ECE:3360 – Lab 2 Report

Oliver Emery

23 February 2022

1 Introduction

The goal of this lab was to construct a simple stopwatch using shift registers, 7-segment displays, and two buttons.

The stopwatch must implement two modes with different timer resolutions. In Mode I, the stopwatch should begin with "0.0" displayed. When the first button is pressed, the display should increment every 0.1 seconds, updating the display to "0.1", "0.2", and so on until the display reaches "9.9". Pushing the first button while the stopwatch is counting up should stop the counter and freeze the display. Pressing the first button while paused should continue the timer. Once 9.9 seconds have elapsed in Mode I, the display should flash "9.9" once every two seconds. Pressing the second button for less than a second in any state should stop the counter and reset the stopwatch to 0.

In Mode II the stopwatch should function identically to Mode I. However, the timer should increment in steps of 1 second instead of 0.1 seconds. Similarly, the stopwatch should now show "00" at the start and "99" for overflows.

Pressing the second button for more than 1 second should reset the stopwatch to 0 and alternate between modes I and II.

2 Schematic

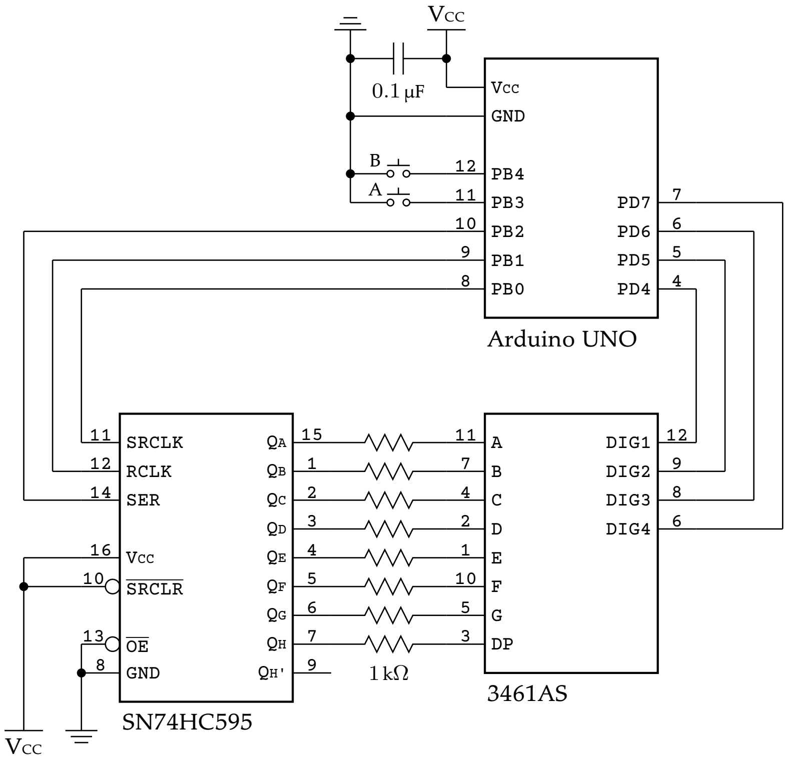

Figure 1: schematic as implemented

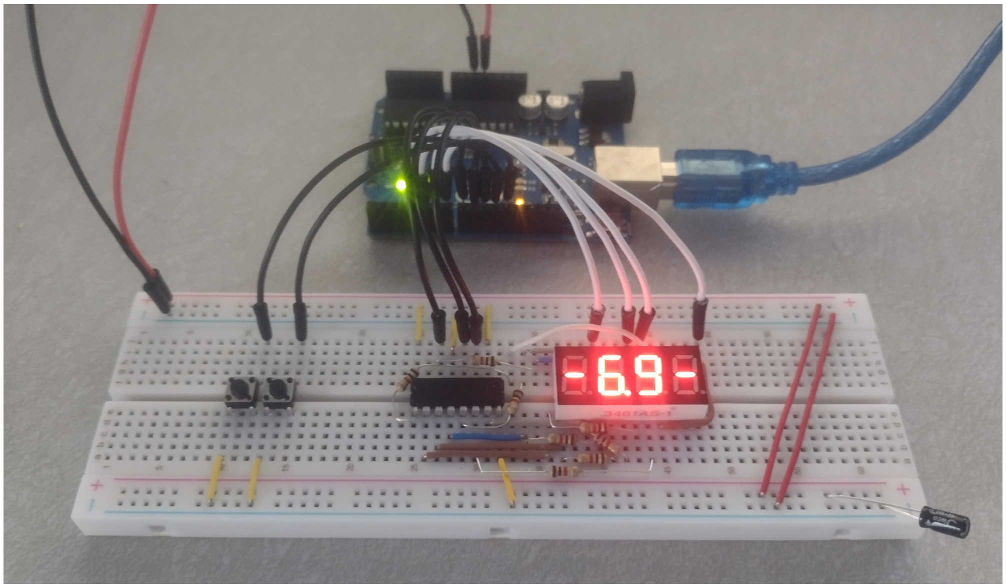

Figure 2: physical implementation

3 Discussion

I used $1\,\mathrm{k}\Omega$ resistors for the seven-segment display in order to keep the current through each segment well beneath the target value of $6\,\mathrm{mA}$. I also included a standard $0.1\,\mu\mathrm{F}$ decoupling capacitor to smooth ripples in the power supply.

3.1 Hardware Design

The final design was almost identical to that prescribed in the lab manual, with the exception of the four-digit display. Fortunately, the pinout of the 3461AS was nearly identical to that of the 5161AS, and only required one additional wire for each digit.

3.2 Buttons and Debouncing

Each button has an associated structure in SRAM containing relevant information. I aimed to keep the debouncing logic as simple as possible: register a change in button state if and only if it maintains that changed state over a minimum period of time. An unregistered change occurs when the button state in memory differs from the button's hardware state. The final implementation requires 50 stable samples at $1\,\mathrm{ms}$ intervals to register a change.

Also included in the button structure is a duration field, incremented every $100\,\mathrm{ms}$ when a button is in the pressed state. This value is used when button B is released to determine whether or not to change the mode.

3.3 Four-digit Seven-segment Display

In pursuit of extra credit, I created the design with a four-digit seven-sigment display. The 3461AS can only display one unique digit at a time, with power controlled by one pin for each digit. One study[1] found that humans can detect flicker at an extreme of $500\,\mathrm{Hz}$. With a system clock of $16\,\mathrm{MHz}$, a refresh rate as large as $1\,\mathrm{KHz}$ still affords an entire 16,000 cycles to spend per refresh. For each refresh, the stopwatch must display each digit for a distinct but equal period of time.

The shift register supports clock speeds up to $20\,\mathrm{MHz}$; the final implementation requires approximately $8\,\mu\mathrm{s}$ (lines 574-634) to shift one byte into the shift register. No display digit is powered while the shift register is loaded. In order to maximize the duty cycle of each digit, it is ideal for each digit to be powered substantially longer than it is not during its 25% of each refresh. Lines 635-648 below produce a delay of approximately $48\,\mu\mathrm{s}$, resulting in a final duty cycle of

$$25\% \cdot \frac{48}{8+48} \approx 21.4\%$$

for each digit. Unsurprisingly, reducing the duty cycle dims the digits. Each refresh in the final implementation takes approximately $230\,\mu\mathrm{s}$ (lines 576-611), which corresponds to a refresh rate of $4.35\,\mathrm{kHz}$. Indeed, no flicker is detectable with the naked eye.

To control power to the display digits, one I/O pin is assigned to each of the four digits. While this method is simple and convenient, it would also be feasible to use another shift register in lieu of I/O pins to control power to each digit. While this would increase I/O pin availability, it would increase hardware cost and increase code complexity marginally.

[1] https://www.nature.com/articles/srep07861

See source comments in Appendix A for more local and detailed discussion.

A Source Code Listing

```
;; project: ece3360-lab02
;; file:    main.S
;; date:    20220223
;; author:  Oliver Emery
;;
;;   The main subroutine of our program performs exactly one function: digit
;;   display. Because the 4x7 segment display I use requires multiplexing,
;;   we must continuously cycle power through each digit of the display, and at
;;   a high enough frequency to avoid flickering.
;;
;;   At 1ms intervals, a timer interrupt is called. If 100ms have passed, the
;;   stopwatch value is incremented. This happens first so the value will be
;;   incremented at strictly constant 100ms intervals. Next, both buttons are
;;   run through the debouncing algorithm, which calls the associated handler
;;   if a button state change is registered.
;;
;;   If we were to try handling all functionality in the main subroutine loop,
;;   cycles required would vary across iterations because of branching. While
;;   performing value increments at precise 100ms / 1.6M cycle intervals would
;;   still be possible, it would be unnecessarily complicated. Using timer
;;   interrupts allows us to separate the time-depedendent code from the rest.
;;
;;   Instead of using an entire IO pin for each digit of the display, I could
;;   have used none. I would have used an additional shift register chained to
;;   the current one. The bits of the new register would be used to control the
;;   currently powered digit.
;;
;;   There weren't enough subroutines or data being passed around, so I didn't
;;   try to establish any sort of calling convention. In anything called by an
;;   interrupt registers MUST be preserved to avoid nasty bugs, but I spread
;;   registers out across several of the display subroutines called in main()
;;   to avoid unneeded stack access.
;;
;; DO=DS40002061B (ATmega48A/PA/88A/PA/168A/PA/328/P Datasheet)
;;
.include "m328Pdef.inc"

;; ********************************** Defines **********************************
        ; inputs from pushbuttons
        .equ    P_BTN_A = PINB3
        .equ    P_BTN_B = PINB4

        ; outputs to shift register
        .equ    P_SER   = PINB0
        .equ    P_RCLK  = PINB1
        .equ    P_SRCLK = PINB2

        ; outputs to 4x7 segment digit pins
        .equ    P_DIG0  = PIND4
        .equ    P_DIG1  = PIND5
        .equ    P_DIG2  = PIND6
        .equ    P_DIG3  = PIND7

        ; timer states
        .equ    S_RESET = 0x01
        .equ    S_COUNT = 0x02
        .equ    S_STOP  = 0x04
        .equ    S_OFLOW = 0x08

        .equ    SEG_COUNT = 4
        .equ    DIG_COUNT = 2

        ; short blink on/off 2^(BLINK_POW-1) times every BLINK_LONG
        .equ    BLINK_POW       = 2
        ; in tenths of a second
        .equ    BLINK_LONG      = 20
        .equ    BLINK_SHORT     = 3

        .equ    OFLOW_STATE_INIT = BLINK_LONG

        ; debounce window
        .equ    BTN_WND_MSEC    = 50

        ; 100 is 1:1 / realtime
        .equ    SUBDIV_MS       = 100
        ; scaling factor of mode `B'
        .equ    RES_MODESCALE   = 10

        ; struct btn_s {
        .equ    btn_pressed     = 0x00  ; 1 if button is pressed, else 0
        .equ    btn_mask        = 0x01  ; 1 << PIN#
        .equ    btn_dwnd        = 0x02  ; detect window
        .equ    btn_duration    = 0x03  ; duration pressed
        .equ    btn_handler     = 0x04  ; change handler subroutine
        ; }
        .equ    sz_btn          = 6


;; ***************************** Global Variables ******************************
.dseg
.org 0x0100
        subdiv_scaler:  .byte 1
        mode_scaler:    .byte 1
        ; stopwatch
        current_state:  .byte 1
        oflow_state:    .byte 1
        ; value displayed on stopwatch
        current_value:  .byte SEG_COUNT
        ; a struct btn_s for button A and one for B
        button_a:       .byte 6
        button_b:       .byte 6


;; ************************** Interrupt Vector Table ***************************
.cseg
; [D0:7.7,12.4]
        .org 0x0000     jmp __reset
        ; counter0 compare match A handler
        .org OC0Aaddr   jmp __isr_oc0a


;; ******************************** Constants *********************************
.org INT_VECTORS_SIZE

; [0-9], '-', + null byte to keep arvasm2 from complaining
digit_bits: .db \
        0b00111111, 0b00000110, 0b01011011, 0b01001111, 0b01100110, \
        0b01101101, 0b01111101, 0b00000111, 0b01111111, 0b01101111, \
        0b01000000, 0


;; **************************** Interrupt Handlers *****************************

;; void __reset()
;;
;;     Called at system reset. Performs initialization tasks, and then transfers
;;     to main().
;;
__reset:
        ; stack pointer must be defined before calls can be made or interrupts
        ; enabled [D0:7.5]
        ldi     r16, high(RAMEND)
        out     SPH, r16
        ldi     r16, low(RAMEND)
        out     SPL, r16
        ; configure MCU functions before interrupts are enabled
        call    init
        ; enable interrupts
        sei
        ; transfer execution to main
        jmp     main

;; void __isr_oc0a()
;;
;;     Timer 0 compare match A handler. Called every 16,000 cycles / 1 ms.
;;     Invokes handler subroutine at 100 ms intervals and processes raw button
;;     input.
;;
__isr_oc0a:
        push    r16
        in      r16, SREG
        push    r16
        push    YL

        lds     r16, subdiv_scaler
        dec     r16
        sts     subdiv_scaler, r16
        brne    __isr_oc0a_fi           ; if (--subdiv_scaler == 0) {

        ldi     r16, SUBDIV_MS
        sts     subdiv_scaler, r16      ;     subdiv_scaler = SUBDIV_MS
        rcall   every_subdiv            ;     every_subdiv()
__isr_oc0a_fi:                          ; }

        ldi     YL, low(button_a)
        rcall   debounce                ; debounce(button_a)
        ldi     YL, low(button_b)
        rcall   debounce                ; debounce(button_b)

        pop     YL
        pop     r16
        out     SREG, r16
        pop     r16
        reti

;; *************************** Setup and Entrypoint ****************************

;; void memclr(Y: void*, r16: len)
;;
;;     clear up to r16 bytes of SRAM at YH:YL
;;
memclr:
        push    r16
        push    r17
        push    YL

        clr     r17
memclr_loop:
        st      Y+, r17
        dec     r16
        brne    memclr_loop

        pop     YL
        pop     r17
        pop     r16
        ret

;; void init()
;;
;;     Called before interrupts are enabled. Configure I/O, timer module, and
;;     power settings.
;;
init:
        ; Configure Timer/Counter 0 to generate an interrupt every 1 ms. This
        ; is done with a combination of:
        ;     * /64 clock prescaling
        ;     * clear timer on compare match (CTC) mode
        ;
        ; In CTC mode, the timer counts to the value held in OCR0A, generates
        ; an interrupt, and then is automatically reset to 0. This allows for
        ; an additional arbitrary scaling factor (up to 256) on top of any
        ; prescaling.
        ;
        ; With /64 prescaling and CTC mode with a compare value of 250, an
        ; interrupt is generated every
        ;
        ;       (16 000 000 Hz / 64 / 250)^-1 = (1000 Hz)^-1 = 1 ms
        ;
        ; Register Configuration Documentation
        ;     OCR0A     [D0:15.9.4]           output compare register
        ;     TIMSK0    [D0:15.5,15.9.6]      compare interrupt enable
        ;     TCCR0A    [D0:15.7.2,15.9.1]    ctc mode
        ;     TCCR0B    [D0:17.1,15.9.2]      /64 prescaling
        ldi     r16, 249
        out     OCR0A, r16              ; OCR0A = 249
        ldi     r16, 1 << OCIE0A
        sts     TIMSK0, r16             ; TIMSK0 = 1 << OCIE0A
        ldi     r16, 1 << WGM01
        out     TCCR0A, r16             ; TCCR0A = 1 << WGM01
        ldi     r16, 1 << CS01 | 1 << CS00
        out     TCCR0B, r16             ; TCCR0B = 1 << CS01 | 1 << CS00

        ; IO setup
        ldi     r16, 1 << P_SER | 1 << P_RCLK | 1 << P_SRCLK
        out     DDRB, r16               ; DDRB = 1<<DDB2 | 1<<DDB1 | 1<<DDB0
        ; inputs
        ldi     r16, 1 << P_BTN_A | 1 << P_BTN_B
        out     PORTB, r16              ; PORTB = 1 << PORTB4 | 1 << PORTB3

        ldi     r16, 1 << P_DIG0 | 1 << P_DIG1 | 1 << P_DIG2 | 1 << P_DIG3
        out     DDRD, r16               ; DDRD = 0xf0
        out     PORTD, r16              ; PORTD = 0xf0

        ; enable sleep instruction and configure for idle mode [D0:10.11.1]
        ldi     r16, 1 << SE
        out     SMCR, r16               ; SMCR = 1 << SE

        ret

;; void main()
;;
;;      Main program entrypoint.
;;
main:
        ; Kind of a hack but works for a program this small. Since our data
        ; starts at offset 0x0100 and is shorter than 256 bytes, it will never
        ; be necessary to modify YH for indirect data references. ZH must be
        ; free for modification and use by the LPM and ICALL instructions.
        ldi     YH, 0x01
        ldi     ZH, high(digit_bits << 1)

        ; Initialize global state variables
        ldi     r16, S_RESET
        sts     current_state, r16      ; current_state = S_RESET
        ldi     r16, SUBDIV_MS
        sts     subdiv_scaler, r16      ; subdiv_scaler = SUBDIV_MS
        ldi     r16, 0
        sts     mode_scaler, r16        ; mode_scaler = 0

        ; Initialize stopwatch value to zero
        ldi     r16, SEG_COUNT
        ldi     YL, low(current_value)
        rcall   memclr

        ; Initialize button structures
        ldi     r16, sz_btn
        ; button_a = { .mask = 1 << PIN_BUTTON0, .handler = button_a_changed }
        ldi     YL, low(button_a)
        rcall   memclr
        ldi     r17, 1 << P_BTN_A
        ldi     r18, high(button_a_changed)
        ldi     r19, low(button_a_changed)
        std     Y+btn_mask, r17
        std     Y+btn_handler, r18
        std     Y+btn_handler+1, r19

        ; button_b = { .mask = 1 << PIN_BUTTON1, .handler = button_b_changed }
        ldi     YL, low(button_b)
        rcall   memclr
        ldi     r17, 1 << P_BTN_B
        ldi     r18, high(button_b_changed)
        ldi     r19, low(button_b_changed)
        std     Y+btn_mask, r17
        std     Y+btn_handler, r18
        std     Y+btn_handler+1, r19

main_forever:                   ;     do {
        lds     r16, current_state
        cpi     r16, S_OFLOW
        brne    main_forever_show       ; if (current_state == S_OFLOW) {
        lds     r16, oflow_state        ;     if (oflow_state &
        sbrs    r16, (8 - BLINK_POW)    ;           (1 << (8 - BLINK_POW))) {
        rjmp    main_forever_show       ;         sleep();
        sleep                           ;         continue;
        rjmp    main_forever            ;     }
main_forever_show:                      ; }
        rcall   show_digits             ; show_digits();
        rjmp    main_forever    ;     } while (1);


;; **************************** Control Subroutines ****************************

;; void every_subdiv()
;;
;;     Called every 100 ms. Handles incrementing stopwatch value, display
;;     blinking in overflow state, and tracking button press duration.
;;
every_subdiv:
        push    r16
        push    r17
        push    YL

        lds     r16, current_state
        cpi     r16, S_COUNT
        brne    every_subdiv_elsif      ; if (current_STATE == S_COUNT) {

        lds     r16, mode_scaler
        tst     r16
        breq    every_subdiv_count_inc  ;     if (mode_scaler) {

        dec     r16
        sts     mode_scaler, r16        ;         if (--mode_scaler > 0) {
        brne    every_subdiv_fi         ;             goto every_subdiv_fi

        ldi     r16, RES_MODESCALE      ;         }
        sts     mode_scaler, r16        ;         mode_scaler = RES_MODESCALE

every_subdiv_count_inc:                 ;     }
        rcall   inc_value               ;     inc_value()
        rjmp    every_subdiv_fi         ; }

every_subdiv_elsif:
        cpi     r16, S_OFLOW
        brne    every_subdiv_fi         ; else if (current_state == S_OFLOW) {

        lds     r16, oflow_state
        mov     r17, r16
        andi    r16, 0xff >> BLINK_POW  ;     byte bwnd = oflow_state & 0x3f;
        andi    r17, (0xff << (8 - BLINK_POW)) & 0xff
        dec     r16                     ;     byte bctr = oflow_state & 0xc0;
        breq    every_subdiv_oflow_blink    ; if (--bctr > 0) {
        or      r16, r17
        sts     oflow_state, r16        ;         oflow_state = bwnd | bctr;
        rjmp    every_subdiv_fi
every_subdiv_oflow_blink:               ;     } else {
        ldi     r16, 1 << (8 - BLINK_POW)
        add     r17, r16                ;         bctr += 1 << (8- BLINK_POW);
        brne    every_subdiv_oflow_blink_blip   ; if (!bctr) {
        ori     r17, BLINK_LONG         ;             bctr |= BLINK_LONG;
        rjmp    every_subdiv_oflow_blink_fi
every_subdiv_oflow_blink_blip:          ;         } else {
        ori     r17, BLINK_SHORT        ;             bctr |= BLINK_SHORT;
every_subdiv_oflow_blink_fi:            ;         }
        sts     oflow_state, r17        ;         oflow_state = bctr;
every_subdiv_fi:                        ; } }

        ; Update duration counter on pressed buttons
        ldi     YL, low(button_a)
        rcall   button_inc_duration     ; button_inc_duration(button_a)
        ldi     YL, low(button_b)
        rcall   button_inc_duration     ; button_inc_duration(button_b)

every_subdiv_ret:
        pop     YL
        pop     r17
        pop     r16
        ret

;; void button_a_changed(Y: *button, r16: is_pressed)
;;
;;     Called when button A is detected as pressed or released. Controls state
;;     transitions caused by button A.
;;
button_a_changed:
        tst     r16
        breq    button_a_changed_ret            ; if (!is_pressed) return;

        lds     r16, current_state              ; switch (current_state) {
        cpi     r16, S_RESET
        brne    button_a_changed_case_count     ; case S_RESET:
        ldi     r16, S_COUNT                    ;     current_state = S_COUNT;
        rjmp    button_a_changed_sto            ;     break;
button_a_changed_case_count:
        cpi     r16, S_COUNT
        brne    button_a_changed_case_stop      ; case S_COUNT:
        ldi     r16, S_STOP                     ;     current_state = S_STOP;
        rjmp    button_a_changed_sto            ;     break;
button_a_changed_case_stop:
        cpi     r16, S_STOP
        brne    button_a_changed_sto            ; case S_STOP:
        ldi     r16, S_COUNT                    ;     current_state = S_COUNT;
button_a_changed_sto:
        sts     current_state, r16              ; }

button_a_changed_ret:
        ret

;; void button_b_changed(Y: *button, r16: is_pressed)
;;
;;     Called when button B is detected as pressed or released. Controls state
;;     transitions caused by button B; regardless of the current state,
;;     releasing B will revert the current state back to the RESET state.
;;
;;     Also toggles stopwatch timescale / "mode" if button was held for at
;;     least <9> tenths of a second.
;;
button_b_changed:
        push    r16
        push    YL

        tst     r16
        brne    button_b_changed_ret    ; if (is_pressed) return;

        ldd     r16, Y+btn_duration
        cpi     r16, 9
        brlo    button_b_changed_no_modeswitch
                                        ; if (btn->duration >= 9) {
        lds     r16, mode_scaler
        tst     r16
        brne    button_b_changed_to_mode1   ; if (!mode_scaler) {

        ldi     r16, RES_MODESCALE      ;         mode_scaler = RES_MODESCALE;
        sts     mode_scaler, r16        ;     }
        rjmp    button_b_changed_no_modeswitch
button_b_changed_to_mode1:              ;     else {
        clr     r16                     ;         mode_scaler = 0;
        sts     mode_scaler, r16        ;     }
button_b_changed_no_modeswitch:         ; }

        clr     r16
        std     Y+btn_duration, r16     ; btn->duration = 0;
        ldi     r16, S_RESET
        sts     current_state, r16      ; current_state = S_RESET;

        ldi     r16, SEG_COUNT
        ldi     YL, low(current_value)
        rcall   memclr                  ; current_value = "0000";

button_b_changed_ret:
        pop     YL
        pop     r16
        ret


;; **************************** Button Subroutines *****************************

;; void button_inc_duration(YL: *button)
;;
;;     Increment the duration field of the passed button.
;;
button_inc_duration:
        ldd     r16, Y+btn_pressed
        tst     r16
        breq    button_inc_duration_ret ; if (btn->pressed) {

        ldd     r16, Y+btn_duration
        inc     r16                     ;     // prevent overflow
        breq    button_inc_duration_ret ;     if (btn->duration + 1) {
```

```
        std     Y+btn_duration, r16     ;         btn->duration++;
button_inc_duration_ret:                ;     }
        ret                             ; }

;; void debounce(YL: *button)
;;
;;     Sample and process raw button input data to reliably detect and handle
;;     button events. Big idea: register a change in button state if and only
;;     if it holds the changed state steady for a specified window of time.
;;
debounce:
        push    r0
        push    r1
        push    r16
        push    ZH
        push    ZL

        clr     r16
        in      r0, PINB
        ldd     r1, Y+btn_mask
        and     r0, r1
        brne    debounce_notpressed
        inc     r16
debounce_notpressed:            ; byte pressed = (PINB & btn->mask) ? 0 : 1;

        ldd     r0, Y+btn_pressed
        cp      r16, r0
        breq    debounce_coda           ; if (btn->pressed != pressed) {

        ldd     r0, Y+btn_dwnd
        dec     r0
        std     Y+btn_dwnd, r0
        brne    debounce_ret            ; if (--btn->dwnd) return;

        std     Y+btn_pressed, r16      ;     btn->pressed = pressed;
        ldd     ZH, Y+btn_handler
        ldd     ZL, Y+btn_handler+1
        ; lol totally unnecessary with only 2 buttons
        icall                           ;     btn->handler();
debounce_coda:                          ; }
        ldi     r16, BTN_WND_MSEC
        std     Y+btn_dwnd, r16         ; btn->dwnd = WND_MSC;

debounce_ret:
        pop     ZL
        pop     ZH
        pop     r16
        pop     r1
        pop     r0
        ret


;; **************************** Display Subroutines ****************************

;; void inc_value()
;;
;;     Increment the current stopwatch value in memory. If it hits the maximum,
;;     enter overflow state and leave the value maximized.
;;
inc_value:
        push    r16
        push    YL

        ldi     YL, low(current_value)  ; byte i = 0;

inc_value_loop:
        ld      r16, Y
        inc     r16
        cpi     r16, 10
        brne    inc_value_exit  ; while (current_value[i] + 1 == 10) {

        clr     r16
        st      Y+, r16         ;     current_value[i++] = 0;

        cpi     YL, low(current_value + DIG_COUNT)
        brne    inc_value_loop  ;     if (i == DIG_COUNT) {

        ; FIXME - should use DIG_COUNT
        ldi     r16, 9
        ldi     YL, low(current_value)
        st      Y+, r16
        st      Y, r16

        ldi     r16, S_OFLOW
        sts     current_state, r16      ; current_state = S_OFLOW;
        ldi     r16, OFLOW_STATE_INIT
        sts     oflow_state, r16        ; oflow_state = OFLOW_STATE_INIT;

        rjmp    inc_value_ret           ; return; }

inc_value_exit:                 ; }
        st      Y, r16          ; current_value[i]++;

inc_value_ret:
        pop     YL
        pop     r16
        ret

;; void show_digits()
;;
;;      Display current stopwatch value on 4x7 segment display. Gimped version
;;      for 2 digits so our friendly TA doesn't have to wait a minimum of 999.9
;;      seconds to verify our overflow functionality. Writes hyphens on the
;;      outer digits and the current value on the middle two.
;;
show_digits:
        push    r16
        push    r17
        push    r18

        ; decimal point position
        clr     r18
        lds     r1, mode_scaler
        tst     r1
        breq    show_digits_not_ones
        inc     r18
show_digits_not_ones:

        ldi     YL, low(current_value)

        ldi     r16, 1
        ld      r17, Y+
        rcall   write_digit

        ldi     r16, 2
        ld      r17, Y
        dec     r18
        rcall   write_digit

        ; hyphens on digits 0 and 3
        ldi     r17, 10
        clr     r18
        clr     r16
        rcall   write_digit
        ldi     r16, 3
        rcall   write_digit

        pop     r18
        pop     r17
        pop     r16
        ret
```

```

;; void write_digit(r16: index, r17: charn, r18: decimal)
write_digit:
        ldi     ZL, low(digit_bits << 1)
        add     ZL, r17
        lpm     r19, Z

        tst     r18
        breq    write_digit_no_dp
        ori     r19, 1 << 7
write_digit_no_dp:

        rcall   put_sr_byte

        mov     r20, r16
        inc     r20
        ldi     r21, ~(1 << 4)
        ; can we get a barrel shifter up in here plx
write_digit_while:
        lsr     r21
        dec     r20
        brne    write_digit_while

        swap    r21
        andi    r21, 0xf0
        out     PORTD, r21

        ; This just needs to be decently longer than the time it takes to load
        ; the shift register. Each digit only gets 25% of total display time,
        ; so we want to maximize the proportion of on time to off time. As it
        ; stands, write_digit up to here takes ~8us (off), and the remainder
        ; takes ~48us (on); each digit is on for ~85% of its period.
        ldi     r19, 255
write_digit_delay:
        dec     r19
        brne    write_digit_delay

        ldi     r19, 0xf0
        out     PORTD, r19

        ret

;; void put_sr_byte(r19: byte)
;;
;;     Put a byte into the shift register.
;;
put_sr_byte:
        ldi     r20, 8
put_sr_byte_while:
        rol     r19
        brcs    put_sr_byte_while_hibit
        cbi     PORTB, P_SER
        rjmp    put_sr_byte_wend
put_sr_byte_while_hibit:
        sbi     PORTB, P_SER
put_sr_byte_wend:
        ; trigger SRCLK, shifting SER into the shift register. note that there
        ; is no need for a delay: even if SBI/CBI only took 1 clock cycle, the
        ; SN74HC595N supports up to 20 MHz while the UNO runs at only 16 MHz
        sbi     PORTB, P_SRCLK
        cbi     PORTB, P_SRCLK

        dec     r20
        brne    put_sr_byte_while

        ; trigger RCLK to transfer shift register data to the storage register
        sbi     PORTB, P_RCLK
        cbi     PORTB, P_RCLK

        ret

.exit
```