***Lab 2: blinky disco with FreeRTOS***

***What will learn in this lab***

* Writing FreeRTOS code to create multiple tasks at different priorities.
* Using a UART to communicate to a host via text (though not programming the UART CSRs).

***Overview***

Our first lab used a simple bare-metal system to make one or more LEDs blink. Bare-metal systems are simple, but limited; while blinking one LED was easy, blinking two LEDs at different frequencies wasn’t.

In this lab, we’ll use FreeRTOS to make the job easier. Each LED will have its own FreeRTOS task; since each task operates mostly independently from each other task, we will be able to write multiple tasks at different frequencies, allowing us to use as many LEDs as we like with few issues.

Blinking LEDs is one simple way to communicate with a user. A more powerful and sophisticated method, of course, is text. Since the Nucleo board doesn’t have a keyboard or monitor, we’ll send text via a UART and USB port to our host PC. But serial communication can involve waiting, which would disrupt our LEDs blinking. To avoid this, we put the UART code into a third task, running at a different priority. Welcome to real-time operating systems!

***Initial set up***

Our first lab only needed *lab1\_main.c* and a few library files. This lab will need numerous additional files: while FreeRTOS is one of the smallest RTOSs available, it still comprises a number of files. To make it easier for you to set this up, we will give you the project mostly built, and you need only add a few library files and flesh out *lab2\_main.c*.

You can grab the initial project set up from the course web page under “Common files for Labs;” it’s the clean project-setup environment for VSCode/FreeRTOS, and is called PIO\_FreeRTOS\_clean.zip. Download and unzip it, and you’ll have a directory tree. Next, rename and move the files so that you end up with a new project called *lab2\_FreeRTOS\_blinky*, as a sibling to last week’s project *lab1\_bare\_metal\_blinky* under the ProjectIO\Projects directory. Note that Windows will likely create nested top-level directories when it unzips, which you’ll have to manually move.

Next, add in *lab2\_main.c*, *lib\_clock.c*, *lib\_gpio.c* and *lib\_UART.c* into the *src* directory, and add in *lab\_ee152.h* into the *include* directory (all of these files are on the course webpage, just like last week). Your project is now set up, and ready for you to modify *lab2\_main.c*. Remember that for this lab, you now get to use the official version of *lib\_gpio.c* from the central library directory rather than the one you wrote for lab #1.

***Modifying main.c for the LEDs***

Look inside of *lab2\_main.c*. You’ll notice that while we’ve given you a framework for this file, it is not fully fleshed out. So, your first task: add enough code to get the two LEDs to blink. Use what you have learned in class last week and build on the same APIs as last week. But this time, you should not use *ms\_delay* (). Instead, use a delay function from FreeRTOS, as we discussed in class.

By making small alterations in your code, you should be able to build three scenarios:

* The red and green LEDs blink in unison twice per second.
* The red and green LEDs blink alternately twice per second.
* The red LED blinks twice per second and the green LED blinks three times per second.

In all three of these cases, each LED should have its own task. Save your code for each of the three scenarios to turn in.

***Adding in the UART: printing and reading***

One of the most powerful capabilities of an RTOS is the ability to have some tasks be at a higher priority than other tasks. With just two tasks, one for each LED, there is very little reason for the two tasks to have different priorities. With a UART, though, we must ask whether the UART task has a higher, lower, or equal priority than the LED tasks.

You can see a task creation of *task\_uart*(), originally commented out, that you should now uncomment to create the UART task. You must still understand its functionality, and then set its priority. To correctly assign the priority, you should first look at how the UART read and write functions are implemented internally. Are they done with spin-wait loops? Are they done with interrupts? You can only decide the task priority once you know what the task will be doing :-).

The UART task gives you the capability of selecting any combination of the two LEDs to blink or not blink by typing R, G, B (both) or N (neither). In order to use the UART, you will need a host terminal for it to talk to. I usually use MOBA, but feel free to use something else if you prefer.

Once you have everything working, please show it to us for your lab check-out.

As part of your check-out, please also explain why you chose the task priorities as you did. If you had chosen differently, what pitfalls might have occurred?

***What to turn in***

Turn in your final *lab2\_main.c* that shows the UART controlling whether the LEDs blink or do not blink. Also show (in your lab report) your code for the three blink scenarios that you implemented.

Finally answer the questions below:

1. We implemented UART with a spin-wait loop, which is rarely an efficient use of resources. In our case it was fine (since blinking an LED takes so little CPU power); but in general, we try to avoid spin-wait loops. Without making the UART interrupt-driven, can you think of any other ways to avoid spin-waits? You need not write the code, but just describe your plan.
2. Last week, you figured out why the function *ms\_delay*(), which works by executing NOP instructions, still works fine no matter which clock speed we set. Now let’s try to figure out how FreeRTOS manages to create an interrupt tick every 1ms no matter how we set the chip’s clock speed. A bit of background: there is a counter deep inside the microcontroller called the *SysTick* counter. It gets preset to a value (and FreeRTOS is what programs that preset value) and then counts down to zero, with one count every clock cycle. When it hits zero, it generates an interrupt and then reloads to the preset and starts counting again. So the trick is that FreeRTOS must know the clock frequency and then set the SysTick preset accordingly. Looking at the file *FreeRTOSConfig.h* (and with perhaps a bit of search-engine help) can you figure out how this trick works?