My favorite software debouncers

By Jack Ganssle
Embedded Systems Design
(06/16/04, 06:00:00 PM EDT)

Click here for reader response to this article

Jack breaks out the real code for taming the recalcitrant but ubiquitous mechanical switch.

After last month's look at the Embedded Systems Conference we're back to the surprisingly deep subject of debouncing switch inputs. But first, a mea culpa for mixing up formulas in May's column ("Solving Switch Bounce Problems," May 2004, on pages 46 and 48). The first and second formulas were accidentally swapped. To see the corrected version, go to www.embedded.com/showArticle.jhtml?articleID=18902552.

Software debounce routines range from some quite simple approaches to sophisticated algorithms that handle multiple switches in parallel. 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 induced by feet shuffling 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 500ms 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 instantaneous state. No matter how we interpret the data (in other words, 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.

No EMI protection is inherent in such a simple approach. An application handling contacts plated onto the printed circuit board 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 500ms 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 integrated circuits, where it belongs. All digital inputs still looked like hash and needed an astonishing amount of debounce and signal conditioning.

Debounce policy

We need to place some basic constraints on our anti-contact-clacking routines, such as minimizing 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 on page 24 of that document 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 60Hz 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 I wired a switch up to the cool R3000 starter kit that Rabbit Semiconductor provides. This board and software combo seems aimed at people either learning embedded systems programming or those of us who just like to play with electronics. I wrote a bit of simple code to read a button and, after a programmable delay, turn on an LED. A 100ms delay is quite noticeable, even to these tired old 20/1,000 eyes. A 50ms delay, 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 presumed 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 that no resources are free, 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: A simple yet effective debounce algorithm

```
#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;
      }
   }
}
```

Listing 1 shows a sweet little debouncer that is called every **CHECK_MSEC** milliseconds by the timer interrupt, a timer-initiated task, or some similar entity.

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.

Listing 2: An even simpler debounce routine

```
// 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;
}
```

An alternative

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

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.

You can 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 just 11 lines of 8051 assembly code.

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 one very 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 bank in parallel?

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

```
#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-1;i++)j=j & State[i];
   Debounced_State=Debounced_State ^ j;
   if(Index>=MAX_CHECKS)Index=0;
}
```

Listing 3 shows one approach. **DebounceSwitch3()**, which is called regularly by a timer tick, reads an entire byte-wide port that contains up to eight 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, another dimension 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).

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 eight debounced switch values.

Exclusive-ORing this with the last **Debounced_State** yields 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.

I prefer a less computationally intensive alternative that splits **DebounceSwitch3()** into these two routines: one routine 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.

Five milliseconds

All of these algorithms assume a timer or other periodic call that invokes the debouncer. For quick response and relatively low computational overhead I prefer a tick rate of a handful of milliseconds. One to five milliseconds is ideal. 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.

Hundreds of other debouncing algorithms exist. The ones I've presented here are just a few of my favorites, offering great response, simple implementation, and no reliance on magic numbers or other sorts of high-tech incantations.

Thanks to many, many readers who wrote in with suggestions and algorithms. I shamelessly stole ideas from many of you, especially Scott Rosenthal, Simon Large, Jack Marshall, and Jack Bonn.

Jack G. Ganssle is a lecturer and consultant on embedded development issues. He conducts seminars on embedded systems and helps companies with their embedded challenges. Contact him at jack@ganssle.com.

Reader Response

I saw your article on switch debouncing and thought you might want to see another approach. The twist on this one is that it's optimized to handle multiple switches in parallel. The logic for the debouncing is similar to that which you describe. My algorithm utilizes 'vertical counters' and boolean logic to implement the debounce counters. It takes only 13 pic instructions to filter 8

inputs: http://www.dattalo.com/technical/software/pic/debounce.html The value for my equivalent of 'MAX_CHECKS' is 4. Which is to say, the switch must be sample in the same state at least 4 times before it is considered changed.

- Scott Dattalo

I am quite surprised that no-one else has published this alternative - I have been using it for years, with, I believe, success.

Re Debouncing - I believe that there is an alternative which you don't appear to have discussed in your articles on embedded.com.

I sometimes (well, mostly) do the following when considering switch transitions from devices external to the CPU board: I presume that there will always be a train of pulses. I consider that the only items of significance are:
a. the initial pulse,
b. the time till the output is steady, and
c. the 'reliability' of the system, ie resistance to induced noise - does the system indicate a transition when the switch has not initiated a transition.

If we make the presumption that the system is immune to induced noise on this line, then we can make the presumption that the initial pulse is in fact a true indicator that the switch has changed state. If we cannot make this presumption, then the machinery is most likely faulty, and correcting electrical faults will almost always cure this problem.

If we then make the presumption that the switch can NOT change state within a certain time interval, we can safely ignore the switch for a 'defined' time period, ie we 'lock out' the following train of pulses. We are then acting similarly to an analog schmitt trigger, but we are working in the digital domain.

What this does for a software driven device:
a. The switch triggering is always consistent - this analogy can be applied to rotating mechanisms with detectors. Rotating mechanisms ALWAYS exhibit jitter at some point - and this triggering mechanism tends to 'get rid of it'.
b. Simplifies the debounce software handling the switching, sometimes to the extent of having NO code to handle debouncing.
c. Unfortunately, unless you document it, which I rarely do, reviewers don't think you have considered switch transitions and pulse trains.

Here is some sample code which demonstrates what I am talking about. (compiles as CPP code, sorry if it doesn't compile with C compiler)

- Bill Mather

eg. presume the 'task' code takes 2 mS to execute.
presume the switches never cause a train of pulses longer than 1.999 mS (ie system stable at 2 mS)
presume timer interrupt occurs every 1uS
presume timer counter never overflows

```
#include

#define forever true
#define OFF false
#define ON true
bool ioSwitch;

// for this example, presume that myTicker is infinite and never wraps
#define PULSE_TRAIN_TIME 1999
time_t myTicker = 0;

//for compilation
bool IO_SWITCH = true;
#define interrupt

//---------------------------------------------------------------------------
void microcontrollerTask (void)
//---------------------------------------------------------------------------
{
// do something
// this task takes (2 mS) to execute
}

//---------------------------------------------------------------------------
void DebounceSwitch (void)
//---------------------------------------------------------------------------
{
// do the Jack Ganssle software debounce
}

//---------------------------------------------------------------------------
// Get Debounced Switch State
bool IsSwitchStateChanged (void)
//---------------------------------------------------------------------------
{
static bool oldState = OFF;
ioSwitch = IO_SWITCH; // grab value from I/O
if (ioSwitch != oldState) // if ioSwitch state has changed, do something
{
DebounceSwitch ();
oldState = ioSwitch; // save state for next comparison
return true; // ioSwitch state HAS
changed
}
return false; // ioSwitch state has
NOT changed.
}

//---------------------------------------------------------------------------
void UseExplicitDebounceMainLoop (void)
//---------------------------------------------------------------------------
{
while (forever)
{
if (IsSwitchStateChanged ())
microcontrollerTask (); // task has been delayed by
```

```
debounce time

//
// everything else which needs doing.
//
}
}

//-------------------------------------------------------------------------
void UseImplicitDebounceMainLoop (void)
//-------------------------------------------------------------------------
{
while (forever)
{
static bool oldState = OFF;
ioSwitch = IO_SWITCH; // grab value from I/O
if (ioSwitch != oldState) // if ioSwitch state has changed, do something
{
oldState = ioSwitch; // save state for next comparison
microcontrollerTask (); // task happens very close to real
ioSwitch time
}

//
// everything else which needs doing.
//
}
}

//-------------------------------------------------------------------------
// presume time_t is infininte and never wraps
// presume interrupts at 1 uS
void interrupt MyTimer (void)
//-------------------------------------------------------------------------
{
myTicker++;
}

//-------------------------------------------------------------------------
bool IsTimeoutComplete (time_t testTime)
//-------------------------------------------------------------------------
{
// presume time_t is infinite and never wraps
return (testTime <= myTicker);
}

//-------------------------------------------------------------------------
// Get Initial Switch State, ignoring ioSwitch for time T after change
bool IsSwitchStateChangedIgnorePulseTrain (void)
//-------------------------------------------------------------------------
{
static bool oldState = OFF;
static time_t oldTime = 0;

if (IsTimeoutComplete (oldTime))
{
```

```
ioSwitch = IO_SWITCH; // grab value from I/O
if (ioSwitch != oldState) // if switch state has changed, save state
{
// DebounceSwitch (); // NOT NEEDED any longer
oldState = ioSwitch; // save state for next comparison
oldTime = myTicker + PULSE_TRAIN_TIME;
return true; // ioSwitch state HAS
changed
}
}
return false; // ioSwitch state has
NOT changed.
}

//------------------------------------------------------------------------
void UseExplicitIgnorePulseTrainTimeMainLoop (void)
//------------------------------------------------------------------------
{
while (forever)
{
if (IsSwitchStateChangedIgnorePulseTrain ())
microcontrollerTask (); // task has been NOT been
delayed, but pulse train is ignored

//
// everything else which needs doing.
//
}
}

//------------------------------------------------------------------------
int main (void)
//------------------------------------------------------------------------
{
// UseExplicitDebounceMainLoop ();
// UseImplicitDebounceMainLoop ();
// UseExplicitIgnorePulseTrainTimeMainLoop ();
return 0;
}

//------------------------------------------------------------------------
```

One thing that I think needs to be added is that bouncing becomes worse over time/wear, the bounce time may increase by an order of magnitude for some switches. My rule of thumb is that 100ms debounce time is needed, unless the switch is proven to be of a really good type. This also means that using stable readings just doesn't work at all. Another thing is that a key press may be as low as 1-2ms for a really quick (or most likely angry (and therefore somewhat likely to call support)) user. This means physically hit and release a keypad switch so that it bottoms out. So my prefered solution is to first make sure the hardware design prevents EMI from triggering any closure readings ever, typically meaning running several mA (rule of thumb says 10mA+ for thin-plated ones anyway) through the switch, and in severe cases (seldom needed) add a RC-link with low enough impedance to prevent false triggers. Second, on the first switch closure reading, save and return clisure, then wait for the switch being constantly open for more than 100ms. If you have multiple switches, you may often just AND together the switches (assuming 0 for closed

state) for debounced value, return any bits gone low, compare the instantanious value with the previous copy of the input state, and copy that input value to the debounced value if it's been the same value for more than 100ms. The fact that the debounce restarts if any other key is depressed is usally never noticed as long as the keypad is operated with only one finger. And this way, the wxecution time of the routine drops to a level where you normally can afford to poll the keyboard at least every ms on a timer interrupt on almost any processor.

A side note on the SR latch circuit in the previous article is that the 'normally closed' contact of a double throw switch typically has significantly lower reliability than the 'normally open' contact, and therefor it's not such a good alternative even if size and cost is no problem.

- Dan Norstedt

Discuss This Article

1 message(s). Last at: Apr 17, 2008 9:01:00 AM

mac_droz
design engineer
commented on Apr 17, 2008 9:01:00 AM

Hi!

This is my debouncer (signal conditioner). It simulates exactly RC network with Schmitt trigger, so even very noisy signals can be read clearly - it may just take more time to get the answer.

Input: raw_input - a bit read from port
Output: debounced_output - debounced (filtered) data
I call it every ms - in fact it is a part of my interrupt.
In this example you have to get at least 10 same samples to change output state.

```
{
signed char debounce_counter;

if (raw_input) debounce_counter++;
else debounce_counter--;

if (debounce_counter > 10) {debounce_counter = 10;
debounced_output=1; }
if (debounce_counter < 0 ) {debounce_counter = 0 ;
debounced_output=0; }
}
```

Enjoy!

Maciej Drozdowski, SQ2AHR

Please login or register here to post a comment or to get an email when other comments are made on this article

Track This Thread

Sends you email alerts when someone comments on this article