Embedded Systems

Brushless Motor Controller

Team Name: Friends

Coursework 2 Report

Team Members:
Sajid Ali 01341393
Tuck Hong 01336059
Lulwa Alkhalifa 01351905

Contents

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.

2 Motor Control Implementation

2.1 Position and Velocity Measurements

The code for updating the position and speed of the motor can be found below. The position of the rotor is updated in the ISR whenever the state changes.

The velocity is calculated in the motor control thread every 100ms using the up to date position. A timer is used to calculate the time-elapsed since the last velocity measurement as opposed to assuming its 100ms; this is to improve the accuracy of the result.

```
1  //Update position (ISR)
2  if (direction_detected == 5) position--;
3  else if (direction_detected == -5) position++;
4  else position += (direction_detected);
5
6  //Calculate velocity (motorCtrlFn)
7  double timeElapsed = velocityTimer.read();
8  velocityTimer.reset();
9  double velocity = (position - positionOld) * (1 / timeElapsed);
```

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:

```
1     //update integral error
2     acc_v_error += vError * 10;
3     if(acc_v_error > MAX_V_ERROR) acc_v_error = MAX_V_ERROR;
4     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.

```
1  //set direction of torque
2  lead = (DE >= 0) ? 2 : -2;
```

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

```
1  void UpdateMotorPower(float power)
2  {
3      power = abs(power);
4      if(power > 1.0) power = 1.0;
5
6      MotorPWM.write(power)
7  }
```

2.3   Position Control

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

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

where k_p and k_d are the tuning parameters, and E_r is the position error.

This is calculated as shown below:

```
1  double posError = targetPosition - position;
2  double posErrorOld = targetPosition - positionOld;
3
4  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:

```
1  //find distance error
2  DE = DistanceError(timeElapsed);
3
4  // add non-linearity to prevent motor oscillation at small speeds (kills the error
       if within 0.5 rots)
5  if(abs(targetPosition - position) < 3)
6  {
7      DE = 0;
8  }
9
10 //find velocity error
11 VE = VelocityError(abs(velocity));
12 if(VE < 0) VE = 0;
13
14 //choose error to pass to motor
15 float power_test = std::min(abs(VE), abs(DE));
```
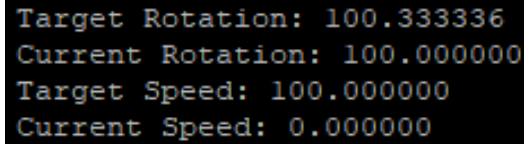
A non-linearity is applied the distance error term to prevent oscillations at lower speeds.

2.4   Accuracy Measurements

Once the controller had been implemented and optimised we carried out tests to ensure the specification would be met. We varied the target speed of the rotor and measured the steady state velocity (i.e. the velocity when the only constraint on the motor power was the target speed). We also noted the final position of the rotor and compared it to the target position.

An example of the terminal output is shown below.

Figure 1: Sample terminal output.

The speed of the motor was varied from 10-100 rps by increasing the speed in intervals of 10 rps. As you can see from the results in the table below, the motor consistently meets both performance criteria of speed and distance error with no overshoot.

Target Speed (rps)	Target Distance (r)	Actual Distance (r)	Steady State Speed (rps)
10	1000.00	1000.00	10.01
20	1000.00	1000.00	19.97
30	1000.00	1000.00	29.99
40	1000.00	1000.00	40.01
50	1000.00	1000.00	49.85
60	1000.00	1000.00	60.01
70	1000.00	1000.00	69.94
80	2000.00	1999.50	80.01
90	2000.00	2000.00	89.97
100	2000.00	1999.50	99.98

Table 1: Position and speed accuracy measurements.

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();

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

    targetPosition_mutex.unlock();

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

A dependency this task has is the semaphore-like signal from the timer ticker. This dependency is released every 100ms (by the timer interrupt).

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)},
         {"E", (int32_t)(1000000 / 329.63)}, {"E#", (int32_t)(1000000 / 349.23)},
         {"E^", (int32_t)(1000000 / 311.13)}, {"F", (int32_t)(1000000 / 349.23)},
         {"F#", (int32_t)(1000000 / 369.99)}, {"F^", (int32_t)(1000000 / 329.63)},
         {"G", (int32_t)(1000000 / 392)}, {"G#", (int32_t)(1000000 / 415.30)},
         {"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.

```
tonesQ_mutex.lock();
tone = tonesQ.front();
tonesQ.pop();
tonesQ.push(tone);
tonesQ_mutex.unlock();

duration = tone.back();
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][#ˆ]?[1-8]){1,16} (where # and ˆare characters)

An interrupt service routine **serialISR** receives each incoming byte from the serial port and places it into a queue (**inCharQ**). 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.

```
1  if (command.back() == '\r')
2  {
3      decodeSpeedCommand(command);
4      decodePositionCommand(command);
5      decodeKeyCommand(command);
6      decodeTuneCommand(command);
7      putMessage(PRINT_MESSAGE, "Got Message: " + command);
8      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.

This task is dependent on a non-empty **inCharQ**. This dependency is released by **serialISR** when a characters are received from the user.

3.4   Outputting Messages

Our system can print many different types of messages each of which may require different size and types of data. To streamline the code we decided to create a single mail struct which could hold a wide variety of data. This means that the size of print items is longer than it could have been, but it uses fewer lines of code and means we need a queue of only a single type.

Items are added to the print queue through the calling of any of the **putMessage()** overload functions. Each functions takes a type and a list of data items to pass to the terminal. The **int type** argument is used by the print thread to output a custom message format for each message type. The message types are defined at the top of the code below.

```
#define PRINT_MESSAGE 0
#define BITCOIN_NONCE 1
#define UPDATED_KEY 2
#define DISTANCE_STATUS 3
#define SPEED_STATUS 4
#define HASH_RATE 5

typedef struct {
  uint8_t    type;
  uint8_t      number;
```

```
11    float     dFloat;
12    float     dFloat1;
13    float     dFloat2;
14    std::string message;
15  } mail_t;
16
17  void putMessage(int type, uint8_t number);
18  void putMessage(int type, double number);
19  void putMessage(int type, std::string message);
20  void putMessage(int type, double posError, double velError);
```

Printing messages to the terminal is handled by the **MessengerThread** which calls the function **outputToTerminal()**. The functions contains an infinite loop which constantly polls the **mail_box** to see if the queue is not empty. Once a message is pulled from the queue the type is checked and a custom print format is applied depending on the type. Finally, the **mail_box** is released.

```
1   //check each mailbox
2   osEvent evt = mail_box.get();
3   if (evt.status == osEventMail)
4   {
5       mail_t *mail = (mail_t*)evt.value.p;
6
7       switch(mail->type)
8       {
9           case(PRINT_MESSAGE):
10              pc.printf("\n\r %s\r\n ", mail->message.c_str());
11              break;
12          case(BITCOIN_NONCE):
13              ...
14          case(HASH_RATE):
15              ...
16          case(UPDATED_KEY):
17              ...
18          case(DISTANCE_STATUS):
19              ...
20          case(SPEED_STATUS):
21              ...
22      }
23
24      mail_box.free(mail);
25  }
```

This task is dependent on a non-empty **mail_box**. This dependency is released by **TerminalListenerThread** when commands are decoded.

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.

```
1  while (true) {
2      BitcoinThread.signal_wait(0x1);
3      //CONST_HASH_RATE = 5000
4      for(int i = 0; i < CONST_HASH_RATE; i++) {
5          newKey_mutex.lock();
6          *key = newKey;
7          newKey_mutex.unlock();
8          hasher.computeHash(hash, sequence, 64);
9          *nonce +=1;
10
11         if(hash[0] == 0 && hash[1] == 0) {
12             putMessage(BITCOIN_NONCE, (uint8_t)*nonce);
13         }
14     }
15 }
```

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.
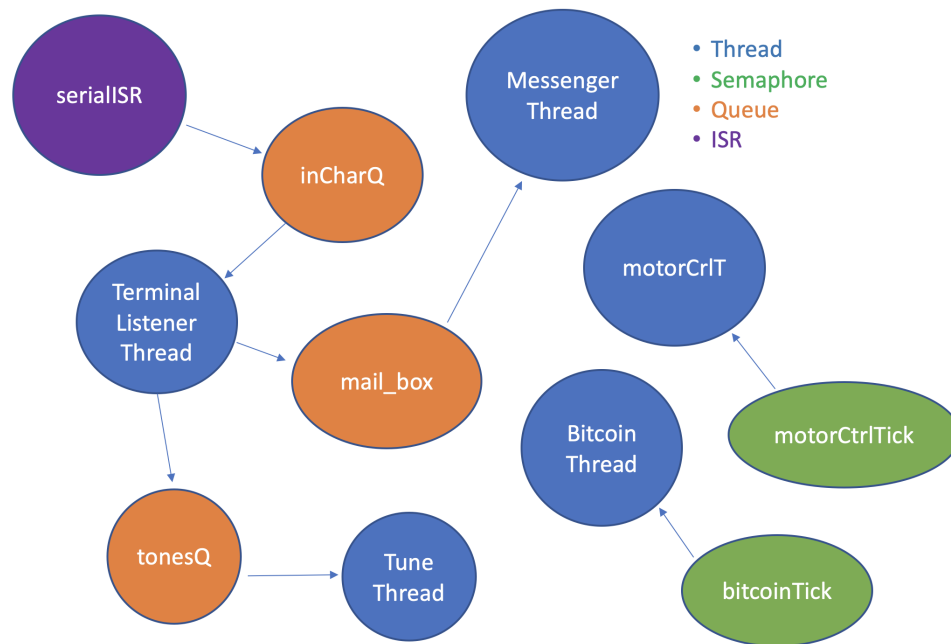
3.6   Dependency Graph

Figure 2: Dependency graph showing no possibility of deadlock. Note that mutexes are excluded from this analysis.

4   Timings and CPU Utilisation

4.1   Methodology

The CPU usage of each task running in the micro-controller was indirectly observed by pulling one of the digital I/O pins high and low to show when a task was executing. Measuring the time

between pulses and the duration of each pulse would tell us the initiation interval and execution time of each task.

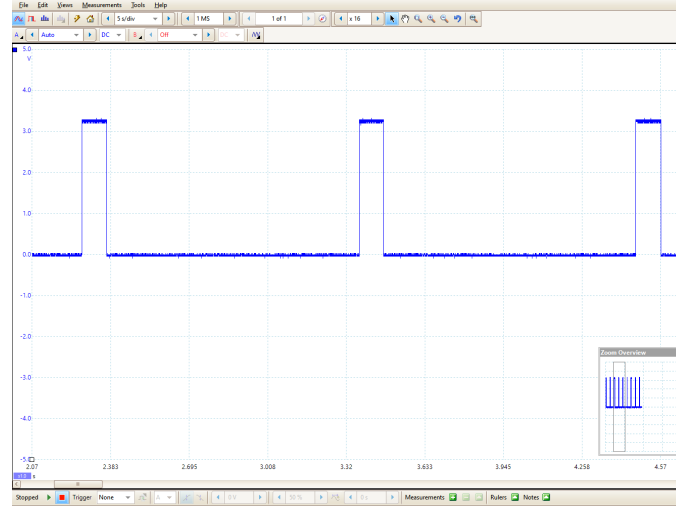We used a USB oscilloscope to take the measurements. An example of a trace we obtained is shown in Figure 3.

Figure 3: Picoscope pulse trigger.

4.2   Measure Results

Task	Initiation Interval (t)	Execution Time (T)	CPU Utilization (U)
ISR	1.28ms	5.0us	0.40%
Output	1.0s	101us	10.11%
Motor Control	100ms	75us	0.08%
Decode	-	53us	-
BitCoin Mining	1.0s	868ms	86.8%
Melody	0.125s	1.0us	0%

Table 2: Task timing and resource utilization.

4.3   Critical Instant Analysis of the Rate Monotonic Scheduler

By assuming the worst case scenario in terms of resource usage, we can test to see if our design will always meet the specifications. To test this we will calculate the CPU time used up by every thread within a single second. If we are able to execute all necessary tasks within this time frame, then we can be confident in our design.

We start by assuming that the motor is running at full speed (i.e. 100 rps). Since the ISR is triggered 6 times per rotation, this means the ISR will execute 600 times per second, or once every 1.7ms for 5us.

For every **one second** the following tasks must execute:

- ISR: 600 * 5us = **3ms**

- Output: **101us**

- Motor Control: 10 * 75us = **0.75ms**

- Decode: **N/A**

- Bitcoin Mining: **868ms**

- Melody: 8 * 1us = **8us**

Adding the CPU time together yields:

$$3ms + 101us + 0.75ms + 868ms + 8us = \textbf{871.9ms} < 1000.0ms$$

$$cpuUtilisation = (871.9ms/1000ms) = \textbf{87.2\%}$$

This means theoretically there is enough CPU time for every task to execute.

However, assuming all the tasks want to execute at a critical instance we can plot the execution to see if the tasks finish executing on time.

The main point of contention is will the motor ISR executing every 1.7ms block other critical tasks from completing on time.

The only additional calculation required is the number of bitcoin hashes which must be calculated on average in 1.7ms.

$$hashTime = 0.0017 * 868ms = \textbf{1.48ms}$$

As you can see in the figure below, if every task want to execute at the same time, the scheduling will allow every task to complete and the ISR will be able to meet its deadline.
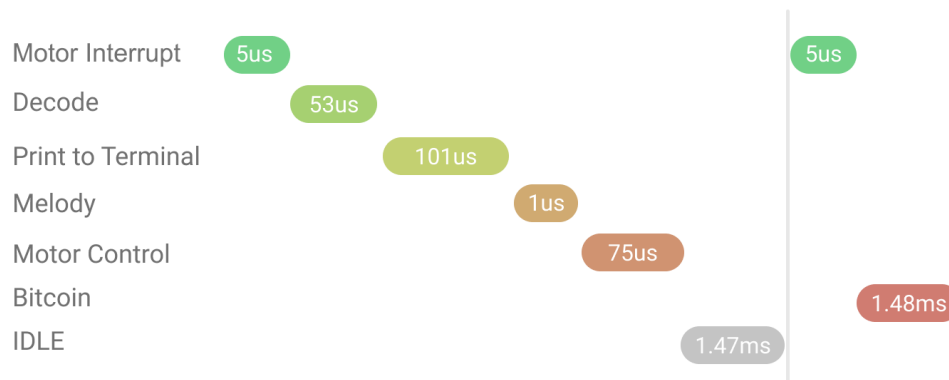

Figure 4: Worst case scenario scheduler.