18-100: Intro to Electrical and Computer Engineering

LAB06: I^{2}C Lab

Writeup Due: Wednesday, March 19th, 2025 at 10 PM

Checkoffs Due: Wednesday, March 19th, 2025 at 10 PM

Name: ______________________________

Andrew ID: ______________________________

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. This demo will occur during office hours.

Question:	1	2	3	4	Total
Points:	18	22	2	10	52
Score:					

Lab Outline

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

Sections

1. Introduction to I^{2}C
2. Temperature Sensor
3. I^{2}C-enabled Button
4. I^{2}C EEPROM
5. LCD Display and Software Libraries
6. Putting it all together

Small Group Check-off Circuits

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

Required Materials

- 1x ADALM2000
- 1x Arduino Nano ESP32
- 1x I^{2}C TMP102 Temperature Sensor
- 1x I^{2}C Button w/ LED
- 1x I^{2}C EEPROM Module
- 1x I^{2}C Sparkfun Display
- 2x JST F-F Cables
- 1x JST Breakout M
- 1x JST Breakout F

Introduction to I^{2}C

The Inter-Integrated Circuit (I^{2}C, “I two C” or “I squared C”) Protocol is a protocol intended to allow multiple “devices” to communicate with one or more “coorindators”.

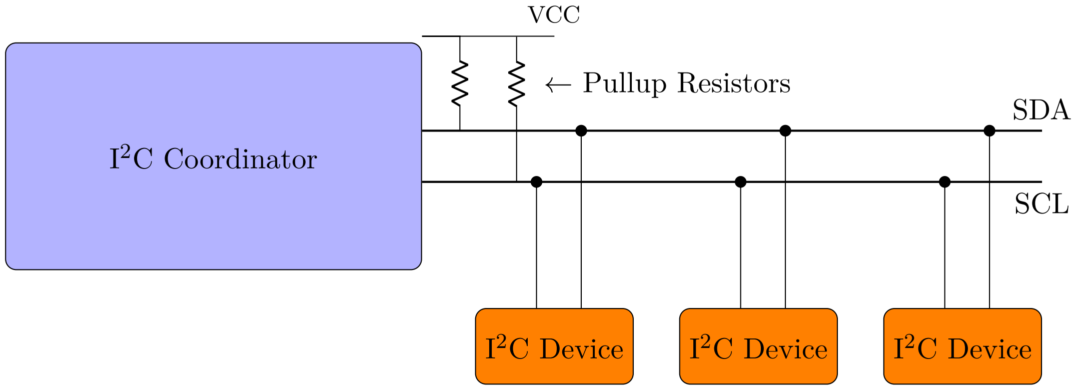


Figure 1: I^{2}C Bus Diagram

I^{2}C Hardware

The main benefit of I^{2}C 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 Arduino Nano ESP32 I^{2}C 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”).

I^{2}C 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.

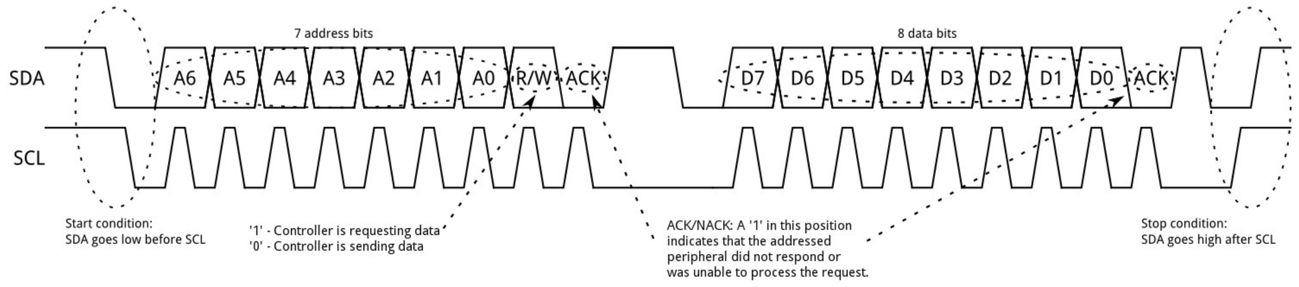

7 address bits
8 data bits
SDA
SCL
A6
A5
A4
A3
A2
A1
A0
R/W
ACK
D7
D6
D5
D4
D3
D2
D1
D0
ACK
Start condition: SDA goes low before SCL
'1' - Controller is requesting data
'0' - Controller is sending data
ACK/NACK: A '1' in this position indicates that the addressed peripheral did not respond or was unable to process the request.
Stop condition: SDA goes high after SCL

Figure 2: I^{2}C 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 the SDA line transitioning from low to high while SCL is 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/A (0) or NACK/N (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.

Arduino Nano ESP32 I^{2}C Support

The Arduino Nano ESP32 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 Arduino Nano ESP32, the I^{2}C pins are A4 (`SDA`) and A5 (`SCL`).** Usually, the device requires two additional pins: the power 3V3(OUT) (`3V3`) and the ground pin (`GND`).

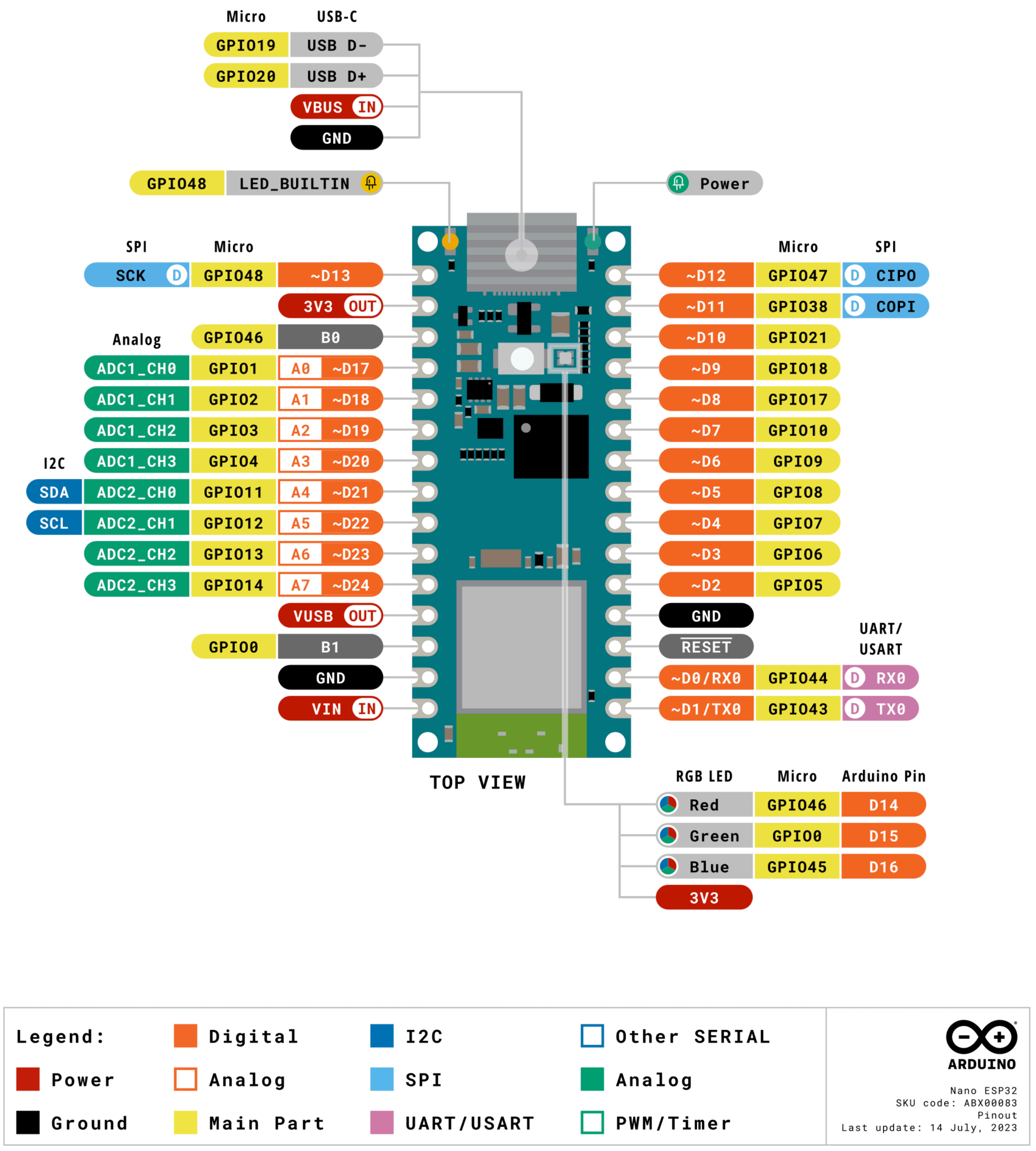
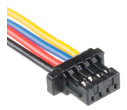
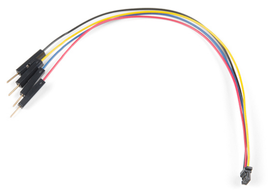
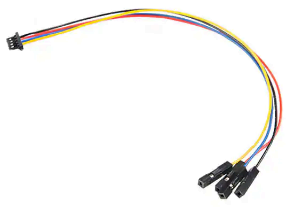


Figure 3: Arduino Nano ESP32 Pin out

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	Nano Net	Nano Pin Number
SCL	Yellow	I^2C SCL	A5
SDA	Blue	I^2C SDA	A4
VCC	Red	3V3(OUT)	3V3
GND	Black	GND	GND

Table 1: Summary of Connections

Bytearrays

This lab includes a bytearray library to make it easier to send data over I^2C. This library contains `bytearray.hpp`, which is the header file to include in your main code, and `bytearray.cpp`, which contains the full source code of this library.

Byte arrays are a data array of bytes. This is super useful with I^2C because it allows users to create arrays of bytes for the coordinator to send.

Since the `Wire` library from Arduino can only send a maximum of 32 bytes over I^2C at a time, **the length of a bytearray is limited to a maximum of 32 bytes. Creating a bytearray longer than this will result in an error.**

Below are the constructors and methods of the bytearray class:

```
    // create a zero-initialized bytearray "x", containing 2 bytes
    bytearray x(2);
    // x contains { 0, 0 }

    // create an int array "arr" with elements 1, 2, and 3
    int arr[3] = { 1, 0x02, 3 };
    // create a bytearray "y" from the int array "arr"
    bytearray y(arr, 3);
    // y contains { 1, 2, 3 }

    // creates a bytearray "z" from the Arduino String "Hello World!"
    bytearray z(String("Hello␣World!"));
    // z contains the String "Hello World!" in ASCII

    // prints the bytearray "z"
    z.print();
    // the ASCII value of each character in "z" will be printed


    // gets the length of bytearray "z"
    int len = z.length();
    // len will have a value of 12

    // converts "z" to a String
    String str = z.toString();
    // str will contain "Hello World!"

    // creates a bytearray "w" by concatenating "x" and "y"
    bytearray w = x + y;
    // w will now contain \texttt{\{ 0, 0, 1, 2, 3 \}}

    // concatenates bytearray "x" to the end of "w"
    w += x;
    // w will now contain { 0, 0, 1, 2, 3, 0, 0 }
```

Debugging Bytearrays

A bytearray can be printed

If a runtime error occurs when using a bytearray, code execution will be halted on the Arduino Nano ESP32 and an error message will be printed to the serial console. This could be caused by a few things:

- Attempting to create a bytearray that exceeds the max length of 32 (31 from Strings)

- Attempting to concatenate bytearrays with a total length that exceeds the max length of 32

- Creating a bytearray from an `int` array containing values greater than 255 (0xFF), or negative values

- Indexing out of bounds

Note: when creating a bytearray from an `int` array, you must enter a length that is smaller than or equal to the length of the `int` array. It is impossible for the bytearray library to check the actual length of the `int` array and throw an error, so this can cause undefined behavior. Always create bytearrays immediately after the `int` array to prevent the wrong length from being entered.

I^{2}C Helper Functions

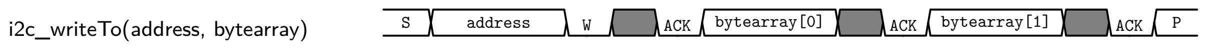
The helper functions are included in `i2c_helper.hpp` and `i2c_helper.cpp`. The following functions are useful in this lab:

`i2c_writeTo(address, bytearray)`

Writes the data in `bytearray` to the I^{2}C 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.

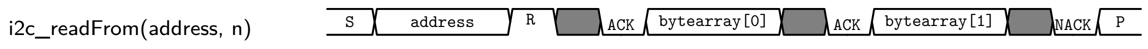
If a device contains multiple registers, the first byte sent, `bytearray[0]`, will be the register of interest. For example, the I^{2}C-Enabled Button has 16 registers, each with their own 8-bit address. For example, the `LED_BRIGHTNESS` register has an address of `0x19`. Here is an illustration of the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



`i2c_readFrom(address, n)`

Requests and reads n bytes from the I^{2}C Address `address`, and returns a bytearray of length `n` containing the bytes.

This function results in the controller sending an address frame, then receiving a number of data frames from the selected device. Here is an illustration of the data frames that are sent on the bus when 2 bytes are requested:

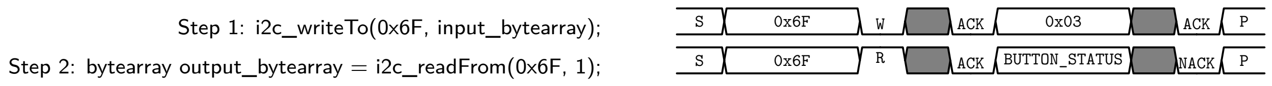


Notice the similarity of data sent on the I^{2}C between this function and the previous one, `i2c_writeTo()`. The only difference is the status of the `R/W` bit sent with the address.

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()` to collect the data like so:

```
    i2c_writeTo(address, input_bytearray);
    bytearray output_bytearray = i2c_readFrom(address, n);
```

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: `0x6F`) `BUTTON_STATUS` register (register address: `0x03`) with `i2c_writeTo()` and `i2c_readFrom()`:


Step 1: i2c_writeTo(0x6F, input_bytearray);
Step 2: bytearray output_bytearray = i2c_readFrom(0x6F, 1);
S
0x6F
W
ACK
0x03
ACK
P
S
0x6F
R
ACK
BUTTON_STATUS
NACK
P

The `input_bytearray` used is a bytearray of length 1, with `0x03` stored as the 0th element. As a result, the byte stored in `BUTTON_STATUS` will be stored in `output_bytearray` where it can be used by the Arduino program. Since this register only stores one byte, 1 byte should be requested for the read, and `output_bytearray` will only contain one byte in this example.

Starter Code

Download `i2c_starter.zip` from Canvas, extract the archive, and open `i2c_lab.ino` using the `File -> Open` menu in Arduino IDE. Then plug in your Arduino Nano ESP32 board using the USB-C to USB-A cable. Use the drop-down menu at the top of the Arduino IDE to select the Arduino Nano ESP32 as the current board.

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

Each bit in raw temperature value is equivalent to 1/16th of a degree Celsius (1 bit = 0.0625 C). Thus if we know the 12-bit value stored in the register, we can multiply the raw binary data by 0.0625 to get the corresponding temperature value.

3 pts

1.1 Convert the following hex values into the binary register values. Then, take the 12 most significant bits and use it to figure out the temperature.

As an example, say we have the hex value 0x1400. This is 0001 0100 0000 0000 in binary. The twelve most significant bits (0001 0100 0000) converted to decimal is 320. So the corresponding temperature in Celsius is 320(0.0625) = 20°C.

Hex Value	**Register Value (base-2)**	**Temperature (°C)**
0x1400	$0001\ 0100\ 0000\ 0000_2$	20
0x0000		
0x0080		
0x0440		
0x1900		
0x7FF0		

When sending data over I^{2}C, we can only send one byte in a frame. Therefore we will need to read the 16-bit register value from the TMP102 as two separate bytes and then use some bit operations to remove the reserved bits and reconstruct the 12-bit value we want. The overall process for getting this value and converting it to the corresponding temperature is outlined by the following steps:

1. Read in two bytes of data a the byte array using `i2c_readfrom()`. 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.
2. Combine these two numbers together by using the `convertTemp()` function given to you in the code.
3. Return this floating point value. This is the value your temperature sensor is reading in Celsius.

4 pts

1.2 Connect the temperature sensor to the Arduino Nano as shown in Figure **??**. Using the starter code provided on Canvas, fill in the `getTemp()` and `readCurrentTemp()` functions according to the steps highlighted above. If your code works correctly, you should see the temperature in Celsius printed out in the Serial Monitor. **Make sure you actually call the `readCurrentTemp()` in your `loop()` when you test**. Ensure that each function matches the specifications given in the function header comments.

```
/** @brief Reads the temperature bytes from the temperature sensor
    @return Returns the temperature as a bytearray */
bytearray getTemp() {

  /* Your solution here */

  return bytearray(1);  // modify this
}
```

```
/** @brief Returns the current temperature from the sensor
    @return Returns the temperature as a float */
float readCurrentTemp() {

  /* Your solution here */

  return 0.0;           // modify this
}
```

Paste a screenshot of your code for `getTemp()` and `readCurrentTemp()` here:

`Paste Screenshot Here`

☐ I have appended the screenshot to the back of my lab writeup

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

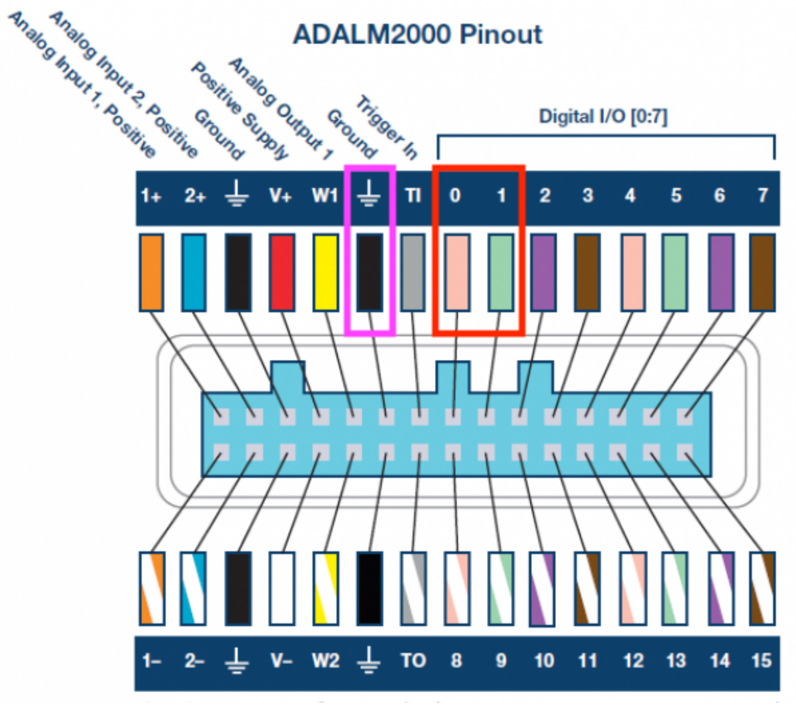
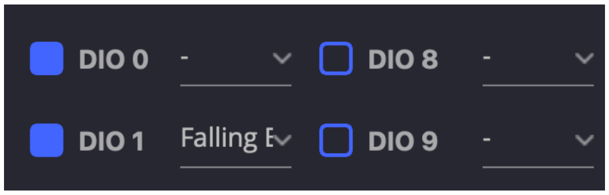


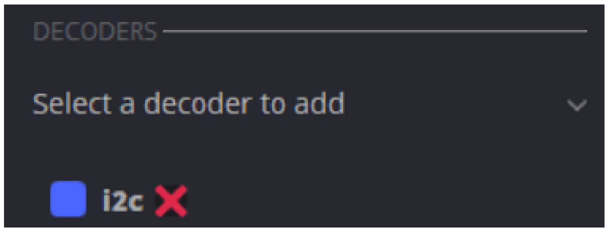
Figure 6: GND, D0, D1 on ADALM2000

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`. Do not connect the red wire.

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.

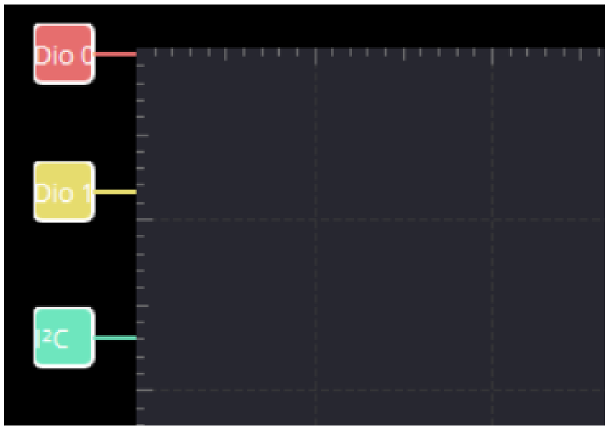


(a) Enable DIO0 and DIO1

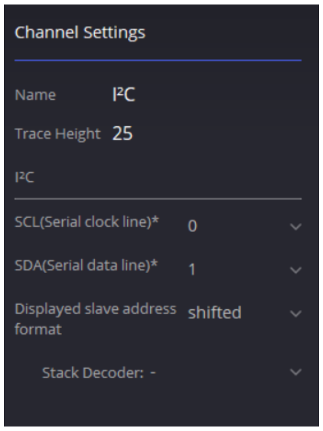


(b) Setup the I^{2}C Decoder

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



Channel Settings

Name I²C

Trace Height 25

I²C

SCL(Serial clock line)* 0

SDA(Serial data line)* 1

Displayed slave address format shifted

Stack Decoder: -

(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.

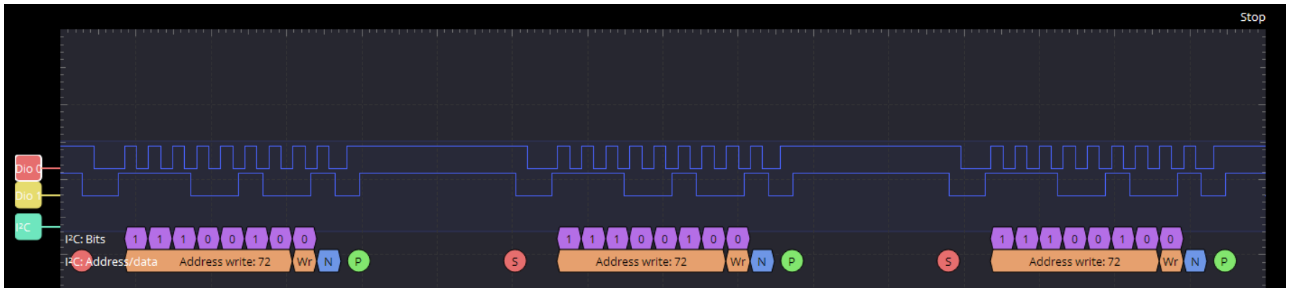


Figure 9: I^{2}C Bit Stream. Note that this is an example, you may receive a different number of bytes

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

`Paste Screenshot Here`

☐ I have appended the screenshot to the back of my lab writeup

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

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?

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

5 pts **1.7** Be prepared to check off your functioning temperature sensor circuit.

⚠ **Do NOT take your circuit apart yet! You will need it for lab checkoff!**

2. I^{2}C-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 I^{2}C 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 I^{2}C-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 `getBtnState()` function. After setting the internal pointer of the button and obtaining your data using 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`.

```
/** @brief Returns the state of the button
    @return Returns true if pressed, false if not */
bool getBtnState() {

  /* Your solution here */

  return false;          // modify this
}
```

Paste a screen shot of your code for getBtnState() here:

Paste Screenshot Here

☐ I have appended the screenshot to the back of my lab writeup

3 pts

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

```
/** @brief Sets the LED brightness of the button */
void setBtnLED(byte brightness) {

  /* Your solution here */

  return;                // remove this
}
```

Paste a screen shot of your code for setBtnLED() here:

Paste Screenshot Here

☐ I have appended the screenshot to the back of my lab writeup

2 pts

2.3 Modify your `loop()` function to turn the LED on (full brightness) whenever the button is pressed. Do this using your `setBtnLED()` and `getBtnState()` functions as well as conditionals. [1]

Paste a screen shot of your loop() code here:

`Paste Screenshot Here`

☐ I have appended the screenshot to the back of my lab writeup

[1] You'll want to comment out the code that is printing the temperature sensor information in your `loop()` function. Don't worry we will still use this later!

4 pts **2.4** Using the logic analyzer, paste screenshots for the following scenarios. Make sure that your screenshots show the entire message with the button status and the brightness value written to the device. You should set the Nr of Samples setting to 7k to see the full message.

i. When the button is not pressed.

`Paste Screenshot Here`

☐ I have appended the screenshot to the back of my lab writeup

ii. When the button is pressed.

`Paste Screenshot Here`

☐ I have appended the screenshot to the back of my lab writeup

10 pts iii. Be prepared to check off your functioning button circuit.

⚠ **Do NOT take your circuit apart yet! You will need it for lab checkoff!**

3. EEPROM

Electrically Erasable Programmable Read-Only Memory (EEPROM) is a non-volatile, high-speed storage technology. We'll be using an I^2C-based EEPROM module to store temperature sensor values for later access.

Using the EEPROM

You'll need to accomplish two operations: programming values into memory and reading values from memory. For this lab, we'll always read/write byte-by-byte. Because the memory has many different addresses we might be interested in, the first step of any request is to specify the address in memory that we want to read from or write to.

Since our EEPROM module takes an 12-bit address we'll need to send 2 separate data frames. It's important to distinguish the memory address from the device address: the memory address is a physical location within the memory module itself that stores a value (just like we discussed with computers), whereas the device address is how we talk to the EEPROM (as opposed to the button or display). The device address for our module is `0x50`, which can be seen in the "Control Byte" frame of 13.

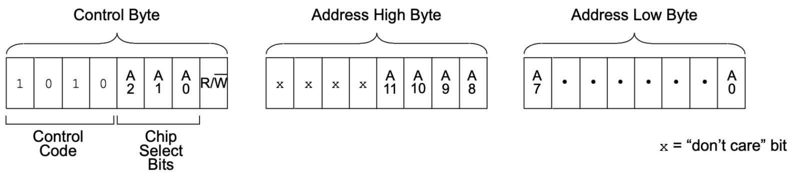


Figure 13: EEPROM Address Frame

Writing to memory is simple: call `i2c_writeTo(EEPROM_ADDR, data)` where `data` is an array containing the memory address followed by the byte you'd like to write. Keep in mind: if our address is `0x7FF`, the higher 4 bits [2] of the address are `0x7` and the lower 8 bits are `0xFF`

Reading from memory is a bit trickier: we need to send the memory address to the I^2C module before reading before we can read the data. This can be accomplished with a call to `i2c_writeTo()` with the memory address of interest followed by a call to `i2c_readFrom()`. Think carefully about how many bytes you want to read!

1 pts

3.1 Fill in the `memWrite()` function. The function takes in three arguments: `address_high`, `address_low`, `data`. `address_high` is the higher four bits of the memory address to write to. After making the I^2C write, you'll want to use a delay for $\geq$ **6 mS** (using `delay()`). This is required for the EEPROM to save the values you've written to it before moving on.

[2] Figure 13 shows the higher 4 bits of the higher byte being 'x'. This means they can technically be either 0 or 1 and are simply ignored by the EEPROM module.

```c
/** @brief Writes data to an address in the EEPROM */
void memWrite(int high_addr, int low_addr, int data) {

  /* Your solution here */

  return;               // remove this
}
```

Paste a screenshot of your your `memWrite()` solution here:

`Paste Screenshot Here`

☐ I have appended the screenshot to the back of my lab writeup

1 pts

3.2 Fill in the `memRead()` function. The function takes in two arguments: `address_high`, `address_low`. `address_high` is the higher four bits of the memory address to write to. You'll need to return the byte you read from memory as a `byte`. Unlike with `write`, you don't need to add any delay after reading.

```c
/** @brief Reads data from an address in the EEPROM
    @return Returns a byte of data */
byte memRead(int high_addr, int low_addr) {

  /* Your solution here */

  return 0;             // modify this
}
```

Paste a screenshot of your your `memRead()` solution here:

`Paste Screenshot Here`

☐ I have appended the screenshot to the back of my lab writeup

4. Putting it Together

By now, you've written code to interface with a button, temperature sensor, and EEPROM module. It's time to use them all at once to see how I^2 can be used to build a complete system!

5 pts

4.1 Modify the `loop()` function in your code to do the following:

i. Read the current temperature from the temperature sensor and print it to the serial console
ii. Fetch the button state. If pressed, turn the button on and store the value to EEPROM. After writing a value to EEPROM, increment the address for the next write by 2.
iii. Read the previously stored temperature value from the EEPROM and print it out. Then, print out the most recently stored temperature.

All of the functionality should be included without having to comment out any parts of the code.

Paste screenshots of your Lab 8 code here, including all functions and constants you've defined except those that you already included in the above questions. Your code should be well documented enough using comments that it is easily readable:

`Paste Screenshot Here`

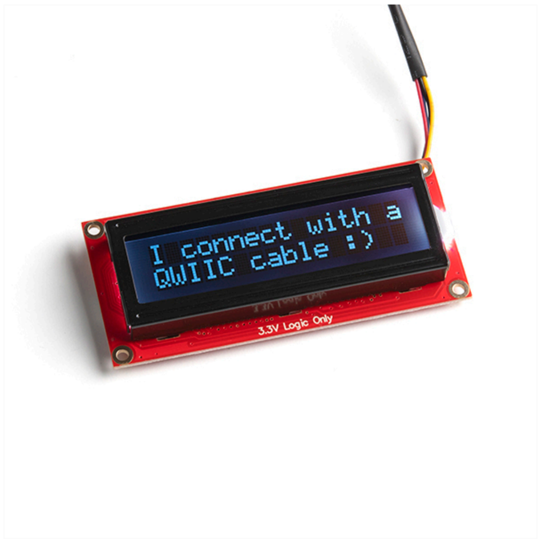
☐ I have appended the screenshot to the back of my lab writeup

5 pts

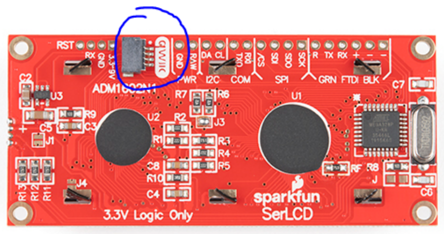
4.2 Be prepared to check off your functioning display circuit.

5. I^{2}C LCD Display (Bonus)

For this section of the lab, we have provided you an LCD display with an on-board microcontroller which is able to process I^{2}C data. Unlike 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 I^{2}C bus via the Qwiic connector on the back (as shown below).



We can write a line of code like `i2c_writeTo(LCD_ADDR, bytes)` where bytes is a byte array containing a string of characters such as "Hello World" to send a message to the LCD. Note that we did not have to prefix the second argument 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 `0x2D` (clear display byte). Again, this can be done with one line of I^{2}C 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)

3 bonus

5.1 Modify the `loop()` function in your code to add the following:

i. Print a line of text on the of the display that says something along the lines of ”Just stored a value!” when the button is pressed.
ii. When the button is not pressed print a line of text along the lines of ”Push the button to capture the temperature!”
iii. Write a function, `setBacklightColor()` that takes in 3 bytes: red, green, and blue, to change the backlight color of the display to given RGB value.

All of the functionality should be included without having to comment out any parts of the code.

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

Paste screenshots of your modified code here, including all functions and constants you've defined except those that you already included in the above questions. Your code should be well documented enough using comments that it is easily readable:

`Paste Screenshot Here`

☐ I have appended the screenshot to the back of my lab writeup

3 bonus

5.2 Be prepared to check off your functioning display circuit.