Coursework 2: Brushless Motor Control

Imperial College London

Electrical And Electronic Engineering

EE3-24 Embedded Systems

Authors:
Ahmed Ibrahim (CID: 00845697)
Mattin Mir-Tahmasebi (CID: 00824754)
James Mccomish (CID: 00827979)
Keyan Sadeghi Namaghi (CID: 00861078)

Date: March 2017

1 Introduction

Brushless DC motors, a type of synchronous motor, are used commonly in situations where operational reliability and efficiency is a necessity. This is due to their absence of current connections between the moving armature and stator. A moving stator field formed through commutation of stator coils allows for a rotor with a permanent magnet to rotate. The motor user-control is instructions is defined in the readme in Appendix A.

2 Motor Movement

2.1 State delay

The motor is controlled by cycling through 6 states determined by three drivers L1, L2, and L3, which causes the direction of the magnetic field to change. When this field changes, the rotor's permanent magnet turns to align with the field spinning the rotor around. Control of the rotation speed initially depended on the wait time or time delay between the switching of different states. A desired angular velocity could be calculated through calculating a wait time by dividing the overall delay by 6. This in effect controlled the individual time delay between the 6 motor states:

```
// Convert velocity to wait time
        wait_time = 1000000.0/((output_angular_velocity)*6.0);
```

From the state diagram in Figure 1, it can be seen how the rotor velocity can be controlled. The velocity is dependent on the delay time between the different states. The motor movement using a state delay has a large disadvantage due to the pulsing of each state. This method will therefore have large errors when controlling to turn at a certain velocity. This is especially evident at speeds greatly below synchronous speed.

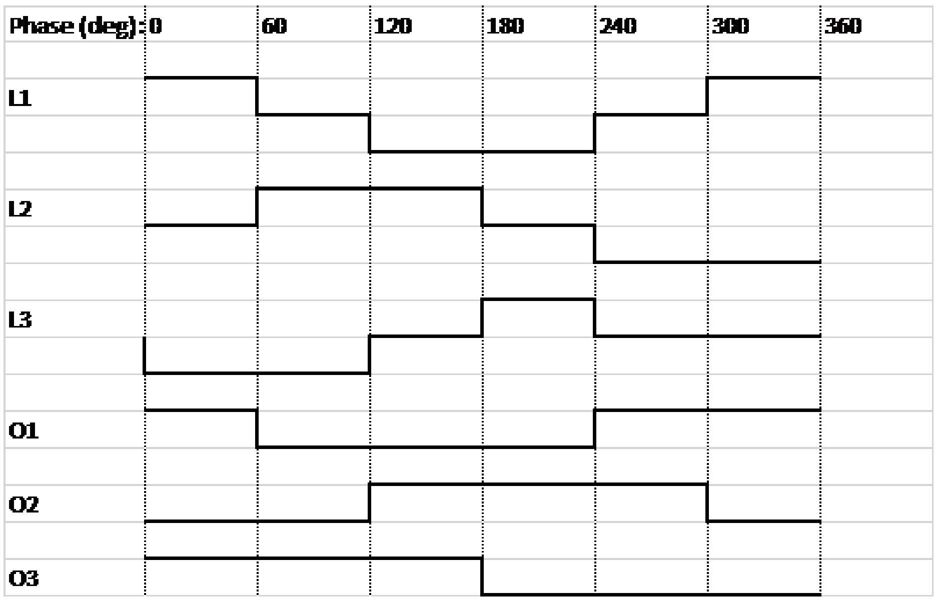

Figure 1: Commutation states

2.2 PWM

Pulse Width Modulation, or PWM, is a technique of manipulating a digital output such that it behaves like an analogue one. A square wave is generated by switching the output between being fully on (3.3V) and off (0V). The ratio of on time to off time is known as the duty cycle, δ . If the frequency of the square wave generated is great enough, the result is as if a controllable steady voltage has been applied to the motor.

The equivalent analogue voltage can be calculated using [1]:

$$\bar{v} = \frac{1}{T}\int_0^T f(t)\mathrm{d}t \tag{1}$$

As $f(t)$ is a pulse wave, its value v_{max} for $0 < t < \delta \cdot T$ and v_{min} for $\delta \cdot T < t < T$. This now becomes:

$$\bar{v} = \frac{1}{T}\left(\int_0^{\delta T} v_{max}\mathrm{d}t + \int_{\delta T}^{T} v_{min}\mathrm{d}t\right) \tag{2}$$

$$= \delta \cdot v_{max} + (1-\delta)v_{min} \tag{3}$$

Therefore the average value of the signal is dependent on δ.
This means, if the duty cycle is set at 0.5, the effect will be the same as applying a constant 1.65V.

The benefit of using PWM rather than simply adding a varying delay between motor state changes is most noticeable at low speeds. When a delay is used, the angular velocity becomes very irregular at low speeds causing the motor to become shaky. However, as PWM gives the effect of a constant voltage the rotation is kept at a constant rate allowing for a far smoother rotation. Another benefit of PWM is that the response to changing the duty cycle is instant where as changing the delay is far slower. As a result, it is far easier to combat steady state oscillations when utilising PWM.

When the motor is controlled using PWM, an appropriate PWM period has to be selected. If the period is made too high, the motors movement will be ”jittery”. If made too low, there would be significant switching losses, and electromagnetic interference would occur. Several values were tested and it was concluded that the default period of 20ms should be used for general operation [2].

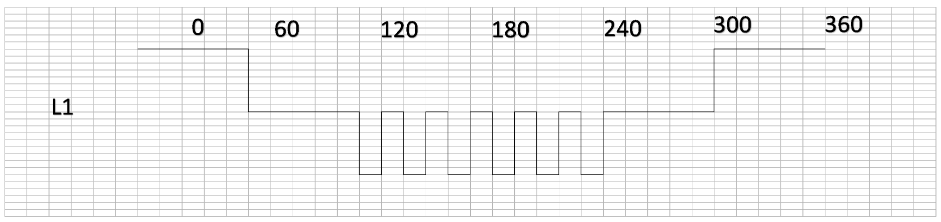

Figure 2: Example of L1 low state being controlled through PWM

In this program, the low states are being controlled through PWM by varying the duty-cycle δ. An example is seen in figure 2, where the low state of output L1 is pulse-width modulate. This motor field is being directly controlled through the motorOut drive state function.

```
//Motor Drive outputs
PwmOut L1L(L1Lpin);
DigitalOut L1H(L1Hpin);
PwmOut L2L(L2Lpin);
DigitalOut L2H(L2Hpin);
PwmOut L3L(L3Lpin);
DigitalOut L3H(L3Hpin);

double delta = 1;
double OGdelta =0;

//Set a given drive state
void motorOut(int8_t driveState){

    //Lookup the output byte from the drive state.
    int8_t driveOut = driveTable[driveState & 0x07];

    //Turn off first
    if (~driveOut & 0x01) L1L = 0;
    if (~driveOut & 0x02) L1H = 1;
    if (~driveOut & 0x04) L2L = 0;
    if (~driveOut & 0x08) L2H = 1;
    if (~driveOut & 0x10) L3L = 0;
    if (~driveOut & 0x20) L3H = 1;

    //Then turn on
    if (driveOut & 0x01) L1L = delta;
    if (driveOut & 0x02) L1H = 0;
    if (driveOut & 0x04) L2L = delta;
    if (driveOut & 0x08) L2H = 0;
    if (driveOut & 0x10) L3L = delta;
    if (driveOut & 0x20) L3H = 0;
    }
```

A side effect of operating a PWM is that it will produce an audio wave of the same frequency due to vibrations caused by the changing magnetic field. If this is in the frequency range of human hearing a buzzing sound will be produced by the motor. Although this range is typically considered to be between 20Hz to 20kHz, human hearing is at its most sensitive between 1kHz and 4kHz (sounds at 100Hz require an amplitude of 2 orders of magnitude larger than a sound at 3kHz [3]). When instructed to play musical notes the motor is programmed to play notes using the ISO's 7th octave ranging from 2093Hz to 3951Hz to ensure it is audible. However, when not instructed to play a tune, this range was avoided as it would be an irritating accidental side effect [4].

3 Command Parsing

The program receives and parses input commands from the user, which determine the program's function. The parsing is done in the main thread, which loops infinitely checking for a keypress. Once a keypress has been detected, the thread reads in the rest of the input until the carriage return. Having stored the input in the `cmd` variable, all currently running threads are terminated, and the rotor is stopped, in anticipation of the new command. It was chosen to bring the rotor to a halt between commands so that a previous command would not impact the result of a new one.

Commands fall under three major groups, denoted by the character they begin with. *R* commands control the maximum number of revolutions the rotor should make, *V* commands control the constant speed at which the rotor should spin, and *T* commands play a melody by modulating the voltage sent to the motor. Parsing begins with reading the first character to identify under which group the command falls.

If the input is an *R* command, `strtod()` is used to convert the next characters into a `double` used as the target number of revolutions, and then again to get the maximum angular velocity, if one has been specified. For a *V* command, the process is very similar, using the same function to get the target angular velocity. The *T* command is slightly more complex, as each note could either be two or three characters long, depending on if it is sharp/flat or not. A lookup function, `get_freq()`, returns the PWM period that corresponds to the frequency of each note. If the note is sharp or flat, it is passed directly into this function, but if it is not, the note is padded with a space to make it two characters long, in order to match the format of the other inputs. The read frequency and duration are stored in a global two-dimensional array, `melody`, which is used by the relevant thread to play the tune.

Once the command has been parsed, only the relevant threads are started, which use the parsed values in order to perform their function.

4 Control Algorithm Used

For controlling the motor speed and the number of revolutions, a Proportional-Integral-Derivative (PID) controller is used. For a *V* command, the desired speed is input to the controller and a value of δ is output. For an *R* command, a target amount is input and the controller outputs a desired output velocity. The main reason for the difference in control parameters is due to the requirement for the added specification for controlling the number of revolutions at a maximum velocity.

4.1 PID

The general form of a PID controller is seen in Figure 3. The output of the controller in the time domain is given by:

$$u(t) = K_p e(t) + K_i \int e(t)\mathrm{d}t + K_d \frac{\mathrm{d}}{\mathrm{d}t}(e(t)) \tag{4}$$

with K_p, K_i and K_d as the proportional, integral and differential weightings respectively and $e(t)$ is the error with respect to the output in time. In the Laplace domain:

$$K_p + \frac{K_i}{s} + K_d s = \frac{K_d s^2 + K_p s + K_i}{s} \tag{5}$$

One can find the appropriate parameters for the weightings by using a model of the system to be able to design for the ideal step response. However, in this case, a mathematical model of the system is not available, so a different method for tuning is used.

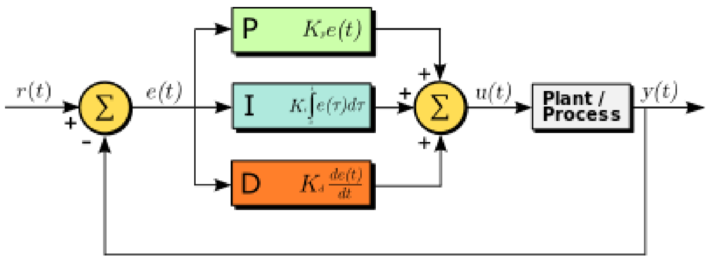

Figure 3: PID diagram [5]

The tuning parameters are approximated using the fundamental characteristics for each of the weightings and the effects they have on the system. An example of this tuning is seen in Table 1:

Table 1: Effects of *increasing* controller parameter [6]

	Rise Time	**Overshoot**	**Settling Time**	**S-S Error**
K_p	Decrease	Increase	Small Change	Decrease
K_i	Decrease	Increase	Increase	Eliminate
K_d	Small Change	Decrease	Decrease	No Change

The general form of a PID controller shown in Equation 4 is in the continuous time domain and must therefore be discretised. The integral and differential terms now become [7]:

$$\int_0^{t_k} e(\tau)\mathrm{d}\tau = \sum_{i=1}^{k} e(t_i)\Delta t \tag{6}$$

$$\frac{\mathrm{d}}{\mathrm{d}t}(e(t_k)) = \frac{e(t_k) - e(t_{k-1})}{\Delta t} \tag{7}$$

The pseudocode for this discretised PID can now be given as:

```
1: while true do
2:     $error \leftarrow target - output$
3:     $integral \leftarrow integral + error \cdot \Delta t$
4:     $derivative \leftarrow \frac{error - previousError}{\Delta t}$
5:     $output \leftarrow k_p \cdot error + k_i \cdot integral + k_d \cdot derivative$
6:     $previousError \leftarrow error$
7: end while
```

This algorithm is used to build the controller for both control of velocity and the number of revolutions.

4.2 Speed control

Using the pseudocode for the PID algorithm, the speed controller can be implemented. For this controller a reference for the target angular velocity in revolutions per second is used as the input. The controller thread uses the current measured velocity to find the error. The thread then changes the value of the duty cycle, δ. The controller is implemented in the thread shown:

```
void pid_vel()
{
    while(true)
    {
        // Using PWM not wait_time
        wait_time = 0;

        // Get time since last PID
        uint64_t time = pid_timer_vel.read_us();
        uint64_t time_since_last_pid = time - prev_time;

        // Calculate errors
        double error = target_ang_velocity - ang_velocity ; // Proportional
        double error_sum = error * (double)time_since_last_pid; // Integral
        double error_deriv = (error - prev_error) /
            ((double)time_since_last_pid*1000000.0); // Derivative

        // Weight errors to calculate delta
        delta = kp_vel*error + ki_vel*error_sum + kd_vel*error_deriv;

        // Store values for next iteration
        prev_error = error;
        prev_time = time;


        Thread::wait(100); // ms
    }
}
```

Controller parameters are set experimentally rather than quantitatively due to the absence of a model for the system. Initially a Ziegler-Nichols method for tuning based on the step-response was used, however, this proved to be difficult due to the non-linearities of the system [8]. These non-linearities arose from the problem of the rotor rotating at non-synchronous speeds, therefore different target velocities would provide large differences in responses. The controller parameters were fine-tuned by measuring the response for speeds. Robustness at low speeds (relatively low oscillations) was compromised with fast tracking (rise time) at high speeds. This is shown in Figure 4.

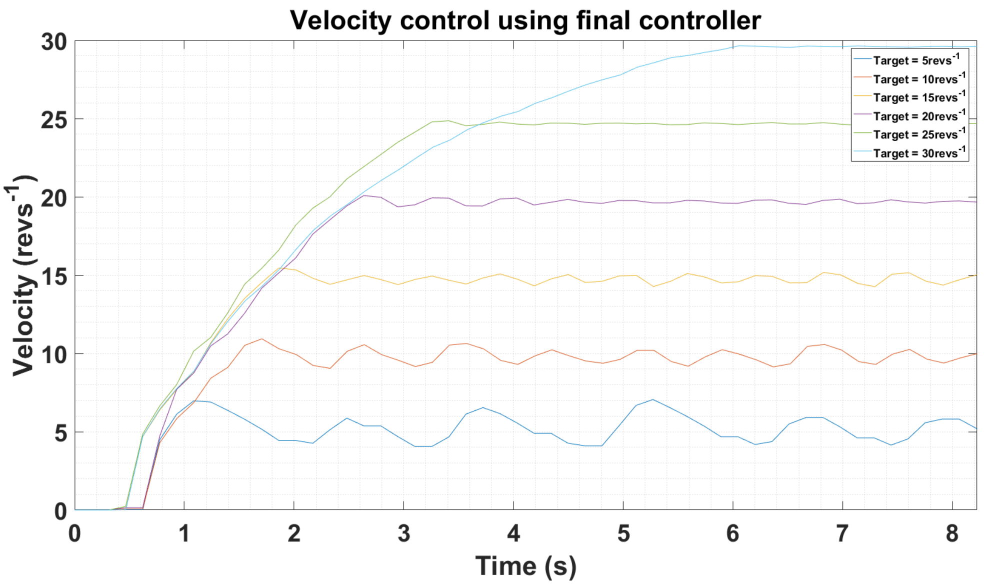

Figure 4: Speed control for different reference velocities

Due to the simplicity of the implemented PID, the controller parameters will be wildly different to that of a more robust controller. The controller parameters used here in this case are $K_p = 1$ and $K_i = 0.000001$. There is no differential term due to the system being already heavily damped. The integral term is to remove the large steady-state error.

4.3 Number of revolutions

In the case of controlling the number of revolutions, the velocity of the motor is dependent upon the target number of revolutions. This value is set as a reference, with the input being the number of revolutions left to reach that target, i.e. the error. However, unlike velocity control, the velocity being set by the controller is not changed by altering the duty cycle but instead by converting the desired velocity to a delay time for changing between motor states. The higher the delay, the longer it takes to switch states, as so the lower the velocity, and vice versa.

The reason for changing the state wait time instead of controlling the duty cycle is because of the relative simplicity in capping the velocity of the rotor, as required by the specification. As a consequence, designing a method for mapping the velocity to a duty cycle is not required. Although this system would be error-prone for controlling the motor to turn at a specific angular velocity, here, our only concern is keeping velocity *below* a certain value, so accuracy is not as relevant.

Due to the inertia of the rotor carrying it past the target, the controller must be heavily damped. During testing, the controller was set to ensure that the error for stopping after a number of revolutions was minimised. To ensure that there is no overshoot, the differential term, K_d for the controller must be relatively large to account for the increasing change in error when reaching the target angular velocity. This is evidently seen in Figure 5, where a number of target revolutions have been experimented. As can be seen, the controller adjusts the output velocity dependent upon the target revolutions giving a characteristic 'arc' shape.

```
void pid_rev()
{
    while(true)
    {
        //Max duty cycle since wait time is being used here for control
        delta = 1;

        // Get time since last PID
        uint64_t time = pid_timer_rev.read_us();
        uint64_t time_since_last_pid = time - prev_time;

        // Calculate errors
        double error = target_revolutions - revolutions ; // Proportional
        double error_sum = error * (double)time_since_last_pid; // Integral
        double error_deriv = (error - prev_error) /
            ((double)time_since_last_pid*1000000.0); // Derivative

        // Weight errors to calculate output
        double output = kp_rev*error + ki_rev*error_sum + kd_rev*error_deriv;

        // Lower limit output to small non-zero value to avoid divby0 error
        double output_angular_velocity = (output > 0) ? output : 0.00000001;

        // Cap at max velocity
        output_angular_velocity = (output_angular_velocity < max_ang_velocity) ?
            output_angular_velocity : max_ang_velocity;

        // Convert velocity to wait time
        wait_time = 1000000.0/((output_angular_velocity)*6.0);

        // Store values for next iteration
        prev_error = error;
        prev_time = time;

        pc.printf("%f \n\r", revolutions);

        Thread::wait(100); // ms
    }
}
```

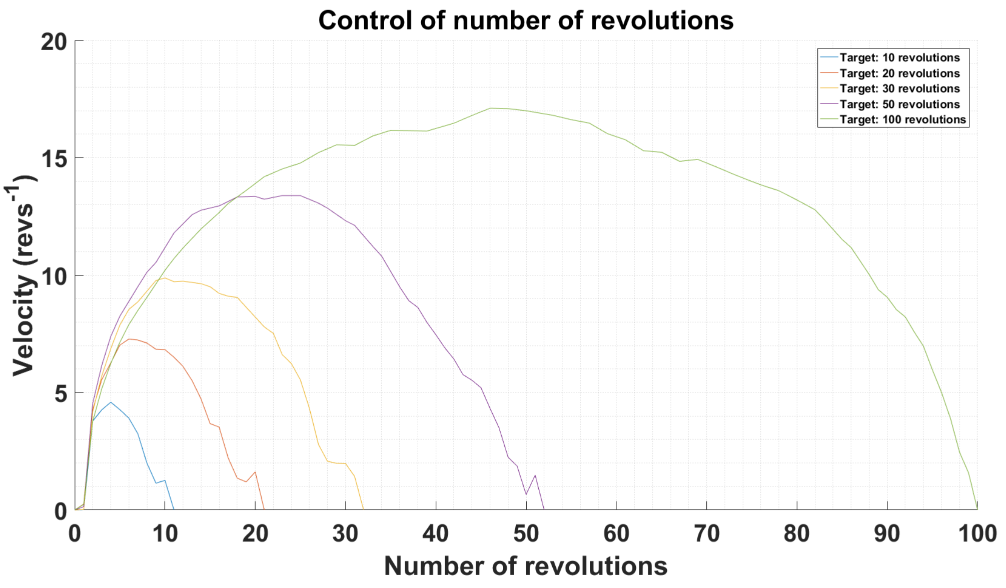

Figure 5: Control of turning for different target numbers of revolutions

The maximum unrestricted velocity is entirely dependent upon the target number of revolutions. It can be seen in Figure 6 that with a larger number of revolutions, the maximum velocity increases in proportion to the target number of revolutions.

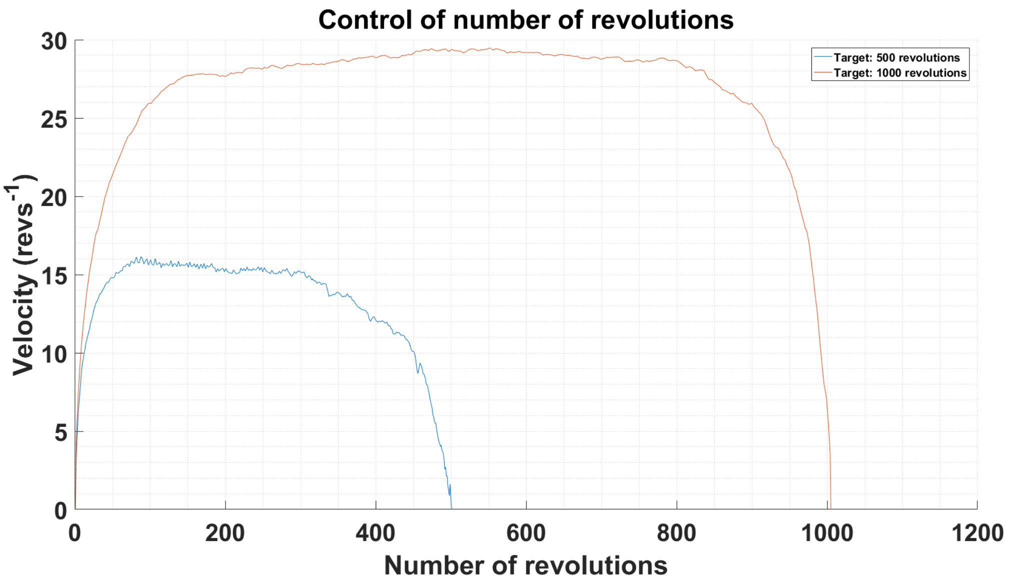

Figure 6: Control of turning for 500 and 1000 revolutions

A small error ($1 < e < 10\%$)[1] can be seen in Figure 5 for each execution as the number of revolutions approaches the target. This is due to the controller being overdamped with the term $K_p \frac{\mathrm{d}}{\mathrm{d}t}(e(t))$ overcompensating for the inertia of the motor. As the motor reaches the target, the angular velocity decreases at a faster rate which forces the controller to increase the velocity again. As a consequence, when the motor reaches the target, the rotor 'wobbles'. Since the interrupt that counts revolutions (`increment_revolutions()`) is activated on the

[1]The percentage error decreases at a higher number of revolutions

rising edge of a photointerrupter, this wobble causes extra rotations to be counted even though no rotation was actually made. This means that although the data may show the target is overshot, in actuality, the system is much more accurate.

```
void increment_revolutions()
{
    revolutions++;

    // Calculate time for previous revolution
    double current_time = (double)rev_count_timer.read_us();
    time_passed = current_time - reference_time;

    ang_velocity = 1000000.0/time_passed;
    reference_time = current_time;
}
```

Another reason for this error is that the velocity is being controlled by way of varying the time between changing motor states. This is slightly less accurate than using PWM but was chosen as it was a simple method that fit into the time constraints for the project.

5 Dependencies

To mitigate the risk of the system being unable to handle the amount of computation that the PID controllers require at high speeds, it was decided that the calculations didnt have to run for every change in rotor state. At max speed the controllers execute around once per 3 rotations which in practice is more than enough as the error does not change that dramatically in such a short time period.

The photo-interrupters are checked regularly and often, and the current angular velocity is calculated using this data gathered. As a result, when the PID has waited and is ready for another calculation, the input data is as recent as possible. The motor state is only changed once a PID thread has been executed but, as it can use the old values whilst the new ones are being calculated, the urgency to produce fresh values is relatively low.

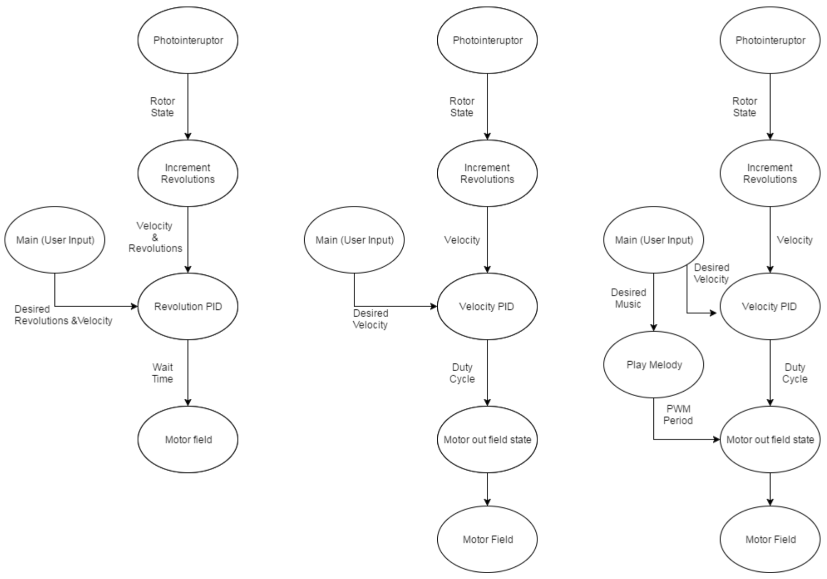

Figure 7: Dependency diagram for the three different command groups (from left to right: *R, V, T*). $A \rightarrow B$ indicates that B depends on A.

With the configuration of threads chosen, the danger of deadlock has been circumvented. There are 4 necessary and sufficient conditions that cause deadlock: mutual exclusion, resource holding, no pre-emption, and circular waits [9]. Figure 7 shows that for any command given, the system has no cyclic dependencies, so no circular wait could possibly exist, and as such, no deadlock can exist.

6 Task Analysis

The system can be classified as *soft real-time*, as late results are still able to be used, but the longer threads take (or equally, the less often they are executed) the worse the responsiveness of the controller is. As such, there are no 'deadlines' per se; the limit to the time the threads can take is dependent on the user's desired performance.

It is difficult to analyse the actual latency or utilisation of tasks because scheduling is not rate-monotonic. This is the case because rate-monotonic scheduling requires tasks to have fixed execution times, and a fixed initiation interval for each task, which the system does not necessarily have. Threads are not initiated at regular intervals, but rather, yield for an amount of time after having finished executing.

What can be attempted is an analysis with representative values for times of execution. Using the execution of a *V* command as the example, the measured execution times for the running threads and interrupts are listed in Table 2.

Process	Time taken (μs)
`check_photo()`	1.875
`increment_revolutions()`	12.385
`move_field()`	35.95
`pid_vel()`	22.35

Table 2: Execution times for different processes

Due to an oversight in design, a `Thread::wait()` was not inserted into `move_field()`, which increases CPU utilisation unnecessarily. Assuming there would be a reasonable wait at the end of the loop in that function, (around 1ms) `check_photo()` would be executed most often, at once per 100μs. Using this period as our deadline, it can be seen that there is enough time for all processes to execute before `check_photo()` is initiated again, as displayed in figure 8.

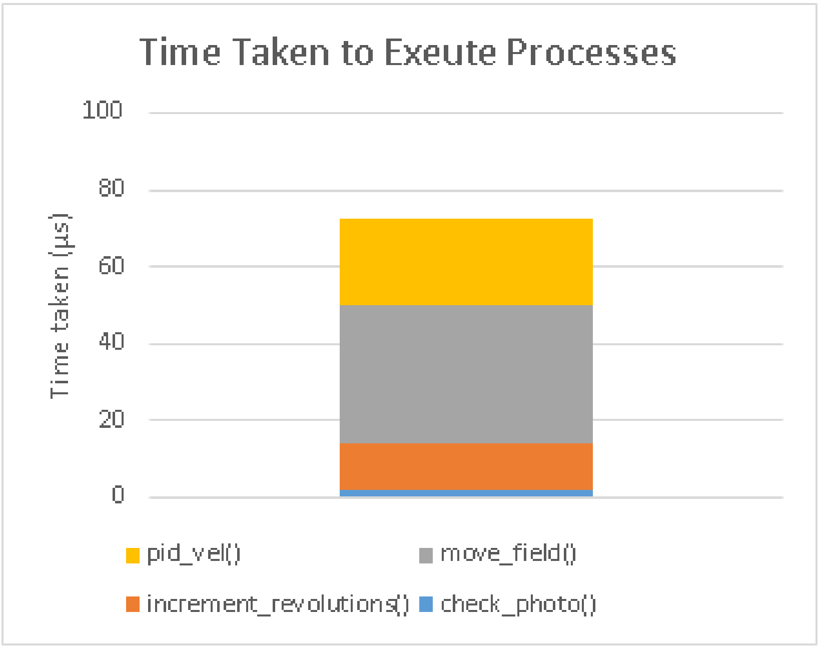

Figure 8: Time taken to execute processes

7 Improvements

There is a bug in the software that arises when entering a target velocity. The program's maximum velocity in this state is 35 *rev/s*, which is well below its maximum of around 70 *rev/s*. This is due to addition of the `play_melody()` thread. When entering a desired melody, the target velocity is 40 *rev/s* which is above the velocity PID maximum.

These errors occur due to the period of the PWM not being reset before the start of each thread. This becomes evident when running the motor to a target velocity via the *V* command, after the melody command. In this case there is no cap on the velocity however the last note played is heard. Again, this is due to he period of the PWM not being refreshed.

Another improvement which can be made is setting a fixed minimum wait time in the `move_field()` thread. The absence of this wait allows the thread to be active for the maximum time the thread can exist in one iteration. This while allows for a higher maximum velocity, greatly increases CPU utilisation.

A README

Revolutions
Enter revolutions and maximum velocity the specified way. If you want to execute a revolutions command after having executed a melody command, first reset the board (due to a bug; see improvements section).
Velocity
Enter the desired velocity the specified way. Maximum velocity is capped to 35rev/s initially (due to same bug as above). Executing a velocity command after having executed a melody command allows it to reach max velocity.

References

[1] Francesco Vasca and Luigi Iannelli. Dynamics and control of switched electronic systems. *Advanced Perspectives for Modeling, Simulation and Control of Power Converters*, 2012. pages 3

[2] Pwmout - handbook — mbed. `https://developer.mbed.org/handbook/PwmOut`. (Accessed on 03/22/2017). pages 3

[3] Steven W. Smith. *The Scientist and Engineer's Guide to Digital Signal Processing*. California Technical Publishing, San Diego, CA, USA, 1997. pages 4

[4] Iso 16:1975 - acoustics – standard tuning frequency (standard musical pitch). `https://www.iso.org/standard/3601.html`. (Accessed on 03/22/2017). pages 4

[5] Pid en - pid controller - wikipedia. `https://en.wikipedia.org/wiki/PID_controller#/media/File:PID_en.svg`. (Accessed on 03/21/2017). pages 6

[6] Control tutorials for matlab and simulink - introduction: Pid controller design. `http://ctms.engin.umich.edu/CTMS/index.php?example=Introduction&section=ControlPID`. (Accessed on 03/21/2017). pages 6

[7] Discrete pi and pid controller design and analysis for digital implementation. `https://www.scribd.com/doc/19070283/Discrete-PI-and-PID-Controller-Design-and-Analysis-for-Digital-Implementation`. (Accessed on 03/21/2017). pages 6

[8] John G Ziegler and Nathaniel B Nichols. Optimum settings for automatic controllers. *trans. ASME*, 64(11), 1942. pages 7

[9] Abraham Silberschatz, Peter Baer Galvin, and Greg Gagne. *Operating System Concepts*. John Wiley & Sons, Inc., New York, NY, USA, 6th edition, 2001. pages 12