Imperial College London

ELEC96018 Embedded Systems

Real Time System

Coursework2: Motor Controller

Shen, Leyang	Lu, Chang	Zhao, Xuying	Kaur,Komal
ls2617@ic.ac.uk	cl6417@ic.ac.uk	xz7817@ic.ac.uk	kk4917@ic.ac.uk

March 27, 2020

Contents

1 Architecture Overview

While designing this project it was decided that message passing would be used, instead of using shared memory, whenever possible. This avoids frequent lock operations (such as mutex) and unnecessary checks for unchanged shared variables, *osPriority* is used to help scheduler decide.

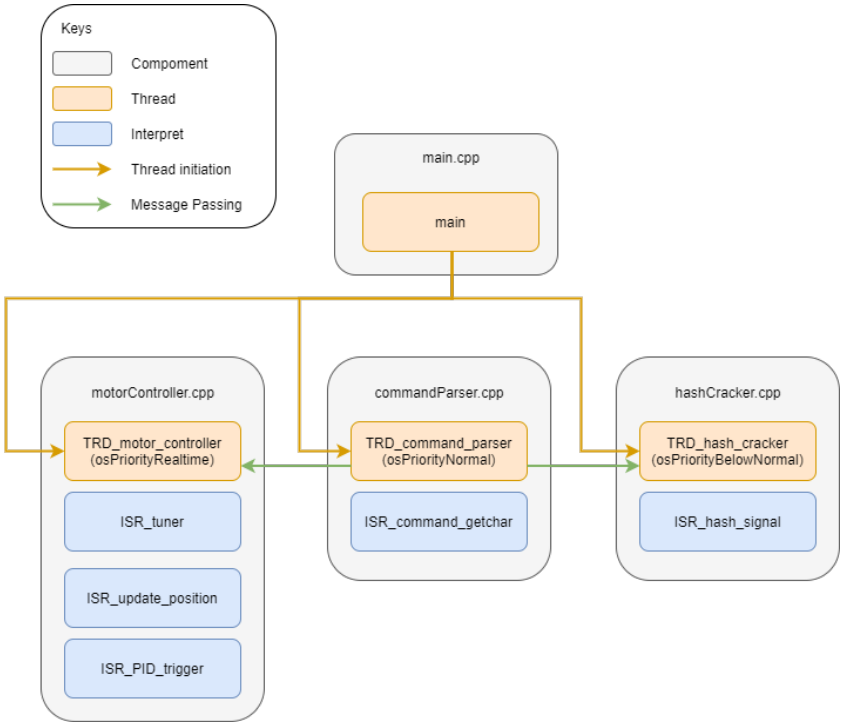


Figure 1: Architecture Overview

1.1 Command Parser

Interrupt **ISR_command_getchar** is used to retrieve the user's input from the serial port, and place it into 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 the circular buffer.

1.2 Motor Controller

Interrupt **ISR_tuner** works independently to set the duty cycle of the motor's PWM wave, allowing the motor to produce a sound. The interrupt also sets a callback for itself, where the duration is its timeout.
Interrupt **ISR_update_position** gets triggered when the encoder changes state, it updates *accPosition* which stores the accumulated encoder position change after the thread has started. Interrupt **ISR_PID_trigger** gets triggered every 0.1 seconds, it records the current *accPosition* as *motorPostion*, updates *motorVelocity* and set flags allowing **TRD_motor_controller** to calculate the necessary torque. Thread **TRD_motor_controller** is used to handle updates on *motorCfg*, and also does the heavy lifting of the PID calculations for both the motor speed and the target rotation.

1.3 Hash Cracker

Thread **TRD_hash_cracker** is computing 5000 hashes per second with different *nonce*, and reports back *nonce* if its hash starts with 16 zeros.
The speed of calculation is controlled by setting the **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 updates on shared-variables **motorCfg** and **hashKey**. Both are protected by mutex to prevent the producer (TRD_command_parser) writing and consumer (TRD_motor_controller and TRD_hash_cracker) reading at the same time, allowing atomic R/W operation.

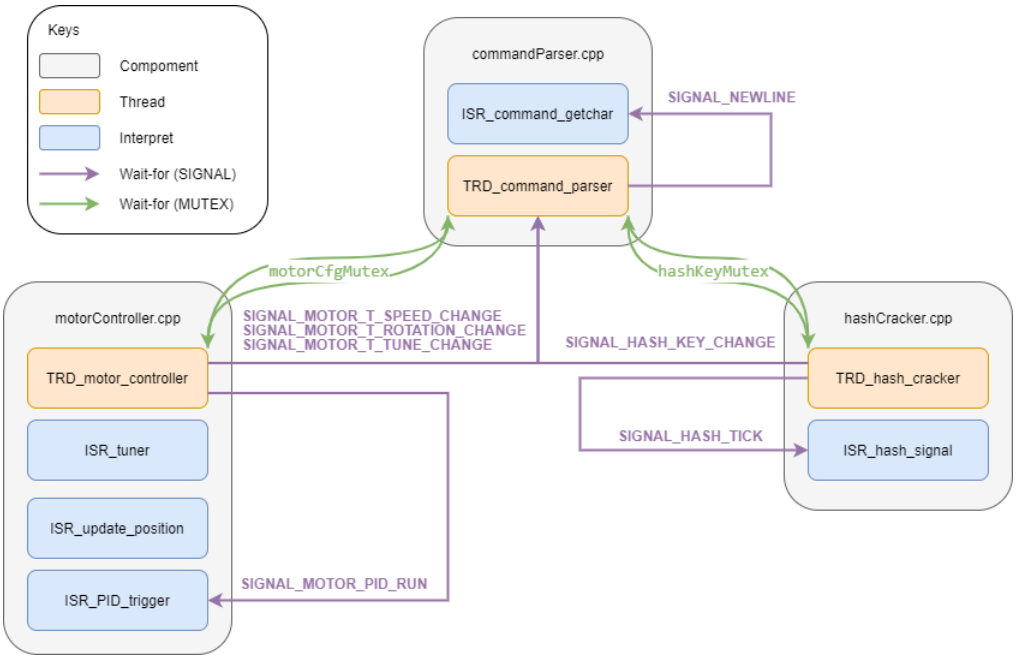


Figure 2: Wait-For Graph

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 as shown is blocked 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 to **TRD_hash_cracker** with two flag inputs.
Note that mutex *motorCfgMutex* and *hashKeyMutex* from **commandParser** cause a cycle in the wait-for graph, meaning a deadlock is theoretically possible. However, since the mutex is used to only ensure atomic R/W, i.e. once a thread acquire the mutex, it will finish its read operation in short interval (without need to obtain further locks) and release the lock, meaning deadlock would not happen. In the future, these mutexs can be completely removed, if proven read for 64bit, float and 8 bit data are all atomic.

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 [1]
PID Controller	0.1 [2]
Position Update	0.000166666667 [3]
Tuner	0.125 [4]

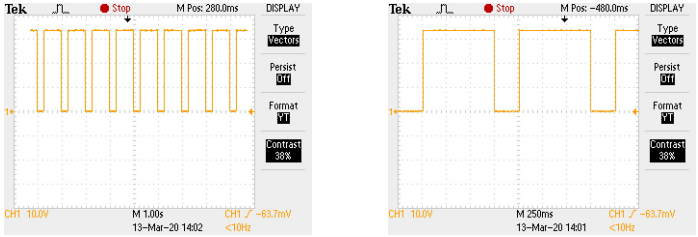
Table 2: Minimum Initiation Interval For Task

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 the oscilloscope and the function **Persistence** is used to track the wave signals 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 time period of the Hash Cracker, the duration of assertion (high) is 0.75 second.

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

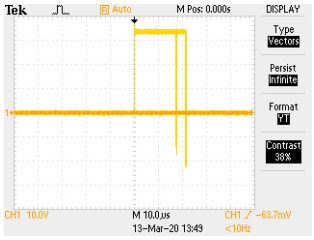
As shown in Figure 4(b) and 4(c), the execution time for Position Update and Tuner are 2 us and 32 us respectively.

[1] Hash Cracker computes 5000 hashes per second, indicating the initiation interval is 1 second.

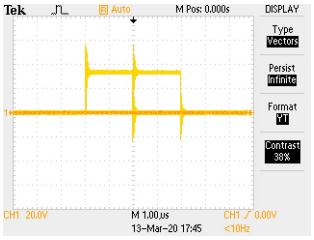
[2] The interrupt is ticked using a Ticker every 100 milliseconds.

[3] 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.

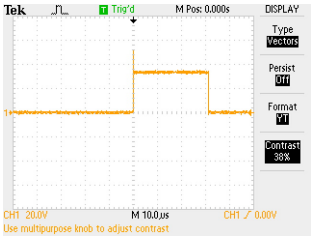
[4] The minimum duration for the notes for melody is 0.125 second.



(a) PID Controller



(b) Position Update



(c) Tuner

Figure 4: Other Execution Time

3.3 Maximum CPU Utilisation

CPU utilisation is the proportion of the time that the 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	0.03%
Total Utilisation			**75.17%**

Table 4: Calculated CPU Utilisation For Each Tasks

To verify the theoretical calculation, the exact measurements are taken by using the function **mbed_stats_cpu_get** to record the UP time and the Idel time. The formula for the real CPU utilisation is:

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

Task Name	CPU time initial	CPU time end	Idel initial	Idel end	Measured CPU utilisation
Hash Cracker	579	60055084	0	16054869	73.27%
PID Controller	610	60055084	0	59913242	0.24%
Position Update	610	60055084	0	59970765	0.14%
Tuner	640	60055084	0	59985114	0.12%
Total Utilisation					**73.76%**

Table 5: measured CPU utilisation For Each Tasks

By comparing Table 4 and Table 5, *Measured CPU utilisation* (measured via software) is close to *Theoretical CPU utilisation* (measured via scope).
For the *PID Controller* task, the *Position Update* and *Tuner*, has a *Measured CPU utilisation* slightly higher than *Theoretical CPU utilisation*, this could be caused by the overhead of software measurement.
However, for the *Hash Cracker* task, the *Measured CPU utilisation* is lower than *Theoretical CPU utilisation*. This is because the *Theoretical CPU utilisation* measured from the scope has been taking the maximum execution time into account in the measurement, and this is when a large proportion of nonce is being printed. But in practice, such a case is unlikely to happen.

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 run 10 times, Task Tuner can run 8 times while Task Position Update can run 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 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 altering the duty cycle of the PWM wave. 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.

The **ISR_update_position** interrupt copies accPosition to the motorPosition, and uses the previous motorPosition to calculate motorVelociy. The motorPosition and motorVelociy is similar to a snapshot of the motor's information when this ISR is called.

A helper function called **setTorque** is used to change the motor's direction via lead and speed by altering the duty cycle of the PWM. It also includes a safety guard to kick start the motor if a torque is set but the motor is not turning.

4.1 Position Control

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 $k_p = 0.1$. The position controller is necessary to slow down the velocity to a stop when the motor is about to reach the distance.

In addition, to make sure that the torque is zero when the error in the position is less than 0.5, $y_r = k_p Er + k_d \frac{dE_r}{dt}$ if $|errorPosition| > 2$ otherwise $y_r = 0$.

4.2 Speed Control

In order to reach targeted position, it is important ensure motor rotation in a direction which minimise the position error, furthermore 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.

This configuration 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.

The speed controller is constructed in such a way that if the motor is still far from reaching the target distance then $E_s = s - v$ (where s is the target speed and v is the measured speed), and the integral term in the PID controller will be updated. On the other hand, if the motor is approaching the target distance, then $E_s = approach\_v - v$ where approach_v is a lower value (10 rotations per second) so that the motor can be slowed down and the error Speed integral will be zero (since k_i is only used for steady state stability).

4.3 Combined Position and Speed Control

Once the position and speed control have been combined, the torque is 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 would be lower than the speed torque and the position torque will be chosen to ensure that the target distance is reached.

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 an array of 16 elements each with 8 bit. The [3..0] bits are used to store the index of the melody lookup table, the [7..4] bits store 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.