Switch Debouncing Logic in C

Asked 5 years, 2 months ago Modified 5 years, 2 months ago Viewed 31k times

3

I came across this code by Ganssle regarding switch debouncing. The code seems pretty efficient, and the few questions I have maybe very obvious, but I would appreciate clarification.

- Why does he check 10 msec for button press and 100 msec for button release. Can't he just check 10 msec for press and release?
- Is polling this function every 5 msec from main the most effecient way to execute it or should I check for an interrupt in the pin and when there is a interrupt change the pin to GPI and go into the polling routine and after we deduce the value switch the pin back to interrupt mode?

```c
#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 will allow a change from the 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;
        }
    }
}
```

c embedded debouncing

Share Edit Follow edited Jan 25, 2018 at 2:18 asked Jan 25, 2018 at 2:10

Pablo
13.2k ●4 ●38 ●58

CodeModeOn
111 ●1 ●2 ●11

2 Answers

Sorted by:
Highest score (default)

5

> Why does he check 10 msec for button press and 100 msec for button release.

As the blog post says, "*Respond instantly to user input.*" and "*A 100ms delay is quite noticeable*".

So, the main reason seems to be to emphasize that the make-debounce should be kept short so that the make is registered "immediately" by human sense, and that the break debounce is less time sensitive.

This is also supported by a paragraph near the end of the post: "*As I described in the April issue, most switches seem to exhibit bounce rates under 10ms. Coupled with my observation that a 50ms response seems instantaneous, it's reasonable to pick a debounce period in the 20 to 50ms range.*"

In other words, the *code* in the example is much more important than the *example values*, and that the proper values to be used depends on the switches used; you're supposed to decide those yourself, based on the particulars of your specific use case.

> Can't he just check 10 msec for press and release?

Sure, why not? As he wrote, it should work, even though he wrote (as quoted above) that he prefers a bit longer debounce periods (20 to 50 ms).

> Is polling this function every 5 msec from main the most effecient way to execute it

No. As the author wrote, "*All of these algorithms assume a timer or other periodic call that invokes the debouncer.*" In other words, this is just *one* way to implement software debouncing, and the shown examples are based on a regular timer interrupt, that's all.

Also, there is nothing magical about the 5 ms; as the author says, "*For quick response and relatively low computational overhead I prefer a tick rate of a handful of milliseconds. One to five milliseconds is ideal.*"

> or should I check for an interrupt in the pin and when there is a interrupt change the pin to GPI and go into the polling routine and after we deduce the value switch the pin back to interrupt mode?

If you implement that in code, you'll find that it is rather nasty to have an interrupt that blocks the normal running of the code for 10 - 50ms at a time. It is okay if checking the input pin state is the only thing being done, but if the hardware does anything else, like update a display, or flicker some blinkenlights, your debouncing routine in the interrupt handler will cause noticeable jitter/stutter. In other words, what you suggest, is not a practical implementation.

The way the periodic timer interrupt based software debouncing routines (shown in the original blog post, and elsewhere) work, they take only a very short amount of time, just a couple of dozen cycles or so, and do not interrupt other code for any significant amount of time. This is simple, and practical.

You can combine a periodic timer interrupt and an input pin (state change) interrupt, but since the overhead of many of the timer-interrupt-only -based software debounces is tiny, it typically is not worth the effort trying to combine the two -- the code gets very, very complicated, and complicated code (especially on an embedded device) tends to be hard/expensive to maintain.

The only case I can think of (but I'm only a hobbyist, not an EE by any means!) is if you wanted to minimize power use for e.g. battery powered operation, and used the input pin interrupt to bring the device to partial or full power mode from sleep, or similar.

(Actually, if you also have a millisecond or sub-millisecond counter (not necessarily based on an interrupt, but possibly a cycle counter or similar), you can use the input pin interrupt and the cycle counter to update the input state on the first change, then desensitize it for a specific duration afterwards, by storing the cycle counter value at the state change. You do need to handle counter overflow, though, to avoid the situation where a long ago event seems to have happened just a short time ago, due to counter overflowing.)

I found Lundin's answer quite informative, and decided to edit my answer to show my own suggestion for software debouncing. This might be especially interesting if you have very limited RAM, but lots of buttons multiplexed, and you want to be able to respond to key presses and releases with minimum delay.

Do note that I do not wish to imply this is "best" in any sense of the world; I only want you to show one approach I haven't seen often used, but which might have some useful properties in some use cases. Here, too, the number of scan cycles (milliseconds) the input changes are ignored (10 for make/off-to-ON, 10 for break/on-to-OFF) are just example values; use an oscilloscope or trial-and-error to find the best values in your use case. If this is an approach you find more suitable to your use case than the other myriad alternatives, that is.

The idea is simple: use a single byte per button to record the state, with the least significant bit describing the state, and the seven other bits being the desensitivity (debounce duration) counter. Whenever a state change occurs, the next change is only considered a number of scan cycles later.

This has the benefit of responding to changes immediately. It also allows different make-debounce and break-debounce durations (during which the pin state is not checked).

The downside is that if your switches/inputs have any glitches (misreadings outside the debounce duration), they show up as clear make/break events.

First, you define the number of scans the inputs are desensitized after a break, and after a make. These range from 0 to 127, inclusive. The exact values you use depend entirely on your use case; these are just placeholders.

```
#define  ON_ATLEAST   10  /* 0 to 127, inclusive */
#define  OFF_ATLEAST  10  /* 0 to 127, inclusive */
```

For each button, you have one byte of state, variable `state` below; initialized to 0. Let's say `(PORT & BIT)` is the expression you use to test that particular input pin, evaluating to *true* (nonzero) for ON, and *false* (zero) for OFF. During each scan (in your timer interrupt), you do

```
if (state > 1)
    state -= 2;
else
if ( (!(PORT & BIT)) != (!state) ) {
    if (state)
        state = OFF_ATLEAST*2 + 0;
    else
        state = ON_ATLEAST*2 + 1;
}
```

At any point, you can test the button state using `(state & 1)`. It will be 0 for OFF, and 1 for ON. Furthermore, if `(state > 1)`, then this button was recently turned ON (if `state & 1`) or OFF (if `state & 0`) and is therefore not sensitive to changes in the input pin state.

Share Edit Follow edited Jan 27, 2018 at 0:33 answered Jan 25, 2018 at 3:18

Nominal Animal
37.7k ● 5 ● 55 ● 85

Thanks for the explanation, that helped a lot. I read the blog post and I noticed the 20 to 50 ms period you re-quoted. I should have explained my confusion a little better. Seems like he put a 100ms release period(or whatever longer period value) in case of a push button switch? In case of a rocker switch it doesn't really matter, I can keep the release and press period as the short period value(10ms or something)? Although a good generic code for all application is what he posted. Or probably let me know if I'm thinking too much :-P – CodeModeOn Jan 25, 2018 at 5:42

@CodeModeOn: I think you are putting too much weight behind the actual values in the example. I see them as clearly set placeholders only, with no particular meaning... not chosen for a practical reason (like "*these are good values to use*"), but to make the code easier to understand and modify to ones own use. I often do the same myself, in my own example code snippets. – Nominal Animal Jan 25, 2018 at 10:04

In addition to the accepted answer, if you just wish to poll a switch from somewhere every *n* ms, there is no need for all of the obfuscation and complexity from that article. Simply do this:

2

```
static bool prev=false;
...

/*** execute every n ms ***/
bool btn_pressed = (PORT & button_mask) != 0;
bool reliable = btn_pressed==prev;
prev = btn_pressed;

if(!reliable)
{
  btn_pressed = false; // btn_pressed is not yet reliable, treat as not pressed
}

// <-- here btn_pressed contains the state of the switch, do something with it
```

This is the simplest way to de-bounce a switch. For mission-critical applications, you can use the very same code but add a simple median filter for the 3 or 5 last samples.

As noted in the article, the electro-mechanical bounce of switches is most often less than 10ms. You can easily measure the bouncing with an oscilloscope, by connecting the switch between any DC supply and ground (in series with a current-limiting resistor, preferably).

Share Edit Follow

answered Jan 25, 2018 at 9:35

Lundin
190k ● 39 ● 249 ● 388