

Lab 2: Device Drivers and MMIO

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

14-642, 18-349 Introduction to Embedded Systems

Code Due: 11:59PM EST Tuesday, March 9, 2021

All Demos Due: Tuesday, March 16, 2021

Contents

1	Introduction and Overview	3
1.1	Goal	3
1.2	Task List	3
1.3	Grading	3
2	Starter Code	3
3	GPIO	4
4	UART	4
4.1	MMIO on the STM32F401	5
4.2	UART (USART) on the STM32F401	5
4.3	Initializing UART (uart_polling_init)	6
4.4	Sending and Receiving Bytes	8
4.5	Testing UART	8
4.6	printf	9
5	I2C	9
5.1	Initializing I2C	9
5.2	Starting and Stopping	10
5.3	Sending Data	10
5.4	Testing I2C	10
6	Seven-Segment LED Display Driver	11
6.1	Initializing the LED Driver	11
6.2	Setting the LED Display	11
6.3	Testing the LED Driver	11
7	ADC	11
7.1	Initializing the ADC	12
7.2	Performing an ADC read	12
7.3	Testing the ADC	12
8	Clap/Snap/Tap Detector and Light Sensor	12
8.1	Requirements	12
8.2	Tips	13
9	Bonus (+5 Points)	13
10	Submission	13
10.1	Github	13
10.2	Demo	13
11	Style	13
11.1	Good Documentation	13
11.2	Good Use of Whitespace	14
11.3	Good Variable Names	14
11.4	Magic Numbers	15
11.5	No "Dead Code"	15
11.6	Modularity of Code	15
11.7	Consistency	15

1 Introduction and Overview

1.1 Goal

The goal of this lab is to use Memory Mapped IO (MMIO) to interface with peripheral embedded devices. You will be responsible for implementing the drivers for UART and I2C serial communication as well as the on-board ADC. You will finally interface with your custom PCB to measure sound and light levels and display them to the seven-segment LED display.

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. Get the starter code from GitHub classroom (see Section 2)
2. Implement UART (see Section 4)
3. Implement I2C Driver (see Section 5)
4. Implement Seven-Segment LED Display Driver (see Section 6)
5. Implement ADC Driver (see Section 7)
6. Implement Light Sensor and Clap Detector (see Section 8)
7. Submit to GitHub (see Section 10). Make sure your code has met style standards!
8. Submit the link to your repo to Canvas
9. Demo UART, LED, Clap detector, and light sensor

1.3 Grading

Start this lab early to give yourself ample time to debug. All code submitted code must compile and execute 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	15
i2c.c	20
led_driver.c	20
adc.c	20
kernel.c (LED, Light Sensor, Clap detection)	15
Style and Submission Protocol (see Sections 10 and 11)	10
Bonus (see Section 9)	(+5)
TOTAL	100 (+5) pts

2 Starter Code

As with Lab 1, use the magic assignment link on Canvas to create your repo for Lab 2. You will need to `git clone` this repo into your VM.

For this lab, you will be modifying the following files:

```
kernel/asm/boot.S
kernel/src/uart_polling.c
kernel/src/i2c.c
```

```
kernel/src/led_driver.c
kernel/src/adc.c
kernel/src/kernel.c
```

You should copy over your bootloader from Lab 1 into `kernel/asm/boot.S` in the current repo. Also, please avoid making changes to files other than those listed above, since this will make it much more difficult to grade your submission.

3 GPIO

Before you begin implementing UART, I2C, and ADC, 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 `kernel/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);
```

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 the Arduino pins, which will connect to the PCB you made in Lab 0.

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, I2C, and ADC later.

4 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 `kernel/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.

4.1 MMIO on the STM32F401

The first step to programming UART (as well as I2C and the ADC) 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 of boundary 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.

4.2 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 individual 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 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 `kernel/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 even 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.

4.3 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 3). 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` parameter to the `gpio_init` function (again, see Section 3).

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 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_APB2ENR`. 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 on in Section 5. 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 4.2. 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 `kernel/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.

4.4 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.

4.5 Testing UART

The easiest way to test if your UART implementation works is by writing an echo program. In your kernel (`kernel/src/kernel.c`), you should call `uart_polling_get_byte` and `uart_polling_put_byte` in a loop. Whenever you receive a 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 be `/dev/ttyACM0`.
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.

4.6 printk

Once you have UART implemented, take a look at `kernel/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.

5 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 `kernel/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.

5.1 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. Check your PCB design to figure out which Arduino pins and which I2C you will be using 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. Indeed, these are included on your PCB. 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 4.
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.

5.2 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.**

5.3 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 the seven-segment LED display on your PCB. For the purposes of this lab, you will only be required to implement `i2c_master_write` and not `i2c_master_read`. This is because reading from the ht16K33 LED driver simply returns the exact data that you wrote to it, which is largely unspectacular. 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.

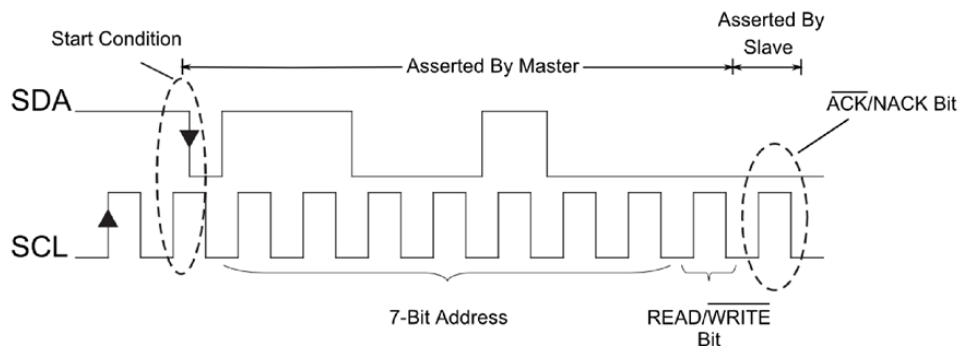


Figure 1: I2C address byte format

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.**

5.4 Testing I2C

Until you implement your LED driver, there is not an elegant way to test your I2C implementation. However, there are a few things you can do to sanity check:

1. Check the output of your SDA and SCL lines using an oscilloscope. Oscilloscopes can be found in the lab (HH1303). If you need help operating the oscilloscope, come to office hours! During transmission, you should see the clock driven at 100kHz on SCL and your data on SDA.

2. Try writing to and reading from the LED driver RAM. You should be able to read out the same value you wrote into the RAM. This will require that you have implemented `i2c_master_read`. You will also need to have an understanding of the LED driver (see Section 6), so this testing method should only be attempted after you have given the LED driver implementation some thought.

6 Seven-Segment LED Display Driver

The LED Driver you have on your PCB is the `ht16K33`. You will use your I2C implementation to communicate with the `ht16K33`. To do so, you will fill in the following functions found in `kernel/src/led_driver.c`:

```
void led_driver_init();
void led_set_display(uint32_t input);
```

You will need to consult the `ht16K33` (found in `docs/ht16K33v110.pdf`). This datasheet contains information on the I2C interface, the I2C transmission timeline, and different commands that can be sent to the device.

6.1 Initializing the LED Driver

Here are a list of steps you must follow to initialize the LED driver:

1. Determine the `ht16K33` slave address. Hint: You may need to look at your PCB schematic.
2. Send commands to the `ht16K33` to turn the oscillator on, turn the display on, and set the display to full brightness. Hint: the brightness/dimming is controlled via PWM.
See <https://learn.sparkfun.com/tutorials/pulse-width-modulation/all> for more information.
3. Clear the RAM.

6.2 Setting the LED Display

When your program calls `led_set_display`, you should print the `input` as hex on the LED display. We have already provided you with `hex_to_seven_segment` in `led_driver.c` to help you convert hex into seven-segment display format.

The `ht16K33` has sixteen 8-bit RAM rows used to store display information. This allows the `ht16K33` to control 16×8 or 128 individual LEDs. Of course, your display only has four seven-segment displays, so you do not need every row. Each of rows 0 (most significant), 2, 6, and 8 (least significant) correspond to a digit in the LED display. You will need to write the seven-segment display formatted hex digits into each of these rows via I2C.

6.3 Testing the LED Driver

Now that you have completed your LED driver, you can now easily test both your LED driver implementation as well as your I2C implementation. Simply set your display to specific values and check to see if the actual LED display matches.

7 ADC

The ADC, or Analog to Digital Converter is used to convert analog sensor values to digital 1s and 0s. You will use the ADC to take in the analog signal from the light sensor and microphone on your PCB. The ADC is the final peripheral you will need to interface with in this assignment. You will fill in the following functions found in `kernel/src/adc.c`:

```
void adc_init();
uint8_t adc_read_pin(uint8_t pin_num);
```

As with I2C, while we will provide you with some guidance, you will need to make use of the datasheets to gather the remaining necessary information to implement the ADC.

7.1 Initializing the ADC

Once again, initialization of the ADC is similar to the initialization of UART and I2C. Here are a few notable points:

1. You must configure your GPIOs based on which pins your light sensor and microphone are connected to. This will depend on your PCB design.
2. When configuring the GPIO to analog input, the output type and speed parameters are ignored.

7.2 Performing an ADC read

Telling the ADC which pin to read from and convert to a digital value can be a little difficult. Take a look at the Arduino pin layout once again. You may notice that each of the analog pins has an alternate function like ADC1/0, ADC1/1, or ADC1/4. The number following the / is the ADC channel. The ADC identifies pins by channel number, rather than the port and number you have become accustomed to previously. You will need to select the correct channel to read from and convert to a digital value.

You will be performing single conversion reads. Make sure you obey timing constraints. You may find that your obtain outdated readings from your sensors otherwise.

7.3 Testing the ADC

In order to test your ADC, you should call your `adc_init` and `adc_read` on the appropriate `pin_num`. The ADC reading for your light sensor is typically more stable and reliable. As you shine a light of varying intensity on the light sensor, you should see the value returned by `adc_read` change.

8 Clap/Snap/Tap Detector and Light Sensor

Now we want you to show us that you can put all of the parts together. You will use your ADC interface to read the value from a light sensor. You will then use I2C to display this value on the seven-segment display on your PCB. Finally, you will program a clap detector to detect when you clap, snap, or tap on the microphone.

8.1 Requirements

1. Implement `kernel_main` that polls for claps and light sensor readings.
2. `kernel_main` should prompt the user for a number from 0 to 9.
3. When the user enters the number 0, you should display the value read from the light sensor on the LED.
4. When the user enters a number between 1-9, you should begin detecting claps. Once you detect the given number of claps, you should print a completion message. This is what we expect as the output from your serial terminal:

```
Enter a number from 0-9:
0
Enter a number from 0-9:
5
Clap detected!
Clap detected!
Clap detected!
Clap detected!
Clap detected!
DONE!
Enter a number from 0-9:
```

8.2 Tips

Your microphone readings will vary from board to board. Play around with different thresholds and methods until you can reliably detect claps. For example, you can try taking the average of several readings, taking peak-to-peak measurements of several readings, etc.

9 Bonus (+5 Points)

If you have finished the lab early and would like some extra credit, here's your opportunity.

You may have realized that while polling in UART, you are unable to perform other useful work. Because of this, while you are waiting for the user to enter a number, you are currently unable to read from the ADC or update the LED display. Modify your program so that your LED displays the current light sensor reading at all times, even while waiting for user input. Below are a few requirements you must meet:

1. Other than this addition of real-time light sensor value display, your program must still meet the requirements of the non-bonus code as listed above.
2. There should be no noticeable slowdown of the UART prints or the clap detector.
3. Your UART, ADC, LED driver, and I2C code should be independent from each other, i.e. UART should not call I2C code, I2C should not call UART code, etc..

10 Submission

10.1 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 your and your partner's name, AndrewID, and the commit-hash you want to submit. If you completed the bonus, please submit an additional issue, titled **Lab2-SubmissionBonus**, with the bonus commit hash. Finally, 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.

10.2 Demo

Bring your board to any of the TA office hours before the deadline and show us your UART, LED, light sensor, and clap detector working.

11 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 an embedded system, we want to make sure you develop the right habits. You are expected to follow the guidelines below.

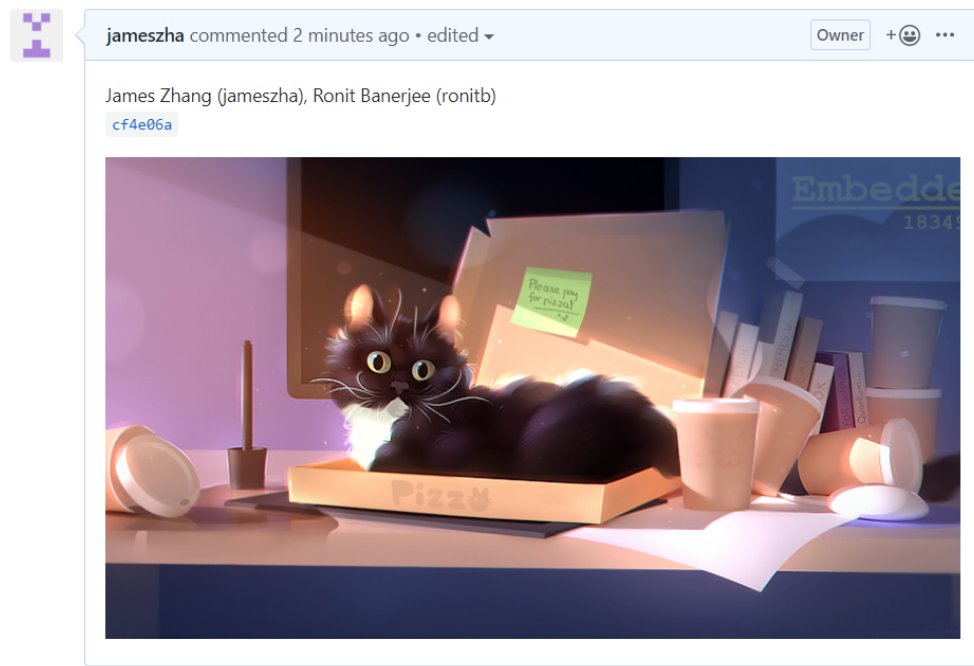
11.1 Good Documentation

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:

Lab2-Submission #1

 Open jameszha opened this issue 2 minutes ago · 0 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.

11.2 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.)

11.3 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 to use "hashtable_array_size" in one place, using "hashtableArray" somewhere else would not be okay.

11.4 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".

11.5 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.

11.6 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.

11.7 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.