18-100: Intro to Electrical and Computer Engineering

LAB08: I²C Lab

Writeup Due: Thursday, November 17th, 2022 at 10 PM

Name: Bonnie Ji

Andrew ID: bbji

How to submit labs:

Download from this file from *Canvas* and edit it with whatever PDF editor you're most comfortable with. Some recommendations from other students and courses that use Gradescope include:

DocHub	An online PDF annotator that works on desktop and mobile platforms.
pdfescape.com	A web-based PDF editor that works on most, if not all, devices.
iAnnotate	A cross-platform editor for mobile devices (iOS/Android).

If you have difficulties inserting your image into the PDF, simply append them as an extra page to the END of your lab packet and mark the given box. ***Do NOT insert between pages.***

If you'd prefer not to edit a PDF, you can print the document, write your answers in neatly and scan it as a PDF. *(Note: We do not recommend this as unreadable lab reports will not be graded!).* Once you've completed the lab, upload and submit it to *Gradescope*.

Note that while you may work with other students on completing the lab, this writeup is to be completed alone. Do not exchange or copy measurements, plots, code, calculations, or answer in the lab writeup.

Your lab grade will consist of two components:

1. Answers to all lab questions in your lab handout. The questions consist of measurements taken during the lab activities, calculations on those measurements and questions on the lab material.
2. A demonstration of your working lab circuits and conceptual understanding of the material. These demos are scheduled on an individual basis with your group TA.

Question:	1	2	3	Total
Points:	15	15	20	50
Score:				

Lab Outline

This lab aims to strengthen students' understanding of the I^2C communication protocol by demonstrating some practical applications of I^2C devices.

Sections

1. Introduction to I^2C
2. Temperature Sensor
3. I^2C-enabled Button
4. LCD Display and Software Libraries

Small Group Check-off Circuits

☐ I^2C circuit with temperature sensor, button, and display (pg. 16)

Required Materials

- 1x ADALM2000
- 1x Raspberry Pi Pico
- 1x I^2C TMP102 Temperature Sensor
- 1x I^2C Button w/ LED
- 1x I^2C Sparkfun Display
- 2x JST F-F Cables
- 1x JST Breakout M
- 1x JST Breakout F

Introduction to I²C

The Inter-Integrated Circuit (I²C, "I two C" or "I squared C") Protocol is a protocol intended to allow multiple "devices" to communicate with one or more "coorindators".

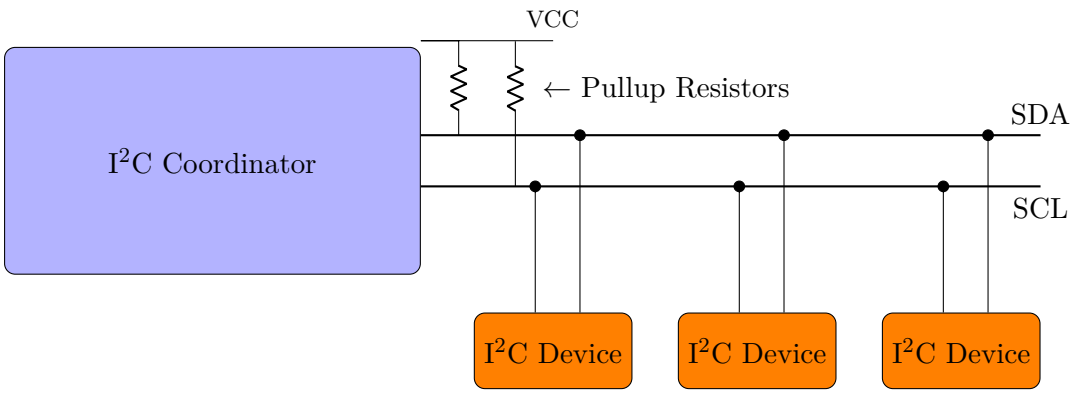


Figure 1: I²C Bus Diagram

I²C Hardware

The main benefit of I²C is how it only uses two wires but can have up to 128 different devices on the bus! The two lines used are called:

SDA Data line, where the bits of data are sent over the bus.

SCL Clock line, keeps everything in sync; tells the devices when to start/stop sending/receiving data.

Each device simply connects its SDA/SCL lines to the rest of the bus. By default both SDA and SCL are NOT pulled up. Once you properly connect the Raspberry Pi Pico I2C lines to the button or temperature sensor, SDA and SCL will get pulled up HIGH by pull-up resistors on those external devices. The clock signal is always generated by the coordinator; some devices may force the clock low at times to delay the device sending more data (or to require more time to prepare data before the coordinator attempts to clock it out, called "clock stretching").

I²C Protocol

Messages are broken up into two types of frame: an address frame, where the coordinator indicates the device to which the message is being sent, and one or more data frames, which are the 8-bit data messages that are shared. Data is placed on the SDA line after SCL goes low, and is sampled after the SCL line goes high.

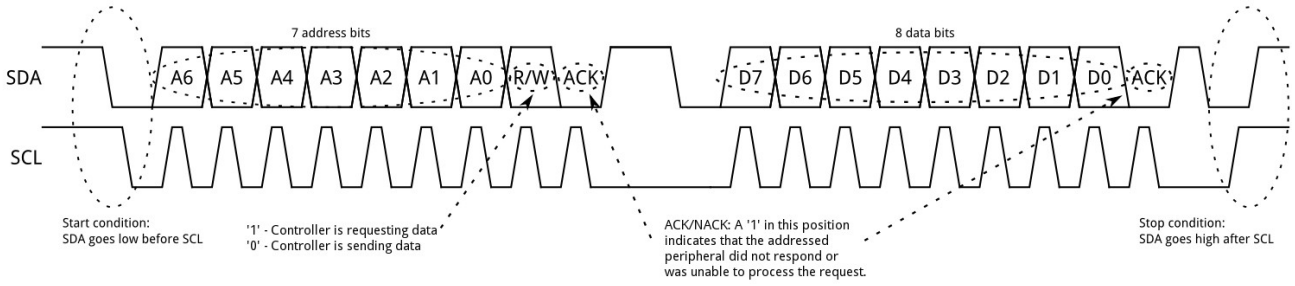


Figure 2: I²C Example Message

The different transfer states are explained on the next page.

Start Condition To initiate the address frame, the coordinator leaves SCL high and pulls SDA low. This puts all devices on notice that a transmission is about to start. If two devices wish to take ownership of the bus at one time, whichever device pulls SDA low first wins the race and gains control of the bus.

Address Frame The address frame is always first in any new communication sequence. For a 7-bit address, the address is clocked out most significant bit (MSB) first, followed by a R/W bit indicating whether this is a read (1) or write (0) operation. The 9th bit of the frame is the NACK/ACK bit. This is the case for all frames (data or address).

Data Frame After the address frame has been sent, data can begin being transmitted. The master will simply continue generating clock pulses at a regular interval, and the data will be placed on SDA by the device/coordinator, depending on whether the R/W bit indicated a read or write operation. Since, the number of data frames is arbitrary, the stop condition will indicate when all the data is sent.

Stop Condition Once all the data frames have been sent, the coordinator will generate a stop condition. Stop conditions are defined by a 0→1 (low to high) transition on SDA after a 0→1 transition on SCL, with SCL remaining high. During normal data writing operation, the value on SDA should not change when SCL is high, to avoid false stop conditions.

After each frame the device receiving the data is given control over SDA and will respond with either ACK (0) or NACK (1) (short for acknowledge/not acknowledge). If the receiving device does not pull the SDA line low , it can be inferred that the receiving device either did not receive the data or did not know how to parse the message. In that case, the exchange halts, and it's up to the coordinator of the system to decide how to proceed.

RPi Pico I^{2}C Support

The RPi Pico has dedicated pins for I^{2}C. **Those pins have built-in pull-up resistors that are required by the I^{2}C protocol. For the RPi Pico, the I^{2}C pins are GP4 (`SDA`) and GP5 (`SCL`).** Usually the device requires two additional pins: the power 3V3 (`VCC`) and the ground pin (`GND`).

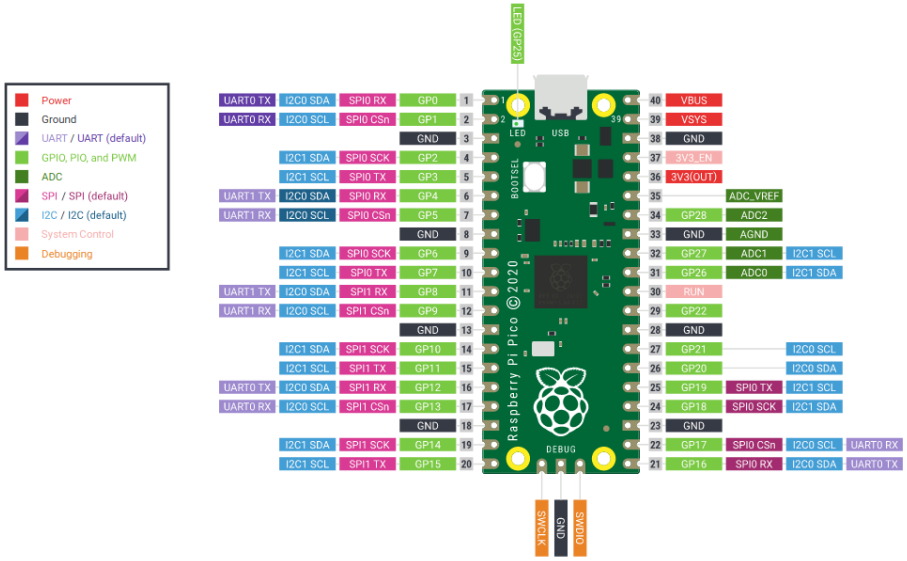
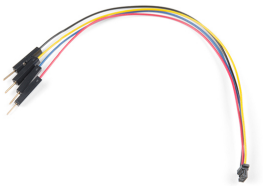


Figure 3: RPi Pico Pin out

The devices you'll use will connect via JST connectors for ease of assembly. On the JST the RED wire is VCC, the BLACK wire is GND, the blue wire should serve as SDA, and the Yellow wire SCL. You are given cables with JST on both ends so you can daisy chain several devices in a sequence (some devices have 2 JST ports so they can be connected in series with a previous and next device).

(a) Male JST Connector



(b) JST to Male Jumper Wires

(c) JST to Female Jumper Wires

Net	JST wire color	RPi Pico Net	RPi Pico Pin Number
SCL	Yellow	I2C0 SCL / GP5	7
SDA	Blue	I2C0 SDA / GP4	6
VCC	Red	3V3(OUT)	36
GND	Black	GND	3

Table 1: Summary of Connections

RPi Pico Busio Package

The following functions are useful in this lab:

`bytearray(n)` returns a bytearray of length n

`bytearray("Hello World!\n")` returns a bytearray containing the string `"Hello World!\n"`

`i2c.try_lock()` Attempts to acquire the I2C lock, returns a boolean of whether the acquisition was successful

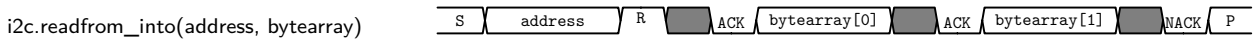
`i2c.unlock()` releases the I2C lock

`i2c.writeto(address, bytearray)` Writes the data in `bytearray` to the I2C Address `address`. After sending the 7-bit address for the device of interest, the coordinator will send a low 8th bit for a write command (`W`). Following the device's acknowledgement (`ACK`), this command will write the bytes from `bytearray` into the device, one by one.

If a device contains multiple registers, the first byte sent, `bytearray[0]`, will be the register of interest. For example, the I^{2}C-Enabled Button has 16 registers, each with their own 8-bit address. For example, the `LED_BRIGHTNESS` register has an address of `0x19`. Here is an illustration of the data frames on the bus when two bytes are sent:

`i2c.readfrom_into(address, bytearray)` Reads a number of bytes **from** the I2C Address `address` and saves the data **into** `bytearray`. Usually, this will be 1-3 bytes (the length of `bytearray` `len(bytearray)`).

This function results in the controller sending an address frame, then receiving number of data frames from the selected device. Here is an illustration of the data frames that are sent on the bus when `bytearray` holds two bytes:



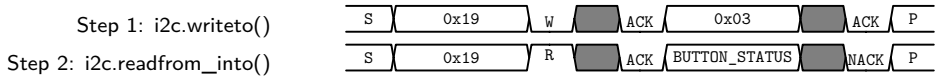
Notice the similarity between this function and the previous one, `i2c.writeto()`. The only difference is the status of the `R/W` bit sent with the address.

`i2c.writeto_then_readfrom(address, output_bytearray, input_bytearray)` Writes the data in `output_bytearray` to the I2C Address `address` then requests `len(bytearray)` bytes from the I^{2}C Address `address` and saves the data in `input_bytearray`.

If a device has data in multiple registers, it will return data from the last register that was written to. Therefore, most data acquisition requires first a call to `i2c.writeto()` to select the register, then to `i2c.readfrom_into()` to collect the data. This function does both, one after the other like so:

```
1    i2c.writeto(address, input_bytearray)
2    i2c.readfrom_into(address, output_bytearray)
```

As an example, here is what the data frames sent on the bus will look like when accessing the I^{2}C-Enabled Button's (address: `0x19`) `BUTTON_STATUS` register (register address: `0x03`) with `i2c.writeto_then_readfrom(address, output_bytearray, input_bytearray)`:



The `output_bytearray` used is simply `bytearray([0x03])`, since only one byte is required to indicate the register of interest. As a result, the byte stored in `BUTTON_STATUS` will be stored in `input_bytearray` where it can be used by the Raspberry PiPico's program. Since this register only stores one byte, `input_bytearray` will only contain one byte in this example.

ByteArrays

Byte arrays are a data array of bytes. This is super useful with I2C because it allows users to create arrays of bytes for the coordinator to send. An example of how to use them is below

```
    w = bytearray([0x02]) #converts hexadecimal into bytes
    x = bytearray(2) # create an array of 2 bytes which are zero initialized
    y = bytearray([1,2,3]) # create an array of 3 bytes with the elements 1, 2, 3
    z = bytearray("Hello World!\n") # convert the string "Hello World!\n" into a bytearray
    print(x[0]) # access the 0th element of x and print it
    print(y[2]) # access the 2nd element of y and print it
```

Which will print:

```
0
3
```

Acquiring the lock (How to read)

In order to communicate on the I2C line, CircuitPython requires that you acquire the I2C lock, this prevents multiple processes from trying to drive the I2C line at the same time. After you are done using the I2C line you should release the lock. An example of how to do this is below:

```
def foo():
    data = bytearray(1) #create a byte array to hold data
    # Try to acquire the lock
    while not i2c.trylock():
        time.sleep(0.1) # if lock acquisition fails, wait 0.1 seconds then try again
    i2c.readfrom_into(MY_ADDRESS, data) # doing i2c stuff
    i2c.unlock() # I'm done with the I2C line for now
    return int(data[0]) # return the 0th element of the array as an int
```

Starter Code

Find the file `Lab8StarterCode.py` on canvas and copy paste the contents into the file `CIRCUITPY/code.py`. We suggests that you download the contents and open `Lab8StarterCode.py` using Mu Python since copy pasting from a web browser can often introduce errors in character encodings with quotation marks (`"`) or carriage return (`\r`).

1. TMP102 Temperature Sensor

The TMP102 is an I^2C-enabled temperature sensor. Your goal for this section is to read the temperature data off the device and convert it to a floating point number and print it to the serial monitor.

How the TMP102 Works

The TMP102 has a 16-bit register that contains the current temperature as a 12-bit binary number:

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
TEMP[11:0]												Reserved			
r	r	r	r	r	r	r	r	r	r	r	r				

TEMP = raw temperature value, r = read only, Reserved = not used.

Figure 5: TMP102 Register Layout

To access this value,

1. Acquire the lock using `i2c.trylock()` as specified previously
2. Create a 2 byte `bytearray` variable using `bytearray(2)`
3. Read in two bytes of data using `i2c.readfrom_into()`. The 0th byte read will contain the eight (8) most-significant bits (MSB) and the 1st byte read will contain the four (4) least-significant bits (LSB) followed by four (4) zeros.
4. Release the lock using `i2c.unlock()`
5. Combine these two numbers together by shifting the MSB 4 bits to the left (using the left shift operation (`<<`)) then ORing it using the bitwise OR operation (`|`) with the LSB shifted 4 bits to the right (using the right shift operation (`>>`)).
   In summary: `data = (msbs << 4) | (lsbs >> 4)`
6. Convert this raw temperature data to Celsius, then `return` the value. Each bit in raw temperature value is equivalent to 1/16th of a degree Celsius (1 bit = 0.0625 C). Multiply the raw data by 0.0625 to get the final temperature value.

3 pts

1.1 Using the process outlined above, convert the following hex values into the binary register values and use it to figure out the temperature.

Hex Value	**Register Value (base-2)**	**Temperature (°C)**
0x0000	0000 0000 0000 0000	0
0x0080	0000 0000 1000 0000	.5
0x0440	0000 0100 0100 0000	4.25
0x1900	0001 1001 0000 0000	25
0x7FF0	0111 1111 1111 0000	127.94

5 pts

1.2 Connect the temperature sensor to the Raspberry Pi Pico as shown in Figure 3. Using the starter code provided on Canvas, fill in the `readTemp()` function. If your code works correctly, you should see the temperature in Celsius printed out in the Serial Monitor. **Make sure you actually call the method when you test**.

```
# @brief    read temperature in Celsius as a floating point number
# @return   temperature in Celsius as a floating point number
def readTemp():
    # TODO: implement readTemp
    while not i2c.try_lock():
        time.sleep(0.1)
    # TODO: put your I2C communication here
    i2c.unlock()
    # TODO: write temperature calculation here
    pass
```

Paste your code here:

```
def readTemp():
```

def readTemp():
  # TODO: implement readTemp
  data = bytearray(2)
  while not i2c.try_lock():
    time.sleep(0.1)
  # TODO: put your I2C communication here
  i2c.readfrom_into(TMP_ADDR,data)
  msbs = data [0]
  lsbs = data[1]
  data = (msbs << 4) | (lsbs >> 4)
  # TODO: write temperature calculation here
  data = data * 0.0625
  i2c.unlock
  return data
  pass

Once you confirm your code is working, we'll connect the ADALM2000's Logic Analyzer to look at the I^2C messages sent by the Raspberry Pi Pico. We'll use the Digital I/O pins `D0` and `D1` on the ADALM.

As Figure 6 shows, the pink box is the `Ground` pin and connects to the `GND` pin on the `qwiic` cable. The red box indicates the Digital I/O pins `D0` and `D1`. Connect `D0` to `SCL` and `D1` to `SDA`.

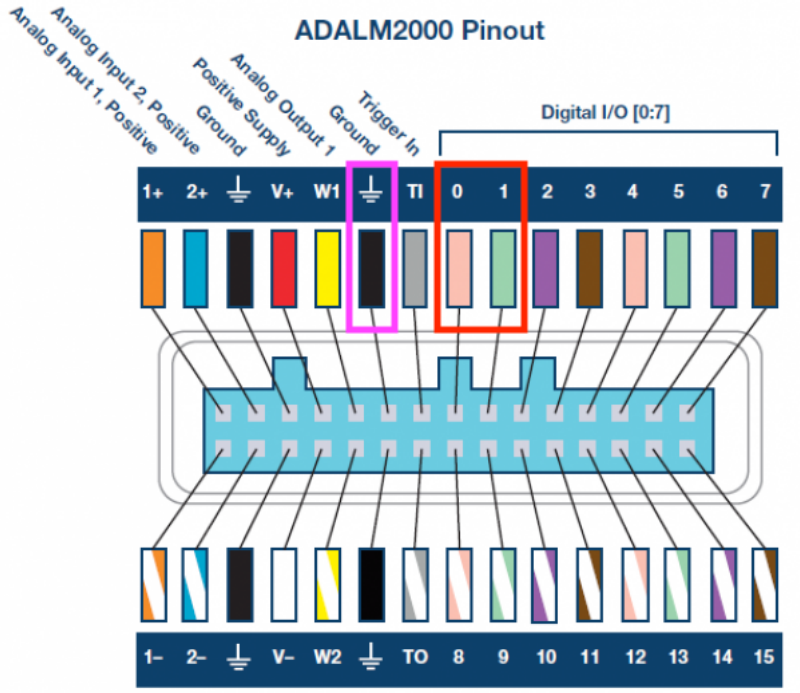
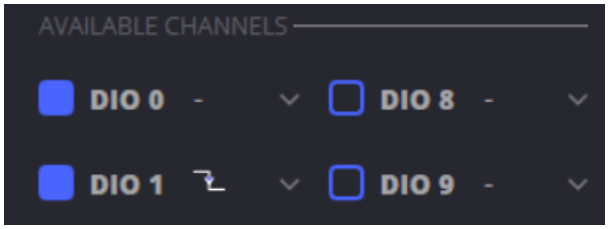
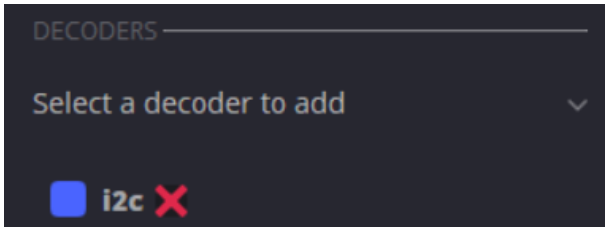


Figure 6: GND, D0, D1 on ADALM2000

Connect the ADALM2000 to your computer and open up `Scopy`. On the left side menu, select `Logic Analyzer`. (You can read up on the Logic Analyzer here: https://wiki.analog.com/university/tools/m2k/scopy/logicanalyzer) Enable `DIO0` and `DIO1` lines and select the `i2c` decode option. Set `DIO1` to a falling-edge trigger.

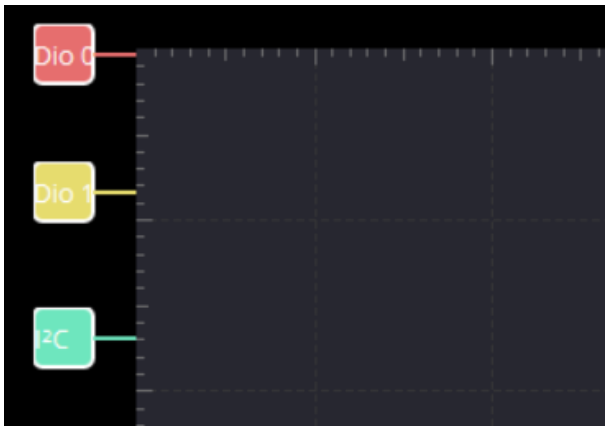


(a) Enable DIO0 and DIO1

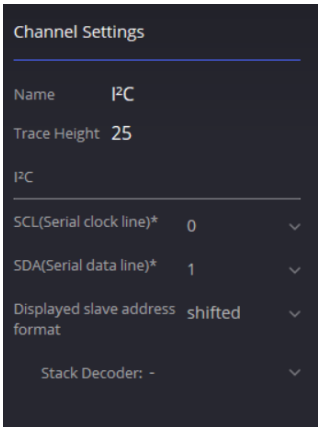


(b) Setup the I^2C Decoder

Next we're going to group `DIO0`, `DIO1` and the `I2C` Decoder. Click the "Group" button and double click each of the channels (a white border should appear). Then click "Done." Then, in the I2C Settings menu, set `SCL` to 0 and `SDA` to 1.



(a) Selected Group Items



(b) Assign Channels in I2C Decoder

Click the gear button on the top-right corner. Adjust the following settings:

Logic Analyzer
Settings
Sample Rate: 5Msps Nr of Samples: 5k samples Run Mode: Oneshot Delay: -100

Click `Single`. You should see the bit stream like that in Figure 9.

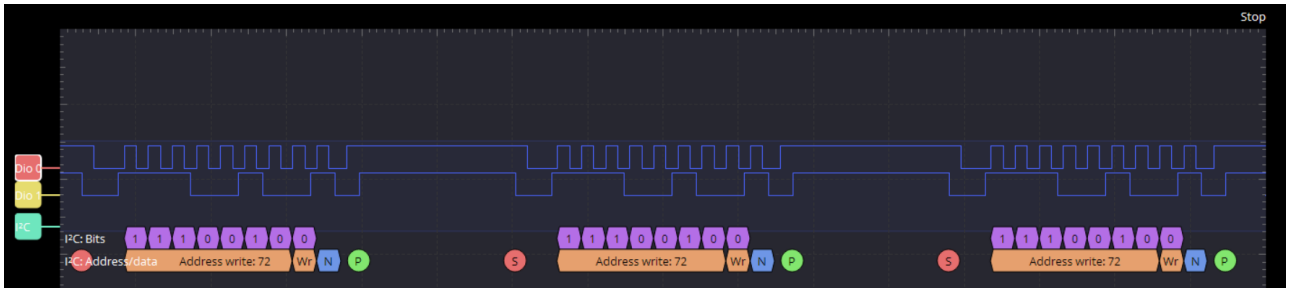
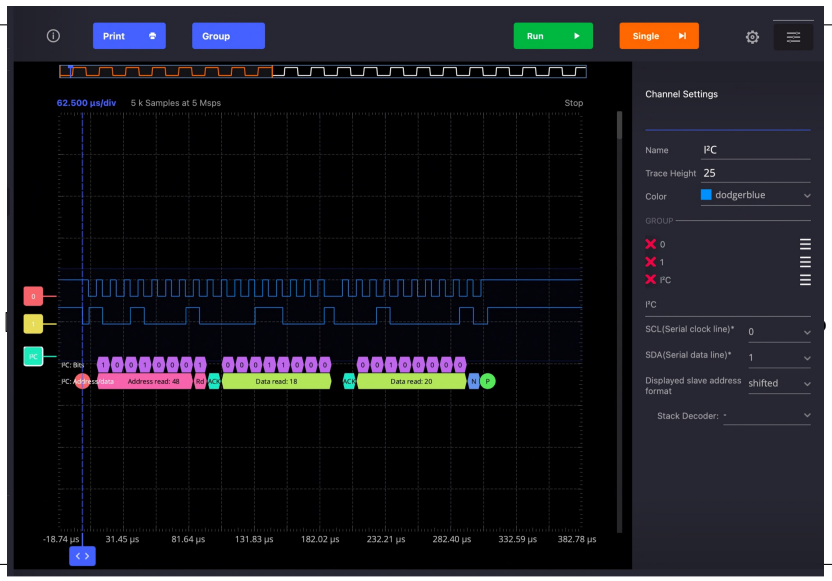


Figure 9: I^2C Bit Stream

4 pts **1.3** Paste a screenshot of your Logic Analyzer Output. Make sure your output includes all parts of the message.



1 pts **1.4** What do S and P on the logic analyzer refer to?

S is start and P is stop

2 pts **1.5** What does the 'A' at the end of each frame refer to? Who sends this bit (coordinator or participant)? What does it mean and what does it indicate about the transmission?

A refers to acknowledge, showing that that the transmission happened. This can be sent by both the coordinator and the participant with different situations. This shows the succeeded transmission so the next data can continue to be send.

1 bonus **1.6** Why is a NACK sent after the last byte?

This makes sure that all the bits are received to the coordinator from the participant

2. I²C-Enabled Button

For this section of the lab, we have provided you with a device that contains a button and an LED connected to a microcontroller which is able to process I²C data. Unlike the the temperature sensor, we want to be able to read and modify parameters of the button device (specifically the on-board LED). In order to do this, we're going to need to access other registers on the device.

Device Registers

The I²C-enabled Button device we've provided for this lab actually has 16 different registers! However for this lab, we're going to focus on two of them: `BUTTON_STATUS` and `LED_BRIGHTNESS`.

```
ID = 0x00,
FIRMWARE_MINOR = 0x01,
FIRMWARE_MAJOR = 0x02,
BUTTON_STATUS = 0x03,
INTERRUPT_CONFIG = 0x04,
BUTTON_DEBOUNCE_TIME = 0x05,
PRESSED_QUEUE_STATUS = 0x07,
PRESSED_QUEUE_FRONT = 0x08,
PRESSED_QUEUE_BACK = 0x0C,
CLICKED_QUEUE_STATUS = 0x10,
CLICKED_QUEUE_FRONT = 0x11,
CLICKED_QUEUE_BACK = 0x15,
LED_BRIGHTNESS = 0x19,
LED_PULSE_GRANULARITY = 0x1A,
LED_PULSE_CYCLE_TIME = 0x1B,
LED_PULSE_OFF_TIME = 0x1D,
I2C_ADDRESS = 0x1F
```

Figure 10: Sparkfun Qwiic Button Register Map

7	6	5	4	3	2	1	0
Reserved					ISPRES	BEENCL	AVAIL
					r	rw	rw

r = read only, rw = readable/writable

Figure 11: Button Status Register

Bits 7:3 Reserved, not used.

Bit 2 `ISPRES`: Button is pressed
- 0: Button not pressed
- 1: Button pressed

Bit 1 `BEENCL`: Button has been clicked (not used in this lab)

Bit 0 `AVAIL`: Button has been clicked (not used in this lab)

7	6	5	4	3	2	1	0
BRIGHT[7:0]							
rw	rw	rw	rw	rw	rw	rw	rw

Figure 12: Brightness Register

Bits 7:0 `BRIGHT`: LED brightness
- 0: LED off
- 1-254: Brightness levels between off and max
- 255: LED max brightness

To access these registers, all we have to do is write a byte to the bus with the address of the register we want to access and then send/request the data we want.

3 pts

2.1 Fill in the `readBtnStatus()` function. After obtaining your data from the i2c communication, you'll need to get Bit 2 from the `BUTTON_STATUS` register. You can do that by *masking* the bit pattern (`register_data & 0x04`, i.e. bitwise AND the data read from the register with `0x04` (hex for `0000 0100`) to clear all the other bits) and return the value as a `bool`.[1] *Hint: you will want to use* `i2c.writeto_then_readfrom()` *If you're stuck, reference the functions at the beginning. What bytearrays should be the input? the output?*

```
# @brief    check if the button has been pressed
# @return   whether or not the button has been pressed
def readBtnStatus():
    pass
```

Paste your code here:

```
def readBtnStatus():
```

```
def readBtnStatus():
    # TODO: implement readBtnStatus, remove pass if implemented
    Button_status = bytearray([0x03])
    dataRead = 0x04
    while not i2c.try_lock():
        time.sleep(0.1)
    # TODO: put your I2C communication here
    12c.writeto_then_readfrom(BTN_ADDR,Button_status,Button_status)
    i2c.unlock()
    return bool(dataRead & Button_status[0])
    pass
```

3 pts

2.2 Fill in the `writeBtnLED()` function. For this problem you can simply write the brightness value to the `LED_BRIGHTNESS` register. Think about how you can use bytearrays and one line of i2c communication code to write `reg_addr` followed by `brightness` to the button.

```
# @brief        set the button LED Brightness
# @param[in]    brightness (0-255) desired brightness the button LED
# @param[in]    reg_addr address to write to
def writeBtnLED(brightness, reg_addr):
    pass
}
```

Paste your code here:

```
def writeBtnLED(brightness, reg_addr):
```

```
def writeBtnLED(brightness, reg_addr):
    # TODO: implement writeBtnLED, remove pass if implemented
    while not i2c.try_lock():
        time.sleep(0.1)
    # TODO: put your I2C communication here
    Button = bytearray([reg_addr, brightness])
    12c.writeto(BTN_ADDR,Button)
    i2c.unlock()
    pass
```

[1] Fun fact: boolean values under-the-hood are actually 8 bit numbers where 0 = false, and anything else = 1! This allows to simplify our conditionals quite easily!

3 pts

2.3 Modify your `while(True):` loop to turn the LED on (full brightness) whenever the button is pressed. Do this using your `writeBtnLED()`/`readBtnStatus()` functions.
Paste your code here:

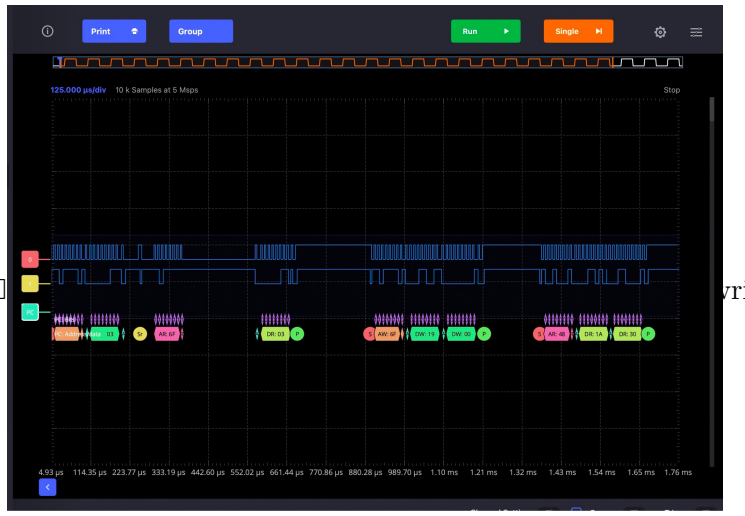
```
while(True):

    while True:
        if readBtnStatus():
            writeBtnLED(255,0x19)
        else:
            writeBtnLED(0,0x19)
```

6 pts

2.4 Using the logic analyzer, paste screenshots for the following scenarios:

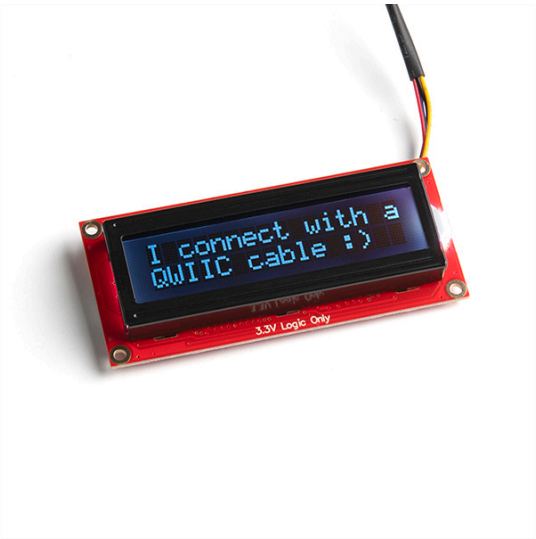
i. When the button is not pressed. (You may have to adjust Logic Analyzer settings so that all data frames are visible.)



ii. When the button is pressed.

3. I²C LCD Display

For this section of the lab, we have provided you an LCD display with an on-board microcontroller which is able to process I²C data. Unlike the the temperature sensor and the button, this device uses a command system. The display has two rows of 16 characters each.



The display will connect to your I²C bus via the Qwiic connector on the back (as shown below).

We can write a line of code like `i2c.writeto(LCD_ADDR, "sample text\r")` to send text to the LCD. Note that we did not have to prefix the second arguement with the command byte. When we want to use a built-in command we will prefix our command with the byte `0x7C`.

To clear the display we will send the bytes `0x7C` (command byte) and the `0x2D` (clear display byte). Again, this can be done with one line of i2c communication code and bytearrays.

Another command is to change the backlight color of the LCD. Sending the command byte `0x7C`, then background color command, `0x2B` and then three 8-bit values. This will change the backlight red/green/blue channels based on the 3 values respectively. (for example `0x7C, 0x2B, 0xFF, 0xFF, 0xFF` will turn the backlight to full white)

5 pts **3.1** Modify the `while(True)` at the end of your code to do the following:

i. Print a line text on the first line of the display (can be anything you want, "Hello World," "I <3 18100," etc.) that changes in some way when the button is pressed

ii. Print the current temperature in Celsius (in a similar format `print()` statement in the starter code)

Make sure to clear your display before writing new information to it! (i.e. on every while(True) loop iteration.

Paste your Lab 8 code here (no need to include `readTemp()`, `readBtnStatus()` or `writeBtnLED()`):

```
# @brief    clear the LCD
def clearLCD():
    while not i2c.try_lock():
        time.sleep(0.1)
    hihi = bytearray([0x7c, 0x2D])
    i2c.writeto(LCD_ADDR,hihi)

    ic2.unlock()

def printLCD(pressed, temp):
    while not i2c.try_lock():
        time.sleep(0.1)
    if pressed == True:
        i2c.writtento(LCD_ADDR, "Hello World\r")
    else:
        ic2.writeto(LCD_ADDR, "{}".format(temp))
        i2c.unlock

def setBackLight(r, g, b):
    # TODO: BONUS: implement setBackLight remove pass if implemented
    pass
lightCounter = 0
while True:
    if readBtnStatus():
        writeBtnLED(255,0x19)
    else:
        writeBtnLED(0,0x19)
    # uncomment this to check which devices are connected
    # print([hex(i) for i in checkDevices()])
    print("It's a lovely {} C today!".format(readTemp()))
    led.value = bool(lightCounter % 2)
    clearLCD(readBtnStatus(), readTemp())
    lightCounter += 1
    # TODO: you'll want to tune this delay to get more frequent results
    time.sleep(0.5) # loop delay
# technically we need to release the I2C object but for our purposes we never will
i2c.deinit()
```

15 pts **3.2** Be prepared to check off your functioning I^2C display circuit.

3 bonus **3.3** Write a function, `setBacklightColor()` that takes in an address and 3 bytes, red, green, and blue and changes the color of the display to given RGB value.