Embedded Systems on multi-threaded motor

Ziwei Chen
zc4417@ic.ac.uk

Zesen Yao
zy4417@ic.ac.uk

Alan Yilun Yuan
ayy17@ic.ac.uk

April 1, 2020

Abstract

This project focuses on a brush-less DC motor with 3 phase system and the aim is to implement precise control over this motor using an embedded system in an efficient and thread-safe environment. In this report, the motor controlling mechanism will be discussed in details. Additionally, there are a few threads performing different tasks simultaneously so each of them will be discussed individually.

Keywords: *Semaphores, Mutex, Multi-threading, Bitcoin Mining, SHA256, Brushless Motor Control, PID control*

1 Introduction

1.1 Background

The motor used in this project has 3 motor drivers (L1, L2, L3) adopting the efficient 3 phase mechanism. Each pair of pins will be assigned as LOW and HIGH to control the current flowing through. Therefore, 8 possible orientations can be obtained by 3 bits. Specifically, by changing the states, the magnetic field angle increases by (60 degrees * magnitude of variable called lead) and thus torque is generated. The misalignment between the rotor field and the generated filed will enforce the motor spinning. Moreover, the direction of spinning is controlled by the variable named *lead*. However, the challenge in this project is to achieve fine control over the velocity and rotation of rotor without extensive overshoot.

2 Multi-threaded tasks

The aim of this project is that the CPU can handle several tasks simultaneously with rational split of execution time.

2.1 Bitcoin Mining - *Compute*

The first task is to compute a hash that meets the requirement of bitcoin mining standards. SHA256 is a well-known algorithm to encrypt character sequences into a 256-character long sequence. The input character sequence consists of the following sections along with the size of memory occupied list as shown in Table 1.

As described in the table, the variable named *sequence* consists of various fields. The goal is to generate a 256 long sequence via SHA256 such that the first 2 bytes are zeros. However, the nature of this encryption algorithm is that it is not computationally symmetric. In other words, it is an irreversible function. Hence, the only way to find out a nonce

Data-name	Size of memory
sequence	48 Bytes
key	8 Bytes
nonce	8 Bytes

Table 1: Memory structure of ”sequence”

which satisfies the condition is brute-force, namely iterating through all possible combinations. The first 48 bytes of the sequence has been fixed in this project. The ”key” can be specified by the user via I/O on the terminal. The correct ”Nonce” values will be reported on terminal once a matched one is found. Further details on manipulating ”key” values will be discussed in sections below.

2.2 Serial Output - *Myprint*

Unlike single thread tasks, collisions are highly likely to occur when multiple tasks acquire the use of output streams. Therefore, a separate thread has been made to handle the use of output streams. A printing thread has been designed by attaching a self-defined printing function in the main function. In order to cope with different types of data, a data structure with the following member data is used: Data, type, key, velocity. However, output requests may happen when the thread is currently occupied. Therefore, a mail queue is adopted such that requests will be delayed until the port is ready. Once the function grabs a mail from the queue, the data would be processed with respect to the ”type” field accordingly. The use of this thread eases the difficulty of generalizing all data as types may vary and be hard to predict. The important part when using mailbox is the capacity defined. Large sizes of queue or data can cause system failures. Hence, it is essential to free the mailbox at the end of each iteration.

2.3 Decoding Input - *Decode*

As mentioned in section 2.1, inputs from users are required in this project. Various commands are defined to control and interact with the chip while it is executing the program. Fig 1 describes the regular expression for the allowed commands. **R** sets the number of rotations the motor should run. **V** sets the maximum speed. **T** sets the sequence notes and duration of the melody which the PWM pin can generate. Another aspect which needs to be carefully taken of is the input sequence of characters. The solution here is similar to writing out data which is discussed in the previous subsection. Another different self-defined function is attached to the serial port in this thread. Only one character is allowed to be read at a time. An input buffer named ”infobuffer” is defined to be a char array of size 128. The decoding function would read the first character as the nametag and decide which case it belongs to. Various operations are made to successfully modify the global control variables, i.e. *target_velocity* etc. It is worth mentioning that mutex is utilised here to prevents data-collisions. Since different threads might access the same global variable

Syntax (regex)	Function	Example
`R-?\d{1,3}(\.\d{1,2})?`	Rotate a number of revolutions	`R-5.5` – Run backwards for 5.5 rotations from current position
`V\d{1,3}(\.\d{1,3})?`	Set maximum speed	`V20` – Rotate at maximum 20 revolutions per second
`K[0-9a-f]{16}`	Set bitcoin key	`K18fe34d19f4543ef`
`T([A-G][#\^]?[1-8]){1,16}`	Set tune	`TA2D4A2F^8` – Sequence of 4 notes and durations

Figure 1: Description of commands

at the same time, locking and unlocking the variable are essential in assuring concurrency. A good example would be reading the key can not happen at the same time while writing in a new key.

2.4 Motor Controlling - *motorCtrlFn*

2.4.1 Position Control

Controlling the position of the motor can be achieved with Proportional-Differentiation (PD) control algorithms. Equation 1 illustrates the function used to calculate the motor power.

$$y = k_p * E_r + k_d * \frac{dE_r}{dt} \tag{1}$$

where E_r represents the error between the target position and the current position.

This function improves the transient response compared with the original proportional control algorithm and reduces the chance of overshooting. However, the constants need to be fine-tuned in various experiments. The table below indicates how k_{pr} is tested and the results are reflected as below. It is noted that k_{pr} and k_{dr} need to be set to minimum to avoid infinite oscillation and overshoot.

VCommand	RCommand	k_{pr}	k_{dr}	Error
V30	**R200**	0.006	0.02	0.33
V40	**R-1000**	0.006	0.02	0.17
V40	**R300**	0.006	0.02	0.5
V50	**R400**	0.006	0.02	0.16
V60	**R1000**	0.006	0.02	1.13

Table 2: Position control results

2.4.2 Velocity Control

Controlling the velocity of the motor can be achieved with Proportional-Integration (PI) control algorithms. Equation 2 illustrates the function used to calculate the motor power.

Similar logic can be seen here when compared with position control. Table 3 indicates how constants are fine tuned.

Vcommand	RCommand	k_{is}	k_{ps}	Error
V30	**R1000**	0.00095	0.05	0.18
V40	**R1000**	0.00095	0.05	0.37
V40	**R-1000**	0.00095	0.05	0.18
V50	**R1000**	0.00095	0.05	0.18
V60	**R1000**	0.00095	0.05	0.37

Table 3: Velocity control results

$$y = k_p * E_v + k_i * \int E_v dt \tag{2}$$

where E_v represents the error between the target velocity set by input and the current velocity. The obtained motor power would be written into PWM pins directly using the command *MotorPWM.write(motor-power)*.

2.4.3 Melody Tuning

Melody tuning can be seen as an independent section in the motor controlling function. Depending on the input read from the decoding thread, a variable named *TUNES* would store the pitch and duration of the melody. Once the readflag is asserted, the MotorPWM pin would be written with a specific pulsewidth corresponding to the input.

3 Thread Dependencies & Priorities

The dependency graph for all threads used is depicted in the Fig 2. It is worth mentioning that the thread *ComputeHash* is not separated from the main. However, other threads are initialised before the *while* loop in the *main* function. A few mutex are added before reading or modifying the variables to preserve data consistency. This is an essential procedure in when adopting multi-thread coding.

MotorISR is defined as an interrupt which has a higher priority than all other threads. Therefore, no dependencies exist between this function and other threads as they must stop whenever the photo-interrupt pins are toggled. The MotorCtrl thread is attached with a 100ms ticker and operates at normal priority. The Bitcoin mining thread has the lowest priority and will be executed when the CPU is polling. *Myprint* thread acts as an I/O buffer and handles all printing requests. A mailbox of size 16 is defined and works as a FIFO buffer. The frequency of requests must be handled with care as the buffer might be overwhelmed and memory failures may occur.

As for blocking dependencies, there are mainly two sources in this project: mutex and control flags. MotorCtrlFn() is known to handle the different input sent from serial ports, and 3 associated flags can cause blocking. If more than one flag is changed, the functions will handle all changes simultaneously. Other than flags, blocking can be caused by mutex

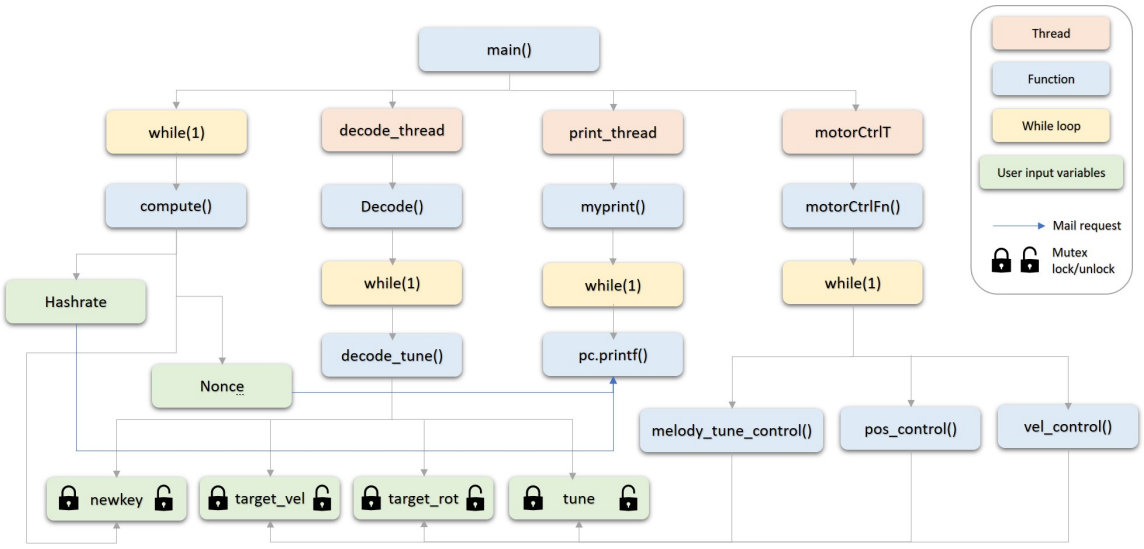

Figure 2: Thread dependency & Mutex blocking graph

which aims to protect the data concurrency when accessing global variables. When the global variable is locked by *Mutex::lock()*, other threads accessing this variable are forced to wait until it is released by *Mutex::unlock()*. Therefore, a minimum amount of blocking time caused by global variable access is possible due to mutex.

Name	Type	Blocks	Controlled by
vel_enter	velocity flag	motorCtrlFn()	Decode()
val_enter	rotation flag	motorCtrlFn()	Decode()
tune_enter	melody flag	motorCtrlFn()	decode_tune()
newkey_mutex	Mutex	main()	Decode()
newvel_mutex	Mutex	motorCtrlFn()	Decode()

Table 4: Flag blocking analysis

It should be noticed that deadlock is impossible in this project. In theory, deadlock exists when functions or threads mutually depend on each other, which means the blocking is indefinite. However, there is no pair of threads in this project which utilises mutex locks on mutually dependent global variables.

4 Measurements

Various measurements are taken and shown in Table 5.

4.1 Time Analysis

4.1.1 Maximum Execution time

The maximum execution time corresponds to the critical path for a specific thread to finish running one iteration.

Task Name	Max-Execution time/s	Min-Initiation time/s	Theoretical CPU Utilisation
ISR (CheckState)	0.000047	0.0049	0.95%
Motor Control	0.000025	0.1	0.025%
Melody	0.000025	0.125	0.04%
Compute Hash	0.72	1	72%

Table 5: Three different measurements

4.1.2 Initiation interval

The minimum initiation interval corresponds to the shortest time before it can accept inputs.

4.1.3 Maximum CPU Utilisation

CPU utilisation can be derived from the above subsections and expressed as

$$\eta = \frac{t}{T} * 100\% \tag{3}$$

where t and T denote the max-execution and min-initiation time respectively.

4.1.4 Timing Constraint and Critical Instant Analysis

The critical time is the time taken for the critical path. *ComputeHash* has an initiation interval of 1s and hence is used to validate the timing constraint. The interrupt service routine called by the motorISR can have a maximum of 6*100(max rotation speed) = 600/s. The default duration set for melody is 0.125s and hence 1/0.125 = 8 times are called per second. A ticker of 0.1s is attached to the motorCtrl thread and hence 1/0.1 =10 times are executed per second. Therefore, the critical path would be to take all events into account. The time consumed can be expressed as

$$t = 0.72 + 8 * 0.000025 + 10 * 0.000025 + 600 * 0.000047 = 0.74865\text{s} \tag{4}$$

Clearly, the critical time is less than the longest time of 1s. In other words, the timing requirements are met.