EE3-24: Embedded Systems
Coursework 2 - Brushless Motor Controller

Group #Kanye2020

Mikhail Demtchenko
Eliot Makabu
Rajan Patel
Patrick Reich

Table of Contents

Overview

Using a NUCLEO-F303K8 controlling a brushless motor, we can investigate real time control algorithms and their efficiency. A Bitcoin miner has been implemented as a proof of work , such that the hashrate of the system can be used to quantitatively evaluate performance. The overall implementation of the code uses a total of three threads for incoming communication, outgoing communication and motor control. There is an ISR for reading from the serial port and and an ISR for the photointerrupters.

Components and Constraints

- **Nucleo-F303K8:** The microcontroller used, 74MHz clock speed max, 64KB Flash, 16KB StaticRAM
- **Brushless Motor:** Brushless motors use pulses of current through motor windings to control the torque and velocity of the motor. To control the torque we modulate the motor current using PWM.

Description of the Motor Control Algorithm

The motor runs through a series of interrupts I1, I2, I3 that attach to the rising and falling edges of the hardware photointerrupters. These update the current position of the motor for the interrupt sequence routine, which is constantly updated in an endless while loop in the motor control thread. The values that are read from the photointerrupters are then used to count rotations for calculating the velocity and position of the motor. The current velocity is calculated by taking the difference of the current and previous motor position and multiplying by 10, since the thread interval is 0.1 and 10 iterations takes one second. We also had a timer implementation that works, this has been commented out in the code. The current position is saved in a local variable currpositon = motorposition and then printed to serial. The targetVelocity and targetRotations are set by the user via serial port. The choice of either the P and PD controllers determine the value of the torque, which is saved to a local variable torque and thus m_torque is changed only once in the thread m_torque = torque. m_torque can also be inputted on the serial port but is overridden if new commands for rotation come in. The choice of which controller to use is described below and the code has been well commented in general to explain what is going on. The way in which the motor control function interacts with the rest of the program is highlighted in the flowchart.

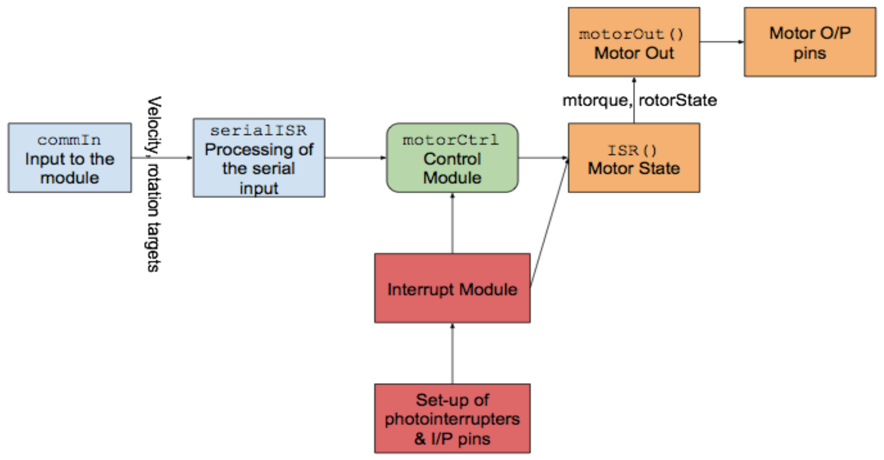


Figure 1. Flowchart showing the motor control algorithm

To choose between the value of torque calculated by the proportion controller and the proportional derivative controller we use the criterion below

$$y = \begin{cases} \max(ys, yr), & v < 0 \\ \min(ys, yr), & v \geq 0 \end{cases}$$

where ys, yr are specified as follows:

$$\begin{aligned} y_r &= k_{p1}E_r + k_d\frac{dE_r}{dt} \\ y_s &= k_{p2}(s - |v|)sgn(E_r) \end{aligned}$$

yr is the torque calculated from the position of the motor and the rate of change of its error (hence the need for a derivative term) and ys is the torque calculated from the speed i.e. velocity of the motor.

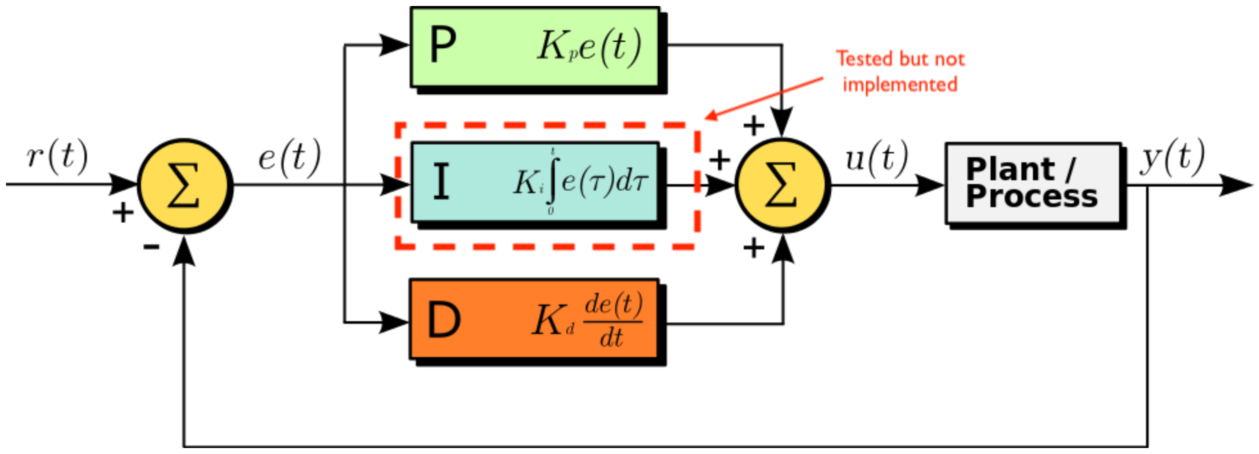
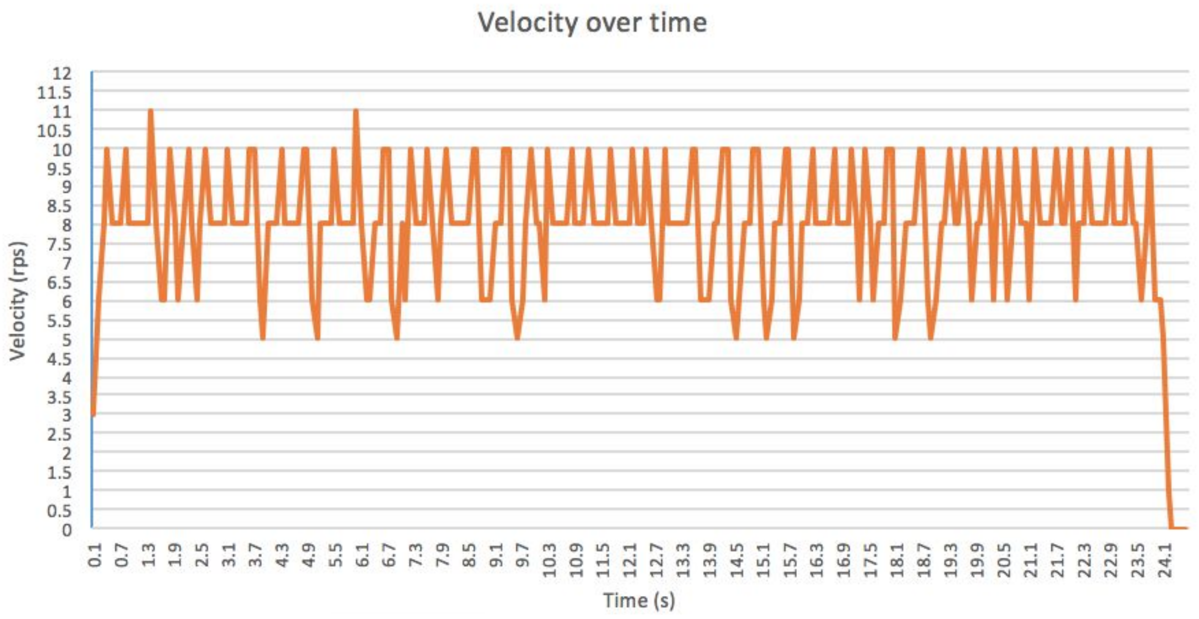


Figure 2. Diagram of a PID controller(Arturo Urquizo, 2011, http://commons.wikimedia.org/wiki/File:PID.svg)

The values of kp1, kp2 and kd were determined heuristically. A kp1 value of 20 and kp2 value of 20 was set to avoid oscillation. A kd value of 10 controlled the system damping relatively well, if it is too small, the motor will overshoot, too large and the

movement will take longer than necessary. An integral controller was considered to track and account for steady state error, how it caused a lot more overshoot and so we chose to omit it.
Another thread, commInT(), is used to take input commands from the user that are preemptively decoded from a command queue to then run the required task (for example bitcoin mining). The thread commOutT(), constantly outputs the hashing rate, velocity and current position. When the hashing algorithm is initialized, upon detecting a correct value, it outputs a 'nonce' and it's value. The graph below shows some fluctuation at the target velocity, as well as a few overshoots above 10. The time taken to get up to speed and back down to zero is relatively quite responsive.



Task Analysis

Originally for each thread the default size of the threads was 2048B but this needed to be reduced due to memory constraints and the program freezing. The heap size for CommInT and CommOutT was then changed to 1024B, and motorCtrlT to 1536B.

Loop	Thread	Execution Time	Stack size (Bytes)	Priority level	Task Deadlines
Motor Control	MotorCtrlT	10ms	1536	High	Every 0.1s
Serial ISR Reader	CommInT	960b/s -> 1.04ms/b 10 bits per char 10.4ms/b	1024	Normal	102.5ms to ensure motorctrl commands are implemented

Bitcoin Miner	CommOutT	133μs	1024	Normal	10.4ms if a new bitcoin func is called, otherwise 0.1 seconds

Task Deadline Analysis:

- The deadline for the motor control is quantified in the specification as the interval, 0.1 seconds. The motor controller thread doesn't execute any sleep functions which could end up missing the deadline, and the execution is well within this deadline so it doesn't run the risk of overrunning.
- The serial ISR reader is very slow and could potentially run the risk of overrunning a task deadline, but as the execution time of the motorctrl is 102.5ms, there is plenty of time for a new command to be inputted and parsed forwards to the control before it gets cleared and overwritten.
- The bitcoin miner is potentially calling 15000 functions a second just from the implementation of mutex to lock and unlock the key, but as this is then parsed to serial ISR, which has a very slow execution, there is no possibility of overrunning the task deadline. It is in fact not very important to quantify the deadline because it essentially runs as an idle task, filling the space up to the deadlines.

Hardware utilisation:

- With an average hash rate of 7500Hz, and a max clock speed of 74MHz, at max speed the main loop code is executing once approximately every 10,000 clock cycles.
- The code uses 56.4/64kB of available flash and 6.1/16kB of available RAM.

Inter-Task Dependencies Analysis

There are two main dependencies in this system, control and data dependencies. The control dependencies will not cause deadlock as there are no functions down the function train that feed back into earlier function, as shown below.

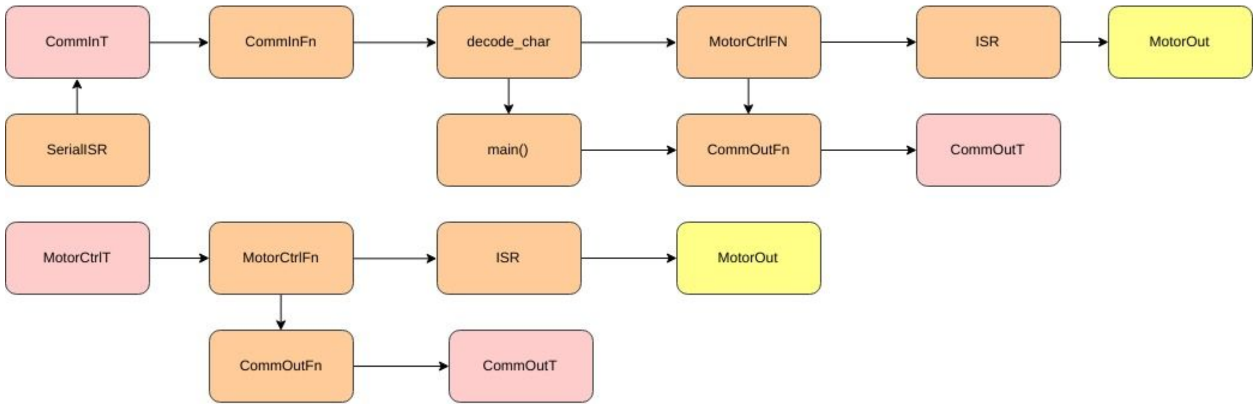
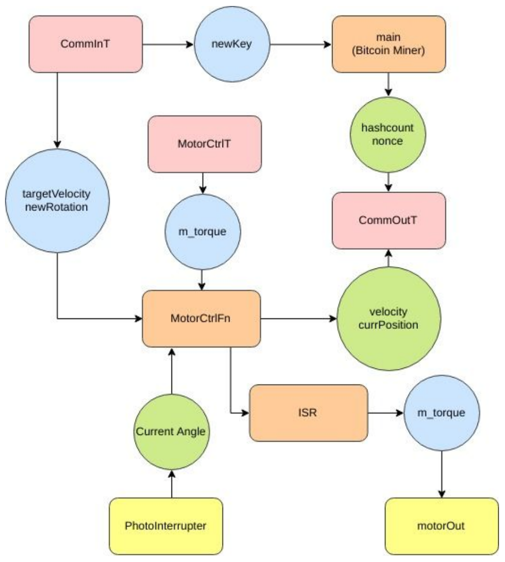


Figure 3. Flowchart of control dependencies



Data dependencies on the other hand aren't about deadlock, but about the simultaneous access of variables. targetVelocity is initialized as a global volatile 32-bit integer and targetRotation is initialized as a global volatile float. The values of these variables can only be changed by user input from the serial input and are not changed anywhere else in the code, the values are read of course. Mutex is used to protect newKey as it will prevent simultaneous access of newKey by the command decoder and the bitcoin miner. The reading of newKey has also been protected with the mutex in the same way as the writing.

Figure 4. Flowchart of data dependencies

m_torque (motor torque) cannot be protected by   a mutex because you cannot use a mutex in the ISR where the variable will be read. However, if the variable is type int32_t we can ensure that variable access will be atomic, thus they take place in a single CPU instruction and cannot be interrupted part way through. Ideally, you would check the assembly output or write the accesses with inline assembler instructions to guarantee this behaviour.

Global variable access

To ensure that we access our global variables at timings that will not create errors we had to protect them in different ways:

Mutex

To make sure that the key we use for our Bitcoin mining doesn't change while it is in use, we lock and unlock newKey_mutex using mutual exclusion (mutex). A mutex protected variable can only be accessed by one thread at a time, meaning that any thread that wants to alter newKey_mutex while another one is using it will have to wait until the initial thread isn't using it anymore.

Atomic accesses

Atomic types can be used to synchronise memory accesses among different threads since they don't cause data races. Even though the variables in our code are not explicitly defined as atomic variables, the way they will be accessed will be atomic. To ensure atomic access, variables should be set as 32-bit or less, since 64 bit variables will require 2 instructions.

Global instances

A global instance of class 'Queue' was defined in our code to queue data from the serial ISR function to the CommInFn function. This type of variable is made to communicate data in between threads and therefore prevents any communication or timing errors. We can be confident that we won't have any of these errors with the C++ Template 'Mail' either for the same reason.

Bitcoin Mining

To determine how much CPU time can be devoted to the main loop with our motor control and I/O communications threads running, we're running a Bitcoin mining algorithm in the background. This kind of algorithm requires a lot of work or iterations in order to find a solution('nonce') and is therefore a good way to determine how much CPU time remains after executing the motor control tasks. It operates by taking in a key that is initialised to zero in our code but can also be changed manually via serial input. The algorithm then uses this key and a predefined 256-bit sequence to run a built-in 'computeHash' function that tries to find a solution. If the computed Hash is a solution, we use our Output Communication thread to print out that a new nonce was found and its value.

Bitcoin Hashing Rate: How it works and performance evaluation.

Hash is the mathematical problem the miner's computer needs to solve, essentially turning the key on a combination lock until it unlocks. SHA256 is just a specific way of hashing. The Hash Rate is the rate at which these problems are being solved. We can use this hashrate to quantitatively evaluate performance of a system.

Nonce Calculation: Taking the average hash rate and dividing by 2^16

- While the motor control algorithm is running we obtain 0.114 n/s, or 1 'nonce' every 8.74s

Bitcoin Mining Algorithm:

Hashcount:

- Uses a ticker to run do_hashcount function every second, which counts the number of hashes computed, uses putMessage to print the hash frequency, then resets the hash count to 0
- We can take the hash frequency directly as hash rate

mutex newKey_mutex:

- The mutex class is a synchronization primitive that can be used to protect shared data from being simultaneously accessed by multiple threads.
- Used to stop the value being changed in use. Locks the value of key, writes to a new variable called *key, then unlocks the value. This reinforces deadlock protection in the hashing algorithm.
- newKey is also assigned to volatile to allow the value to be changed via serial port during incoming communication thread calls.

Key Generation:

- Handled simply by inputting the key register given and appending a 64-bit sequence with the sequence given.

decode_char:

- Scans the input key for the 'k%10llx' (which is stored in the buffer) and then stores the value in newKey, which then bounces back to main to dereference and change the *key value.. Scanf takes care of the regex format, there is no need for a dedicated parser as this would use to much flash memory.

Main Loop

(a) First a ticker 'hashcounter' is defined, which runs do_hashcount, outputting the hash rate every second.

(b) Then it enters a while loop. It looks for new key, if it finds it, assigns it to the *key variable, then computes hash and performs hash count. If it finds a nonce, print 'nonce' and nonce found'. It then increments the nonce by one at every iteration, and adds one to the hashcount variable in the the loop irrelevant of if a nonce was found.