Department of Electronic and Electrical Engineering

Imperial College London

Embedded Systems Report - Motor Control

Author	*CID*
Pedro Monteiro	01209405
Alexander Jarisch	01246653
Ryan Mansoor	01224750
Joe Gardner	01232601

March 22, 2019

1 Introduction

This report details the control system of a brushless motor. Which performs multiple tasks with multiple threads such as position, velocity and SHA-256 computation.

2 Motor Control Algorithm

2.1 Code Initialisation

The code begins by defining all the constants for the photointerrupter and motor drive out pins. The period of the PWM, `PWMPERIOD` is defined to 2ms. The `MAXCOMMANDLENGTH` is defined which limits the maximum number of incoming serial characters to 18 per command. The `message_t` structure is defined for interpreting code messages. The photo interrupter inputs `InterruptIn` are assigned to input pins, the `PwmOut` and `DigitalOut` are initialised and assigned to output pins. These pins map to motor drive output pins previously defined. Global volatile variables `motorTorque,` `endVelocity`, and `endRotation` are initialised. The Mutex required to prevent deadlock is instantiated. The `Mail` instance `mail_box` and the `Queue` instance `inCharQ` are initialised. Following, serial connection through `RawSerial` is instantiated. The incoming and outgoing communication threads had `Thread` priorities set `osPriorityAboveNormal`. `motorCtrlT` had thread priority set to `osPriorityNormal`. Should further accuracy be required, the `motorCtrlT` thread should be given a higher priority. Lastly, all the functions used are declared.

2.2 Speed control

The control algorithm for the motor speed is implemented using a modified proportional control method. The implemented algorithm is shown below.

$$Ts = kps((endVelocity * 6) * 1.1 - |velocity|)$$

where Ts is the output torque, `kps` is the proportionality constant which we set to 36, `endVelocity` is the target speed and velocity is the measured speed. We multiply `endVelocity` by 6 since we have a count of 6 represent one revolution. Furthermore, we multiplied this term by 1.1 as this gave us more precise rotation. We decided on this value through trial and error. Ts is used in the code again to set `motorTorque` which is a global variable and sets the PWM of `L1L`, `L2L` and `L3L`. The torque is however limited by `PWMPERIOD` that we set to 2000 µs. Furthermore, the PWM needs to be positive so the magnitude of Ts is used. When Ts becomes negative we also change the sign of the lead. All this causes `Ts` to be rapidly swapped from negative to positive and causes the motor to slow down quickly. The velocity of the motor is calculated by `motorCtrlFn` every 100 ms by taking the position difference in the last 100 ms and multiplying it by 10 to get the number of rotations per second. We access the current location of the motor through a critical section which is the global variable `motorPosition`. This value gets set by `motorISR` and increases or decreases `motorPosition` by 6 for every full rotation.

2.3 Motor Position Control

For the motor position control we implemented the following equations in code.

$$Tr = kpr * rotationError + kdr * (rotationError - rotationErrorOld)$$

$$Ts = kps((endVelocity * 6) * 1.1 - |velocity|) * sgn(rotationError)$$

where

$$rotationError = endRotation * 0.98 - (locMotorPosition/6)$$

These equations essentially implement position control using a PD controller. `Tr` is the output torque from the rotational position controller, `kpr` is the proportional gain which we have set to 35 and `kdr` is the differential gain which we set to 16. We multiplied `endRotation` by 0.98 as this gave us better precision results. We experimentally came to this number through trial and error. It can be seen that we did not divide our rotation error by the time difference between both rotation readings. However, we judged it to not add much value as it would translate to multiplying by 10 since we would need to divide by 0.1s. This now gives rise to 2 possible torque's. To decide which one we should use for the final `motorTorque` we used the following logic:

$$motorTorque = \begin{cases} \text{min(Tr, Ts), velocity} >= 0 \\ \text{max(Tr, Ts), velocity} < 0 \\ \text{Ts, endRotation} == 0 \end{cases} \tag{1}$$

This last case was implemented for when we set `R0` so that the motor spins indefinitely. From 1 we can see that we always pick the most conservative of speeds meaning the speed that has the smallest magnitude. This will naturally slow down the speed once we approach our desired rotation number. Furthermore, the `motorISR` should change the position of the motor slightly if their is an overshoot so when the motor overshoots and goes over or under by 1/6 of a rotation.

3 Minimum Initiation Interval

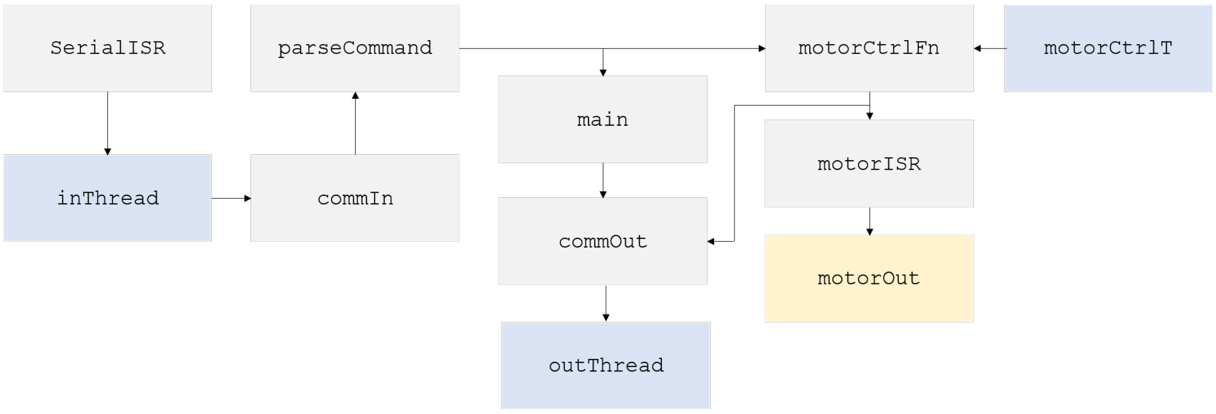
The minimum initiation intervals for the main loops of each thread are given below. These were calculated by adding a timer to each `while` loop to measure the minimum time taken to traverse the loop.

Table 1: An itemisation of all the tasks that are performed by the system

Task	Minimum Initiation Interval	Deadlines Initiation
`commOut`	1s	1s
`commIn`	1s	1s
`motorCtrl`	100ms	100ms
`MotorISR`	15 µs	15ms

4 Inter-task Dependencies

There are 3 threads: `outThread` prints messages to serial output; `inThread` which processes commands sent via serial input; `motorCtrlT` which controls the angular velocity of the motor. As well as these threads there is three `InterruptIn` instances which call the `motorISR` function and a `serialISR` function which is called on every `RawSerial::RxIrq` event. The inter-task dependencies can be divided into control dependencies and data dependencies. The control dependencies illustrated below do not experience deadlock as none of the functions downstream feedback into functions upstream.



The data dependencies are not the same as control dependencies, they involve the concurrent access of data by different functions or threads. `endRotation` and `endVelocity` are two global volatile variables that are not modified by the script itself but through incoming user inputs. The two variables are utilised in the code and can be defined globally for this reason.

A mutual exclusion (mutex) is used to protect newKey to prevent concurrent access by the `parseCommand` function and the bitcoin miner. This is because we want to ensure that the key used for bitcoin mining isn't changed while its being used by the miner. By locking and and unlocking newKey_mutex the variable becomes mutex protected. This means that the variable may only be accessed by one thread at a time. Both reading and writing of newKey are protected.

By setting variables to 32-bit or less, we can ensure atomic access. Atomic data types don't cause data races, and are utilised to synchronise memory access between different threads. Although variables in the code aren't defined as atomic, because they are 32-bit or less they can be executed in one instruction and are automatically accessed atomically.

In order to safely access some shared variables such as MotorPosition, interrupts are disabled to ensure that the reading of data is not interrupted partially through. This was done using the C++ critical section enter and exit functions.

The `Queue` class and the C++ `Mail` template have global instances instantiated, and are typically used for communication between threads. Thus communication and timing errors are not expected of these instances.

5 Timing Analysis

In order to determine the amount of CPU time that the motor tick interval provides for other processes, the timings of motor and serial print functions were summed during the tick interval. It can be seen that the processes add up to 2.371ms out of the 100ms available. This means that alongside these processes the system has upwards of 97% of its CPU time free for bitcoin mining.

6 Quantification of maximum and average CPU utilisation

We have experimentally concluded that we have a maximum rotation speed of around 108 rotations per second and an average speed of 60 rotations per second. Finding the execution time can be found by dividing one by the number of rotations multiplied by 6 since this is the number of times the `motorISR` gets called. Furthermore, the CPU utilisation can be found from dividing initiation interval by execution time.

Table 2: An itemisation of all the tasks that are performed by the system with maximum rotation speed

Task	Initiation interval	Execution time	CPU utilisation
`motorISR`	24.5 µs	1.54 ms	1.64%
`motorCtrlFn`	15.5 µs	1.54 ms	1.00%
`commOut`	5ms	-	-

The total CPU utilisation is approximately 2.64% at high rotations speeds.

Table 3: An itemisation of all the tasks that are performed by the system with average rotation speed

Task	Initiation interval	Execution time	CPU utilisation
`motorISR`	24.5 µs	2.77 ms	0.88%
`motorCtrlFn`	15.5 µs	2.77 ms	0.56%
`commOut`	5ms	-	-

The total CPU utilisation is approximately 1.44% at average rotation speeds.
The hash rates were 5278 at idle state and 5102 at full speed.