DE-BOUNCING PUSH BUTTONS AND SWITCHES

EGR/CS 333 Digital Design and Interfacing
J.Wundelich, Ph.D.

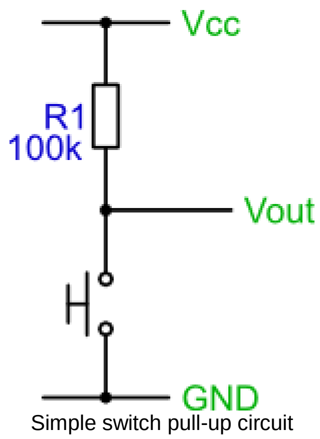
Adapted from some of my previous lecture notes, plus edited references:
http://www.labbookpages.co.uk/electronics/debounce.html by Dr. Andrew Greensted, June, 2010
http://www.all-electric.com/schematic/debounce.htm by "Electronic Kitchen"
http://www.ganssle.com/debouncing.htm by **The Ganssle Group, PO Box 38346, Baltimore, MD 21231.** June, 2008 (**MOST OF THIS HANDOUT**)

WHAT IS SWITCH BOUNCE?

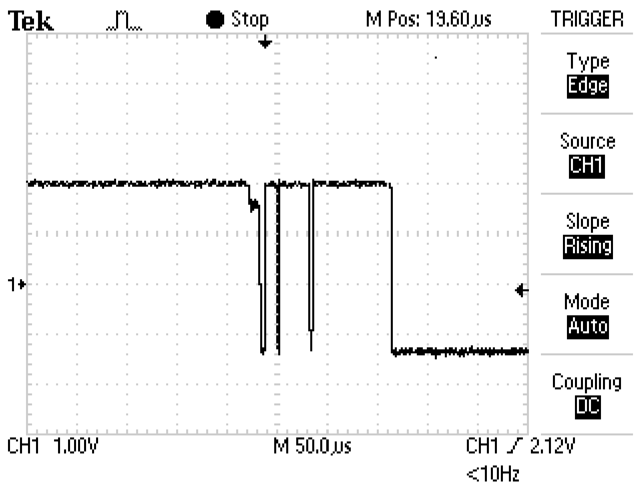
by Dr. Andrew Greensted, June, 2010 http://www.labbookpages.co.uk/electronics/debounce.html

The left-hand image below shows a simple push switch with a pull-up resistor. The right hand image shows the trace at the output terminal, Vout, when the switch is pressed. As can be seen, pressing the switch does not provide a clean edge. If this signal was used as an input to a digital counter, for example, you'd get multiple counts rather than the expected single count. Note that the same can also occur on the release fo a switch.The problem is that the contacts within the switch don't make contact cleanly, but actually slightly 'bounce'. In the switch waveform the bouncing lasts for about 150us

.



Simple switch pull-up circuit



Switch bounce produced on switch press

.

SOLUTION #1:

by "Electronic Kitchen" http://www.all-electric.com/schematic/debounce.htm

You can use CMOS or TTL but the component values change. I'll show the component values for CMOS but it should be noted that the values should also be sweetened to taste. That is to say that you can Taylor the reaction time of the switch depending on your application's requirement and the feel you'd like to give it.

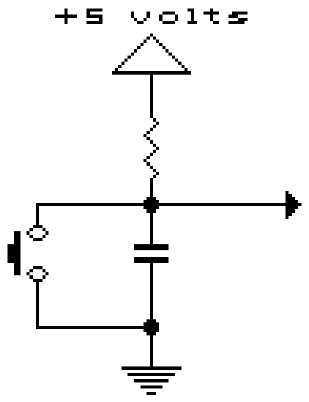
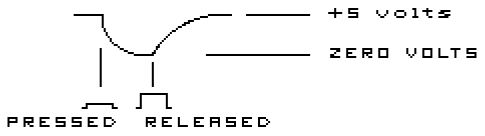
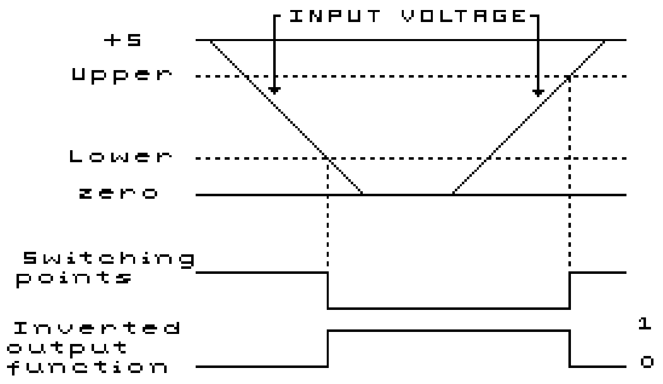


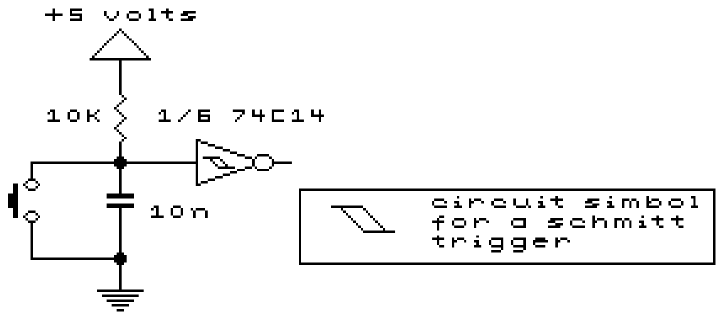
Figure 1 shows the basic arrangement. A resistor pulls a capacitor up to 5 volts (or what ever your positive supply voltage is if you're using CMOS. Up to 15 volts) Because of the time coefficient of the capacitor and resistor, this takes time to occur. The bigger the capacitor or resistor, the longer it takes. Placed at the junction of the resistor and capacitor, the momentary press button pulls the capacitor to ground. It doesn't short out the power supply because of the resistor. When the switch is made, the voltage on the capacitor falls away very rapidly to nothing. or near nothing depending on the switch. (That's another story and I'll get to that) When the switch is released the potential across the capacitor is charged up again slowly by the resistor. The charge/discharge rate is a smooth logarithmic curve. One faster than the other.



However CMOS and TTL don't like smooth transitions. Even TTL has a small linear region and will tend to get into undecided states. CMOS is even worse because it can easily run linear and because of it's high impedance nature can flollop round like an old mattress. A device called a Schmitt Trigger is needed. A Schmitt trigger is a device that will not change state till a certain threshold voltage has been reached. Called a hystereesis level. Actually there are two. An upper hysteresis and a lower. Assume the output is at 'zero'. In order to change this state to a one, the input voltage must attain or exceed the upper hysteresis level. Likewise if the output is a 'one' the input voltage must fall bellow the lower hysteresis level before it will switch back to 'zero'. Shown bellow.



There are a number of parts in both CMOS and TTL. The most common of which is the 7414 hex inverting schmitt trigger. This is basically an invertor with a preset hysteresis function. It is also available in CMOS called a 74C14. Also in CMOS is the 4093 which is a NAND gate schmitt trigger in a quad package. These are commonly used to clean up nasties in data streams. Referred to as "squaring up" since it can take a rough waveform and put square edges back on it. Also useful for resetting microprocessors which is similar to the switch debounce.



Above is the full debounce circuit using one sixth of a 74C14 package. Remember the output is inverted. If the input is high the output is low and visa versa. So If the resistor has pulled the capacitor high then the output of the schmitt trigger will be low. When the switch is engaged it pulls the voltage on the capacitor to near zero. As it crosses the lower hysteresis threshold the output of the schmitt trigger flys high until the switch is released. At which time the capacitor begins charging again and when it charges enough to cross the upper hysteresis level, the output flys low again. The speed at which one can press the switch is determined by the time coefficient of the resitor and capacitor. If the button is pressed again before the capacitor has charged up sufficiently, the output will remain high. It is also possible to give the switch on a time coefficient as well by placing a resistor in series with the switch to ground. However it should be noted that this must be significantly smaller than the value of the pull-up resistor otherwise it will merely equalize somewhere in the middle. No switching will take place since nether threshold was obtained.

Which brings me neatly to a word of caution about momentary pressbutton switches. Most switches have a moment that pressed two bits of metal together shorting the two switch terminals together. This kind of switch has very little resistance. So little as to make no odds as far as our switch is concerned. However it is common these days to find switches made from rubber actuators with conductive surfaces. These typically have a resistance of 50 ohms or so. But can be as high as 200 ohms. This means that when the switch makes, it only pulls down by the resistance across the pad. As if you had placed a resistor in series with the switch. In which case it will have a slight switch-on time depending on the value of the other resistor. This is particularly important to realize with TTL. Since a typical pull-up resistor of 4K7 with a pull down of around 200 ohms probably wont reach the lower threshold. Or worse. It will some times but not when you really need it.

The reason for pulling up with a resistor and not down is also typical of TT Circuits. Though it is possible to turn the whole scheme upside down. As mentioned, TTL inherently pulls it self up. To pull it down may require 200 Ohms or less and this may be difficult to work with. Note that CMOS and it's high impedance nature neither pulls up nor down. and it can be easily used upside down. But it is more general to stick with the conventions of TTL. If you wished to have an output that is stable high and goes low when the button is pressed, run the output through a second invertor instead. Either a normal invertor from another package or a spare one from the Schmitt trigger package.

It should be noted that upon power-up a pulse will be generated from this debounce circuit. This is because the capacitor needs time to charge up initially. The Schmitt trigger's output will rise very quickly on power-up and then fall when the capacitor charges. Ironically this is the simplest way to reset a microprocessor or logic system. If the Microprocessor's reset pin is active high (Meaning it's reset when the pin is high) then the micro will be reset for a time period of the capacitor/resistor. When the cap charges the schmit trigger will go low and the micro will initialize and fetch it's first instruction. Likewise if this initial switching is a problem for further logic that it controls then a reset circuit could be used to inhibit the action of the logic until everything stabilizes. Just make sure that the capacitor/resistor of the reset circuit is something larger than the capacitor/resistor combos of the debounce circuit.

Now I mentioned that this is useful for switches up to 16. After that there are a couple of ICs from Nationa Semiconductor in their CMOS range. Known as MM74C922 and MM74C923. 16 key and 20 key encoders respectively. These single chips solutions can debounce up to 20 switches in a matrix with only a couple of external components. It outputs a 4 or 5 bit code representing the switch pressed and also a signal to notify further logic that a switch has been pressed. It can be configured to work in almost any situation. The DATA on these can be found at National's most excellent Web site. All data sheets are in Adobe's acrobat format.

Debouncing Experiments and Hardware and Software Solutions:

by the Ganssle Group http://www.ganssle.com/debouncing.htm

Some TV remotes work even when you bounce the beam off three walls. Others don't. One vendor said reliability simply isn't important as users will subconsciously hit the button again and again till the channel changes. When a single remote press causes the tube to jump two channels, developers know lousy debounce ***code*** is at fault. When the contacts of any mechanical switch bang together they rebound a bit before settling, causing bounce

Switches tested. The upper left is switch A, with B to its right, working to E (in red), and then F below A, etc.

Many of the switches exhibited wild and unexpected behavior. Bounces of under 100 nsec were common (more on this later). No reasonable micro could reliably capture these sorts of transitions, so instead used scope, connecting both analog and digital channels to switch to see in the analog domain, and how a computer would interpret the data. A 5 volt supply and 1k pull-up completed the test jig. If a sub-100 nsec transition won't be captured by a computer why worry about it? Unfortunately, even a very short signal will toggle the logic once in a while. **Tie it to an interrupt and the likelihood increases**. Those transitions, though very short, will occasionally pervert the debounce routine.

TESTS:

- Trigger switches from an old cheap game-playing joystick (the three yellow ones in the picture)
- The left mouse button from an ancient Compaq computer (on PCB in upper left corner)
- Toggle switches, pushbuttons, and slide switches. Some were chassis mount, others were to be soldered directly onto circuit boards.

Each switch pressed 300 times, logging the min and max amount of bouncing for both closing and opening of contacts, logged every individual bounce time for each actuation into a spreadsheet for half the switches

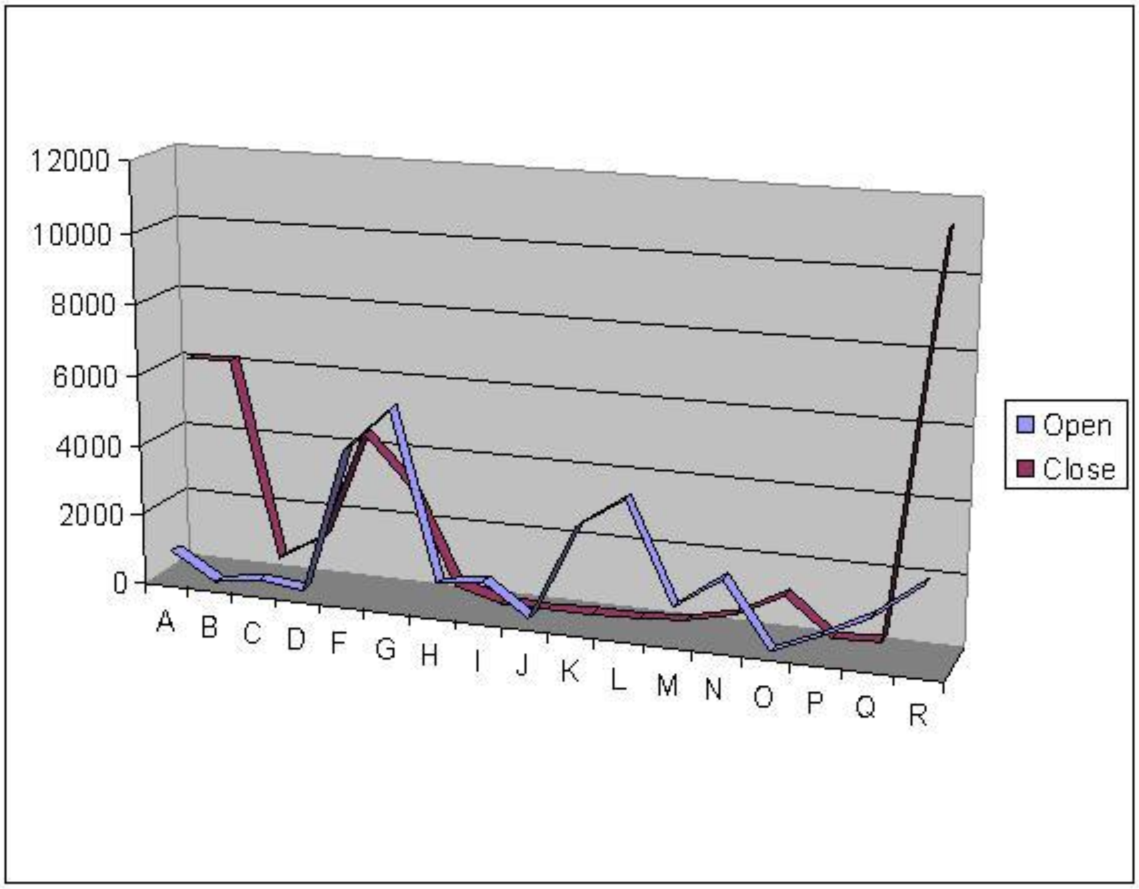
RESULTS:

How long do switches bounce for? The short answer: sometimes a lot, sometimes not at all.

- Only two switches exhibited bounces exceeding 6200 µsec. Switch E, what seemed like a nice red pushbutton, had a worst case bounce when it opened of 157 msec - almost a 1/6 of a second! Yet it never exceeded a 20 µsec bounce when closed.
- Another switch took 11.3 msec to completely close one time; other actuations were all under 10 msec.
- Toss out those two samples and the other 16 switches exhibited an average 1557 µsec of bouncing, with, a max of 6200 µsec.
- Seven of the switches consistently bounced much longer when closed than when opened' for most of the switches many bounces on opening lasted for less than 1 **µ**sec. Yet the very next experiment on the same switch could yield a reading in the hundreds of microseconds.
- Identical switches were not particularly identical. Two matching pairs were tested; each twin differed from its brother by a factor of two.

Different impacts created quite an array of bouncing. So in these experiments I tried to actuate each device with a variety of techniques. Pushing hard or soft, fast or slow, releasing gently or with a snap, looking for different responses.

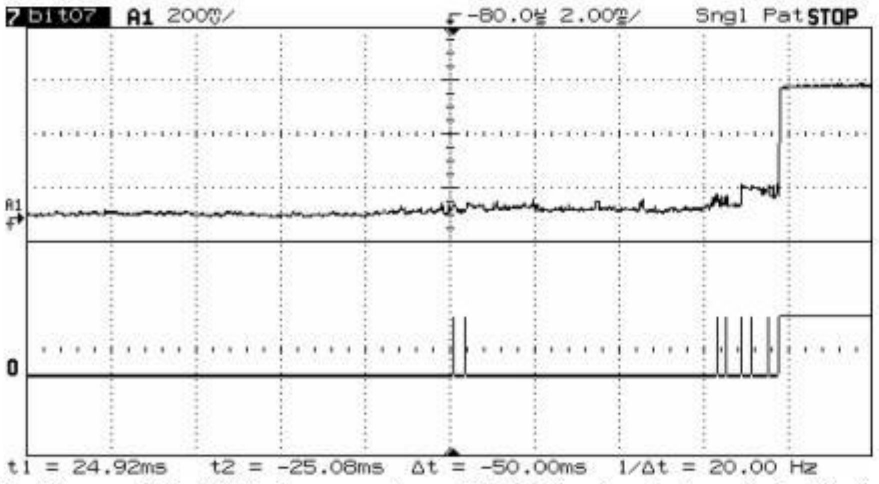
- F, a slide switch, was very sensitive to the rate of actuation.
- Toggle switch G showed a 3 to 1 difference in bounce times depending on how fast its lever was bonked
- .A few others showed similar results but there was little discernable pattern.



Bounce times in microseconds, for opening and closing each switch (number A to R). Switch E was left out, as its 157 msec bounces would horribly skew the graph.

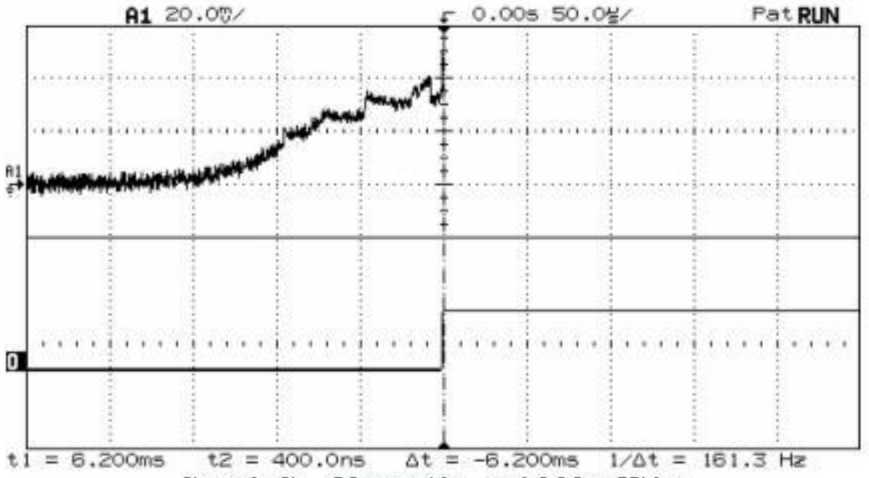
Analog behavior:

- A few operated as expected, yielding a solid zero or 5 volts. But most gave much more complicated responses.
- The MSO responded to digital inputs assuming TTL signal levels. That means 0 to .8 volts is a zero, 0.8 to 2.0 is unknown, and above 2 a one. The instrument displayed both digital and analog signals to see how a logic device would interpret the real-world's grittiness.
- Switch A was typical. When opened the signal moved just a bit above ground and wandered in the hundreds of millivolts range for up to 8 msec. Then it suddenly snapped to a one. As the signal meandered up to near a volt the scope interpreted it as a one, but the analog's continued uneasy rambles took it in and out of "one" territory. The MSO showered the screen with hash as it tried to interpret the data. It was if the contacts didn't bounce so much as wiped, dragging across each other for a time, acting like a variable resistor.



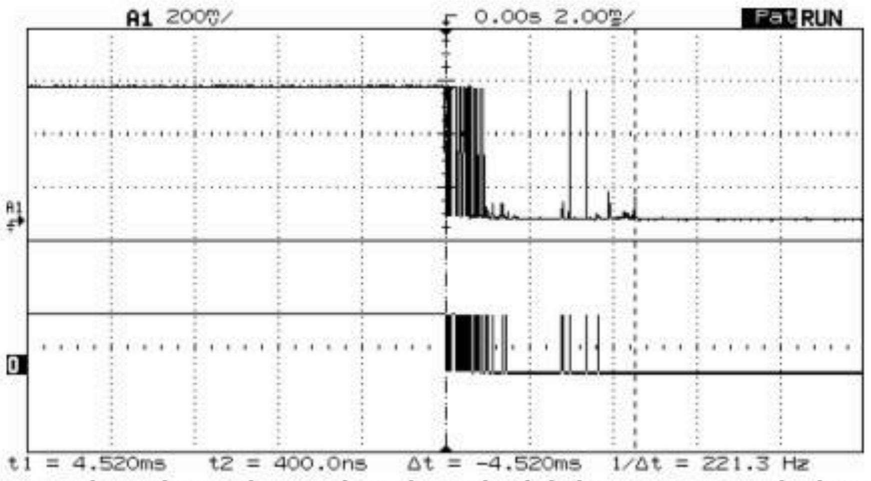
Switch A at 2 msec/div. Note 8 msec of unsettled behavior before it finally decides to open.

Looking into this more deeply, expanded the traces for switch C and, with the help of Ohm's Law, found the resistance when the device opened crawled pretty uniformly over 150 µsec from zero to 6 ohms, before suddenly hitting infinity. There was no bouncing per se; just an uneasy ramp up from 0 to 300 mV before it suddenly zapped to a solid +5.



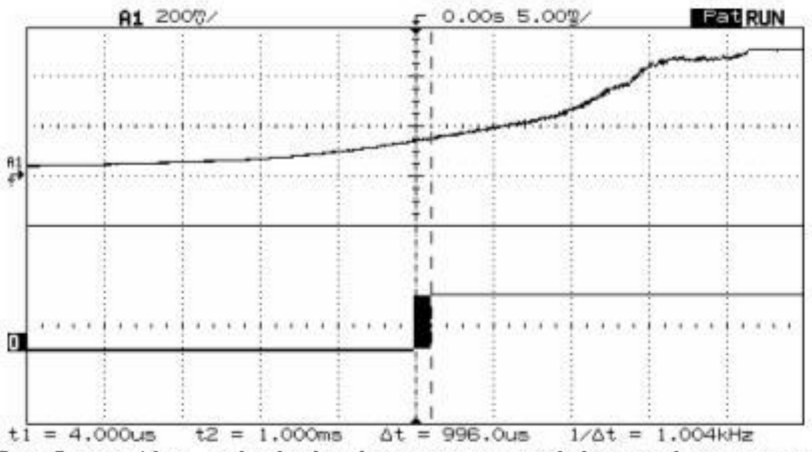
Switch C – 50 µsec/div and 200 mV/div.

Another artifact of this wiping action was erratic analog signals treading in the dreaded no-man's land of TTL uncertainty (0.8 to 2.0 volts), causing the MSO to dither, tossing out ones or zeroes almost randomly, just as your microprocessor would if connected to the same switch.



Switch B – note how the analog peak to the right didn't quite trigger the logic channel.

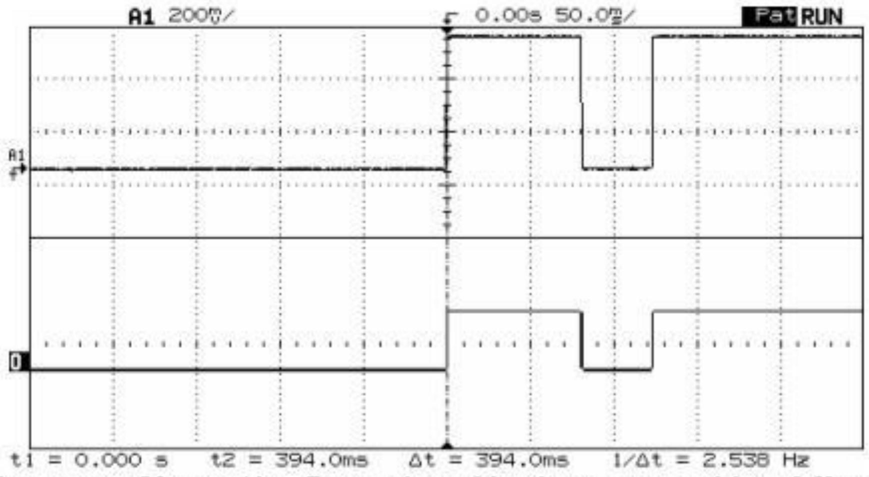
The two from the el cheapo game joystick were nothing more than gold contacts plated onto a PCB; a rubber cover, when depressed, dropped some sort of conductive elastomer onto the board. Interestingly, the analog result was a slow ramp from zero to five volts, with no noise, wiping or other uncertainty. Not a trace of bounce. And yet.. . . the logic channel showed a msec or so of wild oscillations! What's going on?
With TTL logic, signals in the range of 0.8 to 2.0 volts are illegal. Anything goes, and everything did. Tie this seemingly bounce-free input to your CPU and prepare to deal with tons of oscillation - virtual bounces.



Switch K at 5 msec/div – which slowly ramps up and down when actuated. Cool!

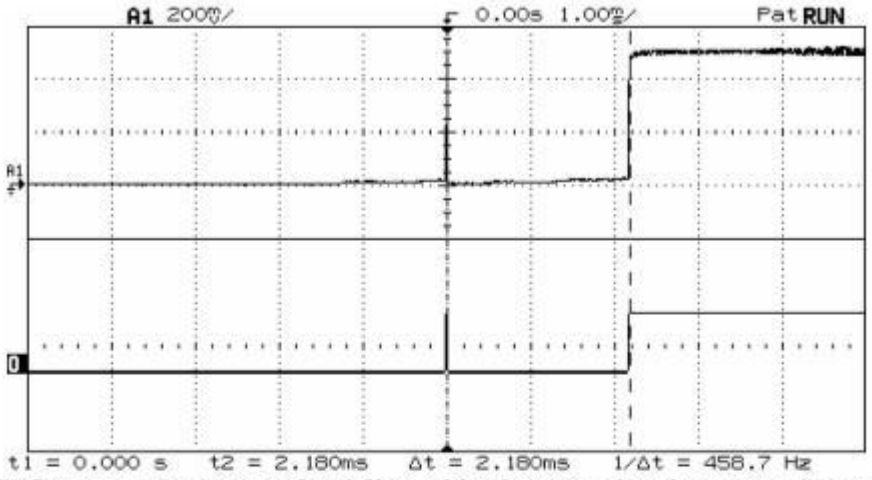
Assessment there's much less whacking of contacts going on than we realize. A lot of the apparent logic hash is from analog signals treading in illegal logic regions. Regardless, the effect on our system is the same and the treatment identical. But the erratic nature of the logic warns us to avoid simple sampling algorithms, like assuming two successive reads of a one means a one.

Anatomy of a Bounce
So we know how long the contacts bounce and that lots of digital zaniness - ultra short pulses in particular - can appear.
But what happens during the bounce? Quite a lot, and every bounce of every switch was different. Many produced only high speed hash till a solid one or zero appeared. Others generated a serious pulse train of discernable logic levels like one might expect. Special interested given to results that would give typical debounce routines heartburn.
Consider switch E again, that one with the pretty face that hides a vicious 157 msec bouncing heart. One test showed the switch going to a solid one for 81 msec, after which it dropped to a perfect zero for 42 msec before finally assuming its correct high state. Think what that would do to pretty much any debounce code!



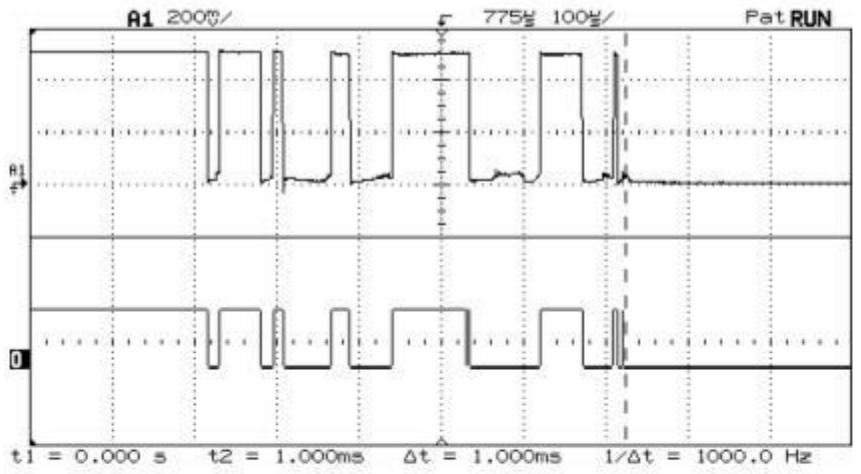
Switch E again, at 50 msec/div. Do you have blood pressure problems? You will after writing code to debounce this!

Switch G was pretty well behaved, except that a couple of times it gave a few microsecond one before falling to zero for over 2 msec. Then it assumed its correct final one. The initial narrow pulse **might escape your polled I/O, but would surely fire off an interrupt**, had you dared wire the system so. The poor ISR would be left puzzled as it contemplates 2 msec of nothingness. "Me? Why did it invoke me? Ain't nuthin' there!"

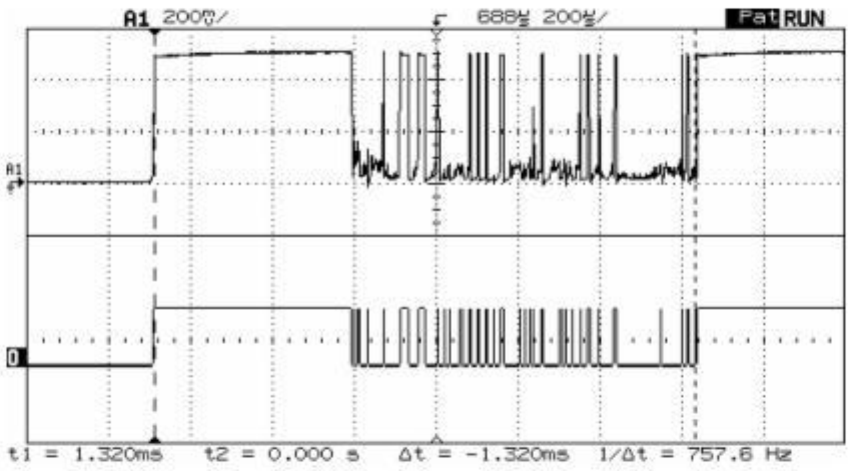


Switch G. One super narrow pulse followed by 2 msec of nothingness. A sure-fire ISR confuser.

O is a very nice, high quality microswitch which never showed more than 1.18 msec of bouncing. But it usually generated a pulse train guaranteed to play havoc with simple filter code. There's no high speed hash, just hard-to-eliminate solid ones and zeroes. One actuation yielded 7 clean zeroes levels ranging in time from 12 to 86 µsec, and 7 logic ones varying from 6 to 95 µsec. Easy to filter? Sure. But not by code that just looks for a couple of identical reads.



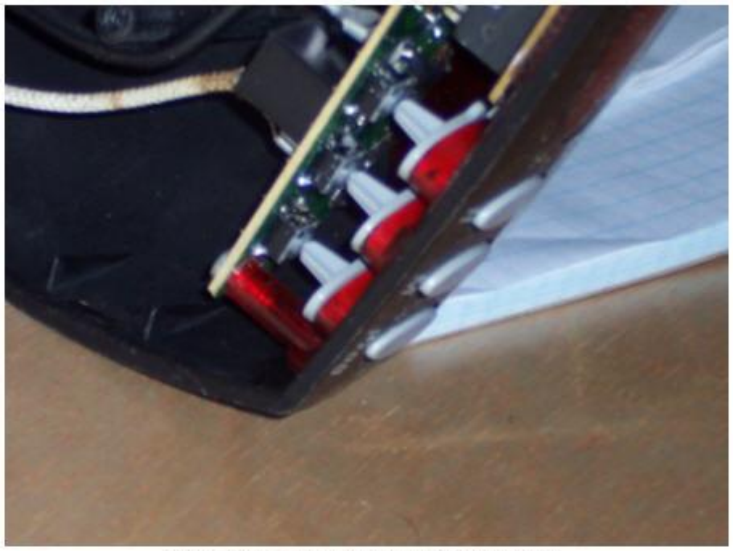
Switch O, which zaps around enough to confuse dumb debouncers.



Switch Q – when released, it goes high for 480 µsec before generating 840 µsec of hash, a sure way to blow an interrupt system mad if poorly designed.

What happens if we press the buttons really, really fast? Does that alter the bouncing in a significant way? a modified experiment, connecting MSP430 board to a sizeable 3 amp four pole relay. Downloading code into the CPU's flash let the relay be toggled at different rates. Bounce times ranged from 410 to 2920 µsec, quite similar to those of the switches, presumably validating the experiment. The relay had no noticeable analog effects, banging cleanly between 0 and 5 volts.

The raucous clacking of contacts overwhelmed our usual classical fare for a few hours as the MSO accumulated bounce times in storage mode. When the relay opened it always had a max bounce time of 2.3 to 2.9 msec, at speeds from 2.5 to 30 Hz. More variation appeared on contact closure: at 2.5 Hz bounces never exceeded 410 µsec, which climbed to 1080 µsec at 30 Hz. Why?   But it's clear there is some correlation between fast actuations and more bounce. These numbers suggest a tractable factor of two increase, though, not a scary order of magnitude or more.



PCB switches in a cheap coffee maker.

HARDWARE DEBOUNCERS

Double-Throw TOGGLE SWITCH

Figure 1 shows the classic debounce circuit. Two cross-coupled NAND gates form a very simple Set-Reset (SR) latch. The design requires a double-throw switch. Two pull-up resistors generate a logic one for the gates; the switch pulls one of the inputs to ground.

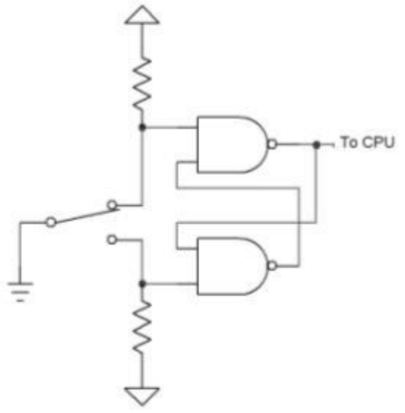


Figure 1: The SR debouncer

With the switch in the position shown the upper gate's output will be a one, regardless of the value of the other input. That and the one created by the bottom pull-up resistor drives the lower NAND to a zero . . . which races around back into the other gate. If the switch moves between contacts, and is for a while suspended in the nether region between terminals, the latch maintains its state because of the looped back zero from the bottom gate.
The switch moves a rather long way between contacts. It may bounce around a bit, but will never bang all the way back to the other contact. Thus, the latch's output is guaranteed bounce-free. **SEE EGR/CS333 LECTURE NOTES FOR DETAILED TIMING TRACES**

The circuit suggests an alternative approach, a software version of the same idea. Why not skip the NAND pair and run the two contacts, with pull-ups, directly to input pins on the CPU? Sure, the computer will see plenty of bounciness, but write a trivial bit of code that detects *any* assertion of either contact. which means the switch is in that position, as follows:

```
        if(switch_hi())state=ON;
        if(switch_lo())state=OFF;
```

switch_hi and switch_lo each reads one of the two throws. Other functions in the program examine variable state to determine the switch's position. This saves two gates but costs one extra input pin on the processor. It's the simplest - and most reliable - debounce code possible.
The MC14043/14044 chips consist of four SR flip flops, so might be an attractive solution for debouncing multiple switches. A datasheet can be found at http://www.radanpro.com/el/dslpro.php?MC14043.pdf.

Single-Throw TOGGLE SWITCH *(or a PUSH-BUTTON !… SEE EGR/CS333 LECTURE NOTES, and discussion above)*

An RC Debouncer

The SR circuit is the most effective of all debouncing approaches. but it's rarely used. Double-throw switches are bulkier and more expensive than the simpler single-throw versions. An awful lot of us use switches that are plated onto the circuit board, and it's impossible to make DP versions of these. So we prefer alternative designs that work with cheap single-throw switches.Though complex circuits using counters and smart logic satisfy our longing for pure digital solutions to all problems, it's easier and cheaper to exploit the peculiar nature of a resistor-capacitor (RC) network. Charge or discharge a capacitor through a resistor and you'll find the voltage across the cap rises slowly; it doesn't snap to a new value like a logic circuit. Increase the value of either component and the time lag ("time constant") increases.

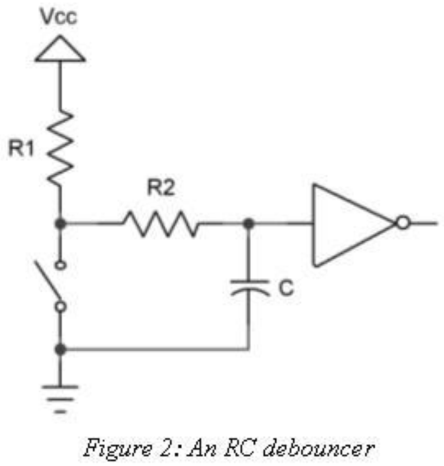


Figure 2: An RC debouncer

Figure 2 shows a typical RC debouncer. Opens the switch, the voltage across the cap is zero, but it starts to climb at a rate determined by the values of R_1, R_2 and C. Bouncing contacts pull the voltage down and slow the cap's charge accumulation. If we're very clever in selecting the values of the components the voltage stays below a gate's logic one level till all of the whacking and thudding ceases. (If the time constant is too long, of course, the system won't be responsive to fast switch actuations). The gate's output is thus a pristine bounce-free logic level.

Now suppose the switch has been open for a while. The cap is fully charged. Snap! The user closes the switch, which discharges the cap through R_2. Slowly, again, the voltage drools down and the gate continues to see a logic one at its input for a time. Perhaps the contacts open and close a bit during the bouncing. While open, even if only for short periods, the two resistors start to recharge the cap, reinforcing the logic one to the gate. Again, the clever designer selects component values that guarantee the gate sees a one until the clacking contacts settle. Most will puzzle over R_2, The classic RC debouncer doesn't use this resistor, yet it's critically important to getting a thwack-free output from the gate. R_2 serves no useful purpose when the switch opens. R_1 and C effectively remove those bounces. But strange things can happen when suddenly discharging a capacitor. The early bouncing might be short, lasting microseconds or less. Though a dead short should instantly discharge the cap, there are no pristine conditions in the analog world. The switch has some resistance, as do the wires and PCB tracks that interconnect everything.

Every wire is actually a complex circuit at high speeds. You wouldn't think a customer flipping the switch a few times a second would be generating high-speed signals, but sub-microsecond bounces, which may have very sharp rise times, have frequency components in the tens of MHz or more. Inductance and stray capacitance raises the impedance (AC resistance) of the closed switch. The cap won't instantly discharge. Worse, depending on the physical arrangement of the components, the input to the gate might go to a logic zero while the voltage across the cap is still one-ish. When the contacts bounce open the gate now sees a one. The output is a train of ones and zeroes - bounces.

R_2 insures the cap discharges slowly, giving a clean logic level regardless of the storm of bounces. The resistor also limits current flowing through the switch's contacts, so they aren't burned up by a momentary major surge of electrons from the capacitor.
Another trick lurks in the design. The inverter cannot be a standard logic gate. TTL, for instance, defines a zero as an input between 0.0 and 0.8 volts. A one starts at 2.0. In between is a DMZ which we're required to avoid. Feed 1.2 volts to such a gate and the output is unpredictable. But this is exactly what will happen as the cap charges and discharges.
Instead **use** "**Schmitt Trigger INVERTOR**". These devices have hysteresis; the inputs can dither yet the output remains in a stable, known state. Never run the cap directly to the input on a microprocessor, or to pretty much any I/O device. Few of these have any input hysteresis.

The Math:
The equation for discharging a cap is:

$$V_{cap} = V_{initial}(e^{\frac{-t}{RC}})$$

where
V_{cap} is the voltage across the capacitor at time t,
$V_{initial}$ is the voltage initially on the cap,
t is the time in seconds,
R and C are the values of the resistor and capacitor in ohms and farads, respectively.

The trick is to select values that insure the cap's voltage stays above V_{th}, the threshold at which the gate switches, till the switch stops bouncing. It's surprising how many opeople pick an almost random time constant. " figger sumpin like 5 msec". Most of the switches examined had bounce times well under 10 msec. Use 20 to be conservative.

Rearranging the time constant formula to solve for R (the cost and size of caps vary widely so it's best to select a value for C and then compute R) yields:

$$R = \frac{-t}{C\ln(V_{th}/V_{initial})}$$

The 7414 **hex inverter is a Schmitt Trigger with great input hysteresis**. The AHCT version has a worst case V_{th} for a signal going low of 1.7 volts. Let's try 0.1 µF for the capacitor since those are small and cheap, and solve for the condition where the switch just closes. The cap discharges through R_2. If the power supply is 5 volts (so $V_{initial}$ is 5), then R_2 is 185K. Of course, you can't actually *buy* that kind of resistor, so use 180K. But. the analysis ignores the gate's input leakage current. A CMOS device like the 74AHCT14 dribbles about a microamp from the inputs. That 180K resistor will bias the input up to .18 volts, uncomfortably close to the gate's best-case switching point of 0.5 volt. Change C to 1 µF and R_2 is now 18K.
$R_1 + R_2$ controls the cap's charge time, and so sets the debounce period for the condition where the switch opens. The equation for charging is:

$$V_{th} = V_{final}(1-e^{-t/RC})$$

Solving for R:

$$R = \frac{-t}{C\ln(1-V_{th}/V_{final})}$$

V_{final} is the final charged value - the 5 volt power supply. V_{th} is now the worst-case transition point for a high-going signal, which for our 74AHCT14 a peachy 0.9 volts. $R_1 + R_2$ works out to 101K. Figure on 82K (a standard part) for R_1.

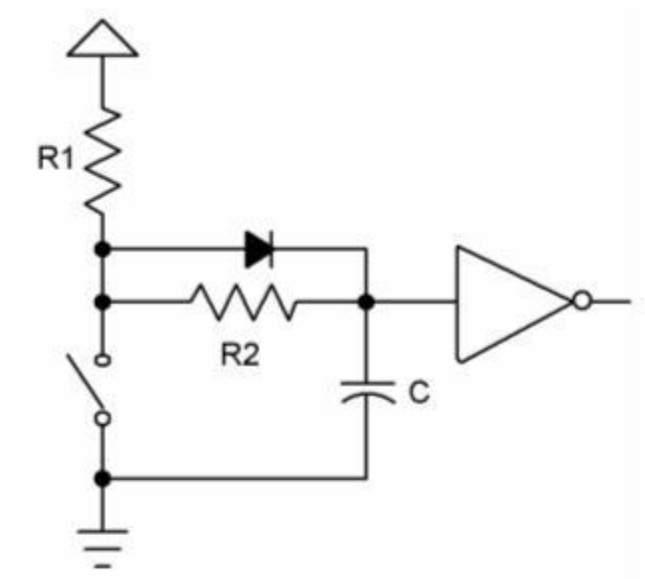


Figure 3: An RC debouncer that actually works in all cases

The diode is an optional part needed only when the math goes haywire. It's 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. In this case the diode forms a short cut that removes R_2 from the charging circuit. All of the charge flows through R_1. The previous equation still applies, except we have to account for drop across the diode. Change V_{final} to 4.3 volts (5 minus the 0.7 diode drop), turn the crank and R_1 pops out. Be wary of the components' tolerances. Standard resistors are usually ±5%. Capacitors vary wildly - +80/-20% is a common rating for electrolytics. Even small ceramics might vary ±30%.

Other Thoughts

- Don't neglect to account for the closed resistance of oddball switches. Some conductive elastomer devices exceed 200 ohms.
- Two of the elastomer switches didn't bounce at all; their output smoothly ramped from zero to +5 volts. The SR and RC debounce circuits are neither necessary nor effective. Better: run the switch directly into a Schmitt Trigger's input.
- Never connect an undebounced switch to the clock of a flip-flop. The random bounce hash is sure to confuse the device. A 74HCT74 has a max rise and fall time spec of 6 nsec - easily exceeded by some of the data I acquired from the 18 switches tested.
- The 74HC109 requires a minimum clock width of 100 nsec. I found pulses shorter than this in my experiments. Some of these parts, like from Philips, have a Schmitt Trigger clock input - it's a much safer part to use when connected to real-world events.
- Similarly, don't tie undebounced switches, even if Schmitt Triggered, to interrupt inputs on the CPU. Usually the interrupt pin goes to the clock input of an internal flip flop. As processors become more complex their datasheets give less useful electrical information; they're awash in programming data but leave designers adrift without complete timing specs. Generally we have no idea what the CPU expects as a max rise time or the min pulse width. Those internal flops aren't perfect, so don't flirt with danger by feeding them garbage.
- The MC14490 is a cool chip that consists of 6 debouncers. A datasheet is at http://engineering.dartmouth.edu/~engs031/databook/mc14490.pdf. But in August of 2004 Digikey wants $5.12 each for these parts; it's cheaper to implement a software debounce algorithm in a PIC or similar sub-$1 microcontroller.
- Always remember to tie unused inputs of any logic circuit to Vcc or ground.

Software Debouncers

Software debounce routines range from the utterly simple to sophisticated algorithms that handle multiple switches in parallel. But many developers create solutions without completely understanding the problem. Sure, contacts rebound against each other. But the environment itself can induce all sorts of short transients that mask themselves as switch transitions. Called EMI (electromagnetic interference), these bits of nastiness come from energy coupled into our circuits from wires running to the external world, or even from static electricity zaps induced by shuffling feet across a dry carpet. Happily EMI and contact whacking can be cured by a decent debounce routine. . . . but both factors do affect the design of the code.

Consider the simplest of all debouncing strategies: read the switch once every 500 msec or so, and set a flag indicating the input's state. No reasonable switch will bounce that long. A read during the initial bounce period returns a zero or a one indicating the switch's indeterminate state. No matter how we interpret the data (i.e., switch on or off) the result is meaningful. The slow read rate keeps the routine from deducing that bounces are multiple switch closures. One downside, though, is slow response. If your user won't hit buttons at a high rate this is probably fine. A fast typist, though, can generate 100 words per minute or almost 10 characters per second. A rotating mechanical encoder could generate even faster transitions.

But there's no EMI protection inherent in such a simple approach. An application handling contacts plated onto the PCB is probably safe from rogue noise spikes, but one that reads from signals cabled onto the board needs more sophisticated software, since a single glitch might look like a contact transition.

It's tempting to read the input a couple of times each pass through the 500 msec loop and look for a stable signal. That'll reject much or maybe all of the EMI. But some environments are notoriously noisy. Many years ago I put a system using several Z80s and a PDP-11 in a steel mill. A motor the size of a house drawing thousands of amps drove the production line. It reversed direction every few seconds. The noise generated by that changeover coupled *everywhere*, and destroyed everything electronic unless carefully protected. We optocoupled all cabling simply to keep the smoke inside the ICs, where it belongs. All digital inputs still looked like hash and needed an astonishing amount of debounce and signal conditioning.

Debounce Policy

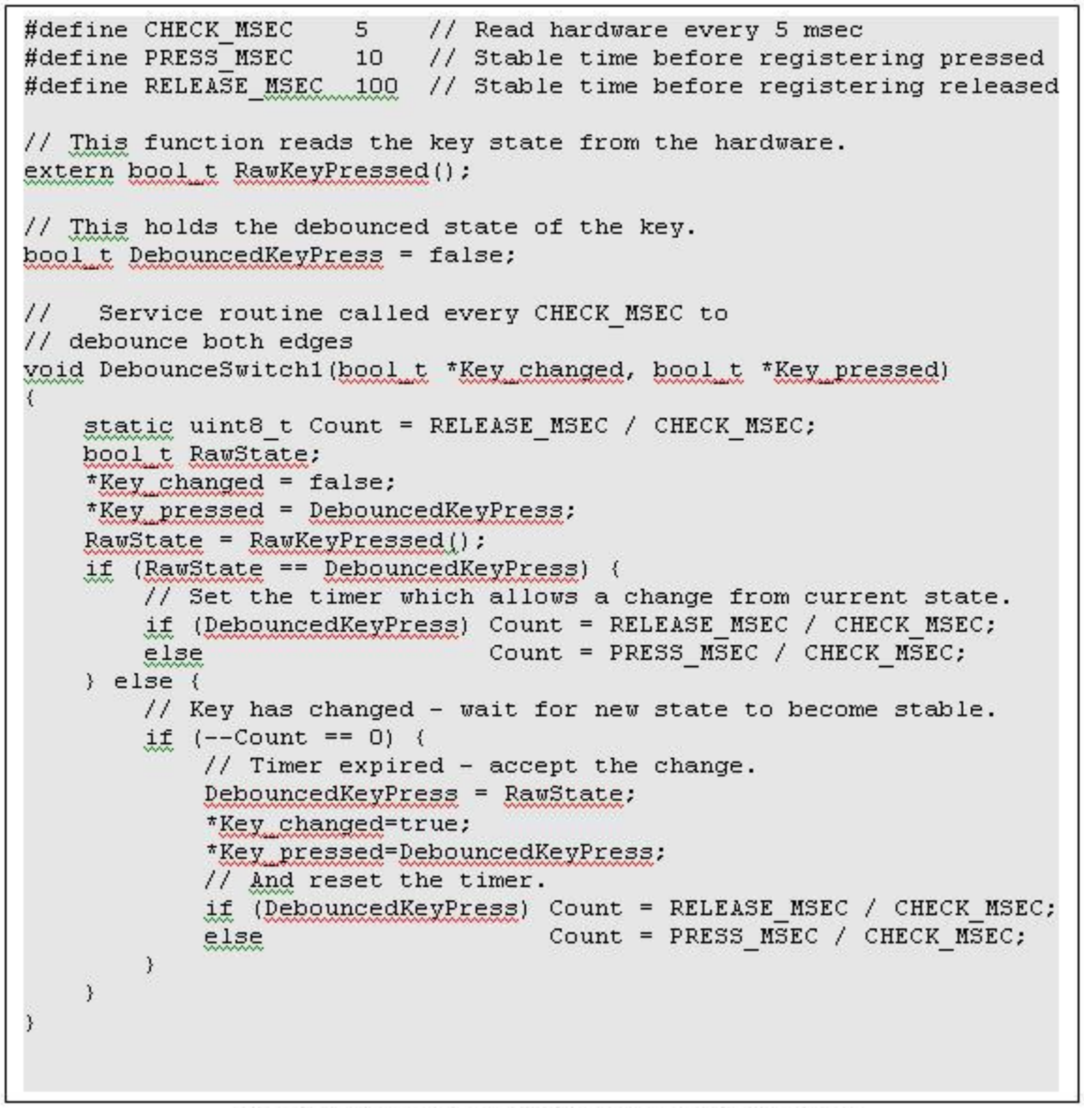
There are some basic constraints to place on our anti-contact-clacking routines. Minimize CPU overhead. Burning execution time while resolving a bounce is a dumb way to use processor cycles. Debounce is a small problem and deserves a small part of the computer's attention.

The undebounced switch must connect to a programmed I/O pin, never to an interrupt. Few microprocessor datasheets give much configuration or timing information about the interrupt inputs. Consider Microchip's PIC12F629 (datasheet at http://ww1.microchip.com/downloads/en/DeviceDoc/41190c.pdf). A beautiful schematic shows an interrupt pin run through a Schmitt Trigger device to the data input of a pair of flops. Look closer and it's clear that's used only for one special "interrupt on change" mode. When the pin is used as a

conventional interrupt the signal disappears into the bowels of the CPU, sans hysteresis and documentation. However, you can count on the interrupt driving the clock or data pin on an internal flip flop. The bouncing zaniness is sure to confuse any flop, violating minimum clock width or the data setup and hold times.
Try to avoid sampling the switch input at a rate synchronous to events in the outside world that might create periodic EMI. For instance, 50 and 60 Hz are bad frequencies. Mechanical vibration can create periodic interference. I'm told some automotive vendors have to avoid sampling at a rate synchronous to the vibration of the steering column.
Finally, in most cases it's important to identify the switch's closure quickly. Users get frustrated when they take an action and there's no immediate response. You press the button on the gas pump or the ATM and the machine continues to stare at you, dumbly, with the previous screen still showing, till the brain-dead code finally gets around to grumpily acknowledging that, yes, there IS a user out there and the person actually DID press a button.
Respond *instantly* to user input. In this fast-paced world delays aggravate and annoy. But how fast is fast enough?
I didn't know so wired a switch up to the cool R3000 starter kit Rabbit Semiconductor provides. This board and software combo seems targeted at people either learning embedded programming or those of us who just like to play with electronical things. I wrote a bit of simple code to read a button and, after a programmable delay, turn on an LED. Turns out a 100 msec delay is quite noticeable, even to these tired old 20/1000 eyes. 50 msec, though, seemed instantaneous. Even the kids concurred, astonishing since it's so hard to get them to agree on anything.
So let's look at a couple of debouncing strategies.

A Counting Algorithm

Most people use a fairly simple approach that looks for *n* sequential stable readings of the switch, where *n* is a number ranging from 1 (no debouncing at all) to seemingly infinity. Generally the code detects a transition and then starts incrementing or decrementing a counter, each time rereading the input, till *n* reaches some presumably safe, bounce-free, count. If the state isn't stable, the counter resets to its initial value.
Simple, right? Maybe not. Too many implementations need some serious brain surgery. For instance, use a delay so the repetitive reads aren't back to back, merely microseconds apart. Unless your application is so minimal there are simply no free resources, don't code the delay using the classic construct: for(i=0;i<big_number;++i);. Does this idle for a millisecond. or a second? Port the code to a new compiler or CPU, change wait states or the clock rate and suddenly the routine breaks, requiring manual tweaking. Instead use a timer that interrupts the CPU at a regular rate - maybe every millisecond or so - to sequence these activities.
Listing 1 shows a sweet little debouncer that is called every CHECK_MSEC by the timer interrupt, a timer-initiated task, or some similar entity.



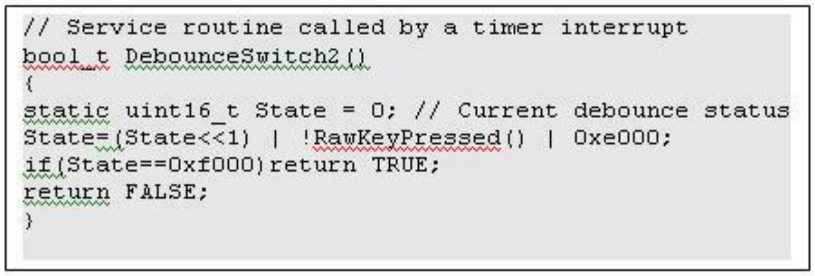
```
#define CHECK_MSEC    5    // Read hardware every 5 msec
#define PRESS_MSEC    10   // Stable time before registering pressed
#define RELEASE_MSEC  100  // Stable time before registering released

// This function reads the key state from the hardware.
extern bool_t RawKeyPressed();

// This holds the debounced state of the key.
bool_t DebouncedKeyPress = false;

//   Service routine called every CHECK_MSEC to
// debounce both edges
void DebounceSwitch1(bool_t *Key_changed, bool_t *Key_pressed)
{
    static uint8_t Count = RELEASE_MSEC / CHECK_MSEC;
    bool_t RawState;
    *Key_changed = false;
    *Key_pressed = DebouncedKeyPress;
    RawState = RawKeyPressed();
    if (RawState == DebouncedKeyPress) {
        // Set the timer which allows a change from current state.
        if (DebouncedKeyPress) Count = RELEASE_MSEC / CHECK_MSEC;
        else                   Count = PRESS_MSEC / CHECK_MSEC;
    } else {
        // Key has changed - wait for new state to become stable.
        if (--Count == 0) {
            // Timer expired - accept the change.
            DebouncedKeyPress = RawState;
            *Key_changed=true;
            *Key_pressed=DebouncedKeyPress;
            // And reset the timer.
            if (DebouncedKeyPress) Count = RELEASE_MSEC / CHECK_MSEC;
            else                   Count = PRESS_MSEC / CHECK_MSEC;
        }
    }
}
```

Listing 1: A simple yet effective debounce algorithm

You'll notice there are no arbitrary count values; the code doesn't wait for *n* stable states before declaring the debounce over. Instead it's all based on time and is therefore eminently portable and maintainable.
DebounceSwitch1() returns two parameters. Key_Pressed is the current debounced state of the switch. Key_Changed signals the switch has changed from open to closed, or the reverse.
Two different intervals allow you to specify different debounce periods for the switch's closure and its release. To minimize user delays why not set PRESS_MSEC to a relatively small value, and RELEASE_MSEC to something higher? You'll get great responsiveness yet some level of EMI protection.

An Alternative

An even simpler routine, shown in figure 2, returns TRUE once when the debounced leading edge of the switch closure is encountered. It offers protection from both bounce and EMI.



```
// Service routine called by a timer interrupt
bool_t DebounceSwitch2()
{
static uint16_t State = 0; // Current debounce status
State=(State<<1) | !RawKeyPressed() | 0xe000;
if(State==0xf000)return TRUE;
return FALSE;
}
```

Listing 2: An even simpler debounce routine

Like the routine in listing 1, DebounceSwitch2() gets called regularly by a timer tick or similar scheduling mechanism. It shifts the current raw value of the switch into variable State. Assuming the contacts return zero for a closed condition, the routine returns FALSE till a dozen sequential closures are detected.

One bit of cleverness lurks in the algorithm. As long as the switch isn't closed ones shift through State. When the user pushes on the button the stream changes to a bouncy pattern of ones and zeroes, but at some point there's the last bounce (a one) followed by a stream of zeroes. We OR in 0xe000 to create a "don't care" condition in the upper bits. But as the button remains depressed State continues to propagate zeroes. There's just the one time, when the last bouncy "one" was in the upper bit position, that the code returns a TRUE. That bit of wizardry eliminates bounces and detects the edge, the transition from open to closed.

Change the two hex constants to accommodate different bounce times and timer rates.

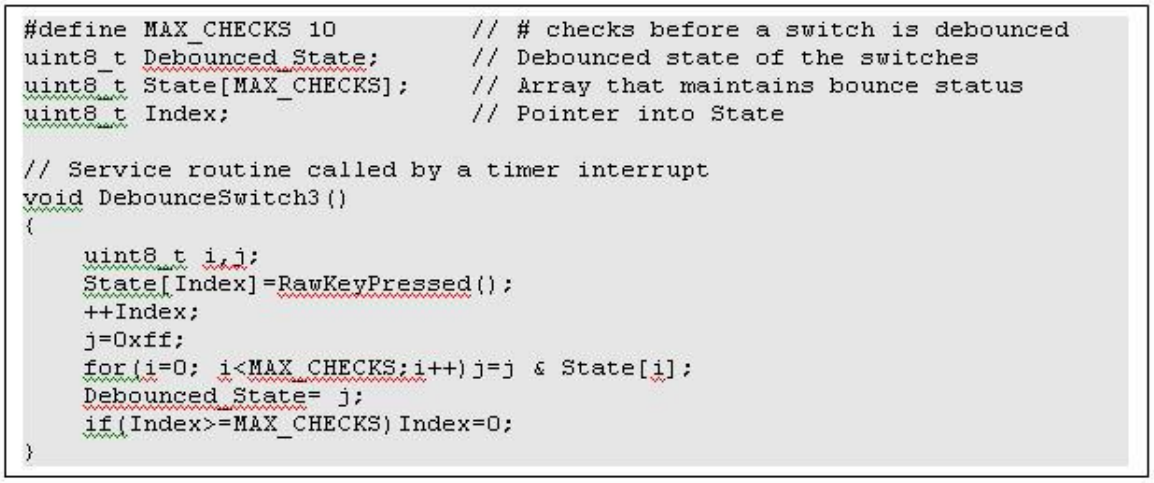
Though quite similar to a counting algorithm this variant translates much more cleanly into assembly code. One reader implemented this algorithm in a mere 11 lines of 8051 assembly language.

Want to implement a debouncer in your FPGA or ASIC? This algorithm is ideal. It's loopless and boasts but a single decision, one that's easy to build into a single wide gate.

Handling Multiple Inputs

Sometimes we're presented with a bank of switches on a single input port. Why debounce these individually when there's a well-known (though little used) algorithm to handle the entire port in parallel?

Figure 3 shows one approach. DebounceSwitch(), which is called regularly by a timer tick, reads an entire byte-wide port that contains up to 8 individual switches. On each call it stuffs the port's data into an entry in circular queue State. Though shown as an array with but a single dimension, a second loiters hidden in the width of the byte. State consists of columns (array entries) and rows (each defined by bit position in an individual entry, and corresponding to a particular switch).



```
#define MAX_CHECKS 10          // # checks before a switch is debounced
uint8_t Debounced_State;       // Debounced state of the switches
uint8_t State[MAX_CHECKS];     // Array that maintains bounce status
uint8_t Index;                 // Pointer into State

// Service routine called by a timer interrupt
void DebounceSwitch3()
{
    uint8_t i,j;
    State[Index]=RawKeyPressed();
    ++Index;
    j=0xff;
    for(i=0; i<MAX_CHECKS;i++)j=j & State[i];
    Debounced_State= j;
    if(Index>=MAX_CHECKS)Index=0;
}
```

Listing 3: Code that debounces many switches at the same time

A short loop ANDs all column entries of the array. The resulting byte has a one in each bit position where that particular switch was on for every entry in State. After the loop completes, variable j contains 8 debounced switch values.

One could exclusive OR this with the last Debounced_State to get a one in each bit where the corresponding switch has changed from a zero to a one, in a nice debounced fashion.

Don't forget to initialize State and Index to zero.

We prefer a less computationally-intensive alternative that splits DebounceSwitch() into two routines; one, driven by the timer tick, merely accumulates data into array State. Another function, Whats_Da_Switches_Now() ANDs and XORs as described, but only when the system needs to know the switches' status.

Summing up

All of these algorithms assume a timer or other periodic call that invokes the debouncer. For quick response and relatively low computational overhead weI prefer a tick rate of a handful of milliseconds. One to five msec is ideal. Most switches seem to exhibit under 10 msec bounce rates. Coupled with my observation that a 50 msec response seems instantaneous, it seems reasonable to pick a debounce period in the 20 to 50 msec range.

Hundreds of other debouncing algorithms exist. These are just a few of my favorite, offering great response, simple implementation, a no reliance on magic numbers or other sorts of high-tech incantations.