Imperial College
London

EE3-24 EMBEDDED SYSTEMS

IMPERIAL COLLEGE LONDON

DEPARTMENT OF ELECTRICAL AND ELECTRONIC ENGINEERING

CW 2: Brushless Motor Controller

Team: Eastern Syndrome
Adil Malik (CID: 01065192)
Dimitris Moniatis (CID: 01095618)
George Gabriel (CID: 01066515)
Styliana Elia (CID: 01228533)

Date: March 22, 2019

Contents

1 Introduction

The aim of this project is to study real-time constraint analysis using interrupts and multi-threading. This will be done by writing the software for a precision brushless motor controller for robotics, to satisfy several specifications.

2 Motor Control

2.1 Motor Operation

The second coursework specification for embedded systems is to control BLDC motor whilst ”simultaneously” running other tasks in the background using threading. The justification for not using a brushed motor and accepting the more complex control requirements of a brushless motor is due to the disadvantages of the former, namely:

- Brushes arc and overheat – limited speed
- Brushes wear out – limited lifetime
- Low power to weight ratio – Not ideal for high performance applications

The motor provided is of the star configuration resulting in 3 output leads (phases), each connecting to a separate pair of coils in the stator. Each phase is connected to a half bridge which is responsible for driving it high or low. The three inputs, L_1 L_2 and L_3 are responsible for setting the magnetic field in the motor. Essentially a magnetic force vector is rotated through 6 steps inside the motor. The rotor of the motor tries to align itself with the artificially created magnetic field in the rotor air gap resulting in torque production and shaft rotation. The three inputs L_1 L_2 and L_3 correspond to $2^3 = 8$ state combinations. States 6 and 7 result in braking action as the 3 motor phases are either shorted to VCC or GND. States 0 to 5 can be used for either positive or negative torque production. In the table below the 6 possible drive states are mapped to motor phase outputs.

Positive rotation commands result in anti-clockwise rotations and vice-versa.

State	L1	**L2**	**L3**
0	H	-	L
1	-	H	L
2	L	H	-
3	L	-	H
4	-	L	H
5	H	L	-
6	H	H	H
7	L	L	L

Table 1: Mapping from sequential drive states to motor phase outputs

2.2 Cascaded PIDs

Cascaded control is one of the most popular complex control structures implemented to improve the disturbance rejection properties, settling time and decrease steady state error of a controlled system.
Real life motors are non-linear due to several factors such as friction, inertia and power saturation. For this reason a cascaded PID controller is used to reduce these non-linearities.

In a dual cascaded structure two loops are used, each tasked with controlling one aspect of the motor's characteristics. In our case the outer loop controls the motor's position while the inner loop controls the motor's velocity. The first error block subtracts the position feedback sensed from the motor (P_fb) from the position set-point (P_sp) set by the user. This results in a position error (P_err) which is then fed to the Position PID. From that, a Velocity set-point is created (V_sp) which is then fed into the second error block. This in turn creates a velocity error (V_err) by subtracting the velocity feedback (V_fb) from the velocity set-point. The error is then provided to the velocity PID controller which outputs velocity. This is in turn fed into the velocity limiter block which caps the motor's maximum allowed velocity and outputs the PWM signals controlling the BLDC motor. Figure 1 demonstrates the cascaded PIDs used for the motor control.

In our implementation, both controllers run inside the `motorControlLoop()` function and are called every 100ms. In case the user wants to cap the motor speed, the V_lim input to the velocity PID can be used to limit the motor's maximum speed.

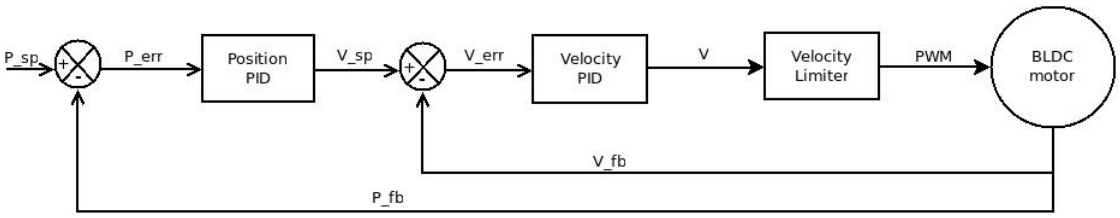

Figure 1: Cascaded PID Motor Control

2.3 Velocity Control

Velocity is controlled using a standard PID loop. Essentially, the standard approach is implemented.

$$v(\tau) = K_P V_{error} + K_I \int V_{error} d\tau + K_d \frac{dV_{error}}{d\tau} \tag{1}$$

where K_P, K_I = and $K_D = 0$ stand for the proportional, integral and derivative gain constants respectively. For the velocity control a high precision counter is used in order to get more accurate time measurements.

2.4 Position Control

Position is controlled using a standard PID loop. Again, the standard approach is used.

$$v(\tau) = K_P V_{error} + K_I \int V_{error} d\tau + K_d \frac{dV_{error}}{d\tau} \tag{2}$$

where K_P, $K_I = 0$ and $K_D = 0$ stand for the proportional, integral and derivative constants respectively. For the position control we use the high precision encoder but only trigger on the rising edge of channel B of the encoder. This is less computationally expensive for the microcontroller and results in about 284 increments per revolution.

2.5 Tuning the controller

To tune the controllers the standard empirical approach was used. K_I and K_D were set to 0 and K_P was increased until oscillations were observed. Following, K_P was decreased by 20% and then K_I and K_D were tuned to obtain appreciable steady state performance and disturbance rejection respectively. For speed control, using only K_P and K_I proved to be sufficient. For position control, K_P and K_D were used instead.

The PID constants for the velocity control are empirically calculated with corresponding values for $K_P = 4000$, $K_I = 15$ and $K_D = 0$. Since K_D constant is zero, a Proportional Integral (PI) controller is essentially implemented which meets all the velocity control requirements.

The PID constants for the position control are empirically calculated with corresponding values for $K_P = 45$, $K_I = 0$ and $K_D = 95$. Since K_I constant is zero, a Proportional Derivative (PD) controller is essentially implemented which meets all the velocity control requirements. It must be noted that applying integral action on the position loop is not straight forward because once the motor halts with some steady state error, the control loop is no longer able to restart the motor and the integral winds-up. As a result integral action can only be applied in conjunction with some heuristic open-loop motor startup code. However, this was deemed unnecessary because our steady state error seems to always be well below 1 revolution.

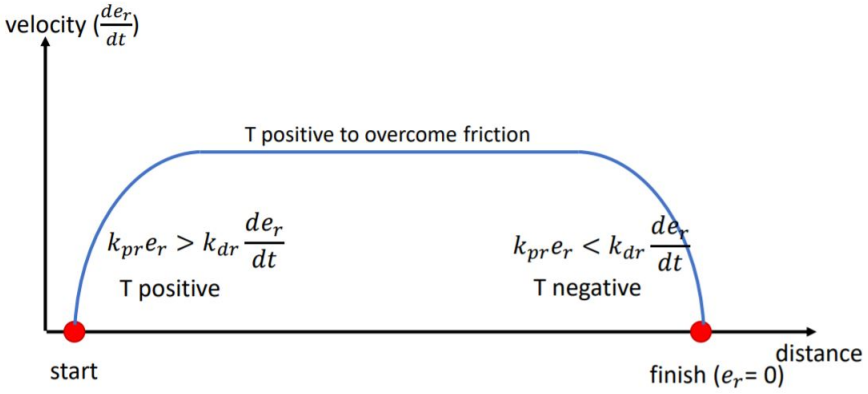

Figure 2: Rotation Control diagram

3 Task Analysis

In this section system task analysis will be examined. It must be noted that for our system design and implementation the lasted version of `mbed OS5` was used.

3.1 Self-starting the motor

The interrupt routine for the closed loop commutation requires a continuous stream of interrupts to increment the motor states and hence rotate. At startup, the motor is stuck in the same state so some open-loop commutation routine is required to manually spin the motor in the correct direction.

```
void motorCommutateOpenLoop(int direction){
   while((abs(highPrecisionCounter)<100) ){ //Loop until you get some
    ticks, direction not so important as commutate cycle forces correct
    direction
   int8_t State = readRotorState(); //Read where the rotor is currently
  for(int i=0;i<5;i++){ //Run a seqeunce from the current state to the
    next 5
   (direction==1) ?  motorOut(((State+i)-orState)%6,100) : motorOut(((
   State-i-1)-orState)%6,100); //Exact rotation direction that important
    as after fist edge, closed loop control takes over.
    wait(0.5); //Delay for some suitable startup speed
    }
    }
}
```

3.2 Playing music

As an optional task it is required to allow the motor to play a melody *while* it is spinning by modulating the control voltage. The way this was achieved was by using the three phase half-bridge exclusively for motor commutation and control loop control while modulating the frequency of the PWM signal of the top P-MOS, set at a very high duty cycle to eliminate large ripple. We note that modulating both the bridge and the top MOSFET will produce further mixing products but most of these are too high and are naturally filtered out by the system.

We opted for this approach because we empirically determined that for our setup, controlling the speed of and commutating the motor using the three phase Half-bridge resulted in superior control and the modulating the top MOSFET frequency resulted in the most audible sounds (perhaps due to stronger mixing products). Due to this, a constant duty cycle of 0.9 is set to the top P-MOS (Q1) and the frequency of the PWM is varied according to the note played using pre-defined frequencies in the function `setBridgePWMFrequency`. The small size of the brushless motor used provides relatively poor filtering to low frequency switching, making the use of high frequency PWM mandatory for good control and low torque ripple. As a result, the period of the the initial musical period is set in the interval of $127 - 225\mu s$ which corresponds to the highest piano octave (key numbers 98 to 108) and frequencies

ranging from 4.4 to 7.9 KHz. Playing the music is handled using an EventQueue. The EventQueue is a class that allows events (callbacks) to be scheduled and executed in the future. This is very elegant, as it avoids race conditions and makes inter-thread communication very simple. There is another thread that is responsible for running those callbacks. They don't introduce particular overhead, as the only thing that happens is changing the PWM frequency. When a new music sequence is received, all old events are cancelled and new ones are scheduled in the queue. We make sure to keep track of the event ids so that we can manage them later.

```
void setMusicFrequency(int period_us){
    topFET.period_us(period_us); //225, 127
    topFET.write(1);
}

void setBridgePWMFrequency(int period_us){
    //Set PWM Frequency to 10KHz
    L1L.period_us(period_us);
    L2L.period_us(period_us);
    L3L.period_us(period_us);;
}
```

3.3 BitCoin mining at the background

The Bitcoin mining task is the computation of SHA-256 hashes of the 64-byte data sequence $< data >, < key >, < nonce >$, where ¡data¿ is 48 bytes of static data, $< key >$ is an 8-byte number specified by a host over a serial interface and $< nonce >$ is an 8-byte number that can be freely chosen.

In order to measure system efficiency, BitCoin sequences are hashed in the background. The lower the system load, the higher the mined hash rate. As expected, the highest hash rates are observed whilst the motor is stopped due to the MCU being interrupted less often. Hash rates for different system loads are tested and their corresponding values are shown below:

	Hashes Rate (per second)	Hash rates (per minute)
Motor stationary/BitCoin only	5803	348,180
Music only: five notes	5750	345,000
Full speed, no music	5250	315,000
Full speed, 5 notes	$\approx$ 5230	313,800

Table 2: Assessing performance with Hash rate

The maximum hash-rate obtained is 5803 hashes per second. This corresponds to running the BitCoin mining operation solely and therefore motor is stationary. As the system load increases, the hash rate per second is decreased. The lowest hash-rate obtained is 5230 hashes per second. For every note, a different hash rate is obtained and therefore the average is computed. This corresponds to setting the motor to run at its maximum speed while playing music (for this testing five notes are used).

Since the system load is now at its maximum (both music played and motor running at full speed), the mined hash rated are decreased by 9.87% which is considerably low as desired.

3.4 Communication with Host

3.4.1 Outgoing Communication

We have a thread that is responsible for all writes to the serial interface. This thread, communicates with the BitCoin mining thread via a MailBox. Note that it does not need to communicate with any other thread, as the only things we need to print are the hash statistics. The Mail class works like a FIFO queue, but it also manages a Memory Pool for dynamically allocating memory as required. This was very convenient as it is all handled by mbed and is fully thread safe.

3.4.2 Incoming Communication

Every time a character is written to the serial input, an interrupt is triggered. The interrupt handler for this interrupt is responsible for adding the character to a queue. We have an Input decoding thread, which continuously polls this queue, while maintaining an internal buffer (that is protected from buffer overflows). When a carriage return character is received, the decoding thread attempts to decode the command and act accordingly. To make decoding as fast as possible, we opted not to use a RegEx library. The decoding thread communicates with the motor controller via some shared variables. We took care to make sure that access to those variables is atomic and race conditions are avoided. In addition, the input thread needs to communicate with the BitCoin mining thread in order to give it a new key when that is available. Sharing this key is synchronised using a Mutex.

4 Inter-task Dependencies

The following tasks are running “in parallel” in our system:

- Input Decoding Thread
- Output Thread
- Music Playing Thread
- BitCoin Mining Thread
- Motor Control Ticker (every 100ms)
- Input Character ISR

The way those tasks communicate with each other is summarised in the diagram below:

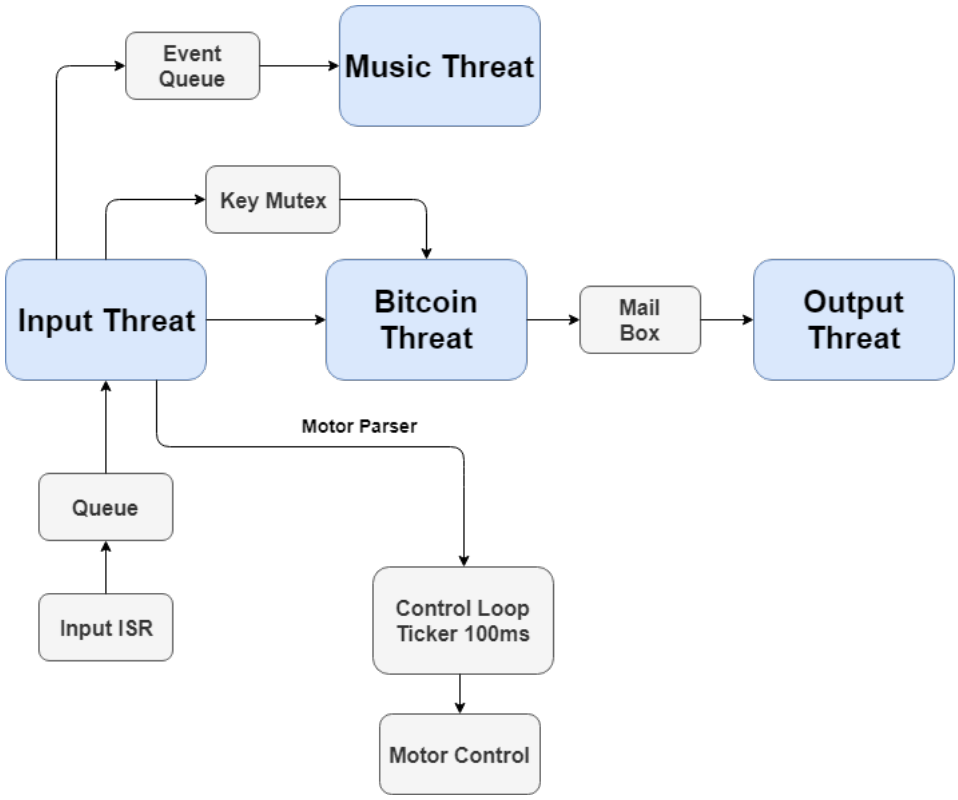

Figure 3: System block diagram

Mutual exclusion is required when accessing some resources. These includes the bitcoin key, and the music sequence. Although we took care to synchronise access to those resources using appropriate mechanisms, we made sure that any circular waits between threads are avoided. This basically guarantees that there will be no deadlocks.

5 Measuring CPU utilisation

A classic way of measuring CPU utilisation is to toggle a pin during any idle time and measure the resulting frequency. This way a large frequency represents low CPU utilisation and low frequencies indicate relatively high CPU utilisation. We measured the CPU utilisation in this manner for various operating conditons:

	Bitcoin On	Bitcoin Off	Test Case
Motor Full speed	2.675kHz	2.262MHz	Average
Motor Stationary	2.957 kHz	2.418 MHz	Best
Full speed with Music	2.27 kHz	2.2487 MHz	Worst
Music only	2.9570 kHz	2.4186 MHz	Average

Table 3: CPU idle times with Bitcoin On and Off

6 Performance

The motor specifications met are demonstrated in the table below. As explained in the previous sections, the motors allows self-starting and can spin for a defined number of rotation at a defined maximum angular velocity and stop without overshooting, This was achieved with the cascaded PIDs motor control and appropriate tuning. Also, the motor can now play a melody using the commmand `T` followed by pairs of notes and durations. This is achieved by using the three phase half-bridge for motor commutation while using the pulse width of the P-MOS controlling the motor voltage to set the motor speed by modulating the motor voltage.

Specifications met	
Motor spins for a defined number of rotations and stops without overshooting	✓
Motor spins at a defined maximum angular velocity	✓
Nearest rotation for a number of rotations	✓
Nearest one rotations per second for angular velocity, down to 5 rotations per second	✓
Motor is self-starting	✓
Motor plays a melody while it is spinning by modulating the control voltage	✓

Table 4: Specifications summary Table

It must be noted that the motor operates at a much higher precision than the one required in the standard specification; the highest precision reached was to the nearest nearest 0.5 rotations for number of rotations instead of the normal precision of a nearest one rotation for number of rotations.

Finally, the motor control behavior is tested and the plot obtained satisfies all the requirements. A plot demonstrating the positional control for 50K counts is demonstrated below:

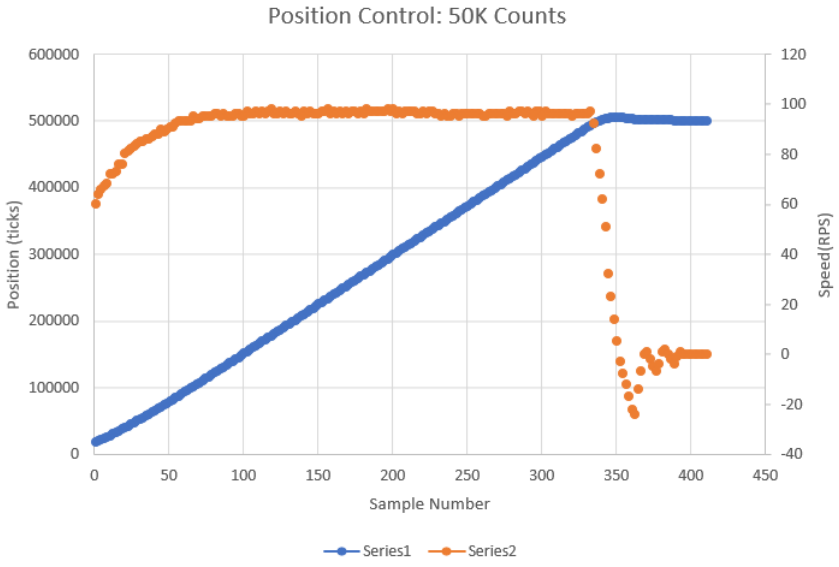

Figure 4: Motor Control: Position and Velocity Control plots

References

[1] E. L. M. S. X. Khin HooiNg, Che Fai Yeong, **Alpha Beta Gamma Filter for Cascaded PID Motor Position Control[PDF]**, 2017. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S1877705812025556 pages