Embedded Systems

Coursework 2 Report

Libang Liang	CID 01204497
Xinyuan Xu	CID 01183830
Zengrui Huang	CID 01204572

1. Motor Control Algorithms

main()

At the start of main(), output setting of the PWM pins is first defined: period is set to 2ms and duty cycle is set to 100%. PWM output controls the speed of the motor. Doing this setup at the very beginning is necessary because the next step is calling the motorHome() function, which needs the motor to be in full power and in stationary, otherwise the initial state might not be what we expect, position and speed may not be correctly measured as well.

Next in main, the interrupt service routine ISR() is attached to the rising and falling edge of the 3 photo-sensor output. This enables us to detect the change of state of the motor, or in other word the movement of the motor.

ISR()

This interrupt service routine is probably better called as "MotorISR". It first reads the current motor state, then calculate the new output state and calls motorOut() to realize the output. ISR also keeps track of the motor position, increment or decrement according to in which direction motor is turning.

motorOut()

This function is the interface to the motor hardware. This should be the only function that changes the output of any pins connected to the motor. First output byte is calculated from drive table. Secondly, speed of the motor is controlled by changing the pulse width (effectively duty cycle) of the PWM pin output. Finally, switches are turned on or off, depending on output drive state, to change motor magnetic field direction.

Controlling speed

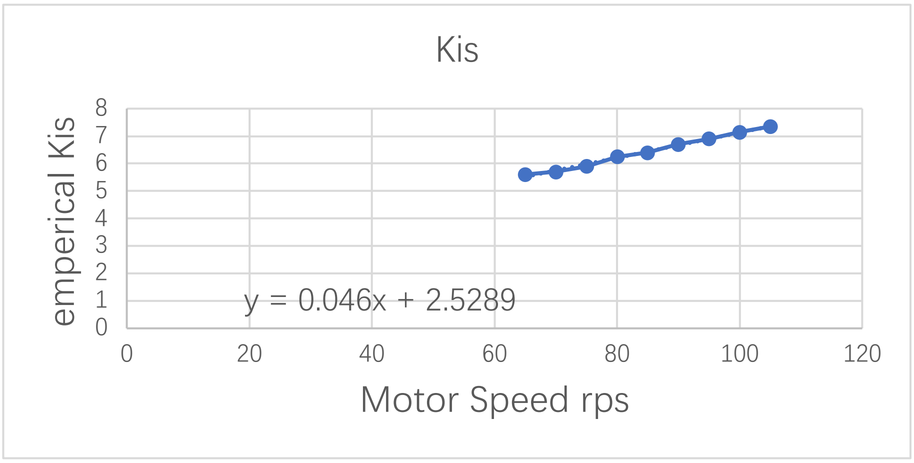
$$T_s = (k_{ps}e_s + k_{is}\int e_s dt)sgn(E_r) \qquad e_s = s_{max} - \left|\frac{de_r}{dt}\right|$$

Where e_s is speed error and the two ‘k’s are constants. Ts is the variable for maximum speed control. In function MotorControlFunction(), part of the code is responsible for controlling the speed. When there is a command that asks the motor to move, Ts needs to accelerate the motor to target maximum speed swiftly and maintain the speed at a stable and accurate level.

$k_{ps}e_s$ is the proportional term. It makes sense that if there is a bigger speed error, we need to apply higher torque to accelerate it. If only this term is used, there would be a steady state error on speed control. For example, in our own case, velocity would be around 41rps when target speed is 50rps.

To solve this problem, we first introduced an integral term $k_{is}\int e_s dt$, making the controller a proportional and integral controller. This integral term has been very effective. For speed lower than 60rps, there is almost no speed error; and if occasionally the speed drops a bit, it would be recovered immediately in the next reading.

The performance for fixed k_{is} at speed higher than 60rps is not enough – steady state error could be bigger than 1prs, for example, command “V105” would have a steady state speed of around 93rps. It has been guessed that this is because at higher speed, friction increased significantly, so we decided to model k_{is} dynamically.



We also tried to reduce the period of MotorControlTicker, which currently invokes MotorControlFunction() in average every 100ms. It was hoped that by updating the torque output more frequently, target speed could be better tracked by actual speed. However, this showed little improvement in our experiment.

It has been noticed that speed control has some difficulty at low speed, for example if a maximum speed of 5rps has been set. The motor might decelerate too much to reach this speed and never being able to recover again due to the “stickiness” or moment of inertia. When the motor stops, a little nudge would help to trigger ISR() again and help the motor to keep running.

The final bit of the equation is $sgn(E_r)$. This is only used when speed control is combined with rotation control. $E_r = Target\ Position - Current\ Position$. If $E_r > 0$, then motor needs to keep spinning forward, thus torque needs to be positive and vice versa.

Controlling rotation

$$T_r = k_{pr}e_r + k_{dr}\frac{de_r}{dt}$$

Apart from speed, the other important part of motor control is rotation, or position.

The first part of control is also a proportional controller: $k_{pr}e_r$. If the motor is far from target, e_r would be big so that larger torque could be applied to speed motor towards target.

The second part of control is a derivative controller: $k_{dr}\frac{de_r}{dt}$. This could be viewed as the break of the motor. At the start of the rotation, magnitude of $\frac{de_r}{dt}$ or speed is low; this is analogous to not using the break. Near the end of rotation, magnitude of the proportional term decreases significantly, and the derivative term would dominate. This is hitting the break when target is almost reached.

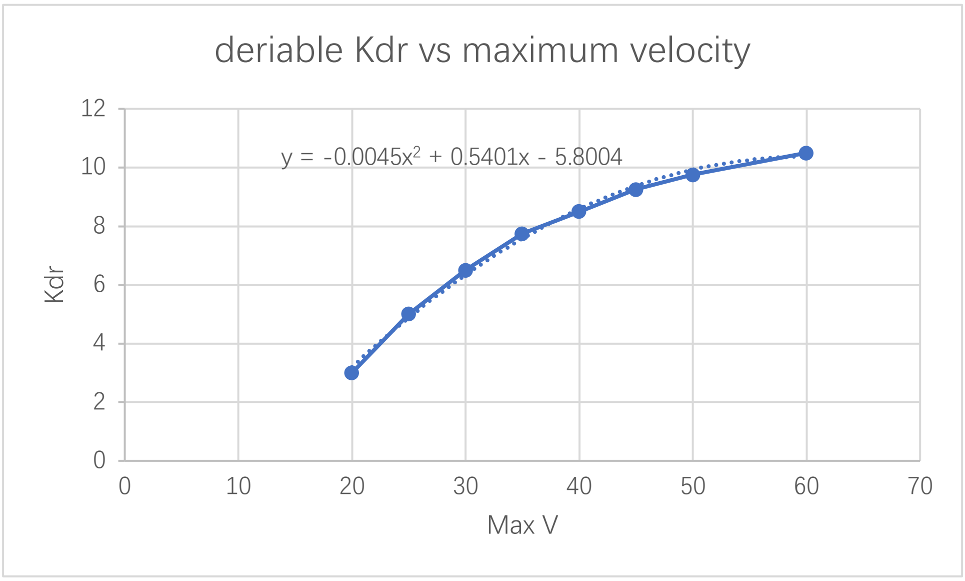


Figure 1 Variable k_{dr}

When testing different values of k_{dr}, it was found that for different maximum velocity, different k_{dr} needs to be used not to overshoot. This is analogous to driving a car as well – when a car needs to be stopped, break needs to be hit harder if it is at a higher speed. So suitable values of k_{dr} were tested and found for different velocity, and the final model is a quadratic relationship. In this way, the k_{dr} in our code is no longer a constant, but a dynamic value fitting the maximum speed.

Potential direction of improvement includes enhancing the performance of rotation control at higher maximum speed but for a small number of rotations, for example using a R40 command at V50. Currently, it is not guaranteed that motor will not overshoot or decelerate too early. At a lower speed, for example V20, a command of low number of revolutions is usually quite precise, at most 3 positions in error or less than half rotation.

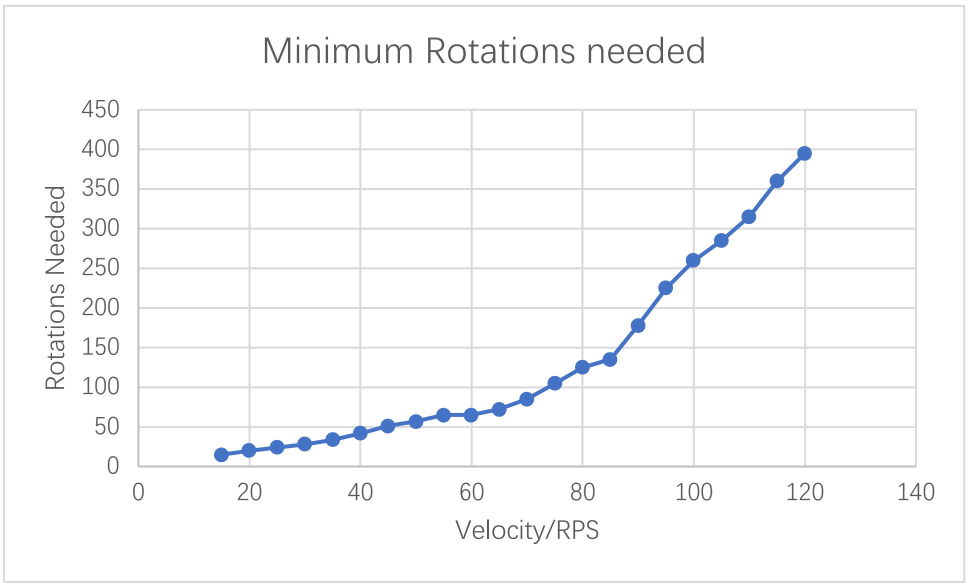


Figure 2 Minimum Rotations Needed

Figure 2 shows that the minimum rotations needed for different pre-set maximum velocity, being tested empirically. For rotations under this minimum rotation, the error might be larger than one cycle. This graph illustrates that there is an exponential increase of minimum rotations as we try to increase maximum velocity. The reason for this is, for a larger maximum velocity, the time (also distance) needed for the motor to decelerate is larger.

Combining speed and rotation control

In motor, the magnitude of the physical torque that drives the motor is proportional to motor current, which could be controlled by PWM duty cycle. In order to control the duty cycle, a few more thing need to be done.

First, $T_r \ and \ T_s$ from the previous two sections need to be combined to get a final motor torque. This is done by choosing which is more “conservative”, because we don’t want the motor to spin faster than target maximum speed or do not slow down when rotation task is near its end.

Second, torque applied can be negative, because the motor needs to decelerate or spin in reverse direction, but duty cycle can’t be negative. This is corrected by making the motor lead negative and taking the absolute value of the torque. If torque is positive, then lead is also positive.

In the end, torque required by the control function could have a large magnitude, but PWM pulse width could not exceed its period. Therefore, torque is capped by the maximum possible pulse width. If magnitude of torque is smaller than the limit, then torque is assigned to pulse width.

Notice: please only input command after seeing “Current Motor State is x”. Only after this statement could the initialization process being viewed as complete. A command too early could distort this process and leads to measurement error, such as measure positive speed as negative.

2. An itemisation of all the tasks

Itemised Task	Minimum initiation intervals estimated	Maximum execution time measured
Send Thread	0.2s	90.24 ms
Receive Thread	0.1s	2.09 μs
MotorControlThread	0.1s	26.6 μs
MotorOut	1ms	22.5 μs
ISR (motor, not Serial)	1ms	23.5 μs
Bitcoin Mining	N\A	160.7 μs

For Send Thread, currently one line of velocity and one line of position are reported every second. Assuming a spare 3 more lines of output to deal with other serial output, like feedback to command R100 given, minimum initiation interval estimated is 0.2s. Assuming one could press the keyboard at most 10 times per second, then the average minimum initiation interval could be estimated as 0.1s.

120rps * 6 state per round = 720 states/s this is the maximum number of times in 1s ISR could be invoked in average. 1s/720 = 1.389ms, we can overestimate by assuming the minimum to become 1ms. MotorOut is called by ISR, so they have the same frequency, then the same minimum initiation interval.

It is noticed that send thread takes particularly long time compared to other tasks, excluding bitcoin mining. This could be explained by the limitation of serial communication baud rate. The baud rate used is 9600 bits/s; a sentence of maximum 30 characters and each character of 8bits is assumed; this means without any communication overhead, it would take 25ms to output the whole sentence. So, the observed 90ms should not be a surprise in the end.

When measuring maximum execution time with oscilloscope, care has been taken for motor related tasks. Different speeds have been tested to see if there's a difference in time taken. The result is that motor speeds don't affect these tasks. This is expected, as motor speed would change initiation interval, but couldn't change the code execution.

Figures below shows the scope images when task latency is measured.

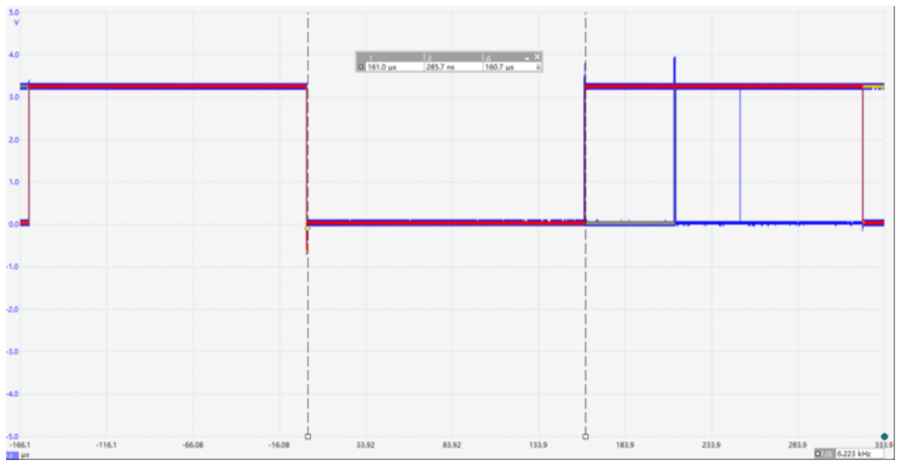


Figure 3 Bitcoin

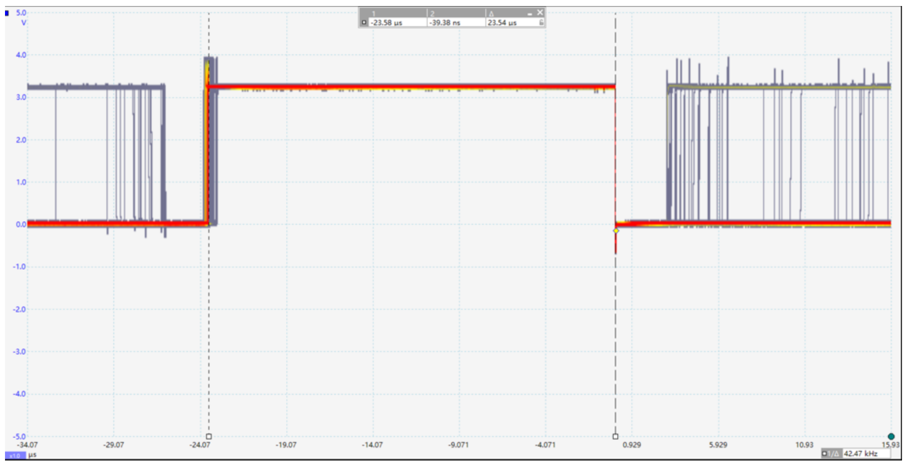


Figure 4 maxspeedISR

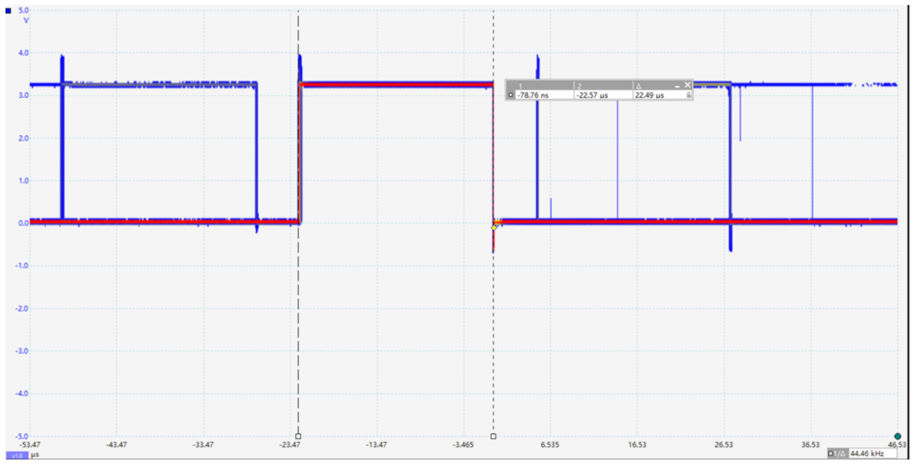


Figure 5 maxspeedMotorOut

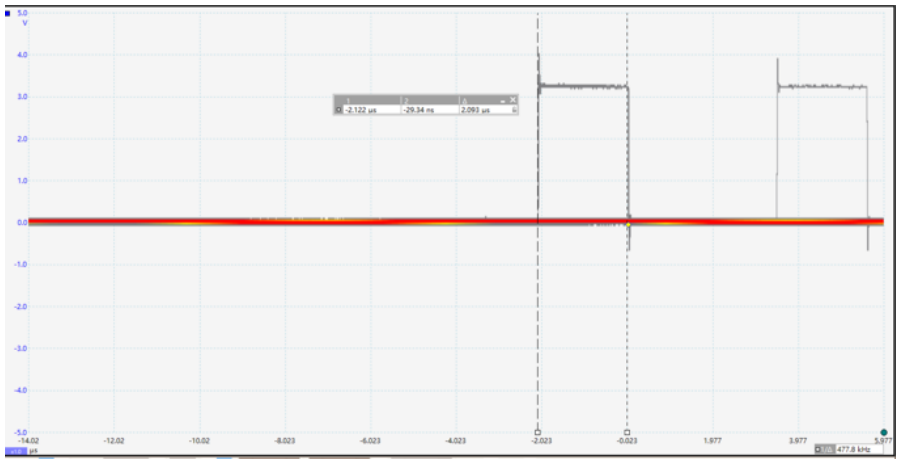


Figure 6 RecieveThread

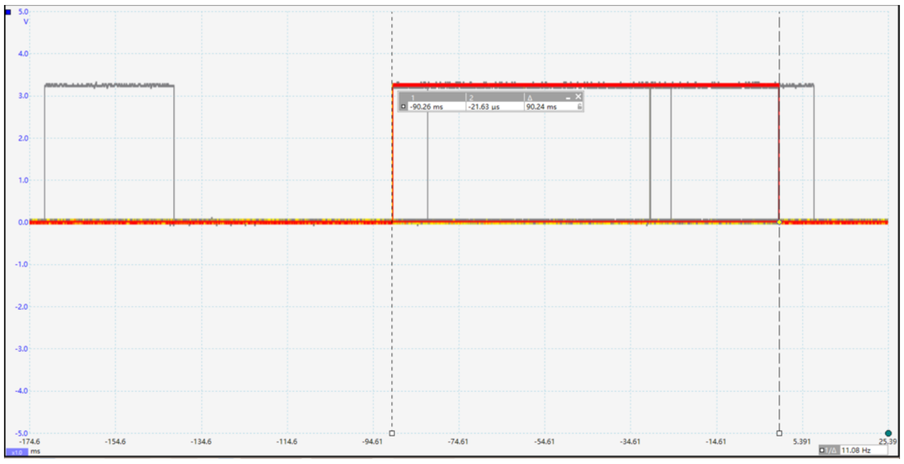


Figure 7 sendThread

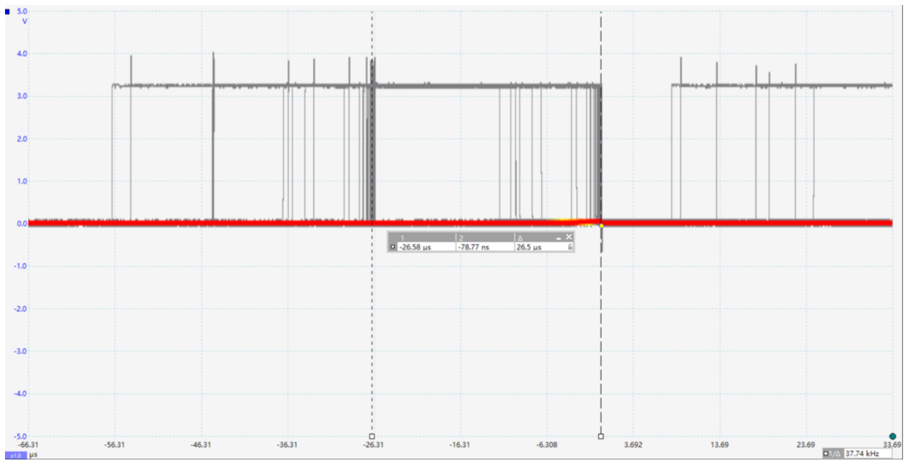


Figure 8 V30MotorControl

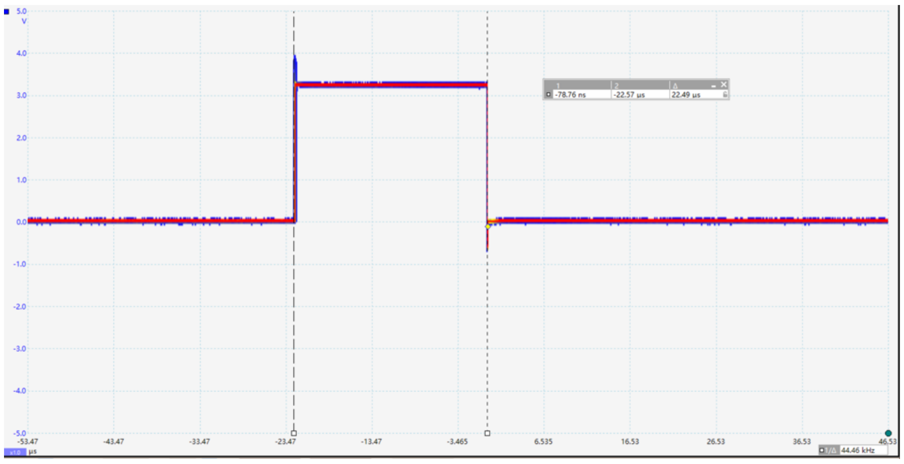


Figure 9 V30motorOut

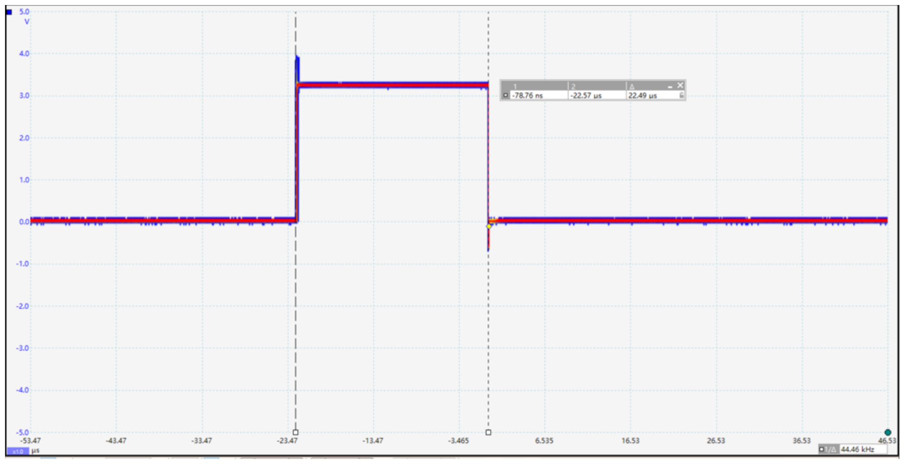


Figure 10 V50motorOut

3. An analysis of inter-task dependencies to show that there is no possibility of deadlock.

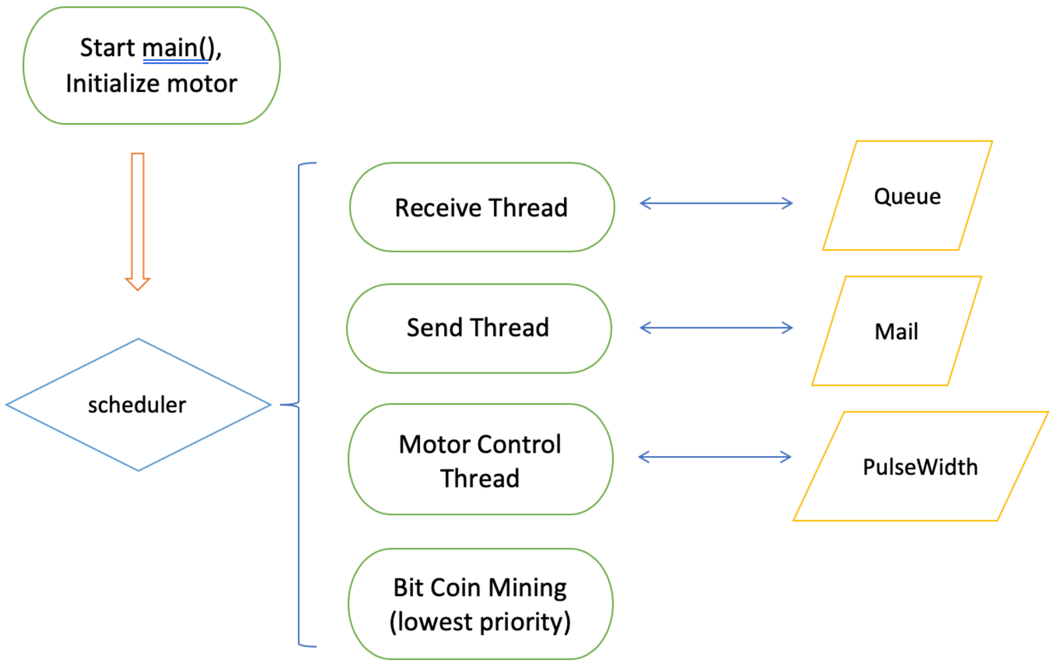
Deadlock is a situation where a set of processes are blocked because each process is holding a resource and waiting for another resource acquired by some other process.

Except variable “newKey”, all other global variables use less than or equal to 32 bits. This means that they are atomic, microprocessor instructions cannot be interrupted when they are accessed. So, for these variables shared by threads, like “PulseWidth”, they are actually thread safe. Therefore, the only mutex protected variable is “newKey”.

For variable “newKey”, it is access at two places. First, it is accessed when calculating bit coin. Second, it is accessed when user input new key. In both cases, only one line of code is involved, and those two lines does not depend on other things at all. So, this mutex protection would cause a deadlock.

The other kind of shared resource apart from global variables is input/output handler, particularly for serial communication and motor interface in our case. Queue has been used to handle incoming communication, as income messages are broken down into characters. Mail, similar to queue, has been used to store outgoing messages. Output statement “pc.printf()” only occurs in Send Thread. The use of mail prevents the contention of using printf simultaneously by different threads. With Mail and Queue, concurrent access to serial communication resources are avoided. For motor interface, function motorOut() is the only one that could directly affect motor. motorOut() is only called by ISR()[1], so no other thread can contend direct motor control.

There would be no deadlock for our code because there is no contention for shared resources and there are no mutual dependencies described above in our functions.



[1] There is one exception at the beginning of main(), where motorOut() is called by motorhome(). But this only happens once.

4. A critical instant analysis of the rate monotonic scheduler

For rate monotonic scheduler, there are several assumptions:

- Single CPU
- Tasks have fixed execution times
- Fixed initiation interval for each task, which is also its deadline
- No dependencies, no switching overheads
- Fixed task priority

And shortest initiation interval gets highest priority.

Task	τ_i	T_i	Number of repeat
Send Thread	0.2s	90.24 ms	1
Receive Thread	0.1s	2.09 μs	2
MotorControlThread	0.1s	26.6 μs	2
MotorOut	1ms	22.5 μs	200
ISR (motor, not Serial)	1ms	23.5 μs	200

Table 1 Real time system analysis

Critical Instance analysis:

τ_i *is the mimimum initiation interval*, T_i *is time taken.*

In 0.2s, number of repeats for initiation of each task is calculated.

$\sum T_i N_i = 0.09949738$ s, which is less than half of the 0.2s.

Thus, under the worst-case condition, all deadlines could be met.

5. A quantification of maximum and average CPU utilisation, excluding bitcoin mining.

According to the previous section, in the worst-case condition, it would take the processor nearly 0.1s to complete all the tasks needed in a period of 0.2s. Thus, the maximum CPU utilisation would be 50%.

Task	Estimated average initialization interval	T_i	Number of repeat N_i
Send Thread	0.4s	90.24 ms	1
Receive Thread	0.2s	2.09 μs	2
MotorControlThread	0.1s	26.6 μs	4
MotorOut	4ms	22.5 μs	100
ISR (motor, not Serial)	4ms	23.5 μs	100

Send Thread average initialization interval is estimated as 0.4s, because at least 2 lines would be printed each second reporting speed and position, and we need to take other lines such as feedback to other input command into account.

It is hard to estimate average initiation interval for receive thread, because it is hard to imagine how often we press the key board. Luckily, the time taken to complete this task is very little, so this problem would have little effect on our estimation.

If we assume average speed of rotation is 40 rounds per second:
40rps * 6 state per round = 240 states/s this is the average number of times in 1s ISR could be invoked in average. 1s/240 = 4.167ms, we can slightly overestimate by assuming the minimum to become 4ms. MotorOut() is called by ISR(), so they have the same frequency, then the same average initiation interval.

MotorControlThread is controlled by a ticker with period of 100ms, so is not affected anyway by this analysis.

$$\sum_{for\ all\ tasks} T_i\, N_i = 0.09495s$$

$$\mu(utilisation) = \frac{0.0945}{0.4} = 23.74\%$$

So, the estimated average CPU utilisation would be approximately 24%.