EE3-25 Embedded System

Coursework 2 Report

Brushless Motor Controller

Group Name: EAT RASPBERRYPI

Jie Yechen
jy317 01390269

Weiyue Leng
wl4817 01351038

Sheng Yu
sy5617 01413225

March 2020

Department of Electrical and Electronic Engineering
Imperial College London

1. Introduction

This project would mainly focus on controlling a brushless and synchronous motor which use non-contract methods to switch windings. The motor was based on an ARM microcontroller unit and a motor PCB.

By the end of this project, the motor was able to complete following tasks:

- The system could perform a Bitcoin mining task. It can test 5000 nonce per second and report matched nonce once it is detected;
- The motor could rotate within a given maximum velocity;
- The motor could rotate for a given number of cycles and stop without either overshooting or undershooting;
- The system would play a melody while the motor is rotating which is achieved by adjusting the control current of the motor;

This report would illustrate the algorithm implementation, blocking dependencies and performance analysis of the project.

2. Tasks and Dependency

In this section, each task embedded in this system will be introduced. The below figure illustrates the overall architecture of the system.

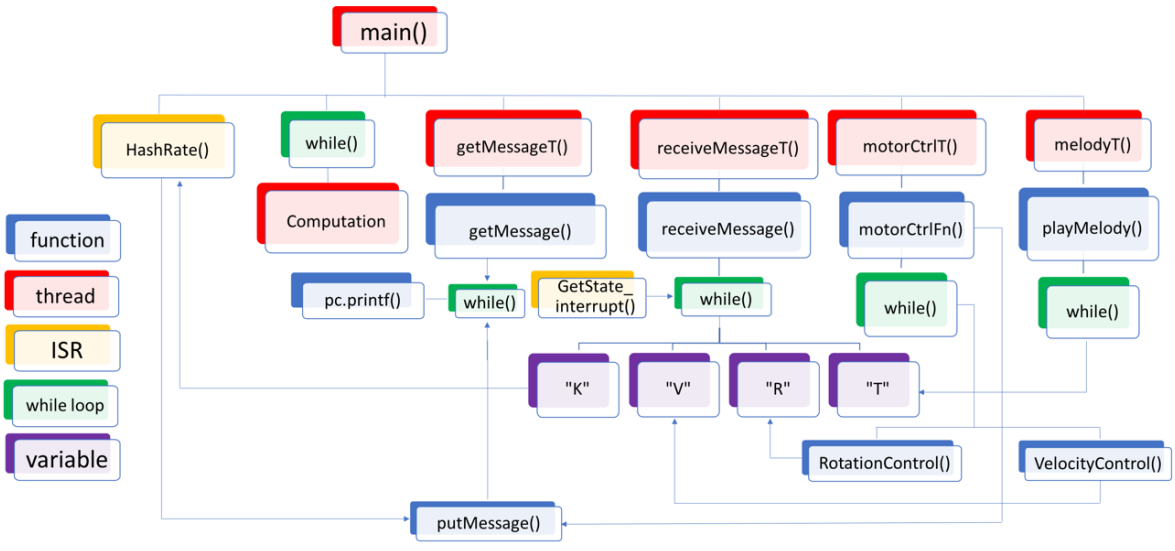

Figure 1. The block chart of the system

In order to achieve the design specification, parameters and threads are initialised in the ***main.cpp***. Threads are widely used to manage tasks which are processed at different running rates. It could allow higher priority tasks to take over the lower priority tasks rather than break them up.

```
11 Thread receiveMessage(osPriorityNormal,1024);
12 Thread getMessage(osPriorityNormal,1024);
13 Thread melody(osPriorityNormal,1024);
14 Thread motorCtrlT (osPriorityHigh,1024);
```

Code Snippet 1. Threads Declaration

2.1 getMessage

The function of this thread is to collect all message data from other parts of the system and sending the received information to the ISR. A data structure called ***Mail*** which contains a First-In-First-Out (FIFO) buffer to queue messages. A function called ***putMessage()*** is used by each thread to add messages to the queue.

```
58 void putMessage(uint8_t type, float variable, uint64_t variable_64){
59     mail_tc *mail = mail_box.alloc();
60     mail->code = type;
61     mail->data = variable;
62     mail->data_64 = variable_64;
63     mail_box.put(mail);
64 }
```

Code Snippet 2. ***putMessage()*** implementation

All the threads have access to the same mailbox in order to be able to print messages through the function ***getmsg()*** defined in Code Snippet 6. This function takes the messages from the queue and print the corresponding information by identifying their type. After this, the message is popped out of the queue to allow storage of new messages.

```
void getmsg(){
    while(1){
    osEvent evt = mail_box.get();
    if(evt.status == osEventMail){
        mail_tc *mail = (mail_tc*)evt.value.p;
        switch(mail->code){
            case(NONCE):
                pc.printf("Nonce is: 0x%llx\n\r",mail->data);
                pc.printf("N%016llX\n\r",mail->data_64);
                break;
            case(COUNT):
                pc.printf("Hash rate is: %d\n\r",mail->data_64);
                break;
            case(KEY):
                pc.printf("K%016llX\n\r",mail->data_64);
                break;
            case(MAX_VEL):
                pc.printf("Max Velocity is %f\n\r",mail->data);
                break;
            case(ROTATE):
                  pc.printf("Target Rotation is: %f\n\r",mail->data);
                break;
            case(ACT_VELOCITY):
                pc.printf("Actual velocity is %f , ", mail->data);
                break;
            case 8:
                pc.printf("Pos_err is %f\n\r", mail->data);
                break;
            case(ERROR):
                pc.printf("ERROR!!!");
                break;
            }
        mail_box.free(mail);
        }
    }

}
```

Code Snippet 3. ***getmsg()*** function Implementation

2.2 receiveMessage

The function of this thread is to decode input commands from the serial port. The message is obtained through an interrupt attached to the Raw PC, which is called every time a new character has been introduced. This character is then pushed into another mailbox, so that ***receivemsg()*** can access it. ***sscanf()*** function is used to scan all elements of the command message to extract required part. The quantitative specification followed by the capital letter is then stored in global variables and passed into the message queue. Each shared variable is protected by a mutex since it is used in two different threads.

```
if(command[i]=='\r'){
    wait_us(100);

    command[i+1] = '\0' ;
    switch(command[0]){
        case 'K':
            sscanf(command,"K%hx",&hex);
            receivedKey = (uint64_t)hex;
            newKey_mutex.lock();
            newKey = receivedKey;
            newKey_mutex.unlock();
            putMessage(KEY, 0, newKey);
            break;
        case 'V':
            velocityEnter=true;
            velocity_mutex.lock();
            sscanf(command,"V%f",&max_velocity);
            velocity_mutex.unlock();
            putMessage(MAX_VEL, max_velocity,0);
            break;
        case 'R':
            rotationEnter=true;
            rotation_mutex.lock();
            sscanf(command, "R%f", &max_rotation);
            rotation_mutex.unlock();
            putMessage(ROTATE, max_rotation,0);
            break;
        case 'T':
            sscanf(command,"T%s", tune);
            note_extraction();
            break;
        }
```

Code Snippet 4. ***receivemsg()*** function Implementation

Mutex Name	Blocked by	Released by
rotation_mutex	MotorCtrlFnT	receiveMessageT
rotation_mutex	receiveMessageT	MotorCtrlFnT
velocity_mutex	MotorCtrlFnT	receiveMessageT
velocity_mutex	receiveMessageT	MotorCtrlFnT
newKey_mutex	Main Thread	receiveMessageT
newKey_mutex	receiveMessageT	Main Thread
tune_mutex	MelodyT	receiveMessageT
tune_mutex	receiveMessageT	MelodyT

Table 1. Mutex Objects

2.3 Bitcoin

The bitcoin mining function computation runs in the main thread with normal priority. The nonce is printed on the serial port as 16 hexadecimal digits once it is found by the ***computeHash()*** function. Note that the value of the nonce depends on the key used, which can be modified from the serial port. Thus, a mutex object has been used to protect the key variable and avoid computing the nonce before the new key has been correctly stored.

The hash rate computes the number of hashes calculated per second. It is reported on the serial port once every second using ticker, which is an interrupt. In the specification, the hash rate is approximately 5000 hashes per second.

2.4 Motor Control

The motor control task is the main task of this project. It is managed by the ***motorCtrlT()*** thread, which should have the highest priority among all threads. There are other three functions, ***motorCtrlFn()***, ***VelocityControl()*** and ***RotationControl()*** within the thread to measure and control the real time behaviours. This thread is blocked and released by itself with a flag, which is set every 100ms.

2.5 Melody

The motor would play a melody while it is spinning by modulating its PWM period, which is determined by the input 'T' command. An array of different frequencies corresponding to each note is initialised. A switch statement is implemented to store the notes and durations into two separate arrays, which are used to set the period and hold times.

3. Motor Controller Implementation

An Interrupt Service Route (ISR) function is declared to control the motor consistently. The motor is controlled by setting the duty cycle, which is the ratio between the high output time and the whole period, on the PWM pin.

3.1 Velocity Control

In this function, we designed a proportional-integral (PI) control feedback system to control the real time velocity. The controller will avoid the velocity of the motor exceeding the maximum speed (***max_vel***) set by the user. If the maximum speed is not set manually, it would be initialized as 100 rotations per second.

The equation of the controller is given below:

$$\boldsymbol{y_s = (k_{ps} * speed_{err} + k_{is} * \int speed\_err\, dt)} \qquad \textbf{eq (1)}$$

where $\boldsymbol{k_{ps}}$ and $\boldsymbol{k_{is}}$ has been set experimentally to reduce oscillation and increase stability. ***Speed_*****err** is calculated by subtracting ***max_vel*** from current velocity. This will generally give a negative torque if the current velocity is lower than the maximum speed and vice versa. Therefore, if the $\boldsymbol{y_s}$ is negative we take the absolute value to

speed up and if y_s is positive we take 0 to stop the motor torque and slow down the speed.

The real time motor velocity controller is implemented by the function ***VelocityControl()*** which is displayed below by Code Snippet 5.

```
162 float VelocityControl(){
163 
164     sign = (velocity>=0)? 1 : -1;
165     speed_err = velocity*sign - max_vel;
166     integral_speed_err = integral_speed_err + speed_err/0.1;
167     if(integral_speed_err > 880){
168         integral_speed_err = 880;
169     }
170     if(integral_speed_err < -880){
171         integral_speed_err = -880;
172     }
173 
174     ys = kps*(speed_err)+kis*integral_speed_err;
175     ys = (ys<0) ? -ys:0;
176     ys = (ys > maxPWM) ? maxPWM : ys;
177     return ys;
178 }
```

Code Snippet 5. ***VelocityControl()*** function

3.2 Position Control

In this section, the position controller is designed with a proportional-differential (PD) control feedback system instead of a PI controller. Since the motor cannot stop instantly when the driving torque is zero, it may lead to overshoot. The differential term represented the changing rate of the position error to avoid overshoot. The equation of the controller is given below:

$$y_r = k_{pr} * position\_err + k_{dr} * \frac{d\ position\_err}{dt} - 0.0025 * |velocity| \quad \textbf{eq (2)}$$

where y_r is the position controller output, and k_{pr} and k_{dr} has been set experimentally to be 0.005 and 0.033 to reach its target rotation without overshoot. The $velocity$ **term** represents the real time rotating speed in order to further reduce the overshoot since the rotation speed will need to decelerate earlier if the current velocity is higher.

```
float RotationControl(){
    float yr;

    lead = (rotation < 0) ? -2 : 2;
    position_err = position_tar - (float)abs(position - startPosition);
    diff_position_err = (float)(position_err - oldPosition_err);
    oldPosition_err = position_err;

    yr = kpr * position_err + kdr*diff_position_err-v_const*(float)abs((int)velocity);
    lead = (yr < 0) ? -lead : lead;
    yr = (yr>=0) ? yr : -yr;
    yr = (yr > maxPWM) ? maxPWM : yr;

    return yr;
}
```

Code Snippet 6. ***RotationControl()*** function

4. Performance Analysis

In order to evaluate the performance of the system, it is essential to investigate the CPU utilization, which represents the time proportion the CPU is busy. It is investigated by firstly calculating the theoretical minimum initiation interval value and secondly measuring the experimental maximum execution time. The relationship between the CPU utilization, initiation interval and execution time is given by the equation 3 below:

$$U = \sum_i \frac{T_i}{\tau_i} \qquad \textbf{eq (3)}$$

where U represents the CPU utilization, T_i stands for the execution time and τ_i means the initiation interval.

Measurements can be carried out based on free pins on the microcontroller. Setting one free test pin D4 of the microcontroller to be high when a task is called and resetting it to be low when the task ends. A USB oscilloscope can be used to monitor the voltage of the corresponding pin of D4 which is TP0 on the PCB board and record data. The initiation interval is defined as the time difference between the rising edges of two consecutive voltage pulses. The execution time is defined as the time difference between the rising edge and the falling edge of one pulse. However, due to the department entrance limitation, we did not have a USB oscilloscope. Instead, a timer is used in each task to measure the execution time. Table 2 summarized initiation intervals, execution times and their corresponding CPU utilizations for different tasks used in the system

For the initiation intervals, the ***computation()*** computes 5000 nonces per second, thus it has an interval of 1s. The MotorCtrlFunction task is ticked every 100ms by using a ticker. The MotorPWM has a period of 2500us, and since there are six position changes within this period. The ISR has an interval of 416us. The minimum duration for melody notes is 0.125 seconds.

Task Name	Initiation Interval (s)	Execution Time (s)	CPU Utilization (%)
MotorCtrlFunction	0.1	0.000075	0.075
ISR	0.000416	0.000024	5.77
HashRate	1	0.76	76
Tune	0.125	0.00096	7.68
getMessage	User dependent	0.000008	User dependent
receiveMessage	User dependent	0.000012	User dependent

Table 2. Table of CPU Utilizations for Different Tasks

4.1 Critical Instant Analysis

From the Table 2 above, the HashRate task would consume the longest initiation interval which is 1 second under the worst-case condition. During this one second, besides the HashRate task, the MotorCtrlFunction could run 10 times, ISR task could run 2403 times and the Tune task could play 8 times. Therefore, the total time required is given by:

$$time = 0.76 + 0.000075 * 10 + 0.000024 * 2403 + 0.0096 * 8 = 0.895222 \quad \text{eq(4)}$$

Since the time required to process all tasks (0.895222 second) is shorter than 1 second, all deadlines are met.