

Lab 1: 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: 3:00PM EST Thursday, February 15, 2018

All Demos Due: 4:00PM EST Tuesday, February 20, 2018

1 Introduction and Overview

1.1 Goal

The goal of this lab is to gain experience interacting with Memory Mapped IO (MMIO) by interfacing with embedded peripheral devices, as well as gaining experience working with timer interrupts. During the first part of this lab, you will implement the supporting software required for the UART peripheral for the Raspberry Pi, as well as configure the built-in timer. You will use this timer to measure performance and see the results of optimizing ARM assembly code. In the second part of the lab, you will implement a driver for the I2C peripheral, and the ADS1015 Analog-to-Digital converter that is off-chip on your breakout board. During this lab you will implement the supporting software required for UART and I2C peripherals for the Raspberry Pi. Finally, you will use this driver to implement a simple clap (impulse) detector that prints a message every time a clap is heard using a simple cyclic executive architecture. Make sure to read through the lab handout and tips carefully before beginning.

1.2 Task List

1. Get the starter code from GitHub classroom (see Section 2)
2. Implement UART (see Section 3)
3. Implement Timer tic/toc (see Section 4)
4. Optimize kernel_optimization challenge (see Section 5)
5. Implement I2C Driver (see Section 6)
6. Implement ADC Driver (see Section 7)
7. Implement Clap Detector (see Section 8)
8. Demo optimized kernel, clap detector and light sensor
9. Submit to GitHub (see Section 9). Make sure your code has met style standards including Doxygen documentation!
10. 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 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.c | 15 |
| tictoc.c | 10 |
| assembly optimization and timing | max: 35 |
| 2x faster | 5 |
| 5x faster | 15 |
| 7x faster | 20 |
| 10x faster | 25 |
| 13x faster | 30 |
| 16x faster | 35 |
| i2c.c | 10 |
| ads1015.c | 10 |
| kernel.c (Clap detection) | 10 |
| Style (see Section 10, Doxygen documentation, following submission protocols) | 10 |
| TOTAL | 100 pts |

1.4 Doxygen

Doxygen is a framework that allows you to automatically generate documentation from comments and markup tags inserted directly into the source. This style of embedding documentation in source is often used in industry. We will be using `doxygen` for code documentation for this course moving forward. The `doxygen` manual is available here if unfamiliar with it: <http://www.stack.nl/~dimitri/doxygen/manual/index.html> Examples of `doxygen` comments in the code can be found in all of our handouts. Steps for setup and use of `Doxygen` are below:

Installation in the VM:

```
$ sudo apt-get install doxygen
```

Generating documentation:

```
$ make doc
```

The handout code contains a default configuration file called `doxygen.conf`. While most of the tags provided in the file already have the correct value, you are responsible for making sure that the `INPUT` tag specifies the correct source files (for this lab and future labs). **Specifically, you will need to have zero `doxygen` warnings for `kernel_optimization` and `kernel_clapdetector`. This means changing line 761 in `349util/doxygen.conf` to `kernel_optimization` for `make doc` to run `doxygen` on `kernel_optimization`.**

If the above command runs successfully, then you should have a `/doc` directory with an `index.html` file. View it locally in a browser to see the documentation created from the code you wrote. When running this command, a file called `doxygen.warn` should have been created in the directory you ran `make doc`. Open this file to inspect any warnings. If there are any documentation warnings in the file about code you have written, then fix them. The TAs will check this file and take off style points if there are *any* warnings in this file. Please see the TA written code in `349libk/` for example `doxygen` code documentation.

2 Starter Code

Use the assignment "magic link" to create your repo for Lab 1. You will need to git clone this repo into your VM.

3 UART

First we will implement UART. This will allow us to debug with `ftditerm.py` using print statements. Review the lecture notes about UART if any of the terminology used in this section is confusing. We will be implementing a **polled** UART interface.

3.1 MMIO on the Raspberry Pi

Before we can configure the UART and I2C, we must understand the MMIO layout on the Raspberry Pi.

All MMIO on the Pi begins at `0x3F000000`. Depending on the peripheral you are accessing, the offset from this address changes. In the `BCM2835.pdf` datasheet, all MMIO addresses given are in **virtual address form**. For example, on page 9 of the `BCM2835.pdf` datasheet, the `AUXENB` register is listed at address `0x7E215004`. Accessing this address directly *will not work*. When running bare metal, the MMU (Memory Management Unit) that does Virtual to Physical address translation is turned off. So we must convert this virtual address to a physical one. Again, for the Raspberry Pi 2, all MMIO on the Pi begins at `0x3F000000`. So the **physical address** of the `AUXENB` register would be `0x3F000000 + 0x00215004 = 0x3F215004`. You basically just replace the upper byte of the address with `0x3F`.

Include `BCM2836.h` (located in `349libk/include/BCM2836.h`), and use `MMIO_BASE_PHYSICAL` to avoid ugly MMIO addressing bugs by writing code like the following (in your `uart.c`):

```
#include <BCM2836.h>
#include <kstdint.h>
```

```
#define AUXENB_REG    (volatile uint32_t *) (MMIO_BASE_PHYSICAL + 0x215004)
```

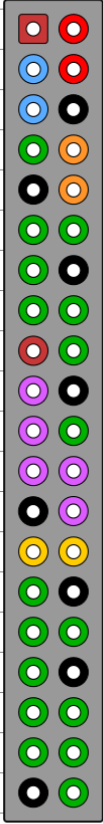
NOTE: Remember `volatile` is used when accessing MMIO because peripherals can change register values outside of the normal sequential control flow.

Insight into why this translation occurs as we have stated previously can be gathered by studying page 5 of the `BCM2835.pdf` datasheet. On this page, the memory map on the left represents the shared virtual memory layout between the GPU and ARM CPU. The memory map in the center represents the physical memory layout seen by an ARM CPU. The memory map on the far right represents the virtual memory layout of a given ARM CPU if we were going to use the ARM MMU. Starting with the memory map on the left, we see that I/O Peripherals (just another name for MMIO) is addressed virtually at `0x7E000000`. Trying to access this address directly would be a problem since our given ARM CPU only has access to the 2 memory maps in the center and on the right depending on if the ARM MMU is on or off. The memory map on the left is only visible to the GPU (remember the GPU is king!).

Depending on if the ARM MMU is on or off, we would use either the memory map on the far right or in the center. For this course **we will not be using the MMU**. We ignore the map on the right and use the one in the center. Now we see that I/O Peripherals are mapped to the physical address `0x20000000` for the ARM CPU. The process of using `0x3F000000` as the base instead is found by substituting in the physical address of GPU peripherals from page 3 of the `BCM2836.pdf` datasheet for the `0x20000000` in the `BCM2835.pdf` datasheet. This is because the Pi 2 uses the BCM2836 SoC and not the BCM2835 SoC. Overall, the MMIO environment between the 2 chips is essentially the same except for this main difference.

3.2 Using GPIO on the Raspberry Pi

The GPIO layout begins on page 89 of the `BCM2835.pdf` datasheet. The MMIO base offset for GPIO is `0x7E200000`, which is `0x3F200000` on the RPi 2. Each of the GPIO pins has multiple functions. This is best illustrated in the table on pages 102 to 103. Each GPIO pin has different functions it can serve as from `ALT0` to `ALT5`. The GPIO pin numbers in the far left column of the table correspond to the following GPIO pin layout on the header which our breakout board is attached to:

| Raspberry Pi2 GPIO Header | | | | |
|---------------------------|------------------------------------|--|------------------------------------|------|
| Pin# | NAME | | NAME | Pin# |
| 01 | 3.3v DC Power |  | DC Power 5v | 02 |
| 03 | GPIO02 (SDA1 , I ² C) | | DC Power 5v | 04 |
| 05 | GPIO03 (SCL1 , I ² C) | | Ground | 06 |
| 07 | GPIO04 (GPIO_GCLK) | | (TXD0) GPIO14 | 08 |
| 09 | Ground | | (RXD0) GPIO15 | 10 |
| 11 | GPIO17 (GPIO_GEN0) | | (GPIO_GEN1) GPIO18 | 12 |
| 13 | GPIO27 (GPIO_GEN2) | | Ground | 14 |
| 15 | GPIO22 (GPIO_GEN3) | | (GPIO_GEN4) GPIO23 | 16 |
| 17 | 3.3v DC Power | | (GPIO_GEN5) GPIO24 | 18 |
| 19 | GPIO10 (SPI_MOSI) | | Ground | 20 |
| 21 | GPIO09 (SPI_MISO) | | (GPIO_GEN6) GPIO25 | 22 |
| 23 | GPIO11 (SPI_CLK) | | (SPI_CE0_N) GPIO08 | 24 |
| 25 | Ground | | (SPI_CE1_N) GPIO07 | 26 |
| 27 | ID_SD (I ² C ID EEPROM) | | (I ² C ID EEPROM) ID_SC | 28 |
| 29 | GPIO05 | | Ground | 30 |
| 31 | GPIO06 | | GPIO12 | 32 |
| 33 | GPIO13 | | Ground | 34 |
| 35 | GPIO19 | | GPIO16 | 36 |
| 37 | GPIO26 | | GPIO20 | 38 |
| 39 | Ground | | GPIO21 | 40 |
| Rev. 1 26/01/2014 | | http://www.element14.com | | |

The breakout board wires the correct GPIO pins to the right peripherals, but you must configure the correct GPIO pins to breakout the desired functions (ALT0 to ALT5) for the right GPIO pins when using UART and I2C. To make this easier on you, the TAs have implemented a GPIO library for you to use when setting up UART and I2C. This code exists in `349libk/include/gpio.h` and `349libk/src/gpio.c`. To help you understand this library, we will walk through an example of how to configure GPIO on the Raspberry Pi with UART.

3.3 UART GPIO Example

We will now walk through how to configure the GPIO for UART to help explain how the GPIO library works. Look at `kernel/include/uart.h`. Notice the GPIO pin numbers for the RX and TX lines of UART. We need to configure these pins to enable UART.

```
/** @brief GPIO UART RX pin */
#define RX_PIN 15
/** @brief GPIO UART TX pin */
#define TX_PIN 14
```

Before we configure the pins to the right function according to the table on page 102 of the `BCM2835.pdf` datasheet, we must handle the pull-up/down resistor on each GPIO pin with the GPIO library. Pull-up/down resistors are commonly used with microcontrollers (MCUs). Sparkfun has a great tutorial on what these are here:

<https://learn.sparkfun.com/tutorials/pull-up-resistors>

NOTE: You **disable** pull-up/down resistors for data lines like RX and TX since they are data lines and should only be read when driven. In your `uart_init` function, you will have the following:

```
// configure GPIO pullups
gpio_set_pull(RX_PIN, GPIO_PULL_DISABLE);
gpio_set_pull(TX_PIN, GPIO_PULL_DISABLE);
```

Now we use the GPIO library to configure the pins for the correct functions listed in the GPIO table page 102 of the `BCM2835.pdf` datasheet:

```
// set GPIO pins to correct function on pg 102 of BCM2835 peripherals
gpio_config(RX_PIN, GPIO_FUN_ALT5);
gpio_config(TX_PIN, GPIO_FUN_ALT5);
```

After this point, the UART interface on the Pi is now available on the GPIO pins we configured! You will need to do this for the I2C interface pins we have defined for you in `kernel/include/i2c.h` when initializing I2C in this lab.

3.4 Setting the baud rate

`ftditerm.py` is a serial console. A serial console is used often in embedded systems for debugging and as a user interface to an embedded system. Most embedded systems don't have a keyboard or mouse, so UART is used as a method of communicating character bytes to a serial console in order to display text almost like a terminal. We will use `ftditerm.py` as our serial console when communicating with the Raspberry Pi. In Lab0 you ran the following command:

```
$ sudo ftditerm.py -b 115200
```

Now that we are implementing UART, you can dive deeper into the parameters of this command. The `-b` flag specifies the baud rate for the serial console (which is 115200 for this case). Then `ftditerm.py` searches for the FTDI minimodule you have connected and starts a serial console on the port the FTDI minimodule is attached to. If you get an error trying to setup a serial console, then you probably have not connected the FTDI minimodule to your computer via the USB cable.

3.5 UART in the BCM2835 datasheet

For implementing UART, you will find pages 9 - 19 of the `BCM2835.pdf` datasheet very useful. Most of the information you will need is in those pages.

NOTE: YOU SHOULD ALWAYS CHECK THE DATASHEET ERRATA BEFORE YOU CODE ANYTHING! it is quite common for datasheets to have incorrect information on them. As an embedded designer, you should always check the datasheet errata before implementing any code based off the datasheet alone. The errata for the `BCM2835.pdf` datasheet can be found [here](http://elinux.org/BCM2835_datasheet_errata):

http://elinux.org/BCM2835_datasheet_errata

The quality of this datasheet is terrible. Read the errata. You'll thank us later.

3.6 The UART Interface

When implementing UART, you will use the predefined interface found in `kernel/include/uart.h`. This file has the function definitions and descriptions of what you must do. Your UART implementation should be in `kernel/src/uart.c`. They are listed here for reference:

```
/**
 * @brief initializes UART to 115200 baud in 8-bit mode
 */
```

```

void uart_init(void);

/**
 * @brief closes UART
 */
void uart_close(void);

/**
 * @brief sends a byte over UART
 *
 * @param byte the byte to send
 */
void uart_put_byte(uint8_t byte);

/**
 * @brief reads a byte over UART
 *
 * @return the byte received
 */
uint8_t uart_get_byte(void);

```

3.7 UART Tips

To help you out, here are a few tips to guide your implementation:

1. The `AUXENB` register is used to enable *access* to the MMIO peripherals of UART. This should be the first thing you do in `uart_init()`.
2. For the equation on page 11 of the `BCM2835.pdf` datasheet, the *system_clock_freq* is 250MHz.. You need to solve this equation for `baudrate_reg` and put that value into the appropriate register.
3. The `AUX_MU_IER_REG` register should be set to 0. You should not enable interrupts for UART (we will do this in lab 2 with the ARM timer instead).
4. In the `AUX_MU_IIR_REG` register, you only care about the bits pertaining to *clearing* the FIFOs.
5. Do not set `DLAB` access inside of the `AUX_MU_LCR_REG` register.
6. Ignore the `AUX_MU_MCR_REG` and `AUX_MU_MSR_REG` registers.
7. Ignore details about CTS and RTS in all UART MMIO registers.
8. The `AUX_MU_BAUD` register is where you should put your baud value after solving the equation on page 11 for `baudrate_reg`.
9. Watch out for pointer math. `#define A ((volatile int *) 0x8000) #define B ((volatile int *) (A+0x10))` does NOT do what you want. It gives 0x8040 for B instead of 0x8010.
10. `uart_put_byte` and `uart_get_byte` are very short. Use `UART_LSR_REG` to determine status and `UART_IO_REG` for IO.

3.8 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")` in an infinite loop in `kernel_main` should show up in `ftditerm` while the serial console is running!

4 ARM timer

Next, you will implement timer functionality that you can use in the next section to profile code. For this part of the lab, you only need to implement two functions. `tic()` and `toc()`. You will extend your timer functions to support interrupts, it may prove useful to make helper functions to `tic()` and `toc()`

4.1 ARM Timer

By default, the 32-bit timer on the ARM is set to decrement, so we want to load the initial value of timer as `0xFFFFFFFF` (max time) and configure the timer to count down at a rate where we can see at least millisecond time granularity. As `kernel_optimization/include/kernel.c` file suggests, `tic()` will zero and start the timer while `toc()` will return the number of milliseconds that have elapsed since `tic()`. Make sure that your code doesn't run so long that the timer underflows. Since the timer value is decrementing, you will need to make necessary adjustments when returning the elapsed time. The description of the ARM timer starts on page 196 of the BCM2835.pdf datasheet. Again we use MMIO registers to configure the timer operations. The function definitions in `kernel_optimization/include/tic.toc.h` describe the interface you need to implement, which is in `kernel_optimization/src/tic.toc.c`.

```
/**
 * @brief Configures the arm timer to start running with the given frequency. The Timer
 *        should run in 32 bit mode, with a prescaler of 1.
 */
void tic();

/**
 * @brief Called to check the value of the timer.
 *
 * @return time ticks that have elapsed since tic()
 */
uint32_t toc(void);
```

HINT: You only need to look at 3 registers, Load, Value and Control. As noted above, use 32-bit mode with no prescaling.

5 ARM Optimization

In the next part of the lab, we will apply your knowledge of assembly programming towards optimizing a simple assembly program. The goal in this case will be to decrease the length of time it takes to execute a section of the program.

5.1 Optimizing ARM Assembly

You can start out by running the kernel optimization part of the lab with:

```
$ sudo make PROJECT=kernel_optimization gdb
```

This will link in and test two assembly code files (`optimize_me.S` and `unoptimized.S`) that are located in the `kernel_optimization/src/` directory. If you run the test kernel with your timer function, you will see that the two functions will have relatively similar system tick counts. Your goal is to modify `kernel_optimization/src/optimize_me.S` file to run faster (goal of 16x faster) by applying optimization techniques mentioned in lectures. The two arrays resulting from two codes must be identical and we might test your new assembly code against an unknown test vector. Don't try to optimize based on the input vector.

6 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. We will be implementing a I2C interface.

6.1 The I2C Interface

When implementing I2C, you will use the predefined interface found in `kernel/src/i2c.h`. This file has function definitions and descriptions of what you must do. The I2C implementation should be in `kernel/include/i2c.c`. They are listed here for reference:

```
/**
 * @brief initializes the I2C module
 *
 * @param clk_div bus clock speed, put this value directly into the CDIV register.
 */
void i2c_master_init(uint16_t clk_div);

/**
 * @brief writes to I2C device
 *
 * @param buf pointer to output data buffer
 * @param len length of output data buffer in bytes
 * @param addr slave device address
 */
uint8_t i2c_master_write(uint8_t *buf, uint16_t len, uint8_t addr);

/**
 * @brief reads from I2C device
 *
 * @param buf pointer to input data buffer
 * @param len number of bytes to read
 * @param addr slave device address
 */
uint8_t i2c_master_read(uint8_t *buf, uint16_t len, uint8_t addr);
```

Take note of the following that we have put into `i2c.h`

```
// I2C pins
#define I2C1_SDA 2
#define I2C1_SCL 3

// I2C Clock speeds
#define I2C_CLK_100KHZ 0x5dc
```

You can pass `I2C_CLK_100KHZ` to `i2c_master_init`.

6.2 I2C tips

1. The I2C documentation goes from page 28 to 37 in the `BCM2835.pdf` datasheet.
2. You do not have to send the I2C address via the data register. It should not be in the `buf` for `i2c_master_write`.

3. Data is always sent MSB first on the Raspberry Pi.
4. You don't have to worry about sending/receiving more than 16 bytes on I2C at a time, so you can assume it will all fit in the FIFO queue. You may return an error if `i2c_master_read` or `i2c_master_write` is given a length > 16.
5. Remember to always check the errata!

Testing I2C by itself is unfortunately difficult without a reference device to test against. In this case, the ADC is probably your best choice (its the only thing wired up to I2C). We also recommend looking at online resources for I2C on the Raspberry Pi if you are stuck. As always, feel free to reach out to the course staff if you get stuck.

7 ADC Driver

The ADC, or Analog to Digital Converter is used to convert analog sensor values to digital 1s and 0s. The ADC driver is a software peripheral that will enable polling of the light and sound sensors on your Raspberry Pi 2 breakout board. To communicate with the ADC, we will use the I2C interface you just wrote! This part of the lab will require looking through the `ads1015.pdf` datasheet of the ADC to understand how it works and how to communicate with it. **WARNING!** Many students get confused by page 8 of `ads1015.pdf` and try to send the I2C address in the data register. You **only** put the I2C address in the address register. The I2C controller takes care of sending this address across the wire.

7.1 ADC Driver Interface

When implementing the ADC Driver, you will use the predefined interface found in `kernel/include/ads1015.h`. This file has function definitions and descriptions of what you must do. Your ADC driver implementation will go inside of `kernel/src/ads1015.c`. The function definitions are listed here for reference:

```
/**
 * @brief initialize ADS1015
 */
void adc_init(void);

/**
 * @brief read a value from the ADC
 *
 * @param channel 0 through 3
 * @return the value read from the ADC
 */
uint16_t adc_read(uint8_t channel);

#endif /* _ADC_DRIVER_H_ */
```

7.2 ADC Driver tips

1. `ads1015.pdf` datasheet describes the I2C setup for the ADC.
2. Do not try to create the "first byte" values on p. 8 of `ads1015.pdf`. Those are handled by the address register of the I2C controller.
3. Pay close attention to Table 5 on page 14 to get the slave address.

8 Clap Detector and Light Sensor

Now we want you to show us that you can put all of the parts together. We will use I2C in our ADC driver to read the light and sound sensors and then use UART to display the raw sensor data in `ftditerm`. Finally, we will use this raw sensor data to detect when a clap or loud impulse occurs.

8.1 Requirements

1. Implement `kernel_main` that polls and prints sensor values for both light and sound.
2. Refer to `rpi_ioboard.pdf` to check how light and microphone sensors are connected to ADC.
3. `kernel_main` should prompt the user for a 0 or 1 to select which ADC channel(multiplexor mode) to listen to.
4. When the user enters a 0, you should print the value of the light sensor over UART and then prompt the user for another sensor to sample.
5. When the user enters a 1, you should sample the microphone sensor continuously until a clap occurs. After the clap, you should prompt the user for another sensor to sample.
6. Microphone sensor data processing is based of a peak-to-peak measurement.
7. You may notice a garbage value when you switch from light to sound or vice-versa. It is OK to read and discard one value when you switch.

Here is the output over `ftditerm` that we are expecting (verbatim):

```
$ sudo make PROJECT=kernel gdb
```

```
Enter a sensor to sample:
Light: 1583
Enter a sensor to sample:
Light: 1584
Enter a sensor to sample:
Light: 1583
Enter a sensor to sample:
Audio: 64
Audio: 61
Audio: 49
Audio: 56
Audio: 62
Audio: 49
Audio: 59
Audio: 50
Audio: 2047
Clap detected!
Enter a sensor to sample:
```

In our circuit, the audio signal is a voltage centered around half of the supply voltage (single ended) that swings up and down as the sound pressure changes. In order to estimate the intensity of the signal you will need to extract a feature that indicates how much energy or volume there is in the signal. One simple approach would be to measure the peak-to-peak intensity of the signal across a number of samples. When returning audio volume, make sure to sample for a significant number of samples (say 100 or 1000) and return the max minus the min as the peak-to-peak value. Experiment with the rate and number of samples to improve your ability to distinguish a clap from the background noise. Keep in mind that the CPU will operate at a different frequency when running from JTAG as opposed to without the debugger enabled.

9 Submission

To submit the final, use the tag `lab1-submit`. Also, be sure to submit the link to your repository on Canvas.

Push documented and completed code to the GitHub Repository. Make sure to read over the submission instructions at [GitHub.pdf](#) for more details. You should always submit what you have done instead of submitting nothing.

9.1 Demo

1. Run your optimized kernel and show the speedup and that it passes the tests.
2. Run your kernel and show the light sensor value changing. (Bring a phone or flashlight to shine on the sensor). Also show the audio sensor detecting a clap

10 Style

Adapted from: <https://www.cs.cmu.edu/~213/codeStyle.html>

10.1 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.

10.2 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.)

10.3 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.

10.4 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"`.

This is especially important when putting constants in memory-mapped registers.

10.5 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.

10.6 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.

10.7 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.