Lab 4: I2C peripheral

Department of Electrical and Computer Engineering

ECE342, Winter 2026

University of Toronto

Introduction

In this lab, you will learn how to connect a peripheral device using the Inter-Integrated Circuit (I^2C) bus. Your task is to interface with a small OLED display driven by the SSD1306 controller.

Inter-Integrated Circuit (I^2C) is a widely used communication protocol in embedded systems. It requires only two signals:

- **Serial Clock Line (SCL)** – synchronizes communication
- **Serial Data Line (SDA)** – carries the data

Multiple devices can share the same bus, each identified by a unique 7-bit address. This allows a single microcontroller to communicate with many peripherals efficiently. Within one I^2C transaction, you can read from or write to different registers on a device.

While you will explore the details of the protocol in lectures, this lab focuses on practical application. You will use the I^2C functions provided by the STM Nucleo library. These functions abstract away the low-level signaling, enabling you to send and receive data from the SSD1306 OLED controller without manually handling the bus protocol.

As you work, you may find it useful to consult the SSD1306 datasheet for details about the controller's registers and functionality. The datasheet is available online: SSD1306 OLED Controller Datasheet.

By the end of this lab, you will:

- Understand how the I^2C bus enables communication between microcontrollers and peripherals
- Use STM Nucleo library functions to send and receive data over I^2C
- Interface with the SSD1306 OLED controller to display information

Background

This section provides the essential background you need to work with the OLED display. There is a lot of detail here, so it is **strongly recommended** that you read it carefully—ideally more than once—before writing any code. As you progress through the lab, refer back to this section often.

Display

The OLED screen used in this lab is monochrome, meaning it supports only a single color. It has a resolution of 128×64 pixels: 128 pixels wide and 64 pixels tall. Each pixel can be set to either black or white.

Note: Depending on the specific screen, "white" may appear as yellow, blue, or another color chosen by the manufacturer.

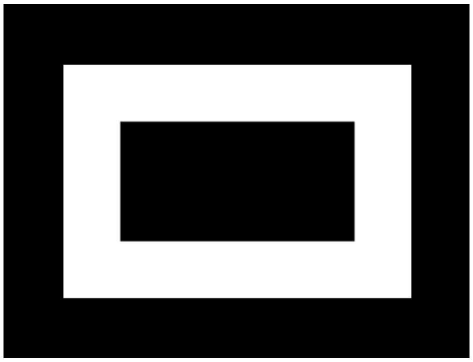
As with all displays, each pixel is addressed by its **row** and **column** position. For this monochrome display, a pixel value of '0' represents black and '1' represents white. For example, Figure 1a shows a simple white rectangle, while Figure 1b illustrates the pixel values required to draw it. With such a display, you can "draw" shapes by setting the appropriate pixels to '1'.

Packing Pixels into Bytes

The screen stores its $128 \times 64 = 8{,}192$ pixel values in a block of GDDRAM memory. This memory is *byte-addressable*, meaning it consists of 1,024 memory locations, each 8 bits wide.

Importantly, the SSD1306 organizes memory into **pages**, where each byte represents a vertical column of 8 pixels. Thus, to set a pixel at a specific row and column, you must calculate both the page number and the bit position within that byte. To turn on the pixel at **row 10, column 5**:

- Write to **page 1, column 5**.



Columns / Rows

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
2	0	0	1	1	1	1	1	1	1	1	1	1	1	1	0	0
3	0	0	1	1	1	1	1	1	1	1	1	1	1	1	0	0
4	0	0	1	1	0	0	0	0	0	0	0	0	1	1	0	0
5	0	0	1	1	0	0	0	0	0	0	0	0	1	1	0	0
6	0	0	1	1	0	0	0	0	0	0	0	0	1	1	0	0
7	0	0	1	1	0	0	0	0	0	0	0	0	1	1	0	0
8	0	0	1	1	1	1	1	1	1	1	1	1	1	1	0	0
9	0	0	1	1	1	1	1	1	1	1	1	1	1	1	0	0
10	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
11	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

Memory

0	1
0x00	0x00
0x00	0x00
0x3F	0xFC
0x3F	0xFC
0x30	0x0C
0x30	0x0C
0x30	0x0C
0x30	0x0C
0x3F	0xFC
0x3F	0xFC
0x00	0x00
0x00	0x00

(a) (b) (c)

Figure 1. (a) A simple white rectangle, (b) the corresponding pixel values for the first 12 rows and 16 columns, and (c) how these pixels are stored in 8-bit memory locations.

- This corresponds to **byte 133** in GDDRAM.
- Set **bit 2** of that byte to 1.

In C:

```
gddram[1][5] |= (1 << 2);
```

Row values		Bit values															
In Hex	In Binary	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0x0000	0000000000000000	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0x0E00	0000111000000000	0	0	0	0	1	1	1	0	0	0	0	0	0	0	0	0
0x0E00	0000111000000000	0	0	0	0	1	1	1	0	0	0	0	0	0	0	0	0
0x1B00	0001101100000000	0	0	0	1	1	0	1	1	0	0	0	0	0	0	0	0
0x1B00	0001101100000000	0	0	0	1	1	0	1	1	0	0	0	0	0	0	0	0
0x1B00	0001101100000000	0	0	0	1	1	0	1	1	0	0	0	0	0	0	0	0
0x1B00	0001101100000000	0	0	0	1	1	0	1	1	0	0	0	0	0	0	0	0
0x3180	0011000110000000	0	0	1	1	0	0	0	1	1	0	0	0	0	0	0	0
0x3180	0011000110000000	0	0	1	1	0	0	0	1	1	0	0	0	0	0	0	0
0x3F80	0011111110000000	0	0	1	1	1	1	1	1	1	0	0	0	0	0	0	0
0x3F80	0011111110000000	0	0	1	1	1	1	1	1	1	0	0	0	0	0	0	0
0x3180	0011000110000000	0	0	1	1	0	0	0	1	1	0	0	0	0	0	0	0
0x60C0	0110000011000000	0	1	1	0	0	0	0	0	1	1	0	0	0	0	0	0
0x60C0	0110000011000000	0	1	1	0	0	0	0	0	1	1	0	0	0	0	0	0
0x60C0	0110000011000000	0	1	1	0	0	0	0	0	1	1	0	0	0	0	0	0
0x0000	0000000000000000	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0x0000	0000000000000000	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0x0000	0000000000000000	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

Figure 2. Example of how each character is stored as a bitmap.

Displaying Text

A major part of this lab is writing text to the display. Characters, numbers, and symbols are formed by setting specific pixels to '1' or '0'. Conceptually, this is similar to the seven-segment displays you may have used in earlier courses.

Figure 2 shows the bitmap for the letter 'A'. Columns are labeled from 15 down to 0, and the red cells indicate pixels set to '1'.

By storing such bitmaps, we can represent each printable character. Concatenating the bits for each row allows us to store the character in binary or hexadecimal form. Collections of these bitmaps are called *fonts*.

Provided Fonts: Two fonts are included in the provided `fonts.c` file: `Font11x18` and `Font7x10`[1]. You will primarily use `Font11x18`, which defines characters that are 11 pixels wide and 18 pixels tall. Both fonts store values as `uint16_t` arrays.

Each font is stored as a large 1D array, with each entry corresponding to a single printable ASCII character. Although the array is split across multiple lines in `fonts.c` for readability, it is logically one continuous row of values. For example, the letter 'A' in Figure 2 comes from the `Font11x18` array (Line 153 in `fonts.c`). Each of the 18 values corresponds to one row of the bitmap, with each 16-bit integer encoding the on/off state of pixels in that row. Printing a character requires correctly indexing into this array.

Understanding ASCII: The font arrays are organized in ASCII order. ASCII is the standard encoding for text in digital systems. For example, `char c = 'A';` is equivalent to `char c = 65;`, since the ASCII value of 'A' is 65. A full list of ASCII values can be found at this website.

The first 32 ASCII values are non-printable (e.g., Null, Backspace, Escape). The arrays in `fonts.c` begin at value 32, the first printable character. To access the bitmap for a character, subtract 32 from its ASCII value. For example, 'A' has ASCII value 65, so its bitmap is stored at index $65 - 32 = 33$ in the `Font11x18` array.

Strings: Strings in C/C++ are stored as arrays of ASCII characters terminated by a special `'\0'` (NULL) character. For example:

```
char str[] = "ECE";
```

is equivalent to:

```
char str[] = {'E', 'C', 'E', '\0'};
```

The `'\0'` marks the end of the string. Although it is simply the value 0, using `'\0'` makes the intent clearer in code.

Key Takeaways

- The OLED display has a resolution of 128×64 pixels, each set to black (0) or white (1).
- The SSD1306 controller stores pixels in GDDRAM, organized into pages of 8 vertical pixels per byte.
- Fonts are stored as bitmaps, with each row of a character encoded as a 16-bit integer in the font array.
- ASCII values map directly to font indices: subtract 32 from the ASCII code to find the correct entry.
- Strings in C are null-terminated, meaning they end with the special `'\0'` character.

Hardware Setup

1. Using jumper cables, connect the SSD1306 OLED screen to the Nucleo board as shown in Table 1.
2. Download the provided kit, unzip it, and navigate to `Core\Src`.
3. Verify that the expected files are present:
   - Familiar: `main.c`, `config.c`, `stm32f4xx_hal_msp.c`, `stm32f4xx_it.c`, `system_stm32f4xx.c`
   - **New:** `ssd1306.c`, `fonts.c`

Table 1. SSD1306 OLED to Nucleo Connections

OLED Pin	Nucleo Pin	Header
GND	GND	CN8
VDD	3V3	CN8
SDA	PF0	CN9
SCL	PF1	CN9

[1] `Font7x10` appears quite small on the 128×64 display, but is available for use in your project if desired.

Checking Your Connections

Before writing any code, verify that your module is properly connected. Open `main.c`. After peripheral initialization, the function HAL_I2C_IsDeviceReady() checks communication with the OLED.
- If successful (HAL_OK), the Green LD1 LED turns on.
- If unsuccessful (HAL_ERROR), the Red LD3 LED turns on.

Note: Double-check wiring before debugging code. Incorrect connections are the most common source of errors.

Understanding the Provided Code

In this lab, you will work with two key files specific to the SSD1306 OLED driver. Begin by opening `Core\Inc\ssd1306.h`.

File: `ssd1306.h`

This header file contains important definitions and prototypes:

- Several `#define` parameters specify the I^2C address of the screen and its resolution in pixels.
- Additional `#define` statements identify registers used to control scrolling. These will be used later in the lab.
- A convenient `typedef` is provided for specifying pixel colors.
- The remainder of the file contains function prototypes for routines implemented in `ssd1306.c`.

File: `ssd1306.c`

This source file contains most of the code you will modify. At the top, three global arrays are declared:

1. `SSD1306_Buffer`: stores pixel values to be displayed on the screen.
2. `SSD1306_Init_Config`: lists the registers required to initialize the OLED.
3. `SSD1306_Scroll_Commands`: an empty array reserved for scrolling commands.

Following these arrays, several helper functions are provided for you:

- `SSD1306_UpdateScreen` – refreshes the display with the current buffer contents.
- `SSD1306_Fill` – fills the screen with a specified color.
- `SSD1306_Clear` – clears the display.

These functions are ready to use without modification. Guidance on when and how to apply them will be provided in later sections of this document.

Implementing New Functionality

This section describes the functionality you must implement for this lab. For clarity, it is divided into several parts. You are **strongly encouraged** to complete each part in order before moving on to the next one.

Part I: I2C Writes

In this part, you will implement basic write operations to the SSD1306 over I^2C .

ssd1306_I2C_Write:
This is the first (and most important) function you should implement. `ssd1306_I2C_Write()` requires four parameters:

1. **address**: The 7-bit address of the device.
2. **reg**: The register on the device to be accessed.

3. **data**: The value to be written to that register.
4. **count**: The number of bytes to be transmitted.

As discussed in lecture, I^2C writes occur in two logical stages:

1. Write the address of the device you want to communicate with.
2. Specify which register you want to write to, followed by the data bytes.

On the SSD1306 module, communication is controlled using a one-byte "control" field:

- `0x00` selects command bytes.
- `0x40` selects display data bytes.

On the Nucleo platform, I^2C writes are performed using the `HAL_I2C_Master_Transmit()` function, which accepts the following parameters:

1. **hi2c**: Pointer to an `I2C_HandleTypeDef` structure containing the configuration for the specified I2C peripheral.
2. **DevAddress**: The 7-bit address of the device.
3. **pData**: Pointer to the data buffer containing the values to be written.
4. **Size**: The number of bytes to send.
5. **Timeout**: A timeout value for the I^2C controller if there is no response from the target device.

Start by writing code that sends **one** byte of data to a single register on the device. Create a small 2-entry array, where the first element is the register (or control) byte and the second element is the data byte. You can then pass a pointer to this array to `HAL_I2C_Master_Transmit()`.

As with `HAL_I2C_IsDeviceReady()`, always check that `HAL_I2C_Master_Transmit()` returns `HAL_OK` to ensure that the transmission completed successfully. Once this function is working, you can move on to implementing `SSD1306_Init()`.

SSD1306_Init:
This function initializes the screen when the Nucleo board powers up. You will configure the SSD1306 by writing the initialization bytes stored in the `SSD1306_Init_Config` array to the display.

To begin, loop over `SSD1306_Init_Config` and write each entry to the command path (using a control byte of `0x00`) via your `ssd1306_I2C_Write()` function. Next, call `SSD1306_Init()` from `main()` to verify that the screen initializes correctly.

One simple test is to change the parameter passed to `SSD1306_Fill()` inside `SSD1306_Init()`. Set it to `SSD1306_COLOR_WHITE` and confirm that the display turns white.

Although black and white are represented internally by 0 and 1, the color is always specified using the enumerated type `SSD1306_COLOR_t`. By using `SSD1306_COLOR_WHITE` instead of just writing 1, your code more clearly communicates that you are setting a color value. This improves readability and reduces the chance of errors.

Part II: Drawing Pixels

The next step is to be able to set individual pixels on the screen.

One (inefficient) option would be to write each pixel directly to the GDDRAM on the OLED module, one byte at a time. However, this would be very slow. Instead, we can take advantage of the I^2C controller's ability to perform chained writes and transfer all 8,192 bytes in a single transmission.

To do this, we will:

- Store the desired pixel values in the `SSD1306_Buffer` array.
- Copy the entire buffer to the GDDRAM in one operation.

Note that `SSD1306_Buffer` is an array of `uint8_t`, and each pixel is represented by a single bit. Therefore, you must determine exactly **which** bit of **which** array element needs to be set for a given pixel. Refer back to the earlier section on packing pixels into bytes.

Based on that discussion, consider the following question: if you want to set **one** specific pixel to '1', how would you do this in code?

Complete the `SSD1306_SetPixel()` function so that it sets a specific pixel (given its `x` and `y` coordinates) to '1' when the color is white. Then, test your implementation by turning on a few individual pixels on the screen. Some example code is provided in `main()` to help you get started.

Keep in mind that simply modifying `SSD1306_Buffer` does *not* automatically update the display. After you finish writing pixels into `SSD1306_Buffer`, you must call `SSD_UpdateScreen()` to transmit the buffer contents to the screen over I^2C . Examine the provided `SSD_UpdateScreen()` function to understand how this transfer is performed.

Once you can reliably set individual pixels to '1', try drawing simple straight lines by calling `SSD1306_SetPixel()` inside a loop to set a continuous range of pixels. Next, experiment with drawing basic shapes such as rectangles and triangles. These tests will also help you validate your code for the next part of the lab.

Part III: Scrolling

The SSD1306 controller allows you to scroll the display content either left or right. Scrolling is performed at the level of *pages*. The GDDRAM is divided into 8 pages, where each page corresponds to 8 rows of the display. Therefore, when enabling scrolling, you must specify the starting and ending pages; only the rows belonging to these pages will be scrolled, while the remaining rows remain unchanged.
Implement scrolling in the `SSD1306_Scroll()` function. The first parameter (`direction`) specifies the direction of scrolling. You must use the enumerated type `SSD1306_SCROLL_DIRECTION` for this, as defined in `ssd1306.h`. Similar to the use of `SSD1306_COLOR_t` earlier, using this enumerated type ensures that your code explicitly selects a scrolling direction and improves readability.

Table 2. Bytes to write for scrolling

Byte #	Explanation	Value
1	Scrolling direction	0x26 for right 0x27 for left
2	Dummy byte	0
3	First page to scroll	start_row
4	Scrolling frequency	0
5	Last page to scroll	end_row
6	Dummy byte	0
7	Dummy byte	0xFF
8	Activate scrolling	0x2F

Here, `start_row` and `end_row` are page indices (each page represents 8 pixel rows), as defined in `ssd1306.h`.

To enable scrolling, you must write the eight bytes shown in Table 2 to the SSD1306 over I^2C .

You may use the `SSD1306_Scroll_Commands` array for this; simply assign the appropriate values inside `SSD1306_Scroll()`. All required constants are defined in `ssd1306.h`, and you should use these `#define` values rather than hard-coding numeric literals.

Once the array is prepared, call `ssd1306_I2C_Write()` to transmit the bytes to the display. You should now observe the display content scrolling left or right depending on the chosen direction.

Scrolling frequency (optional): Byte #4 in Table 2 controls the scrolling speed. The SSD1306 expresses this speed in terms of 'frames'. If you wish to experiment with different speeds, use the byte values listed in Table 3.

Table 3. Values to control the speed of scrolling.

Frequency (frames)	2	3	4	5	25	64	128	256
Byte #4 value	7	4	5	0	6	1	2	3

Part IV: Writing Text

The next step is to display text on the screen. Begin by writing code to display a single character.

Writing a single character: Start by completing the `SSD1306_Putc()` function. Based on the input character (`ch`), you must access the correct bitmap data from the corresponding `font` array. Recall that characters are passed in as ASCII values.

Each character in the `Font11x18` font is stored as an array of `uint16_t` values. For each character:

- iterate over all 18 rows of bitmap data;

- for each row, iterate over the top 11 bits (the remaining 5 bits are unused);
- for every bit that is set, use `SSD1306_SetPixel()` to update the corresponding location in `SSD1306_Buffer`.

The `x` and `y` parameters indicate the starting (x, y) coordinate at which the character should be drawn.

NOTE: Each element of `Font11x18` is a `uint16_t`. Only the most significant 11 bits represent valid pixel data. The unused bits are depicted in grey in Figure 2.

After implementing `SSD1306_Putc()`, test your code by calling it in `main()` with different `x` and `y` coordinates.

Writing strings: Once single-character output works, extend your implementation to support writing strings in the `SSD1306_Puts()` function. This function accepts a null-terminated character array.

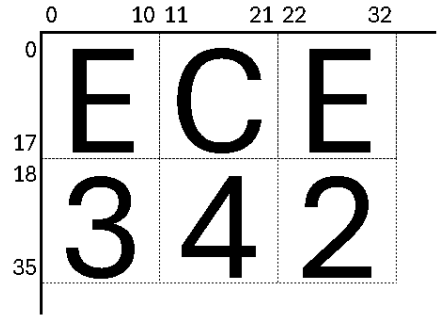


Figure 3. Pixel positions for writing text.

Loop through each character in the string until the 'NULL' terminator is reached. For each character:

1. call `SSD1306_Putc();`
2. update `SSD1306_CurrentX` so that the next character is drawn immediately to the right of the current one.

Figure 3 illustrates the pixel layout when printing the string `"ECE342\n342"` using `SSD1306_Puts()`. Use this figure to confirm that your output appears at the correct positions on the screen.

Support newlines: Before printing each character, check whether it is the newline character (`\n`). If so, increment `SSD1306_CurrentY` to move to the next text line, and reset `SSD1306_CurrentX` to the starting horizontal position.

1. In-lab Demo

During your lab session, to receive full marks, you must show your TA **3 lines** of text scrolling right for 3 seconds and then left for 3 seconds.

If you do not have all the parts working, you can demo the parts you have working for partial marks. If you cannot show text, draw any shape and demo scrolling with that. If you cannot show text or scrolling, just draw any shape.

Your TA may ask questions about your solution. Be prepared to explain your design decisions and your code.