DEPARTMENT OF ELECTRICAL AND ELECTRONIC ENGINEERING
IMPERIAL COLLEGE LONDON

EE3-23
EMBEDDED SYSTEMS
BRUSHLESS MOTOR CONTROL REPORT

GROUP NAME: LES HABIBIS
YASMIN BABA - 01196634
CLEMENTINE BIET - 01201920
GEORGIA GRAND - 01217979
ZED AL-AQQAD - 01202742

22nd of March, 2019

TABLE OF CONTENTS

1. Introduction

The following report explores the design and implementation of a brushless motor controller that additionally performs bitcoin mining, returning candidate nonces as frequently as possible. The motor should obey the following specifications:

- Spin for a defined number of rotations and stop without overshooting
- Spin at a defined maximum angular velocity
- Operate at high precision
- Play a melody task while spinning by modulating the control voltage

2. Motor Control Algorithm

In order to achieve a consistent control during motor operation, a reliable serial port to host message passing is needed as well as a correct decoder. The code in *main.cpp* begins by initialising several parameters and threads. It then starts the threads used to run the main motor tasks as well as an independent Bitcoin mining task. The thread responsible for the motor control algorithm is **motorCtrlT** and is segmented as follows.

2.1 Motor Position and Velocity Measurements

```
Ticker motorCtrlTicker;
motorCtrlTicker.attach_us(&motorCtrlTick,100000);
float oldPosition = 0.0;

double tVel = 0.0;
t_vel.start();
while(1){
    oldPosition = ((float)motorPosition)/6.0;
    motorCtrlT.signal_wait(0x1);

    core_util_critical_section_enter();
    tmpPosition = motorPosition;
    core_util_critical_section_exit();
    t_vel.stop();
    tVel = t_vel.read();
    t_vel.reset();
    t_vel.start();
    v = (1/tVel)*((tmpPosition/6.0) - oldPosition);
    position =  ((float)tmpPosition/6.0);
```

Listing 1: Motor Control Thread

As illustrated in the above code snippet, the thread execution is governed by the statement **motorCtrlT.signal_wait(0x1)** which waits for the function **void motorCtrlTick()** to set the signal and trigger the ISR every 100ms. Once the new rotor position is read and stored, it is compared with the old position to calculate the velocity of the motor, as well as used to store the current position.

2.2 Motor Velocity Control

Within the decode function, the serial port input command sets the rotation speed (case 'V'). The decoder will then assign this speed to the global variable *newV*. In **motorCtrlFn(),** the function **velocityControl()** will then be called to set (and return) the motor torque needed to change the velocity accordingly, as follows.

```
if(newV == 0) Ts = maxPWM;
else{
    if (newV <= 0){
        lead = -2;
    }
    else{
        lead = 2;
    }

    //error term
    vError = newV - abs(v);

    //Integral error
    intvError += vError/0.1;
    if(intvError > 880) intvError = 880;
    if(intvError < -880) intvError = -880;

    Ts = Kps*vError + Kis*intvError;

    if(Ts > maxPWM){
       Ts = maxPWM;
    }
}
return Ts;
```

Listing 2: Velocity Control

The *lead* variable exists in order to determine the direction of the motor. When the velocity is negative, the motor will rotate in the reverse direction with a negative torque and vice versa. The rest of the code accounts for the implementation of the proportional-integral control feedback system used to control and set the new velocity precisely. The control equation is as follows:

$$T_s = \left(k_{ps} e_s + k_{is} \int e_s \, dt \right) \operatorname{sgn}\left(\frac{de_r}{dt} \right)$$

where Ts is the torque returned by the function.

2.3 MOTOR POSITION CONTROL

In a similar manner, once the decoder receives a serial port input command for setting a new rotor position (case 'R'), it will assign the new input position to the global variable *newR.* In **motorCtrlFn(),** the function **positionControl()** will be called to set (and return) the motor torque needed to change the rotor position correctly, as follows.

```
float Tr;
rError = targetPosition - motorPosition/6.0;
rDiff = rError - oldError;
oldError = rError;
Tr = (Kpr*rError + Kdr*rDiff);
lead = (Tr >= 0) ?  2 : -2;
Tr = abs(Tr);
if(Tr > maxPWM){
    Tr = maxPWM;
}

return Tr;
```

Listing 3: Position Control

The main difference from the velocity control implementation is that the overshoot must now be taken into account as the motor cannot instantly stop. As a result, there must be a differential term *Diff* in the controller, which is the rate of change of the position error. The equation for this controller is:

$$T_r = \left(k_{pr} e_r + k_{dr} \frac{de_r}{dt}\right)$$

Where Tr is the torque returns.

3. Tasks and Dependencies

As this embedded system carries out multiple tasks, it must be able to handle task concurrency in order to function reliably. Threads have thus been used to implement tasks in parallel and manage tasks that proceed at different rates. The program includes the following three threads:

```
Thread outputMessagesT(osPriorityNormal,1024);
Thread decodeCommandT(osPriorityNormal,1024);
Thread motorCtrlT(osPriorityHigh,1024);
```

Listing 4: Thread Declarations

The inter-task dependencies have been analysed in order to ensure no deadlocking occurs.

3.1 Outputting Messages

The thread **outputMessagesT()** is responsible for receiving messages from various parts of the code and writing them to the serial port. It does so through a **Mail** object which contains a FIFO buffer in which messages are queued in. This is done to prevent **printf()** functions from blocking one another when different parts of the code simultaneously send messages. The mail structure **mail_t** is defined as follows.

```
/* Mail */
typedef struct {
    uint32_t val;
    uint8_t type;

}mail_t;
```

Listing 5: mail_t Struct

All threads in the program are able to add the messages to the queue via the function **putMessage()**.

```
void putMessage(int32_t current_val, uint8_t type){
        mail_t *mail = mail_box.alloc();
        mail->val = current_val;
        mail->type = type;
        mail_box.put(mail);
}
```

Listing 6: putMessage()

Mail::alloc()returns a pointer to the memory that is used to store the message. The code and data are written into the structure and then passed into the queue via **Mail::put()**. The output function uses a blocking function **Mail::get()**, which will wait for an item to be available in the mail box before executing. Finally, the memory unit used is then returned to the pool via **Mail::free()**. All **Mail** methods are therefore

thread-safe as different threads will not be written into the same memory locations. Therefore, the function can be called by various threads simultaneously with no risk of data corrupted.

3.2 Decoding Messages

The thread **decodeCommandT()** decodes an input command sent by the serial port using the **RawSerial** object pc and **attach()** that adds a callback function when a serial interrupt is met. A valid command is one that begins with a letter followed by 16 hexadecimal digits and ended by a carriage return. Therefore, decoding the command is implemented by iterating through the characters using the function **sscanf()**, detecting a carriage return to indicate the end of the command.

Depending on the first letter of the command, the function will implement a change in the necessary variables accordingly. For example, when the letter is 'K', a new bitcoin key will be written into the variable **newKey** as follow.

```
case('K'):
    newKey_mutex.lock();
    sscanf(message, "K%x", &newKey);
    newKey_mutex.unlock();
    break;
```

Listing 7: Newkey

This variable is fed into the bitcoin function **computeHash()** which will take some time to process. **mutex** is used to block the decode thread in case another command is decoded during this processing time, potentially changing the value in **newKey**. A mutex, or a mutual exclusion, will allow only one thread to access a variable at a time and thus will protect **newKey** from changing elsewhere in the code before **computeHash()** is finished with it.

3.3 Motor Control

The thread **motorCtrlT()** is responsible for all motor implementation functions and control. It is the thread with the highest set priority (set as *osPriorityHigh*), within it, the ISR is triggered every 100ms, the full implementation is discussed in *Section 2.* There are two control systems: one to control the position and the other the speed. The interrupt *ISR* tracks the position of the disc hence controlling the rotation of the motor. The speed controller called an interrupt that computes the difference between the old and new state to find the speed.

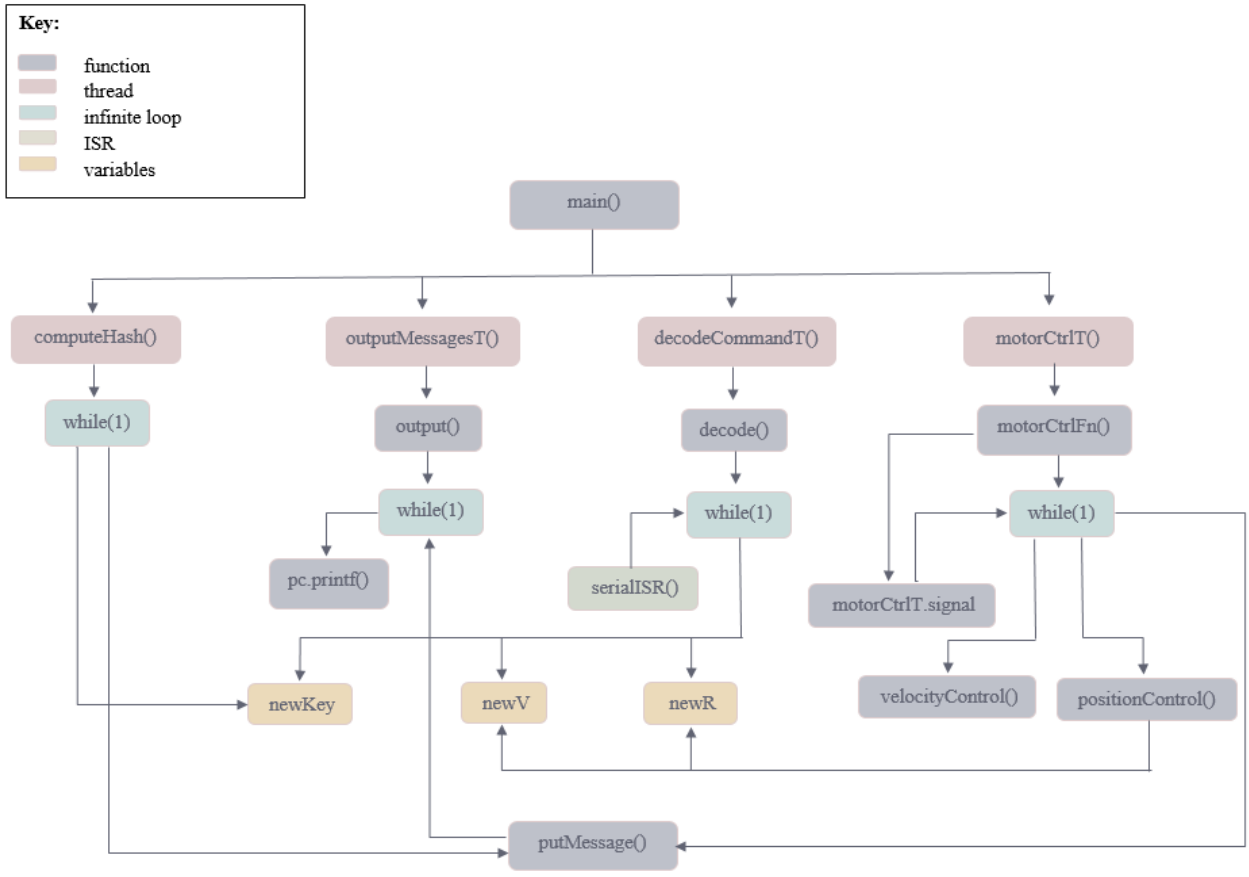
3.4 SHA-256 Hashing

Although not declared as a thread, the bitcoin mining function **computeHash()** acts as the lowest priority thread and runs in an infinite loop. The 'thread' is implemented in order to demonstrate the remaining CPU time available after executing the motor and communication tasks. In other words, it is a 'proof-of-work' type of task to give insight into the CPU capacity.

The task uses the SHA-256 hashing algorithm to generate a 256-bit hash from a sequences of input data within the **computeHash()** function. This function continuously checks as to whether every hash begins with 16 zeros; if so, this is known as a successful nonce and is sent back to the serial host through the mail box.

The number of hashes processed per second is measures as **HASH_RATE**. It is approximately 5200 hashes per second. Since the time taken to perform a hash is constant (dependent upon the SHA-256 hashing algorithm crypto library), the number of hashes processed indicates how much CPU time left as spare once all the motor and communication tasks have been implemented.

3.5 Dependency Flow Chart



4. Task Timings and CPU Utilization

4.1 Performance evaluation

In order to evaluate the performance of the system, the CPU utilization has been investigated. Measurements for the execution times of all tasks being run (excluding bitcoin mining) as well as a theoretical minimum value for their initiation intervals must thus be measured.

In order to measure the values stated, the test pin D13 of the microcontroller was set high at the start of a given task, and then set low at the end of the task. The voltage of its corresponding test pin TP1 on the board was measured using an oscilloscope. An example of the procedure is illustrated below:

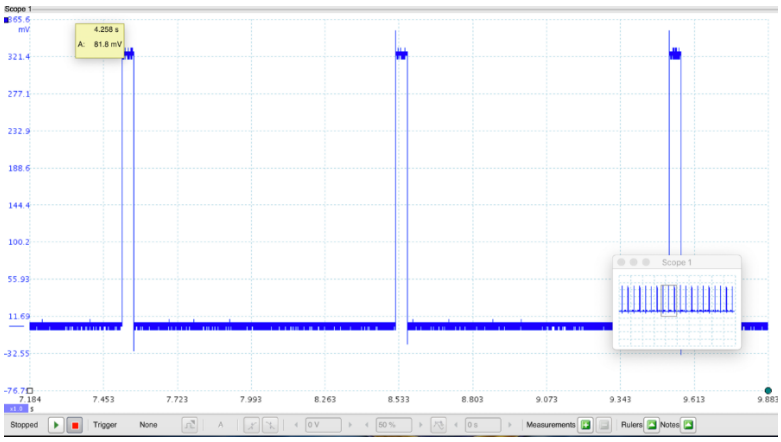


Figure 1: output()

The figure shows the voltage of TP1 while testing the **output()** function. By defining the **initiation interval** (τ) as the time between initiations of a particular task, it corresponds to the difference in time between the rising edge of one plus and the other. It this case it is: 8.495 – 7.495 = 1s. Moreover, the following images shows a zoomed in version of one the pulses:

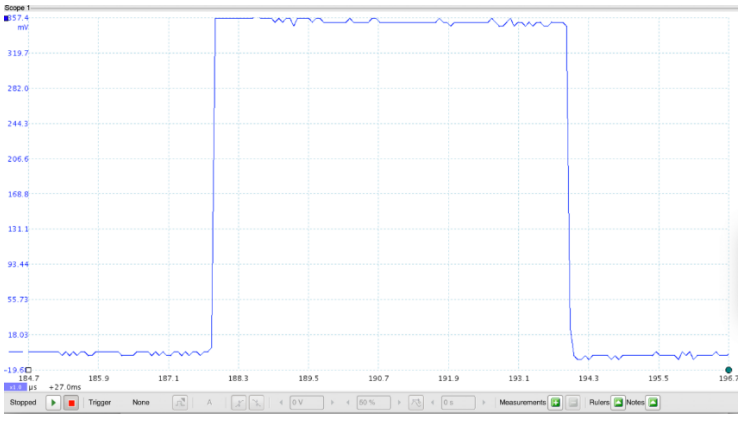


Figure 2: Execution Time

Since the pin goes high at the start of the task and low at the end, the difference in time between high and low is the **execution time** of the task, in this case it is 43.95 μs.

From these values, we can use the following equation:

$$U = \sum_i \frac{T_i}{\tau_i}$$

Where U is CPU utilization, T_i *i*s the exaction time, and τ_i is the initiation interval for a given task. The table below summarized the results for a number of different tasks, following the same procedure the example shown above:

Task	Initiation interval (τ)	Execution time (T)	CPU Utilization (U)
ISR	8.33 ms	6.66 μs	0.08%
Output()	1 s	43.95 μs	0.0044%
MotorCTontrolFn()	100ms	2.6 ms	2.6%
Decode()	User dependent	44.4 μs	User dependent

4.2 Critical Instant analysis of the Rate monotonic Schedule

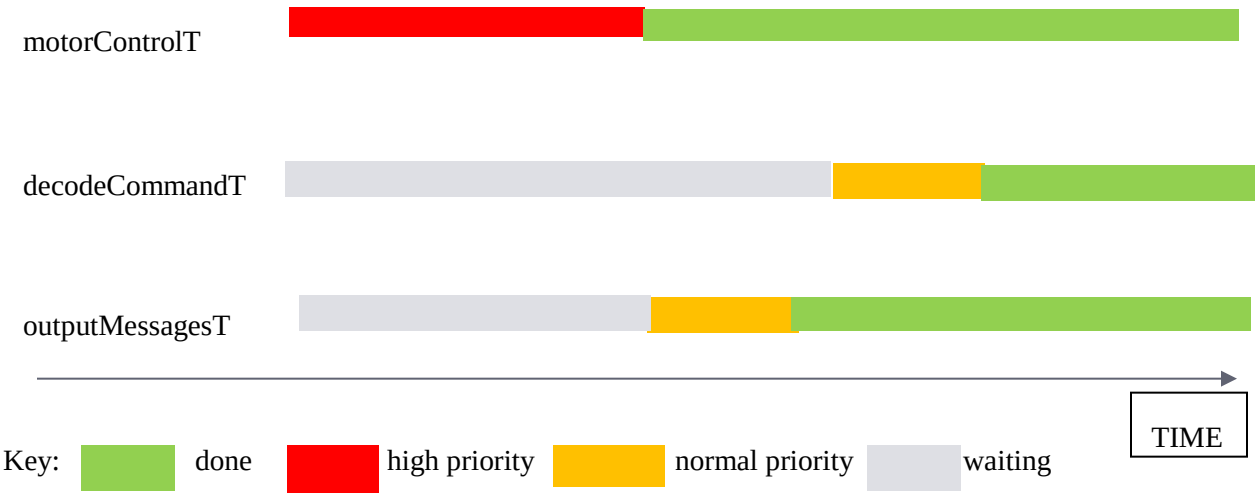
To illustrate and confirm the probe measurement above: to calculate the deadline for the ISR, the motor should be rotating at maximum speed, which is found to be 128.7 rps using the self-test code provided.

$$\omega = \frac{2\pi}{T} \leftrightarrow 128.7 = \frac{2\pi}{T} \leftrightarrow T = 48.82\ ms$$

This means that 48.82 ms are needed to complete a whole revolution. Equally, each interrupt is triggered every 60°, which means the deadline of the interrupt is:

$$T_d = \frac{48.82}{6} = 8.14\ ms$$

The following figure shows a rate-monotonic scheduling of the main tasks (threads) that are run by the system with their theoretical minimum initiation intervals and deadlines (under worst-case conditions):



This shows that all deadlines are met even under worst-case conditions, as the sum of the time required by all 3 threads is below 100ms, which is when the high priority thread will re-iterate (deadline).

5. BONUS

For the bonus, we implement the tune function: by controlling the frequency of the motor plays a tune. To do that, the user inputs a series of notes and duration of the notes. The function then sorts out the notes and durations to convert it to frequencies. The corresponding frequency is used to change the period and the durations to wait for the given time until the next notes. Due to lack of time, the function wasn't integrated in the rest of the system as a specific mailbox for **Char** would have had to be created.