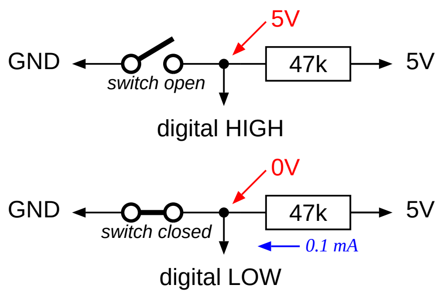
Microcontroller Systems and Interfacing

week 11 experimental lab

1 Digital input

We cannot simply connect a switch to a digital input pin. Just like with analogue inputs, we have to *generate* a voltage that the digital input can recognise as `HIGH` or `LOW`. The usual way to do this is with a *pull-up resistor* and a switch:



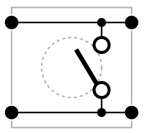
When the switch is *open* (not conducting) the *resistor* connects the digital input to 5V and reading the input will produce `HIGH`. The digital input has a very high resistance, and so almost no current will flow.

When the switch is *closed* (conducting) it connects the digital input to GND and reading the input will produce `LOW`. In this case, the resistor limits the amount of current that will flow from 5V to GND.

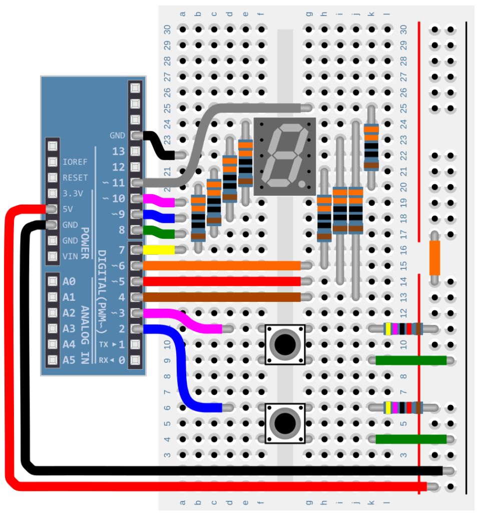
Typical values for the pull-up resistor are between 10k Ω and 100k Ω. For example, if we were to use use 47k Ω, a standard value in the middle of the range, then we can use Ohm's law to calculate that $5\,\text{V}/47000\,\Omega \approx 0.1\,\text{mA}$ will flow from 5V to GND when the switch is closed.

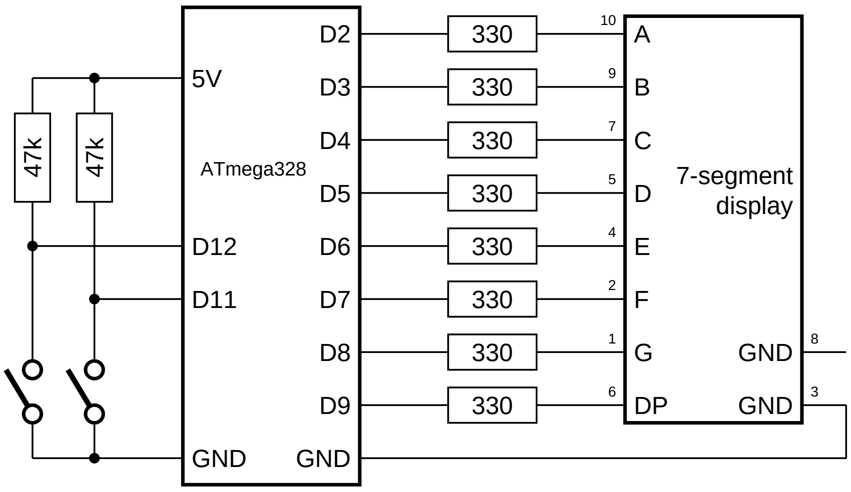
Let's add two switches to our circuit and use them to control an up/down counter. The current value of the counter (a single digit) will be displayed on the seven-segment display. (If you prefer, you can make a single 'dot' move up and down a LED bar graph display instead.)

► Add two switches to a circuit and make them do something.



Extend your seven-segment (or LED bar graph) display circuit with two switches, including a pull-up resistor for each. Connect your switches to digital pins 2 and 3 of the microcontroller and configure those pins as `INPUT`s. (If you have to move your seven-segment display connections to different pins, remember to update your program accordingly.)





The simplest way to make the switches affect a counter is to continuously increment (or decrement) the counter while one (or other) of the buttons is pressed. To recognise when a button is pressed, check if the corresponding digital input is `LOW`. Make one of the buttons increment the counter and the other button decrement the counter.

You should avoid asking for a digit larger than `9`. One way to do this is to take the value of the counter modulo 10, using the '`%`' operator (which is like division, except that it gives you the remainder instead of the quotient).

```
int counter = 15000;
void loop() {
  if (digitalRead(3) == LOW)
    counter = counter + 1;
  displayDigit(counter % 10);
  delay(100);
}
```

► Make the counter change just once each time a switch is pressed.

To make the switches affect a counter once you want to recognise a *transition* from the switch being open to the switch being closed. This means the digital input changing from a `HIGH` level to a `LOW` level.

Hint: start with just one of the pins. When that one works, duplicate the code and modify it to manage the other switch.

```
int oldState = HIGH;
void loop() {
  int newState = digitalRead(3);
  if (oldState == HIGH && newState == LOW) {
    counter = counter + 1;
    displayDigit(counter % 10);
  }
}
```

You will need a variable outside of `loop()` to remember the `oldState` of the switch. Each time `loop()` runs, read the `newState` of the switch. If the `oldState` was `HIGH` and the `newState` is `LOW` then the switch must just have been pressed. If the switch was pressed, update `counter` accordingly and display its new value using `displayDigit()`. In all cases you should store the newState of the switch into `oldState` at the end of `loop()`.

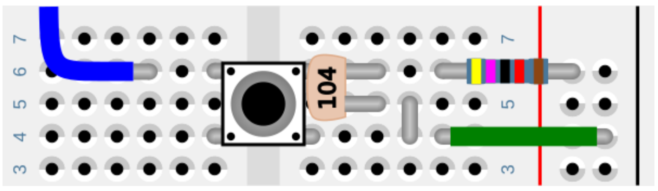
Hint: when it is time to duplicate the code for the other switch, think very carefully about which variables should have two versions (one version for the 'up' switch and the other version for the 'down' switch) to prevent the switches from interfering with each other.

? Do you notice any strange behaviour when you press the switches? If you do, think about what is happening *inside* the switch. Hint: I deliberately chose cheap switches so that their construction would be the simplest it possibly could be.

2 Hardware Debouncing

Your switch is *bouncing*.

▶ Cure the switch bounce using a small capacitor.



The capacitor will discharge instantly when the switch is closed, but will need several milliseconds to charge up again when the switch is opened. While charging, the value on the digital pin will remain `LOW` until the capacitor is almost fully charged at which time the pin will change to `HIGH`.

Begin with the largest value of capacitor you have, about 100 nF. (It is marked '`104`'.) Then try successively smaller values until the switch starts to bounce again. (If you have very small capacitors then use a small 1-row wire link to connect one lead of the capacitor to ground, without stressing it by bending the leads, as shown in the diagram.)

3 Software debouncing

One reason microcontrollers are so useful is that they let us 'move' hardware into software. Try to debounce the switch in *software*, instead of in hardware.

A simple approach to debouncing is to introduce a short delay, to allow the switch to finish bouncing, before letting the program proceed.

▶ Remove the debouncing capacitor(s) from your circuit.

▶ Modify your program to implement the following debouncing strategy:

- if the input pin is `HIGH` then do nothing (the switch is not pressed)
- otherwise, the input pin is `LOW`:
  - perform the associated action (increment or decrement the counter),
  - wait for a short time (using the `delay()` function) to let the switch 'settle',

```
if (digitalRead(3) == LOW) {
  counter = counter + 1;
  delay(100);
}
```

Increase the delay until the switch no longer bounces when you press it.

? Does the switch still sometimes bounce when you release it? Why?

The contacts sometimes 'bounce back' when the switch is released, causing unwanted 'phantom presses' of the switch. One way to cure this is to wait for the switch to become `HIGH` again, then delay again, before continuing. The strategy now looks like this:

- if the input pin is `HIGH` then do nothing (the switch is not pressed)
- otherwise, the input pin is `LOW`:
  - perform the associated action (increment or decrement the counter),
  - wait for a short time (using the `delay()` function) to let the switch 'settle',
  - use a `while` loop to wait for the input pin to return to `HIGH`.
  - wait for a short time (using the `delay()` function) to let the switch 'settle',

```
if (digitalRead(3) == LOW) {
  counter = counter + 1;
  delay(100);
  while (digitalRead(3) == LOW)
    ; // do nothing
  delay(100);
}
```

? What are the disadvantages of this debouncing mechanism?

? What happens if you try to operate both switches rapidly, at the same time?

4 Challenges

4.1 Eliminate the external pull-up resistors

Pull-up resistors are so useful that the microcontroller provides them internally. To enable an *internal* pull-up resistor, configure the pin as `INPUT_PULLUP`. Once configured like this you can remove the two external 47 kΩ pull-up resistors.

4.2 Debouncing with a timer: simulating hardware debouncing in software

Software debouncing, as described above, is not ideal. A slightly better solution is to simulate the capacitor circuit in software, using a timer. To implement a timer we can use the `millis()` function, which returns the number of milliseconds for which the program has been running.

By remembering the millisecond time at which each switch was last pressed, it is easy to simulate a capacitor 'discharging' and 'charging'. If the program is written correctly, there is no need for a dedicated `while` loop and both switches can be operated simultaneously.

You will need two variables, of type `long`, to hold the 'last-pressed' time for each button. Your `loop()` function can simulate a capacitor discharging and recharging for each button, for example:

- if the button input is `HIGH`, do nothing
- if the button input is `LOW`, the button is pressed (or bouncing during a press or release)
  - if the button's 'last-pressed' time is more than T milliseconds ago
    - the button has been stably release for a while, so...
    - perform the action (increment or decrement the counter)
  - set the button's 'last-pressed' time to the current time
    - this simulates the capacitor being completely discharged at this instant in time

▶ Modify your program to debounce using a timer.

Experiment with different values of T until you find a *completely reliable* value. Demonstrate that while one button is held down the other button continues to work normally.

4.3 Use interrupts to manage the buttons

If our `loop()` has a repetitive task to perform then trying to handle switches and other asynchronous input at the same time can be inconvenient. Just like all computers, the microcontroller provides interrupts for handling asynchronous events (such as an input pin changing state) outside the normal cycle of repetitive tasks. We can use interupts associated with digital input pins 2 and 3 to process our switch inputs asynchronously.

▶ Prepare your circuit and program to use interrupts.

If you have not already done so, move the seven-segment display to pins 4 through 11 and move the two button inputs to pins 2 and 3. Verify that your program still works.

▶ Make your `counter` variable 'interrupt-safe'.

Declare `counter` it to be `volatile`. (Declaring it `volatile` warns the compiler that its value might change without warning, during an interrupt.)

▶ Attach the buttons to the interrupts.

Write an *interrupt service routine* (interrupt handler) called `buttonDownISR()` that decrements the counter. Modify your `setup()` function so that it runs `buttonDownISR()` whenever the associated button is pressed. For example, if pin 2 is connected to the 'down' button:

```
attachInterrupt(digitalPinToInterrupt(2), buttonDownISR, FALLING);
```

Write a similar handler called `buttonUpISR()` that increments the counter and then in `setup()` associate it with the other button by calling `attachInterrupt()` with the appropriate parameters.

▶ Modify your program so that the `loop()` function only does one thing: displays the current value of your counter on the seven-segment display.

▶ Test your program to make sure it works.

4.4 Debounce the buttons in your interrupt-based program

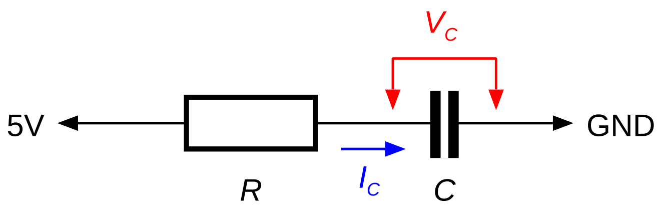
The interrupt-driven version again suffers from bouncing. Cure this by implementing the same `millis()`-based filter in your interrupt routines.

Your switches may still bounce on release. Cure this using a periodic `TimerOne` interrupt that resets the last-pressed time for each button every 10 ms as long as the corresponding input pin is `LOW`.

Microcontroller Systems and Interfacing

week 11 reference material

A Capacitors

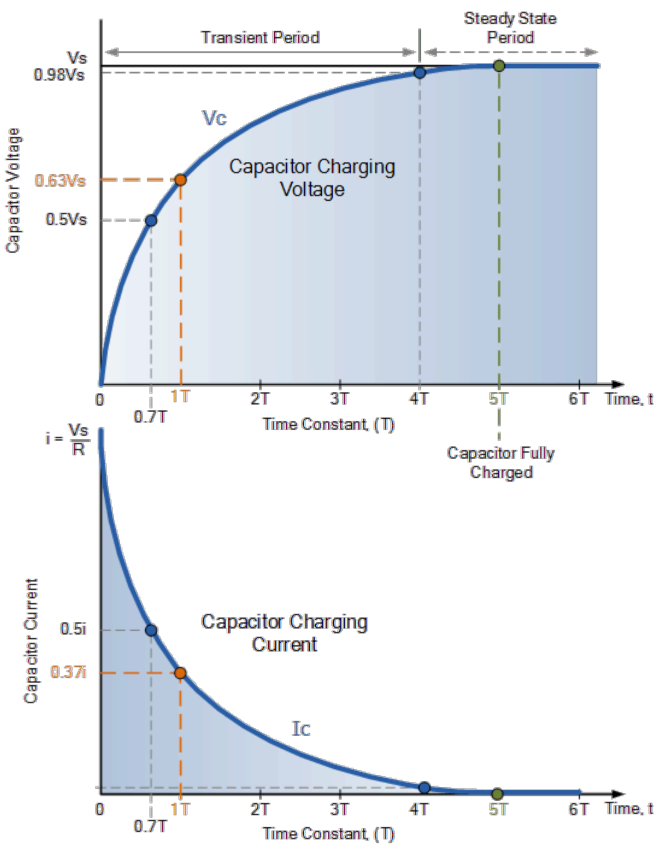


A capacitor is a two-terminal device that stores electrical charge. When it is 'empty' it has a very low resistance to current, and allows a large current to flow into it. As it 'fills up' the resistance increases, and the current decreases. Eventually the capacitor is full and the current flowing into it decreases to zero, effectively giving it an infinitely large resistance.

When placed in a circuit with a resistor, the capacitor behaves like a low-value resistor that gradually increases in resistance as charge flows into it. Considering the two components as a voltage divider, this means the voltage across the capacitor begins at zero ($R \gg R_C$) and rises until it equals the supply voltage ($R \ll R_C$).

The current flowing into the capacitor, and hence the rate at which it accumulates charge and increases its voltage, is proportional to the voltage across its terminals. Therefore the rate of charging decreases as the amount of charge increases. This leads to an initial rapid increase in voltage that gradually slows as more and more charge is accumulated (and the voltage in the device increases, and the current flowing into it decreases).

The *time constant* of a resistor-capacitor series circuit, τ (Greek letter 'tau'), is equal to the product of the resistor and capacitor values. The time constant indicates how long it will take for the capacitor to charge to 63% of its capacity (and hence to 63% of the supply voltage). Multiplying τ by various values tells us useful information about the time taken to reach various conditions:



$R \times C \times \ldots$	*condition reached*
$\tau \times 0.7$	50% of final voltage
$\tau \times 1.0$	63% of final voltage
$\tau \times 4.0$	98% of final voltage

Examples of capacitor applications include timing (for oscillators, timers, switch debouncing circuits), blocking DC voltage while allowing AC signals to pass (in series connection), and stabilising voltage against short-term fluctuations (in parallel connection).

A.1 Capacitor values and markings

Capacitance is measured in *Farads* (after the physicist Michael Faraday). Typical capacitor values fall in the *picofarad* (1 pF = 10^{-12} F), *nanofarad* (1 nF = 10^{-9} F) and *microfarad* (1 μF = 10^{-6} F) ranges.

Capacitors are either marked with their value directly, or (similarly to resistors) are marked with three digits indicating a two-digit value and an exponent giving the value in pF.

Larger values (more than a few μF) are often polarised, and will have the positive and negative terminals marked (and possibly have a positive lead that is longer than the negative, like LEDs). Unlike LEDs, connecting a polarised capacitor the wrong way round is *very damgerous* and can lead to *explosive failure* of the device.

$+ -$

2200 μF
polarised

100 nF
(10×10^4 pF)

marking	*value*	
101	100 pF	100 pF
222	2200 pF	2.2 nF
103	10000 pF	10 nF
104	100000 pF	100 nF