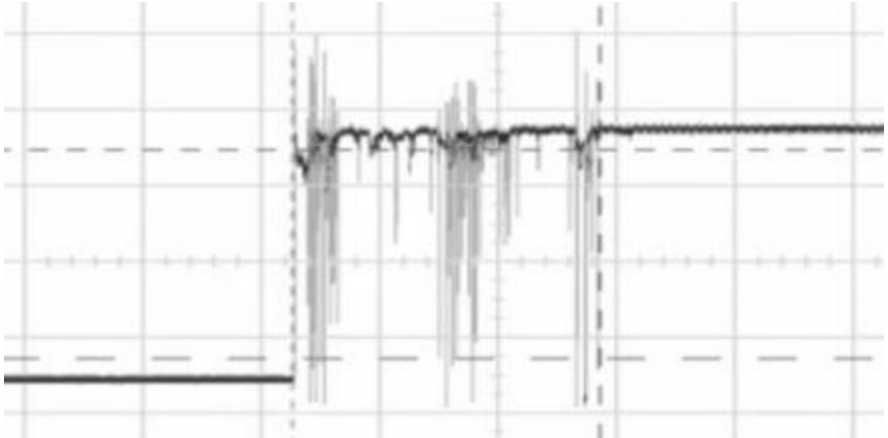
Switch Debouncing and Interrupts

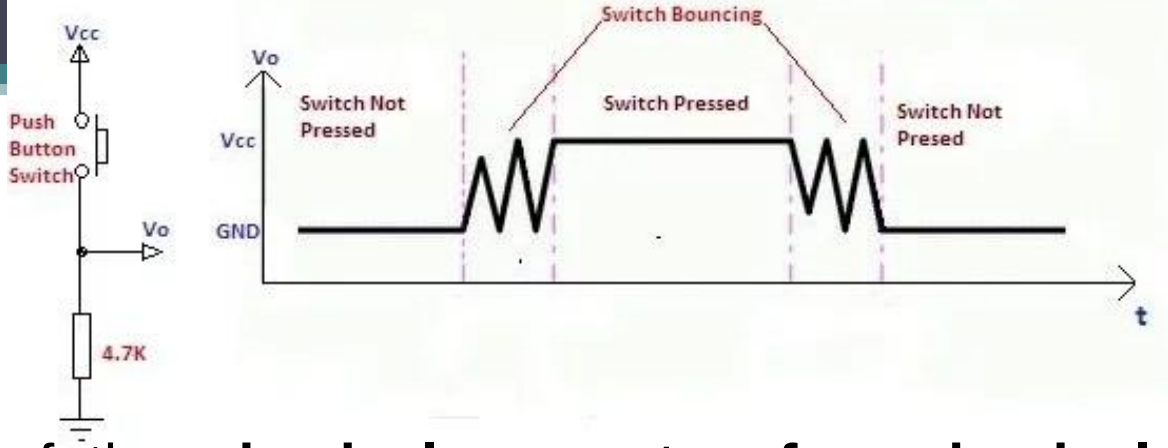
Prepared by
Nowshin Alam
Lecturer, AIUB
Email: nowshin.alam@aiub.edu

What is Debouncing?

- When the state of a mechanical switch is changed from open to closed and vise 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**.



Explanation



- Bouncing is a result of the **physical property of mechanical switches.**
- Switch and relay contacts are usually made of springy metals so when a switch is pressed, its essentially two metal parts coming together. This does not happen immediately, the switch bouncing between in-contact and not-in-contact until it finally settles down.
- 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, let's say we have a button that is connected between a voltage supply and the output probe. We intend to get an output of 5V (logic 1) when the switch is pressed, and 0V (logic 0) when unpressed. 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 → 1 → 0. What we end up seeing instead is the picture above. Before the signal settles to a flat 5V, it bounces between the two logic states many times.

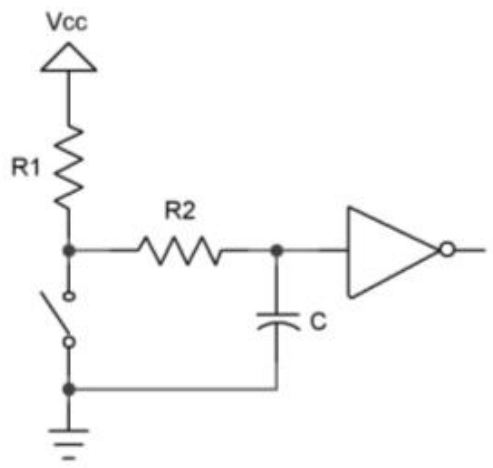
Why we need debouncing?

- 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 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

- There are two general way of getting rid of the bounces:
  - **Hardware debouncing:** adding circuitry that actually gets rid of the unwanted transitions.
  - **Software debouncing:** adding code that causes the application to ignore the bounces.

Hardware debouncing



- There are various implementations of circuits which 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 install a resistor/capacitor network with a time constant longer than the time it takes the bouncing contacts to stop. 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 may be best to go the hardware route.

Software debouncing

- Debouncing in hardware may give raise 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.
- 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

- 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 state.

Interrupts

- An interrupt is the automatic transfer of software execution in response to a hardware event that is asynchronous with the current software execution.

- The hardware event can either be a busy to ready transition in an external I/O device (like the UART input/output) or an internal event (like bus fault, memory fault, or a periodic timer).

Why do we need it?

- Let's think about a real-life example:

- You are heating up your food using the microwave.
- In this case, you can either:
  - Stare at the microwave while it heats up the food
  - Go about your life normally and go to the microwave when you get the signal that the food has been heated up

Why do we need it? (Cont.)

- Which of the above-mentioned options do you think is more efficient?
- Usually, we want to go about our life while the food is being heated up.
- This is because we want to make efficient use of our time.
- It is very similar for processors as well.
- We hear the ting sound from the microwave, stop what we are doing and go to take out the heated food.
- This ‘ting’ sound is an interrupt.

Why do we need it? (Cont.)

- Similarly, if the processor is waiting for some conditions to be fulfilled to perform a specific task (task 1):
  - It can wait and halt other processes till the conditions for that task is fulfilled.
  - It can execute the normal instructions and halt them for a brief time once the conditions for task1 have been fulfilled, finish task 1 and go back to executing the main routine.

Why do we need it? (Cont.)

- Clearly, the 2nd option is better for maximizing the processor's resource utilization.
- This is where interrupts come in handy.
- When the conditions for task1 has been met, it can simply trigger an interrupt, stop the main routine to complete the task and once finished go back to the main routine.

Interrupt Conditions

- Device arm
- NVIC enable
- Global enable
- Interrupt priority level must be higher than current level executing
- Hardware event trigger

Arming the device

- To arm a device means to allow the hardware trigger to interrupt.
- To disarm a device means to shut off or disconnect the hardware trigger from the interrupts.

Nested Vector Interrupt Controller (NVIC)

- The module that controls all the interrupts
- NVIC is device specific

Global Enable

- The main interrupt flag
- This is used to turn all the interrupts on or off
- Example: sei(); //globally enable interrupt

Priority

- Interrupts have priorities.
- Suppose, 2 different tasks must be done at the same time.
- The tasks must be assigned relative priorities order to decide which will be performed first

Trigger

- An asynchronous event which causes the interrupt
- For example, the push button press during experiment 4 can be a trigger.
- It can cause an interrupt which is registered by the module and is reacted to.
- However, what if all the conditions are not met but a trigger flag is set?

- In this case, rather than the request being dismissed, it is held pending, postponed until a later time.

What happens after a trigger?

- Once the interrupt is triggered and processed, the interrupt flag is cleared.
- Clearing the interrupt flag is called **acknowledgement**.

Interrupt Service Routine (ISR)

- The module that is executed when a hardware requests an interrupt.
- There may be 1 large ISR handling all the interrupt requests or, many small ISRs handling the many interrupts (interrupt vectors).
- **Example :**

ISR (TIMER0_OVF_vect) //enabling overflow vector inside timer0 using an ISR

ISR (TIMER0_COMPA_vect) //This is the Timer 0 Compare A interrupt service routine.

Remember, the ISR is a separate routine and requires a separate flowchart to represent.

General rules for ISR

- The ISR should execute as fast as possible.
  - The interrupt should occur when it's time to perform the required action
  - The ISR should perform the action
  - The ISR should end and return to the main function right away.
- Placing backward branches (busy wait, iterations) inside the ISR should be avoided.
- The percentage of time spent executing ISR should be small when compared to the time between interrupt triggers.

References

- ATMega328 manual
- http://www.ganssle.com/debouncing-pt2.htm
- https://mansfield-devine.com/speculatrix/2018/04/debouncing-fun-with-schmitt-triggers-and-capacitors/
- http://users.ece.utexas.edu/~valvano/Volume1/E-Book/C12_Interrupts.htm