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.

I 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 I 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.

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