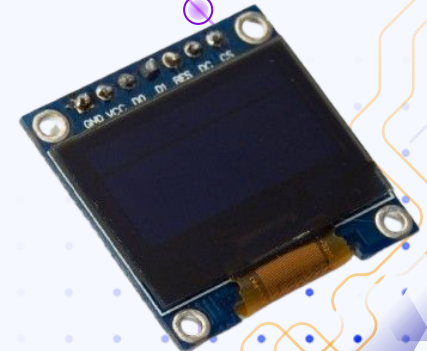


Attendance
Form

STM32 Connectivity: Mastering Peripherals

IEEE Penn State – Projects Committee
SP26



Quick Announcement!

American Society of Agricultural and Biological Engineers (ASABE) is holding the **Robotics Student Design Competition** every year during their Annual International Meeting. This year, the Robotics Competition will be held at Indianapolis, Indiana, from **July 12, 2026 to July 14, 2026**. Please review the following:

<https://asabe.org/event-detail/2026-annual-international-meeting>



Scan me for more info!

A promotional banner for the ASABE Annual International Meeting. The background is a night scene of a field with wind turbines and a city skyline. The ASABE logo and name are in the top left. The event title and location are in the center. The dates are below the title. A 'JOIN US' button is at the bottom left. Two circular insets on the right show a city skyline and a race track.

 **ASABE**
Engineering a Sustainable Future

ASABE Annual International Meeting

JW Marriott Indianapolis
Indianapolis, Indiana

 **July 12 - 15, 2026**

JOIN US

 **ASABE26**  **asabeaim.org**



Announcement Cont.

At Penn State, we are currently forming a **Beginner** Team and an **Advanced** Team to compete at the 2026 ASABE Robotics Design Competition.

If you are interested, please contact one of the faculty advisors listed below:

Student team member qualifications:

1. interested in any of robotics, programming, electrical and mechanical hardware design, Agricultural and Biological Engineering, desire to learn, etc.
2. undergraduate (freshman to senior) or graduate student, team player
3. time availability -
 - * 5hr/week
 - * available to travel and attend the competition meeting on **July 11, 2026 to July 14, 2026**
4. register 1 or more credits of CMPEN/BE 296, 496, or 596 for this summer

Prof. Kyusun Choi, Computer Science and Engineering

Email: kxc104@psu.edu

Office: 325 Leonhard Building
University Park

Office Hour:

1:00pm - 2:30pm Tue.

10:30am - 12:00pm Wed.

Prof. Shirin Ghatreh Samani, Agricultural and Biological Engineering

Email: spg5994@psu.edu

Office: 203 Agricultural Engineering Building
University Park



Let's Get Started!

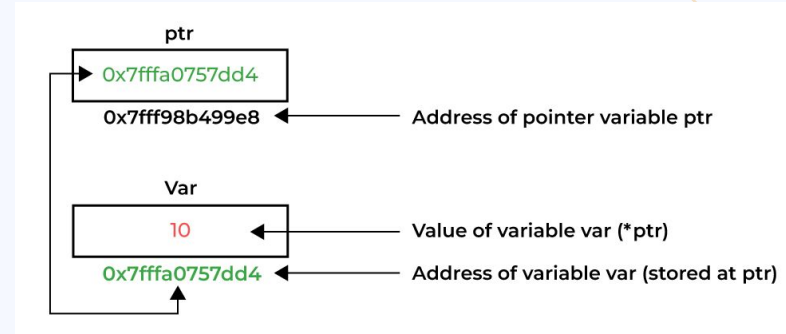


Quick Note

- We are assuming you have been with us through workshop 1, meaning:
 - You have **STM32CubeIDE 1.19.0** installed on your computer
 - You have seen the I/O Configurator before
 - You are familiar with setting up `printf()` and serial wire viewer within the CubeIDE
 - Familiarity with hexadecimal numbers (0x..) and C pointers helpful but not necessary

C Pointers

- You may see **&** and ***** symbols next to variables, what does this mean?
- Your variables are stored in RAM
- Your RAM has memory addresses, they work like a street address
- A **pointer** stores this address, allowing direct access to your variable
 - **&** - reference operator, used to **get the pointer** of a variable
 - ***** - dereference operator, used to **get the value** at that pointer
- They're used to allow the library to read and write information to your variables



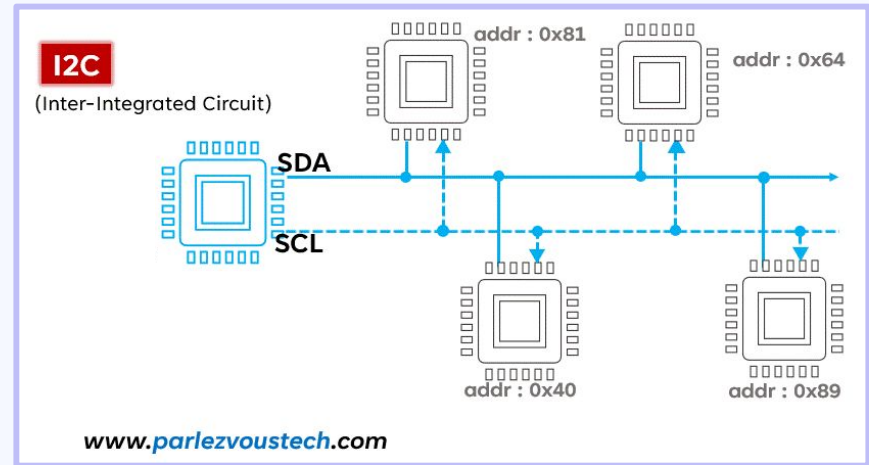


Part 1

Fundamentals of Serial Communication

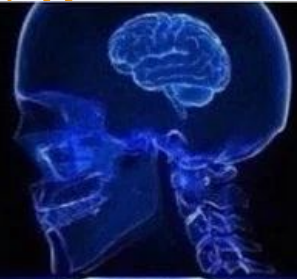
Serial Communication

- How do we communicate with other devices from our microcontroller?
→ **Serial communication** is the most resource efficient answer
- Serial communication protocols work by sending bits one after another
(in series!)
- Examples of serial communication:
 - I2C (today's topic!)
 - SPI
 - UART
 - USB
 - CAN



I2C Breakdown

I2C



I²C



-1C



-299792458M/S



I2C Overview

- I2C is a cost-effective, two wire serial protocol (a.k.a. **TWI**)
- I2C is a **synchronous** communication protocol

SCL & SDA pins on STM32



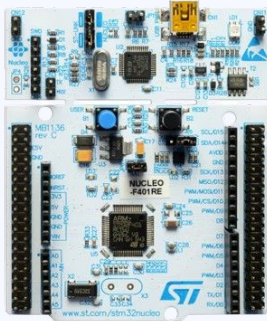
Serial <u>C</u> lock (SCL) line	Serial <u>D</u> ata (SDA) line
<ul style="list-style-type: none">• controls when new data is ready• usually runs at 100 kHz but can go up to 400 kHz	<ul style="list-style-type: none">• where data is sent• bi-directional, meaning data can travel to & from host on the <u>same wire</u>

- Peripherals (including screen & sensor used today) generally use a register-based communication method

I2C Architecture

Controller (Master)

- Is in full control of the SCL line
- Initiates all communication
- Can write to or read from the peripheral device



yo
gurt

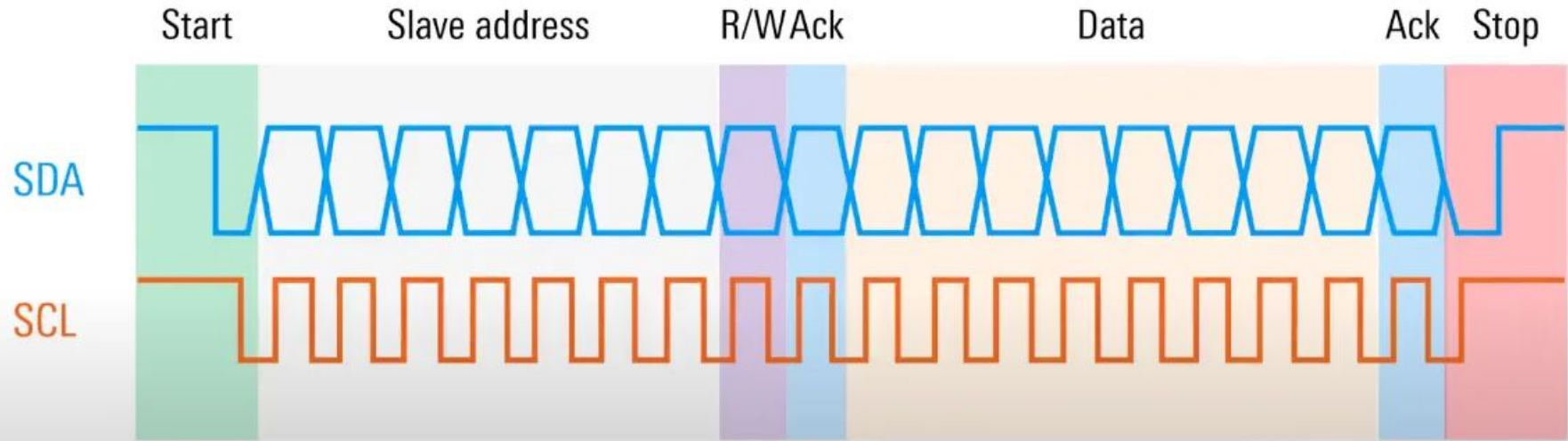
Peripheral (Slave)

- Listens and waits for a START signal from controller
- Data is synced by SCL line
- Will only respond to a predefined address

gurt:



I2C Frame Overview



Register? I hardly know her

- Most I2C devices work using a protocol made of **register addresses**
- Each address contains data or can control a different thing

3B	59	ACCEL_XOUT_H	R	ACCEL_XOUT_H[15:8]
3C	60	ACCEL_XOUT_L	R	ACCEL_XOUT_L[7:0]
3D	61	ACCEL_YOUT_H	R	ACCEL_YOUT_H[15:8]
3E	62	ACCEL_YOUT_L	R	ACCEL_YOUT_L[7:0]
3F	63	ACCEL_ZOUT_H	R	ACCEL_ZOUT_H[15:8]
40	64	ACCEL_ZOUT_L	R	ACCEL_ZOUT_L[7:0]
41	65	TEMP_OUT_H	R	TEMP_OUT_H[15:8]
42	66	TEMP_OUT_L	R	TEMP_OUT_L[7:0]
43	67	GYRO_XOUT_H	R	GYRO_XOUT_H[15:8]
44	68	GYRO_XOUT_L	R	GYRO_XOUT_L[7:0]
45	69	GYRO_YOUT_H	R	GYRO_YOUT_H[15:8]
46	70	GYRO_YOUT_L	R	GYRO_YOUT_L[7:0]
47	71	GYRO_ZOUT_H	R	GYRO_ZOUT_H[15:8]
48	72	GYRO_ZOUT_L	R	GYRO_ZOUT_L[7:0]

- Some registers set pixels on the screen
- Some contain the acceleration data from our accelerometer
- These registers can be found using the manufacturer's provided "**register map**"

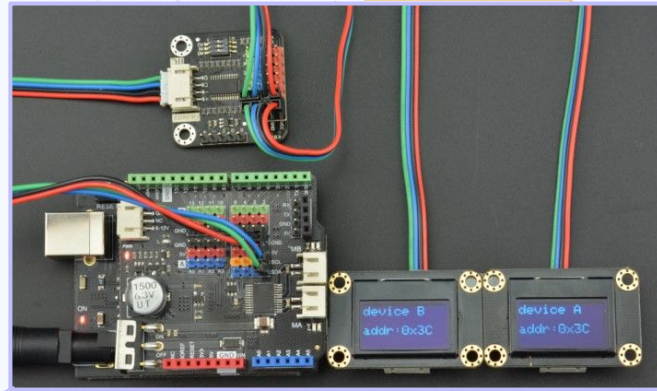


Part 2


Using Peripherals with I2C + HAL

I2C on the STM32

- The STM32 has three I2C interfaces built into the chip
 - each can support up to 128 devices
- Standard 100 kHz or **Fast Mode** (up to 400 kHz)
- Multiple interfaces → multiple sensors can be used (sharing predefined address)
 - you might need a triple OLED set up



...

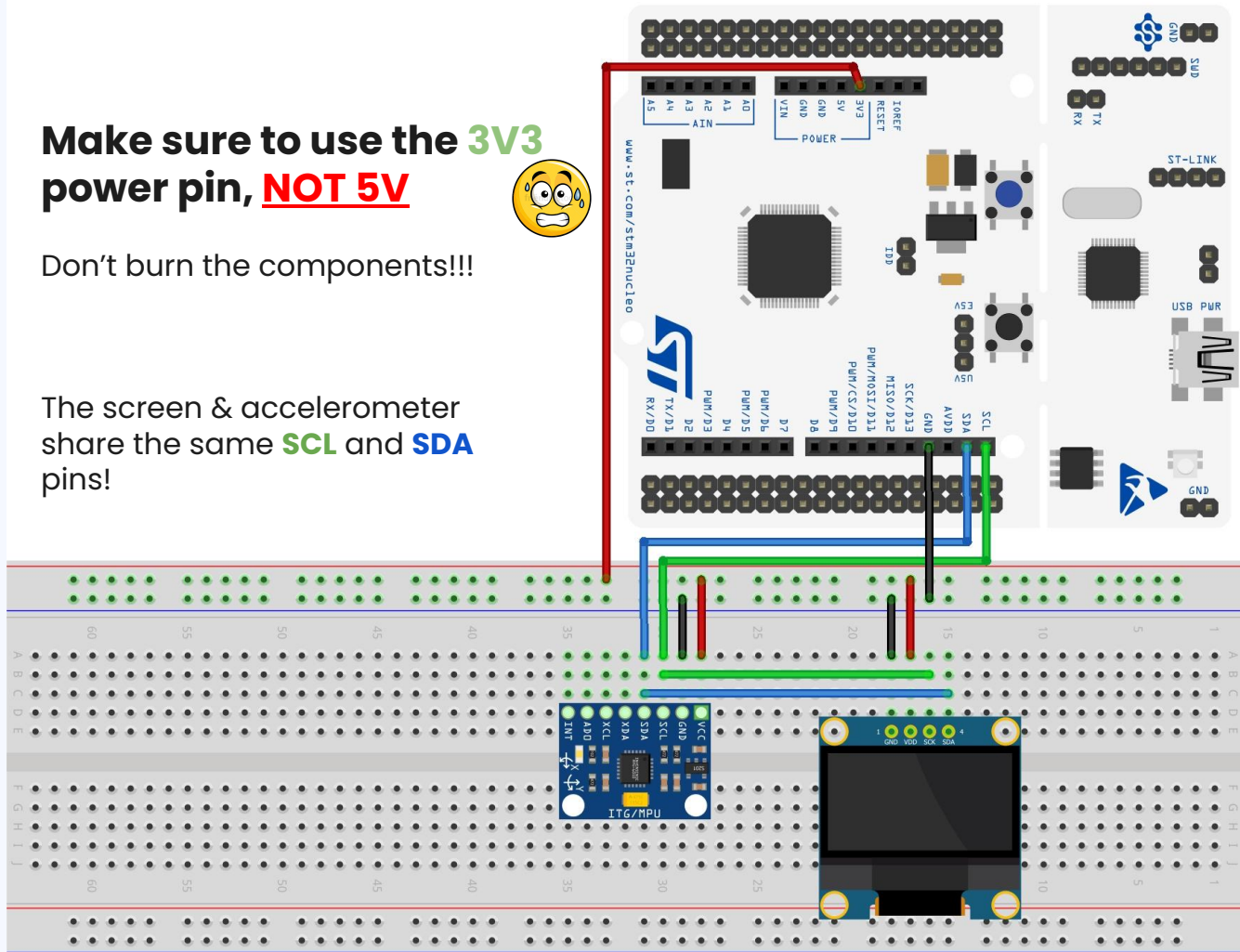
- I/O configurator will generate setup code for us (YIPPEE )
- We only need to tell the configurator what pins we will be using
- The pins labeled **SCL** and **SDA** on your boards are pins **PB8** and **PB9** respectively

Make sure to use the **3V3** power pin, **NOT 5V**



Don't burn the components!!!

The screen & accelerometer share the same **SCL** and **SDA** pins!

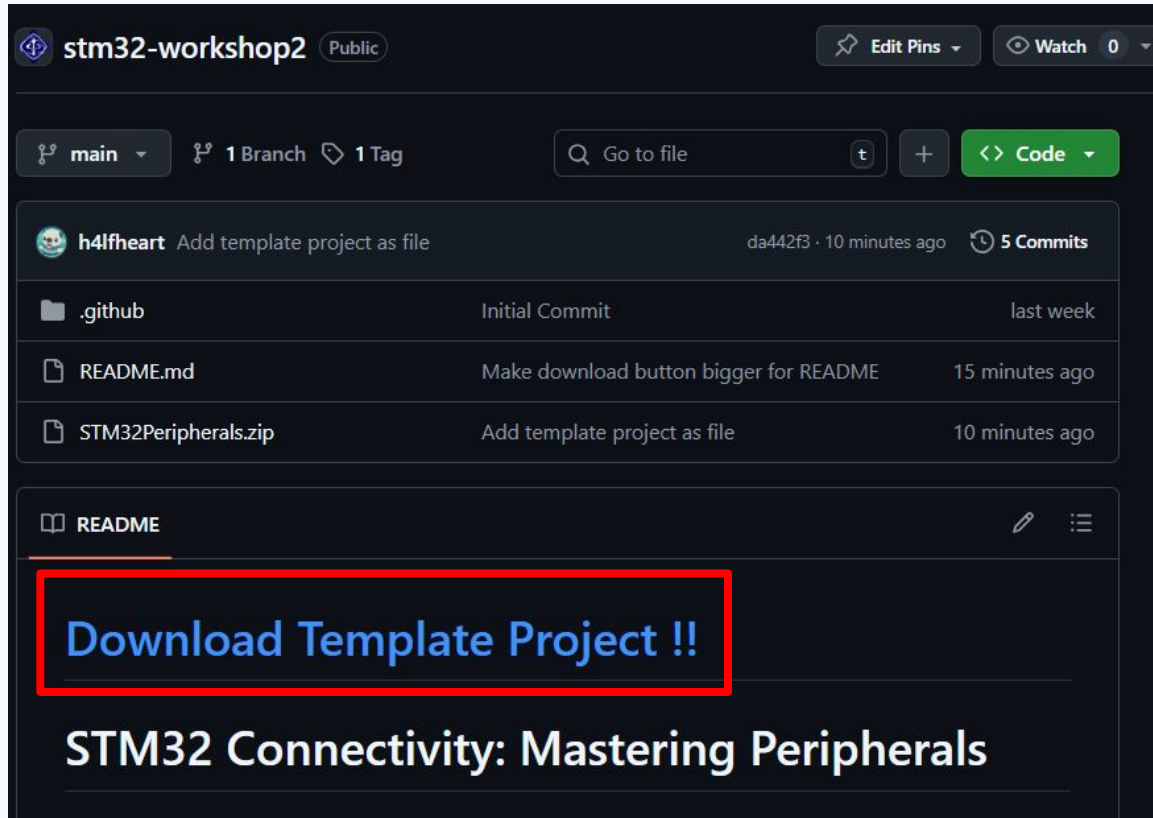


IEEE GitHub Link:

<https://github.com/psuieee/stm32-workshop2>



Downloading the Template Project



The screenshot shows the GitHub interface for the repository 'stm32-workshop2'. At the top, the repository name is displayed with a 'Public' badge. To the right are buttons for 'Edit Pins' and 'Watch' (with a count of 0). Below this, navigation links for 'main' (selected), '1 Branch', and '1 Tag' are shown, along with a search bar 'Go to file' and a green 'Code' button. The commit history table lists three commits: a folder '.github' (Initial Commit, last week), 'README.md' (Make download button bigger for README, 15 minutes ago), and 'STM32Peripherals.zip' (Add template project as file, 10 minutes ago). Below the table, the 'README' tab is active, showing a red-bordered box with the text 'Download Template Project !!' and the title 'STM32 Connectivity: Mastering Peripherals'.

stm32-workshop2 Public

Edit Pins Watch 0

main 1 Branch 1 Tag

Go to file + <> Code

h4lfheart Add template project as file da442f3 · 10 minutes ago 5 Commits

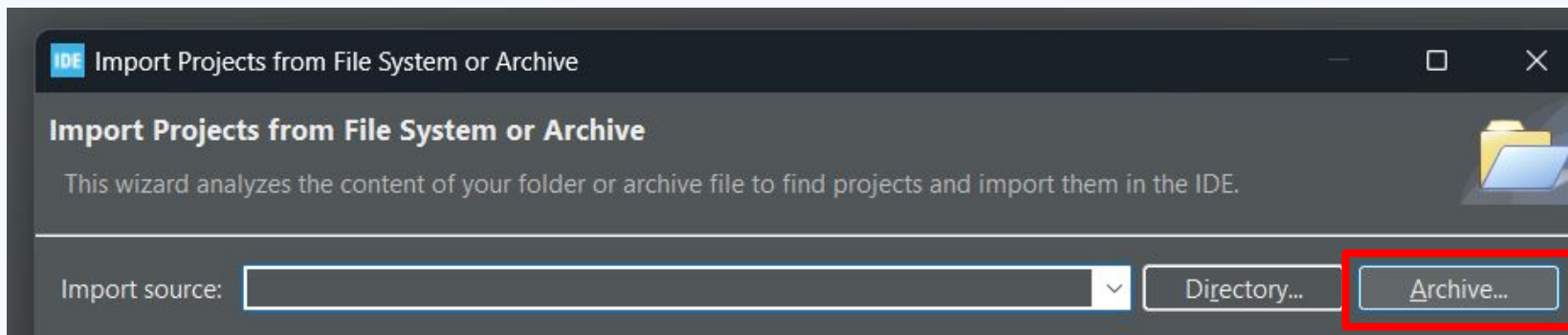
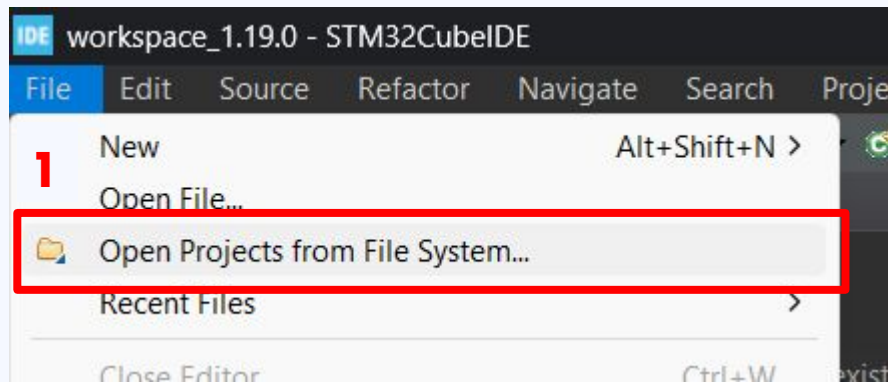
.github	Initial Commit	last week
README.md	Make download button bigger for README	15 minutes ago
STM32Peripherals.zip	Add template project as file	10 minutes ago

README

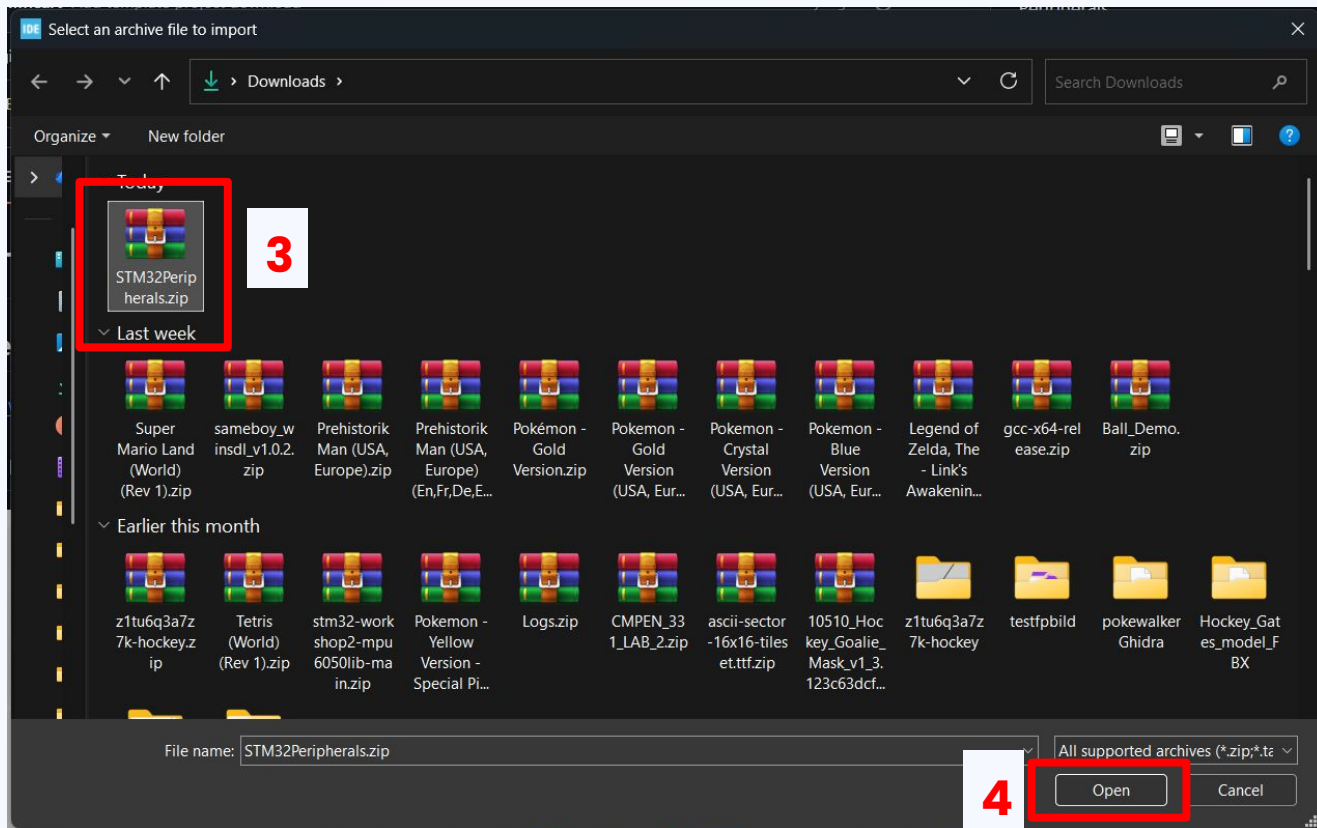
Download Template Project !!

STM32 Connectivity: Mastering Peripherals

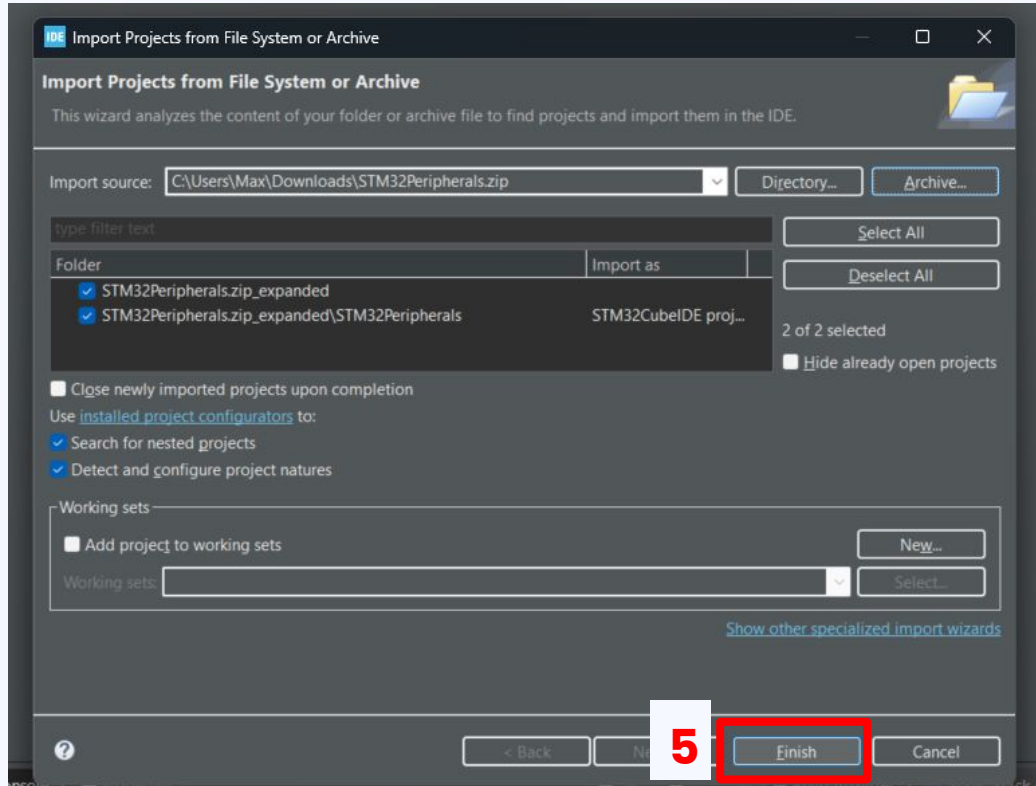
Importing the Template Project



Importing the Template Project



Importing the Template Project





I2C on the STM32 - HAL Code

- The STM32CubeIDE automatically imports the I2C HAL for us
- Cube IDE will also write all of the setup code for us
- `I2C_HandleTypeDef hi2c1;`
 - hi2c1 is a **handler**
 - It's a data structure that stores crucial information about our I2C hardware setup
 - Since C has no classes (unlike python or java), we use these handlers to store what an object would
 - Each call to the I2C HAL must contain this handler to specify what I2C bus we're using

Addressing the Accelerometer



MPU6050

- The MPU6050 had the predefined I2C address of hex 0x68
- The MPU6050 features a **WHO_AM_I** register which contains 0x70
- This register can be used as a quick way to tell if communication is working.
- We need to tell the I2C HAL we want to communicate with a device at that address
- Let's read the **WHO_AM_I** register!

Code

```
104  /* USER CODE BEGIN WHILE */
105
106  /* Initialize all configured peripherals */
107  MX_GPIO_Init();
108  MX_USART2_UART_Init();
109  MX_I2C1_Init();
110  /* USER CODE BEGIN 2 */
111
112  uint8_t whoami;           //variable to store what we read
113
114  HAL_I2C_Mem_Read(&hi2c1,    //i2c handler (created by cubemx)
115                  (0x68 << 1), //MPU6050 i2c address. HAL expects this address bit-shifted left by 1
116                  0x75,        //WHO_AM_I register address
117                  1,           //address size (1 byte for most cases)
118                  &whoami,     //pointer to where we're storing this value
119                  1,           //how many bytes to read from register address (used for grabbing lots of
120                  1000);       //timeout in ms
121
122  printf("Got 0x%x from the accelerometer!\n", whoami);
123  /* USER CODE END 2 */
124
125  /* Infinite loop */
126  /* USER CODE BEGIN WHILE */
127  while (1)
128  {
129      /* USER CODE END WHILE */
```

PROGRAMMING IN C



ON PC



ON A MICROCONTROLLER



Part 3

Drawing to the OLED Screen

SSD1306

- The SSD1306 has the predefined I2C address of hex 0x78, allowing us to use the same I2C port as the accelerometer
- The SSD1306 does not feature a **WHO_AM_I** register, but we can use the HAL **HAL_I2C_IsDeviceReady** function to check if an I2C device is running at the address 0x78
- Let's check if the device is running!

```
/* USER CODE BEGIN 2 */
HAL_StatusTypeDef status = HAL_I2C_IsDeviceReady(&hi2c1, (0x3C << 1), 1, 200);

printf("Are we working? %s\n", status == HAL_OK ? "yes" : "no");
/* USER CODE END 2 */
```

SSD1306 Initialization

- Now that we know the screen is running, let's initialize it and draw some text to the screen!
- Most peripheral libraries will require you to initialize the library with a reference to the I2C handler we set up before to send setup commands and data
- This will also handle checking if the device is running like we just did before with

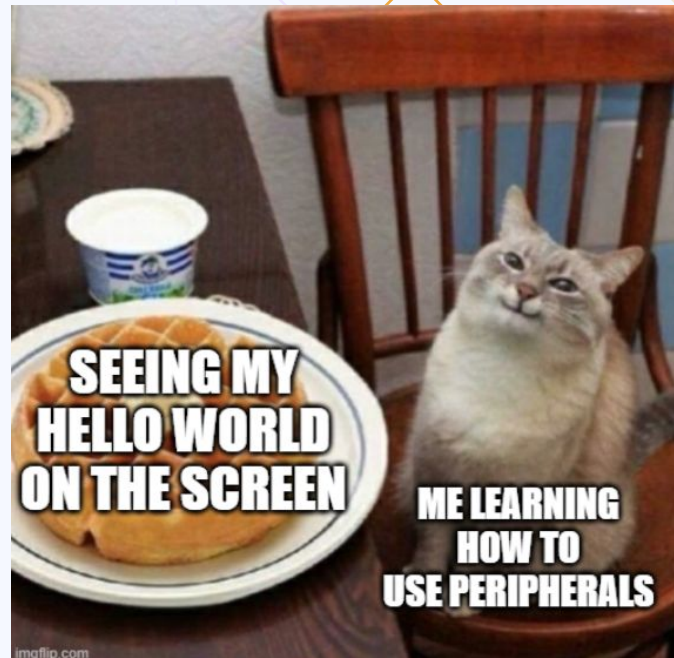
HAL_I2C_IsDeviceReady

```
/* USER CODE BEGIN 2 */  
ssd1306_t screen = {};  
ssd1306_init(&screen, &hi2c1);  
/* USER CODE END 2 */
```

Hello, World!

- From here, we need to draw our text with the `ssdl306_draw_text` function
- Most screen libraries use a framebuffer in memory to store pixels before pushing them to the screen, so we need to call the update function to push this framebuffer

```
/* USER CODE BEGIN 2 */  
ssdl306_t screen = {};  
ssdl306_init(&screen, &hi2c1);  
  
ssdl306_draw_text(&screen, &font_8x8, "Hello, World!", 0, 0);  
ssdl306_update(&screen);  
/* USER CODE END 2 */
```





- [illegible]

Drawing Images

- The flow for drawing an image to the screen is basically the same setup as drawing text, except we use **ssd1306_draw_bitmap**
- This function allows use to specify an image (bitmap), the x & y coordinates, and the size of the image we are drawing
- Let's test this with an image of the IEEE logo!!
 - It has been included in your project with the name **ieee_bitmap**

```
/* USER CODE BEGIN 2 */
ssd1306_t screen = {};
ssd1306_init(&screen, &hi2c1);

ssd1306_draw_bitmap(&screen, ieee_bitmap, 0, 0, 128, 64);
ssd1306_update(&screen);
/* USER CODE END 2 */
```



Part 4

Reading the Accelerometer

MPU6050 Initialization

- Just like the screen, we need to initialize the MPU6050 library
- For this library, we need to specify some extra parameters like the accelerometer range and gyroscope range
- Unlike the screen, this library does not require a mpu6050 object to be made, it handles all state statically within the library itself

```
/* USER CODE BEGIN 2 */
ssd1306_t screen = {};
ssd1306_init(&screen, &hi2c1);

mpu6050_init(&hi2c1, MPU_ACCEL_RANGE_2G, MPU_GYRO_RANGE_250DPS);
/* USER CODE END 2 */
```

(Make sure to keep your screen init, we will be using it as well)

Reading the Accelerometer

- To get the current values of the accelerometer, we can use the **mpu6050_getAccelData** function
- This function requires a pointer to a **mpu6050_3DData** object to be passed in that will receive the accelerometer values

```
mpu6050_3DData accel_data;  
mpu6050_getAccelData(&accel_data);
```

```
typedef struct {  
    double x;  
    double y;  
    double z;  
} mpu6050_3DData;
```

Reading the Accelerometer

- Now that we have the data, we can draw these values to the screen!

```
/* USER CODE BEGIN 2 */
ssd1306_t screen = {};
ssd1306_init(&screen, &hi2c1);

mpu6050_init(&hi2c1, MPU_ACCEL_RANGE_2G, MPU_GYRO_RANGE_250DPS);
/* USER CODE END 2 */

/* Infinite loop */
/* USER CODE BEGIN WHILE */
while (1)
{
    /* USER CODE END WHILE */

    /* USER CODE BEGIN 3 */
    mpu6050_3DData accel_data;
    mpu6050_getAccelData(&accel_data);

    ssd1306_fill(&screen, BLACK); // clear the screen to clean up old text

    // draw the accelerometer values
    ssd1306_draw_text_double(&screen, &font_8x8, "x:", accel_data.x, 0, 0);
    ssd1306_draw_text_double(&screen, &font_8x8, "y:", accel_data.y, 0, 12);
    ssd1306_draw_text_double(&screen, &font_8x8, "z:", accel_data.z, 0, 24);

    ssd1306_update(&screen); // remember to update the screen!

    HAL_Delay(10); // delay 10ms because we don't need values that fast
}
/* USER CODE END 3 */
```



Part 5

Putting it All Together

Bouncing Ball

- Now that you know how to draw to the screen and read the accelerometer, let's put it all together into a single demo!!
- In your project, there is a function called **accel_ball_demo** that uses the accelerometer tilt to drive the physics of a bouncing ball
 - Make sure the accelerometer is on a flat surface during the intro so the accelerometer can calibrate

```
/* USER CODE BEGIN 2 */  
ssd1306_t screen = {};  
ssd1306_init(&screen, &hi2c1);  
  
mpu6050_init(&hi2c1, MPU_ACCEL_RANGE_2G, MPU_GYRO_RANGE_250DPS);  
  
accel_ball_demo(&screen); // has it's own event loop!!  
/* USER CODE END 2 */
```



Thanks for coming!

- We hope you learned a lot about i2c and peripheral communication on STM32!
- Look forward to the next workshop, **FreeRTOS Basics** on **March 22nd (?)**
 - Learn about multi-threading and take full advantage of your STM32
- Be creative! See what you can make with these peripherals. Feel free to adapt the code in the template projects to make something unique.

