ECE 362 Lab Experiment 3: General Purpose I/O

Introduction

In this experiment, you will learn how to connect simple input devices (push buttons and keypad) and simple output devices (LEDs) to an STM32 development board. You will learn how to configure the General Purpose Input/Output (GPIO) subsystems. You will read values from and write values to these pins.

Instructional Objectives

- Connect LEDs and push buttons/keypad to the STM32 GPIO pins.
- Configure the GPIO subsystems.
- Read values from and write values to the GPIO pins.

Table of Contents

* All the points for this lab depend on proper completion of and submission of your post-lab results.

When you are ready for your lab evaluation, review this checklist.

Step 0: Prelab Exercises:

- Be familiar with lectures up to and including (05) General Purpose I/O.
- Read this entire lab document.

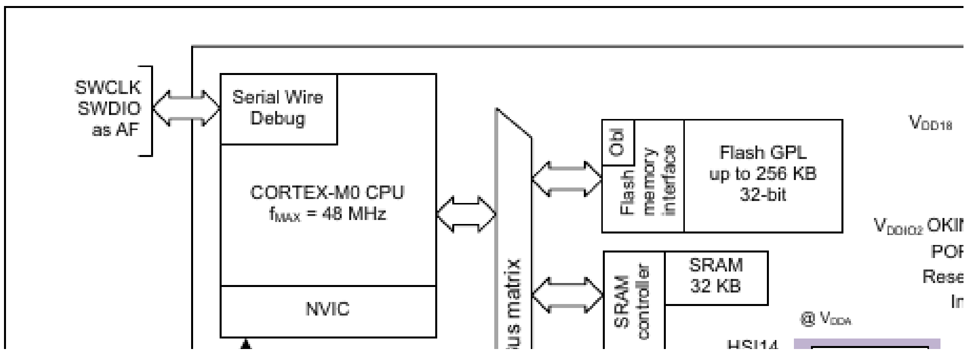
- Ensure that your development board and serial port are working **before** you start the lab.
- Do the prelab exercises and submit them **before** attempting the lab experiment.

Step 1: Background

1.1 Enabling and configuring the GPIO subsystem

By default, every subsystem in a microcontroller is disabled in order to reduce power consumption, and the GPIO subsystem is no exception. Each *port* must first be enabled by turning on a clock for the port controller. Only after that, can the port be configured. The following two steps are needed to configure I/O pins for use:

First, the Reset & Clock Control (RCC) subsystem must be configured to enable a GPIO port for use. This is done by turning on the specific bit in the AHBENR (Advanced High-Performance Bus Enable Register). In doing so, be sure not to turn OFF important bits such as the SRAMEN bit. Use your lecture notes to determine how to do this. For this lab, in particular, you should enable Ports B and C by ORing the appropriate values into the AHBENR register.



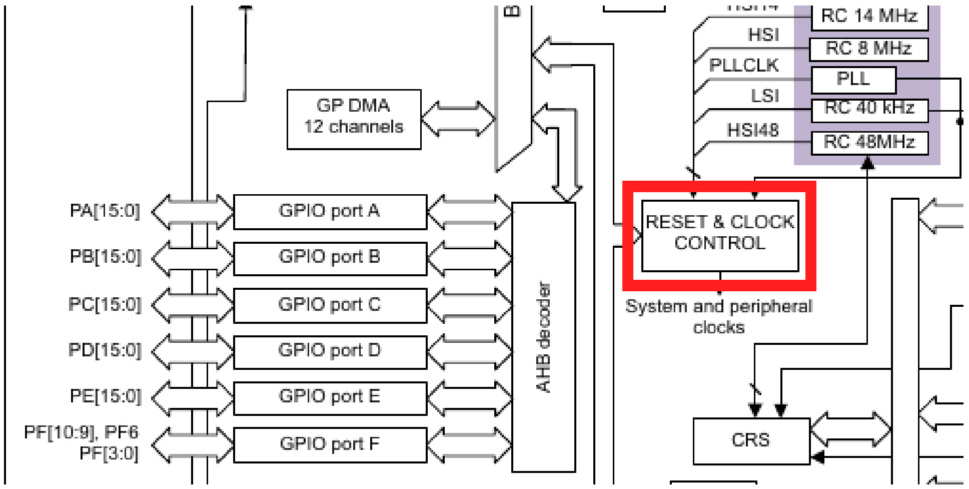
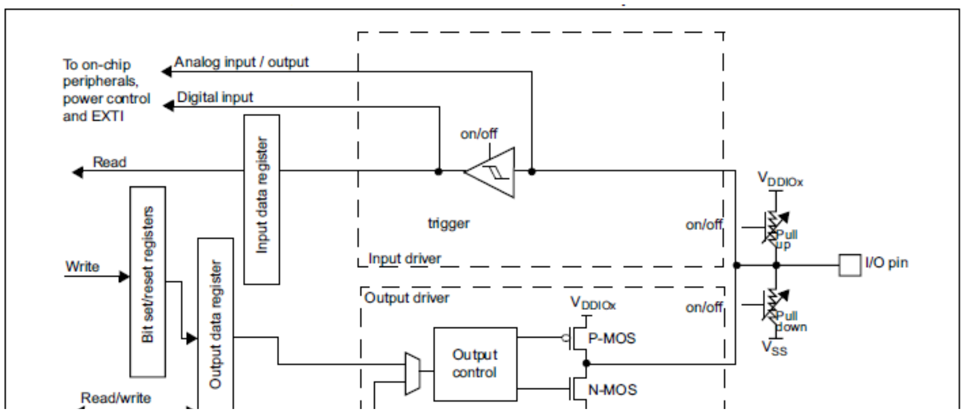


Figure 1: Reset and Clock Control

Second, an enabled port must have its pins configured as either inputs or outputs (or some other functions that we will use in future labs). For this lab, we will initially use Port B pins as inputs and Port C pins as outputs. For instance, the inputs used will be pins PB3 and PB4 (pins 3 and 4 of Port B). For both ports there are some pins whose function you should not modify. In particular, you should not change PA13, PA14, PC14, or PC15 since these are used for an External Oscillator, Debugging, and Programming interfaces. Instead, use code from your lecture notes to learn how to set and clear only the bits you want without altering other bits.

A functional diagram for an STM32F0 GPIO port is shown in Figure 2, below:



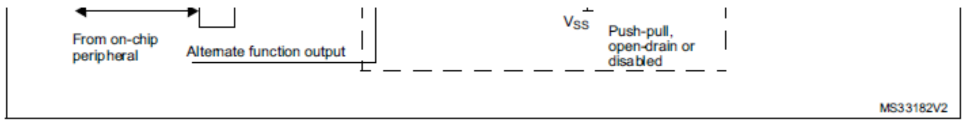


Figure 2: GPIO Pin Functional Diagram

In this experiment, you will enable general purpose (digital) inputs and outputs. That will involve using the input data register (IDR) and output data register (ODR) for each port. We will consider analog inputs and outputs in future lab experiments.

2.0 Experiment

Create a project in SystemWorkbench called "lab3", and delete the automatically-generated main.c file. Create a main.s file in the src directory. In order to expedite setting up the framework for the code you will write, as well as getting all of the constants you will need to throughout the lab, a *template* file is provided for you. You should download main.s and copy the contents into your own lab3/src/main.s file to get started.

This experiment uses a pre-compiled object file that contains test functions to automatically test the subroutines you write for this lab. It is called autotest.o. To incorporate it into your System Workbench project, follow the instructions in the ECE 362 References page for Adding a pre-compiled module to a project in System Workbench.

Every subsection of this step (2.1, 2.2, ..., etc) requires a checkoff by your TA. Do NOT wait until the end of lab to check everything off.

2.1 Configuring Port B

Using the main.s template file provided for this lab, fill in the assembly language procedure for `initb`. This is should **enable the RCC clock for Port B, set pins PB8-PB11 as outputs and set pins PB0 & PB4 as inputs**. Please refer to the notes for a refresher on how this is done. This should be done without disabling other pins in RCC such as SRAMEN, configuring other pins as outputs, etc. Only the bits needed to complete the required configuration should be changed. Recall *what a subroutine is, how it needs to be written in assembly, do not forget to push and pop the right register(s)* .

The global symbol "RCC" is provided for you to use as the base address for the Reset and Clock Control I/O registers. Use the global symbol "AHBENR" as an offset into the RCC registers to access the AHBENR I/O register. For instance, you can read the current value of the RCC_AHBENR into R1 with the code:

```
ldr  r0,=RCC
ldr  r1,[r0,#AHBENR]
```

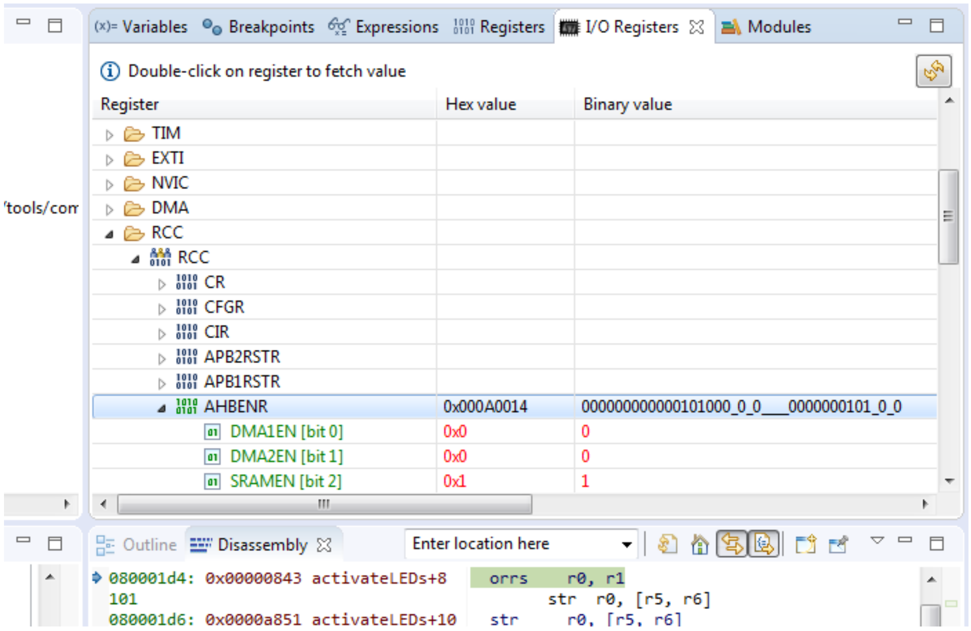
Similarly, the global symbol GPIOB is provided along with the symbols for MODER, IDR, ODR, BSRR, etc. Use these in a similar manner to RCC and AHBENR to configure pins PB8-PB11 as output pins (and to read and write their driven values in later sections.)

If you look at the documentation in the Family Reference Manual regarding a GPIO port's MODER register, you will notice that all pins that can be used as inputs are set as inputs by default. We want you to get some practice clearing bits of the MODER as well as setting bits. Your subroutine should also configure pins 0 and 4 of Port B as inputs *with the assumption that they may have already been configured as outputs*. This means you should clear the two-bit value for each pin in question to `00`.

The **`autotest`** module provided for you will call this and every other subroutine you write muliple times in different ways. It will deliberately set and clear other bits in the AHBENR to make sure that you are doing a proper OR operation to set only the necessary bits and not setting or clearing any other bits. The procedure for ORing bits into a configuration register may seem cumbersome, but it will occur repeatedly for many types of peripherals. You'll soon get used to it.

When debugging individual subroutines you should leave the call to **`autotest`** commented out so that you can step through the operation of each subroutine. You must complete each step of implementation in sequence. You will not get credit for steps 2 through 6 without first satisfactorily completing step 1.

Hint: When debugging your code, you may consult the "I/O Registers" tab in the Monitor window. This tab provides a hierarchical listing of all of the various peripherals utilized by the microcontroller and their sub-registers. By default their values are not shown; double clicking on a given entry will cause the debugger to track the values of that entry and its lower elements. Use this to confirm that the various registers you are writing to are taking on the values which you expect. An example of this is shown in Figure 3, below. When you cannot understand why a GPIO port is behaving as you expect, take a look at the registers and make sure their contents are being set correctly. This is a process that you will use for the rest of the semester.



Double-click on register to fetch value

Register	Hex value	Binary value
TIM		
EXTI		
NVIC		
DMA		
RCC		
RCC		
CR		
CFGR		
CIR		
APB2RSTR		
APB1RSTR		
AHBENR	0x000A0014	000000000000101000_0_0___0000000101_0_0
DMA1EN [bit 0]	0x0	0
DMA2EN [bit 1]	0x0	0
SRAMEN [bit 2]	0x1	1

```
080001d4: 0x00000843 activateLEDs+8      orrs     r0, r1
101                                              str  r0, [r5, r6]
080001d6: 0x0000a851 activateLEDs+10     str      r0, [r5, r6]
```

Figure 3: I/O Registers Viewer

You can also use the I/O Registers viewer to modify values to experiment with register settings. As you progress through the lab experiment, you will find that you can use the I/O Registers viewer to deposit values into the ODR of Port C. This will cause LEDs to illuminate. You can also examine the value of the Port B IDR in real time to determine if a button is pressed. Note that the same rules apply just as if you wrote code to do things programmatically. You will not be able to view or adjust values for any of the registers of a GPIO port before enabling the clock to that subsystem in the RCC AHBENR. You can experiment with doing that manually as well.

2.2 Configuring Port C

For this section, you will need to complete the subroutine **`initc`** in main.s so that GPIOC's clock is enabled in RCC and it configures only pins 4, 5, 6 and 7 of Port C as outputs. These pins will drive the colunms of the keypad (the wiring and schematic for which will be introduced later). Set pins 0, 1, 2, and 3 as inputs and to use the internal pull down resistors by setting the relevant bits in the PUPDR register. These will read the rows of the keypad and need to be pulled down to keep from reading erroneous floating signals. You should use the symbolic constant GPIOC as the address of Port C. The symbol "MODER" is the offset from the base address for the mode register. For instance, you can read the current value of the MODER register of GPIOC into R1 by doing the following:

```
        ldr  r0,=GPIOC
        ldr  r1,[r0,#MODER]
```

A similar process can be used to read and set values in the PUPDR register.

Reminder: Be sure not to change the configuration for pins 13, 14, and 15. Once again, make sure to push and pop the right register(s).

2.3 Set PORTB pin outputs

For this section, you will need to complete the subroutines `setn` in main.s. The first parameter (register R0) is the bit number (the same as the pin number) to be set (by `setn`). The second parameter (given by R1) is the value to set the pin to (either zero or anything not zero). The `setn` subroutine turns off the pin/bit number (R0) in the GPIOB_ODR register if the value in the second parameter (R1) is zero. Otherwise, it turns on the pin/bit number (R0). For instance, if you execute `setn(8,0)`, it would turn pin 8 off. If you invoked it as `setn(8,4)` it would turn pin 8 on. Use the global symbol "ODR" as the offset from a GPIO port to the output data register. You are encouraged to use the BRR and BSRR registers to implement these subroutines.

2.4 Read PORTB pin inputs

Complete the subroutine `readpin` in main.s. This function accepts a bit number as parameter 1 (register R0). It returns the value (in register R0) in GPIOB_IDR at the bit specified by R1. In other words, the subroutine should return 0x1 if the pin is high or 0x0 if the pin is low.

2.5 Wiring LEDs, Push Buttons, and Keypad

Unplug the programmer the USB port to remove power from the development board before you wire components. Anytime you plug things in to the microcontroller when the power is on, you run the risk of connecting something you shouldn't have

and causing damage. By removing power, you have the opportunity to double-check your work before you reconnect power.

After you remove the power from your development board, wire your circuit as shown in Figure 4, below. Ensure that you are connecting resistors for the buttons to the 3V power. You are advised to connect the power buses on your breadboard array to the GND and 3V connections on the development board. (Leave the 5V pins unconnected to avoid using them.)

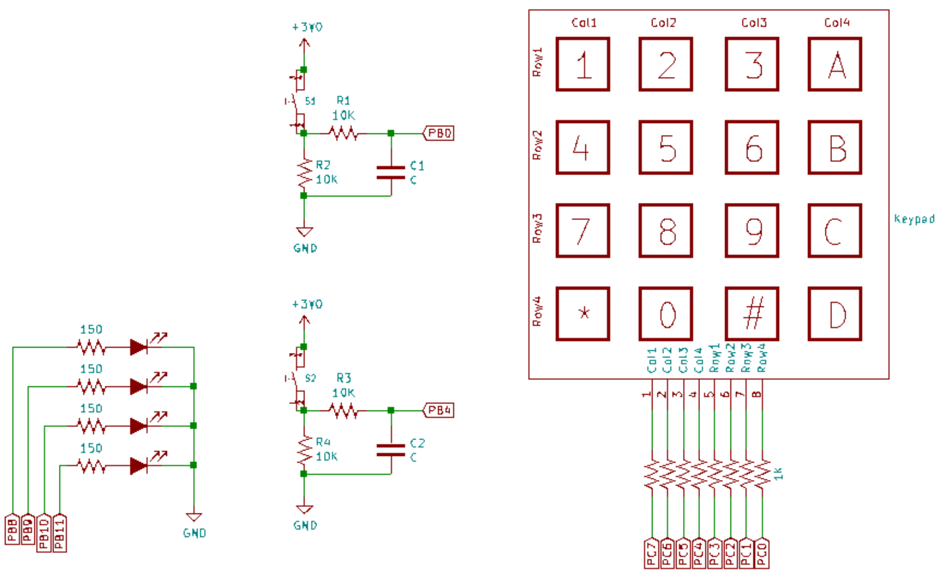


Figure 4: Schematic for wiring LEDs and push buttons

Use your choice of red, yellow, and green LEDs to build the circuit in Figure 4. To determine the orientation of an LED, look at the length of the leads. The long lead of the LED is usually the *anode* (the positive terminal) and the short lead is the *cathode* (the side connected to ground in the schematic). You can also search for a flattened section of the circular rim of the LED. That also indicates the cathode.

Use four-lead "tactile" push buttons for the switches. The 6x6mm push buttons work the same way as the 12x12mm push buttons. Note that pins 1 and 2 are connected together internally as well are pins 3 and 4. The push button is "normally open". When it is pressed, pins 1 and 2 are connected to pins 3 and 4. See the diagrams in Figure 5 for more details.

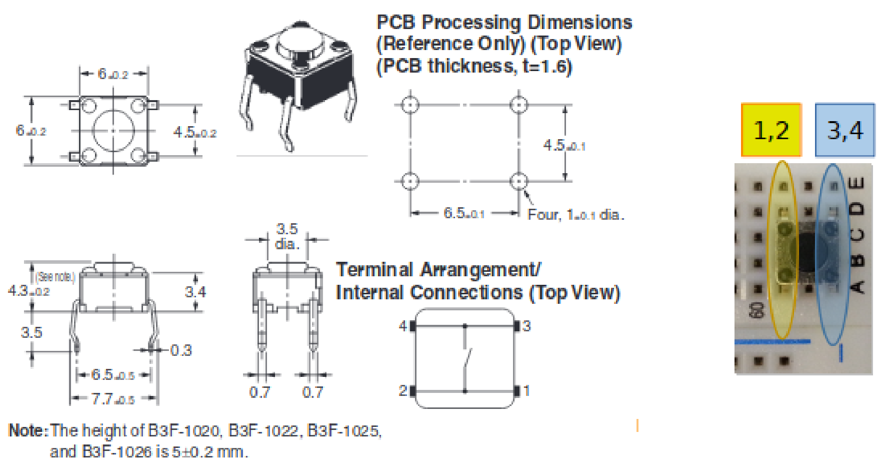

PCB Processing Dimensions
(Reference Only) (Top View)
(PCB thickness, t=1.6)
Four, 1 dia.
Terminal Arrangement/
Internal Connections (Top View)
Note: The height of B3F-1020, B3F-1022, B3F-1025, and B3F-1026 is 5±0.2 mm.
1,2
3,4

Figure 5: Tactile switch diagrams

You will use a 16-button keypad for this lab experiment. One configuration for such an arrangement of buttons would be to have two wires for each button (32 wires total) so that each button could be independently monitored. That would require a lot of pins on the keypad which might make it difficult to use on a breadboard. An optimization might be to have one common power connector and one more connector per button (17 wires total) so that each button could still be monitored individually. But this is still too many pins on the keypad than manufacturers will usually support, and the numbers would only be much worse for keypads with even more buttons.

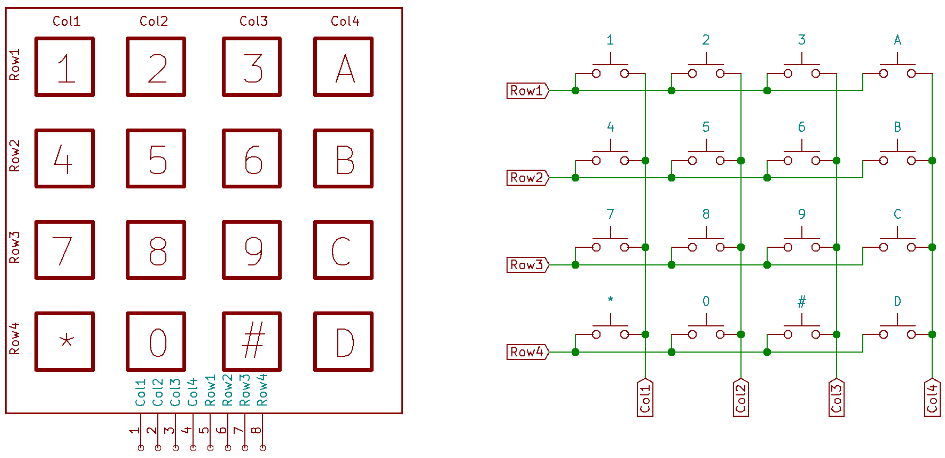


Figure 6: Keypad internal schematic

The typical concession for keypads is to have multiple common pins that connect to groups of buttons. The logical extension to this trend is to configure a keypad as a matrix of rows and columns. If the number of rows and columns is about equal, this practice minimizes the number of pins on the keypad. The added challenge is that the keypad must be scanned one row or column at a time. For instance, a voltage can be applied to a row of buttons, and the columns can be sensed to detect if any of the buttons in the row are being pressed. Then the voltage can be removed from the row and re-applied to another row. By cycling through all the rows, all keys can eventually be detected.

When you are finished, your circuitry should look something like Figure 7, below. You can, of course, arrange your circuitry in any manner you want, but we will be using the keypad in this configuration for the rest of the semester, It might be good if you keep it as close to this picture as possible.

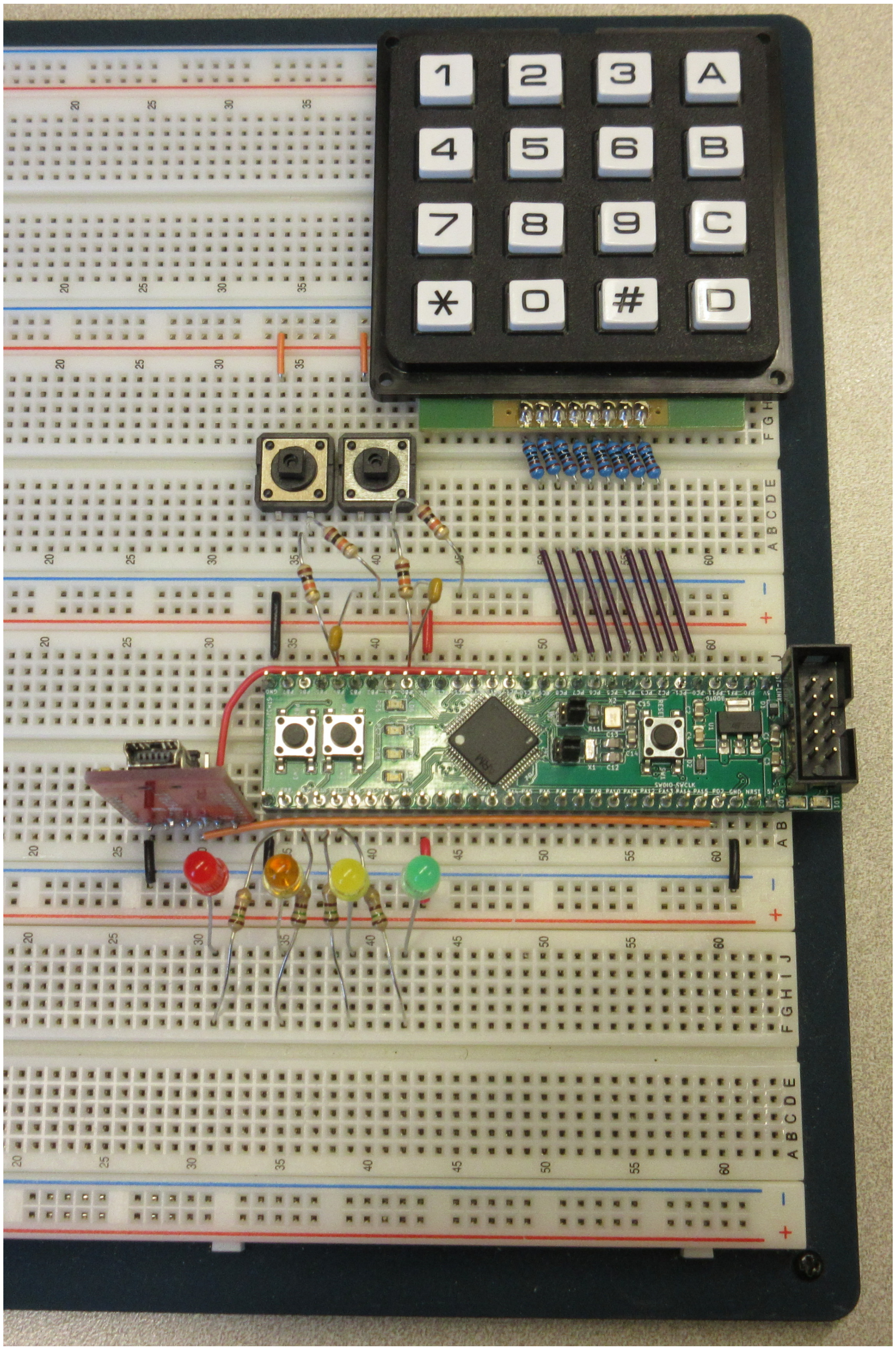


Figure 7: Breadboard wiring

2.6 Buttons and LEDs

Complete the subroutine **buttons** in main.s. This function will be called repeatedly in a loop. The purpose of this function is to map the button presses (inputs on PB0 and PB4) to two of the 4 LEDs (on PB8-PB11). So if a button is pressed an LED should turn on and once it is released the LED should turn off. Same goes for the other button and a different LED. A sample implementation in C is given below, but yours doesn't have to be implemented like this.

```
void buttons(void)
{
  setn(8, readpin(0));
  setn(9, readpin(4));
}
```

2.7 Keypad and LEDs

Complete the subroutine **keypad**. It will be called in an infinte loop. The subroutine should iterate through all the column pins (PC4-PC7), setting one high and the rest low on each iteration. In each iteration, it should check all the row input values (PC0-PC3). For each column, if a button in a certain row is pressed, a light should turn on, and when it is released, it should turn off. No two columns can use buttons on the same row and no two columns can use the same LED. A sample implementation in C is given below, but your implementation doesn't have to match ours.

```
void keypad(void)
{
  for(int c=8; c>0; c = c>>1) {
    GPIOC->ODR = c << 4;
    mysleep();
    int r = GPIOC->IDR & 0xf;
    if (c == 8) {
      setn(8, r & 1);
    } else if (c == 4) {
      setn(9, r & 2);
    } else if (c == 2) {
      setn(10, r & 4);
    } else {
      setn(11, r & 8);
    }
  }
}
```

Our implementation just uses the buttons along the diagonal of the keypad, but you can use any pairs you'd like, given the stipulations above.

Notice the function **`mysleep`** in the body of the function above. This is needed because if you were to press both the 7 and * key (for example), the light wouldn't turn on, even though with the code above, it should. This is because the microcontroller sets and checks the pins too quickly, not giving the driving column enough time to charge the row lines (think of an RC circuit)! With the call to **`mysleep`** inserted in between the set and check, this issue is averted. **`mysleep`** can be implemented simply as a empty spinning for loop.

```
void mysleep(void)
{
  for(int n=0; n<1000; n++)
    ;
}
```

3 Submit your postlab results

Once your lab experiment works to your satisfaction, use autotest to test your subroutines. Show your progress to your TA to get credit.

For this lab, and most to follow, you must also submit the program that you wrote so that it can be checked by the course staff. Either upload the file or copy it from SystemWorkbench and paste it into the text box. Make sure that your entire program is shown there.

4 Clean up your lab station

Make sure you leave your lab station clean for the next person. Log out of your workstation. If you need help, ask your TA.

Lab Evaluation Checklist

Normally, we'll have a checklist of things that you should review before going into steps of your evaluation.

- ☐ Did you do all of the required reading as well as prelab exercises before attempting the lab?
- ☐ Did you confirm that your circuitry is correct before you powered it up?
- ☐ Did you use autotest to ensure that all of your hardware was working properly?
- ☐ Did each subroutine work correctly before you moved on to another? They depend on each other for this lab.
- ☐ Did you clean up your lab station and log out?