Application Note: Push Buttons & Path Switcher

Introduction

This application note presents programming techniques for implementing and debouncing from 1-4 push buttons. This implementation also demonstrates the use of a path switcher (combined with a simple real-time clock[1]) to reduce the execution time of each interrupt. The program relies upon the SX's internal interrupt feature to allow background operation of the clock, buttons, and path_switcher as virtual peripherals.

How the circuit and program work

This firmware module requires no external circuitry, other the push buttons, their pull-up resistors, and an (optional[2]) oscillator crystal, making it quite straight forward. The real-time clock peripheral is described elsewhere (see note 1 below), and will not be discussed here other than that it passes control to the path switcher virtual peripheral once per millisecond.

The path_switch routine simply looks at the lower 2 bits of the real time clock's msec count and jumps to the corresponding push button vector. This allows for only one push button sequence to be run per interrupt and reduces the overall execution time of the interrupt sequence. This feature of path_switch may be used to select from amongst any number of code segments (including other than just push button modules) which do not require execution during each interrupt cycle. For such purposes, it does not even need to be combined with the real time clock[3], which is used here to simplify push button debounce time processing.

The push buttons are wired directly from port B, pins 0-3* to ground, with a 100K pull-up resistor[4] also connected to each port pin, but wired to V_{dd}.

Within a few[5] milliseconds of any pushbutton press, the corresponding ***pbx*** (where ***x***=0-3) push button code sequence will register the press. First, the program checks whether it is a new press by looking at the corresponding ***pbx_down*** flag. If it's not a new press, it is ignored. If it is new, the program makes sure the ***pbx_down*** flag is cleared and then begins incrementing the corresponding ***debouncex*** counter variable upon subsequent passes through the interrupt until it detects that the switch contacts have been sufficiently debounced[6], in which case the ***pbx_pressed*** flag is set along with the ***pbx_down*** flag.

It this code example, is the main program's responsibility, performed by the button_check main loop code sequence, to scan the ***pbx_pressed*** flags to watch for a button press, and to make sure they're reset (cleared) once the appropriate button action has been taken.

If the button actions are short, they may be placed directly in-line in the interrupt code segment for the corresponding button. This has the attractive benefit of avoiding the need for any main loop handling of the buttons whatsoever, but also carries the disadvantage of increasing the overall length of the interrupt routine (which is somewhat compensated for by the path_switcher code module).

---

[1] Described in more detail a separate application note: **Virtual Peripheral Real Time Clock**

[2] If a lot of accuracy is needed on the clock, the SX's internal oscillator may be used by adjusting the msec tick count value to the appropriate count, as described in the above application note.

[3] If it is not to be combined with the real time clock module, some type of counter must still be maintained to control switching.

[*] From 1-4 push buttons may be used in this implementation. The user then sets the *num_buttons* parameter variable accordingly.

[4] The value of the pull-up is not very crucial, since the push button port pins are always set as inputs (i.e. high impedance), and the duration of presses is usually insignificant in terms of power consumption. Lower or higher resistor values can therefore be used.

[5] This value depends on how many push buttons are being used in total. It will range from 0-3 msec, depending.

[6] The amount of time any given switch or button takes to debounce is not obvious by any means, and varies with switch type and press speed and pressure, etc. A good rule of thumb to assure catching rapid sequential presses while avoiding false double triggering is about 10-20 msec for an average "click" type push button (and most other switches). This value can always be experimented with.

Modifications and further options

If the need for processor power between timed events is minimal, the three module routine combination could be modified and set up in conjunction with the watchdog timer instead of the internal RTCC interrupt where the SX is put in sleep mode between watchdog time-outs. This allows for a tremendous savings in overall power consumption.

Program Listing

```
;*****************************************************************************
;       Push Buttons & Path Switcher (with real time clock)
;
;
;       Length: >=74 bytes (depends upon number of buttons & clock type)
;       Author: Craig Webb
;       Written: 98/8/17
;
;       This program implements a software time clock virtual peripheral
;       that keeps a 16 bit count of elapsed time in milliseconds.
;       The option is available to include seconds, minutes, hours and even
;       days to this clock if desired.
;       The code takes advantage of the SX's internal RTCC-driven interrupt
;       to operate in the background while the main program loop is executing.
;
;*****************************************************************************
;
;****** Assembler directives
;
; uses: SX28AC, 2 pages of program memory, 8 banks of RAM, high speed osc.
;       operating in turbo mode, with 8-level stack & extended option reg.
;
              DEVICE  pins28,pages2,banks8,oschs
              DEVICE  turbo,stackx,optionx
              ID      'Buttons'                   ;program ID label
              RESET   reset_entry                 ;set reset/boot address
;
;****************************** Program Variables *************************
;
;****** Program Parameters
;
clock_type    =       0                     ;16 bit msec count only
;clock_type   =       1                     ;include sec, min, hours
;clock_type   =       2                     ;include day counter
;
num_buttons   =       2                     ;number of buttons (1-4)
;
;****** Program Constants
;
int_period    =       163                   ;period between interrupts
;
hold_bit      =       4-(num_buttons/2)     ;debounce period = 2^hold_bit msec
;
tick_lo       =       80                    ;50000 = msec instruction count
tick_hi       =       195                   ; for 50MHz, turbo, prescaler=1
;
mspersec_hi   =       1000/256              ;msec per second hi count
mspersec_lo   =       1000-(mspersec_hi*256) ;msec per second lo count
;
;****** Port definitions
;
button0       EQU     RB.0                  ;Push button 0
button1       EQU     RB.1                  ;Push button 1
button2       EQU     RB.2                  ;Push button 2
button3       EQU     RB.3                  ;Push button 3
;
;****** Register definitions
;
              ORG     8                     ;start of program registers
main           =      $                     ;main bank
;
temp           DS     1                     ;temporary storage
temp2         DS      1
;
```

```
               ORG     010H                    ;bank0 variables
clock          EQU     $                       ;clock bank
buttons        EQU     $                        ;push button bank
;
time_base_lo   DS      1                       ;time base delay (low byte)
time_base_hi   DS      1                       ;time base delay (high byte)
msec_lo        DS      1                       ;millisecond count (low)
msec_hi        DS      1                       ;millisecond count (high)

               IF      clock_type>0            ;do we want sec, min, hours?
seconds        DS      1                       ;seconds count
minutes        DS      1                       ;minutes count
hours          DS      1                       ;hours count
               ENDIF

               IF      clock_type>1            ;do we want day count?
days           DS      1                       ;days count
               ENDIF
;
;
debounce0      DS      1                       ;push button 0 debounce count
debounce1      DS      1                       ;push button 1 debounce count
debounce2      DS      1                       ;push button 2 debounce count
debounce3      DS      1                       ;push button 3 debounce count
pbflags        DS      1                       ;push button status flags
pb0_pressed    EQU     pbflags.0               ;push button 0 action status
pb1_pressed    EQU     pbflags.1               ;push button 1 action status
pb2_pressed    EQU     pbflags.2               ;push button 2 action status
pb3_pressed    EQU     pbflags.3               ;push button 3 action status
pb0_down       EQU     pbflags.4               ;push button 0 down status
pb1_down       EQU     pbflags.5               ;push button 1 down status
pb2_down       EQU     pbflags.6               ;push button 2 down status
pb3_down       EQU     pbflags.7               ;push button 3 down status
;
;*************************** INTERRUPT VECTOR ******************************
;
; Note: The interrupt code must always originate at 0h.
;       A jump vector is not needed if there is no program data that needs
;       to be accessed by the IREAD instruction, or if it can all fit into
;       the lower half of page 0 with the interrupt routine.
;
               ORG     0                               ;interrupt always at 0h
;              JMP     interrupt                       ;interrupt vector
;
;*************************** INTERRUPT CODE ********************************
;
; Note: Care should be taken to see that any very timing sensitive routines
;       (such as adcs, etc.) are placed before other peripherals or code
;       which may have varying execution rates (like the software clock, for
;          example).
;
interrupt                                              ;beginning of interrupt code
;
;****** Virtual Peripheral: Time Clock
;
; This routine maintains a real-time clock count (in msec) and allows processing
; of routines which only need to be run once every millisecond.
;
;       Input variable(s) : time_base_lo,time_base_hi,msec_lo,msec_hi
;                           seconds, minutes, hours, days
;       Output variable(s) : msec_lo,msec_hi
;                           seconds, minutes, hours, days
;       Variable(s) affected : time_base_lo,time_base_hi,msec_lo,msec_hi
;                           seconds, minutes, hours, days
;       Flag(s) affected :
;       Size : 17/39/45 bytes (depending upon clock type)
;              + 1 if bank select needed
```

```
;       Timing (turbo) : [99.9% of time] 14 cycles
;                        [0.1% of time] 17/39/45 cycles (or less)
;                      + 1 if bank select needed
;
;               BANK    clock                   ;select clock register bank
                MOV     W,#int_period           ;load period between interrupts
                ADD     time_base_lo,W          ;add it to time base
                SNC                             ;skip ahead if no underflow
                INC     time_base_hi            ;yes overflow, adjust high byte
                MOV     W,#tick_hi              ;check for 1 msec click
                MOV     W,time_base_hi-W        ;Is high byte above or equal?
                MOV     W,#tick_lo              ;load instr. count low byte
                SNZ                             ;If hi byte equal, skip ahead
                MOV     W,time_base_lo-W        ;check low byte vs. time base
                SC                              ;skip ahead if low

;commented out because of path_switcher/pushbutton routines which use msec count
;               JMP     :done_clock             ;If not, end clock routine
                JMP     done_pbs                ;If not, end clock routine

:got_tick       CLR     time_base_hi            ;Yes, adjust time_base reg.'s
                SUB     time_base_lo,#tick_lo   ; leaving time remainder
                INCSZ   msec_lo                 ;And adjust msec count
                DEC     msec_hi                 ; making sure to adjust high
                INC     msec_hi                 ; byte as necessary

                IF      clock_type>0            ;do we want sec, min, hours?
                MOV     W,#mspersec_hi          ;check for 1000 msec (1 sec tick)
                MOV     W,msec_hi-W             ;Is high byte above or equal?
                MOV     W,#mspersec_lo          ;load #1000 low byte
                SNZ                             ;If hi byte equal, skip ahead
                MOV     W,msec_lo-W             ;check low byte vs. msec count
                SC                              ;skip ahead if low
                JMP     :done_clock             ;If not, end clock routine
                INC     seconds                 ;increment seconds count
                CLR     msec_lo                 ;clear msec counters
                CLR     msec_hi                 ;
                MOV     W,#60                   ;60 seconds per minute
                MOV     W,seconds-W             ;are we at minute tick yet
                JNZ     :done_clock             ;if not, jump
                INC     minutes                 ;increment minutes count
                CLR     seconds                 ;clear seconds count
                MOV     W,#60                   ;60 minutes/hour
                MOV     W,minutes-W             ;are we at hour tick yet?
                JNZ     :done_clock             ;if not, jump
                INC     hours                   ;increment hours count
                CLR     minutes                 ;clear minutes count
                ENDIF                           ;<if> we wanted sec, min, hours

                IF      clock_type>1            ;do we want to count days?
                MOV     W,#24                   ;24 hours per day
                MOV     W,hours-W               ;are we at midnight?
                JNZ     :done_clock             ;if not, jump
                INC     days                    ;increment days count
                CLR     hours                   ;clear hours count
                ENDIF                           ;<if> we wanted day count
:done_clock
;
;****** Virtual Peripheral: Path Switch
;
; This routine allows alternating execution of multiple modules which don't
; need to be run during every interrupt pass in order to reduce the overall
; execution time of the interrupt on any given pass (i.e. it helps the code
; run faster).
; This version runs with the software clock virtual peripheral msec_lo variable
; allowing altenation between the switch positions once each millisecond.
;
```

```
;       Input variable(s) : msec_lo
;       Output variable(s) :
;       Variable(s) affected :
;       Flag(s) affected :
;       Size : 3 bytes + 1 bytes per jump location
;       Timing (turbo) : 8 cycles
;
:path_switch    MOV     W,msec_lo               ;load switch selector byte
                AND     W,#00000011b            ;keep low 2 bits - 4 position
                JMP     PC+W                    ;jump to switch position pointer
:pos0           JMP     pb0                     ;pushbutton 0 checking routine
:pos1           JMP     pb1                     ;pushbutton 1 checking routine
:pos2           JMP     pb2                     ;pushbutton 2 checking routine
:pos3           JMP     pb3                     ;pushbutton 3 checking routine
;
;
;****** Virtual Peripheral: Push Buttons*
;
; This routine monitors any number of pushbuttons, debounces them properly
; as needed, and flags the main program code as valid presses are received.
; *Note: this routine requires the Time Clock virtual peripheral or similar
;        pre-processing timer routine.
;
;       Input variable(s) : pb0_down,pb1_down,debounce0,debounce1
;                           pb2_down,pb3_down,debounce2,debounce3
;       Output variable(s) : pb0_pressed, pb1_pressed, pb2_pressed, pb3_pressed
;       Variable(s) affected : debounce0, debounce1, debounce2, debounce3
;       Flag(s) affected : pb0_down,pb1_down,pb0_pressed,pb1_pressed
;                          pb2_down,pb3_down,pb2_pressed,pb3_pressed
;       Size : 12 bytes per pushbutton + actions (see below**)
;              + 1 byte if path switch not used
;       Timing (turbo) : 7,10, or 12 cycles/pushbutton (unless path switch used)
;                        + actions (see below**)
;
pb0
;               BANK    buttons                 ;select bank (if not done elsewhere)
                JB      button0,:pb0_up         ;button0 pressed?
                JB      pb0_down,:done_pb0      ;yes, but is it new press?
                INC     debounce0               ; and adjust debounce count
                JNB     debounce0.hold_bit,:done_pb0   ;wait till long enough
                SETB    pb0_down                ;yes, flag that button is down

;**If the button activity is short (a few bytes), it can fit here, though be
; careful that longest possible interrupt doesn't exceed int_period # of cycles.
;
; <short code segment can go here>
;
;**Otherwise, use this flag to process button press in main code (and don't
; forget to reset the flag once the button activity is complete).
                SETB    pb0_pressed             ; and set pb0 action flag

                SKIP                            ;skip next instruction
:pb0_up         CLRB    pb0_down                ;button up, clear flag
                CLR     debounce0               ; and clear debounce count
:done_pb0
;
                JMP     done_pbs                ;this needed only if path switch used


pb1
                IF      num_buttons>1           ;more than 1 push button?
;               BANK    buttons                 ;do bank select (if not done elsewhere)
                JB      button1,:pb1_up         ;button1 pressed?
                JB      pb1_down,:done_pb1      ;yes, but is it new press?
                INC     debounce1               ; and adjust debounce count
                JNB     debounce1.hold_bit,:done_pb1   ;wait till long enough
                SETB    pb1_down                ;yes, flag that button is down
```

```
;**If the button activity is short (a few bytes), it can fit here, though be
; careful that longest possible interrupt doesn't exceed int_period # of cycles.
;
; <short code segment can go here>
;
;**Otherwise, use this flag to process button press in main code (and don't
; forget to reset the flag once the button activity is complete).
               SETB    pb1_pressed           ; and set pb1 action flag

               SKIP                          ;skip next instruction
:pb1_up        CLRB    pb1_down              ;button up, clear flag
               CLR     debounce1             ; and clear debounce count
:done_pb1
;
               JMP     done_pbs              ;this needed only if path switch used
               ENDIF                         ;more than 1 push button

pb2
               IF      num_buttons>2         ;more than 2 push buttons?
;              BANK    buttons               ;do bank select (if not done elsewhere)
               JB      button2,:pb2_up       ;button2 pressed?
               JB      pb2_down,:done_pb2    ;yes, but is it new press?
               INC     debounce2             ; and adjust debounce count
               JNB     debounce2.hold_bit,:done_pb2  ;wait till long enough
               SETB    pb2_down              ;yes, flag that button is down

;**If the button activity is short (a few bytes), it can fit here, though be
; careful that longest possible interrupt doesn't exceed int_period # of cycles.
;
;**Otherwise, use this flag to process button press in main code (and don't
;  orget to reset the flag once the button activity is complete).
               SETB    pb2_pressed           ; and set pb2 action flag

               SKIP                          ;skip next instruction
:pb2_up        CLRB    pb2_down              ;button up, clear flag
               CLR     debounce2             ; and clear debounce count
:done_pb2
;

               JMP     done_pbs              ;this needed only if path switch used
               ENDIF                         ;more than 2 push buttons

pb3
               IF      num_buttons>2         ;more than 3 push buttons?
;              BANK    buttons               ;do bank select (if not done elsewhere)
               JB      button3,:pb3_up       ;button3 pressed?
               JB      pb3_down,:done_pb3    ;yes, but is it new press?
               INC     debounce3             ; and adjust debounce count
               JNB     debounce3.hold_bit,:done_pb3  ;wait till long enough
               SETB    pb3_down              ;yes, flag that button is down

;**If the button activity is short (a few bytes), it can fit here, though be
;  careful that longest possible interrupt doesn't exceed int_period # of cycles.
;
;**Otherwise, use this flag to process button press in main code (and don't
;  forget to reset the flag once the button activity is complete).
               SETB    pb3_pressed           ; and set pb3 action flag

               SKIP                          ;skip next instruction
:pb3_up        CLRB    pb3_down              ;button up, clear flag
               CLR     debounce3             ; and clear debounce count
:done_pb3
               ENDIF                         ;more than 3 push buttons
;
done_pbs
;
```

```
done_int        mov     w,#-int_period          ;interrupt every 'int_period' clocks
                retiw                           ;exit interrupt
;
;****** End of interrupt sequence
;
;************************* RESET ENTRY POINT ****************************
;
reset_entry
;               PAGE    start                   ;Set page bits and then
;               JMP     start                   ; jump to start of code
;
;*********************
;* Main Program Code *
;*********************
;
start           mov     !rb,#%00001111           ;Set RB in/out directions
                CLR     FSR                      ;reset all ram starting at 08h
:zero_ram       SB      FSR.4                   ;are we on low half of bank?
                SETB    FSR.3                    ;If so, don't touch regs 0-7
                CLR     IND                      ;clear using indirect addressing
                IJNZ    FSR,:zero_ram            ;repeat until done

                MOV     !OPTION,#%10011111      ;enable rtcc interrupt
;
Main:loop
;
;the following code watches pb0-pb3 for presses and acts on them
button_check
;               BANK    buttons                 ;select pb bank
                MOV     W,pbflags               ;load pushbutton flags
                AND     W,#00001111b            ;keep only 'pressed' flags
                JZ      :no_press               ;jump ahead if not pressed
                MOV     temp,W                  ;store flags temporarily
                CLR     temp2                   ;clear 2nd temp storage reg.
:which_pb       INC     temp2                   ;increment 2nd temp value
                RR      temp                    ;check which button
                SC                              ;skip ahead if not this one
                JMP     :which_pb               ;keep looping
                MOV     W,--temp2               ;get 2nd temp value (less 1)
                MOV     temp,W                  ;save it in temp
                MOV     W,#11110000b            ;get clear mask for pbflags
                AND     pbflags,W               ;clear all "pressed" flags
                MOV     W,temp                  ;get which button pressed
                JMP     PC+W                    ;Go do PB routines
:pb0            JMP     pb0_action              ;do pb0 action
:pb1            JMP     pb1_action              ;do pb1 action
:pb2            JMP     pb2_action              ;do pb2 action
:pb3            JMP     pb3_action              ;do pb3 action

:no_press
;
;       <main program code goes here>
;
                JMP     Main:loop                   ;back to main loop
;
pb0_action
;
;       <pb0 action here>
;
                JMP     Main:loop
;
pb1_action
;
;       <pb1 action here>
;
                JMP     Main:loop
```

```
;
pb2_action
;
;       <pb2 action here>
;
                JMP     Main:loop
;
pb3_action
;
;       <pb3 action here>
;
                JMP     Main:loop
;
;***************
;               END                             ;End of program code
```