

Modern switch debouncing

1 INTRODUCTION

At the time of purely analog electronics, before the invention of logic and especially microprocessors, nobody cared or even knew about switch bounce¹. Over time, bounce became an issue, as digital electronics became ubiquitous, and especially as digital electronics became increasingly faster, able to track and act on switch bounce. So first let's briefly see what is switch bounce.

2 BOUNCE

When 2 pieces of metal come into contact, as in a switch, they either make the contact quickly, being acted upon by spring action, or on the contrary, they slide slowly into position as the switch is activated.

In the former case, the “violent” spring action will make the blades of the switch jump back a minute amount, opening the contact just made, until the contact settles, usually after a few such jumps or bounces.

In the case of sliding contacts (less prevalent nowadays as rotary switches fall out of favors), the (relatively) slowly sliding contacts offer plenty of opportunities for poor or intermittent contacts until settling in position.

2.1 How long does bounce last

Well... It depends. A good switch, meaning in our context, one with little bounce, may have bounce from less than 1 ms to a few ms, say up to 5 or 10 ms. On the other hand of the bounce spectrum, we could see up to 100 ms, possibly more, depending on the construction and environment of the switch. Not so long ago, I worked on a commercial project that had to debounce switches made of blades, that were pushed by slowly rotating cams, in a dirty (dusty, greasy and vibrating) environment that were expected to have bounce in the 100 ms time frame, possibly more².

3 DEBOUNCING

When it appeared that debouncing was required, software did not exist, at least not in a way usable for such menial tasks as debouncing, so hardware solution had to be used.

3.1 Hardware debouncing

Thus for a long time, debouncing was better done in hardware, using a number of techniques.

-
1. I should know, being born in 1949, and dabbling in electronics since about 1961.
 2. I was not able to measure this bounce but had to make sure the debouncing could cope with this bounce and possibly even longer. It is interesting to note that this machine had been designed at the time of germanium transistors discrete logic and was being updated to a state of the art microprocessor control.

One of these techniques rely on waiting for some time after a switch closure is detected, to make sure that after this time the switch is still closed. The drawback of this is that the switch closure can only be acknowledged after a certain delay. While a delay of, say, up to 20 ms is acceptable and largely undetectable by humans, more than that becomes annoying in some cases. Furthermore, if you design the debounce for 10 ms, as the switch ages, there is no guarantee that it will not one day bounce to 12 ms, creating problems with your application. The following figure shows such a debounce circuit.

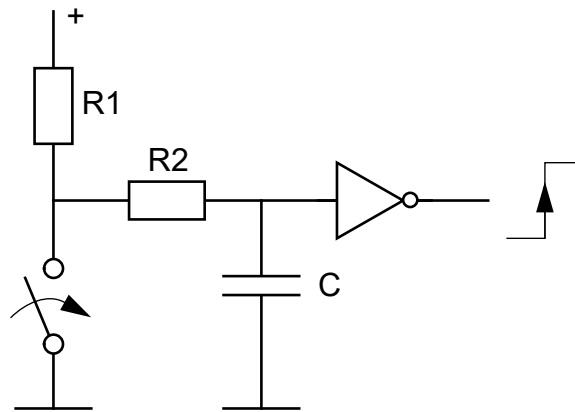


Figure 1. A debounce circuit based on a RC cell and a Schmitt trigger

Initially, with the switch open, the input to the inverter (preferably a Schmitt trigger) is high and therefore the output is low. Once the switch closes, the capacitor will “slowly” discharge through R2, until the voltage on the capacitor reaches the trigger level of the inverter and the output will go high, thus “erasing” any bounce with a proper choice of R and C. When the switch opens, the capacitor charges through R1 + R2, until the trigger level of the inverter is reached and the output will go low. Thus the output is a delayed version of the switch action with the bounce “erased” by the RC cell. This delay can be annoying, and the RC cells, one per switch, may be costly. Besides it is a clumsy solution nowadays as we will see below.

A modern implementation of the circuit might omit the inverter and use a microprocessor input, especially if it has a Schmitt trigger feature.

There are other “old-fashioned” debouncing circuits, one of them, shown Figure 2 at left, is the one-shot or monostable. The one-shot uses a 555 or another monostable implementation. The advantage of this circuit is that the output goes high as soon as the switch closes, so there is no delay. The circuit ignores any bounce for the duration of the pulse, effectively debouncing the switch. The drawbacks of this circuit are that it only detects switch closures (ignores switch openings) and it requires several peripheral components (2 capacitors and 2 resistors). Furthermore if the switch is released after the end of the output pulse, it could trigger another pulse through its bounce.

A better circuit is shown on Figure 2 at right. It is very reliable, but requires a double pole switch (plus 2 gates and 2 resistors). As the switch is flipped, Q will go high (and \bar{Q} low) immediately, without delay, ignoring any switch bounce, and conversely, when the switch is returned to its original position, Q will go low while \bar{Q} returns high. There are ICs that provide a quad implementation of this function, for example the 74LS279.

Another type of IC providing debounce is the MC14490 Hex debouncer, based on a shift register, that operates in a similar way as the one described below in section 3.2 below with the same drawbacks.

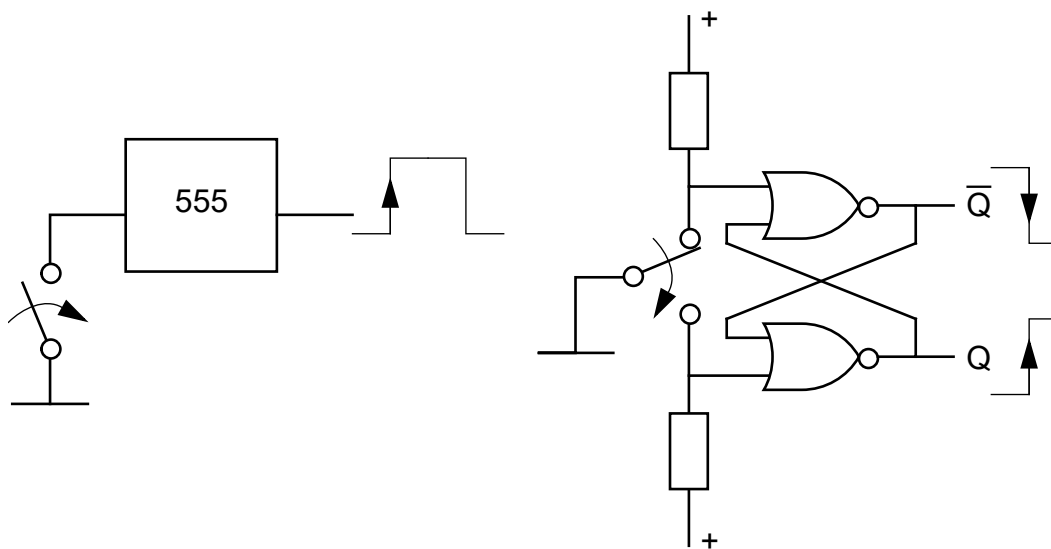


Figure 2. At left a monostable debounce circuit, At right a 2-NOR gates debounce circuit.

3.2 Crude software debounce

Many of the techniques in software published for debouncing are rather crude, judging by what is found on the web¹. Here are some of the techniques I found.

Looking at a switch several times until a certain number of readings concur might be one solution, but it is unreliable in terms of timing (we cannot be sure how long it will take to have a stable reading), so there is unpredictable delay in the output. As mentioned before, if a small delay may be acceptable, no delay is definitively preferable.

Another option is to check after a certain, long enough delay, if the switch is still in its newly detected state and if so accept the new state. Again there is a delay.

All in all, none of the solutions typically available provide a reliable debounce without delay, however looking at the hardware two-gate solution on Figure 2 at right, there is a hardware solution that can inspire a better software solution. More on this in a while.

4 SWITCH ACTION

Let's take a good look at what a switch does in modern circuits and how it does it.

4.1 Four states of a switch

Rather than speak of low or high level, let's use the words opened and, closed to describe the 2 static states of a single one-pole switch, and make to describe the opened to closed transition and break to describe the closed to opened transition. It might not be obvious at first but a switch has these four states:

- It can be opened
- It can just have had a opened to closed transition (make)

1. Obviously, commercial solutions exist that are more sophisticated and provide "commercial grade" results but these are not readily available.

- It can be closed
- It can just have had a closed to opened transition (break)

These distinctions are important, because one might want to take some action on the **make** transition, have some different behavior while the switch is in a closed state, and a different action when the switch opens (makes a **break** transition). Usually no particular action is associated to the opened state.

For instance, pushing a switch (**make**) might be used to turn on a feature or increment a counter. While the switch is in closed state, we might want to increment that counter increasingly faster after a short while, and immediately stop this fast increment when the switch is released (**break**).

If the switch is the left button of a mouse, we may want to select an object when the button is pushed (**make**), drag it while the button is closed, and drop it when the button is released (**break**).

So obviously proper and immediate identification of these various stages in the operation of a switch is crucial for the unfolding a modern software.

5 A VERSATILE SOFTWARE DEBOUNCER

Now comes the time to describe the proposed debounce scheme.

A word of warning first: I have worked with microprocessors for the first time in 1976 using a National Semiconductor SC/MP¹ on a board with 1 k of EEPROM and 256 bytes or RAM. Later on, I worked with early 8-bits microprocessors that were no more generous in RAM with 64 to 128 bytes and less than 2 k EPROM. Needless to say that I was (and still am) aware of the need to be efficient with RAM. Hence the routines described below make relatively efficient usage or RAM.

5.1 Debouncing principle

Thinking of the 2 NOR gates above, we want to have a circuit that reacts immediately to switch closure and opening, and that once triggered disregards any bounce.

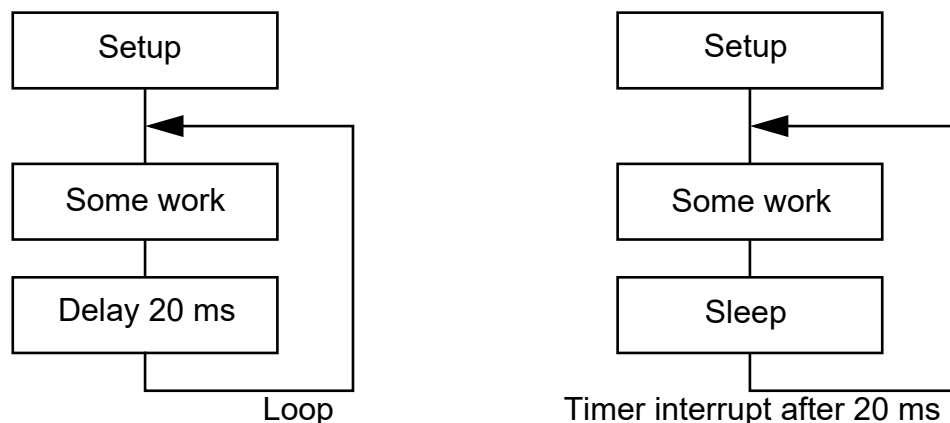


Figure 3. The main loop, that repeats every 20 ms, can be based on a delay or on a periodic timer interrupt.

1. https://en.wikipedia.org/wiki/National_Semiconductor_SC/MP

At this stage we still need to briefly look at the structure of a typical program. Usually we first have a setup routine, then the program falls into a loop that is executed repeatedly. The rate of the repeat will normally not be dependant on actual processing being performed, but on a delay or better a periodic interrupt after the processor has gone to sleep for this loop. This is outlined in the flowcharts of Figure 3. Note that the 20 ms loop time is entirely arbitrary and depends on factors to be discussed later.

With this in mind, let's consider that "Some work" involves looking at the pin connected to the switch(es) to be debounced. Let's look only at one switch. We assign a byte to each switch. Let's call ours `one_sw`. At rest (with the switch in open state), the byte is 0x00 (Figure 4, Top).

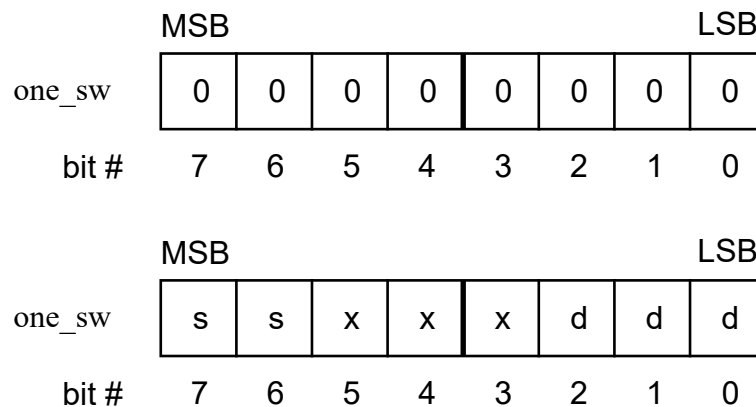


Figure 4. Top: The `one_sw` byte is all zeroes when idle. Bottom: Assignment of the various bits within the byte.

On Figure 4. Bottom, s-bits are a 2-bit shift register reflecting one of 4 states of the switch. x-bits are unused. d-bits are a 3 bit counter. The counter is used to decout a debounce time. We do not necessarily use all 3 bits, with a 20 ms loop time, this can count up to $20 \cdot (2^3 - 1) = 140$ ms. For a faster loop time, we may consider using more bits if required (up to bit 4 included).

At each loop, we look at the pin and if nothing happens (the switch is not pushed, `one_sw` remains at 0x00¹.

When the switch is activated, the level read on the pin is pushed onto the MSB (bit 7), and the debounce time required set in the d-bits. For example for a 60 ms debounce time, `one_sw` becomes:

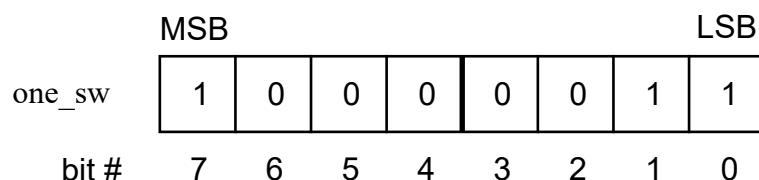


Figure 5. A make transition has just been detected, reflected in bit 7 and the debounce time loaded in the d-bits.

-
1. In reality since the switch is likely connected between a pin and ground, the pin reads 1 when the switch is opened and 0 when the switch is closed. For the sake of simplicity, this reading is inverted, leading to a low when the switch is open and a high when the switch is closed.

Two things happen at this stage, we can act on the make transition just detected, and at each loop we will decrease the d-counter and while it is not zero, totally ignore the pin. This last point effectively debounces the switch for (in our case) 60 ms.

As for the make transition detected we must either use it if needed and remove it (so it is not acted upon a second time) or should we want to ignore it, simply remove it.

We know that the switch is pushed (and perhaps bouncing, but this is irrelevant to us), so to acknowledge the make (to remove it), we set the 2 s-bits to '11':

	MSB							LSB
one_sw	1	1	0	0	0	0	1	1
bit #	7	6	5	4	3	2	1	0

Figure 6. After the make transition has been acted upon (if required), we set the top 2 bits to '11', signaling a steady state of closed switch.

We see here that a static closed switch can be identified by a '11' in bits 6 and 7.

At the next loop, we shift bit 7 to position 6, discard the bit in position 6, and decrease the d-counter by 1:

	MSB							LSB
one_sw	1	1	0	0	0	0	1	0
bit #	7	6	5	4	3	2	1	0

Figure 7. Decrementing the debounce time by 1 for each loop.

We repeat this for the next 2 loops, the d-counter goes to '001', then '000'. At the next loop, the d-counter being 0, we look again at the pin, since the bounce is finished. Bit 7 is shifted to position 6 and the state of the pin is placed on bit 7 one_sw looks like this:

	MSB							LSB
one_sw	1	1	0	0	0	0	0	0
bit #	7	6	5	4	3	2	1	0

Figure 8. After the debounce time, we can resume looking at the pin.

indicating the the switch is still pushed. This process is repeated for each loop until the switch is released. At this point the reading is shifted onto bit 7 and the debounce counter loaded which leads to this:

	MSB							LSB
one_sw	0	1	0	0	0	0	1	1
bit #	7	6	5	4	3	2	1	0

Figure 9. We just detected a break transition, signaled in the 2 top bits, and loaded the debounce time in the d-bits.

The implication is that the s-bits now reflect a break transition, which can or not be acted upon, but must be acknowledged by setting the s-bits to '00'. As for the d-bits, they will prevent looking at the pin until the debounce time has been counted out.

The routine performing the task just described is shown below¹.

```
void readkeys()
{
  byte reading;
  // one switch
  if (!(one_sw & 0x07)) {reading = !digitalRead(ONSW); // if not debouncing, read input
    if (reading == !(one_sw & 0x80)) { // Has there been a pin change
      one_sw >>= 1; one_sw &= 0x40; one_sw |= DEBOUNCE; // Read pin and merge debounce
      if (reading) {reading = 0x80;} else {reading = 0;} // place the new bit in the MSB
      one_sw |= reading; // set new State
    } // if reading
  } // if !one_sw
  else {one_sw--;} // decrease debounce
}
```

Figure 10. The read_key() routine, looking here only at one_sw.

The first line: `if (!(one_sw & 0x07)) {reading = !digitalRead(ONSW);` looks at the switch if the debounce counter is not active. As explained in the footnote on page 5, we are really looking at the inverse of the pin state. ONSW is the #define name of the pin. In the examples given, this is pin 2.

The line: `if (reading == !(one_sw & 0x80)) {` makes sure the next bloc is executed only if the state of the switch is different than the previous one.

If the switch has changed, `one_sw >>= 1; one_sw &= 0x40; one_sw |= DEBOUNCE;` shifts bit 7 in position 6, removes bit 6 that just got moved in (unused) position 5, and sets the d-bits to the #define DEBOUNCE value (in our examples 3).

When we read the port (`{reading = !digitalRead(ONSW);`), the bit read was placed in position 0 (of byte reading). We need it in position 7 to place in one_sw. This is what the line `if (reading) {reading = 0x80;} else {reading = 0;}` does. Note that "else {reading = 0;}" can be omitted since if reading is not true (does not contain a 1), then it is all zeroes.

Then this bit is placed in bit 7 of one_sw: `one_sw |= reading;`

Should the debounce counter not being zero, none of the above will occur, and the last line: `else {one_sw--;} will just decrease the debounce counter.`

Typically there will be one such routine per switch to be looked at and debounced.

With this routine we have a way to:

- Debounce a switch
- Know immediately if a switch has been pushed or released
- Have a way to know the static state of a switch (closed or opened).

To summarize:

- 00: the static state of the switch is open,
- 10: a make just happened,
- 11: the static state of the switch is closed,
- 01: a break just happened.

1. This code is Arduino code, it can easily be changed to standard C.

5.2 Use of this routine

Typically, this routine is placed in the main() loop. The main() loop will likely do some useful work, then it can look at the switches, act on what has been detected, and reset any transition since they have just been acted upon. Finally more useful work can be performed if needed. This is shown in the code below.

```
void loop() {           // this is the main loop
  ... some work
  readkeys();          // read and debounce keys
  rules();             // apply the rules on the above
  ... some work
  clear_trans();
  delay(20);          // 20 ms loop
} // loop
```

Figure 11. The main() loop, making use of the rules() routine.

readKeys(); is the routine we just described.

rules(); is the routine that acts upon the status of the switch just detected.

clear_trans(); is the routine that removes transitions since they have just been taken care of and need to be reset so as not to be still active on the next loop.

rules() makes use of any of the 3 following subroutines that return TRUE or FALSE,

mk_trans(one_sw) which returns TRUE if the top bits of one_sw are '10'

bk_trans(one_sw) which returns TRUE if the top bits of one_sw are '01'

state_sw(one_sw) which returns TRUE if the top bits of one_sw are '11'

We typically have no need to check for '00' (opened state).

We will look further down at actual uses for these.

Finally, we need to clear transitions:

```
void clear_trans() // clear all trans since we acted on them
{
  if ((one_sw & 0xc0) == 0x40) {one_sw &= 0x0f;}
  if ((one_sw & 0xc0) == 0x80) {one_sw |= 0xc0;}
}
```

Figure 12. The clear_trans() routine - the final part of our debounce scheme.

The first line clears break transitions, while the second one clears make transitions. There needs to be one such pairs of lines for each switch.

There are 2 degrees of flexibility in this debounce routine. Because of the number of free bits in the switch byte (5 lower bits), the debounce counter can go up to $(2^5 - 1) = 31$ loops.

Here we arbitrarily set the loop time at 20 ms, but it could be anything else reasonable. Proper values might range from 5 ms to 50 ms. If the loop time has been set at 10 ms, for example and the debounce time is not expected to exceed 100 ms, the debounce count would be set at $0x0a = 10d$.

6 RULES

We now have a mean of debouncing a switch, and know its status at any time. We will first look at simple actions we can take from this knowledge, and then more complex functions. Ultimately, we will be able to tailor the rules section to any need.

6.1 Simple rules

We will look at the concrete example of toggling the LED of an Arduino board for each closure of our switch. A more telling demonstration would be the increase and display of a counter. This is left to the reader as it is an easy modification of the routine shown here.

We already have a `main()` section that reads switches, calls a `rules()` routine and clears transitions. As before we assume that the switch is on `#define PIN ONSW`, and the LED is on `#define PIN LED`. On Arduino boards, the LED is on pin 13. Lets look at the `rules()` routine.

```
void rules()
{
  // toggles LED on mk_trans
  if (mk_trans(one_sw)) {
    if (digitalRead(LED)) {digitalWrite(LED, FALSE);} else {digitalWrite (LED, TRUE);}
  }
}
```

Figure 13. The `rules()` routine to act on a make transition of `one_sw` and toggle a LED.

and let's have a look at the `mk_trans()` subroutine:

```
unsigned char mk_trans (unsigned char button) // true if a button just got pressed
{
  if ((button & 0xc0) == 0x80) {return TRUE;} else {return FALSE;}
}
```

Figure 14. The `mk_trans()` routine returns TRUE if a make transition just occurred.

As expected, we just look for the '10' combination in the 2 top bits of the argument.

Should we be interested in the break transition, we would have called the `bk_trans()` subroutine:

```
unsigned char bk_trans (unsigned char button) // true if a button just got released
{
  if ((button & 0xc0) == 0x40) {return TRUE;} else {return FALSE;}
}
```

Figure 15. The `bk_trans()` routine returns TRUE if a break transition just occurred.

This sketch (`simple_rule.ino`) is available at the link at the end of this document. It is written for an Arduino Nano.

6.2 Complex rules

Modern switch usage, makes use of long presses, and double presses to mention only these. Here we present three schemes using more complex use of switches. To time the needed delays, we will make use of counters in various combinations, and for simplicity, they will always be similar to our `one_sw` byte. The 2 top bits will indicate the state of the counter, and the remaining bits are the counter itself. Since the 2 top bits indicate the state of the counter, we can use the same subroutines as for the switches in order to deter-

mine the state of the counter. Here is an examples, timing 1 s, assuming a loop time of 20 ms.

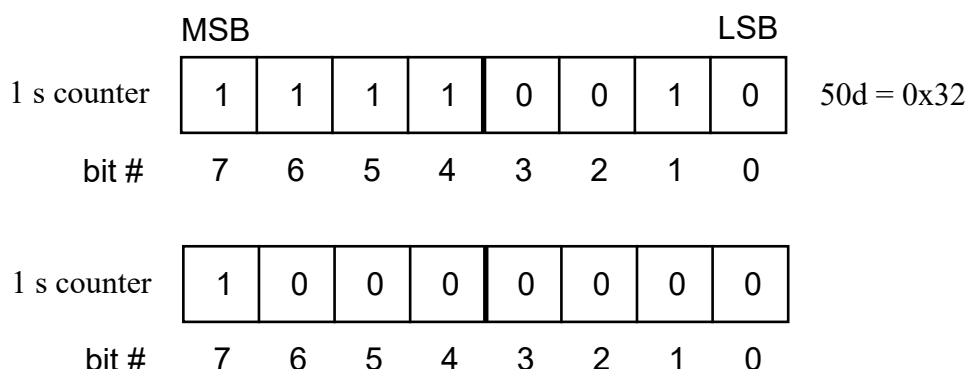


Figure 16. Top: counter setup to show that the counter is active (steady state on) via the 2 top bits, the remaining bits are the counter itself. Bottom: the counter has reached zero, so we signal a make transition in the 2 top bits.

In the top figure, the counter has just been setup. The counter time is set at 50d = 0x32, and the 2 top bits are set as counter active (similar to closed state in a switch).

In the bottom figure, the counter has decreased to 0, signifying that it expired, so the 2 top bits reflect that state (similar to the make transition on a switch)¹.

Of course, we will need a dedicated decoupling routine, that will look for each loop at all such counters, decrease them if active, setting the 2 top bits when they expire. We'll call it `dec_s_cntrs()`.

Finally, the `clear_trans()` routine will also be required to clear these break transitions to '00' (counter inactive). This is shown in the example below.

a) Double click detection

For a double click to be properly detected, the second press must occur within a short, specified time after the first. Here we will use a counter called `s06` to count up to 0.6 s. With our 20 ms loop time, this is a count of 30d = 0x1e. We therefore will load our counter with 0b11011110 = 0xde. This will be in a `#define SEC06`.

This is done when the switch's first **make** is detected. At each loop will now look for a second **make**. If and when it occurs, we check if the counter `s06` is still active (we are still within the 0.6 s window), and if so, we do something. Here we will only toggle the LED on a second click that occurs at the right time, ignoring the first click.

```
void loop() { // this is the main loop
  dec_s_cntrs(); // decrement s counters
  readkeys(); // read and debounce keys
  rules(); // apply the rules on the above
  clear_trans(); // 20 ms loop
  delay(20);
} // loop
```

Figure 17. The `main()` routine, showing the use of the `dec_s_cntrs` routine.

1. This is arbitrary, we could have chosen to signal a break transition, we just have to be consistent when checking the state of the counter.

This is the main loop, it looks similar to what we have previously seen, but there are a couple of changes. First we added the `dec_s_cntrs()` routine which decrements `s06` (and any other similar counter first) and sets the top bits appropriately when it expires. Here it is:

```
void dec_s_cntrs()
{
  if (s06 & 0x3f) {s06--;
    if (s06 == 0xc0) {s06= 0x80;}
  }
}
```

Figure 18. The `dec_s_cntrs()` routine, looking only at `s06`.

The second change is that now our `rules()` routine starts and looks at `s06`:

```
void rules()
{
  // acts on mk_trans
  if (mk_trans(one_sw)) {
    if (state_sw(s06)) {s06 = 0;
      if (digitalRead(LED)) {digitalWrite(LED, FALSE);}
      else {digitalWrite (LED, TRUE);}}
    else {s06 = 0xde;}
  } // if mk_trans
}
```

Figure 19. The `rules()` routine, that acts on the make transition of `one_sw` to toggle the LED.

The first line checks if we just had a **make** of the switch. If we did, then we look at `s06`. If it is active (checked by `state_sw()`), it toggles the LED and resets `s06`, otherwise, it starts `s06`, and we do nothing else with the **make** of the switch.

It makes use of the function:

```
unsigned char state_sw (unsigned char button) // returns true if the button is pressed
{
  if ((button & 0xc0) == 0xc0) {return TRUE;} else {return FALSE;}
}
```

Figure 20. The `state_sw` routine returns TRUE if the switch is in closed state. In the present case, it looks at counter `S06`.

This sketch will effectively toggle the LED only at the second click of a valid double click.

This sketch (`double_click.ino`) is available at the link at the end of this document. It is written for an Arduino Nano.

b) Performing an action if a switch is held more than 3 s

We now want to detect if a switch is held for at least 3 s.

Here again we will use a counter as before, but perform the action (toggle the LED) only if the switch is held for at least 3 s. The little problem here, is that our counter only contains 6 bits. With 20 ms, this is a maximum time of $(2^6 - 1) \cdot 20 \text{ ms} = 1.26 \text{ s}$. We do not want to use a 2 byte counter as this will complicate our scheme needlessly, instead, we prescale the counter. Prescaling by 4 looks good and allows counting up to $\sim 5 \text{ s}$, plenty for our purpose. For that we use a counter called `by4`. Four, being a multiple of 2 is a nice number, it allows us to test `by4` for a count of 4^1 by checking that bits 0 & 1 are '00'. So let's use a counter called `s30`. For a 3 s delay we would need a counter by 150, but pres-

1. So many fours and fors!

caled by 4, the counter need only be by $37d = 0x25$. So with the 2 top bits at '11', we will load $0b11100101 = 0xe5$ in s30. This is #define SEC30.

We'll also add a counter by4 that will also be managed in dec_s_cntrs().

```
void dec_s_cntrs()
{
  by4++;
  if (!(by4 & 0x03)) {
    if (s30 & 0x3f) {
      s30--;
      if (s30 == 0xc0) {s30 = 0x80;}
    } // if (s30 & 0x3f)
  } // if !(by4 & 0x03)
}
```

Figure 21. A modified dec_s_cntrs() routine, making use of a prescaler by 4 to extend the timing capabilities of a counter (here s30).

We increase the counter by4 and look at the 2 lower bits. If they are '00', we just had 4 counts and we can work on s30.

If s30 is counting, decrease it. If it reached zero, let's signal a make transition¹.

The rules() routine now looks like this:

```
void rules()
{
  // toggles LED on a 3 s hold of the switch
  if (mk_trans(one_sw)) {s30 = SEC30;}
  if (bk_trans(one_sw)) {s30 = 0;}
  if (mk_trans(s30)) {if (digitalRead(LED)) {digitalWrite(LED, FALSE);}
                    else {digitalWrite(LED, TRUE);}}
}
```

Figure 22. The rules() routine that toggle the LED only if the switch is held for 3 s.

If we detect a make transition on the switch, we start the 3 s counter. At this stage the 3 s counter cannot already be running since for a break transition of the switch, it will be reset.

If we detect a make transition of s30 (s30 has expired) we act on it by toggling the LED.

Finally, the make transition of s30 will be cleared in clear_trans().

```
void clear_trans() // clear all trans since we acted on them
{
  if ((one_sw & 0xc0) == 0x40) {one_sw &= 0x0f;}
  if ((one_sw & 0xc0) == 0x80) {one_sw |= 0xc0;}
  if ((s30 & 0xc0) == 0x80) {s30 = 0;}
}
```

Figure 23. The clear_trans() routine updated to also clear s30.

This will effectively toggle the LED only after the switch has been held for 3 s. This sketch (three_sec.ino) is available at the link at the end of this document. It is written for an Arduino Nano.

c) Holding the switch will count increasingly faster

Our last example will show how one would use the switch to increment a counter. A pressure of the switch would increment the counter by 1. If the switch is held, it would automatically increase the counter again after 0.8 s. After for example 2 such increments,

1. See footnote page 10.

the counter would increase after 0.4 s a couple of times, then every 0.2 s, and finally even faster until the button is released.

The structure of the program is identical to the examples above. Here we make use of a counter similar to s30 in the previous example, except that its initial timing will be 0.8 s, and set to shorter and shorter values as we hold the switch for a longer time.

Let's call this counter `one_cnt`, and we will need another counter called `speedcnt` to keep track of the number of increases already performed. For the reloading of the counter, we'll use some constants called `SEC08` for 0.8 s, `SEC05` for 0.5 s, `SEC02` for 0.2 s and `SEC01` for 0.1 s. All these constants will have the 2 top bits as '11', and the lower bits to reflect the timing required (in increments of 20 ms).

Constant	delay (dec)	binary	hex
SEC08	40	11101000	0xe8
SEC04	20	11010100	0xd4
SEC02	10	11001010	0xca
SEC01	5	11000101	0xc5

The routine `inc_cnt()` is supposed to increase our counter, but we will replace it in the sketch provided by a blinking of the LED. Here is the `rules()` routine:

```
void rules()
{
  // one switch
  if (mk_trans(one_sw)) {inc_cnt(); one_cnt = SEC08;}
  if (bk_trans(one_sw)) {one_cnt = 0; speedcnt = 0;}
  if (mk_trans(one_cnt)) {inc_cnt(); speedcnt++; one_cnt = SEC01;}
  if (speedcnt < 7) {one_cnt = SEC02;}
  if (speedcnt < 4) {one_cnt = SEC04;}
  if (speedcnt < 2) {one_cnt = SEC08;}
}
```

Figure 24. The `rules()` routine to increment a counter increasingly faster.

The line `if (mk_trans(one_sw)) {inc_cnt(); one_cnt = SEC08;}` is executed if a make transition has been detected. It increments our counter (blink the LED), and starts `one_cnt` for 0.8 s.

The next line `if (bk_trans(one_sw)) {one_cnt = 0; speedcnt = 0;}` cancels `one_cnt` and `speedcnt` if the switch is released, ending the repeat process whether started or not.

The remaining lines increment our counter for each expiry (make) of `one_cnt`, increment `speedcnt` and set `one_cnt` to 0.1 s. Depending on the value of `speedcnt`, `one_cnt` will be loaded with longer values, keeping in mind that at each expiry of `one_cnt`, `speedcnt` increases, loading `one_cnt` with shorter values. This is where the increasingly faster increment of our counter occurs.

We also need to adapt the `dec_s_cntrs()` and `clear_trans()` routines to work on `one_cnt`. This sketch (`faster.ino`) is available at the link at the end of this document. It is written for an Arduino Nano. It only acts on the LED (which will blink increasingly faster), but the reader is encouraged to use a real counter if it can be displayed on a 1 or 2 digit display.

The sketch (faster2.ino) shows another version of the same scheme, using parameters for the fastest and slowest speeds, as well as for the speed increment. See #define SECSL, SECFA and DECSP. :

```
void rules()
{
  // one switch
  if (mk_trans(one_sw)) {inc_cnt(); reload = SECSL; one_cnt = 0xc0 | reload;}
  if (bk_trans(one_sw)) {one_cnt = 0;}
  if (mk_trans(one_cnt)) {
    inc_cnt(); if(reload >= (SECFA + DECSP)){reload -= DECSP;} one_cnt = 0xc0 | reload;}
}
```

Figure 25. This scheme allows a more progressive speed increase (although this is not really noticeable to the user) and is a cleaner solution.

7 SERVICING MULTIPLES BUTTONS

So far we have only been concerned with just 1 switch. However when servicing several switches, the debounce routine can be created as a subroutine.

```
void readkeys()
{
  byte reading;
  // one switch
  reading = !digitalRead(ONSW); one_sw = debounce(reading, one_sw); // read input bit and call debounce
}

unsigned char debounce(unsigned char reading, unsigned char swbyte)
{
  if (!(swbyte & 0x07)) { // if not debouncing
    if (reading == !(swbyte & 0x80)) { // Has there been a pin change
      swbyte >>= 1; swbyte &= 0x40; swbyte |= DEBOUNCE; // Read pin and merge debounce
      if (reading) {reading = 0x80;} // place the new bit in the MSB of reading
      swbyte |= reading; // set new State
    } // if reading
  } // if !one_sw
  else {swbyte--;} // decrease debounce
  return swbyte;
}
```

Figure 26. The debounce routine is here a subroutine called from the readkey routine. This example is trivial since we only have 1 button.

The sketch for this example is simple_rule2.ino.

8 OTHER INPUTS

A similar scheme can be used for inputs that do not need debounce. For instance a 1 Hz square wave from a RTC¹ in a clock circuit or a clean (not bouncing) signal from another part of the circuit. The read_key() routine can be used for these signals as well, but no debounce needs to be performed.

```
// SQ (1 Hz) input
reading = !digitalRead(SQ);
if (reading == !(sqlhz & 0x80)) { // this compares the bit just read with the MSB
  sqlhz = sqlhz >> 1; sqlhz = sqlhz & 0x40; // shift the old MSB into MSB-1, keep only MSB-1
  if (reading) {reading = 0x80;} // place the new bit in the MSB of reading
  else {reading = 0;}
  sqlhz = sqlhz | reading; // set new State
} // if reading
```

Figure 27. Reading an input when debouncing is not needed

1. Real Time Clock.

In the case of our 1 Hz signal, we need to look at the **make transition**, so as to increase counters for each second. No need for debounce, so only the top 2 bits of a status byte are used. While the signal is low, the byte is 0x00, upon a **make transition**, the top 2 bits become '10', so the byte looks like 0x80, and on the next loop, with the 1 Hz signal high, the top 2 bits become '11' for a byte of 0xc0. In this case there is no need to clear a transition, it clears itself. The point here is that we can use the same test routines as before to detect a **make transition**, a **break transition** or a **steady state**.

Finally, for some signals (non bouncing) we may be concerned only with levels, in this case, no need to complicate the program, we just look at the state of the pin where the signal is applied.

9 CONCLUSION

This document has described a method to perform switch debouncing. This method has been shown to have all the features needed for a modern debounce scheme, namely reliability, immediate response and parametrization. It also lends itself to easy adaptation to complex switch operation. Finally it does not require external components¹ and has a small software footprint.

This scheme has shown itself to be usable for many timing tasks in relation to switch servicing, allowing complex operation in rather transparent and simple ways.

Arduino sketches have been given for the practical examples described, which will aid the reader in understanding, and provide a starting point for his own implementation.

Downloads can be found here: [arduino.zip](#)

Olivier PILLOUD - Dec. 2019

1. Pullups are required on the switch inputs, but these are commonly included on most modern microprocessors.