Embedded Systems

Brushless Motor Controller

Team Name: Friends

Coursework 2 Report

Team Members:
Sajid Ali
Tuck Hong 01336059
Lulwa Alkhalifa 01351905

Contents

1 Introduction **1**
- 1.1 Brief . . . 1
- 1.2 Specification . . . 1

2 Motor Control Implementation **1**
- 2.1 Position and Velocity Measurements . . . 1
- 2.2 Velocity Control . . . 1
- 2.3 Position Control . . . 2

3 Tasks and Dependencies **3**
- 3.1 Motor Control . . . 3
- 3.2 Melody . . . 3
- 3.3 Decoding Messages . . . 4
- 3.4 Outputting Messages . . . 5
- 3.5 Bitcoin Mining . . . 5
- 3.6 Dependency Graph . . . 6

4 Timings and CPU Utilisation **6**
- 4.1 Methodology . . . 6
- 4.2 Measure Results . . . 6
- 4.3 Critical Instant Analysis of the Rate Monotonic Scheduler . . . 7

5 Highlighting and footnotes **7**

6 Equations **7**
- 6.1 As part of text . . . 7
- 6.2 In the middle, not numbered . . . 7
  - 6.2.1 And in the middle, numbered . . . 7

7 Adding images **7**
  - 7.0.1 Images side by side . . . 7

8 Citing and referencing **7**
- 8.1 Referencing figures and equations . . . 7
- 8.2 Citing a paper . . . 8

1 Introduction

1.1 Brief

In this report is summarised the implementation of a brushless motor controller which performs additional lower priority tasks in an embedded system.

1.2 Specification

The system was designed to meet the following design criteria:

- Motor will spin for defined number of rotations and stop without overshooting with a precision of 0.5 rotations per number of rotations.
- Motor will spin at a defined maximum angular velocity 5-100 rotations per second, with a precision of 0.5 rotations per second.
- Perform Bitcoin mining task and test 5000 nonces per second. Matching nonces will be sent back to the host.
- Motor will play a melody while it is spinning. The melody is a repeating sequence of notes in C4 octave with durations of 0.125-1 seconds.

This is a paragraph Nam dui ligula, fringilla a, euismod sodales, sollicitudin vel, wisi. Morbi auctor lorem non justo. Nam lacus libero, pretium at, lobortis vitae, ultricies et, tellus. Donec aliquet, tortor sed accumsan bibendum, erat ligula aliquet magna, vitae ornare odio metus a mi. Morbi ac orci et nisl hendrerit mollis. Suspendisse ut massa. Cras nec ante. Pellentesque a nulla. Cum sociis natoque penatibus et magnis dis parturient montes, nascetur ridiculus mus. Aliquam tincidunt urna. Nulla ullamcorper vestibulum turpis. Pellentesque cursus luctus mauris.

And this is a subparagraph Nulla malesuada porttitor diam. Donec felis erat, congue non, volutpat at, tincidunt tristique, libero. Vivamus viverra fermentum felis. Donec nonummy pellentesque ante. Phasellus adipiscing semper elit. Proin fermentum massa ac quam. Sed diam turpis, molestie vitae, placerat a, molestie nec, leo. Maecenas lacinia. Nam ipsum ligula, eleifend at, accumsan nec, suscipit a, ipsum. Morbi blandit ligula feugiat magna. Nunc eleifend consequat lorem. Sed lacinia nulla vitae enim. Pellentesque tincidunt purus vel magna. Integer non enim. Praesent euismod nunc eu purus. Donec bibendum quam in tellus. Nullam cursus pulvinar lectus. Donec et mi. Nam vulputate metus eu enim. Vestibulum pellentesque felis eu massa.

2 Motor Control Implementation

2.1 Position and Velocity Measurements

I'm guessing this has probably changed since yesterday. I'm so sorry about that :(

2.2 Velocity Control

Velocity was controlled using a proportional and integral term in the control equation.

$$y_s = \left(k_p(s-v) + k_i \int (s-v)dt \right) \text{sgn}(E_r)$$

y_s is the speed controller output which is compared with the position controller output to determine the motor PWM. $k_p$and k_i are the tuning parameters, s is the maximum speed (set by the host and initialised at 100 rotations per second upon startup) , E_r is the position error, and v is the measured velocity. The integral term is implemented as follows:

```
    //update integral error
    acc_v_error += vError * 10;
    if(acc_v_error > MAX_V_ERROR) acc_v_error = MAX_V_ERROR;
    if(acc_v_error < -MAX_V_ERROR) acc_v_error = -MAX_V_ERROR;
```

Further processing is done to convert the controller output into a PWM duty cycle. The absolute value of y_s is considered and the value of lead is modified to take the direction of rotation into account.

```
lead = (VE < 0) ? -lead : lead; // torque in opposite direction to slow the motor
    down
```

Also a limit is applied so that the PWM duty cycle (**power**) does not exceed the maximum value of 1

```
void UpdateMotorPower(float power)
{
    power = abs(power);
    if(power > 1.0) power = 1.0;

    MotorPWM.write(power)
}
```

2.3 Position Control

For position control both a proportional and differential term are used.

$$y_r = k_p E_r + k_d \frac{dE_r}{dt}$$

k_p and k_d are the tuning parameters, and E_r is the position error. This is calculated as shown below:

```
double posError = targetPosition - position;
double posErrorOld = targetPosition - positionOld;

return (Kpr*posError) + (Kdr*(posError - posErrorOld)/(timePassed));
```

Combining velocity and position control is done by choosing the most conservative value and is implemented as follows:

```
//find distance error
DE = DistanceError((double)velocityTimer.read());

//find velocity error
VE = VelocityError(abs(velocity));

//choose error to pass to motor
float power_test = 0;
if (velocity >= 0)
    power_test = std::min(VE,DE);
else
    power_test = std::max(VE,DE);
```

3 Tasks and Dependencies

3.1 Motor Control

Motor control is implemented in a high priority thread **motorCtrlT**. Within the main function responsible for motor control **motorCtrlFn**, the interrupt service routine **UpdateMotorPos** is attached to the photo-interrupter inputs, such that the motor PWM is updated whenever there is a change in any of the inputs.

A ticker object is also created within **motorCrtlT** and the callback function **motorCtrlTick()** is attached, such that it runs every 100ms. The purpose of this interrupt is to set a signal using the RTOS function **Thread::signal_set()** to **motorCtrlT**. This ensures that velocity is measured and the control output for power is updated every 100ms.

```
void motorCtrlTick()
{
    motorCtrlT.signal_set(0x1);
}
```

Furthermore, when calculating velocity and distance error, the mutex **maxSpeed_mutex** is used to protect the variable **maxSpeed** and the mutex **targetPosition_mutex** is used to protect the variable **targetPosition** from simulataneous access. This is important because the decode command thread also needs to write to these variables when speed and position commands are received from the host. An example of this is shown below.

```

float DistanceError(double timePassed)
{
    // get current and previous distacne error
    targetPosition_mutex.lock();
    maxSpeed_mutex.lock();

    double posError = targetPosition - position;
    double posErrorOld = targetPosition - positionOld;

    maxSpeed_mutex.unlock();
    targetPosition_mutex.unlock();

    return (Kpr*posError) + (Kdr*(posError - posErrorOld)/(timePassed));
}
```

3.2 Melody

By modulating the PWM period and hence the drive current, the motor effectively behaves like a tune generator. A separate thread **TuneThread** is responsible for changing the PWM period based on the notes present in melody commands given by the user. The notes that are allowed belong to the C4 octave, in other words, frequencies 261.63 Hz (C4) up to 493.88 Hz (B4).

First, a map containing all the allowed notes and their corresponding PWM periods is initialised. Note that the PWM periods (in μs) are calculated using $\frac{1000000}{frequency}$.

```
    std::map<std::string, int32_t> tonefreqmap =
        {{"A", (int32_t)(1000000 / 440)}, {"A#", (int32_t)(1000000 / 466.16)},
         {"A^", (int32_t)(1000000 / 415.30)}, {"B", (int32_t)(1000000 / 493.88)},
         {"B^", (int32_t)(1000000 / 466.16)}, {"C", (int32_t)(1000000 / 261.63)},
         {"C#", (int32_t)(1000000 / 277.18)}, {"D", (int32_t)(1000000 / 293.66)},
         {"D#", (int32_t)(1000000 / 311.13)}, {"D^", (int32_t)(1000000 / 277.18)},
```

```
7           {"E", (int32_t)(1000000 / 329.63)}, {"E#", (int32_t)(1000000 / 349.23)},
8           {"E^", (int32_t)(1000000 / 311.13)}, {"F", (int32_t)(1000000 / 349.23)},
9           {"F#", (int32_t)(1000000 / 369.99)}, {"F^", (int32_t)(1000000 / 329.63)},
10          {"G", (int32_t)(1000000 / 392)}, {"G#", (int32_t)(1000000 / 415.30)},
11          {"G^", (int32_t)(1000000 / 369.99)}};
```

In an infinite loop, the thread repeats the following steps where **tonesQ** is a queue containing the notes and durations that need to be played.

```
1 tonesQ_mutex.lock();
2 tone = tonesQ.front();
3 tonesQ.pop();
4 tonesQ.push(tone);
5 tonesQ_mutex.unlock();
6
7 duration = tone.back();
8 interval = ((int)duration - (int)'0') * 125;
```

Since **tonesQ** is a shared data structure also accessed by **TerminalListenerThread**, the popping and pushing actions are protected with a mutex. Note that after popping a tone from **tonesQ** the same tone is immediately pushed back, as the specification requires the motor to play a given melody on loop.

Following the regex for melody commands where the last entry in the command is an integer corresponding to the duration of the note (number of eighths of a second), the exact duration in milliseconds is obtained in line 8. Note that **(int)'0'** represents the offset value for numerical ASCII characters such that when subtracted from **(int)duration**, duration (which is a **char**) is effectively "converted" into an **int**.

After setting the new PWM period using **tone** and **tonefreqmap** initialised previously, the thread is then put to sleep using **ThisThread::sleep_for(interval)** so that a given note will last for the specified amount of time before the loop continues with the next note to be played. This solves the issue of blocking other threads from running (e.g. the bitcoin thread which will result in the thread failing to meet 5000 nonces per second) when **wait** is used instead.

This task is dependent on a non-empty **tonesQ**. This dependency is released by **TerminalListenerThread** when a melody command is received from the user.

3.3 Decoding Messages

The system receives commands from a host over a serial interface at 9600bps and follows the given syntax specifications:

- Each command ends with a carriage return character.
- The syntax for rotation commands is the regular expression R-?\d{1,4}(\.\d)?
- The syntax for maximum speed commands is the regular expression V\d {1,3}(\.\d)?
- The syntax for setting the bitcoin key is the regular expression K[0-9a-fA-F]{16}
- Matching bitcoin nonces should be sent to the host with a message matching the regular expression N[0-9a-fA-F]{16}
- The syntax for melody commands is the regular expression T([A-G][#^]?[1-8]){1,16} (where # and ^are characters)

An interrupt service routine **serialISR** receives each incoming byte from the serial port and places it into a queue. This is done by using the method **uint8_t RawSerial::getc()** to retrieve a byte from the serial port and the method **Mail::put()** to put a **Mail** message in a **mail_box** queue.

Decoding messages is implemented in **TerminaListenerThread** as a normal priority thread. **serialISR** is attached to serial port events, and in an infinite loop the method **Mail::get()** is used to wait for new characters. Upon receiving a return carriage character from mail, the command message is passed to functions that try to match it with the expected syntax. The command message is then placed in another **mail_box** to be communicated to the host via serial.

```
if (command.back() == '\r')
{
    decodeSpeedCommand(command);
    decodePositionCommand(command);
    decodeKeyCommand(command);
    decodeTuneCommand(command);
    putMessage(PRINT_MESSAGE, "Got Message: " + command);
    command = ""; //Reset input
```

The method used detemine if the command message is valid varies for the different commands.The standard C++ regular expressions library is used to decode speed and position commands. The function **sscanf** is used to decode the new key command, and melody commands are decoded by iterating through the message string as shown below.

```
for(int i = 1; i < input.length(); i++)
{
    if(input[i] == 'A' || input[i] == 'B' || input[i] == 'C' || input[i] == 'D' ||
        input[i] == 'E' || input[i] == 'F' || input[i] == 'G') {
        tone = "";
        tone += input[i];
    }
    if(input[i] == '#' || input[i] == '^') {
        tone += input[i];
    }
    if(input[i] == '1' || input[i] == '2' || input[i] == '3' || input[i] == '4' ||
        input[i] == '4' || input[i] == '5' || input[i] == '6' || input[i] == '7'
    ||
        input[i] == '8') {
        tone += input[i];
        pc.printf("Tone: %s\n\r", tone.c_str());

        tonesQ.push(tone);

    }
}
```

If the message received from the serial port matches any of the expected commands, then the relevant changes are made to implement the command and a **Mail** message to confirm receiving a command is put into a **mail_box** and communicated to the host via serial.

In **decodeSpeedCommand**, for example, the new maximum speed is written into a global variable **maxSpeed** where it will be read by the motor control thread. This variable is protected by a mutex **maxSpeed_mutex** to prevent simultaneous access.

3.4 Outputting Messages

3.5 Bitcoin Mining

The bitcoin mining task is implemented in **BitcoinThread** as a low priority thread. The task is to look for matching nonces such that when combined with a key provided by the user and the

48 bytes of constant payload, will produce a SHA-256 hash that begins with 16 zeros. This is done by simply initialising the nonce to 0 and incrementing by one on each attempt.

In order to regulate the mining task so that it satisfies the throughput specification of exactly 5000 nonces per second, the **Ticker** class is used to set up a timer interrupt that sends a signal to **BitcoinThread** every second. Every time the thread is released by the signal (which happens once per second), 5000 nonces are tested. The specification also requires the thread to send matching nonces back to the host. This is done by putting a **Mail** message pointer of type **BITCOIN_NONCE** in the **mail_box** queue.

```
while (true) {
    BitcoinThread.signal_wait(0x1);
    for(int i = 0; i < 5000; i++) {
        newKey_mutex.lock();
        *key = newKey;
        newKey_mutex.unlock();
        sha256.computeHash(hash, sequence, 64);
        *nonce +=1;

        if(hash[0] == 0 && hash[1] == 0) {
            uint64_t numbernonce = *nonce;
            putMessage(numbernonce);
        }
    }
}
```

A dependency this task has is the semaphore-like signal from the timer interrupt. This dependency is released once per second (by the timer interrupt). Note that a mutex is used here as **newKey** is a shared variable that is also accessed by **TerminalListenerThread** when a new bitcoin key is set by the user.

3.6 Dependency Graph

4 Timings and CPU Utilisation

4.1 Methodology

To ensure the system would meet the design criteria outlined in the introduction it was crucial that we measure and simulate the worst case performance.

4.2 Measure Results

Task	Initiation Interval (t)	Execution Time (T)	CPU Utilization (U)
ISR	0	0	0
Output	0	0	0
Motor Control	0	0	0
Decode	0	0	0
BitCoin Mining	1s	0	0
Melody	0.125s	0	0

Table 1: Task timing and resource utilization.

4.3 Critical Instant Analysis of the Rate Monotonic Scheduler

5 Highlighting and footnotes

You can make words **bold**, *italicise* them, underline words or **make them *stand out* regardless of the surrounding**. You can break a line
mid sentence and make footnotes like this [1].

6 Equations

6.1 As part of text

In total 85 distinct galaxies were identified in the Hubble Deep Field image provided, the list of which can be found in Appendix B at the end of this report. Poisson statistics states that the error in the number of galaxies counted (N_x) is simply the root of the count ($\sqrt{N_x}$), this can be represented as a percentage by the following equation, $N_x^2 + N^2$.

6.2 In the middle, not numbered

$$\alpha\beta \times \frac{2G}{x^2 B_n}$$

$$\frac{-b \pm \sqrt{b^2 - 4ac}}{2a} + \sin x$$

$$\text{mag} = 5 \times \log_{10}\left(\frac{D_l}{10}\right) + \text{M}_{\text{corr}}$$

6.2.1 And in the middle, numbered

$$A = bx + 24 \times F_x \tag{1}$$

7 Adding images

Hello, here is some text without a meaning. This text should show what a printed text will look like at this place. If you read this text, you will get no information. Really? Is there no information? Is there.

Sed commodo posuere pede. Mauris ut est. Ut quis purus. Sed ac odio. Sed vehicula hendrerit sem. Duis non odio. Morbi ut dui. Sed accumsan risus eget odio. In hac habitasse platea dictumst. Pellentesque non elit. Fusce sed justo eu urna porta tincidunt. Mauris felis odio, sollicitudin sed, volutpat a, ornare ac, erat. Morbi quis dolor. Donec pellentesque, erat ac sagittis semper, nunc dui lobortis purus, quis congue purus metus ultricies tellus. Proin et quam. Class aptent taciti sociosqu ad litora torquent per conubia nostra, per inceptos hymenaeos. Praesent sapien turpis, fermentum vel, eleifend faucibus, vehicula eu, lacus.

7.0.1 Images side by side

8 Citing and referencing

8.1 Referencing figures and equations

Expression $4 \times 3 = G \times x$ naturally follows from Eq 1, and both of these things have a lot to do with Fig **??**.

[1] a footnote

8.2 Citing a paper

This statement has a citation at the end of it **?**, and this one has two **??**. A citation with parenthesis is sure to follow [**?**].