Project Part 2

FreeRTOS: Multi-Function Device

Real Time Systems

Group nr.:	1
Student-ID-1: ist1112497	Kristian Guttormsen Gøystdal
Student-ID-2: ist1112398	João Pedro Cabral Miranda
Student-ID-3: ist1112548	Einar Bergslid

Electrical and Computer Engineering
IST-ALAMEDA

2024

1 Introduction

This project focuses on the development of a real-time embedded system for a "Multi-Function Device" using the multitasking kernel (operating system) FreeRTOS. The project is implemented on the microcontroller NXP LPC1768 with the development board mbed LPC1768. The project emphasizes the use of concurrent processes, shared variables and scheduling. The application is programmed and compiled using the mbed online development environment Keil Studio Cloud.

2 Code structure

The project is divided into nine different tasks with different functionality that run concurrently. Communication between tasks is done through a series of queues (when a value needs to be delivered) and tasks notifications, as well as semaphores for synchronization of shared variables.

Most of the user input occurs through the console implemented with a serial port with baud rate 115200. The task monitor is responsible to read the user input and execute the corresponding action. Since it manages user iterations using a busy-wait mechanism, it is assigned the lowest priority in the project, with priority 1 (priority 0 being reserved for the idle task). The maximum number of priorities is set to 5 as this allows the different tasks to run and preempt other tasks when required for the wanted functionality.

About the boards peripherals, the tasks LCD, RGB and Temperature are responsible for managing the LCD, RGB LEDs, Temperature Sensor, respectively. In addition, HitBit Task manages the JoyStick and the four blue LEDs and ConfigSound Task manages the buzzer and both potentiometers.

Some tasks also uses software timers to perform periodic operations. Note that these software timers are assigned priority two, ensuring they never interrupt application tasks, except for the Monitor task, which has a priority of three or four.

3 Requirements and solutions

The main functions to be implemented include, but are not limited to:

1. **Sensing temperature periodically and on-demand while saving records of**

minimum and maximum values with a timestamp. This is solved by using a specific task to record the temperature and send it to the MaxMin, LCD and Alarm tasks for further action. The temperature is recorded using the temperature sensor LM75B built in to the development board and communication between the sensor and the microcontroller is done using the serial protocol I2C. The task records temperature periodically if the variable PMON is a non-zero value, and may also be triggered to record on-demand if the appropriate message is received from the Monitor task.

2. **Updating a live simulation of a bubble level on an LCD-display using the built in accelerometer.** The Bubble Level task is responsible for detecting the orientation of the development board using the built in accelerometer MMA7660. This only happens if the bubble level is enabled. The recorded values for the x and y axes are scaled appropriately and sent to the LCD task to display the "bubble".

3. **Displaying a clock showing time using the built in RTC.** In this situation, an RTC interrupt occurs every second and sends a message to the LCD to update the displayed time. The LCD retrieves the current time using the time(NULL) function. Since this function is thread-safe, no additional synchronization mechanisms are required.

4. **Managing a console for user interaction, with a given list of possible commands.** The monitor task, with a low priority, solves this by handling user input and executes the provided command. By using a low priority for this task, other tasks will not be interrupted by the user. Some commands executes directly in the command provided by the user, while some requires other tasks to handle the main execution, which is accessed using queues. The commands that require arguments handles these accordingly before passing them along to the appropriate task or function.

5. **Have alarms triggered by time or temperature using the on board buzzer.** To implement this, each temperature measurement is sent to the Alarm Task. If the temperature alarm is enabled and the measured value is outside of the range defined by the low and high thresholds, an alarm is triggered. For the time alarm, the RTC generates an interrupt when the specified time is reached. In both cases, a message is sent to the ConfigSound Task to activate the buzzer.

During the alarm duration (TALA seconds), the buzzer sound will be updated five times per second, if the sound configuration functionality is enabled.

In addition to fulfilling all requirements of functionality, the goal of the project is to create system that takes advantage of the functionality offered by the operating system, FreeRTOS. It is also preferred to use the syntax and naming conventions related to FreeRTOS.

4 Main Issues

This section discusses the main challenges encountered during this project.

The first significant issue we encountered was when we were passing queues as arguments to the tasks. This method made it so the pointers for the queue were altered and resulted in system crashes when tasks were executed. To solve this issue, we implemented global queues, which eliminated the need for arguments in our tasks. This solution made the queues available for all tasks with minimum code.

Another issue that appeared was the frequency of the alarm buzzer not being what we expected. Instead of hearing a continuous buzzing sound when an alarm was triggered, we were hearing a clicking sound between 1 and 10 times per second, depending on where the potentiometer controlling period was set. This turned out to be because we used the value from the potentiometer directly as the period of the PWM signal. The solution was to scale this value to a period in an appropriate audible range. This range was chosen to be 1 to 3 ms, resulting in a frequency range of 333 to 1000 Hz. In addition, the period of the PWM signal for the buzzer was only updated based on the potentiometer when a new alarm is triggered. We implemented a more frequent update of this value. As this was not specified in the project description, we decided that updating the period (and duty cycle) 5 times per second was fitting.

A third issue we had to deal with was that the code crashed when a clock alarm was triggered. The reason for this bug was that we were using a regular system call in an interrupt handler, which entered the blocked state. This of course prohibited, then crashing the entire application. The solution was to use the interrupt specific system call functions.

The next issue we encountered were related to the RTC ranges of the year. In the project description it was specified that the year should be able to be set from 0 to 9999. However, trying to set the year to 0 using the set date command, an error occurred in

the conversion from string to the int stored in the RTC. Further testing concluded with that the minimum year possible for this system was 1970. This was also confirmed by the system, as this was the default year after a reboot. The maximum year was found to be 2106, because with 32 bits only 136 years can be stored. As a solution, we changed the allowed range of the year to be between 1970 and 2106, which ensures proper functionality when changing the date.

During an alarm, sometimes the RGB LED turns white, but it reverts to the correct color when a new temperature measurement is received. This issue occurs because all PWM outputs on the board (the RGB LED and the buzzer) share the same period. When the ConfigSound Task changes the buzzer's period, it inadvertently affects the RGB LED's output. To resolve this, the ConfigSound Task now sends a message to the RGB Task every time it updates the buzzer's period, instructing it to refresh the LED colors. This ensures the RGB LED quickly returns to the correct color, preventing the human eye from perceiving the temporary white flash.

Another important problem was configuring interrupts for Joystick and RTC. To enable these interrupts in the ARM processor's interrupt vector, the NVIC_PRIORITY() function needed to be used. Additionally, the Joystick interrupt was assigned a higher priority (lower numerical value) compared to the RTC interrupt, as handling the Joystick was considered more critical than processing RTC events.

5 Conclusion

The project successfully created a real-time system using FreeRTOS on the NXP LPC1768 microcontroller. Key tasks like temperature monitoring, LCD updates, user interaction, and alarms were implemented. Challenges, such as issues with queues, buzzer frequency, and RTC year range, were solved to ensure smooth operation. The project demonstrates how FreeRTOS can be used effectively for multitasking and communication between tasks in embedded systems.