ECSE 444: Microprocessors
Lab 7: I^{2}C Peripherals and OS

Abstract

In this lab you will learn how to configure and use an OS on our B-L475E-IOT01A development board. OS tasks will be used to interact with the user (with a push-button and UART), as well as acquire data from a variety of sensors.

Deliverables for demonstration

- C implementation of initializing, and reading four (4) different I^{2}C sensors
- C implementation of transmitting I^{2}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

- 30% C implementation of data acquisition from I^{2}C sensors
- 30% C implementation of transmitting over UART
- 10% C implementation of mode change using push-button
- 30% C implementation of the above working in FreeRTOS

Changelog

- 28-Aug-2020 Initial revision.
- 23-Oct-2020 Added an alternative serial terminal for Windows 10 users that need it.
- 26-Oct-2020 Added an alternative serial terminal for OS X users that need it.

Overview

In this lab, we'll 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^{2}C peripherals; you'll coordinate OS tasks that read I^{2}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. Once again, much of this will be facilitated by a board support package (BSP).

Resources

[B-L475E-IOT01 BSP Driver Reference]

B-L475E-IOT01A User Manual
HAL Driver User Manual
HTS221 Datasheet: Temperature and humidity sensor
LIS3MDL Datasheet: Magnetometer
LPS22HB Datasheet: Pressure sensor
LSM6DSL Datasheet: Accelerometer and gyroscope
STM32L475VG Datasheet
STM32L47xxx Reference Manual

Part 1: UART and I^{2}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^{2}C Sensors

I^{2}C is another common interface for peripherals, like SPI (Lab 6). Whereas SPI uses chip select signals to identify the peripheral targeted with a command, I^{2}C assigns each register on each peripheral a different address; I^{2}C devices therefore listen (to addresses) and respond (with data) upon request. Our board has a number of sensors connected by I^{2}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).

Like in Lab 6, the board support package provided by STM dramatically simplifies working with these peripherals: functions for initializing and reading them have already been written. These functions also take care of scaling sensor outputs, a relief after Lab 3. All we need to do is configure the I^{2}C pins, and include the appropriate source and header files in our project.

You'll find the I^{2}C interfaces under *Connectivity* in the chip features list on the left hand side. There are three; refer to the B-L475E-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^{2}C interfaces is necessary.

UART

Fun fact: in previous semesters, the UART appeared much earlier, in Lab 3. I, however, couldn't get it working correctly, *because I didn't check the schematic*, until much later.

UARTs are used to exchange data between computers using a serial link. They are often used to provide a user interface for configuring a device, but can also be used for computer-to-computer 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 B-L475E-IOT01A User Manual to determine which UART or USART serves this purpose on the B-L475E-IOT01A board. (The presence of the S in USART indicates that the connection can be synchronous, too; it's 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^2C 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 `stm32l475e_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`.

Now write code to read each sensor value (four of them) at 10 Hz.

You may find B-L475E-IOT01 BSP Driver Reference useful in identifying the appropriate functions to use in each case; otherwise, if you're more comfortable crawling source and header files, start with `stm32l475e_iot01*.*`.

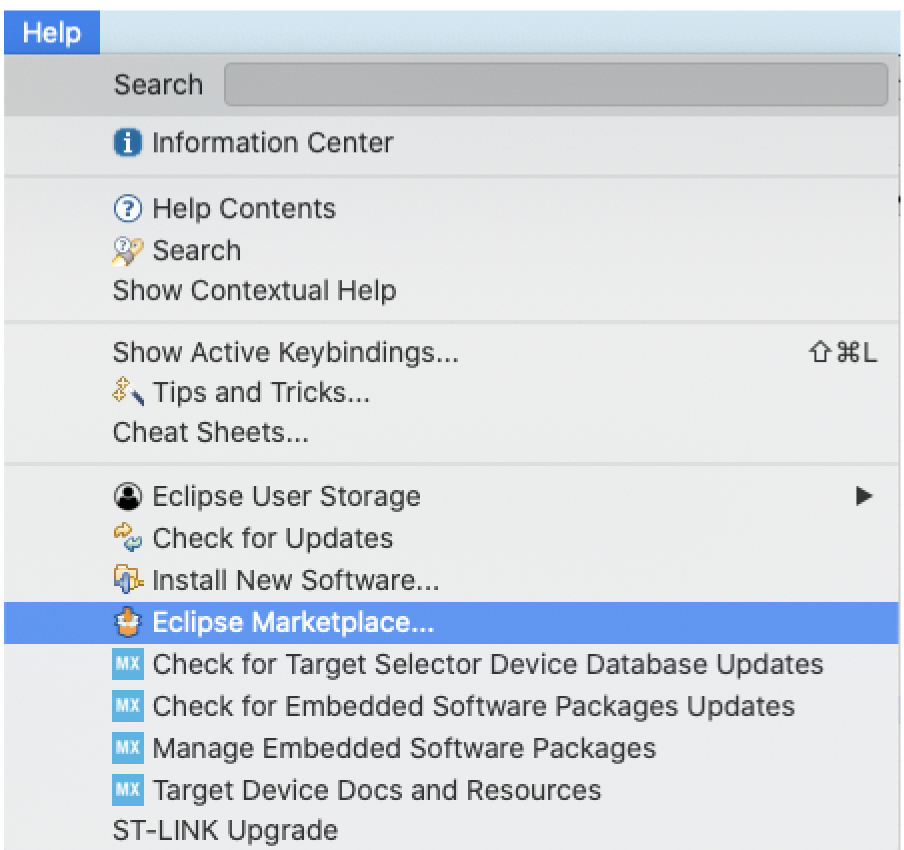
Displaying Sensor Values on UART

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.

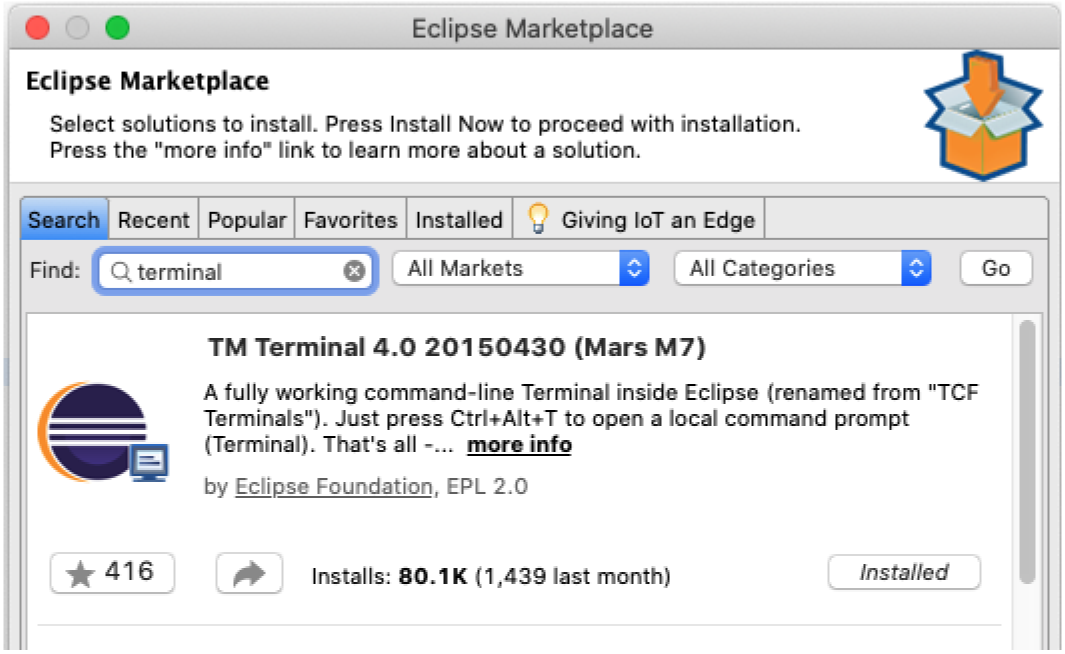
Note: you can use the `stdio` and `string` libraries to assist you here. However, additional configuration and care is required if you want to work with:

- `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.* I still haven’t figured out how to get floating point numbers to print when using FreeRTOS (Part 2); my solution casts all floats to integers before displaying.

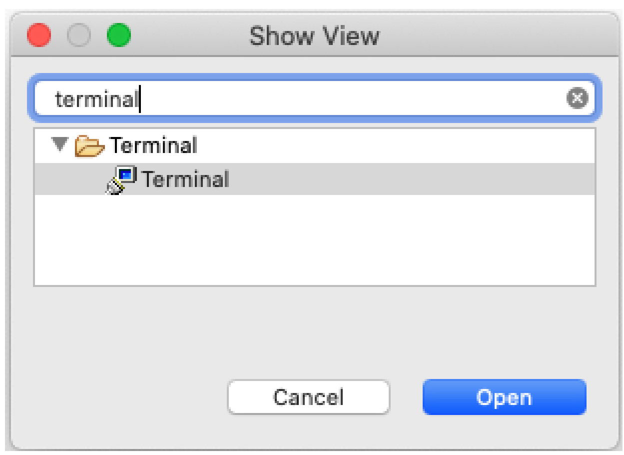
In order to send or receive data over UART, you will need to have an appropriate serial terminal program installed. There are many such programs, and they vary from platform to platform. However, it is also possible to install a terminal in Eclipse. Select the *Help* pulldown menu in STM32CubeIDE, and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



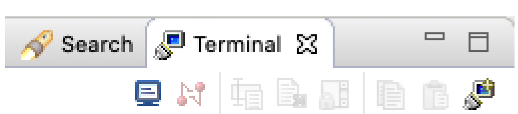
Search for “terminal” and install *TM Terminal.*



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.

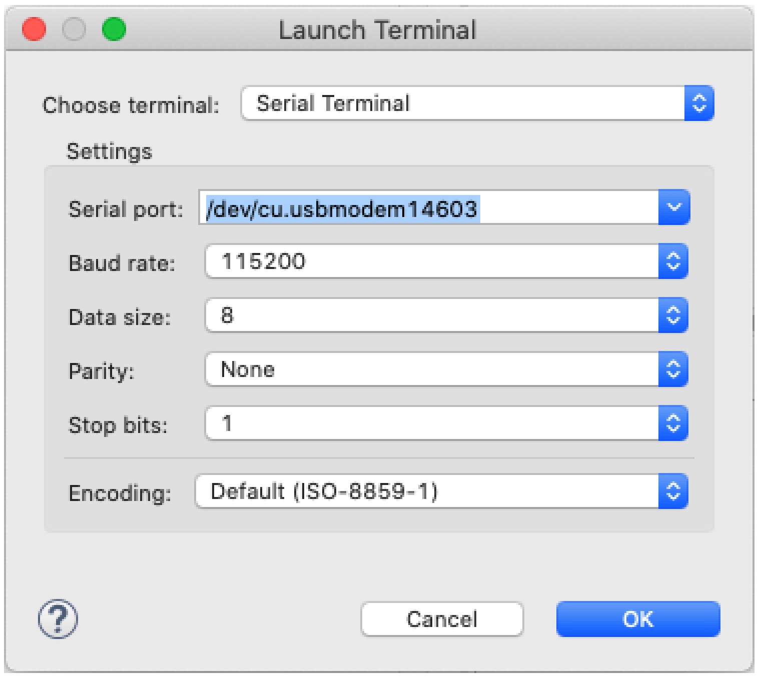


Click  to connect. That will open a new window. Select *Serial Terminal*, and then the appropriate serial port:

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

- On Windows, …???

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.



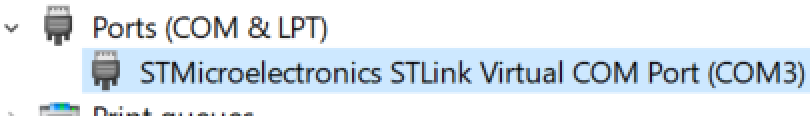
Click OK to connect, and the terminal will begin to display whatever your MCU is sending over UART.

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:

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

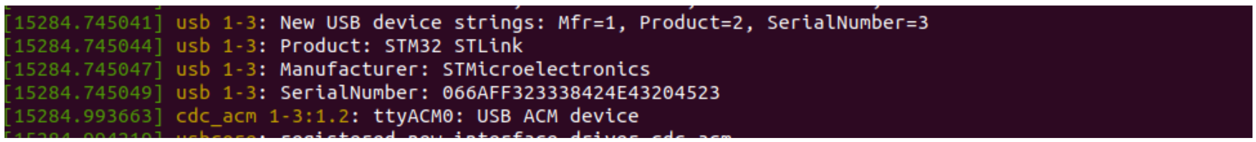
If you do, then you may need to use an alternative serial terminal. SerialTools is available for free on the App Store, and works out of the box. Simply select the appropriate port and connect.

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; we recommend [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.



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 mincom 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 OS 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:

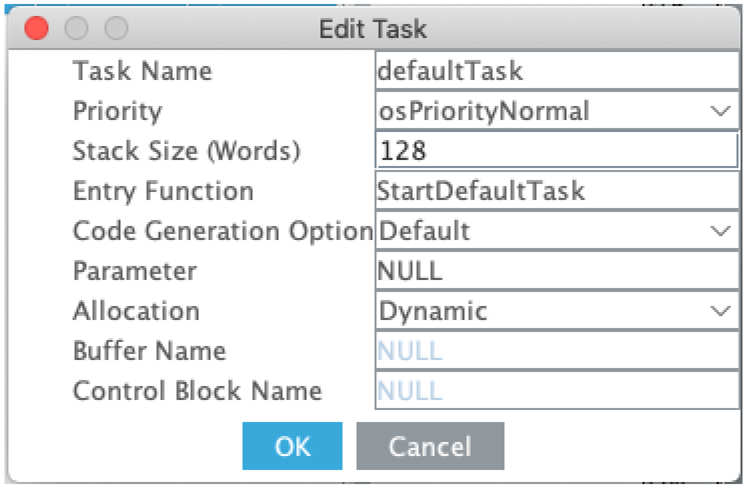
- One task that determines when the button has been pressed, and changes the mode of the application to output data from the next sensor in the sequence;
- One task that transmits this data to the terminal using the virtual com port UART; and,
- One task that reads sensor data.

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 that I cannot determine, `osDelay(...)` calls should appear at the beginning of the `for(;;)` loop, not the end.

When you're 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, I have not yet figured out how to get this to work with FreeRTOS. If you do, let me know; otherwise, cast to `int`.
- Don't forget that debugging changes the relative timing of events; something may work with breakpoints and break without them; 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 I had to do, and I'm still not sure why things didn't work the first time, or why they are working now.*