

# ECSE444 Microprocessors

## Winter 2021

### Lab 4: I<sup>2</sup>C Peripherals and OS

This exercise configures an embedded OS and uses sensors external to the processor on your B-L4S5I-IOT01A board.

#### Overview

In this lab, you will be introduced to embedded real-time operating systems (RTOS). The key advantage of an RTOS is the ability to control how often different parts of your program execute. By breaking `main()` into a number of tasks (i.e., threads), and using OS directives to put them to sleep, wake them up, coordinate between them, and set their relative importance, it is possible to ensure critical work is done in a timely fashion, preempting other, less critical work when necessary. While this is possible without an OS, an OS makes this significantly easier. This lab will introduce UART, and I<sup>2</sup>C peripherals; you will coordinate OS tasks that read I<sup>2</sup>C sensor values and print them to a terminal. Which sensor value is printed will be controlled by another task that manages the push-button. Use of sensors will be facilitated by a board support package (BSP).

#### Resources

[B-L4SI-IOT01A User Manual](#)

[HAL Driver User Manual](#)

[Board Support Package Files on MyCourses](#)

[HTS221 Datasheet](#): Temperature and humidity sensor

[LIS3MDL Datasheet](#): Magnetometer

[LPS22HB Datasheet](#): Pressure sensor

[LSM6DSL Datasheet](#): Accelerometer and gyroscope

Optional (earlier board with the same peripherals): [B-L475E-IOT01 BSP Driver Reference](#)

#### Part 1: UART and I<sup>2</sup>C Peripherals Configuration

##### *Initialization*

Start a new project in STM32CubeMX, and initialize it as in earlier labs. For this lab, we need:

- The blue button, and associated external interrupt; and,
- (Optional) LEDs to indicate errors or progress.

##### *I<sup>2</sup>C Sensors*

I<sup>2</sup>C is a common interface for peripherals that assigns each register on each peripheral to a different address; I<sup>2</sup>C devices therefore listen (to addresses) and respond (with data) upon request. Our board has a number of sensors connected by I<sup>2</sup>C: a humidity and temperature sensor (HTS221); a 3-axis magnetometer (LIS3MDL); a 3D accelerometer and gyroscope (LSM6DSL); a barometer (LPS22HB); and, time-of-flight and gesture detection sensor (VL53L0X).

The board support package (BSP) by STM simplifies working with these peripherals: functions for initializing and using them are already written. These functions also take care of scaling sensor outputs, a relief after Lab 3. All we need to do is configure the I<sup>2</sup>C pins and include the appropriate source and header files in our project. These files are part of STM32L4 package – for your convenience, we uploaded them on: [Board Support Package Files on MyCourses](#).

You can find the I<sup>2</sup>C interfaces under *Connectivity* in the Cube MX chip features list on the left-hand side. Refer to the [B-L4S5I-IOT01A User Manual](#) to determine which to enable. Check the schematic to ensure that MX enables the appropriate pins. No further configuration of the I<sup>2</sup>C interfaces is necessary.

### *UART*

UARTs are often used to provide a user interface for configuring a device and for computer-to-peripheral communication. Many development boards provide a UART-based virtual com port to facilitate debugging, and more.

You will also find a number of UARTs and USARTs under *Connectivity*. Refer to the class notes and manuals to determine which UART or USART serves this purpose on your board. (The presence of the S in USART indicates that the connection can be synchronous, too; its absence indicates that the interface is only asynchronous.) Enable the appropriate interface, and then *check the schematic* and adjust the pinout accordingly. No further configuration of the UART is necessary; however, you may need the parameters in *Parameter Settings* to configure your terminal in order to see the output from the UART.

Now generate code; we'll get everything working as usual before coming back to MX and configuring the operating system.

## **Implementation**

### *Reading I<sup>2</sup>C Sensors*

The first step is to copy the BSP files into your project. BSP functions for each sensor are defined in one or more header files, named like `stm32l***_iot01_hsensor.h`; these must be included in `main.c`.

Choose one thing to measure using each of the following, HTS221, LIS3MDL, LSM6DSL, and LPS22HB; e.g., the HTS221 can output either temperature or humidity. Pick one (e.g., humidity). Note that while MX will initialize the I2C interface, it does not initialize the peripherals. Do so by calling the appropriate initialization functions (e.g., `BSP_HSENSOR_Init()`) in `main.c`.

Next, write code to read each sensor value (four of them) at 10 Hz sampling rate.

There is the [B-L465E-IOT01 BSP Driver Reference](#) for previous board that might be still useful in identifying the appropriate functions to use in each case; otherwise, if you're more comfortable exploring source and header files, start with `stm32l***_iot01*.*`.

### *Displaying Sensor Values on UART*

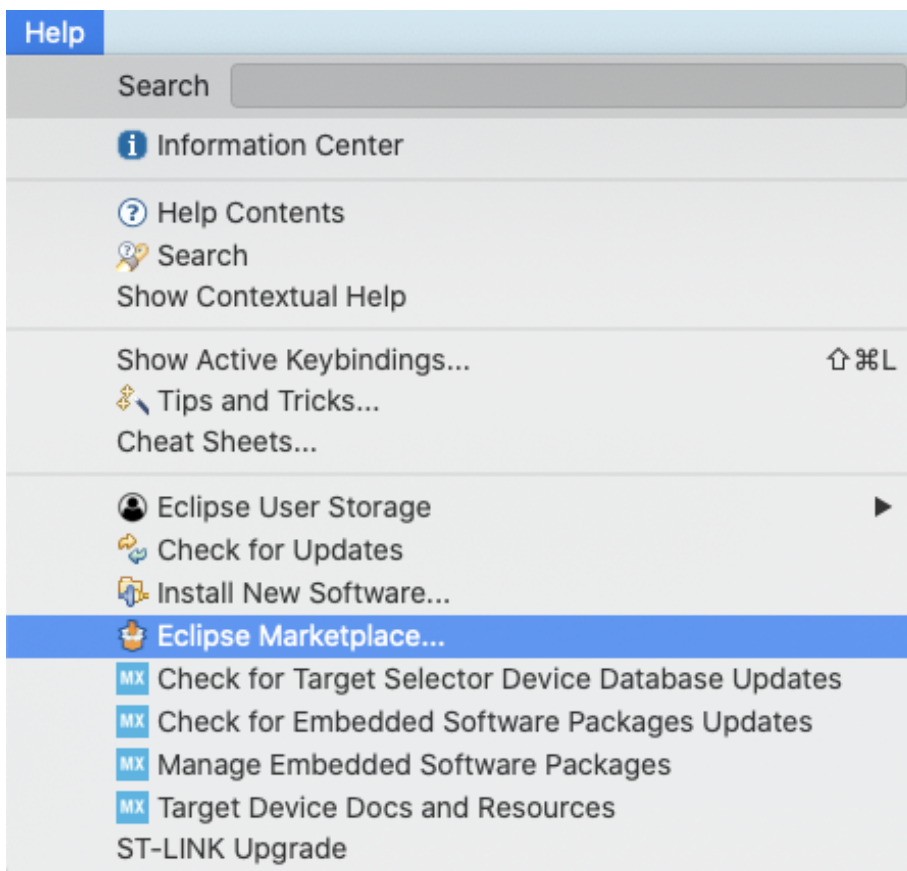
Now we want to print sensor values to a terminal. Refer to the [HAL Driver User Manual](#) for the functions required to work with the UART. Choose one sensor value to display and call the appropriate HAL function to transmit it over UART. Be sure to clearly indicate which sensor value

is being displayed.

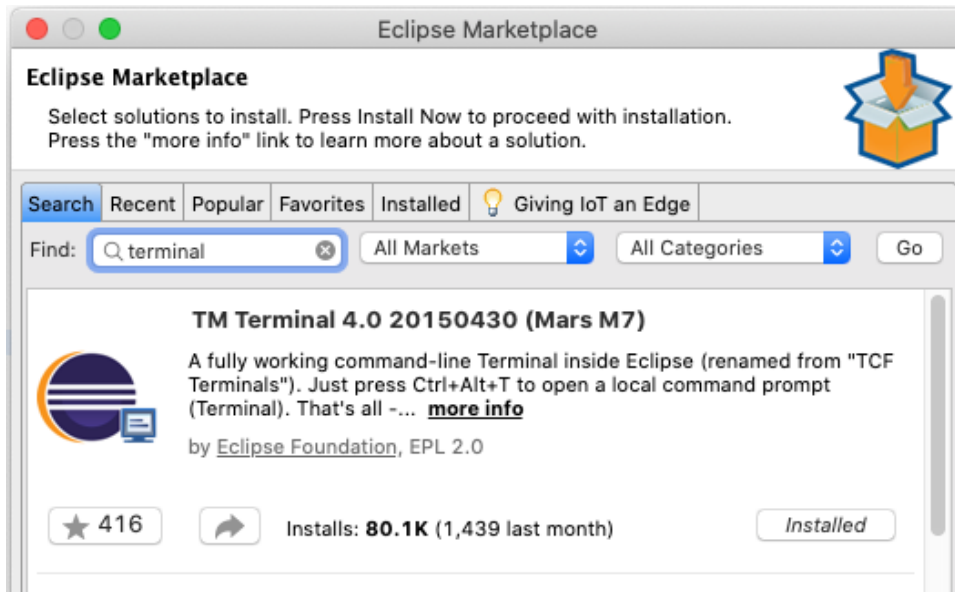
*Note:* you can use the `stdio` and `string` libraries to assist you here. However, additional configuration and care is required if you want to work with:

- `printf`; this requires that you overwrite `__weak` implementations of low-level IO functions to redirect output to UART. This is doable but is not covered here. `sprintf` is an alternative that almost works out of the box.
- `sprintf(buf, "Look, it's a floating point number: %.2f", temp);`  
Formatting floating point numbers requires that you change a compiler flag; STM32CubeIDE will direct you to the appropriate place, and this works fine *until we incorporate an operating system*. If not getting floating point numbers to print when using FreeRTOS (Part 2) feel free to cast all floats to integers before displaying.

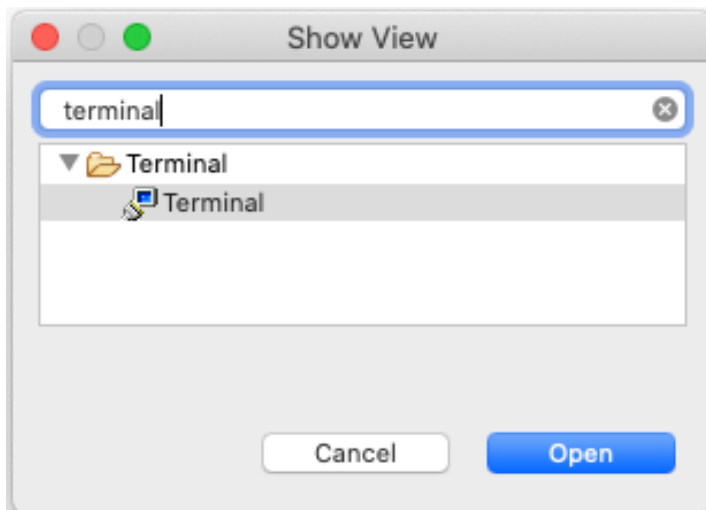
To send or receive data over UART, you will need to have an appropriate serial terminal program installed. There are many such programs, and they vary from platform to platform: Windows programmers most often rely on TeraTerm. However, it is also possible to install a terminal in Eclipse. Select the *Help* pulldown menu in STM32CubeIDE, and then *Eclipse Marketplace*.



Search for “terminal” and install *TM Terminal*.




Then, when you are in the *Debug* perspective, select the Window pulldown menu, and Show View > Other. Choose Terminal.



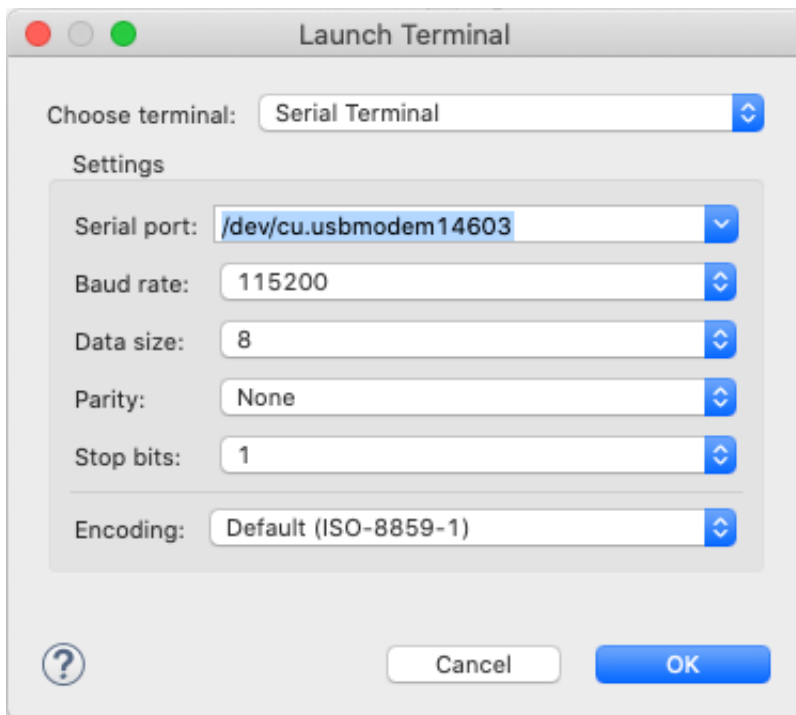
This will add a new tab.



Click  to connect. That will open a new window. Select *Serial Terminal*, and then the appropriate serial port:

- On OS X or Linux, it'll be something like `/dev/cu.usbmodemxxxxx`

The rest of the parameters should be set appropriately by default, but it is always a good idea to compare them with the configuration in MX.



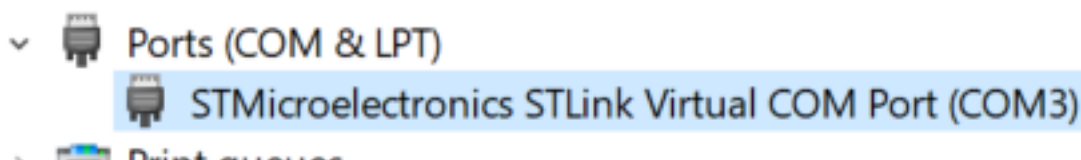
Click OK to connect, and the terminal will begin to display whatever your MCU is sending over UART.

If in OS X you cannot find a port as illustrated above, first confirm that such a port is active. Open the Terminal program, and `ls /dev/cu.*`. You should see something like:

```
macbook-pro-2016:dev $ ls cu.*
cu.Bluetooth-Incoming-Port cu.SOC
cu.MALS                    cu.usbmodem14603
```

If you do, then you may need to use an alternative serial terminal. SerialTools is available for free on the App Store, and works out of the box. Simply select the appropriate port and connect.

Windows 10 users may have some trouble using the integrated terminal, in particular, identifying the appropriate COM port. First, ensure that you have checked the schematic and have USART1 assigned to the appropriate pins in MX. If you find you still can't output to the terminal, go to the Windows 10 Device Manager. Under Ports (COM & LPT) you should see "STMicroelectronics STLink Virtual COM Port (COM##)" as pictured below (where ## is the number assigned by your PC).



If you observe this, and still cannot get the integrated terminal to work, please download and install a third-party COM terminal. There are many such programs available; besides Teraterm, there is

[Docklight](#). A trial version is freely accessible and should satisfy your needs for this lab.

Ubuntu/Linux users can identify the port of connection, using `$dmesg` command.

```
[15284.745041] usb 1-3: New USB device strings: Mfr=1, Product=2, SerialNumber=3
[15284.745044] usb 1-3: Product: STM32 STLink
[15284.745047] usb 1-3: Manufacturer: STMicroelectronics
[15284.745049] usb 1-3: SerialNumber: 066AFF323338424E43204523
[15284.993663] cdc_acm 1-3:1.2: ttyACM0: USB ACM device
[15284.994219] usbcore: registered new interface driver cdc_acm
[15284.994222] cdc_acm: USB Abstract Control Model driver for USB modems and ISDN adapters
[15285.013235] usb-storage 1-3:1.1: USB Mass Storage device detected
[15285.013720] scsi host2: usb-storage 1-3:1.1
[15285.014017] usbcore: registered new interface driver usb-storage
[15285.141213] usbcore: registered new interface driver uas
[15286.032343] scsi 2:0:0:0: Direct-Access    MBED    microcontroller    1.0    PQ: 0 ANSI: 2
```

For example, in the image above, `ttyACM0` is identified as the ST-LINK connection port.

Once you have identified the port, open the connection to UART using `minicom` using the following command,

```
$ sudo minicom -D /dev/ttyACM0
```

If the configuration is correctly done, you'll be able to see the UART logs on this minicom console.

### *Changing Sensors with the Push-button*

Now extend your implementation such that each time the button is pressed, data from a different sensor is displayed.

*Ensure that your program works before moving on, as debugging basic functionality is significantly more difficult once the OS is running, too.*

## **Part 2: CMSIS RTOS and FreeRTOS**

The key advantage of an embedded operating system is that it makes it easy to more carefully control when different parts of our program execute. For instance, perhaps we want to sample one sensor at 1 Hz, another at 10 Hz, and another at 100 Hz. Maybe we only want to check that a button has been pressed every 500 ms. And perhaps we want to log data (to display or otherwise take action) any time a new sample is taken. Implementing this with a single `main()`, even with timers and interrupts, may make it difficult to meet performance requirements.

### **Configuration and Implementation**

Back in MX, on the left-hand side there is a *Middleware* section, in which you will find *FreeRTOS*. Select it and choose *CMSIS\_V1* mode. There are many parameters available to configure FreeRTOS; we will leave everything set to default, with the exception of *Tasks and Queues*. *Mutexes*, and *Timers and Semaphores* may also be of interest, depending on how you wish to communicate between tasks and synchronize access to shared resources and data. Strictly speaking, however, they are not necessary for this assignment.

The first thing to do once *CMSIS\_V1* is enabled is change the timebase of the system. FreeRTOS uses the SysTick clock to determine when to perform context switches; this makes using `HAL_Delay` based on SysTick problematic: FreeRTOS wants a relatively low priority timer (because context switches should not interrupt interrupts), but HAL requires a relatively high priority timer (so timekeeping continues even during interrupts). Choose *SYS* from *System Core* on the left-hand side and change *Timebase Source* to another timer. Good choices are TIM6, TIM7,

TIM16, and TIM17. These timers have relative less functionality than the others.

Your objective now is to run your application in three tasks, rather than out of a single main() function:

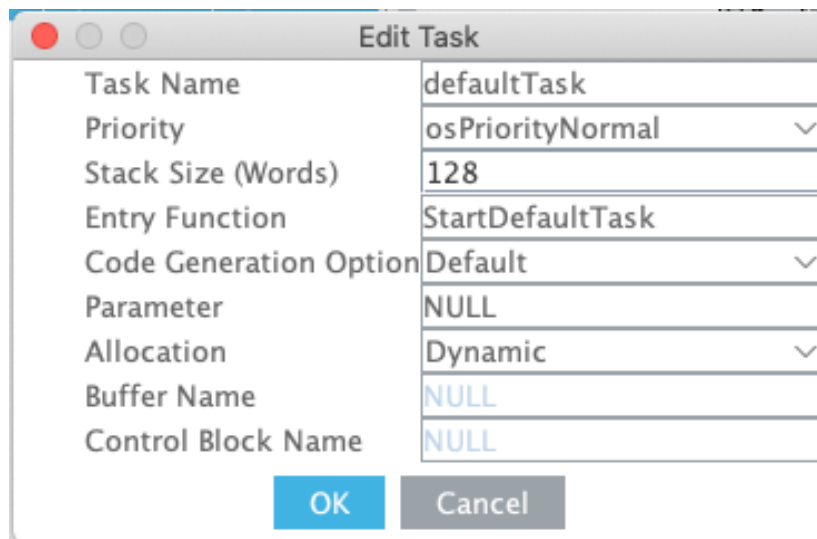
- One task that determines when the button has been pressed, and changes the mode of the application to output data from the next sensor in the sequence;
- One task that transmits this data to the terminal using the virtual com port UART; and,
- One task that reads sensor data.

Once you enable FreeRTOS, regenerating your code will create the first (default) task automatically. It is started by the OS *just before* main() enters the infinite while loop; at this point, the infinite while loop should be unreachable. *My recommendation is that you get everything working again in this single task* before you begin to create additional tasks and migrate functionality into them. The task is named defaultTask and is started when the OS calls a function in main.c, StartDefaultTask().

In your task, you'll see a new function, osDelay(...). This functions much like HAL\_Delay(...), with a key difference: osDelay(...) puts a thread to sleep, handing control back to the OS; HAL\_Delay(...) is blocking, pausing all user code execution. You need osDelay(...) to allow other threads of the same priority to execute; the delay you put in determines how long before the thread wakes and can execute again.

*Note:* for some reason, osDelay(...) calls should appear at the beginning of the for(;;) loop, not the end.

When you are ready, return to MX, and the FreeRTOS configuration, to add more tasks, under *Tasks and Queues*. Double click anywhere on the default task to pull up its configuration.



Rename the task to something more descriptive and update its entry function accordingly. The rest of the parameters can be left as is. As always, these changes will automatically update your code when you generate it.

Now add another task; pick suitable task and entry function names. The rest of the parameters can be left as is. Again, my recommendation is that you move functionality into this task, and get

everything working again, before repeating this process to add a third task.

### *Notes*

- Debugging with an OS is painful. Set breakpoints at the beginning of your tasks; chances are that problems originate there, and not in the OS itself, even if the call stack appears to suggest otherwise.
- Debugging systems with persistent RAM can be painful, too. Remember: if you don't power the board off, data from earlier runs may be resident in memory, and accessed (because C lets you touch anything not explicitly protected). This is especially true of dynamically allocated memory on the heap (the default for tasks), since the heap is not initialized (unlike statically allocated variables).
- Hard faults are the segmentation faults of embedded systems. If your code accesses memory that it shouldn't, encounters a stack overflow, or some other problem (including trying to format floating point numbers, or inconsistent configuration of peripherals, for instance), a hard fault interrupt will be triggered. It is difficult to work out what code caused the interrupt; single-stepping can be quite useful.
- If you are using `sprintf` or similar functions to format floating point numbers and don't get this to work with FreeRTOS, cast to `int`.
- Don't forget that debugging changes the relative timing of events; something may work with breakpoints and break without them (i.e., Heisenbugs); tracing with ITM is useful in these cases, as this has fewer side effects.
- Complex functions from standard libraries may require a larger stack (because of nested function calls) than provided by default; you can change the stack size for each task, or the minimum for all tasks, in MX. If you set the minimum too high, however, tasks may silently fail to start; an X instead of a ✓ in front of *FreeRTOS Heap Usage* indicates you're allocating too much memory (though MX will not prevent you from generating code like this).
- The location of `osDelay(...)` appears to matter. I'm not sure why! Make them the first thing that happens inside each task's `for(;;)` loop.
- If all else fails, start over, with your working code from before enabling FreeRTOS. *That's what happens with complex tools and flows, both open (GNU, Eclipse) and closed (e.g., Xcode).*



## Experimental Results to Demo

You are asked to reach the following milestones:

- C implementation of initializing, and reading four (4) different I<sup>2</sup>C sensors
- C implementation of transmitting I<sup>2</sup>C sensor data over UART to a terminal
- C implementation of push-button changing what sensor data is transmitted
- C implementation of the above using three different tasks in FreeRTOS

## Grading

- C implementation of data acquisition from I2C sensors
  - 30%
- C implementation of data transmission over UART
  - 30%
- Mode change using push-button
  - 10%
- Implementation of above using FreeRTOS or CMSIS-RTOS
  - 30%

## Final Report

Once you have all the parts working, include all the relevant data to your report. The report should concisely explain your solution to the problem given, including the final code. You should use the established 2-column IEEE format. Please capture the screen shots and relevant code snippets and include them in the Appendix. All code should be well documented. Any performance evaluation and correctness validation should be apparent from your written report.

## Due Dates

The lab demonstration in which all the parts (interrupts, timer, DMA) are put together will be on

**Mar. 23<sup>rd</sup> and 25<sup>th</sup>**

and will include showing your source code and demonstrating a working program for all test cases.

The final report will be due on

**Friday, Mar. 26<sup>th</sup>**