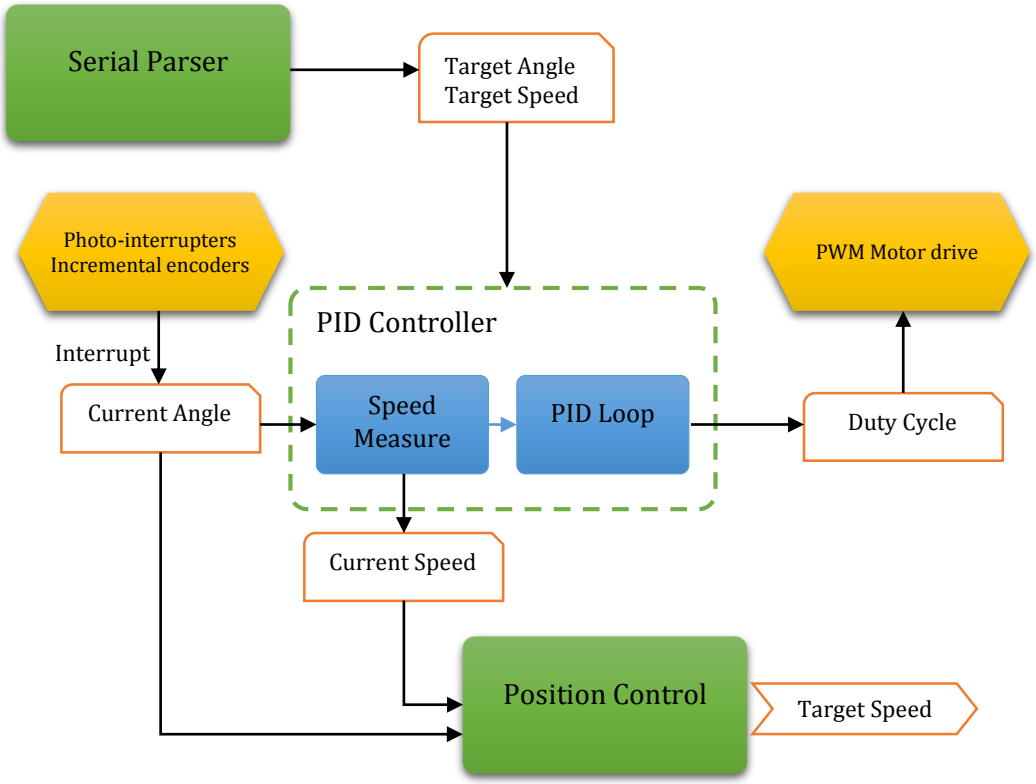


Figure 5: Thread overview: sharing of global variables

The PID controller has higher priority over the other threads due to making speed measurements with a period in the order of tens of milliseconds. This is in contrast to position control thread, which is only needed during deceleration and needs to wait for a change in duty cycle to take effect on the motor. Similarly, the serial parser and motor state change have longer deadlines. The table below shows the execution times for a single iteration of the different threads.

Loop	**Execution Time**
Motor state updater	1.25 µs
PID Controller	5.02 ms
Serial Parser	2.13 µs
Position Control	3.00 µs

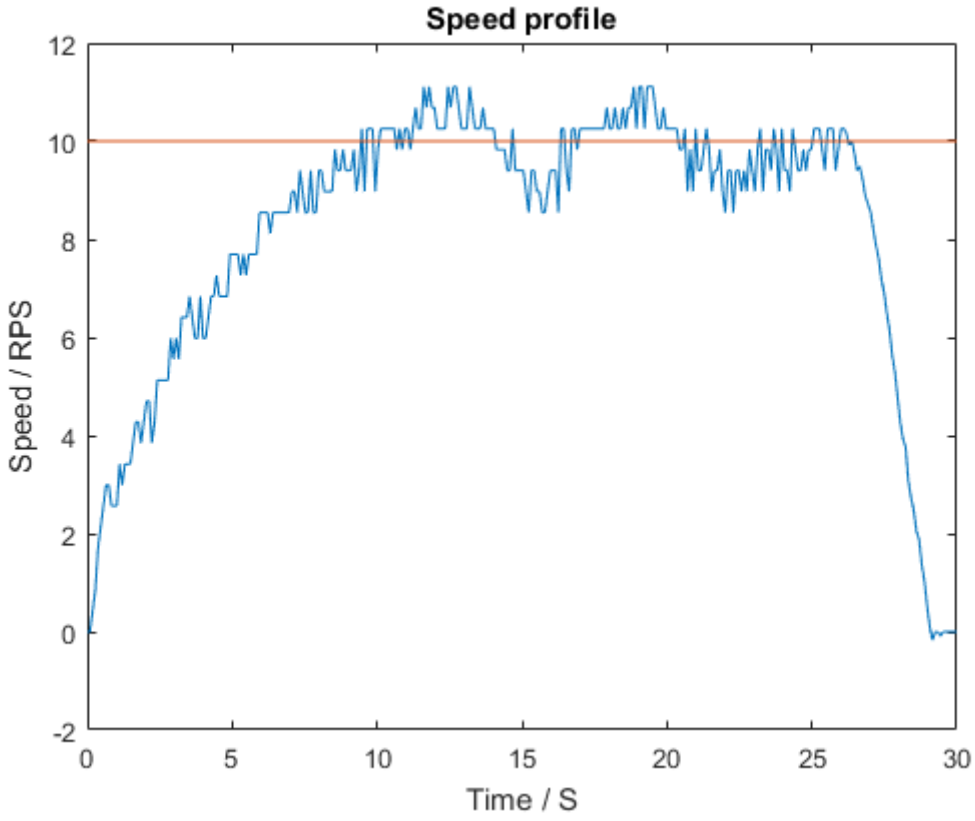
Once the fixed number of rotations is reached, both the PID controller and position control threads are terminated and can only be started again in the serial parsing thread.

Inter-task Dependencies

The four threads mentioned above run concurrently and pass information and parameters to each other through global variables. This means there cannot be any deadlocks; if a parameter is late to be updated then any dependent threads will simply use the previous value. Our solution therefore implements a **soft real-time** system, where latency is a quality metric. The exception to this is the position control thread, where missed deadlines are a quality metric as they can directly cause overshoot.

Verification

The plot below shows the angular speed against time when the serial command demands 200 rotations at a speed of 10 rotations per second (RPS). Within 10 seconds, the motor accelerated from zero to 10 RPS. The PID controller stabilised the speed to within 10% fluctuations within 13 seconds of reaching the target speed. Thereafter, deceleration was constant and took 3 seconds. There is some insignificantly small bounce-back at the end, of less than 0.1 RPS for 0.5 seconds.



Further work

Deceleration

To account for different inertia, we planned to time the acceleration at the start and from this determine how much to scale our deceleration time interval by. Because of the massive rotor has more momentum, the increase in the time required to reach speed should be proportional to the time required to stop. We could use this relationship to tune the deceleration profile dynamically, allowing for precision at all masses.

Melody

We also planned to implement the melody playing function, but run out of time. This would have been a simple matter of modifying the serial parsing thread to take regular expressions concerning the input tune and using those values to set the period of the PWM outputs controlling the motor.