EE3 EMBEDDED SYSTEMS

Brushless Motor Control

TEAM BGJR

Babalola Ajose – CID: 01243854
Georgina Ramage – CID: 01192008
Jamie Thompson – CID:
Rohan Tangri

Table of Contents

MOTOR CONTROL ALGORITHM **2**

Controlling Motor Velocity 2

Controlling Position 2

Controlling Speed and Position 3

REAL-TIME PROGRAMMING TECHNIQUES **3**

Interrupts 4

Threads 4

Analysis of Inter-task Dependencies 4

PERFORMANCE AND TIMING **5**

Task Timing Analysis 5

Motor Rotation *5*

Motor Control Thread *5*

Decode Thread *6*

Print Thread *6*

Critical Instant Analysis 6

CPU Utilisation 6

Motor Control Algorithm

Variables for motor control; velocity and precision; are set by command from the serial port. The motor control algorithm is then implemented in the motor control thread. Here, control functions are implemented that set a torque value.

Controlling Motor Velocity

Initially, motor velocity was controlled using a proportional controller. This is implemented by first calculating the speed error term e_s:

$$e_s = s_{max} - |s|$$

EQUATION 1 - SPEED ERROR

The control function is then:

$$T_S = K_{Ps} e_s$$

EQUATION 2 - PROPORTIONAL CONTROLLER FOR VELOCITY

The proportional control constant for the velocity K_{Ps} is set to the optimum value that allowed the velocity to reach a maximum as it approached the target velocity before the onset of oscillation. A limitation of this control method comes due to friction on the motor, which causes a steady-state error. Thus the controller is optimised with the addition of an integral term creating a PI controller:

$$T_s = K_{Ps} e_s + K_{Is} \int e_s \, dt$$

EQUATION 3 - PI CONTROLLER FOR VELOCITY

The addition of the integral term helps to overcome the steady state error, with K_{Is} chosen as its maximum while avoiding speed overshoot. In practice, a simpler solution was devised with the addition of a constant term to overcome the steady-state error:

```
torque_V = (0.08*es+0.62)* sgn_v;
```

CODE SNIPPET 1 - CONTROLLER FOR VELOCITY

The compromise of increased simplicity leads to less accuracy and the assumption that the effect of friction is constant at all speeds.

Controlling Position

Position controlled is defined to allow for the motor to spin for a predetermined number of rotations before coming to a halt. In this program, position control is implemented using a proportional differential (PD) controller, defined below:

$$T_r = K_{Pr} e_r + K_{Dr} \frac{de_r}{dt}$$

EQUATION 4 - PD CONTROLLER FOR POSITION

The derivative in this case is a simple difference:

```
erderiv = er - erold;                              //derivative of position error
erold = er;                                         // set er previous version
```

CODE SNIPPET 2 - POSITION DERIVATIVE

The optimum values for the constants K_{Pr} and K_{Dr} are 30 and 20 respectively, implemented as below:

```
torque_R = 30*er +20*erderiv;
```

CODE SNIPPET 3 - POSITION CONTROLLER

Controlling Speed and Position

In the case in which velocity and position have been set concurrently, the chosen torque is dependent on the sign of the velocity. For a positive velocity, the minimum torque is chosen; for a negative velocity, the maximum torque is chosen:

```
if(sgn_v > 0){
    if(torque_V > torque_R){
        torque = torque_R;
    }
    else{
        torque = torque_V;
    }
}
else{
    if(torque_V > torque_R){
        torque = torque_V;
    }
    else{
        torque = torque_R;
    }
}
```

CODE SNIPPET 4 - CONTROLLING SPEED + POSITION

Real-Time Programming Techniques

The diagram below shows the flow of code within our program, including our interrupts and thread

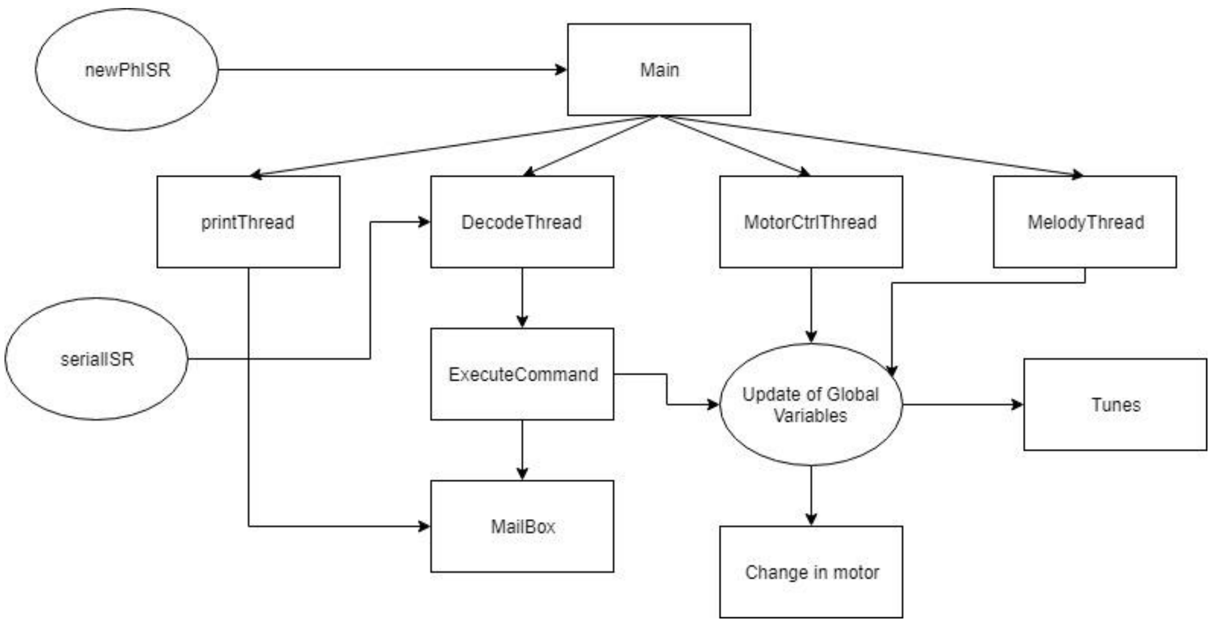

FIGURE 1 - CODE-FLOW DIAGRAM

Interrupts

Two interrupt service routines are implemented in the program. The photo-interrupter ISR (`newPhISR()`), is used to keep track of the motor position. This task is carried out here as it is a high priority task.

The other ISR utilised in the program is the `serialISR()`, which serves to place characters received from PC instructions into the queue `inCharQ`. An interrupt is used in this case due to the fact that data is received asynchronously from the serial port.

```
void serialISR(){
    uint8_t newChar = pc.getc();
    inCharQ.put((void*)newChar);
}
```

CODE SNIPPET 5 - SERIAL PORT ISR

Threads

Three threads are utilised to perform specific tasks. The motor control thread `motorCtrlT` is defined with medium priority and a stack size of 1024.

```
Thread printThread;
Thread decodeThread;
Thread motorCtrlT (osPriorityNormal,1024);
```

CODE SNIPPET 6 - THREAD INITIALISATION

The motor control thread is triggered by a Ticker class timer interrupt which send a signal to start running motor control task:

```
void motorCtrlTick(){
    motorCtrlT.signal_set(0x1);
}
// :
// :
// :
void motorCtrlFn(void){
    // :
    // :
    Ticker motorCtrlTicker;
    motorCtrlTicker.attach_us(&motorCtrlTick,100000);
    // :
    // :
}
```

CODE SNIPPET 7 - TICKER INTERRUPT

The thread `printThread` is used for communication with the serial port since it is not possible to use `printf()` with multiple tasks attempting to use it at once. The final thread, `decodeThread`, is used to decode incoming instructions from the serial port.

Analysis of Inter-task Dependencies

To deal with the fact that we have different threads that are using the same resources, `newkey` and `duty`, `Mutex` was used to lock the variables and synchronise them between threads.

```
Mutex newKey_mutex;
Mutex motor_mutex;
```

CODE SNIPPET 8 - MUTEX INITIALISATION

When a thread is using a mutex resources it is locked and therefore a thread wishing to use the resource is queued and waits till it has been unlocked before the queued thread can use the shared resources. For example, `newKey` is used in the function `executeCommand()` which sets the value of the key specified from the serial command, and in the main function during the hash calculation:

```
void executeCommand(char* command){
    switch(command[0]){
        case 'K': newKey_mutex.lock();
            sscanf(command,"K%llx",&newKey);
            newKey_mutex.unlock();
            // :
            // :
// :
// :
int main() {
    // :
    // :
    while (1) {
        newKey_mutex.lock();
        *key = newKey;
        newKey_mutex.unlock();
    // :
    // :
}
```

Code Snippet 9 - Mutex Lock/Unlock for newKey

Moreover, these global variables have been declared `volatile` to tell the compiler that these variables can change unexpectedly when running and prevent the compiler from making optimisations that can cause run-time errors.

Performance and Timing

Task Timing Analysis

Motor Rotation

Motor position is defined using 6 states, each 60° apart. The motor rotation ISR is called every time the position changes, i.e. 6 times per rotation. The minimum initiation interval is then given by the initiation interval when the motor is rotating at its top speed. With the time period for one rotation given by:

$$\omega = \frac{2\pi}{T} \Longrightarrow T = \frac{2\pi}{\omega} = \frac{2\pi}{90\text{ rps}} = 69.8\text{ ms}$$

Equation 5 - Minimum Rotation Period

The initiation interval is then given by:

$$\tau = \frac{T}{6} = \frac{69.8}{6} = 11.6\text{ ms}$$

Equation 6 - Minimum Initiation Interval Motor ISR

Motor Control Thread

The motor control thread is initiated by a Ticker timer which calls the function every 100ms. Thus the minimum initiation interval is 100ms.

The execution time was measured as 6ms.

Decode Thread

The decode thread read new characters from the serial port, places them in an array and then decodes the contents of the command. The thread is initiated upon the receipt of any character. With 8 bits/char this gives:

$$\tau = \frac{8\text{ bits}}{\{bit\ rate\}} = \frac{8\text{ bits}}{9600\text{ bits}/s} = 833\mu\text{s}$$

Equation 7 - Minimum Initiation Interval Decode Thread

The execution time was measured at 6.06µs, giving a large amount of time between completion and the task's deadline.

In reality, the thread's initiation interval is limited by the time taken for the commands to be entered, which, for the shortest command, is limited to about 0.5s.

Print Thread

The print thread is responsible for outgoing communications over the serial port. The minimum time between initiations occurs with the shortest possible outgoing message (15 characters). This gives the minimum initiation interval:

$$\tau = \frac{15 * 8\text{ bits}}{9600\text{ bits}} = 12.5\text{ms}$$

Equation 8 - Minimum Initiation Interval Print Thread

In reality, the print thread is not active so frequently, and is actually called approximately once every second to print the velocity, giving a more realistic value $\tau = 1s$.

The execution time for the longest message was measured equal to 10.6µs.

Critical Instant Analysis

Critical instant analysis assumes the circumstance in which all tasks are initiated at the same time. The task with the longest initiation interval is the print thread with $\tau = 1s$. In this time the motor control thread will be initiated 10 times, and the decode thread will be initiated twice.

The sum of all the execution times is given by:

$$(1 * 10.6\mu) + (10 * 6m) + (2 * 6.06\mu) = 0.06s < 1s$$

Equation 9 - Critical Instant Analysis

This shows that the system is capable of meeting its deadlines in all possible scenarios.

CPU Utilisation

The total CPU utilisation is given by:

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

Equation 10 - CPU Utilisation

For this system, the theoretical utilisation is estimated at $U = 6\%$. For the actual CPU utilisation, the maximum and minimum hash rate was obtained:

Min Hash Rate	5135
Max Hash Rate	5373

TABLE 1 - MIN AND MAX HASH RATE

The hash rate was then obtained with all motor tasks disabled at 5600. An approximate CPU utilisation can then be obtained:

$$\frac{5600 - 5373}{5600} < U < \frac{5600 - 5135}{5600}$$

$$4\% < U < 8\%$$

EQUATION 11 - MEASURED CPU UTILISATION