Lab #1: Intro to Real Time Operating System

Leonardo Fusser (1946995)

Objectives:

- Create a Task in FreeRTOS.
- Set Task Priorities in FreeRTOS.
- Use a delay with and without blocked state in FreeRTOS.

Hardware: simulated.

To hand in:

These sheets including answers to all questions on **Teams**.

C code on **GitHub**:

- Indented and commented multi-source file on Git Hub.
- Relevant screenshots. Must include explanations.
- All functions must have a prolog header.
- All files must have their prolog headers: file name, description, date, author, and version history.
- Use of symbolic constant is mandatory, all macros in capital letters.
- Use preprocessor instruction to enable/disable snippets of code.
- You must use names that are representative for all variables, macros, and functions.
- The software must respect the following qualities: modularity, maintainability (readability, understandability) and portability.
- Screen shots with white background.
- Answer all questions by filling in the lab sheets using a computer. Must be submitted in paper format.

You must follow the following encapsulation and modularization rules:

- Abstract all configuration related code by creating functions or macros.

```
#define NB_BALLS  10
int create_balls(int);
```

If the requirements are not fulfilled, the student will be asked to re-submit.

Reference:

FreeRTOS Reference Manual_unlock.pdf

Lab work:

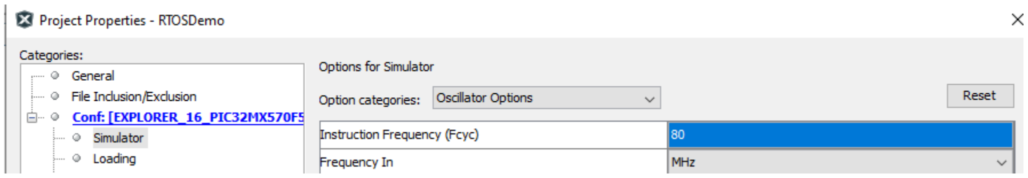
Make sure you are logged into GitHub. Copy-paste the following URL to accept an invitation for this lab:

https://classroom.github.com/a/GRner5Yq

You will get a private new repository for this lab. You will push your project to this repo at the end of this lab

https://github.com/Embedded-OS-Vanier/lab1_intro_rtos-yourName

Set the simulator to 80 MHz:


Project Properties - RTOSDemo

Categories:
- General
- File Inclusion/Exclusion
- Conf: [EXPLORER_16_PIC32MX570F5
  - Simulator
  - Loading

Options for Simulator

Option categories: Oscillator Options

Instruction Frequency (Fcyc)	80
Frequency In	MHz

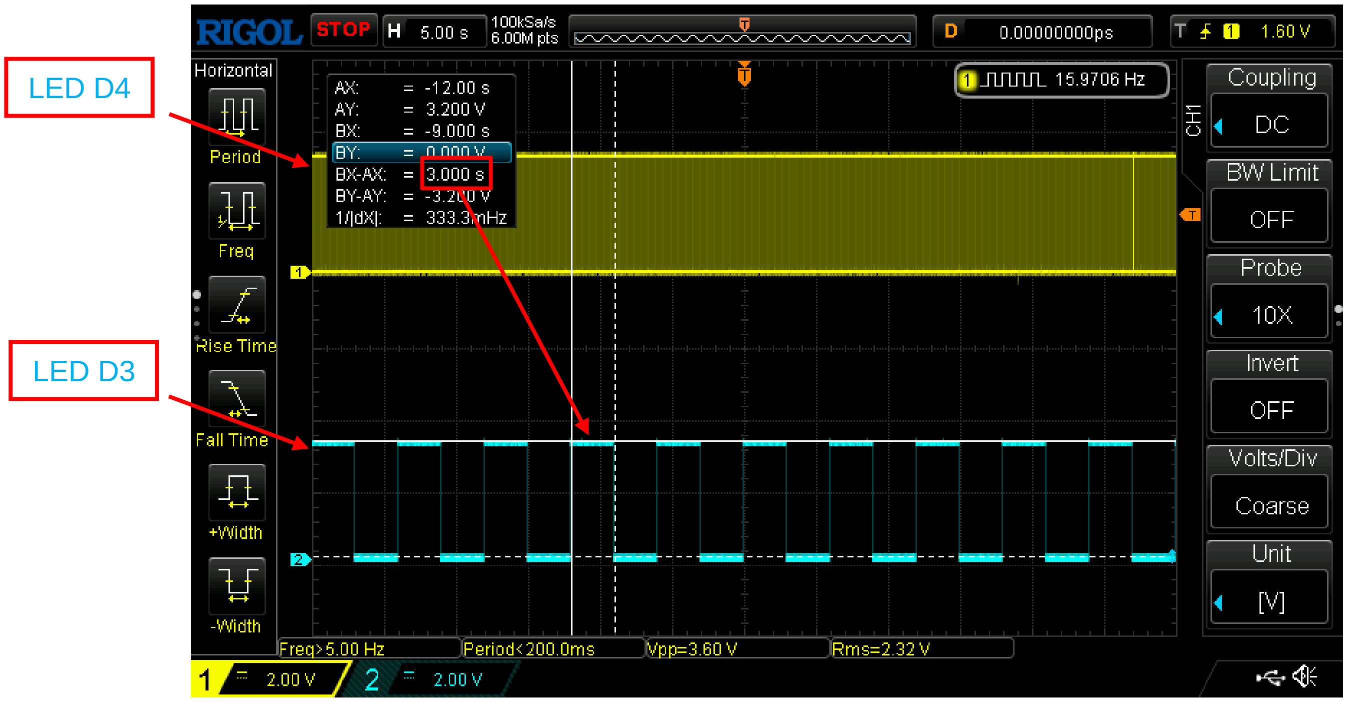
Requirements

You must populate the following two tasks; `vTask1` and `vTask2`.

Task `vTask1` must print the message “Task1” every 30mS using `vTaskDelay()`

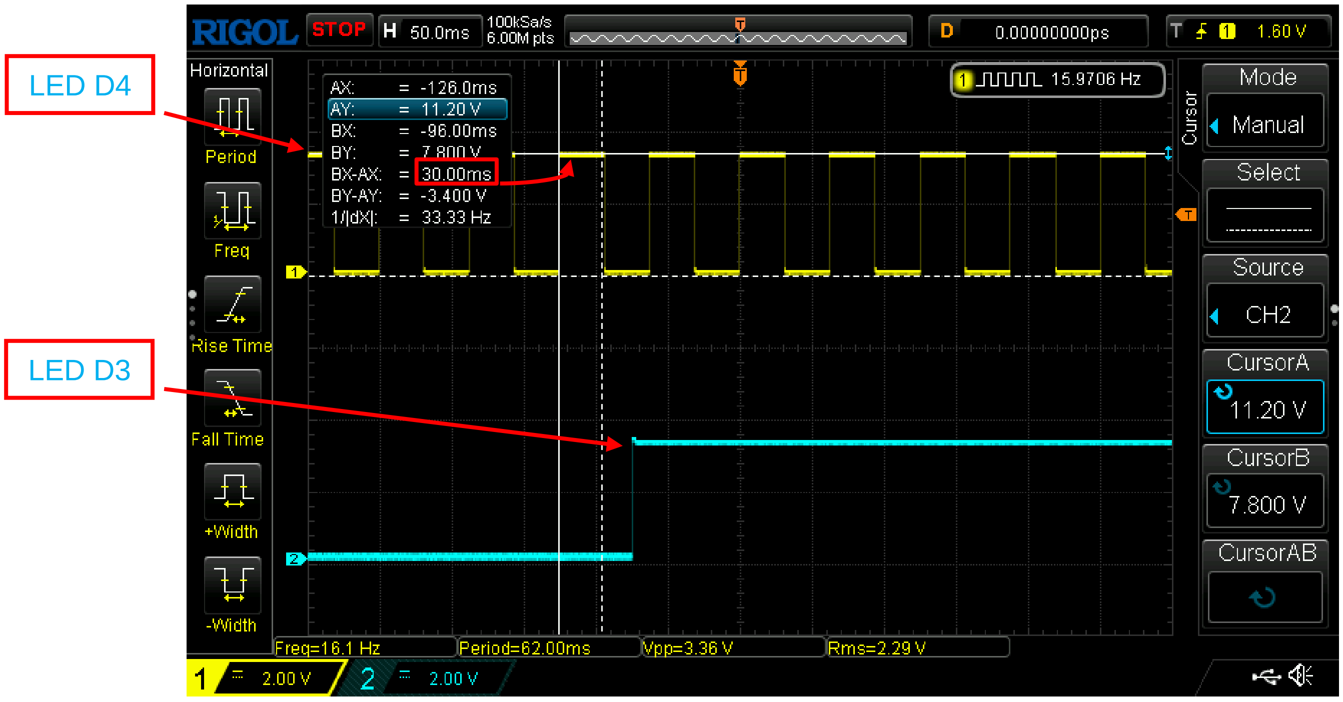
Task `vTask2` must print the message “Task2” using a crude for loop (about 150000 loops).

Refer to screenshot(s) fond on the next page.



```
H 5.00 s   100kSa/s 6.00M pts   D 0.00000000ps   T 1 1.60 V
1 15.9706 Hz
AX:     =  -12.00 s
AY:     =  3.200 V
BX:     =  -9.000 s
BY:     =  0.000 V
BX-AX:  =  3.000 s
BY-AY:  =  -3.200 V
1/|dX|: =  333.3mHz
Freq>5.00 Hz   Period<200.0ms   Vpp=3.60 V   Rms=2.32 V
1 = 2.00 V   2 = 2.00 V
```

Figure 1. Screenshot from two toggling LEDs on the Explorer 16/32 board. Yellow signal: toggling LED D3 controlled in vTask2 and blue signal: toggling LED D4 controlled in vTask1. The LED controlled in vTask1 toggles around every 3 seconds and the LED controlled in vTask2 toggles around every 30mS.



```
H 50.0ms   100kSa/s 6.00M pts   D 0.00000000ps   T 1 1.60 V
1 15.9706 Hz
AX:     =  -126.0ms
AY:     =  11.20 V
BX:     =  -96.00ms
BY:     =  7.800 V
BX-AX:  =  30.00ms
BY-AY:  =  -3.400 V
1/|dX|: =  33.33 Hz
Freq=16.1 Hz   Period=62.00ms   Vpp=3.36 V   Rms=2.29 V
1 = 2.00 V   2 = 2.00 V
```

Figure 1. Screenshot from two toggling LEDs on the Explorer 16/32 board. Yellow signal: toggling LED D3 controlled in vTask2 and blue signal: toggling LED D4 controlled in vTask1. The LED controlled in vTask1 toggles around every 3 seconds and the LED controlled in vTask2 toggles around every 30mS.

Task priority

vTask1 priority = vTask2 priority:

- Set both tasks' priority to 1.
- Explain the result:

When both task priorities are set to 1 (equal), the result shows that they seem to be running in parallel. This is because the RTOS scheduler is time-slicing between the two tasks every 1mS.

LED D4, which is controlled in vTask1, toggles around every 3 seconds and LED D3, which is controlled in vTask2, toggles around every 30mS on the Explorer 16/32 board.

vTask1 stays in the blocked state until 3 seconds has elapsed, then it will switch to ready state and finally, to running state (LED D4 then toggles). On the other hand, vTask2 will never enter the blocked state. It is always in the ready state because of the nature of the crude NULL loop. Inside the crude NULL loop, it just keeps toggling the LED D3.

In theory, vTask2 would be the one always in the running state because of the nature of the crude NULL loop, and it will starve vTask1 of running time. But since the task priorities are equal, the RTOS scheduler has no choice but to split the running time equally between the two tasks (around 1mS for each task). We will see how the change of task priorities has an impact on the system behaviour below.

vTask1 priority < vTask2 priority:

- Set `vTask2` to a higher priority.
- Run the code explain the result:

When vTask2 has a higher priority (set to 2) and vTask1 has the same priority as before (set to 1), the result shows that the two tasks seem to be no longer running in parallel. This is partially because of how the RTOS scheduler is now handling the running time for the two tasks; it is no longer time-slicing between the two tasks every 1mS.

LED D4, which is controlled in vTask1, no longer toggles at all and LED D3, which is controlled in vTask2, still toggles around every 30mS on the Explorer 16/32 board as before.

The true problem lies in vTask2; it is preventing vTask1 from ever being able to enter the running state (vTask2 never enters the blocked state). vTask1 will be able to switch from the blocked state to the ready state, but since vTask2 is using a crude NULL loop and has a higher priority set than vTask1, the RTOS scheduler will never let vTask1 enter the running state. As predicted from before, the RTOS scheduler favors vTask2 for running time and starves running time for vTask1.

vTask1 priority > vTask2 priority:

- Set `vTask1` to a higher priority.
- Run the code and explain the result:

When vTask1 has a higher priority (set to 2) and vTask2 has a lower priority (set to 1), the outcome is the same as when both tasks were set to equal priorities (as shown previously). The two tasks now seem to be running in parallel once again.

LED D4, which is controlled in vTask1, toggles around every 3 seconds and LED D3, which is controlled in vTask2, toggles around every 30mS on the Explorer 16/32 board.

The reason is simple: although vTask2 has the crude NULL loop, the RTOS scheduler will now favor vTask1 for running time because it has a higher priority set than vTask2. vTask1 will keep going into the blocking state around every 3 seconds, then to the ready state, and finally to the running state (LED D4 then toggles). For vTask2, it still will never enter the blocked state because of the nature of the crude NULL loop. The difference now is since vTask1 has more importance (higher priority) than vTask2, so the RTOS scheduler will favor vTask1 when it is time for the scheduler to decide which task gets running time. This will only happen when vTask1 becomes unblocked (in the ready state).

Note: when vTask1 enters the blocked state, it does not starve running time for other tasks. Instead, the RTOS scheduler lets other tasks enter the running state until the higher priority tasks become unblocked (enter ready state). The RTOS scheduler will then prevent the lower priority tasks from running and will only let the higher priority tasks run.

Approval required

- Comment, commit and push the project to GitHub.

Questions:

Q1- How would you modify the code so that if vTask1 priority is lower than vTask2 priority **both port pins toggle**?

In order to maintain both port pins toggling, with vTask1 set to a lower priority than vTask2, vTask2 needs to be restructured in the way it normally behaves. vTask2 should not be using the crude NULL loop. Instead, it should be using the same blocking delay found in vTask1.

By structuring vTask2 this way, it will allow the system to execute other tasks that are in the ready state. vTask2 will now behave like vTask1: when vTask2 enters the blocked state, the RTOS scheduler will allow another task to enter the running state if it deems that it is appropriate to do so. Even with the different priorities, vTask2 will no longer starve any other tasks from running time.

Q2- When vTask1 priority is lower than vTask2 priority; what are the possible states for the tasks (B, RU, RY)?

To put it short: vTask1 will be stuck in the ready state forever and vTask2 will be stuck in the running state forever (vTask2 starves vTask1 of running time).

When vTask1 has a priority lower than vTask2, the system will behave in the following way:

For vTask1: the task initially enters the ready state but will never enter the running state because of how vTask2 operates.

For vTask2: the task initially enters the ready state, then the task will enter immediately into the running state (before vTask1 can ever do so) because vTask2 has a higher priority set. vTask2 will then never enter the blocked state because of the way it is structured.

Refer to the previous sections for an alternate explanation.

Q3- When vTask1 priority is greater than vTask2 priority; what are the possible states for the tasks (B, RU, RY)?

To put it short: vTask1 will no longer be stuck in the ready state forever and vTask2 will no longer be stuck in the running state forever. Although, if it were not for the RTOS scheduler, vTask2 would still starve vTask1 from running time.

When vTask1 has a higher priority than vTask2, the system will behave in the following way:

For vTask1: the task initially enters the ready state, then the task will enter the running state immediately (before vTask2 can do so) to toggle LED D4. It will then enter the blocking state, letting the RTOS scheduler give running time to other tasks.

For vTask2: the task initially enters the ready state, but it will not enter the running state until vTask1 becomes blocked (because of the priorities set). When vTask1 enters the blocked state, the RTOS scheduler will allow vTask2 to enter the running state. vTask2 will then execute for as long as it can before vTask1 unblocks (enters the ready state). vTask2 will then be forced to halt execution until vTask1 enters the blocked state once again.

Refer to the previous sections for an alternate explanation.

Q4- Does the processor get some slack time (spare processing capacity left)? Explain.

With how the two tasks behave different and affect how the system behaves, the processor doesn't really have any slack left. This is mainly because of the way vTask2 behaves.

Due to the nature of the crude NULL loop found in vTask2, it will really never let the processor get some slack time. The only task that gives some slack time for the processor is vTask1. This is because of the blocking delay found inside it.

The blocking delay inside in vTask1 allows the processor to have some slack when vTask1 doesn't need to run (it frees up running time and system resources when the task is idle).

Refer to the previous sections for an alternate explanation.

Appendix

Use the following function for simulation purpose, see `console32.c` file:

```
Function: int  fprintf2(int mode, char *buffer)

   Precondition:
      initUartx must be called prior to calling this routine.

   Overview:
      This function prints a string of characters to the selected
          console. Possible choices are _UART1, _UART2, _LCD

   Input:
        mode: select the console C_UART1, C_UART2 or C_LCD
        buffer: Pointer to a character string to be outputted

   Output: returns the number of characters transmitted to the console

Function: void initUartx( void)
    Overview: this function initializes the UARTx serial port
```

Example to print to LCD in target mode and to UART2 in simulation mode:

```
#define SIMULATION

#ifdef SIMULATION
    #define CONSOLE     C_UART2
#else
    #define CONSOLE     C_LCD

#endif
initUart1();      // initializes simulation UART1 display tab
//initLCD();            // initializes target LCD display

fprintf2(CONSOLE,"debugging message 1\n");
```