Lab 4 - Breadboarding + Capacitive Touch

Goals

- Develop circuits using a breadboard
- Use the Microbit to interact with external components

Equipment

- Computer with build environment
- Micro:bit and USB cable
- Breadboard + various components (we'll distribute these in lab)
- Microbit breakout adapter

Documentation

- nRF52833 datasheet: https://docs-be.nordicsemi.com/bundle/ps_nrf52833/attach/nRF52833_PS_v1.7.pdf
    - Online version: https://docs.nordicsemi.com/bundle/ps_nrf52833/page/keyfeatures_html5.html
- Microbit schematic: https://github.com/microbit-foundation/microbit-v2-hardware/blob/main/V2/MicroBit_V2.0.0_S_schematic.PDF
- Lecture slides are posted to the Canvas homepage

Github classroom link: https://classroom.github.com/a/w6M-ZUlC

Lab 4 Checkoffs

You must be checked off by course staff to receive credit for this lab. This can be the instructor, TA, or PM during a Friday lab session or during office hours.

- **Part 2:** Breadboarding
    a. Demonstrate that you can power a single LED
    b. Demonstrate that you can properly read an analog input
    c. Demonstrate your application that controls an RGB LED with various inputs
        - Also discuss the code you wrote to do this
    d. Return your cleaned up parts
- **Part 3:** Capacitive Touch
    a. Demonstrate your rise time measurement
    b. Demonstrate that you can detect touch once at boot
    c. Demonstrate that you can continuously detect touch state
        - Also discuss the code you wrote to do this

Also, don't forget to answer the lab questions assignment on Gradescope.

Lab Steps

Part 1: Setup

1. Find a partner

- Rule: you can pick any partner you want, but you can't pick the same partner twice
- You MUST work with a partner
    - If you can't find someone, talk to Branden

2. Create your Github assignment repo

- There is a github classroom link on the first page of this document. Click it!
- Pick a team name
- Pick your partner
- Generally, do what it says
- At the end, it should create a new private repo that you have access to for your code
    - Be sure to commit your code to this repo often during class!
- That link might 404. If it does, you first have to go to https://github.com/nu-ce346-student and join the organization
- **Important: both of you should join the repo before you can do the next step**

3. Set up an additional Git remote

- Open a terminal if you haven't yet
- `cd` into your "nu-microbit-base" repo
- At the top right of your shiny new private repo on the Github website, there is a green button that says "Code". If you set up an SSH key, you can click the SSH tab to get that URL, otherwise you should get the HTTPS URL. Either way, copy the URL so you can enter it into terminal
- `git remote add lab4 <YOUR-REPO-URL-HERE>`
    - This adds a "remote" repo hosted on github as a source for this repo
    - (Both of you should still do this step too)

4. Individual Setup Portions

- **ONLY ONE OF YOU** should do the following steps
    - `git fetch lab4`
        - This gets the most recent commits from the new remote source
    - `git checkout lab4/main`
        - This changes your current commit to the remote source's main branch
    - `git switch -c lab4-code`
        - This makes a new branch for your lab code
    - `git push -u lab4 lab4-code`
        - This tells the new branch to push code to the new remote source
        - From now on, you can just pull, commit, and push as normal
- **THE OTHER STUDENT** should do this **AFTER** the first student finished the above steps:
    - `git fetch lab4`
    - `git switch lab4-code`
- **BOTH STUDENTS** should do this
    - `git submodule update --init --recursive`
        - Makes sure all git submodules are initialized and updated

Part 2: Breadboarding

1. Find the app starter files for this lab

- `cd software/apps/breadboard/`
    - This lab will use the files in this directory. Your changes will be in `main.c`

    - **IMPORTANT:** This lab builds upon itself. So keep things on the breadboard unless told to remove them. They'll be useful for the final checkoff.

2. Talk with your partner about your prior experience

- In this lab especially, you should make sure that the person with the least hardware experience hooks stuff up on the breadboard. The more experienced person can watch as they hook stuff up (and warn them when something is wrong by mistake).

  That usually means CS majors plugging stuff in and ECE majors advising them.

3. Get hardware parts

- Every group need various parts. Come grab them! For starters, you need:
    - Breadboard
    - Microbit breakout adapter
    - Red LED
    - 1 kΩ resistor (Color code: Brown, Black, Red)

- Come grab more stuff as you need it.

- Be gentle with the stuff please!
    - Especially the Microbit adapter!

- If you need to complete the hardware parts of the lab after the lab session ends, you can borrow the stuff for the week. We have boxes that you can borrow to hold the stuff in if you ask us for one!

4. Power an LED

- Connect the Microbit to your breadboard

  The Microbit breakout should go in column “j” of the breadboard, rows 1 and down. The Microbit plugs into the breakout with the LED Matrix side up.

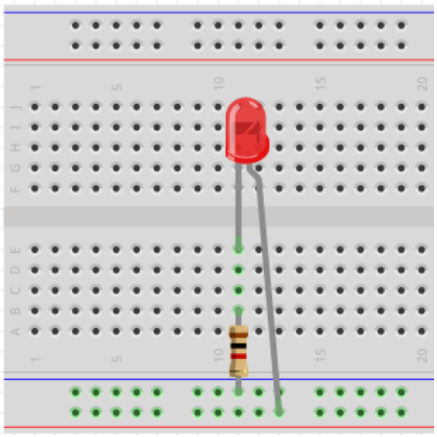
- Connect breadboard VCC and Ground rails to the Microbit power supply

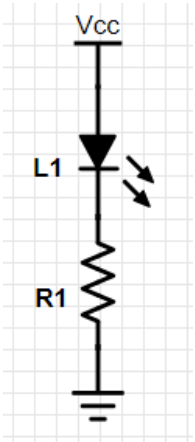
  Use a jumper wire to connect Ground (either of the white “GND” pins) to the Ground rail on the breadboard (the blue “-” column). Use a second jumper wire to connect VCC (the pin labeled 3V3, for 3.3 volts) to the VCC rail on the breadboard (the red “+” column).

- Power an LED

  WARNING: Always ensure that there is a resistor between your LED and Ground!!! If you do not do this, you are creating a short-circuit, which will possibly damage your LED or even your Microbit.

  - Connect a 1 kΩ resistor from the Ground rail to a row on the breadboard. (Color code: Brown, Black, Red)
  - Connect the short leg of the LED to the row with that resistor (the side of the LED with the shorter leg is also flattened on the plastic LED diffuser).
  - Connect the long leg of the LED to the VCC rail on the breadboard.
  - The end result should look like this and the LED should light up! If not, double-check that the Microbit is powered and that your circuit is correct.





- **Checkoff**: demonstrate that the LED works

5. Control the RGB LED

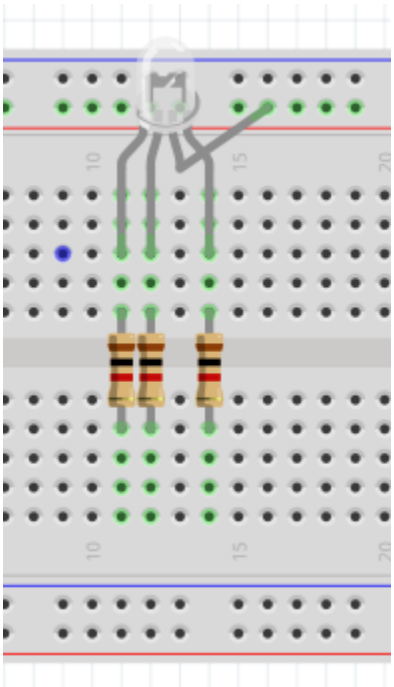
- Remove the existing LED

  That was just a test to make sure that power is working and you can plug components into the breadboard properly. Everything from here on out will be relevant for the final checkoff.

- Wire the RGB LED to digital outputs

  See Page 3 of the LED datasheet for details on its pins.

  - The longest pin should connect directly to the VCC rail.
  - The shorter pins should each connect to separate rows in the breadboard.
  - Connected to each row should be a resistor (1 kΩ or 10 kΩ are fine). The other side of the resistor should connect to the row for the digital pin that controls that color channel of the LED. Use pins 13-15, as specified for each color in `main.c`
  - When you are finished, the layout should look something like this, but with the resistors connecting to the same rows as the Microbit breakout pins 13-15 respectively.



- Configure the pins and write to them to control the LED

  Use the nRF GPIO library to configure and set the pins.

  As you've done in previous labs, each pin needs to be configured as a digital output. There are `#defines` at the top of `main.c` that you can use as the GPIO pin number. They are renames of pins from `boards/microbit_v2/microbit_v2.h`

  Based on how you wired the LED channels, they will be active low. Clearing the GPIO pin should light up the respective color of the RGB LED.

- Test that you can control each color channel of the LED.

  A common mistake is that a GPIO pin controls a different color channel than expected. To overcome this, you can either re-wire the circuit, or simply change the `#defines` at the top of `main.c` to swap pins.

  If it doesn't work, make sure that the correct LED pin is connected to the VCC rail.

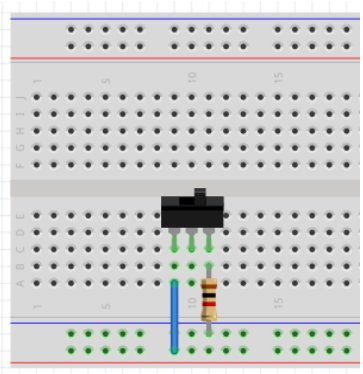
- **No checkoff**: continue to the next step

6. Read a switch

- Wire up the switch

  We'll connect a slide switch to the Microbit as a digital input.

  - The switch should be installed in the breadboard so that each pin is in a different row. It takes a little bit of force to push into the breadboard. Don't worry about it.
  - Connect one edge pin of the slide switch to the VCC rail with a jumper wire.
  - Connect the other edge pin of the slide switch to the Ground rail with a 1 kΩ resistor. **IMPORTANT:** use a resistor to connect to ground. Our switches seem to be crappy and are briefly shorting their outer pins when switching, causing the Microbit to reset if no resistor is used.
  - Connect the middle pin of the slide switch to pin 16 of the Microbit breakout with a jumper wire.
  - The slide switch on the breadboard should look like this, except with the middle pin connected to the Microbit breakout with a jumper wire.



- Configure and read in a digital input

  The pin will need to be configured as an input. No pulling resistor is needed, since the input signal will always be either VCC or Ground. Afterwards, reading the pin should give you the state of the switch and should change as you slide the switch.

- Test that you can properly read in the state of the switch

  If it doesn't work, check that you've connected to the edge pins correctly. The three pins of the switch occupy three consecutive rows when installed.

- **No checkoff**: continue to the next step

7. Read an analog input

- Configure and read the ADC

  You'll be using the nRF SAADC library. Setup code has been provided for you in `main.c`

  - adc_init() configures the ADC to read in 12-bit samples. It also initializes each channel with the proper pin and with a range of 0 to 3.6 volts. (It connects the internal 0.6 Volt reference and applies a gain of 1/6 to the signal.)

  - `adc_sample_blocking()` allows you to read an analog sample. Samples are read in as a 16-bit signed number, but will typically range from 0-4095. (Occasionally you'll get a small negative number, for example -6. I don't know why or how that is working…)

- Convert ADC counts to volts

  The units of each sample in `adc_sample_blocking()` right now are "ADC counts". You will need to convert those into volts in order to properly interpret the data. The 4096 values represent voltages from 0 to 3.6 volts.

- Test your conversion code by connecting the analog input to VCC and Ground

  Briefly connect Microbit breakout pin 2 to the VCC rail or the Ground rail. Sample the ADC voltage for channel 1 and print it out. It should be either 3.3 volts or 0 volts respectively. Try the other rail as well to make sure things are working.

  If you aren't reading what you expect, make sure you're connecting to breakout pin 2 and reading channel 1. Then double check your conversion code. You can also print out the raw ADC counts, which should be either about 3700 and 0 with attached to the VCC and Ground rails respectively.

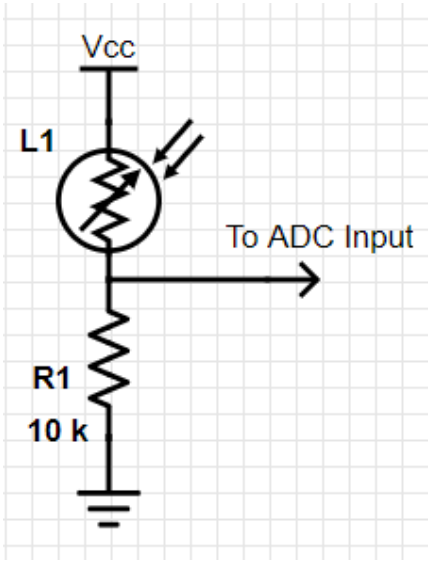
- **Checkoff**: demonstrate that you can read in analog values for VCC and Ground
  - *Question:* how did you do the conversion from counts into volts?

8. Read the light sensor

- Wire up the light sensor in a resistor divider

  Form a resistor divider with a 10 kΩ resistor (Color code: Brown, Black, Orange).

  - Connect one end of the light sensor to the VCC rail on the breadboard. Doesn't matter which end, the component is reversible.
  - Connect the other end of the light sensor to a row on the breadboard.
  - Connect the 10 kΩ resistor from that same row to the Ground rail.
  - Connect a jumper wire from that same row to pin 2 on the Microbit breakout.
  - The schematic for what you are connecting looks like this:



- Create a helper function to convert measured voltage into light states

  Unfortunately, an exact formula to convert between resistance and lux seems to be difficult to calibrate. Instead, you should determine voltage thresholds to distinguish between at least three states: dark, medium-bright, and very-bright. You'll have to do so manually with a little experimentation. You can decide how to define each state.

- Test that you can distinguish between states on the light sensor
- **No checkoff**: continue to the next step

9. Read the temperature sensor

- Wire up the temperature sensor in a resistor divider

  The schematic is the same as with the light sensor. Pairing the temperature sensor with a 10 kΩ resistor for the lower half of the divider seems reasonable. Connect this to pin 1 on the breakout (since pin 2 is in use for the light sensor).

  The temperature sensor looks like this:

- Create a helper function to convert measured voltage into temperature

  This will take two steps: converting voltage to resistance (based on your resistor divider) and then converting resistance to temperature.

  - Convert voltage to resistance. You can use a traditional voltage divider equation to do so. Details on resistive voltage dividers.

  - Convert resistance to temperature.

    We can use a conversion equation for an NTC thermistor.

$$T = \frac{B}{\ln(R/r_\infty)}. \quad \text{where } r_\infty = R_0 e^{-B/T_0}.$$

    For our thermistor:

    B = 3470.0
    R = measured resistance
    R_0= 10000.0
    T_0= 298.15
    T = resulting temperature in degrees Kelvin

You can use the `log()` and `exp()` functions from `math.h` to calculate the natural log and exponential respectively. Make sure all the numbers you are using are floats so the math doesn't truncate. Include a decimal point in each to be sure. Be sure to convert back into Celcius. Note that R_∞ could be calculated in advance if you prefer.

- Test that you can accurately measure temperature.

  Room temperature is usually around 25 C. You can hold the thermistor between your fingers to increase the temperature.

- **No checkoff**: continue to the next step

10. Control the RGB LED with various inputs

- The red channel of the RGB LED should be controlled by the light sensor

  The microcontroller should read the analog value of the light sensor, and should use that value to determine whether the LED should be on or off. You can determine whatever light state you want for switching the LED on and off, as long as it is straightforward to demonstrate switching between them.

  You can sample and change LED states at whatever rate you want (once per second, ten times per second, etc.).

- The green channel of the RGB LED should be controlled by the temperature sensor

  Same as the light sensor. You can choose whatever temperature value you want for switching. Be careful to choose something that's easy to demonstrate with the heat from your fingers. It's okay if it doesn't change back to the original state right away.

- The blue channel of the RGB LED should be controlled by a switch

  The switch should be read by the microcontroller as a digital input. Then based on that input the LED state should be changed. The switch does not need to immediately change the LED's state. Sampling it at whatever rate you sample the light and temperature sensors is sufficient.

- Once per second, print out the measurements from each sensor as well as the state of the switch

  Temperature measurements should be printed in units of and degrees C. Light states should be printed as human-understandable strings (something like "Bright", "Dark", etc.)

- **Checkoff**: demonstrate this application and your code to course staff

11. Cleanup

- Take apart your breadboard and put everything back into the proper locations they came from. Please and thank you!
  - If you didn't finish during the lab session, instead return stuff during next week's lab session. When you return it, we'll give you this checkoff.

- **Checkoff:** return your cleaned up parts, breadboard, and Microbit breakout adapter

Part 2: Capacitive Touch

1. Find the app starter files for this lab

- `cd software/apps/capacitive_touch/`
    - This lab will use the files in this directory. Your changes will be in `capacitive_touch.c` and `main.c`

2. Determine Rise Time

- Code is already written to trigger a GPIO interrupt when the pin goes high

  We're using the NRFX_GPIOTE Library to control GPIO interrupts. The function `start_capacitive_test()` is doing all the work to make that happen. Take a look through it and make sure you understand what's going on.

- Code is already written to initialize a Timer and start it from 0 when performing a capacitive test

  We're using the NRFX_TIMER Library to control the Timer. It's initialized in `capacitive_touch_init()` and cleared at the start of each test.

- Add code to the `gpio_handler()` to determine the COUNTER value from the Timer when the GPIO interrupt fires. Use this to print out the time that has passed since the start of the capacitive test in microseconds.

    - You'll need to do a capture into a CC[n] register, where n is referred to as the CC channel. Which channel you choose doesn't matter.
    - Warning: make sure you use the right function to actually perform a capture.

- Try it a few times without and with a finger touching it to get a feel for the rise time.

    - Hitting the reset button will let you trigger it multiple times.
    - When you have a finger on it, the time will be greatly increased. Also your finger may cause the voltage to fluctuate and trigger multiple interrupts. This is fine.
    - When touching it, the capacitance might be so high that the signal actually *never* goes high until you stop touching. That's fine! Clearly that's *longer* than with no finger present, and distinguishing between touching or not is the goal.

- **Checkoff:** demonstrate a rise time measurement for course staff
    - *Question:* how much longer is the rise time when a finger is touching?

3. Implement a Timeout

- Using the Timer, implement a timeout for the capacitive test.
    - You'll want to trigger the interrupt when more than ~10x the typical rise time has passed. The exact number you choose should depend on your platform's typical rise time.
    - The interrupt should be triggered using the compare register. Set it in `capacitive_touch_init()` using one of the library functions. You can use any channel for doing the compare operation, but it must be different from the channel you were using for the capture.
- When either interrupt occurs, disable both interrupts.
    - Otherwise, a timeout could occur after a GPIO interrupt, or vice versa.
    - You'll probably want to write a helper function to do this that can be called from both functions.
    - Stopping the timer is sufficient to prevent the interrupt from occurring.
- Add a `printf()` to the callback handlers to determine which one is occurring during a test. Test it under multiple scenarios to make sure that the correct callback is occurring.
- **Checkoff:** demonstrate that you can determine if the logo is being touched or not on reset of the system

4. Periodically run capacitive tests

- Create an App Timer to periodically run capacitive tests.
    - Add this to `capacitive_touch.c`
    - The period of your timer MUST be greater than the timeout, or else it won't work right. The period should be short enough that the system feels responsive on human timescales.
    - When the timer fires, run `start_capacitive_test()`
- Write to the `touch_active` variable based on the result of each test.

  This enables requests to the driver to be served with whatever the most recent result of a test was, which is already written in `capacitive_touch_is_active()`.
- Add code to the loop in `main()` to periodically check if the user is touching the logo and print the result.
- **Checkoff:** demonstrate that your app continuously determines whether the user is touching the logo or not
    - Also demonstrate the code you wrote to get that working