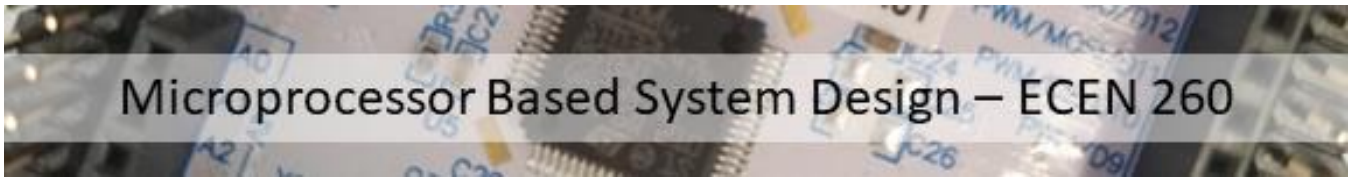


ECEN 260 Lab 11
Displays

Lab Objectives

- Gain more insight and practice using peripherals
- Learn how to display characters on a graphics LCD (GLCD) and character LCD
- Learn how to use the SPI and I^2C communication protocols.
- Prepare for the final project.

Required Parts

- Nucleo-L476RG board and USB cable
- PCD8544 GLCD screen
- 1602 LCD with I^2C daughterboard
- several jumper wires

Background: SPI and I^2C

The UART communication protocol we used in the last lab is *asynchronous* and is therefore traditionally used for low-speed data transfer between microcontrollers that each have an internal clock. On the other hand, peripheral I/O devices, like screens, often don't have their own internal clock. With such devices, it is common to share a clock signal with them from the microcontroller via a *synchronous* communication link. SPI and I^2C are both synchronous serial protocols commonly used for communicating with peripheral I/O devices.

SPI (Serial Peripheral Interface): The SPI protocol, developed by Motorola, uses a master/slave configuration, in which the *main* device (the master) controls the communication channel between it and one or more peripheral *sub* devices (slaves).

Note: The "master / slave" terms are older terminology that is being slowly replaced with less offensive terms, such as "main / sub," "controller / peripheral," or "primary / secondary." But many existing technical documents and standards still use "master / slave" terminology, so you need be able to recognize it.

- The master provides the clock - Serial Clock (SCLK).
- The data line from the main to a sub is the Main Out Sub In (MOSI) line.
- The data line from a slave to the master is the Main In Sub Out line (MISO).
- Because there may be more than one peripheral device, there is a fourth line called Slave Select (SS) – also called Chip Select (CS) or Chip Enable (CE) – to each peripheral device to activate that device. This line typically employs *negative true logic,* meaning that the line defaults to a high voltage when not in use, and goes low when activated.

The figure on the right shows an example of a controller with 3 peripherals sharing the MOSI/MISO bus lines, with each peripheral having its own SS activation line.

When the controller wants to send a message to a peripheral, it lowers the SS line for that peripheral and send the data serially as high and low voltages on the MOSI line, synchronized with the SCLK. The controller may receive a message the same way by looking at the MISO line during transmission. Because the MISO line is separate from the MOSI line, SPI is *full duplex*.

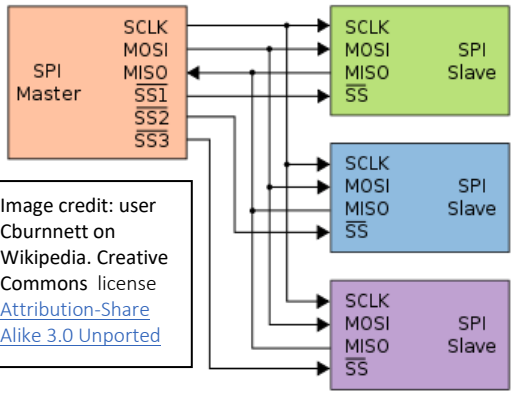


Image credit: user Cburnnett on Wikipedia. Creative Commons license Attribution-Share Alike 3.0 Unported

I²C (Inter-Integrated Circuit): Like SPI, I²C is synchronous and so there is a shared serial clock line (SCL). Unlike SPI, I²C is *half duplex* – there is a single wire for serial data (SDA) that can be used to send or receive data. The controller or a peripheral device take turns sending data on the SDA line. To avoid a floating voltage when neither device is transmitting, the SDA line is attached to a pull-up resistor, keeping the voltage high when not in use. Then, when a device wants to transmit, it pulls the voltage low for a logic '0' and releases it back to a high voltage for a logic '1.' The serial clock line (SCL) also uses a pull resistor to remain high when data is not being transmitted.

Instead of using a unique enable line for each peripheral like SPI, I²C has the transmitter indicate the intended recipient by sending an *address* with each message. Each device listens on the data line, and if the transmitted address is the same as its address, it knows to read the rest of the message.

I²C is advantageous when you have limited pins or want to reduce wire usage. SPI is advantageous for higher speed data transfer (like with an SD card) or when full duplex is required.

Background: SPI on the STM32-L476RG

The STM32-L476RG chip has three built-in SPI communication modules: SPI1, SPI2, and SPI3. These are the main pin options that can be connected to each:

	SCK	**MISO**	**MOSI**
SPI1 (Port A, alternate function 5)	PA5	PA6	PA7
SPI1 (Port B, alternate function 5)	PB3	PB4	PB5
SPI2 (Port B, alternate function 5)	PB13	PB14	PB15
SPI2 (Port B/C, alternate function 5)	PB10	PC2	PC3
SPI3 (Port B, alternate function 6)	PB3	PB4	PB5
SPI3 (Port C, alternate function 6)	PC10	PC11	PC12

Note that the SS/CS/CE pins are not part of the communication module, as they don't need to be clocked. We can make them regular GPIO output pins and reset/set them when needed. We will use the option highlighted above in yellow because it is mapped to the Arduino pins of the same purpose.

In the device configuration tool, the desired set of pins from the above options can be specified as SPI pins. For this lab, we will only be sending data one way (from the microcontroller to the screen), so we don't need the MISO line.

Background: I^2C on the STM32-L476RG

The STM32-L476RG chip also has three built-in I^2C communication modules: I2C1, I2C2, and I2C3. These are the main pin options that can be connected to each:

	SCL	**SDA**
I2C1 (Port B, alternate function 4)	PB6	PB7
I2C1 (Port B, alternate function 4)	PB8	PB9
I2C2 (Port B, alternate function 4)	PB10	PB11
I2C2 (Port B, alternate function 4)	PB13	PB14
I2C3 (Port C, alternate function 4)	PC0	PC1

While any of the above options are available, we will use the option highlighted above because it is mapped to the Arduino pins of the same purpose – labeled SCL and SDA on the Nucleo board.

Background: The PCD8544 GLCD Screen

The PCD8544 is a screen developed by Nokia and was used on cell phones during the early 2000s. It has a grid of 84 (columns) by 48 (rows) of black and white pixels. The 48 rows are grouped into 6 banks (8 rows per bank) so that 8 pixels may be written at once with a single byte of data (1 bit per pixel). Figures 3 and 4 in the PCD8544 datasheet show the pixel layout. Each column (X) and bank (Y) pairing has an X-Y coordinate. Writing a byte of data to that coordinate changes those eight pixels.

In addition to sending data to change pixel values, we can also send commands to the screen. Here are some of the commands we will be using:

- Change the column (X position) / change the bank (Y position)
- Adjust the contrast / operating voltage / bias voltage / temperature coefficient

We communicate with the PCD8544 using SPI. In addition to the SCLK, MOSI, and SS lines, the PCD8544 also has:

- a reset pin (RST) that must receive a low pulse to reset the screen on startup;
- a $D/\bar{C}$ pin (DC) that tells the screen if the byte of data received is a packet of "Data" (D) for updating pixels or if the byte of data received is a packet containing a "Command" (C) for configuring the screen (1 = Data, 0 = Command);
- a light or LED pin that needs a high voltage (3.3V) for the screen's backlight;
- a VCC pin that also needs a high voltage (3.3V); and
- a Ground pin that needs a low voltage.

Depending on the manufacturer (some which made misprints), the pins may have the following equivalent names:

- LIGHT = LED = BL (backlight) = DL (misprint)
- VCC = YCC (misprint)
- SS (slave select) = CE (chip enable) = CS (chip select) = SCE (slave/serial chip enable) = SCS (slave/serial chip select)
- CLK = SCLK = SCK (serial clock)
- MOSI (master out data in) = DIN (data in) = SDIN (slave/serial data in)
- DC = D/C' = DO (misprint)

Part A: Displaying on the PCD8544 via SPI

For this part of the lab, you will start with the provided scaffolding code that we created together in class. If you already have the text "ABC" showing on your screen from following the in-class "Code-with-me," you may skip parts A1, A2, and A3 and move directly to part A4.

Part A1: Configuring the SPI module and other screen communication pins

1. Create a new STM32 project for the NUCLEO-L476RG board called "Lab10_PartA" or something similar.
    a. Do not choose Empty.
    b. Select "Yes" for initializing peripherals to their default configuration.
2. In the device configuration tool "Pinout view":
    a. Change PC13 to "RESET_STATE" – we won't be using the button.
    b. Change PA2 and PA3 to "RESET_STATE" – we won't be using the UART.
    c. Change PA5 to "SPI1_SCK" – we will be using this pin for the serial clock on SPI1 instead of for onboard LED2. This pin is labeled "SCK/D13" on your board.
    d. Change PA7 to "SPI1_MOSI" – we will be using this pin for the Master Out Slave In data line on SPI1. This pin is labeled "PWM/MOSI/D11" on your board. As a reminder, we won't be using a MISO pin, because we are only transmitting *to* the display.
    e. Change PB6 to "GPIO_Output" – we will be using this pin as the Slave Select (SS), aka Chip Select (CS) or Chip Enable (CE) pin. This pin is labeled as "PWM/CS/D10" on your board.
    f. Change PA0 to "GPIO_Output" – we will be using this pin as the Data/Control pin.
    g. Change PA1 to "GPIO_Output" – we will be using this pin as the Reset pin.
3. In the device configuration tool under the "Categories" tab, expand the "Connectivity" option and select "SPI1."
4. In the "SPI1 Mode and Configuration" pane, change "Mode" to "Transmit Only Master."
5. Under "Configuration" in the "Parameter Settings" tab:
    a. Change "Data Size" to "8 bits."
    b. Change the "Prescaler" to 32 to reduce the Baud Rate to 2.5 MBits/s (the max clock speed supported by the screen is 4 MBits/s).
6. Save and generate code.

Part A2: Writing functions for communicating with the screen

1. In the "USER CODE BEGIN PM" section, add the following private macros (PMs) that define the GPIO pins and the GLCD dimensions:

```
#define CE_PORT     GPIOB        // PB6 chip enable (aka slave select)
#define CE_PIN      GPIO_PIN_6

#define DC_PORT     GPIOA        // PA0 data/control
#define DC_PIN      GPIO_PIN_0

#define RESET_PORT  GPIOA        // PA1 reset
#define RESET_PIN   GPIO_PIN_1

#define GLCD_WIDTH  84
#define GLCD_HEIGHT 48
#define NUM_BANKS   6
```

Note: Do not copy and paste any code for Part A2 or A3. Instead, you should type all the code yourself to practice the syntax and better understand what the code does.

2. In the "USER CODE BEGIN PFP" section, add the following private function prototypes (PFPs):

```
void SPI_write(unsigned char data);
void GLCD_data_write(unsigned char data);
void GLCD_command_write(unsigned char data);
void GLCD_init(void);
void GLCD_setCursor(unsigned char x, unsigned char y);
void GLCD_clear(void);
void GLCD_putchar(int font_table_row);
```

You will add code to define the above functions. Start with SPI_write. To send a byte of data to the screen using the SPI protocol , you must first assert the Chip Enable pin (low is asserted), then transmit the byte data using the HAL_SPI_Transmit function, then de-assert the Chip Enable pin (high is no longer asserted)

3. After main in the "USER CODE BEGIN 4" section, add the following function definition:

```
void SPI_write(unsigned char data){

	// Chip Enable (low is asserted)
	HAL_GPIO_WritePin(CE_PORT, CE_PIN, GPIO_PIN_RESET);

	// Send data over SPI1
	HAL_SPI_Transmit(&hspi1, (uint8_t*) &data, 1, HAL_MAX_DELAY);

	// Chip Disable
	HAL_GPIO_WritePin(CE_PORT, CE_PIN, GPIO_PIN_SET);
}
```

Next, define the GLCD_data_write and GLCD_command_write functions. Each of these functions sends data using the SPI_write function just defined, but with the DC pin configured appropriately.

If the DC pin is HIGH, the values sent will be interpreted by the screen as pixel "data." IF the DC pin is LOW, the values sent will be interpreted by the screen as a "command."

4. In the “USER CODE BEGIN 4” section, add the following function definitions:

```
void GLCD_data_write(unsigned char data){

	// Switch to "data" mode (D/C pin high)
	HAL_GPIO_WritePin(DC_PORT, DC_PIN, GPIO_PIN_SET);

	// Send data over SPI
	SPI_write(data);
}

void GLCD_command_write(unsigned char data){

	// Switch to "command" mode (D/C pin low)
	HAL_GPIO_WritePin(DC_PORT, DC_PIN, GPIO_PIN_RESET);

	// Send data over SPI
	SPI_write(data);
}
```

With the data and command write functions, you can now write the functions to initialize the screen, change the cursor coordinates, and clear the screen. The GLCD_init function set the CE pin to the default HIGH mode when not transmitting, resets the screen by sending a low pulse on the RESET pin, and then sends several commands (as described in the datasheet) for configuring the screen. The GLCD_setCursor function uses a command to set the X coordinate (column) and another command to set the y coordinate (bank), also as described in the datasheet. The GLCD_clear function loops through every bank and every column and writes a 0x00 to clear the pixels at each address, then returns the cursor to the top left of the screen.

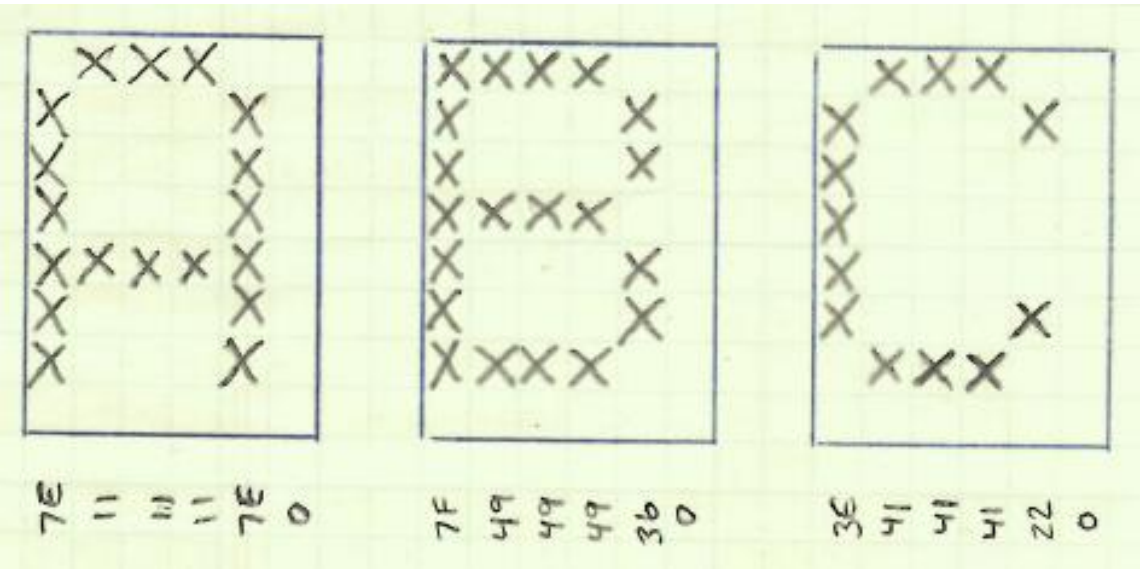
5. In the “USER CODE BEGIN 4” section, add the following three function definitions:

```
void GLCD_init(void){

	// Keep CE high when not transmitting
	HAL_GPIO_WritePin(CE_PORT, CE_PIN, GPIO_PIN_SET);

	// Reset the screen (low pulse - down & up)
	HAL_GPIO_WritePin(RESET_PORT, RESET_PIN, GPIO_PIN_RESET);
	HAL_GPIO_WritePin(RESET_PORT, RESET_PIN, GPIO_PIN_SET);

	// Configure the screen (according to the datasheet)
	GLCD_command_write(0x21); // enter extended command mode
	GLCD_command_write(0xB0); // set LCD Vop for contrast (this may be adjusted)
	GLCD_command_write(0x04); // set temp coefficient
	GLCD_command_write(0x15); // set LCD bias mode (this may be adjusted)
	GLCD_command_write(0x20); // return to normal command mode
	GLCD_command_write(0x0C); // set display mode normal
}

void GLCD_setCursor(unsigned char x, unsigned char y){
	GLCD_command_write(0x80 | x);    // column
	GLCD_command_write(0x40 | y);   // bank
}
```

```
void GLCD_clear(void){
	int i;
	for(i = 0; i < (GLCD_WIDTH * NUM_BANKS); i++){
		GLCD_data_write(0x00); // write zeros
	}
	GLCD_setCursor(0,0);	// return cursor to top left
}
```

Part A3: Creating and displaying font for "ABC"

Now you will add code for a font table. Each character will be 6 pixels wide and 8 pixels tall (1 bank tall), so each character's font representation will consist of 6 bytes (for six columns wide), with each of those bytes being the 8 bits that specify the pixel on/off values for the 8 rows in the bank.



The figure above should be read one column at a time. In a given column holding 1 byte, the most significant bit is the bottom pixel in that column and the least significant bit is the top pixel.

In the code, you will enter the font encoding using a two-dimensional char array. Each row of the array will hold the 6 bytes for the font symbol for one character. You are provided in this section with the font code for the space character, the letter 'A', the letter 'B', and the letter 'C'.

1. Before main, in the "USER CODE BEGIN PV" section, add the following private variable (PV):

```
const char font_table[][6] = {
                              {0x00, 0x00, 0x00, 0x00, 0x00, 0x00}, // space
                              {0x7E, 0x11, 0x11, 0x11, 0x7E, 0x00}, // 'A'
                              {0x7F, 0x49, 0x49, 0x49, 0x36, 0x00}, // 'B'
                              {0x3E, 0x41, 0x41, 0x41, 0x22, 0x00}, // 'C'
};
```

Now, we can write the code for the putchar function. This function will take the row of the font table as the input parameter, and then do a data write for each of the 6 bytes (columns) of that character, putting that character on the screen.

2. After main, in the “USER CODE BEGIN 4” section, add the following function definition:

```
void GLCD_putchar(int font_table_row){
    int i;
    for (i=0; i<6; i++){
        GLCD_data_write(font_table[font_table_row][i]);
    }

}
```

Next, you can add code into main that will initialize and clear the screen, and put ‘A’, ‘B’, and ‘C’, on the screen.

3. In main, before the while loop in “USER CODE BEGIN 2” add the following function calls:

```
GLCD_init();  // initialize the screen
GLCD_clear(); // clear the screen
GLCD_putchar(1); // 'A'
GLCD_putchar(2); // 'B'
GLCD_putchar(3); // 'C'
```

4. Build and run the code. You should see “ABC” on your screen.

Troubleshooting Contrast: Different screens require different contrast values. If you can’t see “ABC” on your screen you may need to adjust Vop and the bias mode in the GLCD_init function:

Code to change the Vop:

The previous code used 0xB8 for the Vop, shown again below.

```
GLCD_command_write(0xB0);  // set Vop
```

Try the following options: 0x90, 0xA0, 0xB0, 0xC0.

Code to change the Bias Mode:

The previous code used 0x14 or the bias mode, shown again below.

```
GLCD_command_write(0x15); // set bias mode
```

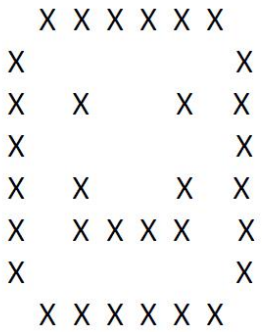
Try the following options: 0x10, 0x11, 0x12, 0x13, 0x14, 0x15, 0x16, 0x17.

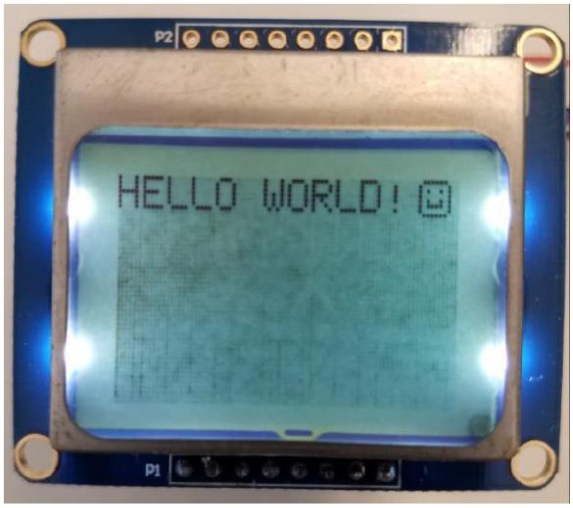
Try different combinations of the Vop and Bias Mode until you get a good contrast to see the characters on the screen. Some screens also require 5V instead of 3.3V. Experiment with that, too.

Part A4: Completing the alphabet and displaying “HELLO WORLD! 😊”

1. To finish part A, complete the “sample font table” in the provided code to include all 26 letters, a space, and punctuation.
    a. You may use the Font Work Sheet in I-Learn to determine which hexadecimal numbers are required to create each character representation.
    b. You may also use this tool, created by Caleb Malcarne, a former T.A. for this course.

2. Add the required code to the main function that will display the message “HELLO WORLD!” (use putchar for the row for H, then the row for E, then the row for L, then the row for L again, etc.).
3. Add a smiley face to the end of your message. HINT: Print it as two separate character blocks merged together. Each block is 8 pixels tall and 6 pixels wide. Make the face 8 pixels tall and 8 pixels wide. See the example face below.





Part B: Displaying on the 1602 LCD via I^{2}C

This part of the lab exists simply to provide you with an example of how to use I^{2}C to display text on the 1602 LCD that comes in Arduino kits. You don’t need to write any code for this part. Simply follow the steps below. You will use different pins, so you can leave the PCD8544 screen from Part A connected while you do Part B.

(The steps for Part B are modified from this tutorial by EMDEDDEDTHERE to work on the Nucleo-L476RG.)

1. Create a new STM32 project called “Lab10_PartB” or something similar with the default options.
2. In the device configuration tool “Pinout view”:
    a. Change PC13 to “RESET_STATE” – we aren’t using Button1.
    b. Change PA5 to “RESET_STATE” – we aren’t using LED2.
    c. Change PA2 and PA3 to “RESET_STATE” – we aren’t using UART2.
    d. Change PB8 to I2C1_SCL – this is for the serial clock pin in the I2C1 module. (This pin is labeled “SCL/D15” on your board.)
    e. Change PB9 to I2C1_SDA – this is for the serial data pin in the I2C1 module. (This pin is labeled “SDA/D14” on your board.)
3. Under “Categories,” expand the “Connectivity” option and select “I2C1.”
4. In the “I2C1 Mode and Configuration” pane:
    a. Change the “I2C” option from “Disable” to “I2C.”
5. Save and generate the code.

For this section, you *may* copy and paste the code.

6. In the "USER CODE BEGIN PM" section, add the following private macros:

```
#define I2C_ADDR 0x27 // I2C address of the PCF8574
#define RS_BIT 0 // Register select bit
#define EN_BIT 2 // Enable bit
#define BL_BIT 3 // Backlight bit
#define D4_BIT 4 // Data 4 bit
#define D5_BIT 5 // Data 5 bit
#define D6_BIT 6 // Data 6 bit
#define D7_BIT 7 // Data 7 bit
#define LCD_ROWS 2 // Number of rows on the LCD
#define LCD_COLS 16 // Number of columns on the LCD
```

7. In the "USER CODE BEGIN PV" section, add the following private variable:

```
uint8_t backlight_state = 1;
```

8. In the "USER CODE BEGIN 0" section, add the following function definitions:

```
void lcd_write_nibble(uint8_t nibble, uint8_t rs) {
  uint8_t data = nibble << D4_BIT;
  data |= rs << RS_BIT;
  data |= backlight_state << BL_BIT; // Include backlight state in data
  data |= 1 << EN_BIT;
  HAL_I2C_Master_Transmit(&hi2c1, I2C_ADDR << 1, &data, 1, 100);
  HAL_Delay(1);
  data &= ~(1 << EN_BIT);
  HAL_I2C_Master_Transmit(&hi2c1, I2C_ADDR << 1, &data, 1, 100);
}

void lcd_send_cmd(uint8_t cmd) {
  uint8_t upper_nibble = cmd >> 4;
  uint8_t lower_nibble = cmd & 0x0F;
  lcd_write_nibble(upper_nibble, 0);
  lcd_write_nibble(lower_nibble, 0);
  if (cmd == 0x01 || cmd == 0x02) {
    HAL_Delay(2);
  }
}

void lcd_send_data(uint8_t data) {
  uint8_t upper_nibble = data >> 4;
  uint8_t lower_nibble = data & 0x0F;
  lcd_write_nibble(upper_nibble, 1);
  lcd_write_nibble(lower_nibble, 1);
}
```

```c
void lcd_init() {
  HAL_Delay(50);
  lcd_write_nibble(0x03, 0);
  HAL_Delay(5);
  lcd_write_nibble(0x03, 0);
  HAL_Delay(1);
  lcd_write_nibble(0x03, 0);
  HAL_Delay(1);
  lcd_write_nibble(0x02, 0);
  lcd_send_cmd(0x28);
  lcd_send_cmd(0x0C);
  lcd_send_cmd(0x06);
  lcd_send_cmd(0x01);
  HAL_Delay(2);
}

void lcd_write_string(char *str) {
  while (*str) {
    lcd_send_data(*str++);
  }
}

void lcd_set_cursor(uint8_t row, uint8_t column) {
    uint8_t address;
    switch (row) {
        case 0:
            address = 0x00;
            break;
        case 1:
            address = 0x40;
            break;
        default:
            address = 0x00;
    }
    address += column;
    lcd_send_cmd(0x80 | address);
}

void lcd_clear(void) {
   lcd_send_cmd(0x01);
    HAL_Delay(2);
}

void lcd_backlight(uint8_t state) {
  if (state) {
    backlight_state = 1;
  } else {
    backlight_state = 0;
  }
}
```

9. In the “USER CODE BEGIN 2” section, add the following code:

```
// I2C pull-up resistors
GPIOB->PUPDR |= 0b01 << (8*2);
GPIOB->PUPDR |= 0b01 << (9*2);

// Initialize the LCD
lcd_init();
lcd_backlight(1); // Turn on backlight

// Write a string to the LCD
lcd_write_string("Hello, world!");
```

10. Connect the screen: VCC is 5V. For the other pins connections, revisit step 2.
11. Run the code. You should see “Hello, world!” on your character LCD screen.

Note: The contrast for these LCD screens is adjusted with a potentiometer you turn with a flat-head screwdriver on the back of the screen.

Part C: Do something fun/useful

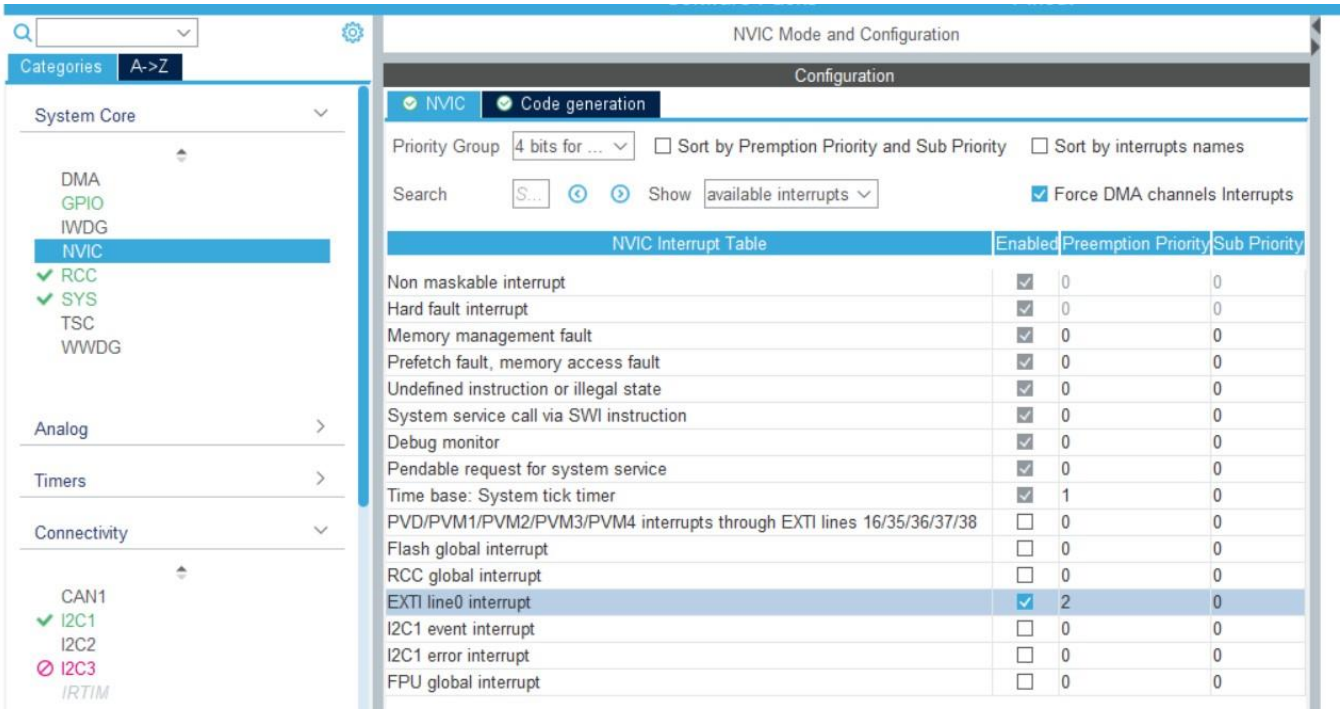
For this part of the lab, you may choose **one** of the following options. When choosing an option, consider your the final project. Which option would help you make progress in that direction?

1. Create an animation on your screen (either screen). To make an animation, just draw one thing, have a short delay, then draw the next thing, etc. To delay, use `HAL_Delay(ms)` where ms is the number of milliseconds to delay.

   or

2. Connect the keypad (from Lab 8) and display key presses onto the display. You may do this option with either the SPI display or the I2C display. If you do it with the I2C display, please see the note below on interrupt priorities. For help with setting up the interrupts, see the other note on the bottom of the next page.

Interrupt Priorities - Using the I2C display within an ISR: The function that transmits over I2C includes a delay with the SysTick Timer. If you want to transmit to the I2C display inside of the callback for an interrupt (such as for the keypad), you'll need to adjust the priorities between the timer and the external interrupt. Go to the ioc, under "System Core," select NVIC, then under "NVIC Mode and Configuration" change the "Preemption Priority" for both "Time base: System Tick Timer" and for "EXTI lineX interrupt" (where X is the pin number). The EXTI priority number needs to be a higher than the System Tick priority number. For example, you can change the System Tick priority to "1" and the EXTI priority to "2" (see screenshot on next page).



Using Interrupts with the HAL Drivers: Revisit the Interrupts lab for complete steps on how to set up interrupts with the keypad encoder. Here's a short summary:

- Configure the interrupt pin as GPIO_EXTI in the device configuration tool.
- Under "Categories" select "System Core" -> "GPIO."
- In the Configuration pane, under the "GPIO" tab, select the pin, which opens the option to select the edge(s) you want to detect. Choose the edge you are looking for.
- Switch to the "NVIC" tab and enable the interrupt for that line (same as the pin number).
- After you generate code, add this function, which is called when any GPIO interrupt occurs:

```
void HAL_GPIO_EXTI_Callback(uint16_t GPIO_Pin)
{
    if(GPIO_Pin == GPIO_PIN_X) // change 'X' to pin number
    {
       // Do something
    }
}
```

Part D: Extra Credit (+20 points)

For extra credit, do the *other* option from Part C. Include both the code and a video demonstration.

Demonstration of Correct Operation

When you are finished with the lab, upload the following to the "Lab 11 Demo & Code" assignment in I-Learn:

1. For Part A: a photo of "Hello World! 😊" on the PCD8544.
2. For Part A: a file with the code for your font table.
3. For Part B: a photo of the 1602 LCD with the text.
4. For Part C: a video showing an animation or the keypad/display working.
5. For Part C: a copy of your main.c file.
6. (extra credit) A video for Part D.
7. (extra credit) The main.c file for Part D.

Lab Report Instructions:

Once again, your lab report should be created using LaTeX. Adapt the template from Lab 10 as needed. Upload the generated lab report to the "Lab 11 Report" assignment in I-Learn. Your schematic should have both displays. Your test plan should be for Part C. If you chose the keypad/display option, create a test plan to test each key press. If you chose the animation option, write your test plan to indicate what animation should appear when the system it is powered on, including images showing what the animation should look like.

Don't forget that I love you! :)