Lab 0x00 - Interrupt Callbacks and ADC Reading

New Attempt

- Due Oct 2 by 12:10pm
- Points 20
- Submitting a file upload
- File Types pdf

Overview

This assignment is to be completed in your lab groups in one week. Your team will use an analog to digital converter, commonly referred to as an ADC, to measure the capacitor voltage in an RC circuit. ADC data will be collected at a frequency of 500-1000 Hz while a step change is applied to the input of the RC circuit. You will then generate a step-response plot showing the ADC voltage plotted against time. Finally you will determine a time constant based on the step response data and then compare this value to one calculated from the component values in your RC circuit.

For this lab you will not need to use the Romi hardware with the modified Shoe of Brian. Instead, you will be working with the unmodified Shoe of Brian and the spare Nucleo provided to each team.

Familiarization

Before you begin the assignment, familiarize yourself with the hardware by connecting your Nucleo board and running some commands from the REPL; the REPL, or read-evaluate-print-loop, is what we use as a command line to interact with Python or MicroPython.

The following instructions are written assuming you are using a PC running Microsoft Windows to communicate with the Nucleo. If you are using a different platform it will be your responsibility to adapt the instructions to your own platform.

1. Connect your Nucleo board to a PC using a USB Mini-B cable; be sure to plug into the USB port on the "Shoe of Brian" (the bottom PCB), not the one on the Nucleo itself. You can interact with the Nucleo board using a serial monitor like PuTTY; the following steps will assume you are using PuTTY specifically on a Windows PC.

    A. With the Shoe of Brian connected to a PC through USB, navigate to "Device Manager" and browse to "Ports (COM & LPT)". In the drop-down, look for a device listed as "USB Serial Device (COMxx)" and make note of the port number, probably COM3 or COM4.

B. Open PuTTY and navigate to the serial connection settings. Configure the serial monitor with the following settings:

Serial Monitor Settings

Serial line to connect to	COMxx
Baudrate	115200
Data Bits	8
Stop Bits	1
Parity	None
Flow control	None

C. Navigate back to the "Session" category in PuTTY and change the "Connection type" to "Serial". To avoid having to reconfigure these settings in the future, highlight the saved session called "Default Settings" and click "Save" to store the COM port settings.

D. Click on the "Open" button to connect to the hardware and open the REPL. Confirm your connection is working properly by pressing Ctrl-C and then Ctrl-D to first cancel any running code and then soft-reset the device. You should be presented with the MicroPython REPL, indicated by three right-facing carets, `>>>`. You can also learn a little about MicroPython and the REPL by running the `help()` function.

2. Start by interacting with some of the basic features built into the Nucleo board, like the User LED. Try creating a pin object associated with pin **PA5**.

```
from pyb import Pin
PA5 = Pin(Pin.cpu.A5, mode=Pin.OUT_PP)
```

You can then turn the LED on and off using the state of **PA5**.

```
PA5.high()
PA5.low()
```

3. The pin, **PA5**, driving the LED is one of the pins associated with the timer peripherals on the STM32 microcontroller; specifically **PA5** is connected to **Timer 2 Channel 1**. Information about which peripherals connect to which pins can be found in the datasheet for the microcontroller.

Search online for "STM32L476 Datasheet" and look for a PDF document provided by ST Microelectronics. Be careful that you find the datasheet for that precise part number or the information will likely not match the lab hardware. You should then skim through the datasheet or

read the table of contents until you find something called the "Alternate Function" table. This is a matrix showing what features each pin is capable of on the STM32L476. This particular document (and table) will be one of the most common references for you throughout the term, so it may be smart to bookmark the document.

Create a timer object, and from that a channel object; then you can try adjusting the brightness of the LED by manipulating the PWM duty cycle for the timer channel. The code below shows how to set up a timer channel to toggle the LED pin at 1Hz; is that fast enough for human eyes to perceive a reduction in brightness? For the human eye to "average out" the blinks of the LED into a uniform brightness, the LED must blink fast enough that persistence of vision occurs. This is the same reason why modern video plays at a high framerate to look smooth and jitter free. Experiment with the timer frequency until you determine a minimum frequency that looks to the eye as a dimmed LED instead of one that blinks noticeably. Once you figure out the right frequency you can adjust the pulse-width-percent, also referred to as the duty cycle, of the timer channel to adjust the apparent brightness of the LED.

```
from pyb import Timer, Pin
tim2 = Timer(2, freq=1)
t2ch1 = tim2.channel(1, pin=Pin.cpu.A5, mode=Timer.PWM, pulse_width_percent=50)
```

The various features of the many timers in the STM32 allow very precise timing. The timers run in hardware, so it is only necessary to configure that hardware using code; once configured the timers automatically operate in the background without requiring active code to run in the foreground.

You can adjust the timer channel duty cycle without redefining or reconfiguring the timer or timer channel using a different function. The example below sets the duty cycle to 75%. The internal registers in the MCU buffer values like the duty cycle so that changes in value take affect on the next period of the PWM waveform. For more details read about the CCR (compare capture register) that is part of the timer module.

```
t2ch1.pulse_width_percent(75)
```

4. Now try interfacing with the User button on the Nucleo through pin **PC13**. First, reset the Nucleo by using "Ctrl-D" at the REPL. Resetting the microcontroller will turn off features like PWM output. Then you can set up a simple callback function as follows:

```
from pyb import ExtInt, Pin
PA5 = Pin(Pin.cpu.A5, mode=Pin.OUT_PP)
button_int = ExtInt(Pin.cpu.C13, ExtInt.IRQ_FALLING,
                    Pin.PULL_NONE, lambda p: PA5.value(0 if PA5.value() else 1))
```

You should now be able to toggle the LED state by pressing the blue user button on your Nucleo

board. When the user button is pressed, the ensuing falling edge of **PC13** will trigger an interrupt request, or IRQ. The MicroPython interpreter handles interrupt requests automatically by "calling back" a function that you've written in Python. In effect, your function will run once, automatically, each time the user button is pressed.

The preceding code uses what is called a "Lambda Function", also called an anonymous function in other programming languages. This Lambda function is just a very concise way of writing a short function.

```
lambda p: PA5.value(0 if PA5.value() else 1)
```

The snippet defines a Python object representing a function that takes in a single parameter, `p`, and then, depending on the state of the LED pin, it will set the LED pin to the opposite of its present value. The function must be declared with the input parameter `p` because the function must be ready to accept a variable passed in when the callback runs. The variable passed into the callback allows the code within the callback function to determine which pin triggered the callback. That way you can trigger the same callback function from many different pins while still being aware of which pin triggered individual callbacks.

It may be more clear to read through the standard function definition below which accomplishes the same thing as the more concise lambda function.

```
def button_LED_toggle(the_pin):
    if PA5.value():
        PA5.value(0) # or PA5.low()
    else:
        PA5.value(1) # or PA5.high()
```

With the function written as above, instead of as a Lambda function, the interrupt must be configured in a slightly different manner. A complete code snippet is shown below.

```
from pyb import ExtInt, Pin

PA5 = Pin(Pin.cpu.A5, mode=Pin.OUT_PP)

def button_LED_toggle(the_pin):
    if PA5.value():
        PA5.value(0) # or PA5.low()
    else:
        PA5.value(1) # or PA5.high()

button_int = ExtInt(Pin.cpu.C13, ExtInt.IRQ_FALLING,
                    Pin.PULL_NONE, button_LED_toggle)
```

Running Code From A File

The instructions in this handout are broken down into many discrete steps. It is possible to type in each line, one at a time, through the REPL, and produce the correct behavior; however, it is almost always more convenient to put code in a script (a file) so that you can simply run the script to produce results.

There are a few important things to remember when running code from a script:

- Code must be flashed onto the MCU before it can run. You cannot run a script stored on your computer on the MCU without first transferring, or "flashing" the code.

- Code cannot (or should not) be run while the MCU is being flashed with new code. The LED on the Nucleo labeled **LD2** will illuminate while the code is being flashed to the MCU. Do not reset the MCU while the LED is illuminated. If the MCU pin associated with **LD2** is configured for a different feature in actively running code the LED may not illuminate while code is being transferred. It is typically a good idea to select other pins to avoid conflict with the intended purpose of the user LED.

- Python is an interpreted language, not a compiled language. Therefore, some errors will only appear as the interpreter reaches that code during runtime.

- Code on the MCU is subject to corruption if a reset occurs at the wrong time or other unpredictable events occur. *Always store your code locally (on your own computer) and only flash code to the MCU when you want to run the code.* That is, do not try to work in-place by modifying code saved to the MCU.

- Avoid pressing the hard-reset button entirely, unless advised by your instructor. To reset the board use a soft-reset instead, by issuing a a "Ctrl-D" command from the REPL. Remember that you can cancel active code and return to the REPL by issuing a "Ctrl-C" command.

To flash code to the MCU and run that code, follow these steps:

1. Make sure that the USB cable is plugged into the Shoe of Brian, not the Nucleo, or you will not be able to flash code. When the USB cable is attached to your computer, you should see a flash drive enumerate, similar to standard USB thumb drives. It will most likely enumerate as the D: drive unless you have multiple hard drives or partitions in your computer. The drive should be called "PYBFLASH" in your file explorer.

2. Either drag-and-drop your desired Python file to the enumerated flash drive, or use save-as to move the code. As soon as you initiate the file transfer look at the indicator LED, marked **LD2**, which should illuminate while the data is transferring.

3. If there is a file named `main.py` present in the root directory on the flash drive, you can run the code automatically by soft-resetting the MCU. On each startup the code in this main script will run automatically.

   If you want to run code stored in a different Python file, use the `execfile()` function instead; however, it can be a pain to type this each time and using a main script is recommended.

It is probably wise to figure out the flashing procedure before you begin the assignment in full. Once you are comfortable with your familiarity with the hardware, basic usage of the REPL, and how to flash code, you can move on to the assignment.

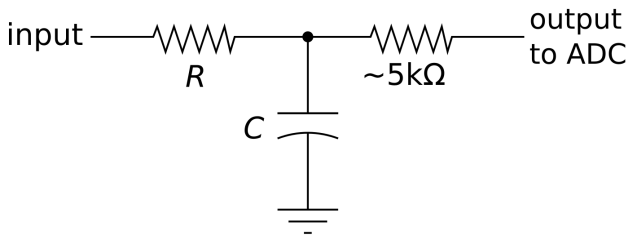
Assignment

Before you begin the assignment, read through all of the steps below. The steps outline the general process needed to complete the lab, but with many gaps you will have to fill in on your own.

It is *strongly* recommended that you build your program incrementally, slowly adding features and lines of code as you build and test the functionality. If you attempt to write the finished program line-by-line top-to-bottom you will have a much harder timer than building functionality in small pieces that can then be assembled into a working program.

1. Disconnect all power (the USB Cable) from your Nucleo board.

2. Build the following RC circuit using components found in your lab kit or provided by your lab instructor.

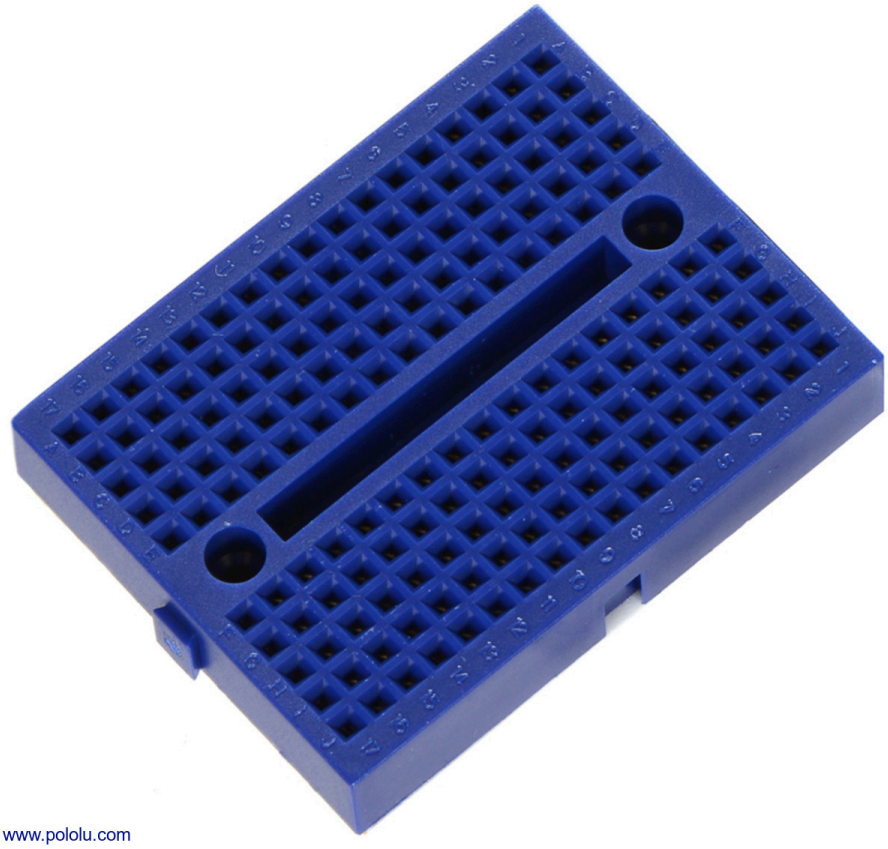


   Find a capacitor in the range of 2 to 10 µF and a resistor in the range of 20 to 100 kΩ; your instructor may supply specific components or you may need to search in the cabinets in the lab. These parts should be in good condition, but they may have already been handled by students in the past. Aim for an RC time constant between 0.1 and 1 seconds.

   The additional 5 kΩ resistor on the output side of the circuit is to protect the Nucleo in case your circuit is connected incorrectly or the MCU pin is configured incorrectly; in normal operation the

additional resistance should have no effect on the ADC readings.

If you haven't worked with a breadboard in a while, recall that each "half row" of five connection points is shorted together, but none of the rows are shorted with each other and the divider in the middle separates the two halves. The mini 170 point breadboards available for this lab do not have supply rails like most larger breadboards do; they should not be necessary to build the RC circuit.



3. Connect the input of the RC circuit to your Nucleo so that you can trigger a step response by changing the state of a digital pin on the Nucleo. Use pin **PC1** as your digital output, connected to the input of your RC circuit. You may find it helpful to examine a figure showing the pin names as available through the top connectors on the Nucleo. You can find some useful figures here: **https://os.mbed.com/platforms/ST-Nucleo-L476RG/ (https://os.mbed.com/platforms/ST-Nucleo-L476RG/)** .

   Always reference pins by their MCU pin name in the form **Pxy** where **x** is the port, and **y** is the pin; for example, **PC1** is pin 1 on port C.

4. Connect the output of the RC circuit, through the 5 kΩ resistor, to your Nucleo so that you can measure the capacitor voltage using the built in ADC. Use pin **PC0** for your ADC connection.

5. Reconnect your USB cable and use the Python REPL in PuTTY to read the ADC values manually and confirm that you are able to measure the voltage on the RC circuit correctly. You can make an

ADC object using the `pyb.ADC` class. Refer to the ADC documentation on the MicroPython website: **https://docs.micropython.org/en/latest/library/pyb.ADC.html (https://docs.micropython.org/en/latest/library/pyb.ADC.html)** .

It may also be wise to hook up an oscilloscope to your circuit so that you can confirm that the hardware is working properly.

6. Set up a timer to trigger callbacks at a rate of 500 Hz to 1 KHz using an object of the `pyb.Timer` class; either **Timer 6** or **Timer 7** would be a good choice for something like periodic callback generation because these two timers do not have any physical pins associated with them; that is, these timers are purpose built for use in software. Select a sampling frequency that will give you at least 200 datapoints but no more than about 1000 datapoints during a typical step response; this will give you sufficient resolution in your collected data while easily fitting within memory limitations of the MCU.

   Write your callback function so that it reads a single value from the ADC and stores it in an array. Since this function will be more complex than appropriate for a Lambda function, you will need to write a standard Python function. Start with the following skeleton below.

```
def tim_cb(tim):
    global data, idx
    # Code goes here
```

   This skeleton defines a function that takes in a single input argument representing a timer object. The function must be defined with this input parameter because when the timer callback is triggered, the timer object is automatically passed into the callback function. Unlike the **ExtInt** (external interrupt) callback that you set up in the familiarization section above, the **Timer** callback will run automatically at a specified interval rather than when a button is pressed. The callback can therefore sample from the ADC quickly and automatically, simply storing values on each iteration to be retrieved later by other code.

   The `global` keyword indicates that the same instances of variables `data` and `idx` can be modified both within and outside the function definition.

   To set up the callback to run at a regular interval, first create a timer object and then assign to it the callback function you want to run. This method uses an entire Timer for one purpose, triggering the callbacks. In many cases this can be a waste of resources, as it is possible to use a single channel of a Timer to achieve the same behavior while leaving other channels free for other purposes.

   For example, to run the preceding function at 100Hz with **Timer 7**, you could use the following code shown below.

```
tim7 = Timer(7, freq=100) # create the timer object
```

```
tim7.callback(tim_cb)     # assign the callback
```

You can also turn off a callback using a similar line of code as shown below.

```
tim7.callback(None)     # disable the callback
```

7. Write code that enables the callback, toggles the input to the RC circuit, and collects data over a long enough time window to fully capture your step response. This should be at least 4 or 5 times your estimated time constant.

   There are a variety of ways to store data in Python, but for this assignment it will be best to work with a numerical array of fixed size; read about them here: **https://docs.python.org/3/library/array.html (https://docs.python.org/3/library/array.html)**. The example snippet below shows how to create an array of zeros.

   ```
   from array import array
   data = array('H', 1000*[0])
   ```

   The array initializer requires two inputs, first, the type code, selected as `'H'` in this example, and an initial value for the array, selected as a Python list made up of 1000 zeros. Why is `'H'` the best choice of data type for this assignment?

   You can access or assign data to array elements using standard indexing with square brackets. For example, the following code snippet will assign the value 42 to the element of the array called `data` at position `idx` (counting from zero). Make sure that you aren't indexing out of bounds!

   ```
   data[idx] = 42
   ```

   If you want your program to simply wait for a period of time while the step response is running you can use one of the sleep commands. For example, the following code sleeps for 1500ms. For most assignments this quarter we will avoid using any sort of blocking code, like sleep or delay commands; however, for the scope of this initial assignment, you can write blocking code.

   ```
   from time import sleep_ms
   sleep_ms(1500)
   ```

8. Once the data collection has been completed, print the data to the Python REPL in a comma separated format with time in the left column and ADC readings in the right column. For the sake of this assignment you may assume that the timer runs at an exact interval matching the selected timer frequency. In future labs you will timestamp data as it is collected.

   To print data in a comma separated format you can simply loop through each index of the data and then print a formatted string. For example, the following code will print a single row of comma

separated data to PuTTY. Formatted strings, like the f-string used in the example, will be very useful throughout the quarter.

```
print(f"{idx}, {data[idx]}")
```

A more elegant, or "Pythonic", way to print the data would be to use a loop along with the `enumerate()` function to iterate through the collected data, rather than looping through indices and indexing the array within the loop. Practice with this if you feel interested.

When you print the data, make sure to convert the numerical values into appropriate units to use for your plot. That is, convert values to seconds and Volts before you send the data to your PuTTY window.

You can copy and paste the comma separated values into a text editor and then save the file with a name ending in ".CSV". The CSV file can then be used with programs like Excel, MATLAB, or with a Python script running on a PC to produce the deliverables described below. Note that for all future assignments plots will need to be produced using Python code.

Note: to copy in PuTTY all you need to do is highlight text, do not use Ctrl-C, as that keystroke is used to cancel code running on the Nucleo. Similarly, if you wish to paste into PuTTY you simply need to right click inside the terminal window and the text from your clipboard will be pasted into PuTTY.

Requirements and Deliverables

The following steps summarize the expected behavior of your program, in the same rough sequence as the code should be placed in your script:

1. Import any needed modules.

2. Create objects, such as pins, timers, and arrays. Configure or initialize the objects as needed.

3. Define the callback function that is to run each time an ADC sample is to be collected.

4. Make sure that the RC circuit is properly discharged before starting data collection.

5. Start data collection and then trigger the RC circuit to begin charging. Collect at least one data point before the circuit begins to charge.

6. Wait long enough for the entire collection of data to be sampled, or, alternatively, wait for an indication that the data collection is finished. How can you use the index counter or another variable

to know when the data is done collecting? In either case, make sure the callback is set up to avoid indexing out-of-bounds by collecting too much data; this will likely be handled best by disabling the callback. When all the data is collected, discharge the capacitor so that you can collect another sample.

7. Once all the data is collected, iterate through the data and print the data so that you may access the data on your computer through PuTTY.

8. For a fully featured program, steps 5 through 7 can be wrapped in an additional loop so that you can collect multiple sets of data without needing to restart the script. It may help to prompt the user to initiate data collection if you choose this option.

Once completed you should be able to trigger a step response automatically by pressing the blue user button on your Nucleo board or by interacting with your Nucleo through PuTTY. Once you've collected your data and imported the data into another tool on your PC, complete the following tasks listed below.

Create a plot showing the step response with voltage on the y-axis and time on the x-axis; include both the theoretical and experimental step responses; you may use the final value and time constant from your experimental results to create the theoretical step response data. Then, create a second plot with a linearized step response on the y-axis and time on the x-axis; also include a best-fit line based on the linearized experimental data. Your instructor may provide support on this in class.

Determine the slope of the line from your second figure and use that to calculate the time constant associated with the RC circuit. Annotate the plot with time constant.

Compare the value of your time constant, as calculated from your linearized plot, to your estimated time constant, as calculated from the resistor and capacitor values used in your circuit. Is the difference in these values commensurate with the tolerances of the resistor and capacitor used to build the circuit? Note that many standard components have nominal values with 5 to 10% tolerance.

You will submit a brief and concise memo with the required plots and analysis. Include your Python source code as attachments to the memo. You must use a standard memo format for your submission; if you are unfamiliar refresh yourself on proper memo formatting before submission. The memo will be submitted through Canvas as a single PDF document, once per group.

Assignment grading will be based on the following checklist. If your instructor is able confirm each item on the checklist your team will receive a "Complete" grade, otherwise your team will receive an "Incomplete" grade.

- Code runs and collects data properly when prompted through REPL or by pushing the blue button on the Nucleo.

- Upon completion of data collection, the sampled ADC readings are printed to the REPL in a CSV-style format.
- A pair of suitable plots have been created in Excel, MATLAB, or with Python code, showing experimental and theoretical values presented neatly with the measured time constant annotated clearly.
- A brief comparison has been made between the experimental and theoretical time constants.