ENCM 369 PIC Activity 4

ENCM 369 ILS Winter 2021 Version 1.0

Activity Objectives

- Understand CMOS input requirements
- Design and build a button-input to add to the embedded system from Activity 3
- Modify the 6-bit counter to increment based on user input from the button
- Use trial, error, and investigation to gain a strong understanding of processor clock speed and comparatively slow human input
- Practice following coding requirements and code review of a peer

Deliverables

- One image file that includes a picture or screenshot of your circuit diagram beside the picture of your program flow charts
- Check-in of you code to Github
- Completed rubric review of someone's code in your learning community

Background Information

Getting digital input to a microcontroller is an interesting task. From a hardware perspective, you must create a circuit that can provide either Vdd or Vss to a pin that you have configured as an input. The little trick is that you must DRIVE these voltages on the pin. In other words, you must make a connection to Vdd or Vss either directly or through a "pull-up" or "pull-down" resistor. You cannot simply apply Vdd and then take it away and expect the pin voltage to drop to 0.

An unconnected input pin "floats" and can end up floating high or low depending on different circumstances. Most microcontrollers use CMOS logic, and while these circuits tend to float high, you would never count on this. The amount of charge that will parasitically accumulate on a floating input is very small, and simply touching the circuit will change it. So, if you ever observe strange input behaviour in a digital system especially if you physically touch the circuit and something changes, checking for floating pins is a good start.

In this activity, the input will be on RB5. It is expected that you already know what register you need to set up to make RB5 an input. You should also already know what register you will have to read to get the input signal. You have glimpsed the concept of "bit-masking" to pick out individual bits in a register. To check your understanding, make sure you know the answers to the following questions:

1. What register and what value must you write to this register to make RB5 an input?
2. What bitmask value will you use to access bit RB5?
3. What bitwise logical operation will you use with your bitmask value and the input register to test if RB5 is set? You have to think like a microcontroller, which means that

every question you ask needs a yes or no / true or false / 1 or 0 answer. Don't be fooled by other bits in the register.

Don't proceed without knowing these answers. If you know the answer and someone asks you for help, respond with a question to help the person find the information or make sense of the information that they already have. At this point, you have all the information you need to answer these questions. This is fundamentally important to understand.

To make sure you are bit masking correctly in firmware, you will also tie RB4 to Vdd so it is always high.

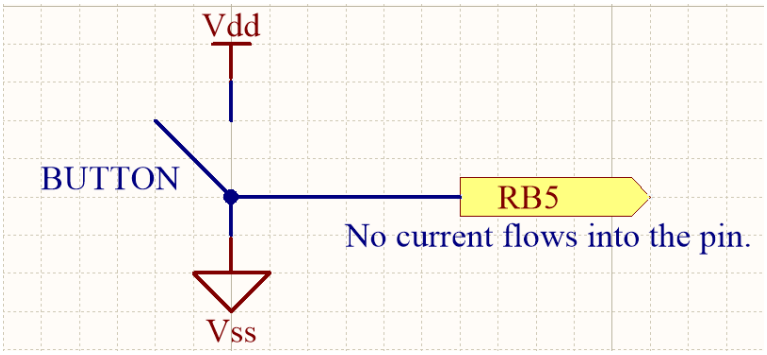
Hardware Design

Using one of the buttons from your ILS hardware kit, design a circuit that will have Vss on RB5 when the button is not pressed, and Vdd on RB5 when the button is pressed. Is this an active low or active high circuit?

Hint 1: The only new piece of information you need is this: **It is very important to know that a GPIO pin configured as an input looks like a very high impedance to anything connected to the pin.** In other words, no current will flow into the pin, the pin just "looks" at the voltage but does not load it in any appreciable way. As far as your circuit is concerned, there is no pin connection.

Remember that all a button does is make or break a single connection. Think of it as a wire that you cut or connect back together. If you don't know how the 4 pins on the button are connected, use the datasheet link in your hardware kit documentation to go find the circuit diagram for the part. If you think that there are two switches inside the button because it has 4 pins, please read the datasheet again. Keep reading it until you are convinced there is only one switch in there.

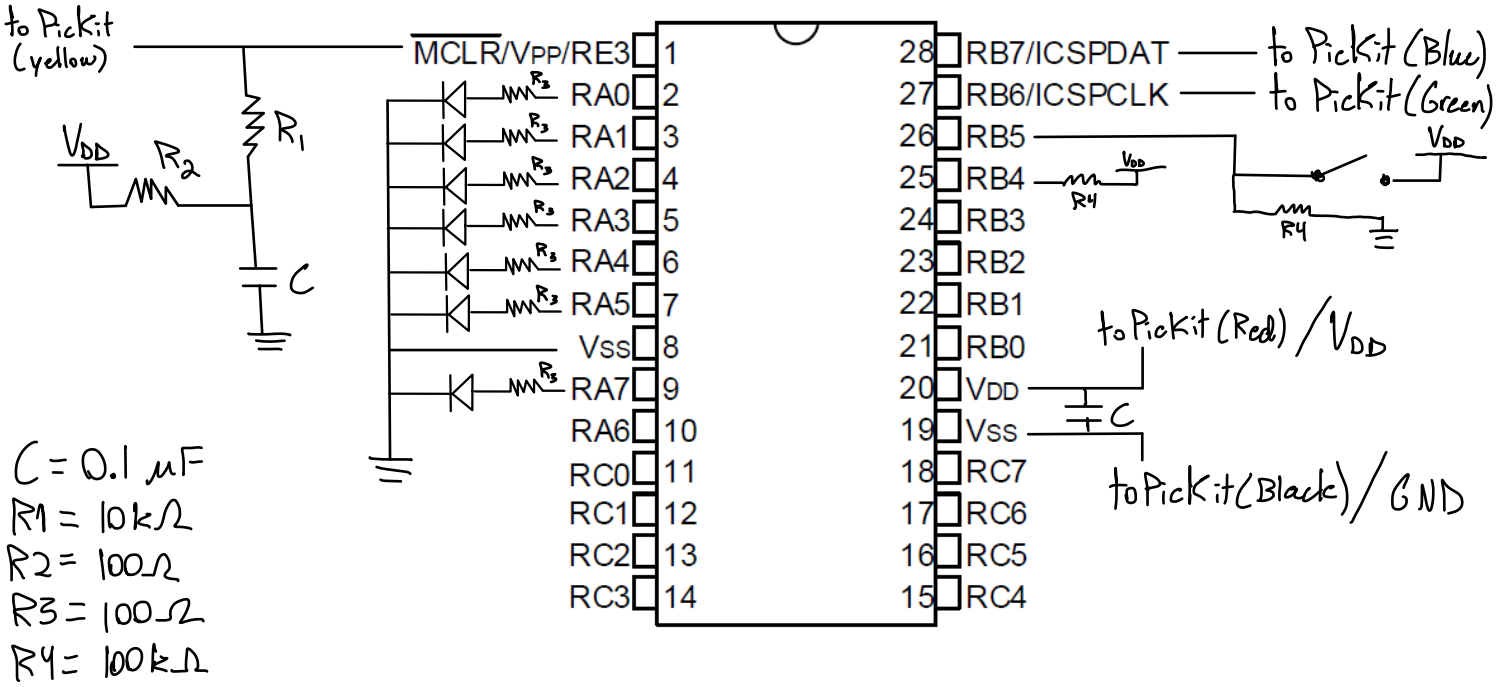
Hint 2: the circuit below is a good starting point but is **NOT correct** and **<u>will DAMAGE</u>** your breadboard and power supply if the button is pressed. Why?



Hint 3: How will you use a 100k "pull-down" resistor to make this circuit safe?
Hint 4: See Hint 1 again.

Once you are confident you have a button circuit that will work correctly, draw in the circuit components on the image below to create your circuit schematic. Don't forget to **also tie RB4 to Vdd so it is always high.** Do this by connecting RB4 to Vdd with a 100k resistor.

You may start with the circuit diagram from PIC Activity3 and just add in the new hardware. **This image is one of the deliverables for this activity**, so be neat. Label the parts with values so someone looking at your schematic could build the circuit. Don't forget power and ground connections, and PIC Kit connections (just indicate "to PIC Kit").



Hardware Implementation

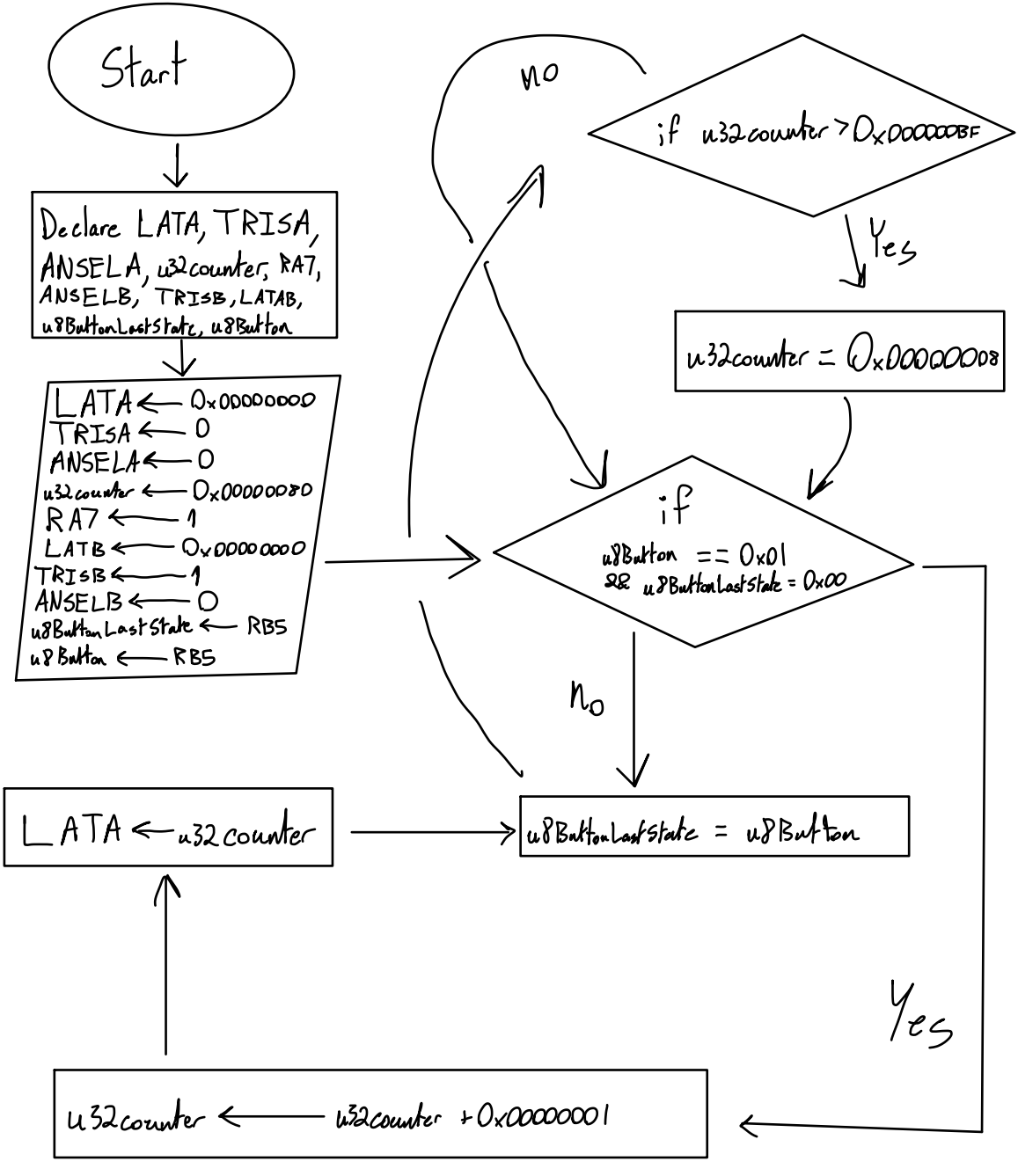
Add the new hardware to your breadboard circuit. Use Scopy to probe RB5 and make sure that the signal indeed toggles between Vss and Vdd when you press the button. It's fine for the PIC to be running the previous program during this step as long as you have not written any code to change port B to outputs. Why?

Program Design Part 1

It is extremely important to design programs before you ever start typing. Complicated problems must be broken down into manageable steps and functions. The code for this activity is not long or complicated in implementation, but if you don't design it and code to that design, you will lose a ton of value from the exercise and will fill the discussion board with posts to which the replies will all be, "look at your design and figure out why your code isn't working."

*****Note the firmware steps in this activity will purposely fail in various ways as you work through them because you need to understand what is happening in the system*****

PIC Activity 4 Design Flowchart



The program is operationally simple to understand: every time the button is pressed, the counter should increment by one. Take a moment to visualize how you would expect this system to work when you use it. Draw the flow chart for the firmware. Remember that UserAppRun() is the only place where you should be writing code for the main program, and the system must return to the main loop continuously to allow other functions in the system to run, sleep, and repeat (even though we are not yet running other functions or sleeping but UserAppRun() doesn't know that).

This design is the second deliverable for this activity.

Programming Part 1

From this point on, you are always expected to follow the ENCM369 coding conventions for PIC Activities and any project code you write. You will not be reminded again.

Branch PIC_Activity3 to a new branch called PIC_Activity4. Make sure PIC_Activity4 code is the active branch. Before you write any new code, delete the code that adds the delay between counter increments. You should not have any delay code any more. If you reset your counter in Activity 3, delete this, also. The 6 LSBs will continue to counter accurately even if the higher bits are accumulating. You will have to make sure you only write the 6 LSBs to LATA when you update the counter on the LEDs. RA7 must stay on!

Run the code now to make sure it still (sort of) works. What do you expect to see on the LEDs when you run the code without the delay? If it's not what you expect, explain it.

Next, implement your design from the previous section making sure you are following the coding conventions. You will again use GpioSetup() for the register initialization and this is still the only place in your code that you are allowed to turn on RA7. There should be no code that worries about the button state during GpioSetup(). Use UserAppRun() for the rest of the code.

Please use a 32bit variable for the counter even though you will only be displaying the 6 LSBs.

Don't forget that you need to declare the counter variable "static" so it doesn't reset every time UserAppRun() is called. This concept will likely be useful for other parts of this activity.

Run your code to test the operation of the circuit. Try pushing the button just once, then halt the code and look at the counter variable in the debugger. You should set this up in a watch window so you get some more debugger experience. What happens when you press the button? What value is in your counter? Why?

Program Design Part 2

This part assumes you did not anticipate how fast the processor runs and thus you are getting many increments every time you press the button. Even if you anticipated this, you probably need to improve your program so that only one increment occurs with each button press. In other words, you need to detect a TRANSITION from low to high instead of the steady state

value. In other words, if you read the input register value to look at the button state and it is logic low, then the next time you look at it it's logic high, you know a transition has occurred because the button was pressed. How do you know when the button is released?

Update your program design to ensure that only a change of the button state from low to high will trigger the counter to increment. What do you have to do to get the next increment?

Programming Part 2

Code the updated solution and make sure it works properly. One or two strategically placed breakpoints should really help you verify the code is working properly. If it's not working, talk yourself through the code. Remember the PIC is processing 16 millions instructions per second (it's safe enough to assume that's close to 16 million lines of your code per second).

There is a very good chance that you might get a few increments even with a single button press. This is because the mechanical action of pressing or releasing a button can cause multiple contacts to occur very rapidly (like within a few milliseconds). That's fast, but not to the PIC that is processing in microseconds. Try putting an oscilloscope probe on RB5 and triggering the scope on the first rising edge. Set the time scale to get about 30ms of data and you should be able to capture a good image of the problem. This phenomena is called "contact bouncing" and proper firmware would handle this. Think about how you would write code to "debounce" the hardware. In the interest of time, you are NOT required to code this, but you might want to know the answer because it's a good job interview question.

What to Hand In / Review

Take a pic of your hardware, and a screen shot of your two firmware design flowcharts. Commit your code and push it to Github. Perform a code review on someone's code from your learning community who you have not already done a code review for. Upload the two pictures and the code review spreadsheet to the D2L Dropbox for the assignment.

Further Learning

OPTIONAL. This is NOT part of the required activity.

If you're bored during reading break and want to do some more:

- Write a debounce routine to fix the contact bounce problem. A good debounce time is 10ms and with this program it would be fine to add that delay right in UserAppRun() so the processor waits the 10ms inside the function (not ideal as you are "blocking" any other function from running, but we don't know how to avoid this yet).
- add another button that allows you to decrement the counter
- add code to increment (or decrement) the counter continuously if the button is held. Again, think about how most systems like this work – you would hold the button which would cause the first increment, then after a short delay the counter would then start incrementing. That delay is critically important to make it usable and surprisingly difficult to code. Even if you don't write the code for this, draw out the flow chart. Or you might need to elevate your thinking a little and consider a state diagram instead.