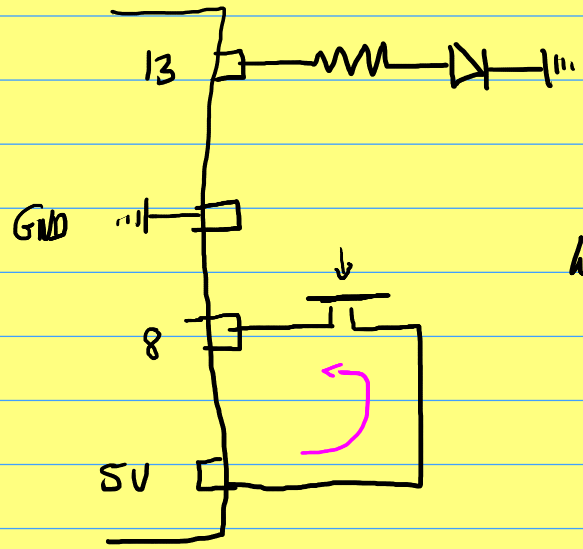
Lecture 3 CS3514 19/9/17

Digital Input

Connecting a Switch to the Arduino.



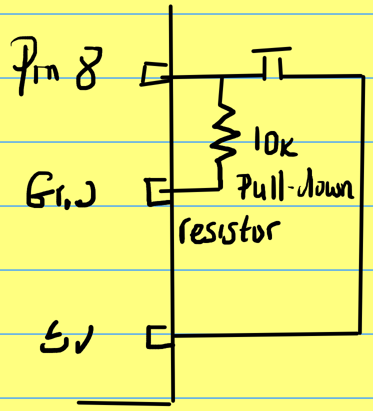
When the Switch is pressed, the Circuit is closed and 5V is 'seen' @ Input Pin 8.

When the Switch is open, the voltage on pin 8 is questionable. Static electricity from the previous closing of the Switch, or from the Surrounding environment, may present a voltage to the pin and this could be read as a non-zero logic level!

We illustrated this by touching the pin with the bare hand.

At times it was enough to bring the hand near to the pin for the pin to register a different Input.

To combat this uncertainty, we introduce a 'Pull-down' resistor. When the switch is open, any charge on the pin will flow through the resistor to ground - leaving 0V on the Pin.



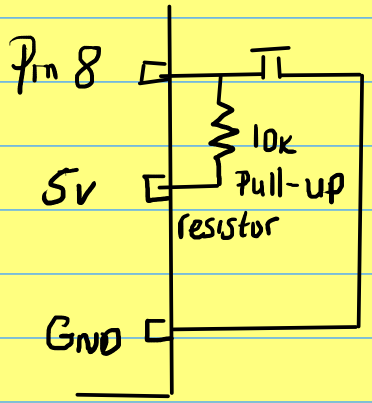
When the switch is closed, very little current flows through the resistor because its value is so large:

$$I = V/R = \frac{5}{10,000} = 0.0005A$$
$$= 500\mu A$$

And 5V is still 'seen' by the Input pin 8.

Pull-up resistors can be used to present a logic 1 to an input pin when a switch is open, closing the switch would then result in the pin

'Seeing' a logic 0.



The input 'resistance' (impedance) of the arduino pins is very high.

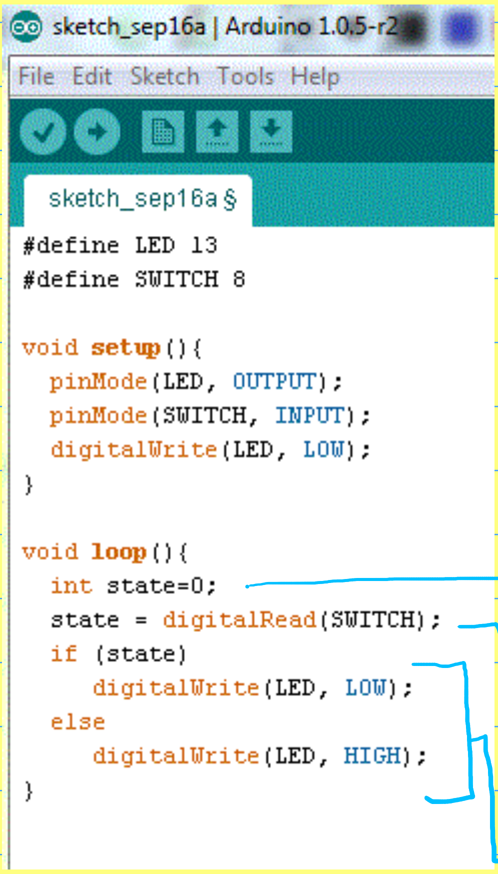
Therefore, the voltage drop across the pull-up resistor is small (in Comparison to the voltage drop at the pin).

As a result, Pin 8 'sees' close to 5V when the switch is open.

When the Switch is closed, all the voltage is dropped across the resistor as the current flows to ground.

Thus, Pin 8, 'sees' 0V (or logic 0), as required, When the switch is closed.

The Code we used to read the value on pin 8 and to respond to this value was:



```
#define LED 13
#define SWITCH 8

void setup(){
  pinMode(LED, OUTPUT);
  pinMode(SWITCH, INPUT);
  digitalWrite(LED, LOW);
}

void loop(){
  int state=0;
  state = digitalRead(SWITCH);
  if (state)
     digitalWrite(LED, LOW);
  else
     digitalWrite(LED, HIGH);
}
```

} name the pins being used

} set as input/output & Initialize o/p pin

→ declare variable to hold state of Input pin (← not needed!)

→ Read value on Pin

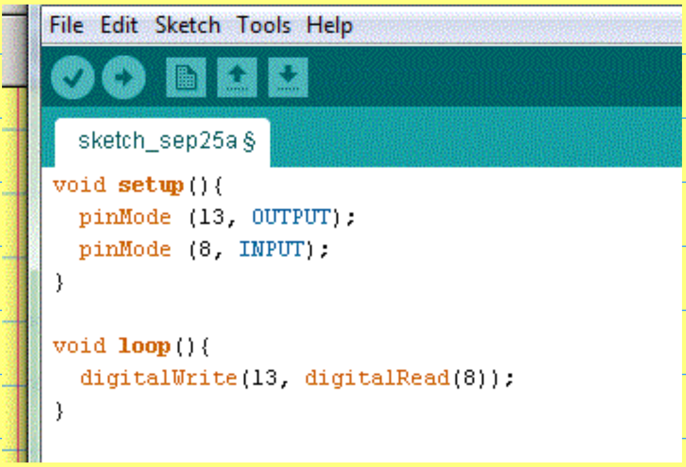
→ Do Something.

Note how we can change the external circuitry from a pull-down to a pull-up and so change the behaviour of the output (default on or default off)

We can also change the behaviour in the code. The choice is ours.

The important thing to ensure is that the input pin is always in a well-defined state and not left to FLOAT

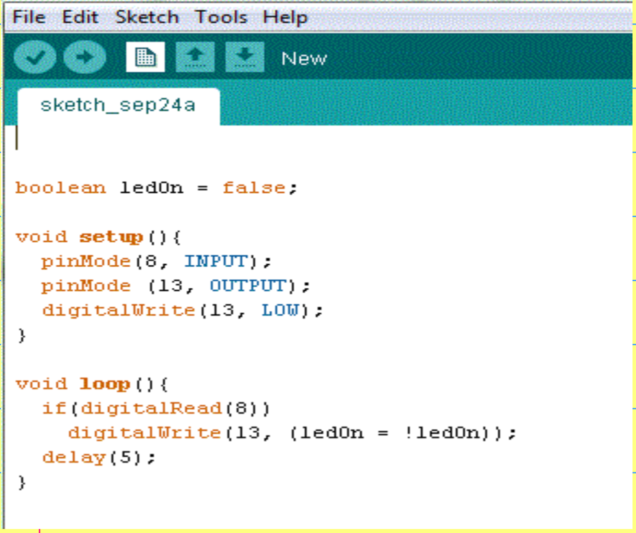
If the code associated with the circuit above simply lights LED 13 when the switch is closed and extinguishes it when the switch is open, all will appear to work well.



```
void setup(){
  pinMode (13, OUTPUT);
  pinMode (8, INPUT);
}

void loop(){
  digitalWrite(13, digitalRead(8));
}
```

However, if we wish to do something more complicated like toggle the LED on and off with each button press then we will discover that sometimes the LED will stay on when it should go off and visa versa.!



```
boolean ledOn = false;

void setup(){
  pinMode(8, INPUT);
  pinMode (13, OUTPUT);
  digitalWrite(13, LOW);
}

void loop(){
  if(digitalRead(8))
    digitalWrite(13, (ledOn = !ledOn));
  delay(5);
}
```

- Consider this code as a possible solution.
- The LED does appear to toggle when the momentary switch is pressed.
- However, it only works sometimes.

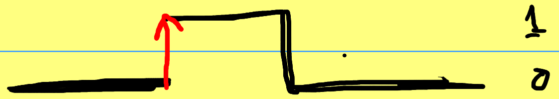
What's happening here?

→ The loop function is continuously being called. at a rate that is much faster than we can press and release the switch. In essence, the LED will toggle if the switch is closed for an odd number of invocations of the loop functions
→ obviously, this is not what we want. !

We could try playing with the delay value, so that we could release the switch before the the loop function is called again. But, again, this approach will not produce predictable results.

Obviously, we need a more sophisticated solution

We will proceed by detecting when the the input goes from 0 → 1



This is known as a positive-edge trigger

Input switch

When we detect this transition, we can toggle the LED.

- We'll find that the LED still won't toggle predictably.

The reason is due to a physical characteristic of the switch (and indeed all mechanical switches).

Contact Bounce

Switches are noisey: when a switch is pressed, metal contacts touch against each other to complete the circuit. This coming together of contacts is rarely 'clean'. Due to the mechanical nature of the switch, the contact may make and break several times in the course of milliseconds before they finally come together to close the circuit.

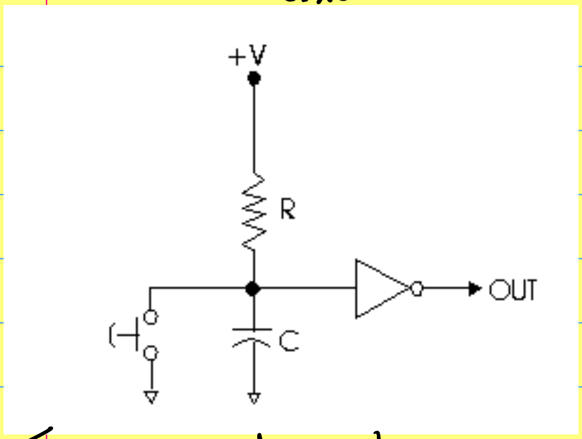
This phenomenon is called "bouncing"

During the bouncing phase the microcontroller may 'see' its input pin change state many times in the same period that the switch is pressed once.

Switch bouncing can be addressed in hardware or in software

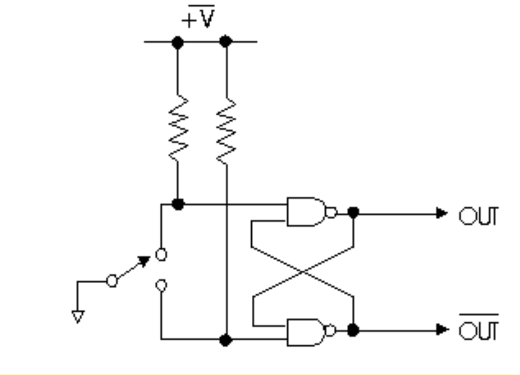
Hardware Solutions:

method 1



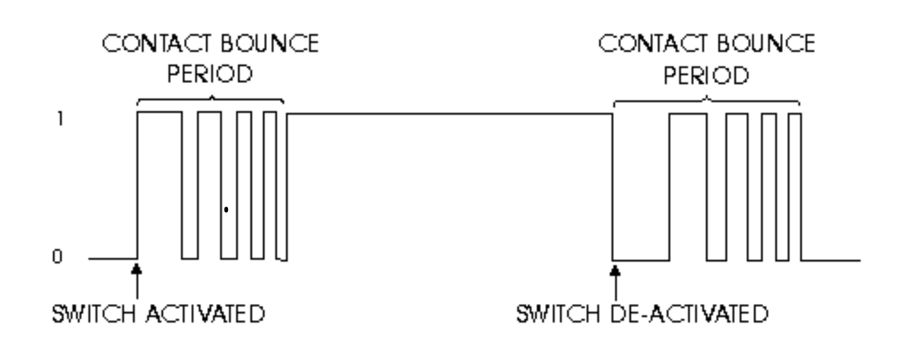
Time needed between switch presses for Capacitor to recharge.

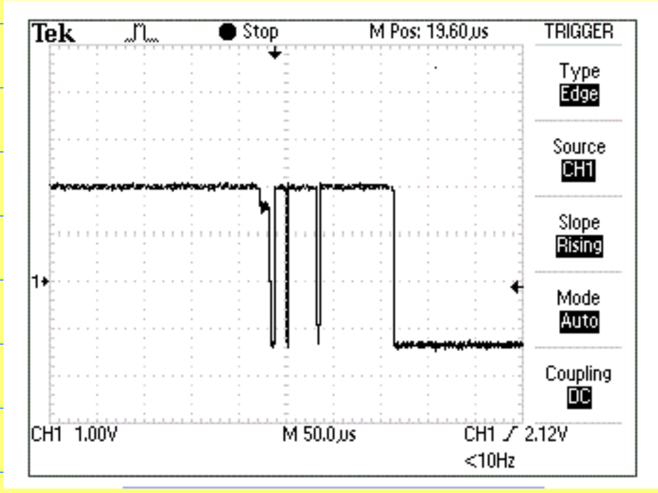
method 2



No recovery delay needed in this arrangement (S-R flip flop)

When the switch is open, C is charged through R. when C is sufficiently charged the o/p of the Inverter is logic Ø. when the switch is closed, C discharges through the switch. when C is sufficiently discharged the o/p of the Inverter flips to logic 1.





● Stop
M Pos: 19.60us
TRIGGER
Type Edge
Source CH1
Slope Rising
Mode Auto
Coupling DC
CH1 1.00V
M 50.0us
CH1 ∕ 2.12V
<10Hz

Software Solution

This Involves writing an algorithm to determine when the input has finished bouncing and to then report an input state transition.

There are many algorithms on the web proporting to solve this problem. Not all work in the general case.
The best solutions use timer interrupts. We will look at these later.

Inferior solutions use delays and make assumptions about timings that may be dependant on the type of switch being used. For our Immediate purposes these solutions are good enough, but it is Important that we recognise their limitations.