Department of Electronic and Electrical Engineering

Imperial College London

Brushless Motor Control: Embedded Systems Coursework 2

Authors	*CID*
Belen Barbed	01098555
Mwanakombo Hussein	00936105
Zoë Slattery	01056822

March 23, 2018

Contents

1 Motor Control Algorithm Description

1.1 Velocity Control

In order to control motor velocity a proportional speed controller was used. This is an improvement over general on-off control, which involves providing either full or no power to the motor. This can result in poor control and large speed variations, and is also inefficient. The proportional control system method uses linear feedback to provide an error signal which is proportional to the difference between the measured motor velocity and the desired velocity, as provided by the input command. This, unlike on-off control, will cause slight reductions or increases in power provided to the motor due to changes in error being gradual, which results in smoother control and smaller overshoot.
In the case when a command is parsed starting with 'V', the speed controller output, which determines the speed at which the motor is driven, is given by

$$y_s = k_p(s - |v|) \tag{1}$$

where

- y_s = speed controller output
- k_p = experimental constant
- s = maximum speed, determined by 'V' command
- v = measured velocity

The speed controller output is used to calculate the torque of the motor, denoted in code by `motorPower`, which is converted to a PWM value using the command `pulsewidth_us`. This ensures the period of the PWM wave supplied to the motor driver is constant, but the duty cycle will vary, whilst remaining at below 50%. The value k_p, denoted in code by `k_p_v` must be chosen experimentally, given that a low value will decrease the torque value for a given error, which can result in steady state error, but a high value will increase the torque value for a given error, which can cause fluctuations and overshoot. The chosen value was 100.

1.2 Position Control

In order to control motor position, such that the motor is able to spin for a defined number of rotations before stopping, a proportional-derivative (PD) controller was used. This is an improvement over the original position control used for implementing velocity control, as the differential term will slow the motor as it approaches the required position. This reduces the risk of overshoot and allows finer control to reach the desired position.
In the case when a command is parsed starting with 'R', the expression for motor power is given by

$$y_r = k_p E_r + k_d \frac{dE_r}{dt} \tag{2}$$

where

- y_r = speed controller output

– k_p = experimental constant

– k_d = experimental constant

– E_r = position error

When the motor is far from the correct position the initial, non-derivative term, denoted in code as `E_r`, will dominate so the torque output will be high and the motor will rotate quickly toward the final position. However, as it approaches the correct position the derivative term, denoted in code as `Er_d`, will start to dominate, meaning that the motor slows down due to reduced torque, reducing the likelihood of overshoot. The values of k_p, denoted in code as `k_p_p`, and of k_d, denoted in code as `k_d_p` were chosen as 30 and 20 respectively. The value k_p amplifies the position error, which causes the torque output to be higher for a longer period as the motor approaches its final position, so must be low enough so as to not cause oscillation. The value k_d acts as a damping factor, which if too small will cause overshoot, but if too large will cause the motor movement to be slow.

1.3 Combining Velocity and Position Control

In order to combine velocity and position control, the following method of selecting the `motorPower` variable was implemented:

$$\begin{aligned} \texttt{motorPower} = & max(\texttt{motorPower\_v}, \texttt{motorPower\_p}), v < 0 \\ & min(\texttt{motorPower\_v}, \texttt{motorPower\_p}), v >= 0 \end{aligned} \tag{3}$$

In Equation 3, the following variables are used:

– `motorPower_v` - the calculated torque value to provide the motor based on maximum velocity.

– `motorPower_p` - the calculated torque value to provide the motor based on the number of rotations requested.

In choosing the minimum value of `motorPower` possible, this algorithm ensures that at far distances from the destination the motor will spin quickly, yet as the motor approaches its final destination it will spin at a slower speed.

1.4 Benchmark Values Achieved

The final benchmarks achieved for the algorithms used were as follows:

- R0 allows for infinite rotations.
- V0 allows for maximum rotational velocity.
- Accuracy exists down to a value of 10 rotations per second, attributed to the trade-off in choosing values of `k_p_v`.

2 Optimisations

2.1 PID Control

As an alternative method to the proportional-derivative control method used in velocity control, a proportional-integral-derivative control method was implemented, using the equation:

$$y_r = k_p E_r + k_d \frac{dE_r}{dt} + k_i I \tag{4}$$

where

- y_r = speed controller output
- k_p = experimental constant
- k_d = experimental constant
- k_i = experimental constant
- E_r = position error
- I = integral, cumulative error

This was in order to eliminate steady state error by cumulatively summing it then adding this to the torque equation. In reality, it was found that introducing the integral mode had a negative effect on the overall stability of the system and increased oscillation, so was not included in the final version.

2.2 Derivative Smoothing in Proportional-Differential Control

To prevent large changes in derivative value which could cause jumps in motor speed, and to ensure smooth running of the motor, the variable `Er_d` was calculated using:

$$\texttt{Er\_d} = 0.9 * \texttt{Er\_d} + 0.1 * \texttt{Er\_d\_old} \tag{5}$$

This ensured that the motor did not undergo sudden changes in velocity so the initial and final rotations were smoother.

2.3 Linear Offset in Velocity Control

The data below demonstrates the deviation between input velocity and actual motor speed which was recorded at intervals of 10 rotations per second. To correct for this difference, a best-fit linear graph of the input variable `maxspeed` against the observed deviation was plotted. The variable `diff` was then set to be a function of the input `maxspeed` as shown in Equation 6.

$$\texttt{diff} = (1.0502) * \texttt{maxSpeed} + 0.62; \tag{6}$$

This variable is then used in the proportional controller to make the controller more sensitive to very high as well as very low `maxspeed` inputs, which in turn ensures the motor angular velocity is accurate to one rotation per second.
Table 2 demonstrates the values of input speed and the difference between this and measured velocity used to create the graph shown in Figure 2.1

Table 1: Linear Regression of Input Maxspeed and Actual Motor Velocity

Speed	Min Output	Max Output	Average	Difference
0	0	0	0	0
5	1	5	3	2
10	6	10	8	2
20	16	20	18	2
30	26	30	28	2
40	36	40	38	2
50	45	50	47.5	2.5
60	55	58	56.5	3.5
70	65	68	66.5	3.5
80	75	78	76.5	3.5
90	83	86	84.5	5.5
100	93	96	94.5	5.5
110	103	106	104.5	5.5
120	111	115	113	7
130	120	123	121.5	8.5

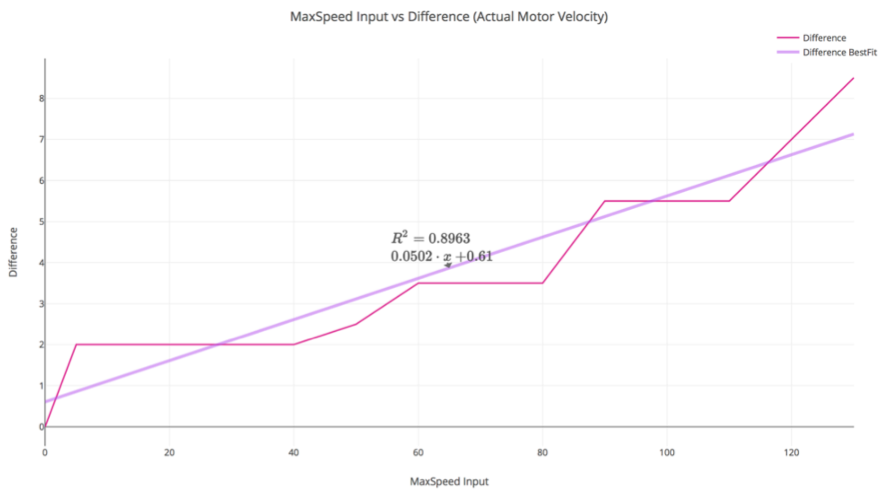

Figure 2.1: Linear Regression of Velocity Deviation

3 Application of real-time programming techniques

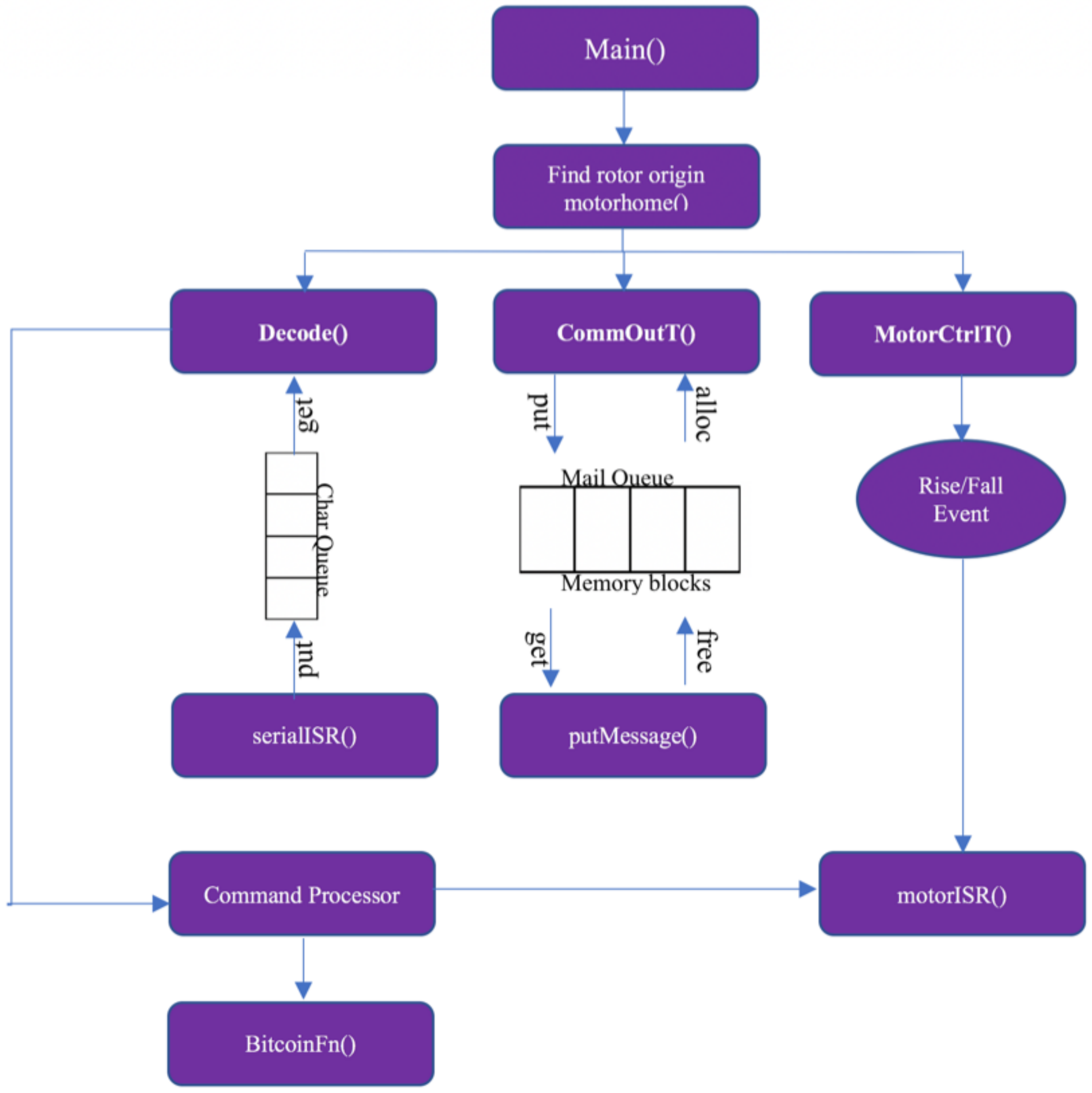

Figure 3.1: Code Flowchart

Figure 3.1 above diagrammatically depicts the codeflow of our threads, interrupts and task functions representing their relationships and dependencies as well as classes utilised such as `Queue` and `Mail`. The 3 threads : `Decode()`, `CommOutT()`, `MotorCtrlT()` although coded sequentially, they are depicted above in parallel as the threads run concurrently.

The tasks to be performed upon certain commands sent by the serial port are as follows:

1. Set the Bitcoin mining key to a specified value
2. Set the motor to a specified angular velocity
3. Set the motor to rotate for a specified number of rotations

3.1 Interrupts

The serial port receives bytes of serial data asynchronously so they are best handled by an interrupt. The ISR should be as short as possible so the serial data received can be placed into the queue `inCharQ` to be later decoded by the `DecodeFn` thread, as shown in the code section below.

```
//SerialISR - retrieves a byte from the serial port(pc)
//and places it on the queue
void serialISR(){
    uint8_t newChar = pc.getc();
    inCharQ.put((void*)newChar);
}
```

As per the specification, keeping track of motor position is best carried out in the photointerruptor ISR since any alternative would require a high-priority thread to count rotation events and the overhead of this would be high.

The Ticker class is used to set up a timer interrupt. The function `motorCtrlTick()` will be an interrupt service routine triggered by the Ticker. Once the timer expires, the timer interrupt will send a signal to the thread informing it to start running the task in `MotorCtrlFn()`. Note that `motorCtrlTick()` does not do any computation itself to avoid blocking the CPU and simply sends a signal back to the motor control thread.

```
void motorCtrlFn() {
    // execute every 100ms (0.1s)
    Ticker motorCtrlTicker;
    motorCtrlTicker.attach_us(&motorCtrlTick, 100000);
    //....
}
```

Initialization of threads:
There are 3 threads that perform distinct tasks and although `MotorCtrlT` should have a higher priority, when the thread was initialized with a higher priority than the other 2 threads, the code performed sub-optimally. Therefore, the initialized threads have the same level of priority `osPriorityNormal, 1024` that ensured the code performed more efficiently.

```
Thread MotorCtrlT(osPriorityNormal, 1024);
Thread CommOutT(osPriorityNormal, 1024);
Thread Decode(osPriorityNormal, 1024);
```

4 Inter-task dependencies

Thread for decoding commands
When a thread is created it is allocated an independent stack in a block of memory assigned from the heap. It is important to ensure there is enough stack space to store all the local variables required at worst-case scenario of function calls.
`Mutex` is used on the following variables: `noRotations`, `maxSpeed` and `newKey`, because these variables are shared resources between different threads and `mutex` synchronizes the execution of threads.

As there are only 2 threads which share resources it was decided not to implement a semaphore, which can be useful to manage thread access to a pool of shared resources of a certain type as seen below. By utilizing `mutex` appropriately, it is ensured that there is no possibility of deadlock. If a variable is shared between threads and can be accessed in an atomic transcation, for example `motorPosition`, it does not need to have a corresponding `mutex` variable as it will be updated in a single cycle due to being of type `int32_t`. If a semaphore were to have been implemented, it would have implementation similar to that shown in Figure 4.1.

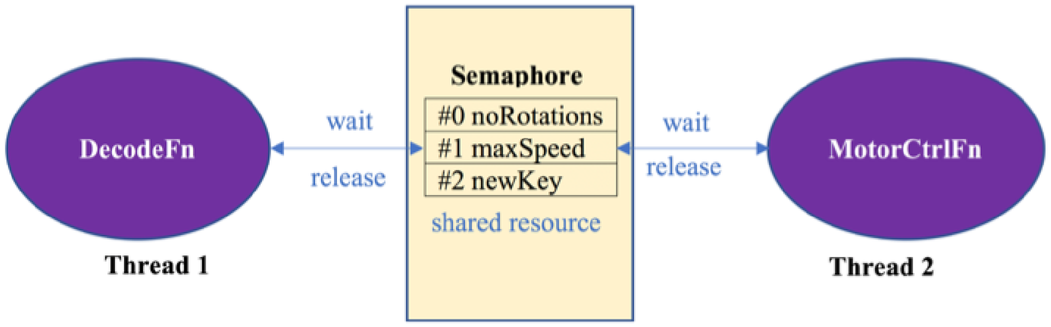

Figure 4.1: Semaphore Implementation

5 Bitcoin Hashing Rate

To determine code efficiency, the number of bitcoin sequences tried per minute was measured to the nearest 10,000 and the results provided in the table below. Computing SHA-256 hashes is computationally expensive, so a higher number received indicates a greater level of code efficiency, as more time is devoted to this task. At different motor velocities, the recorded attempted hashes varied slightly due to the increased volume of calls to `motorISR()`, which occur six times per motor rotation.

Table 2: Number of Sequences Attempted Per Minute

Velocity	Number of Sequences Attempted Per Minute		
	1	2	3
No Rotations	450,000	460,000	460,000
Maximum Rotational Velocity	430,000	440,000	440,000