Lab 2: Device Drivers and MMIO

You cannot wait for inspiration. You have to go after it with a club.

04-633 Embedded Systems Development

Code Due: Monday, February 12, 2024

Demos Due: Monday, February 19, 2024

Contents

1 Introduction and Overview

1.1 Goal

The goal of this lab is to use Memory Mapped IO (MMIO) to interface with various peripherals on an embedded device. You will be responsible for implementing the drivers for UART and I2C serial communication as well as an LCD. You will breadboard circuit components which will interface with your STM32 board to add functionality!

Make sure to read through the entire lab handout before starting. It is highly recommended to take notes as you do so as to not miss important implementation details.

1.2 Task List

1. Clone the starter code from GitHub classroom (see Section 3)
2. Get the components required for the lab. You should have:
   (a) a breadboard
   (b) an I2C extender board
   (c) wires
   (d) resistors
   (e) Other parts (buttons, LEDs, header pins, keypad, and LCD screen)
3. Understand GPIO (see Section 4)
4. Implement UART (see Section 5)
5. Implement Button Inputs (see Section 7)
6. Implement LED Outputs (see Section 6)
7. Implement Keypad Input (see Section 8)
8. Checkpoint (see Section 9)
9. Implement I2C (see Section 10)
10. Implement the LCD Driver (see Section 11)
11. Put it all together (see Section 12)
12. Submit the link to your repo to Canvas

1.3 Grading

Start this lab early to give yourself ample time to debug. All code submitted code must be compiled and executed properly to receive full credit. Portions of this lab will also be critical components for future labs. A significant portion of the lab is devoted to style, documentation, and following proper submission protocol.

Task	Points
uart_polling.c	20
i2c.c	20
lcd-driver.c	15
keypad-driver.c	15
main.c	10
Style and Submission Protocol (see Sections 9 and 14)	10
Checkpoint	10
TOTAL	100 pts

2 Teams

Lab 2 is to be accomplished in teams of 2.

The goal, of course, is to share the work and learn from each other. As such, you are expected to work together: communicate well, discuss your design, share responsibilities throughout the project. If you are having teammate issues, please talk to the course staff right away so we may help you to solve them. Don't wait until the end, at which time the staff is out of options and won't be able to help much.

Grace days used will be charged to both partners of the team.

As a team, you should be using Git as a collaborative tool. Your GitHub Classroom repository is a team repository. The first one to follow the invitation link should make a team repo. Name your team repo with both Andrew IDs. The second partner needs to make sure to join the correct team.

Then, communicate with your partner about workflow. How do you intend to collaborate? Will you branch whenever you are doing work on a feature? Will you only merge when you have a working feature? Figure this out ahead of time with your partner. As with any team collaboration, communication is key. Talk to your teammate about the organization. Talk to your teammate about how you will communicate (email, slack, telegraph, . . .).

The course and university policies about academic integrity apply to this project, even though it is a team project. All work must be that of your team. Make sure your teammate is properly responsible for all their code, as you don't want to be charged with an AIV for their conduct.

Hints from others can be of great help, both to the hinter and the hintee. Thus, discussions and hints about the assignment are encouraged. However, the project must be coded and written up as a team (you may not show, nor view, any source code from other teams). We may use automated tools to detect copying.

Use Piazza to ask questions or come visit us during office hours (or email for an appointment at another time). We want to see you succeed, but you must ask for help.

3 Starter Code

Use the assignment link on Canvas to create your repo for Lab 2. Pay attention to the teammate's instructions about the repo from the previous section. You will need to git clone this repo into your VM or computer.
For this lab, you will be modifying the following files:

src/uart_polling.c
src/i2c.c
src/lcd_driver.c
src/keypad_driver.c

src/main.c

Also, please avoid making changes to files other than those listed above, since this will make it much more difficult to grade your submission.

4 GPIO

Before you begin implementing UART and I2C, you need to first understand how to control the GPIOs on your Nucleo board. We have already provided you with the functions to do so in include/gpio.h, so you will not need to write GPIO code. You will only be responsible for calling the functions we have provided:

```
void gpio_init(gpio_port port, unsigned int num, unsigned int mode, unsigned int otype,
               unsigned int speed, unsigned int pupd, unsigned int alt);

void gpio_set(gpio_port port, unsigned int num);

void gpio_clr(gpio_port port, unsigned int num);

uint32_t gpio_read(gpio_port port, unsigned int num);
```

Take a look in gpio.h for a short description of each of the functions, as well as a list of different initialization parameter macros. You should also pay attention to docs/nucleo_f401re_2017_9_19_arduino_right.png and docs/nucleo_f401re_2017_9_19_arduino_left.png. These images show the names and alternate functions of theArduino pins, which will connect to the PCB you will make in later labs.

Notice that the GPIO pins have names like PA_0, PB_7, PC_3, etc. There are three main groups, or ports, of GPIOs on your board, A, B, and C. Each of these has sixteen pins, 0-15. In order to control a specific GPIO pin using the provided functions, you will need to know the letter and number corresponding to the pin. For example, if you want to initialize Arduino pin D0 (the bottommost pin on the right side), you would need to find that the pin is PA_3. Then, you would call gpio_**init()** with parameters: gpio_port port = GPIO_A and unsigned **int** num = 3. Some of the other options for initialization, like pull-up resistors and alternate functions, will be discussed as required for UART and I2C later.

5 UART

First, you will implement UART. This will allow you to debug with minicom or a serial terminal of your choice using print statements. Review the lecture notes about UART if any of the terminology used in this section is confusing. You will be implementing a **polled** UART interface. You fill in the three functions found in src/uart_polling.c:

```
void uart_polling_init(int baud);
void uart_polling_put_byte(char c);
char uart_polling_get_byte();
```

In the next few sections, we will walk you through how to implement UART step-by-step. This is to help ease you into reading datasheets and interacting with MMIO. However, for I2C and ADC, we will not be providing such detailed and thorough guidance. Furthermore, in Lab 3, you will be expanding on your UART implementation to use interrupts rather than polling. As such, you should really try to understand the reason behind each of the steps.

5.1 MMIO on the STM32F401

The first step to programming UART (as well as I2C) is to understand the MMIO layout on the STM32F401. As implied by the name MMIO (memory-mapped IO), the registers for each peripheral device are mapped to memory addresses. This means that in order to communicate with a peripheral, you only need to know which memory addresses corresponds to each register. So how do we find the correct memory addresses?

Most of the information you will need will be in the STM32F401 Reference Manual, which can be found in docs/m4_reference_manual.pdf. First, take a look at pages 38-39 of the reference manual, which has a table ofboundary addresses per peripheral. The lower boundary address is referred to as the base address. The table also

tells you where to find the register map for each peripheral.

As an example, go to the register map for the GPIOs. Here, you will find a table of different registers, as well as an address offset. Because many devices share the same register layout, but different base addresses, it is more efficient to list offsets rather than the full address for each register. To find the full address of a register, simply add the offset to the base address.

5.2 UART (USART) on the STM32F401

The documentation for UART begins on page 506 of the reference manual. You will notice that the STM32 has USART, rather than UART. This is because the STM32 supports synchronous transmission (the S stands for synchronous). However, we will only be implementing the asynchronous UART. One of the reasons for this is that synchronization requires an additional clock line. Unfortunately, this line happens to conflict with our I2C clock. From here on, we will refer to USART as UART.

You should read through the introduction to UART and take a glance at the main features. But there is no need to read through the whole UART section in its entirety. This may be the first time you have had to read through a datasheet, especially one of this length. An important lesson to learn is that it is rarely helpful to read through the entire document. It is a good skill to be able to quickly find specific information as you need it.

Documentation for the different UART registers begins on page 548 of the reference manual. The layout and description of each bit is described here. During your implementation of the UART code, you will need to regularly consult this section of the reference manual.

Rather than hardcoding pointers to each register, the easiest and cleanest way to write to and read from these registers is by using a struct. By lining up the start of the struct with the base address of the peripheral, you can simply index into the fields of the struct to interact with each register. An example using UART is shown below. This code is already been provided for you in src/uart_polling.c.

```
struct uart_reg_map {
    volatile  uint32_t SR;
    volatile uint32_t DR;
    volatile uint32_t BRR;
    volatile uint32_t CR1;
    volatile uint32_t CR2;
    volatile uint32_t CR3;
    volatile uint32_t GTPR;
};

#define UART2_BASE  (struct uart_reg_map *) 0x40004400
#define UART_EN (1 << 13)

struct uart_reg_map *uart = UART2_BASE;
uart->CR1 |= UART_EN;
```

As you can see, even without comments, it is evident that the code above enables UART by setting a bit 13 in UART Control Register 1. Take note of a few things in the example above:

1. Remember volatile should be used when accessing MMIO because peripherals can change register values outside of the normal sequential control flow.
2. Pay attention to the macro for UART_EN. By defining a macro with an informative name, the reader of the code can easily tell what was accomplished without comments (this isn't to say that you do not need to comment. You still do!). Further, by using (1 << 13) rather than the equivalent hex value 0x2000, it is easy to tell that Bit 13 is the enable bit, allowing you to effortlessly compare with the bit layout in the reference manual. This makes it much easier to catch mistakes in your macro definitions.

3. When manipulating specific bits, especially in control registers, it is important not to clobber other bits. By using bitwise OR and AND operators (|= and &=) rather than =, you ensure that you only touch the bits you want.

4. You may have noticed from the datasheet that there are three different UARTs. We will be using UART2, as its RX and TX are mapped directly to the USB port.

We expect your code to conform to the example code above. You may be penalized otherwise.

5.3 Initializing UART (uart_polling_init)

Before you begin sending and receiving bytes, you must properly initialize UART. This can be a fairly tricky process, so we'll walk you through the steps in detail for UART. Pay attention here, you will need to do this yourself for I2C and the ADC!

GPIO Pins

Let's begin with GPIO pin configuration (see Section 4). Let's break down the gpio_init function:

```
void gpio_init(gpio_port port, unsigned int num, unsigned int mode, unsigned int otype,
               unsigned int speed, unsigned int pupd, unsigned int alt);
```

Recall the Arduino pin layout from earlier. You saw that the TX and RX lines for UART2 correspond to pins PA_2 and PA_3, respectively. This gives us the port and num parameters to the gpio_init function (again, see Section 4).

You also saw that each pin can serve several different functions, like digital output, analog input, UART, I2C, etc.. You must specify whether you want the pin to behave as an input, output, or alternate function (like UART). You can do so using the mode argument.

In UART, the TX line must be pushed high and pulled low during transmission. Thus, you should specify the output type (otype) to be push-pull. Because the line is always either actively pulled to ground or pushed to VDD, there is no case in which the line is left floating, so there is no need for additional pull-up or pull-down resistors (pupd).

On the other hand, the RX line is driven by the device you are communicating to (your computer's USB port in this case). So you should leave output type as open drain, with no pull-up or pull-down resistors.

Output speed (speed) refers to slew rate, or the speed at which output voltage changes. Typically, increasing slew rate results in increased circuit noise. Thus, it is good practice to keep slew rate as low as possible for your desired application. In our case, typical UART baud rates are relatively slow, so you may use the lowest output speed.

Lastly, some pins may have multiple alternate functions (alt). For example, PA_7 on the right-side Arduino header can act as an ADC input, SPI MOSI, or PWM output. So you will need to find the correct alternate function for your UART pins. Unfortunately, the GPIO configuration differs slightly in different STM32F401 models. We are using the STM32F401RE, so the information you will need is found in a separate datasheet from the reference manual, the STM32F401xD/E datasheet (docs/stm32f401re.pdf). Look at the table beginning on page 45 of the STM32F401xD/E datasheet. Find the entries for pins PA_2 and PA_3.

You now have all the information you need to initialize the GPIO pins for UART.

NOTE: As mentioned before, we have been helpful enough to have already defined macros in gpio.h that can simply be plugged into gpio_init.

Reset and Clock Control (RCC)

The UART peripheral (i.e. the circuitry handling UART) requires a clock signal to operate. So the next step is to enable the peripheral clock for UART. This is done through the RCC, for which information can be found starting on page 91 of the reference manual. **Note that the clock signal to the device must be enabled before you can interact with any of the device's MMIO registers.** If you find that your MMIO registers are always stuck at 0, you may have forgotten to enable the clock.

There are four different RCC clock enable registers, RCC_AHB1ENR, RCC_AHB2ENR, RCC_APB1ENR, and RCC_APB1ENR. These registers are responsible for enabling the peripheral clock to devices on the Advanced High-Performance Bus (AHB) and Advanced Peripheral Bus (APB).

As a peripheral, UART is naturally on the APB. On reset, the APB peripheral clock runs at 16 MHz. This will be important when setting the baud rate in the next subsection, as well as for configuring I2C later. As a challenge, you can try to find why the APB runs at 16 MHz on reset. Hint: Look at the Clocks section starting on page 93 of the reference manual.

UART's clock enable bit is bit 17 in the RCC_APB1ENR register. You should set this bit to 1 in the same way as shown in the example in Section 5.2. To save you from having to type out a long struct definition, we have already provided you with the register map struct and RCC base address in include/rcc.h.

Baud Rate

Next, let us work out how to set the baud rate. You should review the lecture notes on UART to understand the significance of the baud rate.

On the STM32F401, the baud rate is controlled by the Baud Rate Register (BRR). You should have seen this register in the UART register map already. This register contains the value referred to as USARTDIV. This value is used to divide the UART peripheral clock frequency (f_{ck}) to produce a desired baud rate frequency. The exact formula used can be found on page 519 of the reference manual.

You will need to solve this formula for USARTDIV in order to produce the desired baud rate of **115200** bps. As mentioned previously, UART operates on a 16 MHz clock by default. OVER8 is the oversampling mode. By default, it is set to 0.

Once you have calculated the value of USARTDIV, you will need to convert it to the format used in the BRR register. The top 12 bits of this register contain the mantissa (the part of the number to the left of the binary point) and the lower 4 bits contain the fraction (the part of the number to the right of the binary point).

NOTE: The `uart_polling_init` function calls for the baud rate (baud) as an input. You can choose to input baud = 115200 and perform the baud rate calculation at run time. Alternatively, you can also choose to precompute the USARTDIV value to be written to the BRR register, then define a macro for this value. You can pass then this macro in for the baud argument.

UART Control Register

The final step in UART initialization will be to configure the UART control registers. Documentation for these registers begins on page 551 of the reference manual. In particular, you will need to enable the transmitter, receiver, and UART itself. Each of these three bits can be found in Control Register 1. You should set each of these bits to 1.

Fortunately, the remainder of the options (like word length and number of stop and parity bits) default to the desired value at reset. You will not have to write to these bits, but you should still take a look through them to see what

options you have.

5.4 Sending and Receiving Bytes

After initializing UART, you can now begin sending and receiving bytes. You will do so through the uart_polling_put_byte and uart_polling_get_byte functions.

Put Byte

In order to send a byte via UART, you need to place the byte into the data register. However, before you do so, you need to check that the data register is currently empty. Otherwise, you risk writing over data that has yet to be sent. You should look at the UART status register to find the current status of the data register.

So, you should wait in a while loop while watching this particular bit. Once the data register is empty, you should write your byte and return.

Get Byte

Similarly, in order to receive a byte, you need to read a byte out from the data register. Again, you need to check the current status of the data register. You should wait in a while loop until you see that there is indeed a byte in the data register to be read, before returning with this byte.

You may be wondering how it is possible to use the same register for both sending and receiving data. Won't writing to this register potentially clobber a byte that we just received before we could read it? Fortunately, the data register is in fact composed of two different shift registers. You will interact with the appropriate register depending on whether you are performing a read or a write on the MMIO address.

5.5 Testing UART

The easiest way to test if your UART implementation works is by writing an echo program. In your main function (src/main.c), you should call uart_polling_put_byte and uart_polling_get_byte functions in a loop. Whenever you receivea byte, you should immediately echo the byte back.

You will need to set up minicom (or another serial terminal if already have one you prefer). To do so, follow these steps:

1. Run sudo minicom -s to bring up the configuration window.
2. Navigate to Serial port setup.
3. Press A to change your serial device to the Nucleo. This will likely by /dev/ttyACM0 on Linux or /cu.usbmodem<some number> on Mac.
4. Press E to change the communication parameters. Press E to select a baud rate of **115200**.
5. Return to the main configuration window.
6. Navigate to Screen and keyboard.
7. Press T to set add carriage return to Yes.
8. Save the setup as default.

Now, when you have your board connected to your VM, you can run sudo minicom, which will start minicom up with your saved parameters. You can simply type into the serial terminal. If your implementation is working, you will see the same characters that you typed outputted back at you.

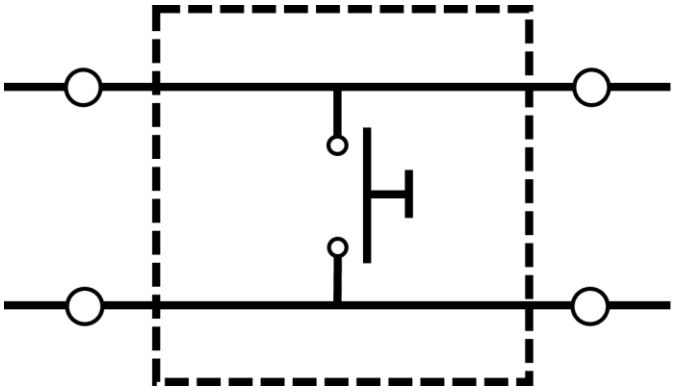


Figure 1: Typical button pinout. Two pairs of legs are internally connected.

5.6 printk

Once you have UART implemented, take a look at src/printk.c. This is a TA written file that imitates *some* of the functionality of the familiar printf you know and love for debugging. This implementation of printk depends on your UART implementation to output characters. If your UART implementation works, then calling printk("hello world") should print to your serial console.

6 LED Outputs

Now, you will learn how to control LEDs with your STM32! Add two LEDs to the breadboard and connect them to two GPIO pins. You should add a resistor between the LED negative pin and ground!

You will need to look up the port and num from the Arduino pin layout. The output type for these should be push-pull. No need for pull-up or pull-down resistors. These pins should have an alternative mapping of 0.

Note: pins D3 and D5 are used for JTAG, so do **not** connect anything to those pins!

7 Button Inputs

Lets get some inputs to our board! Add two push buttons to the breadboard. One end of the push button goes to ground while the other end goes to a GPIO pin. See Figure 1 above for the pinout.

You will need to look up the port and num from the Arduino pin layout. Should these be configured as an input or an output? The output type for these can be push-pull (though this doesn't really matter...why?). Read the push buttons' values using gpio_read. Experiment with adding internal pull-up or pull-down resistors. What does each do? Understanding this will be important for your keypad implementation. You can print the buttons' values to minicom for testing purposes. These pins should have an alternative mapping of 0.

8 Keypad Input

You will be implementing the following functions in include/keypad driver.h:

```
void keypad_init();
char keypad_read();
```

8.1 Initializing the Keypad

To initialize the keypad, take a look at the schematic in docs/sparkfun_keypad.pdf and breadboard the keypad. Using the keypad schematic and the STM32 pinout documentation, figure out how to connect the pins of the keypad to your board and initialize the corresponding GPIO pins in the `keypad_init` function. How should each pin be initialized? Please carefully read the schematic and make sure you understand it before writing any code. Hint: there are 12 buttons on the keypad, but only 7 header pins.

8.2 Reading from the Keypad

Fill in the `keypad_read` function. The function should read each key from the keypad and return an ASCII character corresponding to the key being pressed. If no keys are pressed, it should return the null character (`'\0'`). If multiple keys are being pressed at the same time, it should return any one of the pressed keys.

9 Checkpoint Submission

You should be here in about a week for a checkpoint. To submit this checkpoint, create an issue on GitHub titled Lab2-Submission-Checkpoint. In the comments section, include your and your partner's name, AndrewID, and the commit-hash you want to submit.

10 I2C

Inter-Integrated Circuit (I2C) is a serial protocol for two-wire interface to connect devices such as microcontrollers, I/O interfaces, A/D and D/A converters and other peripherals in embedded systems. It only uses two separate wires called SCL (serial clock) and SDA (serial data). Unlike Serial Peripheral Interface (SPI) protocol, I2C can have more than one master to communicate with all devices on bus. Therefore, it maintains low pin count compared to other protocols. Virtually any number of slaves and masters can be connected onto two signal lines mentioned above. Your next task will be implementing an I2C interface. You will fill in the following functions found in src/i2c.c:

```
void i2c_master_init(uint16_t clk);
void i2c_master_start();
void i2c_master_stop();
int i2c_master_write(uint8_t *buf, uint16_t len, uint8_t slave_addr);
```

Now that you have gotten some practice reading datasheets and interfacing with MMIO for UART, we will not provide you with as detailed a walk-through in the remaining sections. It will be your job to read the datasheets and gather the information you need to properly implement I2C.

10.1 Initializing I2C

As with UART, you will need to first initialize I2C using i2c master init. **Review the UART initialization process thoroughly, and understand why each step is being performed based on the data sheet.** The process for I2C is largely the same as UART. However, there are a few key things to note:

1. You will be using Arduino pins D14 and D15 for I2C communication.
2. In I2C, both SDA and SCL are open-drain. This means that the line can only be actively pulled to ground, and not pushed to VDD.
3. Because SDA and SCL are both open-drain, a pull-up resistor is required to ensure the line is never left floating. It is common practice to always utilize external pull-up resistors for I2C. These should be included in the I2C extender board you are given. Thus, you do not need to use the internal pull-up resistors.
4. You will need to enter the Peripheral Clock Frequency into the I2C Control Register 2. If you need a reminder as to what value this frequency is, refer to Section 5.
5. The clk argument should be used to configure the I2C clock speed to 100kHz. As with the UART baud rate, you will need to perform some calculations based on the peripheral clock frequency. Again, you may either perform this calculation at run time, or simply define a precomputed macro.
6. Use 7-bit addressing.

10.2 Starting and Stopping

As mentioned on page 474 of the reference manual, all transmissions over I2C begin with a Start condition and end with a Stop condition. As the master, we need to generate both of these conditions in software. You will need to fill out i2c_master_start and i2c_master_stop with code to generate these conditions.

Unlike UART, I2C has many timing events. It is important to meet these in order to avoid undefined behavior or race conditions. See the transmission timeline on page 482 of the reference manual. You must check for each of the events. For example, after sending the Start condition, you must wait for EV5 (i.e. wait for SB to be asserted). **You will be penalized for ignoring these events, even if your code works during the demo. Thus, be sure to read the datasheet carefully.**

10.3 Sending Data

Finally, you will fill out i2c_master_write to send data to a specified slave address. You will be able to use the I2C interface to control an LCD display later in the lab. For the purposes of this lab, you will only be required to implement i2c_master_write and not i2c_master_read. However, you may find that implementing i2c_master_read is a good way to double-check and debug your i2c_master_write.

You'll notice from the transmission timeline on page 482 that after sending the Start condition and before sending data, you must transmit the slave address. Review the lecture notes if you need a reminder of the function of the I2C slave address. The address is packaged into a byte with the following format.

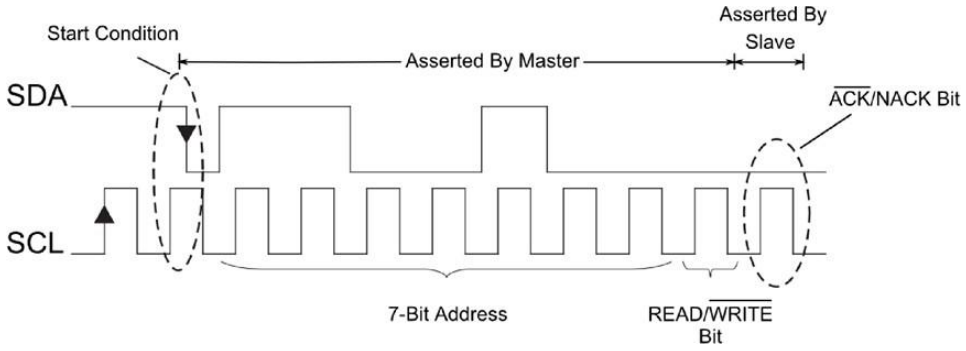


Figure 2: I2C address byte format

As you can see, the topmost 7 bits are slave address (we are using 7-bit addressing). The bottom bit determine whether you are writing to or reading from the device. A 1 in this position indicates a read while a 0 indicates a write. Your job is to take the slave_addr, pad it with the read/write bit, then send it to the SDA line.

After sending the address, you can now send your data (found in buf) one byte at a time. Again, make sure you obey the timing requirements found in the transmission timeline. **Pay close attention to how status bits are cleared. Some bits require multiple conditions to be satisfied to be cleared.**

10.4 Testing I2C

Until you implement your LCD driver, there is no elegant way to test your I2C implementation with code. However, you can sanity check the output of your SDA and SCL lines using an oscilloscope. Oscilloscopes can be found in the lab (A309). If you need help operating the oscilloscope, please come to office hours! During transmission, you should see the clock driven at 100kHz on SCL and your data on SDA.

11 Liquid Crystal Display (LCD) Driver

The LCD you have in your lab kits uses the HD44780U dot-matrix liquid crystal display controller. You will use your I2C implementation to control and print to the LCD. We will be using the PCF8574 extension board to communicate with the LCD using I2C because the HD44780U does not directly accept I2C commands.

The PCF8574 extension board receives signals from the microcontroller through I2C and forwards these signals to the LCD. You can think of the PCF8574 as a translator to convert 8-bit I2C commands to GPIO output to the LCD. Refer to Figure 3 for these connections.

This LCD should be breadboarded and connected to the I2C PCF8574 extender board in your lab kit. This is the black board with the blue potentiometer. To connect properly, ensure the 14 header pins on the LCD align with the 14 header pins on the extender board. The four pins sticking out of the board (GND, VCC, SDA, SCL) should be

connected with wires to your lab board to their associated GPIO pins.

You will use your I2C implementation to communicate with the PCF8574, which connects to the HD44780U-based LCD. To do so, you will need to implement the following functions found in src/lcd_driver.c:

```
void lcd_driver_init();
void lcd_print(char *input);
void lcd_set_cursor(uint8_t row, uint8_t col);
void lcd_clear();
```

You will need to consult the HD44780U and PCF8574 data sheets (found in docs/HD44780.pdf and docs/PCF8574.pdf), respectively. This datasheets contain information on the I2C interface, the I2C transmission timeline, and different commands that need to be sent to the device. See Figure 3 for the internal wiring between the PCF8574 extension board and the LCD.

11.1 Helpful Notes

The PCF8574 takes 8 bits communicated over I2C mapped to P0-P7. It converts the serial I2C commands to a parallel GPIO output. See page 13 of docs/PCF8574.pdf for the order of the I2C bitstream and how it maps to P0-P7. Ensure that you understand the values sent to each P0-P7 pin.

- P0 (connected to RS on the LCD) toggles between instruction mode and data mode. RS=0 indicates that you are sending an instruction (clearing the display, moving cursor, etc.). RS=1 indicates you are writing data (sending characters to display).
- P1 (connected to RW on the LCD) should always be set to 0.
- P2 (connected to E on the LCD) needs to be pulsed (set then unset after a 4 bit packet is sent).
- P3 (not shown in figure) is connected to an internal transistor controlling the LCD backlight and should always be set as 1.
- P4-P7 (connected to D4-D7 on the LCD) controls the data bits for sending instructions and characters to the LCD. The LCD takes xinstructions and displays characters in **8 bits**, but you can only write **4 bits** of data at a time (see Figure 3). The PCF8574 expects a 4 bit packet to be sent once with E=1 followed by the same 4 bit packet with E=0.
- Sending an 8 bit instruction or character should be done within a **single** I2C transmission (between a Start condition and Stop condition).

11.2 Initializing the LCD Driver

Here are a list of steps you must follow to initialize the LCD driver in lcd_driver_init:

1. Carefully read and understand the helpful notes (Section 11.1) above. You may be asked questions about it during checkoff.
2. Find the slave address for PCF8574. You will use this to initialize I2C communication. Hint: look for the slave address on page 13 of the PCF8574 datasheet and reference Figure 3 if needed. You will need to shift this address by 1 bit to the *right* before inputting it into i2c_master_write to account for you adding a write bit before sending data to the data register in your I2C code. As a note A0, A1, and A2 changes the slave address and are set to HIGH (Vcc) as shown in the diagram in Figure 3.
3. Send instructions to initialize the HD44780U. See page 46 in docs/HD44780.pdf for the initialization sequence. Remember P3 should always be set to 1. We will be using the LCD in the **4-bit interface**. Only send the first **four** instructions in the flow chart and ignore the wait times. Make sure to set RS=0 to send instructions. Note that the interface is **8 bits** long for the first four instructions.

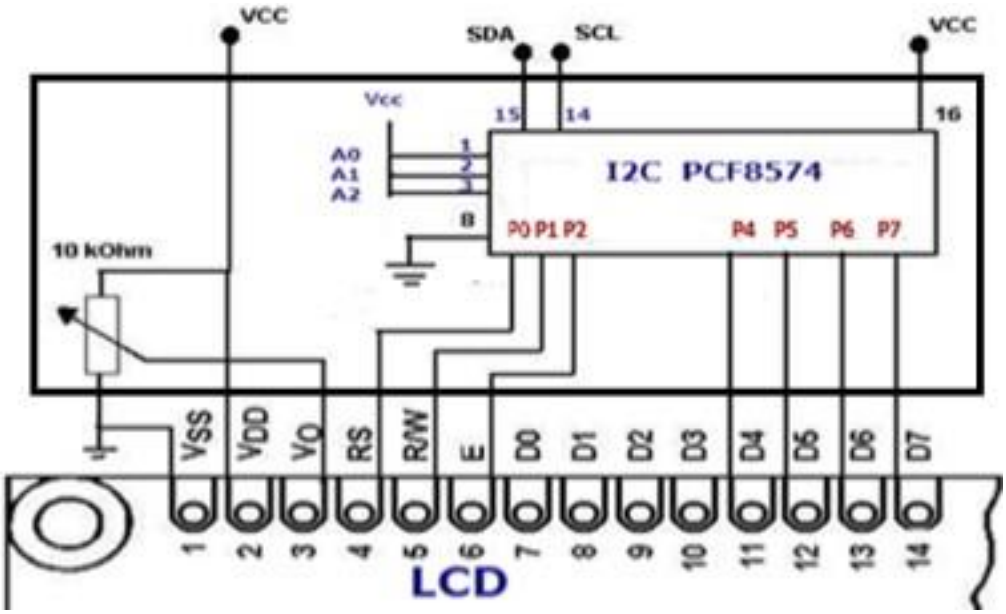


Figure 3: Diagram showing PCF8574 extension board connected to LCD. Ensure you have the extension board and the LCD display in your lab kit. These are separate pieces of hardware. The wired connections shown in the figure are internal and do not require physical connections.

4. Send an instruction to clear the display of the LCD. Refer to page 24 of docs/HD44780.pdf for the clear display instruction.
5. If you don't see anything at first, you might have to change the potentiometer value by turning it with a screwdriver. You can find screwdrivers in the lab during OH. You should see a blinking cursor.

11.3 Printing to the LCD Display

1. lcd_print should print the input string to the LCD display (the input string should be null-terminated).Read carefully through page 11 of docs/HD44780.pdf to understand the addressing of Display Data RAM (DDRAM). Note: we have a 16-character by 2-line display.
2. lcd_set_cursor should move the cursor to the correct row and column. It is important to notice the address offset between the two rows (Hint: see page 11 and 24 of docs/HD44780.pdf). Use the row offset and column number to find the expected location of the cursor and send a command to the LCD driver.
3. lcd_clear should clear the display. Look at the datasheet to see what instruction needs to be sent. Hint: You should have done this already in lcd_driver_init. It can take a while to clear the display, so you should spin in a for loop for several cycles after sending the clear instruction. Clearing takes about 1-2 seconds to take effect.

11.4 Testing the LCD Driver

Now that you have completed your LCD driver, you can now easily test both your LCD driver implementation as well as your I2C implementation. First, call lcd_driver_init. Then, simply print characters to your display by calling lcd_print and check to see if the actual LCD display matches your input. It should print after the cursor. You can move the cursor with lcd_set_cursor. You can clear the display and reset the cursor by calling lcd_clear.

12 Putting it All Together

Now it's time to create an interactive system that uses everything you implemented in this lab! At the end of the day, your interactive system should incorporate all the components from the steps above. You will be implementing a passcode based locking and unlocking system driven by an STM32. Here is what your system should do:

1. On startup, the system should block until **Start** is entered into minicom.
2. Once the system starts, the keypad is used to type in a passcode. Every digit you type should show up on the LCD. The correct passcode should be #633.
3. You will need two buttons. The first button is to "enter" the typed passcode to try and unlock the system. The second button is used to lock a currently unlocked system. Both buttons clear the LCD.
4. You will need two LEDs. The first LED (preferably red) is on whenever the system is locked (passcode is not #633). The second LED (preferably green) is on whenever the system is unlocked (passcode is #633).
5. Every time the system is locked, unlocked, or an incorrect passcode is entered, the minicom should print Locked,Unlocked, or Incorrect passcode, try again!, respectively.

13 Submission

13.1 Github

If you have any comments or suggestions, please feel free to let us know in README.txt.

To submit, create an issue on GitHub titled Lab2-Submission. In the comments section include you and your partner's name, AndrewID, and the commit-hash you want to submit. Be sure to submit the link to your repository to Canvas.

When you are done, it should look like the screenshot on the next page. Pictures of cute animals are not needed but are recommended.

13.2 Demo

Bring your board to any of the TA office hours before the deadline and show us your UART, LCD, LEDs, Keypad and buttons working.

14 Style

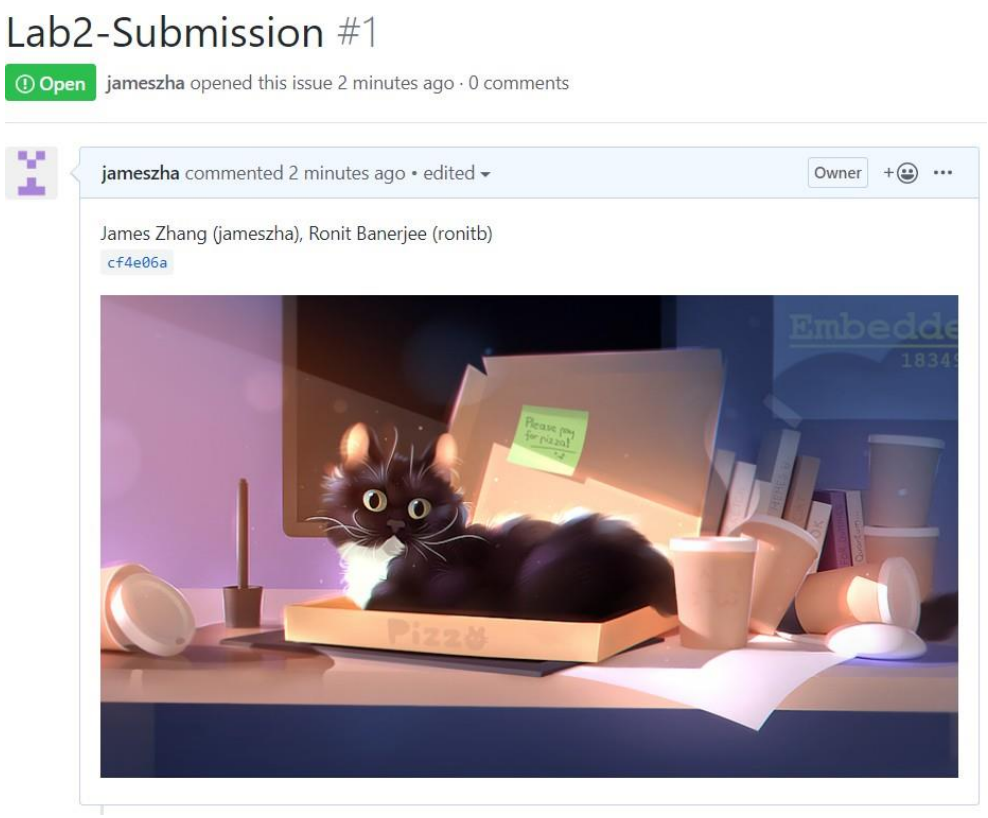
We adhere to the following guidelines very closely while grading, and it is very easy to rack up large point penalties from not conforming to them. Given that for a lot of you this is the first time you are writing code for a embedded system, we want to make sure you develop the right habits. You are expected to follow the guidelines below.

14.1 Good Documentation

Good code should be mostly self-documenting: your variable names and function calls should generally make it clear what you are doing. Comments should not describe what the code does, but why; what the code does should be self-evident. (Assume the reader knows C better than you do when you consider what is self-evident.)

There are several parts of your code that do generally deserve comments:

- File header: Each file should contain a comment describing the purpose of the file and how it fits in to the larger project. This is also a good place to put your name and email address.



- Function header: Each function should be prefaced with a comment describing the purpose of the function (in a sentence or two), the function's arguments and return value, any error cases that are relevant to the caller, any pertinent side effects, and any assumptions that the function makes.
- Large blocks of code: If a block of code is particularly long, a comment at the top can help the reader know what to expect as they're reading it, and let them skip it if it's not relevant.
- Tricky bits of code: If there's no way to make a bit of code self-evident, then it is acceptable to describe what it does with a comment. In particular, pointer arithmetic is something that often deserves a clarifying comment.
- Assembly: Assembly code is especially challenging to read – it should be thoroughly commented to show its purpose, however commenting every instruction with what the instruction does is excessive.

14.2 Good Use of Whitespace

Proper use of whitespace can greatly increase the readability of code. Every time you open a block of code (a function, "if" statement, "for" or "while" loop, etc.), you should indent one additional level.

You are free to use your own indent style, but you must be consistent: if you use four spaces as an indent in some places, you should not use a tab elsewhere. (If you would like help configuring your editor to indent consistently, please feel free to ask the course staff.)

14.3 Good Variable Names

Variable names should be descriptive of the value stored in them. Local variables whose purpose is self-evident (e.g. loop counters or array indices) can be single letters. Parameters can be one (well-chosen) word. Global variables should probably be two or more words.

Multiple-word variables should be formatted consistently, both within and across variables. For example, ”hashtable_array_size” or ”hashtableArraySize” are both okay, but ”hashtable_arraySize” is not. And if you were touse ”hashtable_array_size” in one place, using ”hashtableArray” somewhere else would not be okay.

14.4 Magic Numbers

Magic numbers are numbers in your code that have more meaning than simply their own values. For example, if you are reading data into a buffer by doing ”fgets(stdin, buf, 256)”, 256 is a ”magic number” because it represents the length of your buffer. On the other hand, if you were counting by even numbers by doing ”for (int i = 0; i ¡ MAX; i += 2)”, 2 is not a magic number, because it simply means that you are counting by 2s.

You should use #define to clarify the meaning of magic numbers. In the above example, doing ”#define BUFLEN 256” and then using the ”BUFLEN” constant in both the declaration of ”buf” and the call to ”fgets”.

14.5 No ”Dead Code”

”Dead code” is code that is not run when your program runs, either under normal or exceptional circumstances. These include ”printf” statements you used for debugging purposes but since commented. Your submission should have no ”dead code” in it.

14.6 Modularity of Code

You should strive to make your code modular. On a low level, this means that you should not needlessly repeat blocks of code if they can be extracted out into a function, and that long functions that perform several tasks should be split into sub-functions when practical. On a high level, this means that code that performs different functions should be separated into different modules; for example, if your code requires a hashtable, the code to manipulate the hashtable should be separate from the code that uses the hashtable, and should be accessed only through a few well-chosen functions.

14.7 Consistency

This style guide purposefully leaves many choices up to you (for example, where the curly braces go, whether one-line ”if” statements need braces, how far to indent each level). It is important that, whatever choices you make, you remain consistent about them. Nothing is more distracting to someone reading your code than random style changes.