Imperial College London

ELEC96018 Embedded Systems

Real Time System

Coursework2: Motor Controller

Shen, Leyang ls2617@ic.ac.uk

Lu, Chang cl6417@ic.ac.uk

Zhao, Xuying xz7817@ic.ac.uk

Komal, Kaur kk4917@ic.ac.uk

March 27, 2020

Contents

1 Architecture Overview

While designing this project, message passing is used instead of using shared memory whenever possible. This avoids frequent lock operation (such as mutex) and unnecessary checks for unchanged shared variable.

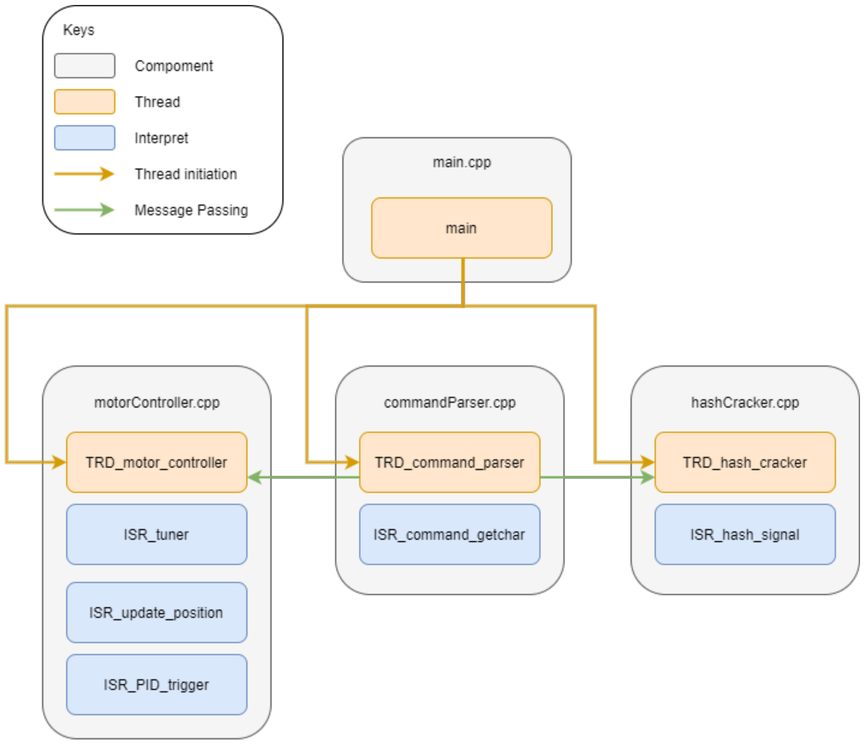


Figure 1: Architecture Overview

1.1 Command Parser

Interrupt **ISR_command_getchar** is used to retrieve user's input from serial port, and place them in a circular buffer. When a $\backslash r$ is detected, it will send a flag to Thread **commandParser**, requesting the thread to parse one command from circular buffer.

1.2 Motor Controller

Interrupt **ISR_tuner** works independently to set motor PWM period, allowing motor to make sound, it also sets callback for itself with duration being its timeout.
Interrupt **ISR_update_position** gets triggered when encoder changes state, it updates *accPosition* which stores the accumulated encoder position change after thread has started. Interrupt **ISR_PID_trigger** gets triggered every 0.1 seconds, it will record the current *accPosition* as *motorPostion*, update *motorVelocity* and set flags allowing **TRD_motor_controller** to calculate torque. Thread **TRD_motor_controller** is used to handle updates on *motorCfg*, and heavy lifting of PID calculation for both motor speed and target rotation.

1.3 Hash Cracker

Thread **TRD_hash_cracker** is computing 5000 hashes per second with different *nonce*, and report back *nonce* if its hash starts with 16 zeros.
The speed of calculation is controlled by setting **SIGNAL_HASH_TICK** flag every second by calling Interrupt **ISR_hash_signal** using *hashTicker*.

2 Dependency Analysis

As shown in figure 1, the system has two data dependencies across the components, which is used to signal update on shared-variable **motorCfg** and **hashKey**. Both are protected by mutex to prevent the producer (TRD_command_parser) write and consumer (TRD_motor_controller and TRD_hash_cracker) read at the same time, allowing atomic R/W operation.

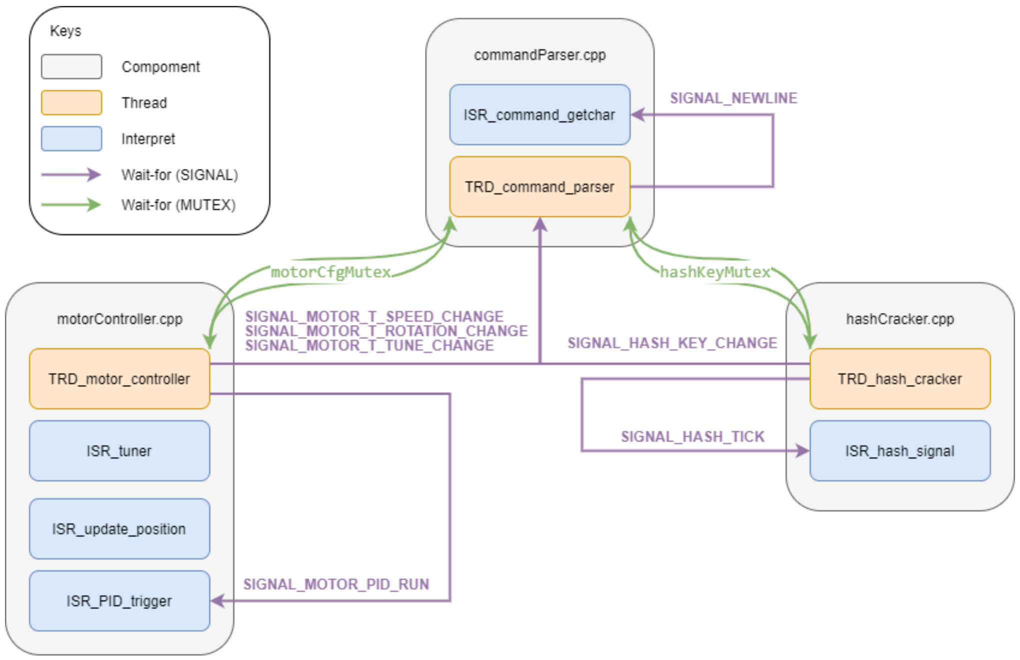


Figure 2: Wait-For Graph

Dependency Name	Type	Blocking	Released by
SIGNAL_NEWLINE	flags	TRD_command_parser	ISR_command_getchar
SIGNAL_MOTOR_T_SPEED_CHANGE	flags	TRD_motor_controller	TRD_command_parser
SIGNAL_MOTOR_T_ROTATION_CHANGE	flags	TRD_motor_controller	TRD_command_parser
SIGNAL_MOTOR_T_TUNE_CHANGE	flags	TRD_motor_controller	TRD_command_parser
SIGNAL_MOTOR_PID_RUN	flags	TRD_motor_controller	ISR_PID_trigger
SIGNAL_HASH_KEY_CHANGE	flags	TRD_hash_cracker	TRD_command_parser
SIGNAL_HASH_TICK	flags	TRD_hash_cracker	ISR_hash_signal
motorCfgMutex	mutex	TRD_motor_controller	TRD_command_parser
motorCfgMutex	mutex	TRD_command_parser	TRD_motor_controller
hashKeyMutex	mutex	TRD_hash_cracker	TRD_command_parser
hashKeyMutex	mutex	TRD_command_parser	TRD_hash_cracker

Table 1: Blocking Dependencies Between Tasks

Table 1 shows blocking dependencies between tasks, and Figure 2 shows the wait-for graph. **TRD_motor_controller** is shown as blocking by four different flags, as soon as one flag is set, the **TRD_motor_controller** will continue to handle change caused by that flag. If multiple flags are set then all set flags will be processed in one go. The same applies for **TRD_hash_cracker** with two flag inputs.

Note that mutex *motorCfgMutex* and *hashKeyMutex* from **commandParser** cause a cycle in wait-for graph, meaning a deadlock is theoretically possible. However, since the mutex is used to only ensure atomic R/W, i.e. they only get locked for negotiable period in TRD_motor_controller and TRD_hash_cracker, the deadlock would not happen in practise. In the futures, these mutex can be completely removed, if proven read for 64bit, 32bit and 8 bit are atomic by the CPU ISA.

3 Time Analysis

3.1 minimum initiation interval

The definition of minimum initiation interval is the time the function takes before it can accept the inputs. It can be obtained from the analysis of code or calculation. Table 2 shows the initiation interval for each task.

Task Name	initiation interval / s
Hash Cracker	1
PID Controller	0.1
Position Update	0.000166666667
Tuner	0.125

Table 2: Minimum Initiation Interval For Task

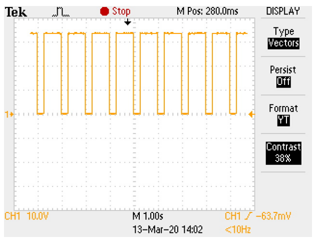
As discussed in the report above, the thread for Hash Cracker computes 5000 hashes per second, indicating the initiation interval is 1 second. For the PID controller, the interrupt is ticked using a Ticker every 100 milliseconds. The minimum duration for the notes for melody is 0.125 second. For the position update, the maximum speed of the motor is 100 rotation per second. However, one rotation is corresponding to one rising edge and one falling edge for three pins which will all trigger the interrupt **ISR_update_position**. As a result, it runs 600 (3*2*100) times in 1 second.

3.2 maximum execution time

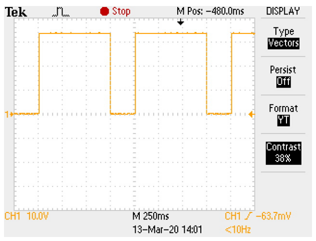
The maximum execution time is measured by setting one of the free pins high at the beginning of each task and setting it back to low at the end of the task. The time is measured through oscilloscope and the function **Persistence** is used to track all the wave for each task. Table 3 shows the measured execution time.

Task Name	maximum execution time / s
Hash Cracker	0.75
PID Controller	0.000021
Position Update	0.000002
Tuner	0.000032

Table 3: maximum execution time For Each Task



(a) Hash Cracker



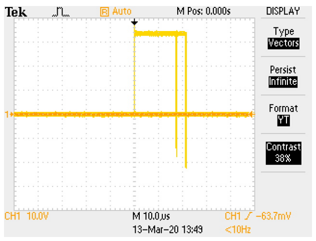
(b) Detailed Hash Cracker

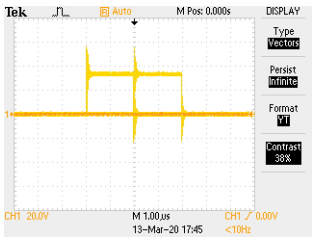
Figure 3: Hash Cracker

Figure 3 shows the period for the Hash Cracker and the time for the signal to be high is 0.75 second.

Figure 4(a) shows the latency for the task PID Controller. The maximum time period is 21 us.

As can be shown in Figure 4(b) and 4(c), the execution time for Position Update and Tuner are 2 us and 32 us respectively.


(a) PID Controller


(b) Position Update

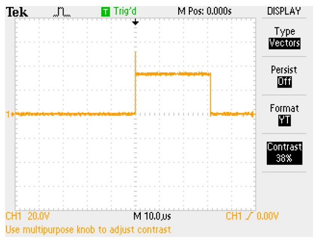

(c) Tuner

Figure 4: Other Execution Time

3.3 maximum CPU utilisation

CPU utilisation is the proportion of time that CPU is busy. It can be calculated using the formula:

$$U = \frac{T}{\tau}$$

where U is the CPU utilisation, T denotes the execution time while τ denotes the initiation interval. Table 4 shows the data calculated.

Task Name	maximum execution time / s	initiation interval / s	Theoretical CPU utilisation
Hash Cracker	0.75	1	75%
PID Controller	0.000021	0.1	0.02%
Position Update	0.000002	0.00167	0.12%
Tuner	0.000032	0.125	0.04%
Total			**75.18%**

Table 4: Calculated CPU Utilisation For Each Tasks

For further improvement, the exact measurements are taken by using the function **print_cpu_stats** to record the change of UP time and the change of Idel time. The formula for the real CPU utilisation is:

$$U = 1 - \frac{\Delta Idletime}{\Delta UPtime}$$

The statistics for each task are shown in Table 5 and it is similar to the calculated CPU. Noticed that there are still some difference between the real CPU utilisation and the expected CPU utilisation. The reason to this is the inaccurate measurement on the oscilloscope.

Task Name	CPU time initial	CPU time end	Idel initial	Idel end	Measured CPU utilisation
Hash Cracker	579	60055084	0	16054869	73.266%
PID Controller	610	60055084	0	59913242	0.23%
Position Update	610	60055084	0	59970765	0.139%
Tuner	640	60055084	0	59985114	0.154%
Total					**73.789%**

Table 5: measured CPU utilisation For Each Tasks

3.4 critical instant analysis

The longest initiation interval is 1 second which comes from Hash Cracker. As a result, during this 1 second interval, Task PID controller can be done 10 times, Task Tuner can be done 8 times while for Position Update is 600 times. The analysis is:

$$Time = 0.75 + 0.000021 \times 10 + 0.000002 \times 600 + 0.000032 \times 8 = 0.751666 < 1$$

4 Motor Control Characteristics

In order to control parameters such as the position and velocity, it is necessary to know which state the rotor is in, which is done through the three photo-interrupter LEDs that indicate the position of the disc. Once the state of the rotor is known, it is possible to alter the position by creating a torque which can be varied by the duty cycle of the PWM cycle. The motor torque is created by the dis-alignment of the magnetic field generated by the motor windings compared to that of the permanent magnet of the spindle.

4.1 Position Control

The **ISR_update_position** interrupt updates the accumulated position variable when the encoder has changed states, which allows the velocity and the distance to be calculated. Position Control consists of a **setTorque** function which changes the PWM cycle duty width and also includes a safety check that ensures the motor is moving in the correct direction by checking the rotor state update and consequently updating the lead. If the distance command has been set the **Signal_Motor_PID_Run** flag will be set and the error between the current position and the final position is calculated and differentiated with respect to time, in order to solve the following equation so that the torque can be set by

$$y_r = k_p Er + k_d \frac{dE_r}{dt} \tag{1}$$

It was experimentally found that the distance is reached better without the use of a differentiator and with the $k_p = 0.1$. The position controller is necessary to slow down the velocity to a stop when the motor is about to reach the distance.

4.2 Speed Control

In order to control the velocity, it is important to be able to change the direction of the motor if a command has been set that changes the said parameter in the opposite direction. After taking this into account, a PID controller has been implemented in order to have better and more precise control of the velocity. The PID parameters have been experimentally chosen to be $k_p = 0.1, k_i = 0.05, k_d = 0$ with an integral cap of 25 to ensure a low steady state error. The torque is then set based on the decision of choosing to implement position control or velocity control. The decision is made by y = max(y_s, y_r) if $v < 0$, otherwise y = max(y_s, y_r) if $v \geq 0$. In a way, the torque calculated for the position control is prioritised over the torque calculated for the speed control. One example to explain this would be when the distance is near the target distance, the position torque (the torque calculated for the position) would be lower than the speed torque (the torque calculated to maintain the speed) and the position torque will be chosen to ensure that the target distance is reached.

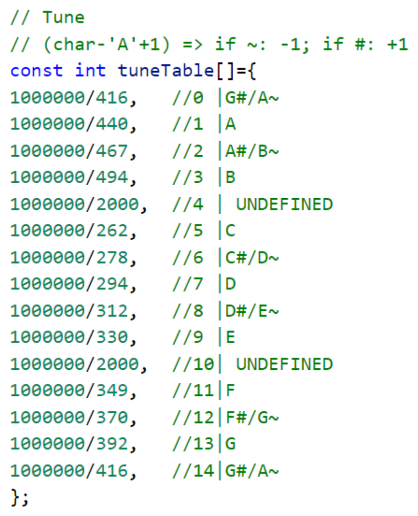
The current configuration of $k_p = 0.1, k_i = 0.05, k_d = 0$ is able to keep the velocity constant for a majority of the time however it does oscillate by a value of 1 at certain times. To combat this error, k_d was increased until the system was critically damped, however in practice it is very difficult to reach zero oscillations in steady state. So after much experimentation, it was found that even after the system was critically damped and had become over-damped, the steady state value would always have oscillations of around 1. And the higher k_d became, the lower the steady state value became, and so it was realised that k_d was unnecessary for this task.

5 Appendix

5.1 Data Compression on Melody Command

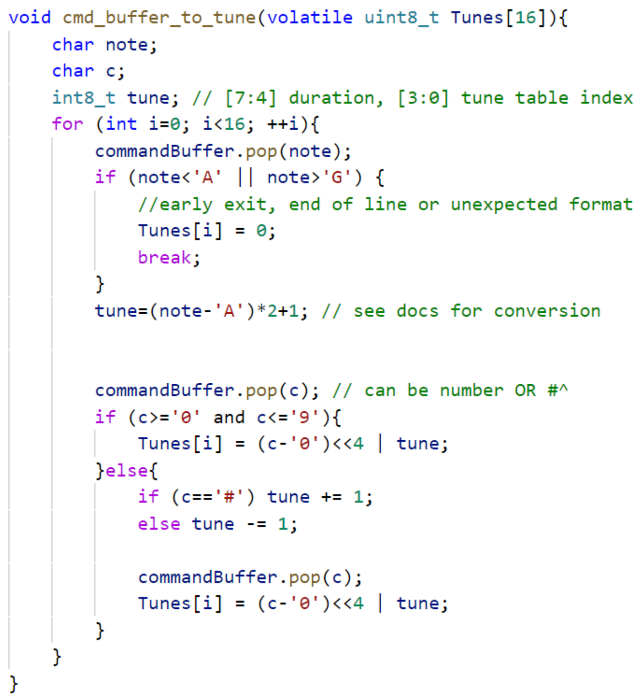
The regular expression for melody, contains redundant information, which can be compressed to reduce the size of this shared variable.

The result of our compression is a array of 16 element each with 8 bit. The [3..0] bit used to store the index of the melody lookup table, the [7..4] bit stores duration of the melody with factor of 0.125 second. Duration with 0 has a special meaning as terminator, indicating the end of the melody sequence.



```
// Tune
// (char-'A'+1) => if ~: -1; if #: +1
const int tuneTable[]={
1000000/416,   //0 |G#/A~
1000000/440,   //1 |A
1000000/467,   //2 |A#/B~
1000000/494,   //3 |B
1000000/2000,  //4 | UNDEFINED
1000000/262,   //5 |C
1000000/278,   //6 |C#/D~
1000000/294,   //7 |D
1000000/312,   //8 |D#/E~
1000000/330,   //9 |E
1000000/2000,  //10| UNDEFINED
1000000/349,   //11|F
1000000/370,   //12|F#/G~
1000000/392,   //13|G
1000000/416,   //14|G#/A~
};
```

(a) Melody LookUp Table



```
void cmd_buffer_to_tune(volatile uint8_t Tunes[16]){
    char note;
    char c;
    int8_t tune; // [7:4] duration, [3:0] tune table index
    for (int i=0; i<16; ++i){
        commandBuffer.pop(note);
        if (note<'A' || note>'G') {
            //early exit, end of line or unexpected format
            Tunes[i] = 0;
            break;
        }
        tune=(note-'A')*2+1; // see docs for conversion


        commandBuffer.pop(c); // can be number OR #^
        if (c>='0' and c<='9'){
            Tunes[i] = (c-'0')<<4 | tune;
        }else{
            if (c=='#') tune += 1;
            else tune -= 1;

            commandBuffer.pop(c);
            Tunes[i] = (c-'0')<<4 | tune;
        }
    }
}
```

(b) Code For Compress The Melody Sequence

Figure 5: Hash Cracker

Figure 5a shows the look up table being used, and figure 5b shows the compression step.