

University of Hawaii
Department of Electrical Engineering

EE 361L Digital Systems and Computer Design Laboratory

Lab 2.3: Hardware Interrupts

Last updated by Galen Sasaki September 13, 2011

1. Introduction

The purpose of this laboratory experiment is to gain some experience with interrupts. An overview of interrupts is given in the PIC Information--Part 3 that comes with Lab 2.3. Read that first.

2. Prelab

Do this before lab to get some experience with interrupts and power down (sleep) mode. If you don't have a copy of MPLab on your computer, you can go to the HP Lab in the second floor of POST. The computers there should have copies.

Simple Interrupt Example

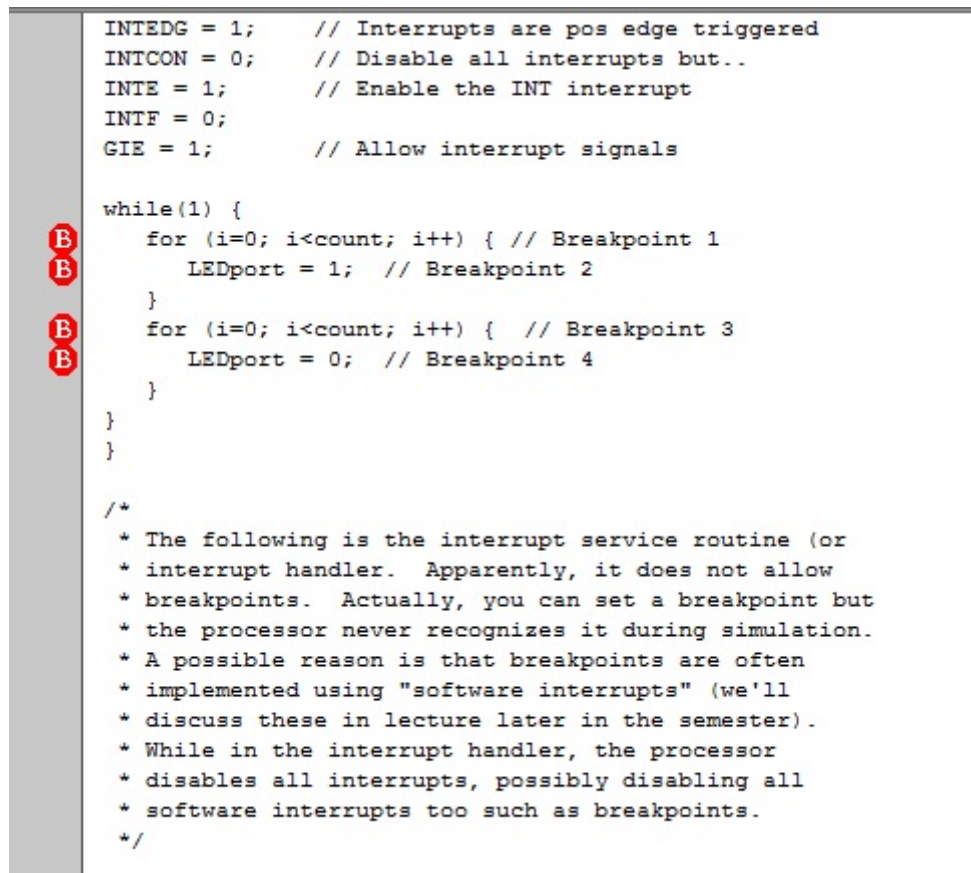
Attached to this document is an example C program called "simpleInt.c" which continually outputs pulses on an LED. The duration of each pulse is proportional to the number of interrupts the processor has seen so far.

Step 1. Read the comments.

Step 2. With the Project Wizard of MPLAB, create a project for the 16F648A processor. Set configuration bits as usual.

OSC: XT
WDT: Off
PUT: Disabled
MCLE: Enabled
BODEN: Enabled
LVP: Disabled
CPD: Disabled
CP: Off

Step 3. Set breakpoints indicated in the code – see comments of the code. There are four of them. To set a breakpoint, move the cursor all the way on the left of the line and set a breakpoint by right clicking. Below is a picture of the breakpoints.



```
INTEDG = 1;    // Interrupts are pos edge triggered
INTCON = 0;    // Disable all interrupts but..
INTE = 1;      // Enable the INT interrupt
INTF = 0;
GIE = 1;       // Allow interrupt signals

while(1) {
    for (i=0; i<count; i++) { // Breakpoint 1
        LEDport = 1; // Breakpoint 2
    }
    for (i=0; i<count; i++) { // Breakpoint 3
        LEDport = 0; // Breakpoint 4
    }
}

/*
 * The following is the interrupt service routine (or
 * interrupt handler. Apparently, it does not allow
 * breakpoints. Actually, you can set a breakpoint but
 * the processor never recognizes it during simulation.
 * A possible reason is that breakpoints are often
 * implemented using "software interrupts" (we'll
 * discuss these in lecture later in the semester).
 * While in the interrupt handler, the processor
 * disables all interrupts, possibly disabling all
 * software interrupts too such as breakpoints.
 */
```

Step 4. Set views of

Stimulus: Under Asynch, include PIN/SFR to RB0 (for the interrupt input) and Action to Low

Watch: Add “count” and “PORTB”

Step 5: Rebuild. Reset the processor. Fire the Stimulus on RB0 by clicking the fire button “>”.

Step 6: Run the simulation (button F9) and go through the break points a few times. Check the values under Watch.

Step 7: To cause a change of voltage at the RB0 input, go to the Stimulate window and change the Action value to High. Then fire the input again by clicking “>”. This will cause an upward transition at the interrupt port, and once you resume running the simulation, you should see “count” increase its value.

Step 8: Keep simulating and change the value of RB0 a few times. Make sure the processor increments the value of “count”.

Sleep Mode Example

Suppose you want the processor go to sleep after one pass through the while-loop. Add the line

```
SLEEP();
```

at the bottom of the while-loop.

Step 1: Rebuild and reset the processor

Step 2: In the Stimulus window Fire RB0 at Set Low

Step 3: Run until the first breakpoint

Step 4: In the Stimulus window, set RB0 to Set High but don't fire yet

Step 5: Run through the breakpoints. At some point, the simulator will just keep running (and not stop at breakpoints) because it executed the sleep instruction. Since it's asleep, it doesn't run the program or any breakpoints.

Step 6: In the Stimulus window, fire RB0, which you previously set to Set High. This will cause an interrupt and the processor will wake up. It will go to the interrupt handler and then back to the main routine. Then it will reach the first breakpoint. In the Watch window, "count" should equal 2.

Keep simulating a while longer by creating a few more interrupts. Play around with it.

Watch Dog Timer

Let us change this design by adding a Watch Dog Timer. The program "simpleWDT.c" attached to this document is a modification of the above programs that uses the timer. This program will go to sleep as before, but if there is no interrupt then a time-out by the Watch Dog Timer will wake it up. Subsequently, it will turn the LED on and off. Since the time-out period is about 36 ms, every 36 ms the LED will turn on momentarily.

Create another project for this program. Everything is the same as before except in the configuration bits set WDT to ON.

Set the four break points as shown in the code. Then simulate. Let it run through the while-loop a few times. Then look at the view from the Logic Analyzer. The signal from output RB1 will have spikes in it, which are when the LED goes on. The LED is on for a relatively short time because the Watch Dog Timer time-out is very long.

See if you can get it to interrupt and cause the "count" variable to increment.

3. Assignment

Here is the scenario for this assignment. You are doing a summer internship at a company specializing in custom designed scientific equipment. A UH professor (a customer) comes to your company requesting a device to measure the activity of wild life in the Krauss Hall fish pond (see Figure 1). The professor is interested in the toad activity in the pond. She has designed a special sensor that looks like a lily pad as shown in Figure 2. Whenever the lily pad is jumped on by a toad, the sensor creates an electric signal of 5 volts on its output wire. When there is no toad on the lily pad, the sensor output has an electrical signal of 0 volts. Toads are very territorial so at most one toad will be on a lily pad at any time.



Figure 1. Three views of the Krauss Hall pond.

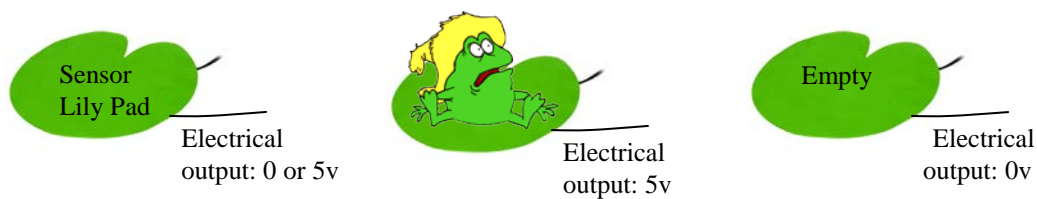


Figure 2. Lily pad sensor, toad on sensor, and empty sensor.

The professor wants a circuit that will count the number of times toads jump on the lily pad sensor. In addition, she wants the toad-count to be transmitted to a PC in her office in Krauss Hall. The transmission will be done using an infrared laser as shown in Figure 3. The toad-count is encoded into a transmission as a sequence of pulses of the laser, where the number of pulses is equal to the toad-count. For example, if the toad-count is five then the transmission will have five pulses. Each pulse is 0.25 seconds in duration and spaced apart by 0.25 seconds. A transmission for a toad-count of 5 is shown in Figure 3.

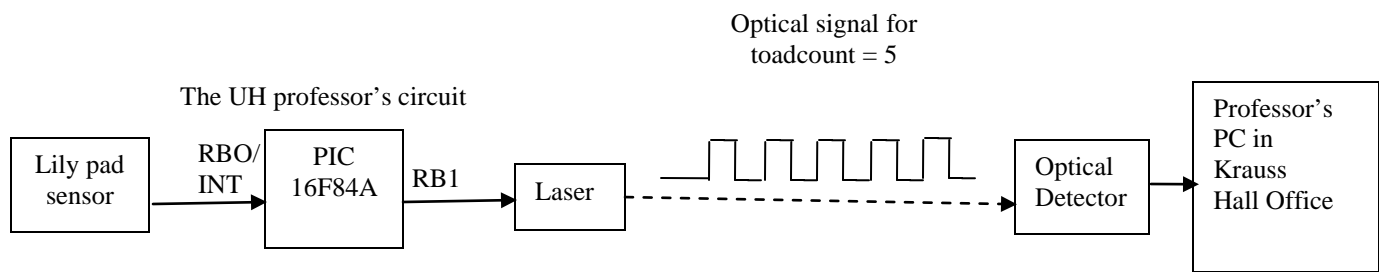


Figure 3. Lily pad sensor circuit with laser communication.

In addition, the circuit will run off a small battery, and it is expected that it will take data for a year. Thus, the circuit must be of low power.

Your boss gives you the responsibility of designing the circuit. Since your company has bought a huge number of PIC 16F648As at a discount price, you are required to use one for this project. Your boss gives you the following idea for the design.

The PIC 16F648A should be configured to have one output (e.g., RB1) that controls the infrared laser, turning it on and off, with a high and low output voltage, respectively. It should have the INT/RB0 interrupt input connected to the lily pad sensor as shown in Figure 3.

The software for the microcontroller is as follows. There should be a global variable **toadcount** which is equal to the number of toads counted so far. Initially, after reset, toadcount should be cleared to 0. The main program will continually check the global variable toadcount. The program will transmit a sequence of pulses (each of duration 0.25 seconds and spaced apart by 0.25 seconds) by driving the laser. The number of pulses in a sequence is equal to the toadcount. After transmitting a sequence of pulses, the main program will wait in a delay for 3 seconds before checking the toadcount again. Thus, the main program continually sends sequences of pulses (assuming toadcount > 0) and these sequences are spaced apart by 3 seconds of delay.

The software will have an interrupt handler that will increment the toadcount variable every time an interrupt occurs.

Subassignment 3.1. Implement your boss's idea. Write the software, program the microcontroller, and then configure the microcontroller with an interrupt input signal and an LED instead of the laser. Demonstrate your circuit to your boss (i.e., the TA).

Your boss will show you how to measure the average current drawn by the PIC. Measure and record the average current.

Don't use the Watch Dog Timer. Actually, to make this system robust, using the timer could be useful. But it will complicate the implementation. Don't use the timer for Subassignment 3.2 either. Yeah, we won't be winning any engineering awards this year.

For the interrupt input signal, use a mechanical switch circuit. Figure 5 shows a naive example of a mechanical switch circuit. If the switch is open (not pressed) then the output is pulled up to 5 volts. If the switch is closed (pressed) then the output is grounded to 0 volts. This circuit has a problem because it is a mechanical device, and in particular when the switch is pressed down, it may bounce a few times before making final contact. The bouncing may occur in a small amount of time (in 100s of nanoseconds) but for electronic circuits, this is a very long time. Thus, the electrical output will “ring” a little before becoming stable as shown in Figure 4.

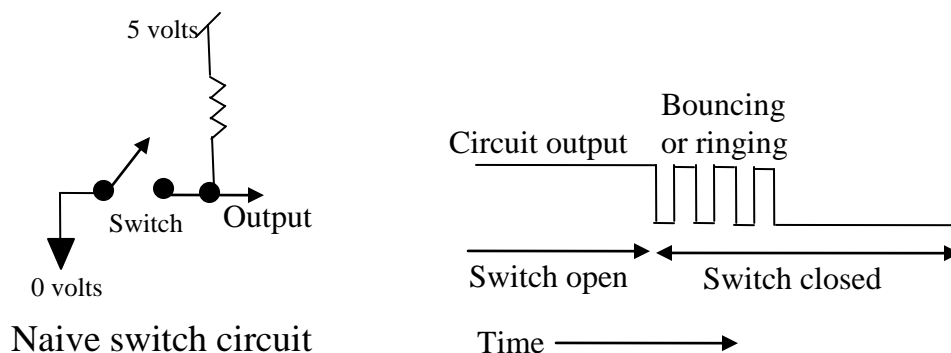


Figure 4. Naive switch circuit, and an example of bouncing or ringing.

A *debounced switch* will avoid the ringing and provide a clean transition. Figure 5 shows an example debounced switch. The mechanical switch is known as a *single-pole double-throw switch*, and it has three ports. This is different than the *single-pole single-throw* switch in Figure 4 which has two ports. (If you would like more information about switches and their definitions, see the wikipedia article <http://en.wikipedia.org/wiki/Switch>.) Appendix A has a explanation of the debounced switch circuit in Figure 5.

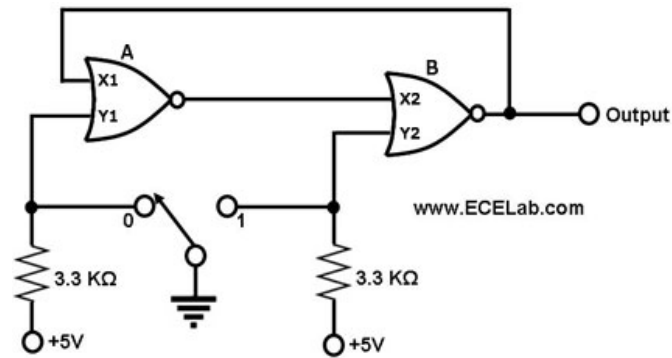


Figure 5. RS latch debounced switch circuit.

SubAssignment 3.2. After implementing Subassignment 3.1, you discover that the battery life of the circuit is only one month. The implementation drains too much power. You suggest to your boss that the circuit should be modified so that it transmits laser pulses only when the toad-count changes. This will reduce the power that is used to drive the laser.

Being a diligent intern, on your own, you read the entire data sheet of the PIC to understand all its capabilities and requirements. (Since your boss is busy and cannot oversee you all the time, he encourages you to learn on your own by allowing you to read all the available manuals and explore the lab facilities.) You discover that the microcontroller has a sleep mode, and realize that this can conserve power. You bring this up with your boss and get permission to redesign the circuit using the sleep mode as well as your previous suggestion to reduce the laser transmissions.

Modify the implementation of Subassignment 3.1 circuit so that the microcontroller will increment the toadcount whenever a toad jumps on the lily pad. Then the microcontroller will send a sequence of pulses to indicate the toad count to the professor's PC, after which the microcontroller goes to sleep. The microcontroller will awaken when another toad jumps on the lily pad. Thus, the new implementation is normally asleep unless a toad jumps on the lily pad sensor.

In your design, have one of the PIC's pins attached to an LED, which will refer to as the Awake LED. When the PIC is running, the Awake LED should be on. Just before the PIC goes to sleep, the Awake LED should be turned off. When the PIC awakes, then the Awake LED should go on again.

Demonstrate your working circuit to your boss (the TA). (Note: that the compiler may have a macro “sleep();” that is essentially the SLEEP instruction for use in C code. Also, the compiler may also allow `asm(“sleep”)` which will insert one line of assembly instruction into the C code -- in this case “sleep”. Ask your TA.) You should explain how you put the PIC to sleep, and explain your code.

Measure the current of the PIC while it's asleep. Compare the current with the measured current in Subassignment 3.1.

Appendix A. Debounced Switch

Before discussing the switch in Figure 6, we will go over some interesting circuits. Figure A.1(a) is a circuit of two inverters, which are driving each other. If Q and Q^* have opposite values (0 and 1 or 1 and 0) then the circuit will store these values forever. This is a simple circuit to store a bit Q , and is the basis for many storage circuits, e.g., computer memory RAM.

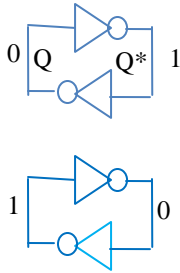


Figure A.1(a). Storing a bit: two cases.

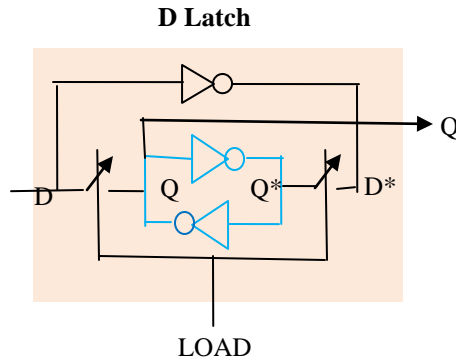


Figure A.1(b). D latch.

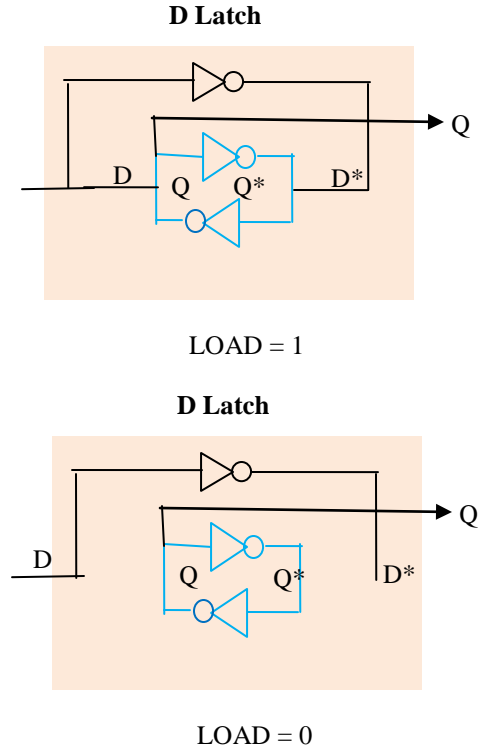


Figure A.1(c). D latch operating modes.

Figure A.1(b) is a modification of the circuit in Figure A.1(a). It has two electronic switches (transistors) which are controlled by an input control signal called **LOAD**. It also has an input D . Note that D^* is the complement of D . The circuit is called a D latch and it works this way:

- $\text{LOAD} = 1$, forces $Q = D$ and $Q^* = D^*$ as shown in Figure A.1(c).
- $\text{LOAD} = 0$, will make the circuit store the value as shown in Figure A.1(c).

Thus, LOAD causes the D latch to load the value at input D. This causes the output Q to be equal to D. If LOAD is disabled then the D latch stores (or holds) its value.

Now we will turn our attention to Figure 6. This circuit is explained in Figures A.2(a,b,c). Figure A.2(a) has a description of a 2-input NOR circuit including truth table. Shown is the NOR configured where one of the inputs A is a “Control”:

- Control = 1: the output of the NOR equals 0
- Control = 0: the output of the NOR equals the complement of input B. Thus, the NOR behaves like a voltage inverter for input B.

You can verify this behavior with the truth table.

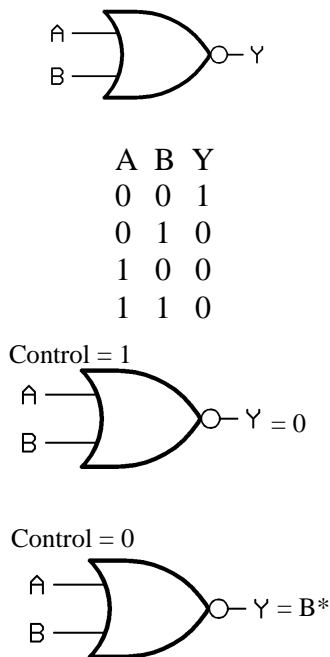


Figure A.2(a). NOR.

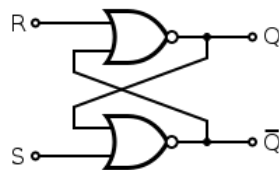


Figure A.2(b). RS latch.

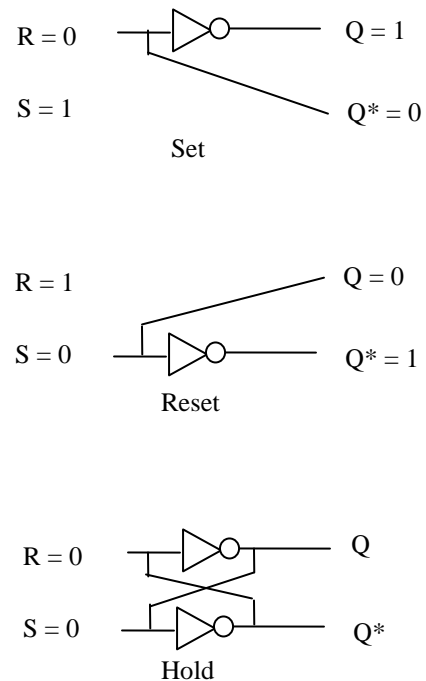


Figure A.2(b). Three cases for the RS latch.

Figure A.2(b) is an “RS latch”. As an aside, note that a “latch” more generally refers to a combinational circuit that has feedback (outputs are connected to inputs). As we have already learned, a latch is useful because it can function as memory, e.g., the D latch. But in many cases latches can be bad. The bad cases are when latches are unintentionally created in a circuit design. We will discuss bad latches in lecture later in the semester.

Let us continue discussing the RS latch. The circuit works as shown in Figure A.2(c) (we use Figure A.2(a) to determine how the NORs should behave in each case):

- $R = 0, S = 1$, forces $Q = 1$ -- this is known as Set to 1.
- $R = 1, S = 0$, forces $Q = 0$ -- this is known as Reset to 0.
- $R = 1, S = 1$, will make the circuit store the value, which we call “hold”
- $R = 0, S = 0$ is “illegal” because it causes problems so we will ignore it

Figure A.3 illustrates how the circuit in Figure 6 avoids bouncing. Initially, $R = 0, S = 1$, and $Q = 1$. Next, the switch is thrown. Then $R = 1, S = 0$, and $Q = 0$. The switch may bounce. But when it bounces, it is somewhere in the middle of the two ports, which means $R = 1$ and $S = 1$. This is the “hold” state, and Q remains 1. Thus, there is no ringing at Q . Finally, the switch will settle down and it will remain $R = 1, S = 0$, and $Q = 0$.

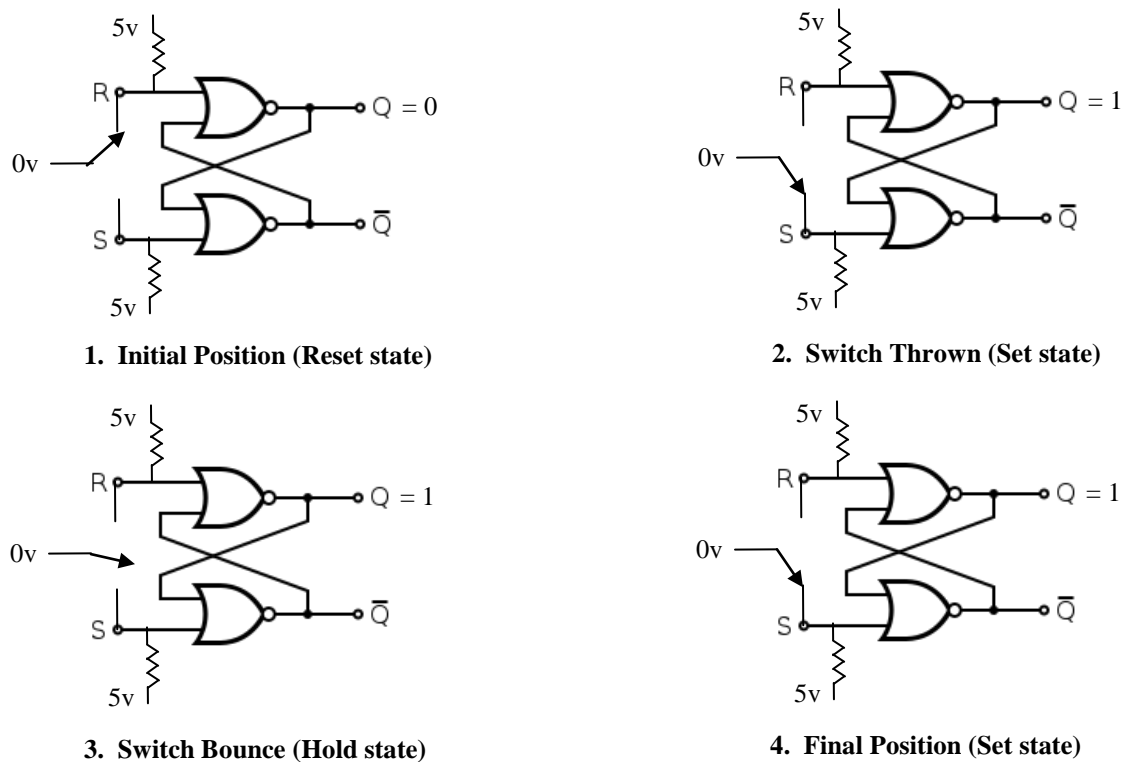


Figure A.3. Switch being thrown in a debounced switch circuit, steps 1, 2, 3, and 4.

Shown are states of the RS latch.