Imperial College
London

COURSEWORK

IMPERIAL COLLEGE LONDON

EEE DEPARTMENT

Embedded Motor Controller

Authors:
Douglas Brion, Thomas Nugent, George Punter, Agni Iyer

Date: March 23, 2018

1 Motor Control Algorithm

We found that the basic proportional control to be insufficient for a high performing system. Instead, we implemented a PID loop for the proportional control as follows:

$$E_i \mathrel{+}= E_r \times T$$

$$y_r = K_p E_r + K_i E_i + K_d \left(\frac{E_r}{T} \right),$$

where E_r is the positional error and K_p, K_i, K_d are constants. T is the time passed since the last iteration of the loop, which was implemented with a **Timer** to get higher accuracy. The control loop applies continuous PID loop control theory to the discrete system we have. The K_p term is what increases the output to reach the target value, a higher value of K_p means we will get there faster, but also risk overshooting and oscillating. K_i affects the steady state error. K_d helps reduce overshooting by decreasing the output as we get closer to the target. We first adjusted K_p, K_i and K_d at according to the Ziegler-Nichols tuning rules heuristic and then, empirically, finding the best results that were: $K_p = 58$, $K_i = 2$, $K_d = 36$.

In order to rapidly adjust the constants during tuning to save compiling for each update we allowed K_p, K_i and K_d to be changed whilst running in our **decodeCommands** function with the letters *'q'*, *'w'* and *'e'* respectively.

For velocity control we found proportional control to be sufficient. The control system is as follows:

$$y_s = K_p(s - |v|)sgn(E_r)$$

Target Rotations	Actual Rotations
100	105
90	95
80	84
70	75
60	63
50	54
40	43
30	33
20	23
10	12
5	6
3	3
2	2

Table 1: Motor Control Performance

In adjusting the K_p for velocity control, empirically, we found that $K_p = 85$ gave the best results. The performance for our position control, specifying the number of rotations with a max velocity of 20 are shown in table 1, and the error, which increases as the target rotations increases, in figure 1.

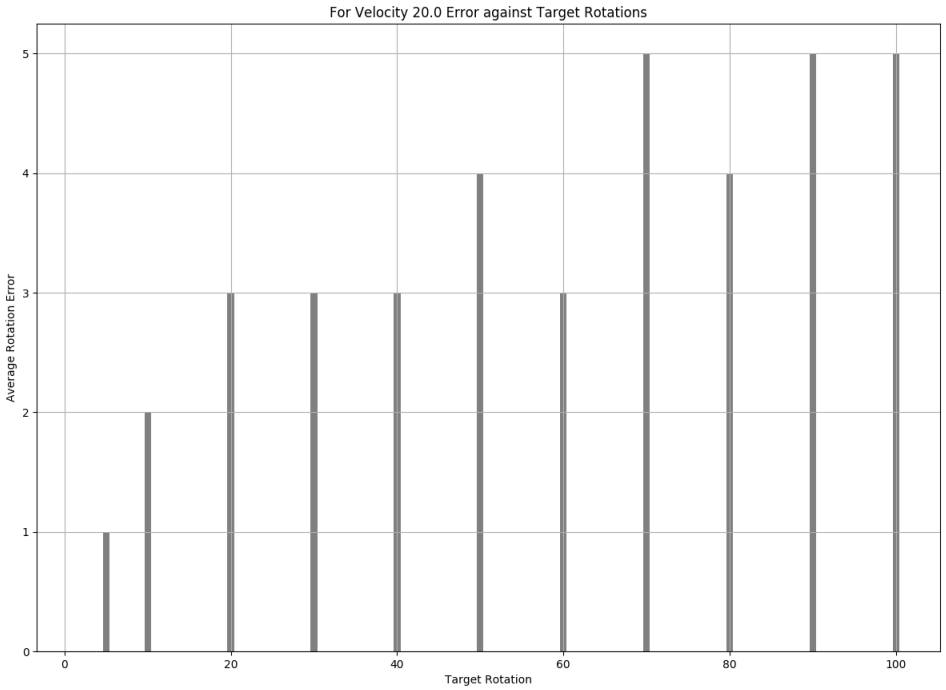


Figure 1: Actual Rotation Error against No. Target Rotations, with max velocity = 20

2 Tasks Performed by the System

The term *minimum initiation interval* is interpreted as the time taken for one iteration of each of the threads described below. We timed this by starting a **Timer** at the beginning of execution and printing out the time value at the end of execution, whilst deactivating all the other threads besides the **serialPrint** thread which was necessary to print out the times recorded.

2.1 serialPrint

This thread pulls a message off the **Mail** queue and decodes it to print to the serial monitor. In order to make this process easier, we created two enums for the types of messages we can receive. In enum **printCodes** are the possible types of message we might want to print. The most general type is **MSG** which will print a standard message with no data payload, or data that is globally available. The other print codes are for having a specific data payload, such as the nonce just found.

The second enum **msg_types** contains the specific MSG types available, such as telling the user an incorrect order has been issued, or reporting current status of velocity, rotations, position .etc.

This thread has a reasonably strict deadline, but is not the most important thread. If messages are printed out of sync with the system, the system will continue to function since this is not crucial to performance of the motor, but it will affect the usage of the motor from a human perspective. The minimum initiation interval for this thread is 40.0ms, which is quite high, most likely due to the `pc.printf` functions and accessing the messages from the **Mail** queue.

2.2 decodeCommands

This thread reads `chars` from the `char` buffer and processes them as commands using `sscanf`'s regular expression functionality to match the characters exactly and extract the required values, if the command is in an incorrect format it prints an error message instead. In order to get the desired regular expression we had to add two unsigned integers either side of the decimal point for setting velocity and the number of rotations. Here is the regular expression for setting rotations: `"%*[rR]%c%3u.%2u"`. A function was written to then convert these integers into a single floating point number. The thread waits until a carriage return character appears before decoding the last command pushed onto the char buffer. It then sets the necessary parameters depending on what the command was for.

If this thread is not completed quickly, the performance of the system is not critically affected, but control of the motor, by us, will be delayed. If this is delayed only by a short while then there are no problems, but if this is delayed longer than 20 seconds

or so then the system becomes fairly unusable. The minimum initiation intervals for this thread are documented in table 2.

Command Type	Time (ms)
Invalid Command	0.020
Change Key	0.055
Set Rotation	1.07
Set Velocity	32.1

Table 2: Minimum initiation intervals for decodeCommands

2.3 velocityCalc

This thread measures the current velocity and calculates the control loop results. As mentioned in section 1, we used a **Timer** in this thread to get better accuracy with our calculations.

This thread is triggered approximately every 100 ms by a **Ticker** that calls a function that signals to **velocityCalc** to do one iteration of calculations. This is the minimum initiation interval, but the actual interval may be larger due to other interrupts or threads taking over.

This thread has a particularly strict deadline since we need fast calculations to keep the motor control accurate. If this is not completed in time, we can get unstable and unreliable performance from the motor, hence this thread has the highest priority except the interrupts, **osPriorityHigh**. In fact, changing a number of numeric types from `float` to `integer` drastically improved the performance of our motor controller. The minimum initiation interval for this thread is 0.036 ms.

2.4 SHA256 Hashing

The final task performed by the system is the SHA256 hashing algorithm. This is the task with least priority since it is not essential to the running of the system. This runs in the main loop and puts a message on the queue when a nonce is successfully found. The hash rate, while spinning at full speed, averages around 6350/s and whilst idle is 6975/s. At different motor velocities the hashing rate changes due to the increased demand of the **updateMotor** function.

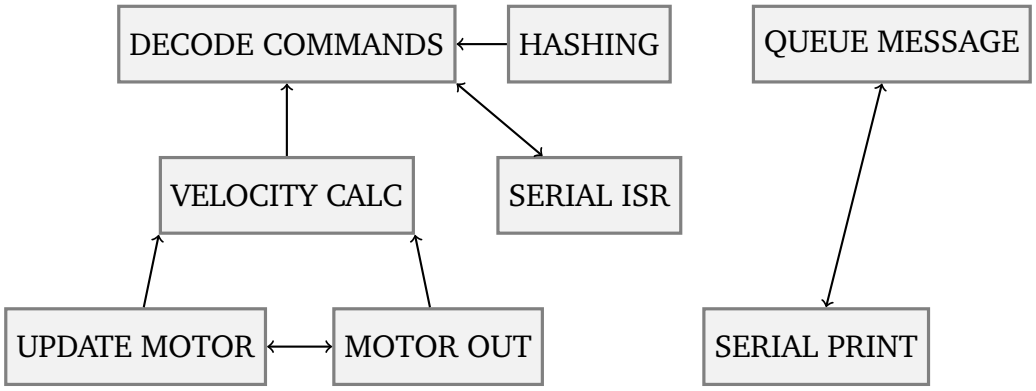
2.5 Motor Functions

motorOut and **updateMotor** are both for control of the motor. **motorOut** does the actual controlling of the motor by turning on the right coils to turn the motor, and setting the PWM duty cycle according to the torque calculated in **velocityCalc**. **updateMotor** is attached to the photo-interrupters, updates the motor position and calls **motorOut** to move the motor to the next state. Since these are both triggered by

the photo-interrupters, their initiation time is dependent on the speed that the motor is running. These tasks are extremely important, since if they are not completed in time the motor will not run properly.

3 Inter-task Dependencies

Below is a dependency graph showing the different operations happening. We notice that there are no cycles in the graph so deadlocks will never occur.



3.1 Shared data

There are a number of variables which are shared across the various threads. The **serialPrint** thread has access to **cmd_torque**, **velocity**, **motor_position**, **target_rotations** and **rotations**, however since we only perform reads on these variables, and they are all 32 bit and can be accessed with one memory access, no concurrency protection is needed in this case.
For reading the motor position in the **velocityCalc** thread, since it is updated by the photointerrupter routine, it is necessary to disable interrupts while copying this value into a local variable to ensure it is not being edited at that point, since the velocity calculations are critical to the system and need to be correct. Setting **cmd_torque** later on according to these calculations does not need protection since it is an atomic instruction.
Similarly, setting the max velocity and number of rotations in **decodeCommands** does not need concurrency protection since it's atomic. We have a mutex for the setting a new key for the SHA256 hashing algorithm since the key variable is a long int (64 bit), which requires two memory accesses to update. The mutex is locked, the variable updated, and then the mutex can be unlocked.