Light Controller FreeRTOS

University of Aveiro

Diogo Marques, Marco Silva

Light Controller FreeRTOS

Department of Electronics, Telecommunications and Informatics

University of Aveiro

Diogo Marques, Marco Silva
(83639) d.marques@ua.pt, (84770) masilva@ua.pt

Chapter 1

Introduction

This document approaches the implementation of a light controller based on the PIC32 micro-controller, the Max32 board and using the real-time computation model. The micro-controller firmware was implemented with the FreeRTOS real-time kernel. The light controller receive inputs from a UART, connected to the Max32 board through USB, switches, push-buttons and a Light Dependent Resistor (LDR), and actuate on a LED.

The controller have several modes of operation and can be selected either at the terminal interface or by two switches, the current mode is permanently displayed at the terminal. The four modes of the system are:

- Manual Mode : the light is either on or off depending on user inputs;
- Dimmer Mode : the light intensity can be manually adjusted by pressing the "+" and "-" keys or by the two push-buttons;
- Automatic switch : the light should go on and off depending on the information from the light sensor;
- Automatic adjust : the controller adjust continuously the light intensity, so that the light level detected by the sensor remains constant.

Since it is a real-time application, the system must ensure a correct operation in the logic and time domains in order to ensure that the system can execute in the right time. Analyzing the task characteristics as the response time is necessary demonstrate that the system is schedulable, satisfying its requirements.

Chapter 2

Architecture

The system consists in nine tasks that communicate with each other via shared memory using semaphores to ensure exclusive access and in one case, a queue is used to ensure communication and synchronization between tasks.

2.1 Tasks

In the tasks that constitute the system, the task prints and actuation are used as a data output of the system, interact with the system's user controlling the led and the information that appears in the terminal. The decision task is the "brain" of the system and the remaining tasks have the job of obtaining inputs from the user and the environment. The Figure 2.2 represents the system architecture where it is possible to verify which task access to the shared memory and which are the inputs. The only task that does not access to the shared memory is the actuation task since receives the data through a queue.

The interface tasks are separated since it makes perfect sense for each to have an independent period as they interact with different components. They also have the same priority so that none of them is interrupted because it is considered that they all have the same importance.

2.1.1 Buttons Interface - btInt

This task read the buttons and update light intensity according to its state. It is a periodic task and has a priority of 2 and a period of 150 ms once a human will not press the button more than 6 times per second.

2.1.2 Switches Interface - swInt

This task read the switches and can update the mode or the on/off variable. Like the previous task it has a priority of 2, but the period is 350 ms because the speed of interaction is lower.

2.1.3 Keyboard Interface - keyInt

The keyboard interface task has the same objective of receiving information from the user, but in this case through the keyboard/terminal. It has a period of 100 ms because the user will not press more than 10 keys per second.

The user interacts with the system as follows:

- Press "1" - Change to mode 1;
- Press "2" - Change to mode 2;
- Press "3" - Change to mode 3;
- Press "4" - Change to mode 4;
- Press "A" - Disables the Switches Modes;
- Press "T" - Turn the LED ON and OFF;
- Press "Y" - Disables the Switches On/Off;
- Press "+" - Increase light intensity;
- Press "-" - Decrease light intensity;

2.1.4 Sensor acquisition - sensorAcq

Is a periodic task, with 100 ms period, that samples the light sensor with an ADC. The priority is also 2.

2.1.5 Decision - decision

All previous tasks received user input, however in the decision this does not happen anymore. For each mode, the value of the light intensity is calculated and if it should be ON or OFF according to the values that are configured. The task accesses the values through shared memory and is periodic with a period of 80 ms in order to ensure that all changes made are considered.

This task has a higher priority because there are multiple interface tasks that are only used depending on the current mode and it is considered that calculate the correct value to the light is essential in the system.

2.1.6 Actuation - actuation

The actuation task is sporadic with a minimum inter-arrival time of 80 ms because in the worst case it runs whenever the decision runs. This task takes as input the result of the Decision task 2.1.5 and sets the right light intensity. For the same reason has a priority of 3.

2.1.7 Prints - prints

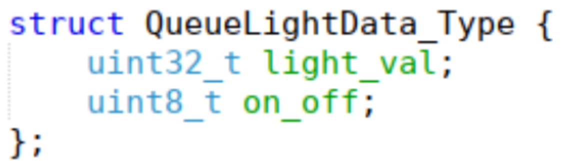
The last task of the system is the one that prints in the terminal the current state. Is periodic and has the highest period of the system, 500 ms. Having the lowest priority of tasks, we consider it as the less important job.

2.1.8 Interruptions

To control the LED intensity we use a PIC32 Timer and the Output Compare module to generate a PWM signal. Using interrupts, they occur at such a high frequency compared to the rest of the tasks, they are consider as a task with priority 4 for the calculation of response times.

2.2 Inter Process Communication and Synchronization

To all the tasks be able to communicate between them, an Inter Process Communication (IPC) was needed. In this system two types of **IPC!** (**IPC!**) were used: global variables and a queue. The first is used across most task and allows that tasks like the Keyboard Interface or the Switches Interface change the same variable, always with exclusive access controlled with the use of mutex semaphores, which allow the priority inheritance mechanism and allow tasks that don't need to be blocked to continue their execution. The second one, that is, the queue, which has a structure as seen in Figure 2.1, allows both communication and synchronization between the Decision and Actuation tasks, making that the Actuation task only runs when there is information in queue, which happens only when the Decision puts it there. This synchronizes both and makes the Actuation always run after the Decision.



```
struct QueueLightData_Type {
    uint32_t light_val;
    uint8_t on_off;
};
```

Figure 2.1: Queue Structure

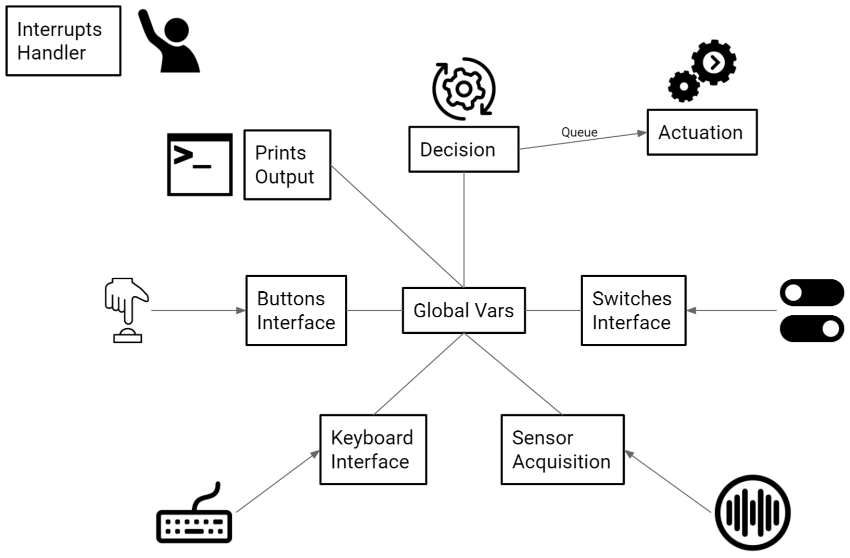


Figure 2.2: System Architecture

Chapter 3

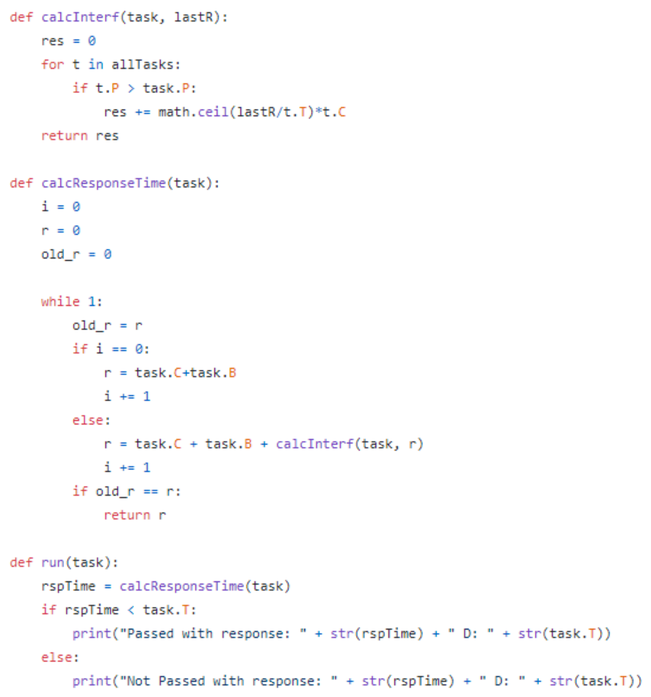
Real Time Behaviour

In order to understand the system real-time behaviour, it was necessary, above all, to measure each task execution time. To do this, a completely clean new project was made, without FreeRTOS and a hardware timer was used to count the number of cycles each task took to execute in the worst case scenario. The value was then converted to milliseconds by dividing it by the peripheral bus clock, obtaining the result in seconds, and then multiply it by 1000, converting to the desired milliseconds. Each task was run 200 times in its worst case scenario in order to get an average worst case time.

Besides this, for each task, the blocking time was measured as well. With this two values and the corresponding execution time, it was possible to apply the Equation 3.1, obtaining the response time for each one of the tasks always considering its worst case scenario.

To apply this equation, the python script, from Figure 3.1, was developed and, using the $C, B, D = T$ and *Priority* values from Table 3.1, produced the R values that can be seen in that same Table 3.1, as well as, the outputs that can be seen in Figure 3.2, showing that every task had a Worst Case Response Time (WCRT) lower that its deadline.

$$\begin{cases} R_i^{(0)} = C_i + B_i, & \text{if } s = 0 \\ R_i^{(s)} = C_i + B_i + \sum_{h:P_h>P_i} \left\lceil \frac{R_i^{(s-1)}}{T_h} \right\rceil C_h, & \text{if } s > 0 \end{cases} \tag{3.1}$$



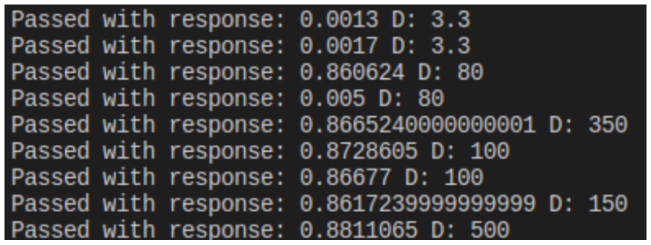
```python
def calcInterf(task, lastR):
    res = 0
    for t in allTasks:
        if t.P > task.P:
            res += math.ceil(lastR/t.T)*t.C
    return res

def calcResponseTime(task):
    i = 0
    r = 0
    old_r = 0

    while 1:
        old_r = r
        if i == 0:
            r = task.C+task.B
            i += 1
        else:
            r = task.C + task.B + calcInterf(task, r)
            i += 1
        if old_r == r:
            return r

def run(task):
    rspTime = calcResponseTime(task)
    if rspTime < task.T:
        print("Passed with response: " + str(rspTime) + " D: " + str(task.T))
    else:
        print("Not Passed with response: " + str(rspTime) + " D: " + str(task.T))
```

Figure 3.1: Python script



```
Passed with response: 0.0013 D: 3.3
Passed with response: 0.0017 D: 3.3
Passed with response: 0.860624 D: 80
Passed with response: 0.005 D: 80
Passed with response: 0.8665240000000001 D: 350
Passed with response: 0.8728605 D: 100
Passed with response: 0.86677 D: 100
Passed with response: 0.8617239999999999 D: 150
Passed with response: 0.8811065 D: 500
```

Figure 3.2: Python Script Equation Results

Task	C **Execution Time (ms)**	R **Response Time (ms)**	$D = T$ **Deadline (ms)**	B **Blocking Time (ms)**	**Priority**
Timer 2 Interrupt	0.0013	0.0013	3.3	0	4
Output Compare Interrupt	0.0017	0.0017	3.3	0	4
Actuation	0.0020	0.0050	80	0	3
Decision	0.1015	0.8606	80	0.7561	3
Sensor Acquisition	0.0113	0.8729	100	0.7550	2
Keyboard Interface	0.0050	0.8668	100	0.7552	2
Buttons Interface	0.0017	0.8617	150	0.7535	2
Switches Interface	0.0040	0.8665	350	0.7560	2
Prints	0.7524	0.8811	500	0	1

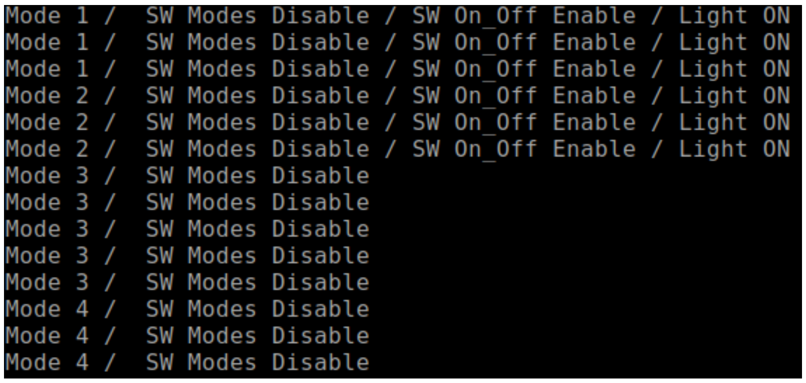
Table 3.1: Response Time Analysis

Chapter 4

Results

As said in chapter 3, the python script showed that every task has an WCRT lower than its deadline, which is equal to its period. This means that each task will complete its execution, even in the worst case, before reaching its deadline. With this in mind, it can be concluded that the system is schedulable and, can operate in real time without the risk of not meeting a deadline.

In Figure 4.1 one can see an example of the system output during execution. In order to verify if the system is fully functional many tests were performed during its execution, in all 4 modes, without ever finding any deficiencies, and so, it can be concluded that the system is working perfectly in real time, with all its tasks working in harmony and meeting their deadlines.



```
Mode 1 /  SW Modes Disable / SW On_Off Enable / Light ON
Mode 1 /  SW Modes Disable / SW On_Off Enable / Light ON
Mode 1 /  SW Modes Disable / SW On_Off Enable / Light ON
Mode 2 /  SW Modes Disable / SW On_Off Enable / Light ON
Mode 2 /  SW Modes Disable / SW On_Off Enable / Light ON
Mode 2 /  SW Modes Disable / SW On_Off Enable / Light ON
Mode 3 /  SW Modes Disable
Mode 3 /  SW Modes Disable
Mode 3 /  SW Modes Disable
Mode 3 /  SW Modes Disable
Mode 3 /  SW Modes Disable
Mode 4 /  SW Modes Disable
Mode 4 /  SW Modes Disable
Mode 4 /  SW Modes Disable
```

Figure 4.1: System Execution Example

Chapter 5

Conclusions

In conclusion, the developed system made for the PIC32 micro-controller is schedulable, with a big margin in each task to meet its deadline. With this in mind, one could save some power by reducing the micro-controller clock frequency, or switch to a cheaper and slower one and still be able to use the system in real time with all tasks meeting their deadlines.

Acronyms

WCRT Worst Case Response Time

LDR Light Dependent Resistor

IPC Inter Process Communication