Software debouncing of buttons

snigelen

February 5, 2015

1 Introduction

Connecting a button as an input to a micro-controller is a relatively easy task, but there are some problems. The main problem is that buttons bounce, i.e. when you press (or release) a button it will often change level a couple of times before it settles at the new level. So if you, for example, connect the button to a pin with an external interrupt enabled, you will get several interrupts when you press the button once. This behavior is normally not wanted. Even if the button's didn't bounce (with filtering hardware for example) we still want to capture the event of a pushed button and take some action one time for every button press, so we need to keep track of the state of the button as well.

One technique, used in this tutorial, to handle this is to check (poll) the button(s) periodically and only decide that a button is pressed if it have been in the pressed state for a couple of subsequent polls.

We will end up with a solution that is functionally equivalent with the debounce routines written by Peter Dannegger (danni at avrfreaks.net).

The examples in this tutorial is for the 8-bit AVR micro controller family and can be compiled with avr-gcc. The reader is supposed to have some basic knowledge about C and AVR and some knowledge of logical operations (and, or, xor, not) and how to read and write individual bits in a byte.

2 Connecting the Button

When we connect a button to an input pin on a MCU we need to have a well defined state both when the button is open and when it's closed. That can be accomplished with a resistor that pulls the port pin in one direction when the button is open. We can choose an active high or active low configuration, see figure 1. The resistor values

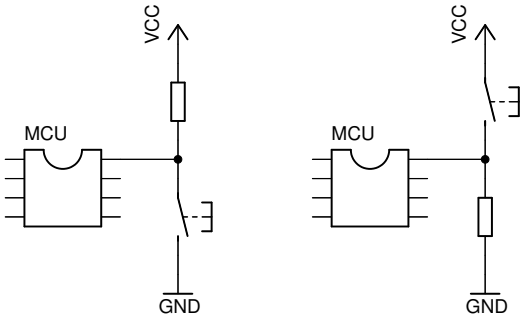

Figure 1: Left figure shows an active low connection, active high to the right.

can be around 10 kΩ to 50 kΩ. Since AVR's have the option to activate an internal pull-up resistor we usually choose the active low configuration shown on the left in figure 1. Then we can skip the external resistor and activate internal pull-up instead. If you want some extra protection you can replace the wire between the MCU and the button with another resistor (around 500 Ω) between the MCU and the button to prevent a short cut if you happen to miss-configure the port pin as an output.

3 Debouncing One Button

3.1 A Bad Example

If we want to toggle a LED on or of every time a button is pressed we can try this small program

```
#include <avr/io.h>

// Connect a button from ground to pin 0 on PORTA
#define BUTTON_MASK (1<<PA0)
#define BUTTON_PIN  PINA
#define BUTTON_PORT PORTA

// Connect a LED from ground via a resistor to pin 0 on PORTB
#define LED_MASK (1<<PB0)
#define LED_PORT PORTB
#define LED_DDR  DDRB

int main(void)
{
    // Enable internal pullup resistor on the input pin
    BUTTON_PORT |= BUTTON_MASK;

    // Set to output
    LED_DDR |= LED_MASK;

    while(1)
    {
```

```
        // Check if the button is pressed. Button is active low
        // so we invert PINB with ~ for positive logic
        if (~BUTTON_PIN & BUTTON_MASK)
        {
            // Toggle the LED
            LED_PORT ^= LED_MASK;
        }
    }
}
```

If we run this we'll see that the LED state after we press the button is more or less randomly, because the LED is toggled (very) many times before the button is released. An easy, but bad, solution to this is to add a delay for say one second after the LED toggle. But we normally don't want to use delays (at least not that long delays) in real programs and the button must be released within one second and we can't have more than one button press within one second.

3.2 Debouncing Algorithm

We now want to check the button at regular intervals and consider the button to be pressed if it's in the same state for a couple of readings. Polling about every 10 ms and require four subsequent equal readings before the buttons state is changed usually works fine. Something like this in pseudo code

```
if (time to check button)
   read current button state
   if (current button state != button state)
      Increase counter
      if (counter >= 4)
         change state
         reset counter
      end
   else
      reset counter
   end
end
```

We could write a function to do this and return a value that indicates if the button is considered to be pressed or not. But we are soon going to end up with a solution driven by a timer interrupt so we use a global variable instead of a return value to tell if a button is pressed or not.

Here's an implementation of the algorithm in the function debounce() together with defines and a main for a complete test program.

```
#include <avr/io.h>
#include <util/delay.h>

// Connect a button from ground to pin 0 on PORTA
#define BUTTON_MASK (1<<PA0)
```

```
#define BUTTON_PIN  PINA
#define BUTTON_PORT PORTA

// Connect a LED from ground via a resistor to pin 0 on PORTB
#define LED_MASK (1<<PB0)
#define LED_PORT PORTB
#define LED_DDR  DDRB

// Variable to tell main that the button is pressed (and debounced).
// Main will clear it after a detected button press.
volatile uint8_t button_down;

// Check button state and set the button_down variable if a debounced
// button down press is detected.
// Call this function about 100 times per second.
static inline void debounce(void)
{
    // Counter for number of equal states
    static uint8_t count = 0;
    // Keeps track of current (debounced) state
    static uint8_t button_state = 0;

    // Check if button is high or low for the moment
    uint8_t current_state = (~BUTTON_PIN & BUTTON_MASK) != 0;

    if (current_state != button_state) {
        // Button state is about to be changed, increase counter
        count++;
        if (count >= 4) {
            // The button have not bounced for four checks, change state
            button_state = current_state;
            // If the button was pressed (not released), tell main so
            if (current_state != 0) {
                button_down = 1;
            }
            count = 0;
        }
    } else {
        // Reset counter
        count = 0;
    }
}

int main(void)
{
    // Enable internal pullup resistor on the input pin
    BUTTON_PORT |= BUTTON_MASK;

    // Set to output
    LED_DDR |= LED_MASK;

    while(1)
    {
        // Update button_state
```

```
        debounce();
        // Check if the button is pressed.
        if (button_down)
        {
            // Clear flag
            button_down = 0;
            // Toggle the LED
            LED_PORT ^= LED_MASK;
        }
        // Delay for a while so we don't check to button too often
        _delay_ms(10);
    }
}
```

This example suffers from a delay in the main loop, but that could easily be replaced with for example a flag that's set by a hardware timer.

4 More Buttons

In the previous example we needed four variables. One that keeps track of the debounced buttons state, one for the current button state, one to tell if a button is pressed and one counter.

If we want to add more buttons we can simply add a set of variables for each button and code to check and count each of them. That can be a lot of variables and more code that we may not want in a small micro controller.

Since we only use one bit in each of the three 8–bit variables `current_state`, `button_state` and `button_down` we can do some savings by using one bit in each of these variables for each button, and therefor use up to eight buttons with only thees three variables. To check the state of all the pins on a port we can simply do

```
    // Check state of all pins on a port (inverted for positive logic)
    uint8_t current_state = ~BUTTON_PIN;
```

But we rather want to know which pins have changed and we can find that with a simple xor operation

```
    uint8_t state_changed = ~BUTTON_PIN ^ button_state;
```

`state_changed` will now contain zero for the bits that don't have changed and one where current state differs from the saved state.

The counter variable is used to count from zero to four, so three bits is used there. If we had 2–bit variables we could still count four steps if it could overflow. It doesn't matter if we count up or down and the start point doesn't matter either. With an eight bit variable we could use a start value of, for example, 3 and count 256 steps with something like this

```
count = count - 1;
if (count == 3) {
    // Counter overflow after 256 steps
...
    // Reset counter
    count = 3;
```

If count were an unsigned two bit variable it would overflow at three (and underflow at 0). So let's create some two bit counters...

4.1 Vertical Counting

We know that an eight bit variable can have 256 different values, but we can also treat it as eight 1–bit variables, each holding the value zero or one. With two bytes we can use them as eight 2–bit variables, like these random bits

Bit number	7	6	5	4	3	2	1	0
High byte	0	0	1	1	0	1	0	1
Low byte	1	1	0	0	1	1	0	1
Value	1	1	2	2	1	3	0	3

If we're going to use them for counting we want to be able to increase or decrease the counters. First we recall the truth tables for the logical operations not (∼), and (&), or (|) and xor (^).

B=∼A

A	B
0	1
1	0

C = A & B

A	B	C
0	0	0
0	1	0
1	0	0
1	1	1

C = A | B

A	B	C
0	0	0
0	1	1
1	0	1
1	1	1

C = A ^ B

A	B	C
0	0	0
0	1	1
1	0	1
1	1	0

The operations increase and decrease have the following truth tables

Increase			
Before		After	
H_b	L_b	H_a	L_a
0	0	0	1
0	1	1	0
1	0	1	1
1	1	0	0

Decrease			
Before		After	
H_b	L_b	H_a	L_a
0	0	1	1
0	1	0	0
1	0	0	1
1	1	1	0

If we choose the decrease operation, it can be written like this in C

```
// Decrease vertical two bit counters
high = ~(high ^ low);
low  = ~low;
```

or one operation shorter, like this

```
low  = ~low;
high = high ^ low;
```

But we don't want to decrease all eight counters at once, only those who corresponds to the buttons that is being pressed (or released), the other counters should be reset. We also want to detect an overflow for the bits we count and we choose the binary value 11 for the reset state, since it can be detected with a single and operation. Let's call that operation for "decrease or set" with the following truth table, where M stands for mask and indicates if a counter is going to be decreased or set to binary 11.

	Decrease or set			
	Before		After	
M	H_b	L_b	H_a	L_a
0	0	0	1	1
0	0	1	1	1
0	1	0	1	1
0	1	1	1	1
1	0	0	1	1
1	0	1	0	0
1	1	0	0	1
1	1	1	1	0

We can implement that operation like this in C

```
high = ~(mask & (high ^ low));
low  = ~(low & mask);
```

This can be written a little shorter (maybe add an H&M column in the truth table to see this easier)

```
low  = ~(low & mask);
high = low ^ (high & mask);
```

4.2 Putting It All Together

We have a variable `button_state` that holds the current debounced state, each bit position holds a 1 if a button is in the pressed state, a 0 if it's in the released state. When we enter the debounce routine we want to know if any buttons at this moment is in a different state than the debounced `button_state`. This can be done with xor.

```
uint8_t state_changed = ~BUTTON_PIN ^ button_state;
```

(where $\sim$ was to invert to positive logic when we use the active low configuration) then `state_changed` will contain a 1 in the positions corresponding to the buttons who's state differs from the current debounced states.

Now we want to count the number of times we had a 1 in a position in `changed_state`. For the positions where we have 0 in `changed_state` we want to reset that counter to binary 11. If we let the (static) variables `vcount_low` and `vcount_high` be our vertical counter we can perform a "decrease or set" operation with them and let `changed_state` be the mask.

```
// Decrease vertical counters where state_changed has a bit set (1)
// and set the others to 11 (reset state)
vcount_low = ~(vcount_low & state_changed);
vcount_high = vcount_low ^ (vcount_high & state_changed);
```

The bit pairs in `vcount_low` and `vcount_high` will now contain binary 11 for the pairs that were reset and for the pairs we are counting that have rolled over (from 00 to 11, since we decrease). We can tell which of these to conditions it is since `state_changed` tells which counters we decreased and which counters we reset.

Now we can update `state_changed` to only hold a one in the positions where the counters did overflow.

```
// Update state_changed to have a 1 only if the counter overflowed
state_changed &= vcount_high & vcount_low;
```

If there are any ones left in `state_changed` we know that these counters overflowed and we update `button_state` with that information. We can use xor again to change the bits in the positions were the updated `state_changed` have a 1.

```
// Change button_state for the buttons who's counters rolled over
button_state ^= state_changed;
```

Last step is to update the `buttons_down` variable with those changed states were we detected a button press (not release). And since there can be other positions in that variable that have not been confirmed by the main loop yet we must or in the new result, otherwise any previous unhandled button presses will be lost.

```
// Update button_down with buttons who's counters rolled over
// and who's state is 1 (pressed)
button_down |= button_state & state_changed;
```

That's all. Now the main loop can check the bits in `buttons_down` and take action if it finds a 1, and clear that bit so the action only is done once for that button press. We can do that in a function that check if a bit is 1 and if so, clears that bit and return non zero. See the function `button_down` in the following section. We have now

implemented the debounce algorithm using only logical operations. Up to a full port of buttons can be handled at about the same cost as with the one button routine.

4.3 The Result

To sum up all of this we end up with this debounce routine,

```
/*
  debounce.h. Snigelens version of Peter Dannegger's debounce routines.
  Debounce up to eight buttons on one port.  $Rev: 577 $
 */
#ifndef DEBOUNCE_H
#define DEBOUNCE_H

#include <avr/io.h>
#include <avr/interrupt.h>
#include <util/atomic.h>

// Buttons connected to PA0 and PA1
#define BUTTON_PORT  PORTA
#define BUTTON_PIN   PINA
#define BUTTON_DDR   DDRA
#define BUTTON1_MASK (1<<PA0)
#define BUTTON2_MASK (1<<PA1)
#define BUTTON_MASK  (BUTTON1_MASK | BUTTON2_MASK)

// Variable to tell that the button is pressed (and debounced).
// Can be read with button_down() which will clear it.
extern volatile uint8_t buttons_down;

// Return non-zero if a button matching mask is pressed.
uint8_t button_down(uint8_t button_mask);

// Make button pins inputs and activate internal pullups.
void debounce_init(void);

// Decrease 2 bit vertical counter where mask = 1.
// Set counters to binary 11 where mask = 0.
#define VC_DEC_OR_SET(high, low, mask)          \
    low = ~(low & mask);                        \
    high = low ^ (high & mask)

// Check button state and set bits in the button_down variable if a
// debounced button down press is detected.
// Call this function every 10 ms or so.
static inline void debounce(void)
{
    // Eight vertical two bit counters for number of equal states
    static uint8_t vcount_low  = 0xFF, vcount_high = 0xFF;
    // Keeps track of current (debounced) state
    static uint8_t button_state = 0;

    // Read buttons (active low so invert with ~). Xor with
```

```
    // button_state to see which ones are about to change state
    uint8_t state_changed = ~BUTTON_PIN ^ button_state;

    // Decrease counters where state_changed = 1, set the others to 0b11.
    VC_DEC_OR_SET(vcount_high, vcount_low, state_changed);

    // Update state_changed to have a 1 only if the counter overflowed
    state_changed &= vcount_low & vcount_high;
    // Change button_state for the buttons who's counters rolled over
    button_state ^= state_changed;

    // Update button_down with buttons who's counters rolled over
    // and who's state is 1 (pressed)
    buttons_down |= button_state & state_changed;
}
#endif
```

```
/*
  debounce.c
 */

#include "debounce.h"

// Bits is set to one if a depounced press is detected.
volatile uint8_t buttons_down;

// Return non-zero if a button matching mask is pressed.
uint8_t button_down(uint8_t button_mask)
{
    // ATOMIC_BLOCK is needed if debounce() is called from within an ISR
    ATOMIC_BLOCK(ATOMIC_RESTORESTATE){
        // And with debounced state for a one if they match
        button_mask &= buttons_down;
        // Clear if there was a match
        buttons_down ^= button_mask;
    }
    // Return non-zero if there was a match
    return button_mask;
}

void debounce_init(void)
{
    // Button pins as input
    BUTTON_DDR &= ~(BUTTON_MASK);
    // Enable pullup on buttons
    BUTTON_PORT |= BUTTON_MASK;
}
```

A test program:

```
/*
  Test program. F_CPU around 8 MHz is assumed for about
  10 ms debounce intervall.   $Rev: 563 $
 */
```

```
#include <avr/io.h>
#include <avr/interrupt.h>
#include "debounce.h"

// Called at about 100Hz (122Hz)
ISR(TIMER0_OVF_vect)
{
    // Debounce buttons. debounce() is declared static inline
    // in debounce.h so we will not suffer from the added overhead
    // of a (external) function call
    debounce();
}

int main(void)
{
    // BORTB output
    DDRB = 0xFF;
    // High turns off LED's on STK500
    PORTB = 0xFF;

    // Timer0 normal mode, presc 1:256
    TCCR0B = 1<<CS02;
    // Overflow interrupt. (at 8e6/256/256 = 122 Hz)
    TIMSK0 = 1<<TOIE0;

    debounce_init();

    // Enable global interrupts
    sei();

    while(1)
    {
        if (button_down(BUTTON1_MASK))
        {
            // Toggle PB0
            PORTB ^= 1<<PB0;
        }
        if (button_down(BUTTON2_MASK))
        {
            // Toggle PB1
            PORTB ^= 1<<PB1;
        }
    }
}
```

I did choose to call the debounce routine directly from a timer interrupt. Therefore I put `debounce` in the header file declared as static inline, to avoid the overhead associated with function call from interrupt routines.

Of course you can just set a (volatile) flag in the ISR and let main main check the flag and call debounce if it's set (and clear the flag). If you prefer that you can move the debounce function to debounce.c instead and remove the static inline keywords.