Imperial College London

COURSEWORK

IMPERIAL COLLEGE LONDON

DEPARTMENT OF EEE

EE3-24 EMBEDDED SYSTEMS (2017-2018)

Brushless Motor Controller

Author: In Chris We Trust
Constantinos Nicolaou (CID: 00953169)
Loizos Papachristoforou (CID: 00941676)
Christodoulos Stylianou (CID: 00819357)

Date: March 27, 2018

Contents

1 Introduction

This report presents the implementation of thread-safe firmware, using an embedded system, for precisely controlling a brushless synchronous motor. The system has three main objectives. The first objective is for the motor to spin for a defined number of rotations and stop without overshooting. The second objective is to spin at a defined angular velocity. The final objective is to execute a bitcoin mining kernel in the background. The parameters required by each objective can be set by the user via the serial port.

2 Control Algorithm

Over the years the use of motors in embedded systems has been increased. Any device that is able to move, uses a motor. There is a particular interest in systems which use motors for precision movements such as robotic and healthcare. Motors being used for precision movement operations require a control algorithm which is responsible to control the movement of the motor. To achieve the required behavior there are two main parameters which must be controlled, position and speed.

2.1 Motor

In this project, a brushless, synchronous motor is used which disregards the mechanical drawbacks of a brushed motor. The spinning of the motor is synchronised to the signal applied to it. There are 6 separate coils positioned round the outside of the motor, connected in star arrangement. The motor has three inputs L1,L2,L3 responsible for driving the magnetic field. Each input is connected to two transistors (drivers) and it can be driven high or low. Current is applied to the coils (motor windings) and a magnetic field is generated (stator field). According to the switching state of the drivers, the stator field can be generated at a particular angle. There are 6 possible driver states hence 6 possible orientations of the field. By stepping through these states the magnetic field is rotating. Inside the motor there is a permanent magnet situated on the rotor and a constant magnetic field (rotor field) is created around it. In doing so, the rotor spins to align it self with the stator field.

When the rotor field is not aligned with stator, a turning force is created, called motor torque. If the stator field leads the rotor field, an accelerating torque is created causing the rotor to accelerate. On the other hand if the stator field lags rotor field, a decelerating torque is created causing the rotor to slow down. In other words, controlling the torque via controlling the stator field, leads to the control of the motor. This concept demonstrates how the motor can rotate and accelerate or decelerate.

In order to be able to precisely control the motor, the angle of the rotor must be sensed so that the coil energising sequence will start from the proper state. This is crucial to determine if the stator field leads or lags the rotor field. Since a brushless

motor is used, it requires external components for sensing its rotor position. For this reason, an optical disc alongside with 3 photointerrupts is used. Each photointerrupt can be high or low, depending on the position of the disc. The photointerrupts read the disc state. Rotor and disc positions are offset by some random angle at assembly. Therefore, the position of the disc can be mapped to the rotor state by adjusting the offset using a correction constant. Effectively, recording the position of the disc allows for tracking the position of the rotor indicating the appropriate coil energising sequence to follow in order to rotate stator field and drive the motor.

As already mentioned, the rotor is able to accelerate or decelerate, relative to the phase difference of the stator and rotor fields, by creating torque. However in precision control it is crucial to control how fast or slow the motor can accelerate or decelerate and when. This is achieved by controlling the torque via modulating the current to the motor windings using PWM, from the micro-controller, to generate the coil energising sequence. By varying the duty cycle of the PWM, the current to the motor windings varies hence the strength of the stator field can now vary, creating variable torque. In other words, controlling the duty cycle of the PWM is what will make the motor to accelerate or decelerate via a certain amount by creating a torque proportional to the duty cycle. To decide by how much and when, a controller is introduced being responsible to control the duty cycle of the PWM via controlling the position and speed of the motor.

2.2 Controller

The main idea behind the controller is that the motor will be able to spin at a desired velocity for a defined number of rotations, both defined by the user, and stop without overshooting or oscillating. To be able to implement the controller, the velocity and number of rotations of motor have to be measured first. Every time the rotor covers 60° a position variable is incremented or decremented by one after comparing the current rotor state to the previous. Therefore the position variable represents the number of rotations times 6. Velocity of the motor then can be calculated as the derivative of position. For better results the velocity is calculated on a fixed time interval.

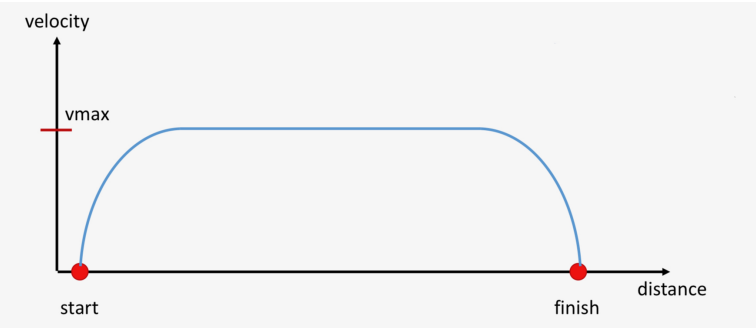

Figure 1: Motor velocity for a given number of rotations

The challenge implementing the controller is that the motor must decelerate before the rotation target is reached and stop it on the target. In other words the motor should behave as shown in Figure 1. To achieve the desired performance two controllers were implemented, speed and position controller which are both combined by choosing each time which of the two outputs to apply.

2.2.1 Position Controller

This is a proportional derivative (PD) controller, since the motor cannot stop instantly when it reaches its target, given by

$$y_r = k_{pr} * E_r + k_d * \frac{dE_r}{dt} = k_{pr} * (p - r) + k_d * \frac{dE_r}{dt} \quad (1)$$

where y_r is the output of position controller, E_r the position error, p is the reference rotations defined by the user, r is the measured rotations, k_{pr} is the proportional gain and k_d the derivative gain both which have to be set experimentally. Position controller is responsible for stopping the motor when it reaches its target. Since forcing the motor to stop will cause overshoot, the stopping cannot be done by heuristic methods such as turning it off earlier. Differential controller is responsible to deal with impulses and overshoot behavior acting as the system's damping which is control by k_d.

2.2.2 Speed Controller

This is a proportional (P) controller, since velocity is the first integral of acceleration, given by

$$y_s = k_{ps} * E_v * sgn(E_r) = k_{ps} * (s - \mid v \mid) * sgn(E_r) \quad (2)$$

where y_s is the output of speed controller, E_v the speed error, s is the reference speed defined by the user, v is the measured velocity, $sgn(E_r)$ is the sign of the position error indicating the direction at which the motor is rotating and k_{ps} is the proportional gain which has to be set experimentally. Speed controller is responsible to force the motor reaching from its current speed at a defined speed and keep rotating at that value.

2.2.3 Combining Controllers

Following the performance indicated by Figure 1 the motor must accelerate while at rest to a constant reference velocity and then come back at rest after a defined number of rotations. This can be achieved, as mentioned in Section 2.1 by controlling the duty cycle of the PWM. The outputs of the controllers can be transform into PWM widths each time setting the duty cycle at a different value, controlling the torque and hence the behavior of the motor.

If it is desired to rotate the motor forever, then only y_s is used in order to accelerate the motor at a desired speed and keep it rotating forever. On the other hand if

it is desired to rotate the motor at a maximum speed, the maximum pulse width is passed to set the duty cycle at maximum value.
If however the desired characteristic is to rotate the motor at a defined speed for a defined number of rotations then the two controllers have to be combined as follows:

$$y = \begin{cases} min(y_s, y_r) & \text{if } v \geq 0 \\ max(y_s, y_r) & \text{if } v < 0 \end{cases} \tag{3}$$

Both outputs of the controllers are calculated but the chosen one is the one with the lowest or most negative value depending on the direction of rotation, which is defined by the sign of velocity. In other words the most conservative controller output will be selected to set the duty cycle. This implies that, assuming positive direction, while the current position is far away from the desired position then the output of y_r will be a large positive value but at the same time the output of y_s will be fluctuating around zero once the motor reaches maximum speed.

To produce the effect shown in Figure 1 where the velocity of motor ramps up and stays constant then the output of y_s must be chosen. On the other hand once the target position is near, the motor should slow down and come at rest. While the motor approaches the target position, y_r will become negative to damp the motion of the motor and y_s will stay positive since the motor slows down and current speed is below reference. Therefore here the value of y_r must be selected.

2.2.4 Tuning the Controllers

The important part once the controllers will be designed, is to tune them properly since a number of trade-offs have to be considered. Each controller has been tuned separately.

Regarding the proportional gain term of the speed controller, increasing its value may cause fluctuations and make the controller lagging, especially for slow speeds. On the other hand, reducing it to a low value it will increase the steady-state error making the motor spin much slower than the reference speed. To tune the speed controller, the gain was initially set to 1. The experiment was to increase the gain by one each time until the first fluctuations show up. That value was recorded, say as k, and the value of k_{ps} was set to $k_{ps} = k - 1$. The experiment was implemented for slow and fast speeds leading to a selection of two k_{ps} values, one for speeds below 20 and one for over.

The proportional gain term of the position controller has to be set close or just above k_{ps}. The important part in the position controller is the derivative term since is the one that selects the system damping. Setting it to small value then the motor won't be able to stop on time resulting to overshoot. On the other hand, large values will damp the system too much making the movement last longer than necessary. The experiment was started via selecting $k_d = 20$ as proposed in the lab sessions. There was no overshoot observed at various speeds and the motor was slowly spinning as it was kept approaching the reference position, so the value of k_d was reduced until

overshoot occurs. Then the value of k_d was set to the value where overshoot occurs plus 1. For slow speeds the system didn't need any significant damping so it was set to 1.

3 Task Analysis

The starter code was responsible for driving the motor at a maximum speed via continuously checking the state of photointerrupts. As a result to that, the system had limited extra processing power to perform other tasks since resources was wasted. To allow other tasks to take place the polling loop was converted into interrupts. The final system can be itemized into 5 main tasks performing the following:

- Motor rotation
- Serial interface
  - Input from serial port
  - Output to serial port
- Speed control
- Bitcoin mining at the background

Figure 2 shows a graphical overview of all the tasks performed by the implemented system. Each of the tasks is explained below in a more detail way always referring to this figure.

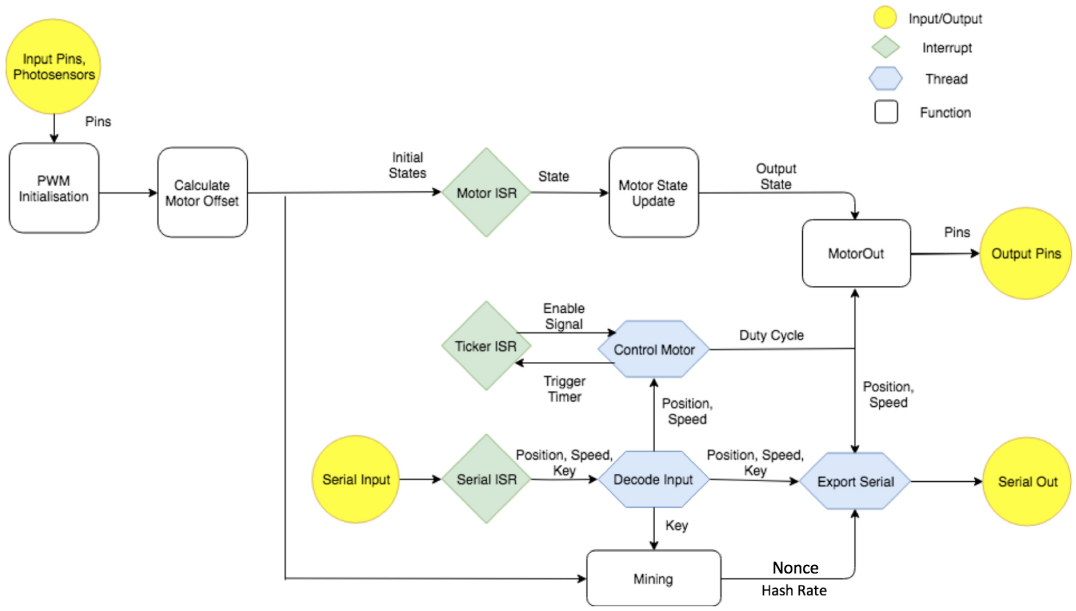

Figure 2: Overview of the tasks performed by the system

The highest priority tasks are all the interrupts. The threads in this implementation have all the same priority (Normal) and the lowest priority task is the Mining task.

3.1 Communication with Host

A requirement for the system is to be able to communicate with the host through the serial port. Splitting the communication task into two separate tasks, one for receiving messages and one for sending messages, will help in avoiding problems such as clashes causing instability with the serial port resources between different tasks. Moreover considering communication as two distinct tasks will also make it easier to analyse the behavior and performance of the system in terms of timing constraints and latency adding flexibility to the way the resources are used.

3.1.1 Outgoing Communication

The first task of the system is responsible for receiving messages from different parts of the code and export them to the serial port. The task is performed by `commOutT` thread. In order to be able to communicate and pass information from other threads or pieces of code, a `Mail` structure was used, which essentially provides a FIFO queue, allowing up to 16 messages to queue. This can allow concurrent threads to independently send data back to the host.

Messages in the queue are added via `putMessage` function which takes in the assigned code of the message, to distinguish between various messages, and the data. This is helpful since it is a reentrant function hence the messages can be added in the queue from any point in the code without changing the results. The functionality of the thread is performed by `commOutFn` which is basically responsible to take the message from the queue and print it to serial port. The function is an infinite loop which waits for a message to be added in the queue. Once the message is added in the queue the thread is released and prints the message to the serial port.

As it can be observed from Figure 2, the `Export Serial` block, which represents `commOutT`, receives three different messages - for Position, Speed and Key - from `Decode Input`. Each message is added to the queue once typed by the user. These messages serve as a feedback to acknowledge that the input parameters set by the user were successfully read. Moreover the Hash Rate and Nonce are sent from the `Mining` block, the first every 1s and the later once a successful hash is computed, to be printed to the serial port. Also every 1s the current Position and Velocity of the motor are sent from `Control Motor` to be printed as well.

Since the thread is activated once a new message is added, the initiation time of the task can be calculated as:

$$t_i = \frac{\#\,of\ bits\ of\ smallest\ message}{baud\ rate} = \frac{21\ characters * 8\ bits + 64\ bits}{9600\ bits/s} = 24.167\ ms \quad (4)$$

3.1.2 Incoming Communication

It is desired not only to sent messages to the host via the serial port but also to be able to act on commands given through the serial port. The second half of the communication therefore is performed by the `Incoming Communication` task. This task is responsible to accept commands from the host over the serial interface and act on them accordingly. The commands are consisted from the rotation command, the maximum speed command and the bitcoin key command. Since it is unknown when each command will be instructed hence the bytes of the serial data arrive asynchronously, they are handled by an interrupt, `Serial ISR`. The ISR is responsible to get the new character and place it into a queue for later decoding. The queue is created as a global instant of class `Queue`. Once a character is added in the queue it is passed for decoding in `Decode Input` block.

The block is essentially another thread called `decCmdT` which is attached to the ISR. The function of the thread is performed by `decCmdFn`. Each new character is placed at the end of an array called `newCmd` with size of $COMMAND\,LENGTH = 20$, in order to accommodate the longest possible command. An error message is added to notify the user if the command exceeds the preset command length. The carriage return character indicates the end of the command therefore the command now can be decoded.

The first character of the command is tested to determine which of the three commands was sent. Each command is then decoded according to the specifications. It is worth noting that the command for setting the bitcoin key is decoded using the `Mutex` and writing the value in a global volatile memory. If the command for setting the required number of rotations or velocity is decoded for the first time, the `motorCtrlT` thread is started. Each of the decoded command sends data to such us Key Value, Speed and Position to `Export Serial` block and the Key Value to `Mining` block.

Since the thread is activated once a new byte is added in the queue, the initiation time of the task can be calculated as:

$$t_i = \frac{\#\,of\ bits\,of\ one\,transmitted\,byte}{baud\,rate} = \frac{10\,bits}{9600\,bits/s} = 1.042\,ms \tag{5}$$

3.2 Motor Rotation

This task is responsible for keeping the motor rotating using interrupts in order to allow for further functionality in the system. As described in Section 2.1 in order for the motor to accelerate at a desired acceleration the field has to be advanced relative to the rotor state using PWM. Therefore the first part of the block is to initialise the PWM period as shown in `PWM initialisation`.

The rotor state is calculated using the state of the photointerrupts according to the position of the disc. The offset between the disc and the rotor however has to be first

calculated to be used later on in order to find the correct rotor state. Responsible for that is the `Calculate Motor Offset` block.

To keep the motor field updated relatively to the rotor position and hence keep the motor turning, an interrupt was created called `Motor ISR`. The interrupt was attached on the rising and falling edges of each photointerrupt, in total 6 interrupts, triggering every 60°. Every time there is a change in the state of one photointerrupt, the ISR routine checks for the state of the rotor position and advances the field accordingly. The `lead` variable is used in order to advance the field by 120° either in the positive or negative direction. Moreover the ISR keeps track of the position the motor covered by incrementing or decrementing the global `motorPosition` variable according to the direction of rotation.

As described in Section 2.1, the acceleration of the motor depends on the amount of torque created which is effectively controlled by the duty cycle of the PWM limiting the current in the motor outputs L1,L2,L3. The output of the `Control Motor` block which is effectively the output of the controller is passed to `Motor Out` in order to be used as the pulse width of the PWM simulating the torque created. Since the output of the controller can be positive or negative, extra care must be taken in order to convert it to the right value of the PWM width which can only be positive. Therefore if there is a negative value, then it must be made positive. To distinguish if the motor needs to accelerate or decelerate, the sign of the `lead` and the sign of the output of the controller are essentially compared. If they are in the same direction, then the motor needs to accelerate, so `lead` stays the same, otherwise it needs to decelerate and hence `lead` changes sign.

To calculate the initiation time of the ISR, the motor should be rotating at maximum speed.

$$\omega = \frac{2\pi}{T} = 160rps \Rightarrow T = 39.270ms \tag{6}$$

According to Equation 6, to complete a whole rotation 39.270ms are required. Each interrupt however is triggered every 60°. As a consequence to that, the initiation time of the interrupt is:

$$T_i = \frac{39.270ms}{6} = 6.545ms \tag{7}$$

3.3 Speed Control

The `Speed Control` task is effectively the speed and position controller described in Section 2.2. The task is performed by the `motorCtrlT` thread which is executed every 100ms. To ensure that the thread will be executed at the desired time and without performing costly executions, it is attached to a timer interrupt (`Ticker ISR`) that runs every 100ms. The thread runs a function which contains an infinite loop and waits for the ISR to set the thread signal to 1 in order to start execution.

The thread receives the Target Velocity and Position, set by the user, which are set as

global variables and were written by the `Decode Input` block. The functionality of the task is to measure the current velocity at which the motor rotates and how many rotations has covered. After that it is responsible to perform control to these quantities relatively to the desired ones. In other words, determines what the pulse width of the PWM should be. The result is written in a global variable called `motorPower` which is shared to the `Motor Rotation` task. The process by which the output of the controller is calculated is as explained in Section 2.2. Note that every 1s messages for current velocity and position are printed to the serial port via `CommOutT`. Since the execution of the thread is dictated by the ticker, the initiation time is 100ms.

3.4 Bitcoin Mining

The final task that was assigned was the bitcoin mining task. The `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.

A hash is the output of the hash function while the hash rate is the computation rate in hashes per second which is set to be printed every second. Note that bitcoin mining is running in the background, therefore this is the lowest priority task performed in the system. Since the system deals with interrupts and threads that are of higher priority, bitcoin mining, most of the times, will remain dead last to execute. It can be concluded that when threads and interrupts require maximum utilisation, the utilisation left for Bitcoin mining is the absolute minimum, thus resulting in minimum hash rate. Maximum utilization for threads and interrupts occurs when the motor is controlled at maximum velocity.

On the other hand the maximum hash rate is given by the number of hashes performed when the system runs only the `Mining` task i.e when the motor is off. The best and worst case hash rates will give an indication about the quality of the firmware therefore it is worth comparing between the two. To calculate the hash rate of the worst case, the motor is set to spin at maximum speed for 1s and the number of hashes are counted. The same process is repeated for the best case so the motor here stays off. The resulting hash rates are:

Case	Rate (Hashes/s)
Worst	118.7
Best	127.6

From the table above it can be observed that at the worst case the hash rate has decreased as expected. However it was decreased by 6.97% indicating good performance from the system.

4 Inter-Task Dependencies

One of the requirements of the project was to write a thread-safe firmware preventing the possibilities of race conditions and deadlocks. A deadlock will occur when threads communicate in a circular way. As there is no circular chain of thread deadlock does not occur.

While there are shared variables, the number of them was kept to minimum. The ones that exists however are all declared as variables of up to 32 bits, ensuring the accesses to them are atomic. This in turn ensures that they are executed as a single CPU instruction without any risk of interruption. There is only one shared variable which is 64 bits, and that is the Key value for the `Mining` task. To protect their shared data, a `Mutex` was implemented, ensuring only one thread can access the variable at a time. Moreover, all the shared variables are read only once in each thread and stored into a local variable.

Another precaution taken was the use of `Queue` and `Mail` for communicating data between the threads. `Mail` class particularly provides a FIFO queue allowing for concurrent threads to send data to the host independently without any losses. Moreover the `putMessage` function which effectively adds the messages in the queue, is a reentrant function, meaning that it can be executed more than once simultaneously without changing the result.