PIC De-bouncing

Debouncing; What is it and what is used for? Well, if you ever looked at the current through (or the voltage across) a mechanical switch or a relay as it was changing states you would notice that the transition is not always a nice, clean step function. Often there are several changes before the switch makes its transition. In this case, the switch is said to be "bouncing". However, the same behavior can be observed in dirty switches or switches that are switching a heavy load. Some times the bouncing can be removed with mild (a good vague term) analog filtering and a comparator with hysteresis. Unfortunately this won't work for all cases. And even if it did, the extra cost in hardware may not be justifiable; especially since you can do most of the work in software.

Typical debouncing routines sample the switch's state at a fairly high rate (vague term # 2). When a change in the state is detected, the routine will count anywhere from 1 to 4 to 8 to whatever samples to make sure the transition was not a glitch and that the switch has settled to its new state. If you have many switches that need to be debounced simultaneously, it gets rather unwieldy keeping track of the counters and etc. This is emperically demonstrated by the brute-force debounce routines you typically see. Here's a routine that will simultaneously debounce 8 inputs and count 4 samples before declaring a switch has changed states. There are no loops or gotos. The eight 2-bit counters are arranged as "vertical counters". In other words, the low order bit of the eight counters occupy one byte in RAM while the high order one another. The counters are incremented (actually decremented) using Boolean logic. Different examples of the vertical counters are available on the vertical counters page.

A special thanks goes to Terje Mathisen whose insight led to the current version of this routine being 33% more efficient then the previous version.

```
;*************************************************************
; de_bounce
;
; The purpose of this routine is to debounce, i.e. digitally low pass filter
;inputs. The algorithm handles upto 8 bits at a time. An input is considered
;filtered if it has not changed states in the last 4 samples.
;
; 2-bit cyclic vertical counters count the 4 samples. As long as there is no
;change, the counters are held in the reset state of 00b. When a change is detected
;between the current sample and the filtered or debounced sample, the counters
;are incremented. The counting sequence is 00,01,10,11,00... When the counters
;roll over from 11b to 00b, the debounced state is updated. If the input changes
;back to the filtered state while the counters are counting, then the counters
;are re-initialized to the reset state and the filtered state is unaffected.
;In other words, a glitch or transient input has been filtered.
;
;The algorithm will return in W the state of those inputs that have just
;been filtered.
;
; Here's the C-psuedo code:
;
;static unsigned clock_A,clock_B,debounced_state;
;unsigned debounce(unsigned new_sample)
;{
;  unsigned delta;
;  unsigned changes;
;
;  delta = new_sample ^ debounced_state;   //Find all of the changes
;
;  clock_A ^= clock_B;                     //Increment the counters
```

```
;  clock_B  = ~clock_B;
;
;  clock_A &= delta;                       //Reset the counters if no changes
;  clock_B &= delta;                       //were detected.
;
;  changes = ~(~delta | clock_A | clock_B);
;  debounced_state ^= changes;
;
;  return changes;
;}
;
; The 2-bit counters are arranged "vertically". In other words 8 counters
;are formed with 2 bytes such that the corresponding bits in the bytes are
;paired (e.g. MSBit of each byte is paired to form one counter).
; The counting sequence is 0,1,2,3,0,1,... And the state tables and Karnaugh
;maps are:
;
;State Table:     Karnaugh Maps:
;pres  next       B
; SS  SS          0   1
; AB  AB        +---+---+    +---+---+
;--------   A 0|   | 1 |    | 1 |   |
; 00  01        +---+---+    +---+---+
; 01  10       1| 1 |   |    | 1 |   |
; 10  11        +---+---+    +---+---+
; 11  00       A+ = A ^ B     B+ = ~B
;
; Here's the PIC code that implements the counter:
;       MOVF    SB,W    ;W = B
;       XORWF   SA,F    ;A+ = A ^ B
;       COMF    SB,F    ;B+ = ~B
;
;
;
; 13 instructions
; 14 cycles
; Inputs:
;   csa - The current sample
; Outputs
;   cva - The current value (filtered version of csa)
;   W   - Bits that have just been filtered.
;
; RAM used
;   count_A,
;   count_B - State variables for the 8 2-bit counters
;
de_bounce:
    ;Increment the vertical counter
        MOVF    count_B,W
        XORWF   count_A,F
        COMF    count_B,F

    ;See if any changes occurred
        MOVF    csa,W
        XORWF   cva,W

    ;Reset the counter if no change has occurred
        ANDWF   count_B,F
        ANDWF   count_A,F

    ;If there is a pending change and the count has
    ;rolled over to 0, then the change has been filtered
```

```
    XORLW   0xff            ;Invert the changes
    IORWF   count_A,W       ;If count is 0, both A and B
    IORWF   count_B,W       ;bits are 0

;Any bit in W that is clear at this point means that the input
;has changed and the count has rolled over.

    XORLW   0xff

;Now W holds the state of inputs that have just been filtered
;to a new state. Update the changes:

    XORWF   cva,F

;leave with W holding the state of all filtered bits that have
;change state

    RETURN
```

And here's more software.

BACK HOME

This page is maintained by Scott Dattalo. You can reach me at home: scott@dattalo.com
Last modified on 01JAN06.