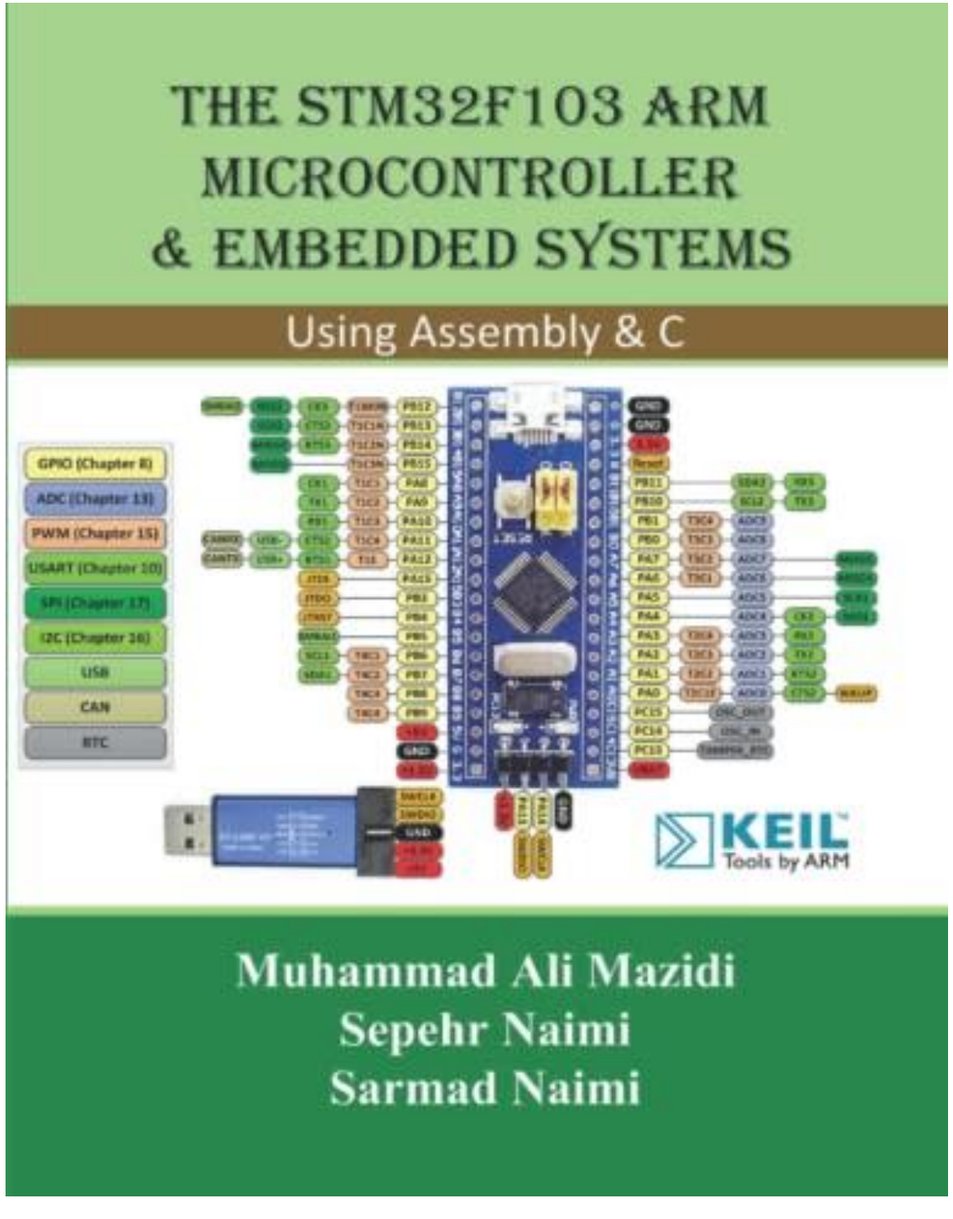


Lecture # 04M:

Switch Debouncing

Course Teacher: **Prof. Dr. Engr. Muhibul Haque Bhuyan**
Professor, Department of EEE
American International University-Bangladesh (AIUB)
Dhaka, Bangladesh

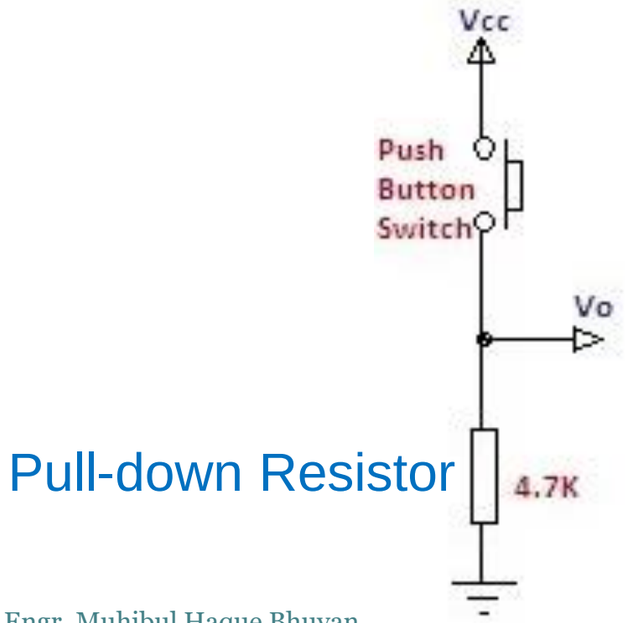
What is Debouncing?

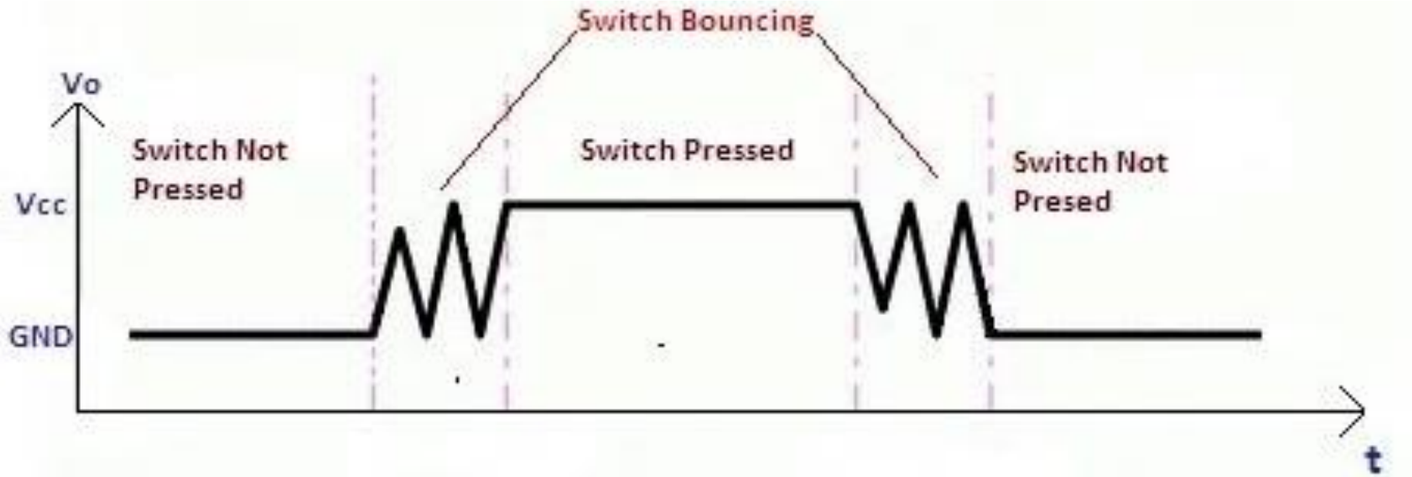
When the state of a mechanical switch is changed from open to closed and vice versa, there is often a short period of time when the input voltage will jump between HIGH and LOW levels a couple of times before it settles at the applied value. These transitions are called **bouncing** and getting rid of the effect of the transitions is called **debouncing**. If you want to input a manual switch signal into a digital circuit, you will need to **debounce the signal,** so a single press doesn't appear like multiple presses.

Alternatively, when a mechanical switch is flipped/pressed, the metal contacts inside may not open or close cleanly. In the microseconds before the switch achieves a **good solid connection**, the switch's contacts may "**bounce**" against each other, turning the **switch on and off** in rapid succession.

Why does Bouncing happen?

- Bouncing is a result of the **physical property of mechanical switches.**
- Switch and relay contacts are usually made of metallic springs so when a switch is pressed, it is essentially two metal parts that come together. This does not happen immediately, the switch bounces between in-contact (close the contact) and not-in-contact (open the contact) until it finally settles down.

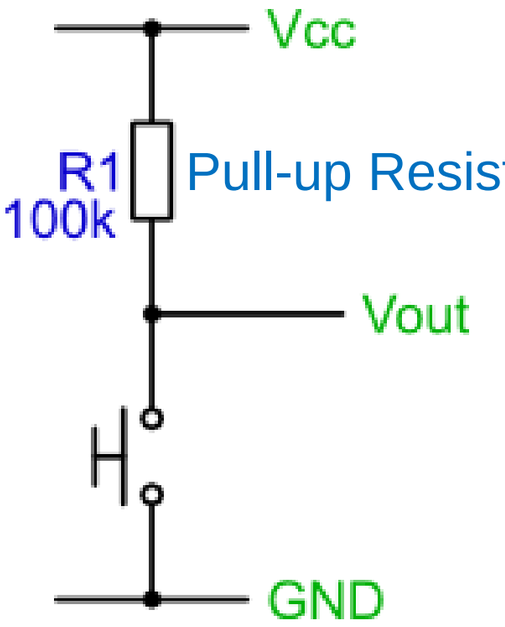


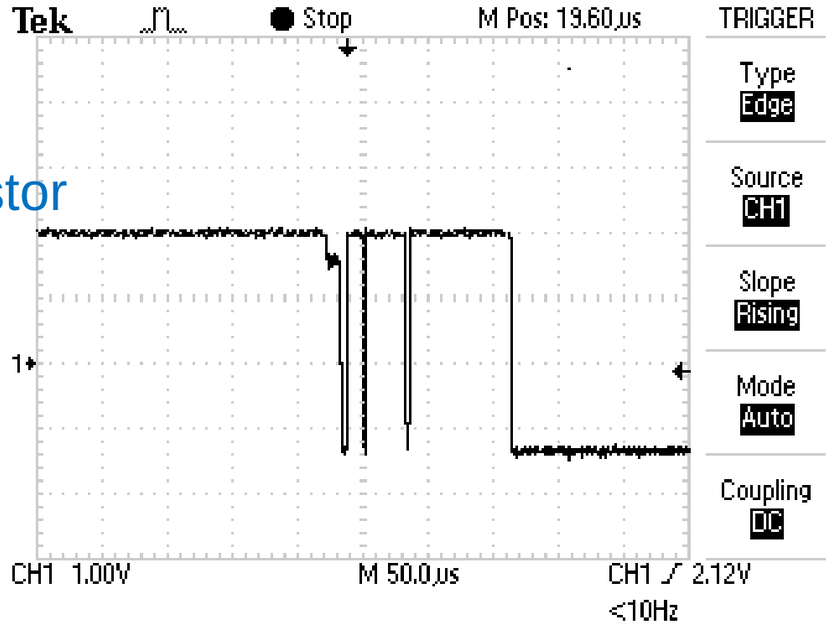


Why does Bouncing happen?

- Figure shows a simple push switch with a pull-up resistor and its right side, the trace shows the output signal, V_{out}, when the switch is pressed. As can be seen, pressing the switch does not provide a clean edge. If this signal is used as an input to a digital counter, for example, you will get multiple counts rather than the expected single count.

- The same can also occur on the release of a switch.
- The problem is that the contacts within the switch don't make contact cleanly, but slightly 'bounce'. The bounce is quite slow, so you can recreate the trace, and the problem quite easily.



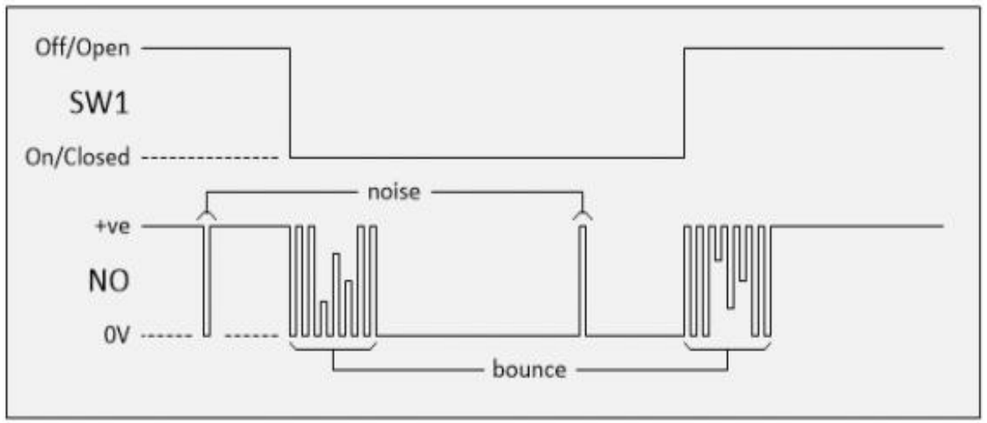


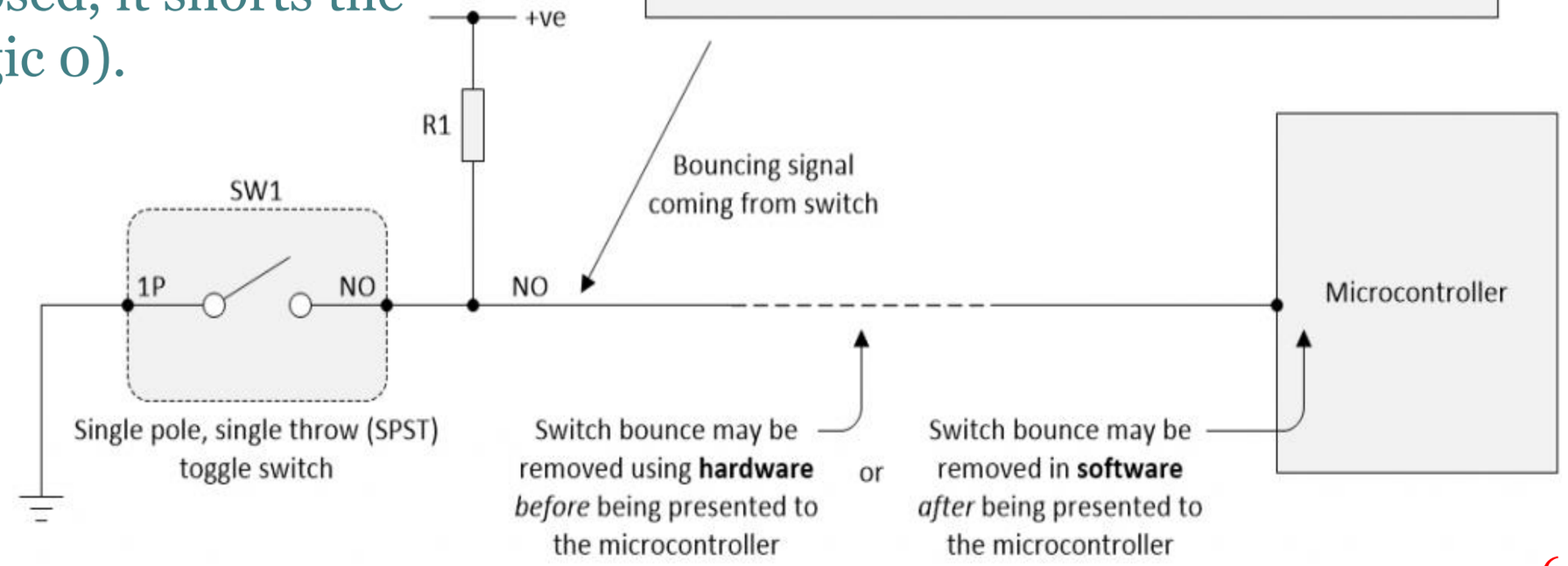
Why does Bouncing happen?

- The bouncing phenomenon is associated with the **Single Pole Single Throw (SPST)**, **Single Pole Double Throw (SPDT)**, and similar other types of toggle switches. The bouncing case data may be analyzed to see how long the switch bounce persists, how wide the individual bounce pulses are, and how many bounces are observed.
- It is essential that the technique is not fooled by the occasional noise "glitch" or "spike" caused by-
  - Crosstalk,
  - EMI (electromagnetic interference)
  - RFI (radio frequency interference)
  - ESD (electrostatic discharge).

How does Bouncing happen?

- In this diagram, resistor R_1 is acting as a pull-up resistor.
  - √When the switch is open, R_1 pulls the NO (Normally Open) contact to a positive (logic 1) value, usually +5 V.
  - √When the switch is closed, it shorts the NO contact to 0 V (logic 0).





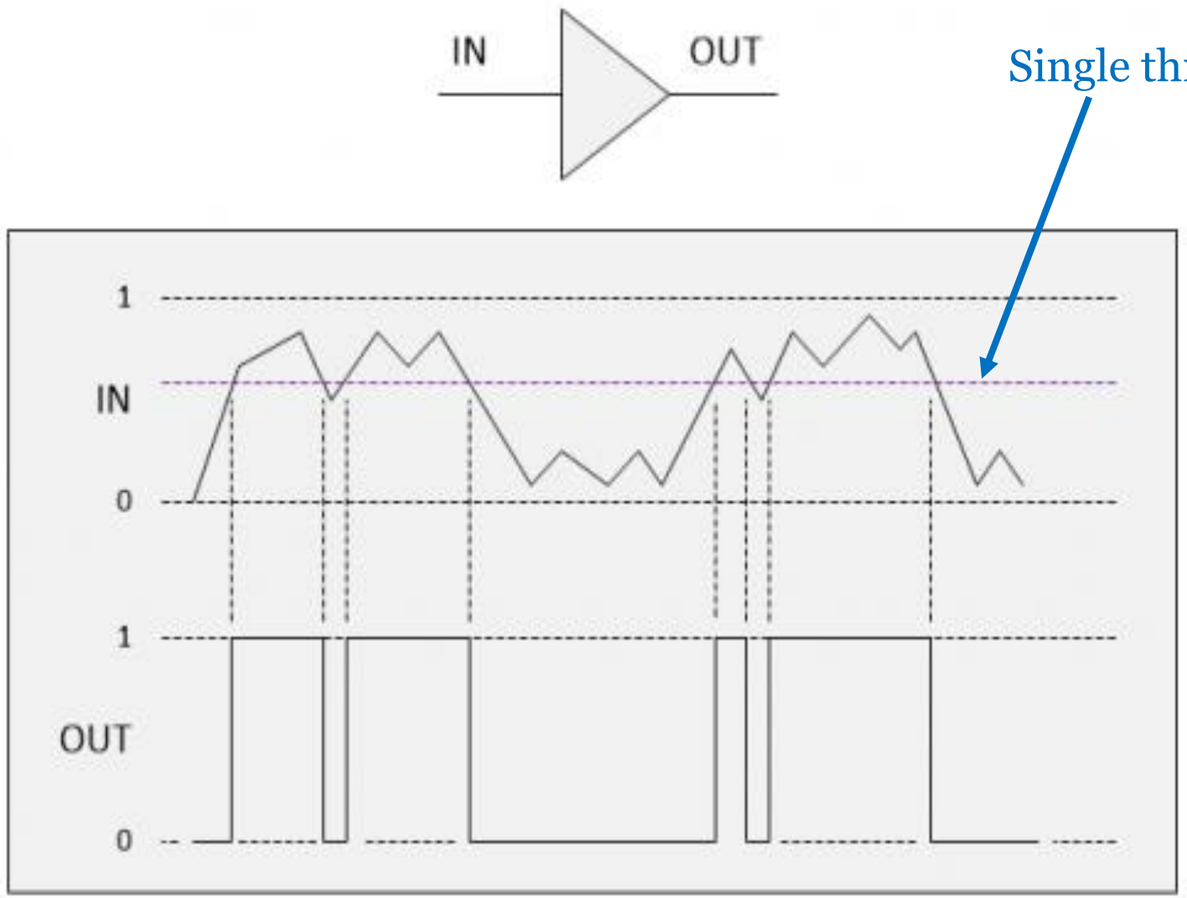
How does Bouncing happen?

- When people push a mechanical switch button, they expect one reaction per push. As the buttons tend to bounce around when pressed and released, this will mess up the signal from them.
- For example, we have a button that is connected between a voltage supply and the output probe. We intend to get an output voltage of 5 V (logic 1) when the switch is pressed, and 0 V (logic 0) when it is not pressed. If we probed the output signal coming from the button during the transition from pushing it down to letting go, we expect an immediate and clean transition $0 \rightarrow 1 \rightarrow 0$. What we end up seeing instead is the picture above. Before the signal settles to a fixed 5 V, it bounces between the two logic states several times.

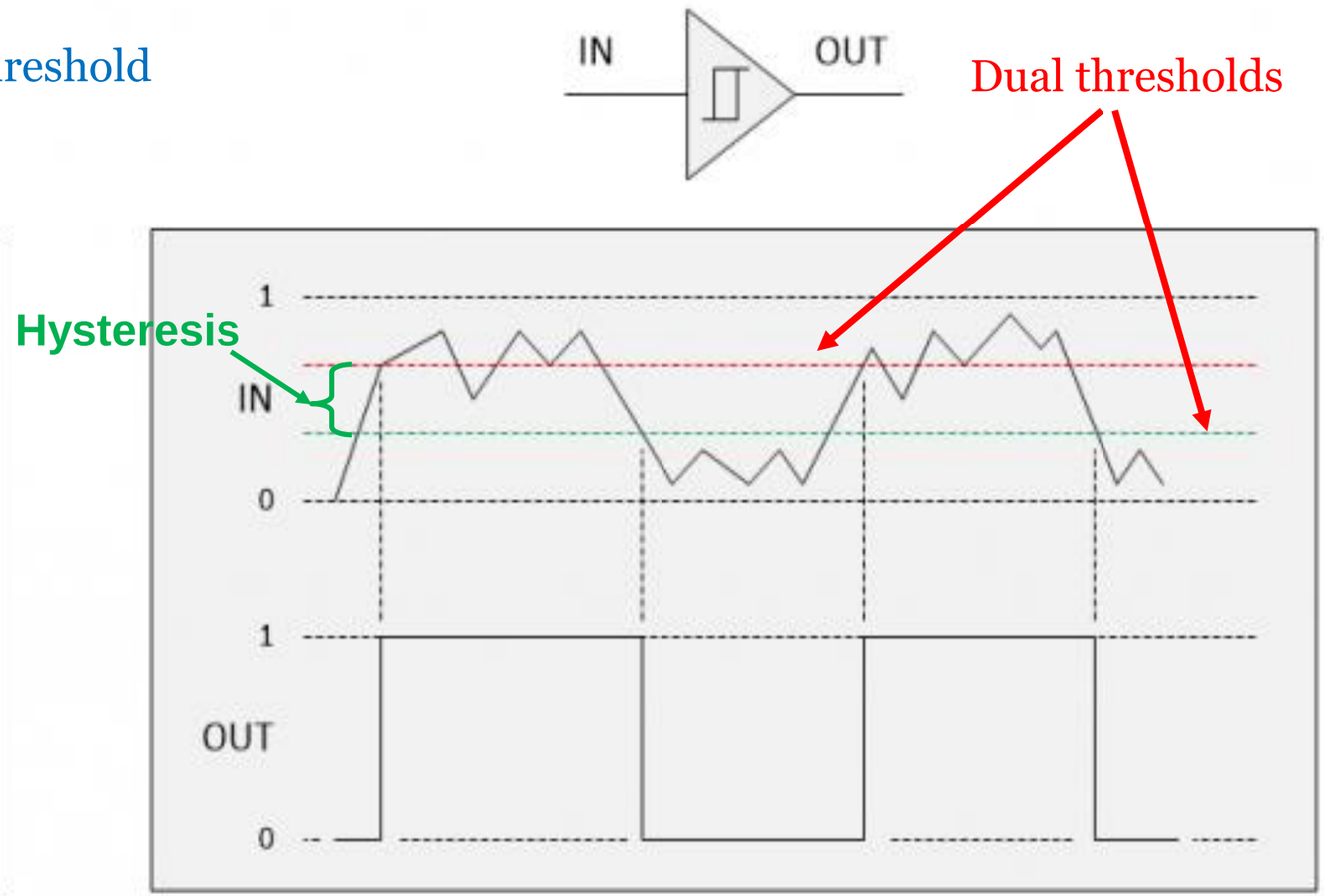
How to avoid Bouncing?

- The digital functions in general and microcontrollers don't like to see input signals wandering around in the **undefined region** between a "good" logic 0 and a "good" logic 1. ,
- In 1934, a young graduate student named **Otto Herbert Schmitt** invented a circuit known as the **Schmitt trigger** (this invention was a result of Otto's study into the propagation of neural impulses in the nerves of squids- a kind of sea fish). The best way to describe this is by means of a figure as illustrated in the next slide.

How to avoid Bouncing?



Standard buffer with single threshold



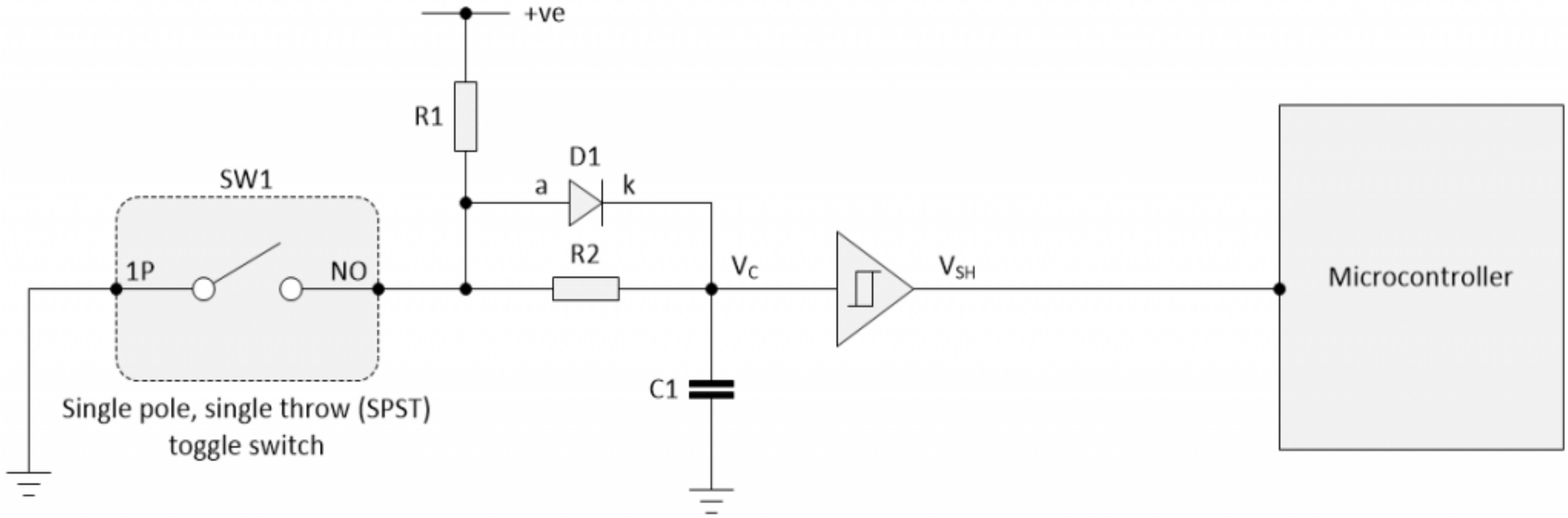
Schmitt trigger buffer with upper and lower thresholds

How to avoid Bouncing?

- A standard buffer has only a **single switching threshold**, which means that a noisy signal including a signal with a bit of ripple on it, can result in multiple transitions at the output.
- In comparison, a **Schmitt trigger buffer** that is, a buffer with a Schmitt trigger input, has **two thresholds**. The output changes only when the input crosses the upper threshold or the lower threshold. This **dual-threshold action** is called **hysteresis** (the dependence of the state of a system on its history), which implies that the trigger possesses **memory**.

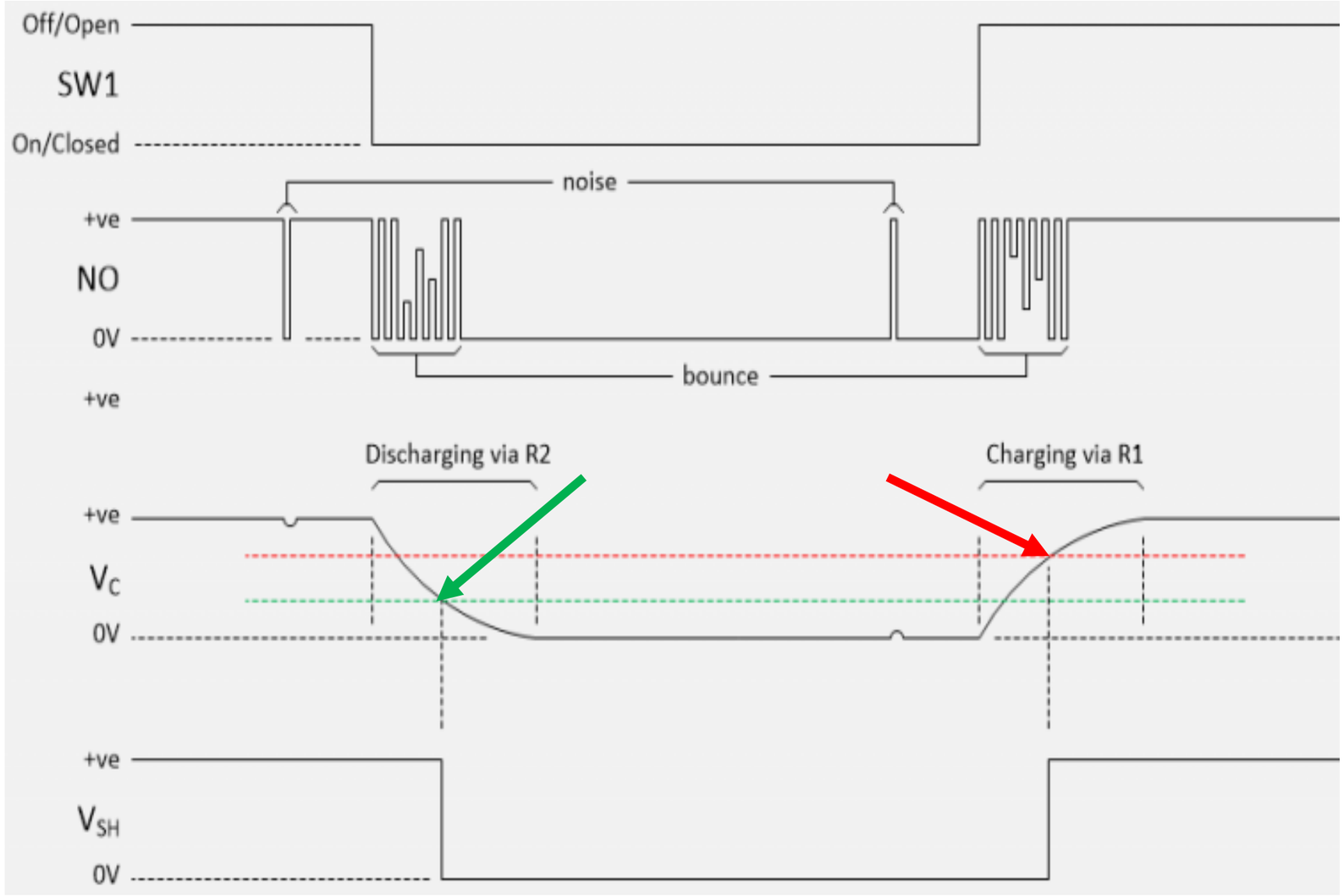
How to avoid Bouncing?

- In the RC network, we use a simple SPST switch debounce circuit, we would add a **Schmitt trigger buffer** between the RC network and the microcontroller as illustrated below.



How to avoid Bouncing?

- We have shown a **non-inverting Schmitt trigger buffer** to keep things simple (i.e., all the signals going up and down in a common direction).
- It is common to use an **inverting Schmitt trigger buffer** because inverting functions are faster than their non-inverting counterparts.



Why we need Debouncing?

- Bouncing often causes problems in the application, where pressing the switch once should theoretically cause only one transition.
- If we connect the switch button to a pin with an external interrupt enabled, we will get several interrupts from pressing the button just once. This behavior is normally **not wanted** and can cause multiple false button presses.
- **Example:** Imagine using a button in a TV remote controller for the selection of a channel. If the button is not being debounced, one press can cause the remote to skip one or more channels.

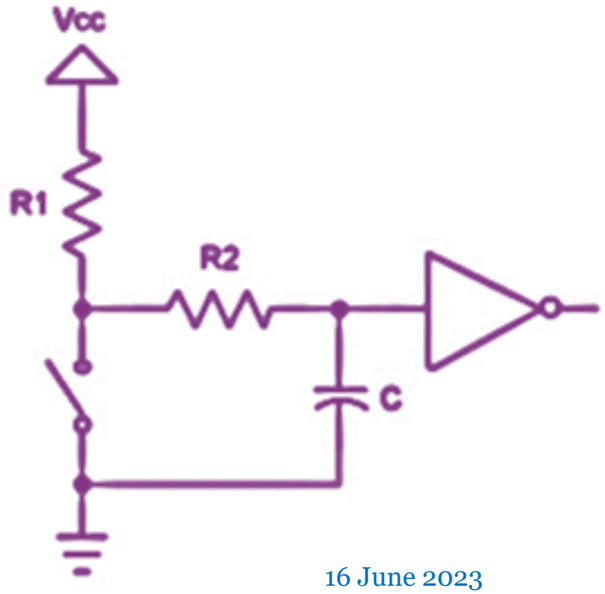
Types of Debouncing

- We can remedy this problem by debouncing the switch. There are many ways to do this. There are **two general ways** of getting rid of the bounces:
  - **Hardware debouncing:** adding circuitry that actually gets rid of the unwanted transitions.
  - **Software debouncing:** adding code/program that causes the application to ignore the bounces.
  - **Digital Switch Debouncing:** achieved in the same way as the software approaches
  - **Switch Debouncer using VHDL:** The VHDL entity description is used.

Hardware Debouncing

- There are various implementations of circuits that can be used for eliminating the effect of switch debouncing right at the hardware level.
- One of the most popular solutions using hardware is to employ a **resistor/capacitor network** with a time constant (τ) longer than the time it takes the bouncing contacts to stop, i.e., $\tau > t_{bounce}$. Often it is also followed by a Schmitt trigger which helps get rid of the capacitor voltage values midway between **HIGH** and **LOW** logic.

- If we wish to **preserve program execution cycles**, it is best to use the **hardware debouncing approach**.



Hardware Debouncing

- In hardware debouncing, we may use a flip-flop to "latch" the signal, using a capacitor to **absorb the bouncing periods**.

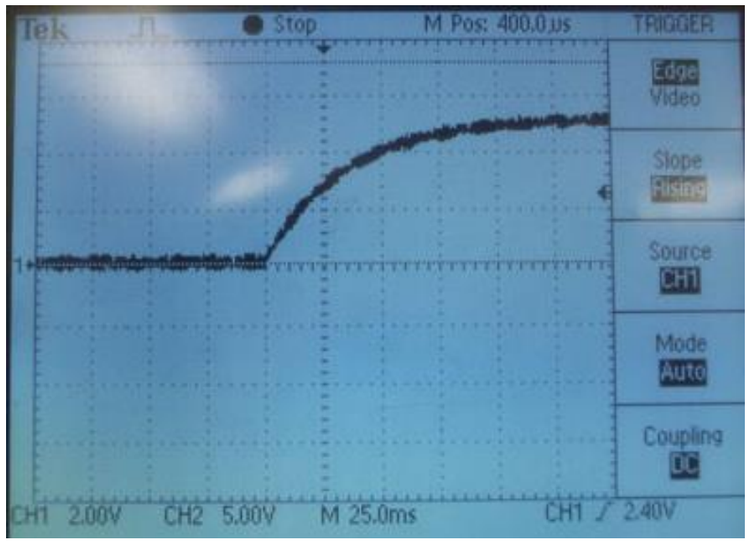
Example Circuit

In the next slide, there is a simple circuit for debouncing a switch for digital logic. When the switch is closed, the voltage is pulled down through the switch, and when the switch is open, the output voltage is pulled HIGH through the resistor. The resistor prevents a short circuit when the switch is closed. The capacitor absorbs the bounces when the switch changes states. Keep in mind that this circuit will cause a short delay in switching because the capacitor takes time to charge through the resistor. The delay can be minimized by varying the values of the capacitor and resistor, but if the resistor is too small, it may not be sufficient to absorb the bounces.

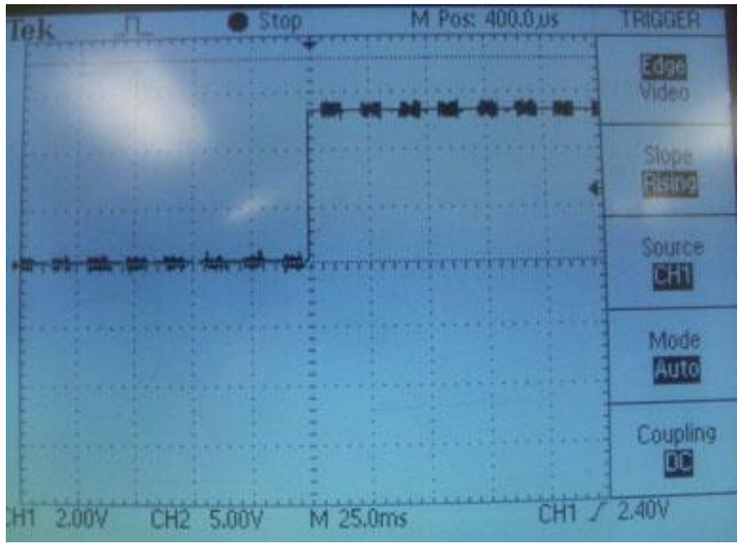
Hardware Debouncing

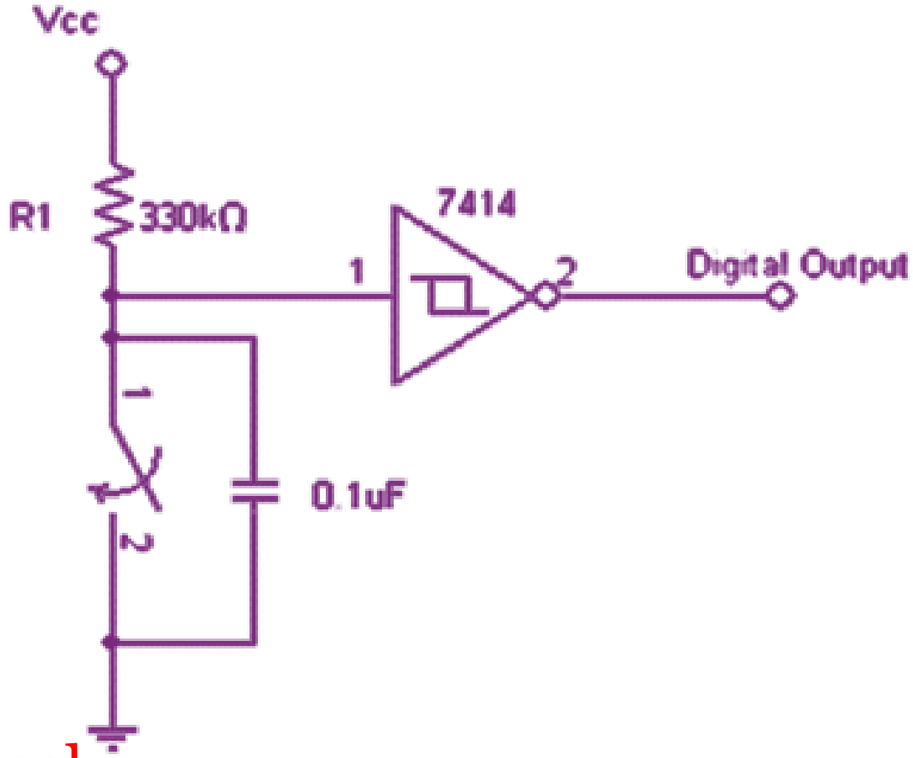
There is one problem: the capacitor exhibits a curve when charging, and parts of this curve will fall in the region where a logic chip is undecided whether to treat it as a HIGH or LOW. To fix this, a **Schmitt trigger** is added to the output.

Without a Schmitt trigger:



With a Schmitt trigger:





Remember that the 7414 Schmitt trigger will invert the signal.

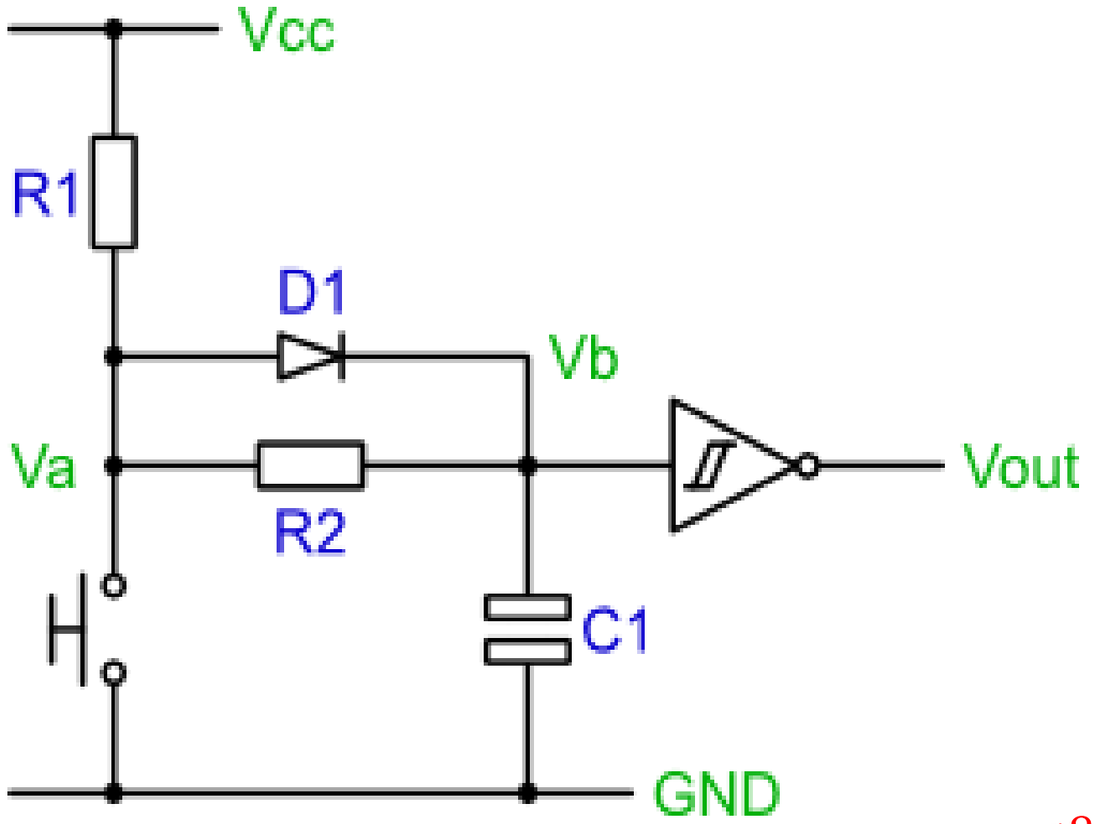
Hardware Debouncing

Circuit Design using the 7414 is a **Schmitt trigger (inverting type buffer)** with the **input hysteresis:**

The equation for discharging a capacitor is given by-

$$v_c = v_{th} = V_{final} e^{-t/RC}$$

V_{final} is the voltage from where the capacitor starts to discharge after the capacitor is completely charged.



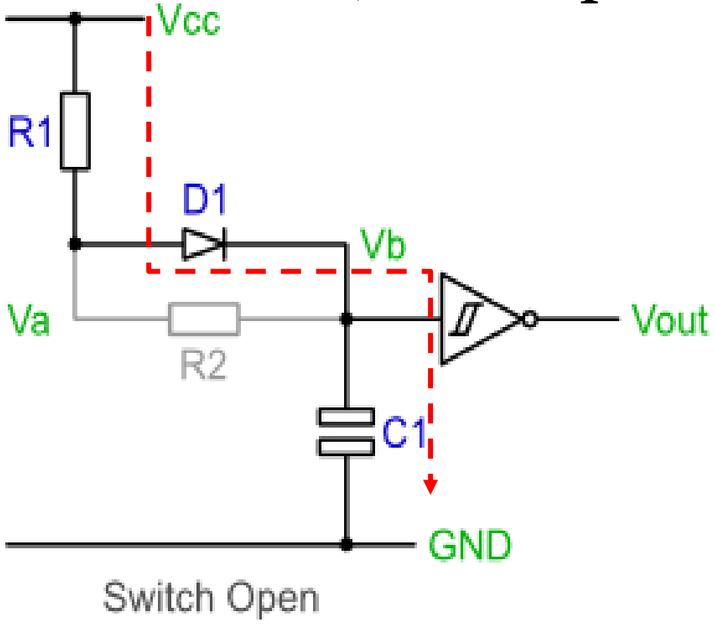
Debouncing circuit design using the 7414 Schmitt trigger (inverting type buffer)

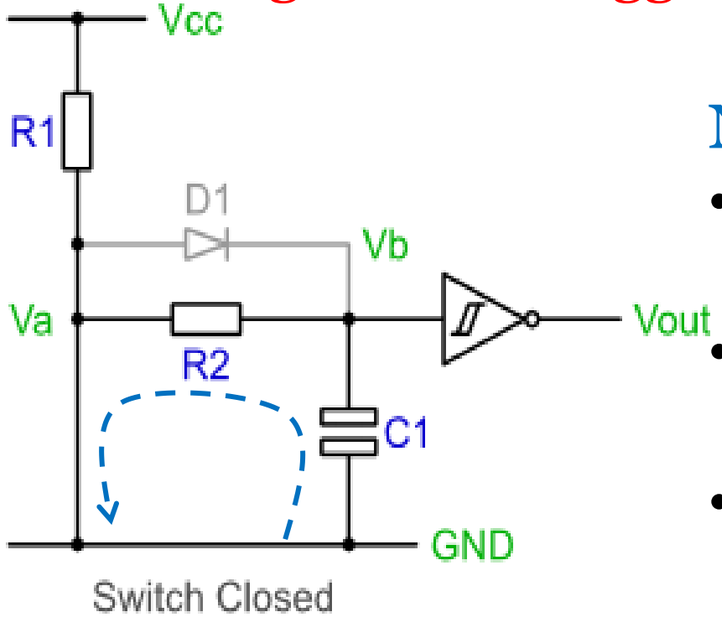
Hardware Debouncing

The circuit's operation can be explained by looking at the equivalent circuits formed in the two switching states, e.g., open and closed.

Starting with the switch **OPEN**:

- The capacitor C_1 will **charge** via R_1 and D_1.
- In time, C_1 will charge and V_b will reach within 0. 7 V of V_{cc}.
- Therefore, the output of the inverting Schmitt trigger will be a logic 0.





Now, **CLOSE** the switch:

- The capacitor will **discharge** via R_2.
- In time, C_1 will discharge and V_b will reach 0 V.
- Therefore, the output of the inverting Schmitt trigger will be a logic 1.

Hardware Debouncing

What about the bounce conditions?

If bounce occurs and there are short periods of switch closure or opening, the capacitor will stop the voltage at V_b immediately reaching V_{cc} or GND. Although bouncing causes slight charging and discharging of the capacitor, the hysteresis of the Schmitt trigger input stops the output from switching.

What about the diode?

The resistor R_2 is required as a discharge path for the capacitor. Without it, C_1 will be shorted when the switch is closed. Without the diode, D_1, both R_1 and R_2 would form the capacitor's charging path when the switch is open. The combination of R_1 and R_2 increases the capacitor's charging time, and thus slows down the circuit. However, when the switch is closed, R_1 is connected across the supply rails to limit the current flow to the circuit.

Hardware Debouncing

Numerical Example:

A CMOS device like the 74AHCT14 dribbles about an mA from the inputs.

Both R_1 and R_2 control the capacitor's charging time, and so set the **debounce period** for the **switch open state**.

The equation for charging is:

$$V_c = V_{final}\left(1 - e^{-t/RC}\right)$$

V_{final} is the voltage towards where the capacitor will reach after the capacitor is completely charged. This will be V_{CC} in this case.

Hardware Debouncing

Numerical Example:

The goal is to select values that ensure the capacitor's voltage stays above the V_{th}, the threshold at which the gate switches till the switch stops bouncing.

Most of the switches exhibit bounce times well under 10 ms, so the use of this value would be a conservative choice. Now increase that by the bounce duty cycle. Based on experimental data, we can expect about a 50% duty cycle, giving us 20 ms of a time period.

Rearranging the **capacitor discharging formula** to solve for R (the cost and size of capacitors vary widely so it is the best option to select a value for C and then compute R) yields:

$$R = -\frac{t}{C \ln \frac{V_{th}}{V_{final}}}$$

$$v_c = v_{th} = V_{final} e^{-t/RC}$$

Hardware Debouncing

Numerical Example:

The AHCT version has a worst-case V_{th} for a signal going low of 1.7 V.

Let us try with a 0.1 µF capacitor, since this is small, cheap, and solve for the conditions in which the switch is closed. The capacitor discharges through R_2. If the power supply is 5 V (i.e., $V_{final} = 5$ V) then $R_2 = \frac{20\times10^{-3}}{0.1\times10^{-6} ln\frac{1.7}{5}} =$ 185 kΩ. Since a resistor with this value is not available, so we use 180 kΩ.

But the analysis ignores the gate's input leakage current. A CMOS device like 74AHCT14 dribbles about 1 µA from the inputs. That 180 kΩ resistor will bias the input up to 0.18 V ($V = IR = 1 \times 10^{-6} \times 180 \times 10^{3} = 0.18$ V), uncomfortably close to the gate's best-case switching point of 0.5 V. Therefore, change the capacitor's capacitance to 1 µF and use 18 kΩ for R_2.

Hardware Debouncing

Numerical Example:

$R_1 + R_2$ controls the capacitor's charging time, and so sets the debounce period for the condition when the switch opens. The equation for charging is:

$$V_c = V_{final}\left(1 - e^{-t/RC}\right)$$

Solving for R:

$$R = -\frac{t}{Cln\left(1 - \frac{V_{th}}{V_{final}}\right)}$$

V_{final} is the final charged value of the 5 V power supply. V_{th} is the worst-case transition point for a high-going signal, which for our 74AHCT14 is 0.9 V. So,

$R_1 + R_2 = \frac{20\times10^{-3}}{1\times10^{-6}ln\left(1-\frac{0.9}{5}\right)} = 100.78\text{k}\Omega$ (if no diode is used). Since, already we have calculated $R_2 = 18$ kΩ, hence, $R_1 = 82$ kΩ.

Hardware Debouncing

Numerical Example:

The diode is an optional part needed only when the math goes haywire. It is possible, with the wrong sort of gate where the hysteresis voltages assume other values, for the formulas to pop out a value for $R_1 + R_2$ which is less than that of R_2. The part forms a shortcut that removes R_2 from the charging circuit. All the charge flows through R_1.

Equations will be the same except that we have to take the voltage drop across the diode into account. Change V_{final} to 4.3 V (5 minus the 0.7 V of diode's forward voltage drop) and solve for R_1 again.

Be wary of the components' tolerances! Standard resistors are usually ±5-10% and the capacitors are ±20% (for electrolytic) and ±30% (for small ceramic capacitors) are error prone from their nominal rating.

Software Debouncing

- Debouncing in hardware may give rise to additional cost, and it is more difficult to determine a good debouncing for all the push button switches that will be used. So, it may be preferable to debounce the switch in software.
- Debouncing a switch in software is very simple. The basic idea is to sample the switch signal at a regular interval and filter out any glitches. There are a couple of approaches to achieving this. These approaches assume a switch circuit like that shown in the explanation of switch bounce- a simple push switch with a pull-up resistor.
- While numerous algorithms exist to perform the debouncing function, we are going to limit ourselves to implementing two during the lab. Both approaches use a timer.

Software Debouncing

Approach 1

The first approach uses a counter to time how long the switch signal has been low. If the signal has been low continuously for a set amount of time, then it is considered pressed and stable.

```
1 Setup a counter variable, initialized to zero.
2 Setup a regular sampling event, perhaps using a timer. Use a period of about 1 ms.
3 On a sample event:
4   if the switch signal is HIGH then
5     RESET the counter variable to ZERO
6     SET internal switch state to released
7   else
8     Increment the counter variable to a maximum of 10
9   end if
10   if counter=10 then
11     SET internal switch state to pressed
12   end if
```

Software Debouncing

Approach 2

The second approach is similar to the first but uses a shift register instead of a counter. The algorithm assumes an **unsigned 8-bit register value**, such as that found in 8-bit microcontrollers.

```
1 Set up a variable to act as a shift register and initialize it to xFF.
2 Set up a regular sampling event, perhaps using a timer. Use a period of about 1 ms.
3 On a sample event:
4   SHIFT the variable towards the most significant bit
5   SET the least significant bit to the current switch value
6   if shift register value = 0 then
7     SET internal switch state to pressed
8   else
9     SET internal switch state to released
10  end if
```

Software Debouncing

- There are **two more approaches to software debouncing**.
- In the first technique, we wait for a switch closure, then test the switch again after a **short delay** (**15 milliseconds** or so). If it is still closed, we determine that the switch has changed state.
- In the second technique, we **test the switch periodically** to see if it **has changed its state**.

Software Debouncing

- When working with microcontrollers, we can deal with switch bounce in a different way that will save both hardware space and money. Some programmers do not care much about bouncing switches and just add a 50 ms delay after the first bounce. This will force the microcontroller to wait 50 ms for the bouncing to stop, and then continue with the program. This is actually not a good practice, as it keeps the microcontroller occupied with waiting out the delay.

- Another way is to use an interrupt for handling the switch bounce. Be aware that the interrupt might be fired on both the rising and falling edge, and some microcontrollers might stack up one waiting interrupt.

Software Debouncing

The following is a simple software debounce code for Arduino written in IDE

```
int inPin = 7;         // the pin number of the input pin
int outPin = 13;       // the pin number of the output pin

int counter = 0;       // how many times we have seen new value
int reading;           // the current value read from the input pin
int current_state = LOW;    // the debounced input value

// the following variable is a long because the time, measured in milliseconds,
// will quickly become a bigger number than can be stored in an int.
long time = 0;         // the last time the output pin was sampled
int debounce_count = 10; // number of millisecond/samples to consider before declaring a debounced input

void setup()
{
  pinMode(inPin, INPUT);
  pinMode(outPin, OUTPUT);
  digitalWrite(outPin, current_state); // setup the Output LED for initial state
}
```

```
void loop()
{
 // If we have gone on to the next millisecond
 if(millis() != time)
 {
  reading = digitalRead(inPin);

  if(reading == current_state && counter > 0)
  {
   counter--;
  }
  if(reading != current_state)
  {
    counter++;
  }
  // If the Input has shown the same value for long enough let's switch it
  if(counter >= debounce_count)
  {
   counter = 0;
   current_state = reading;
   digitalWrite(outPin, current_state);
  }
  time = millis();
 }
}
```

millis () function:

This function is used to return the number of milliseconds at the time, the Arduino board begins running the current program, that is, this Arduino function returns the present time in milliseconds from the moment the Arduino board is powered on or reset. The return value of millis is the number of milliseconds through an unsigned long variable since the program in Arduino started.

This number overflows i.e., goes back to zero after approximately 50 days. millis() function Syntax. millis (); This function returns milliseconds from the start of the program.

Software Debouncing

- The following program toggles two LEDs connected to a PIC microcontroller.

```
// INCLUDES
#include <stdio.h>
#include <stdlib.h>
#include <xc.h>

// CONFIG
#pragma config FOSC = INTOSCIO  // Oscillator Selection bits (INTOSC oscillator: I/O function on RA6/OSC2/CLKOUT pin, I/O function on RA7/OSC1/CLKIN)
#pragma config WDTE = OFF       // Watchdog Timer Enable bit (WDT disabled)
#pragma config PWRTE = OFF      // Power-up Timer Enable bit (PWRT disabled)
#pragma config MCLRE = ON       // RA5/MCLR/VPP Pin Function Select bit (RA5/MCLR/VPP pin function is MCLR)
#pragma config BOREN = ON       // Brown-out Detect Enable bit (BOD enabled)
#pragma config LVP = ON         // Low-Voltage Programming Enable bit (RB4/PGM pin has PGM function, low-voltage programming enabled)
#pragma config CPD = OFF        // Data EE Memory Code Protection bit (Data memory code protection off)
#pragma config CP = OFF         // Flash Program Memory Code Protection bit (Code protection off)

// DEFINITIONS
#define _XTAL_FREQ 4000000
#define LED1 PORTBbits.RB3
#define LED2 PORTBbits.RB2
#define BTN PORTBbits.RB5
```

```
// VARIABLES
char BTN_pressed = 0;
char BTN_press = 0;
char BTN_release = 0;
char BounceValue = 500;

// MAIN PROGRAM
int main(int argc, char** argv) {
    // Comparators off
    CMCON = 0x07;

    // Port directions, RB5 input, the rest is output
    TRISA = 0b00000000;
    TRISB = 0b00100000;

    // Port state, all low
    PORTA = 0b00000000;
    PORTB = 0b00000000;

    // Starting with LED1 high and LED2 low
    LED1 = 1;
    LED2 = 0;
```

```
while (1)
  {
    // If BTN is pressed
    if (BTN == 1)
    {
      // Bouncing has started so increment BTN_press with 1, for each "high" bounce
      BTN_press++;
      // "reset" BTN_release
      BTN_release = 0;
      // If it bounces so much that BTN_press is greater than Bounce value then the button must be pressed
      if (BTN_press > Bouncevalue)
      {
        // This is initial value of BTN_pressed.
        // If program gets here, button must be pressed
        if (BTN_pressed == 0)
        {
          // Toggle the LEDs
          LED1 ^= 1;
          LED2 ^= 1;
          // Setting BTN_pressed to 1, ensuring that we will not enter this code block again
          BTN_pressed = 1;
        }
```

```
BTN_press = 0;
          }
       }
       else
       {
          // Increment the "low" in the bouncing
          BTN_release++;
          BTN_press = 0;
          // If BTN_release is greater than Bouncevalue, we do not have a pressed button
          if (BTN_release > Bouncevalue)
          {
             BTN_pressed = 0;
             BTN_release = 0;
          }
       }

    }
    return (EXIT_SUCCESS);
}
```

References

- ATMega328 manual
- http://www.ganssle.com/debouncing-pt2.htm
- https://mansfield-devine.com/speculatrix/2018/04/debouncing-fun-with-schmitt-triggers-and-capacitors/

Thanks for Attending….