Embedded Systems Hardware

PBL

Date : 12/12/2022

Guilherme Guimaraes Ruas
Submitted to: Nick Markou

PURPOSE

As detailed in the Project 2022 document, the purpose of this activity was to condense various skills and theory to produce a system that:

- Manipulates memory mapped devices
- Uses serial communication to send and receive data using USART
- Uses the processor's interrupt to respond to asynchronous events
- Examine signals and output processed information

CIRCUIT DIAGRAM

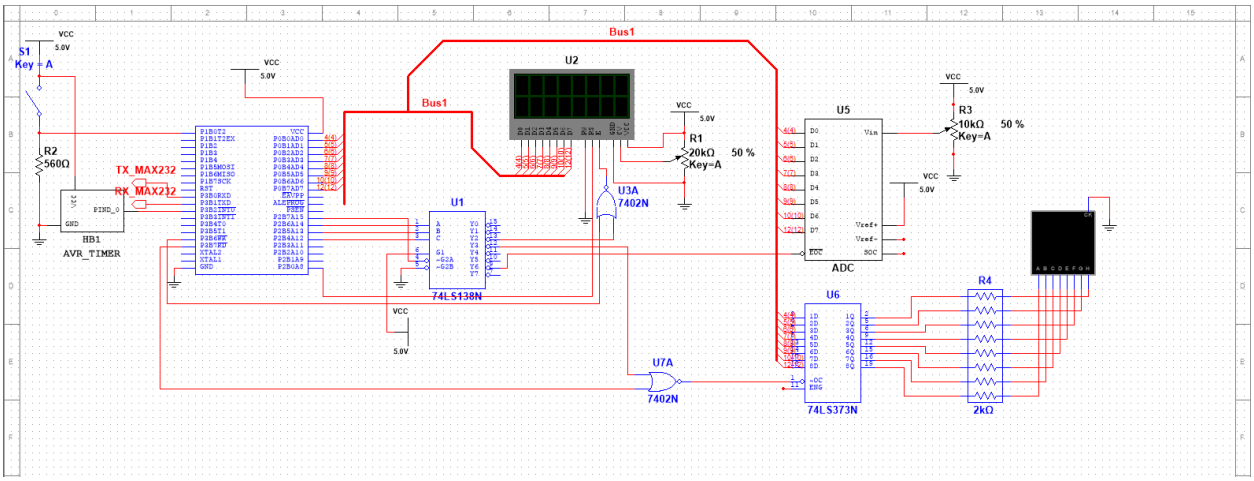

Fig 0: Circuit Diagram

PROGRAM CODE

main.asm

```
.dseg
.equ BUSON_INT0 = $82       ;Modified MCUCR macro to allow for interrupt 0
.equ FLAG1 = $060    ;Running state flag
.equ FLAG2 = $061    ;Running sub-state flag
.equ FLAG3 = $062    ;State change flag
.equ HH = $063              ;Higher-than-threshold counter storage
.equ LL = $064              ;Lower-than-threshold counter storage
.equ HHmsb = $065    ;HH ASCII convertion
.equ HHlsb = $066
.equ LLmsb = $067    ;LL ASCII convertion
.equ LLlsb = $068
```

```
.equ TRIM = $069     ;Trim level storage
.equ MAX = 3         ;MAX input buffer chars

.org $75
input_buff: .byte MAX

.org $80
seconds: .db 1
minutes: .db 1
hours: .db 1
seg_secondH: .db 1
seg_secondL: .db 1

.org $100
sample_buff: .db 256

.org $205
.equ samplesumHH = $205
.equ samplesumHL = $206
.equ samplesumLH = $207
.equ samplesumLL = $208

.cseg
reset:  rjmp init0
init_int0:    rjmp isr0     ;Interrupt 0 sbr

.org  $20
init0:
       ldi R16, low(RAMEND)    ;first things first, init stack pointer
       out SPL, R16
       ldi R16, high(RAMEND)
       out SPH, R16
       ldi r16, $00
       out DDRB, r16
       ldi r16, $FF
       out DDRE, r16

       clr r19                     ;Time keeping seconds register
       clr r20                     ;Time keeping minutes register
       clr r21                     ;Time keeping hours register
       inc r21                     ;This is done to start the clock at 01:00:00 to
demonstrate
                                   ;resetting function
       sts seconds, r19
       sts minutes, r20
       sts hours, r21
       clr r24
       clr r25
       clr r0
       sts FLAG1, r0 ;Clearing that memory space to use it as an 8 bit Running-state flag
       sts FLAG2, r0 ;Clearing that memory space to use it as an 8 bit Running-substate
flag
       sts FLAG3, r0 ;This is what i call a state-change flag, and it stays up for one
second before it is cleared.
                     ;It serves to indicate to external loops (such as gets) that it
must exit to main to have the other state-flags evaluated

init_bus:
```

```
	ldi r17, $40
	out GICR, r17		;Int0 mask
	ldi r17, BUSON_INT0
	out MCUCR, r17		;Initiation of bus + interrupt 0
	clr r17
			;Setting External Interrupt flag

init_serial:
	rcall init_uart

init_lcd:
	rcall lcd_start		;see lcd.inc

init_banner:		;Initial banner at boot-up
	ldi r31, high(buff_lcd<<1)
	ldi r30, low(buff_lcd<<1)
	rcall clcd_outs1	;see lcd.inc
	ldi r31, high(buff_lcd0<<1)
	ldi r30, low(buff_lcd0<<1)
	rcall clcd_outs2	;see lcd.inc

	rcall newl
	ldi   R30, low(banner<<1)   ;UART banner
	ldi   R31, high(banner<<1)
    rcall puts

	rcall delay_4s		;4 second delay using brute force sbr because i cant
use the interrupt for this
					;or the clock will appear before the time has
passed

init_vars:		;Initializing dseg vars
	ldi r16, $7F
	sts TRIM, r16
	sts HH, r0		;values and also makes the hex to ASCII conversion for ease of
access later in the main display loop
	sts LL, r0
	sei

;The main code loop functions as a finite state machine that evaluates and manages global
flags to determine which
;state it is in currently, and how to interpret the input it receives. The two main
states are Node and Supervisor,
;and the rest of the sub-states are taken care of by those previous two.

main:
	sts FLAG3, r0
	lds r16, FLAG1
	cpi r16, $01
	breq main_node1

main_sup:
	rcall lcd_start

//	lds r17, FLAG2
//	cpi r17, $0
//	brne sup_loop
```

```
       ldi r17, $E0               ;In case of invalid substate flag, this will default to
$E0
       sts FLAG2, r17

sup_loop:
       ldi r31, high(buff_lcd1<<1)
       ldi r30, low(buff_lcd1<<1)
       rcall clcd_outs1
       ldi r31, high(buff_lcd4<<1)
       ldi r30, low(buff_lcd4<<1)
       rcall clcd_outs2

       rcall newl
       ldi   R30, low(buff_utx1<<1)
       ldi   R31, high(buff_utx1<<1)
    rcall puts
       rcall getdata
       rcall input_check_sup

       rjmp fini_sup

main_node1:   rjmp main_node

sup_loop_T:
       push r16
       mov r16, r21
       rcall hex2binary

       push r17
       ldi r17, $CB
       sts $2000, r17
       rcall delay1ms
       pop r17
       push r17
       rcall clcd_outs_xmanual
       mov r17, r16
       rcall clcd_outs_xmanual
       ldi r17, ':'
       rcall clcd_outs_xmanual

       mov r16, r20
       rcall hex2binary
       rcall clcd_outs_xmanual
       mov r17, r16
       rcall clcd_outs_xmanual

       mov r16, r19
       rcall hex2seg                     ;This returns the 0 offset of each char with
R17:R16. This is to be used
                                                 ;together with the seg7code array
to display the correct
                                                 ;char on a 7seg display
       cli
       push r16
       mov r16, r17
       sts seg_secondH, r16
       pop r16
       sts seg_secondL, r16
```

```
        rcall update_led

        pop r17
        pop r16
        sei

exit_T:
        ret

sup_loop_P:

        ret

sup_loop_R:
        clr r21
        clr r20
        clr r19
        ret

update_led:
        lds r16, seg_secondL
        ldi r31, HIGH(seg7code<<1)
        ldi r30, LOW(seg7code<<1)
        add r30, r16
        lpm r16, Z
        sts $3000, r16
        ret

fini_sup: rjmp main

main_node:                  ;In their respective main loops, the sub-state flags are
checked against references to determine
                            ;which sub-state to enter. This is repeated at the end of
every loop since they all return to main
                            ;to have the flags constantly re-evaluated.
        rcall lcd_start
        lds r17, FLAG2
        cpi r17, $D0        ;Reference towards ADC displaying
        breq node_loop_D
        cpi r17, $B0        ;Reference towards ADC SUM displaying
        breq node_loop_S1
        cpi r17, $E0        ;Reference towards ADC TRIM level modifying
        breq node_loop_T

node_loop:
        ldi r31, high(buff_lcd2<<1)
        ldi r30, low(buff_lcd2<<1)
        rcall clcd_outs1
        ldi r31, high(buff_lcd3<<1)
        ldi r30, low(buff_lcd3<<1)
        rcall clcd_outs2

        rcall newl
        ldi   R30, low(buff_utx2<<1)
        ldi   R31, high(buff_utx2<<1)
    rcall puts
        rcall getdata
```

```
       rcall input_check    ;This function is the one that will reset and change sub-state
flags according to input.
                                          ;It only ever needs to be called after an input
has been processed.
       rjmp fini_node

node_loop_D:               ;This loop contains custom LCD display code to be able to
display individual segments of string.
                                          ;This loop is responsible for displaying the ADC
sampling that is stored in their respective memories in dseg.
       ldi r31, high(buff_lcd2<<1)
       ldi r30, low(buff_lcd2<<1)
       rcall clcd_outs1

       ;Output TRIM level
       ldi r31, high(buff_nodeT<<1)
       ldi r30, low(buff_nodeT<<1)
       rcall clcd_outs_manual
       push r17
       lds r16, TRIM
       rcall hex2asc
       rcall clcd_outs_xmanual           ;TRIM msd
       mov r17, r16
       rcall clcd_outs_xmanual           ;TRIM lsd
       pop r17

       ;Output HIGH count to LCD
       ldi r31, high(buff_nodeH<<1)
       ldi r30, low(buff_nodeH<<1)
       push r17
       ldi r17, $C0
       sts $2000, r17
       rcall clcd_outs_manual
       pop r17
       push r17
       lds r17, HHmsb
       rcall clcd_outs_xmanual
       lds r17, HHlsb
       rcall clcd_outs_xmanual
       pop r17

       ;Output LOW count to LCD
       ldi r31, high(buff_nodeL<<1)
       ldi r30, low(buff_nodeL<<1)
       rcall clcd_outs_manual
       push r17
       lds r17, LLmsb
       rcall clcd_outs_xmanual
       lds r17, LLlsb
       rcall clcd_outs_xmanual
       pop r17

       rcall newl
       ldi   R30, low(buff_utx2<<1)
       ldi   R31, high(buff_utx2<<1)
    rcall puts
       rcall getdata        ;Always checking for input and reverting back to main, in case
any flag has changed.
```

```
       rcall input_check

       rjmp fini_node

node_loop_S1: rjmp node_loop_S

node_loop_T:
       rcall newl
       ldi   R30, low(adc_trim<<1)
       ldi   R31, high(adc_trim<<1)
    rcall puts
       rcall getdata
       ldi r31, high(input_buff)
       ldi r30, low(input_buff)
       push r16
       clr r17

node_T_check:
       inc r17
       ld r16, Z+

       cpi r16, $30
       brlo node_T_error

       cpi r16, $40
       brlo node_T_success

       cpi r16, $41
       brlo node_T_error

       cpi r16, $47
       brlo node_T_success

node_T_error:
       rcall newl
       ldi   R30, low(adc_trim_error<<1)
       ldi   R31, high(adc_trim_error<<1)
    rcall puts
       rjmp node_T_fini

node_T_success:
       cpi r17, 1
       breq node_T_check

       push r16
       ldi r31, high(input_buff)
       ldi r30, low(input_buff)
       rcall asc2hex
       sts TRIM, r16
       pop r16

node_T_fini:
       sts FLAG2, r0
       pop r16
       rjmp fini_node

node_loop_S:
       ldi r31, high(buff_lcd2<<1)
```

```
       ldi r30, low(buff_lcd2<<1)
       rcall clcd_outs1
       ldi r31, high(buff_nodeS<<1)
       ldi r30, low(buff_nodeS<<1)
       rcall clcd_outs2

       push r17
       lds r17, samplesumHH
       rcall clcd_outs_xmanual
       lds r17, samplesumHL
       rcall clcd_outs_xmanual
       lds r17, samplesumLH
       rcall clcd_outs_xmanual
       lds r17, samplesumLL
       rcall clcd_outs_xmanual
       pop r17

       rcall newl
       ldi   R30, low(buff_utx2<<1)
       ldi   R31, high(buff_utx2<<1)
    rcall puts
       rcall getdata        ;Always checking for input and reverting back to main, in case
any flag has changed.
       rcall input_check

       rjmp fini_node

fini_node:
       rjmp main


;This below is the interrupt sbr. The AVR's INT0 pin is receiving a high clockpulse at
1KHz (or as close as possible to it).
;The isr will then count up to a second using the special arithmetic reg pair 25:24.

isr0:
       push r16                    ;This step was very important, since if we do not save
SREG's state,
                                   ;the URTX portions simply do not work with a constant
interrupt
       in r16, SREG
       push r16
       adiw r25:r24, 1
       cpi r25, $01
       breq delay1s_check
       pop r16
       out SREG, r16
       pop r16
       reti

delay1s_check:
       cpi r24, $F4
       breq delay1s
       pop r16
       out SREG, r16
       pop r16
       reti
```

```
;The program is made to trigger lots of things at the 1second mark. Instead of checking
inside a loop for
;a change in the seconds counter, it is better to use the isr's 1 second trigger to make
nested calls
;to other sbrs. WARNING: this can make the isr become veeery lengthy and thus, the more
code put in here the
;less accurate the time keeping will also become. This is still less time consuming than
checking states
;inside a program loop.

delay1s:
	inc r19
	sts seconds, r19
	sts minutes, r20
	sts hours, r21
	clr r25
	clr r24
delay1s_sbr:
	rcall clock_update		;Internal clock update function (has no mode for
displaying yet)
//	sts FLAG3, r0			;Clearing state change flag in the next second
	rcall mode_switch		;This function will check at every second, for a change
in the DIP switch that indicates
							;a change in running-states (Svr or
Node). Any change will reset the program to the main
							;loop & clear the sub-state flag.
	lds r16, FLAG1
	cpi r16, $0
	brne exit_int
	rcall sup_modecheck

	;rcall display_seconds	;This function displays in the terminal, the ticking of
seconds in HEX. Used for debugging as of now.

exit_int:
	pop r16
	out SREG, r16
	pop r16
	reti

sup_modecheck:
	lds	r16, FLAG3
	cpi	r16, $0
	brne exit
	lds r17, FLAG2
	cpi r17, $E0			;FLAG2 State flag for T_sup mode = $E0
	breq sup_T_call
	cpi r17, $B0			;FLAG2 State flag for P_sup mode = $B0
	breq sup_P_call
	cpi r17, $C0			;FLAG2 State flag for R_sup mode = $C0
	breq sup_R_call

	ret

sup_T_call:
	rcall sup_loop_T
	ret
```

```
sup_P_call:
	rcall sup_loop_P
	ret

sup_R_call:
	rcall sup_loop_R
	ret


clock_update:
	cpi r19, $3C
	brne exit
	clr r19
	inc r20
	cpi r20, $3C
	brne exit
	clr r20
	inc r21
	cpi r21, $18
	brne exit
	clr r21
exit:
	ret

mode_switch:			;To stop unnecessary flag checking, this will ensure
that the flag has changed at all before updating its state
	push r17
	push r16
	in r17, PINB
	andi r17, $01
	lds r16, FLAG1
	cp r16, r17
	brne mode_switch_chk
mode_switch_continue:
	pop r16
	pop r17
	ret

mode_switch_chk:
	sts FLAG1, r17			;inserting new state into flag1
	inc r17
	sts FLAG3, r17			;inserting non-zero value in flag3 (state change
flag)
	sts FLAG2, r0		;resetting flag2 after state change
	rjmp mode_switch_continue

display_seconds:			;Debugging terminal-output of seconds (stored in R19)
	push r16
	push r17
	mov r16, r19
	rcall hex2asc
	push r16
	mov r16, r17
	rcall putchar
	pop r16
	rcall putchar
	rcall newl
	pop r17
```

```
	pop r16
	ret

adc_sample:		;This function will perform the ADC sampling and will divide
it into two different
				;memory spaces that represent each a counter.
	push r16
	push r17
	push r18
	clr r17
	clr r18
adc_sample_loop:
	ldi r16, '.'
	rcall putchar
	ldi r16, $FF
	out PORTE, r16
	rcall ms2_wait
	mov r16, r0
	out PORTE, r16
	lds r16, $6000
	ldi r27, HIGH(sample_buff)
	ldi r26, LOW(sample_buff)
	st X+, r0

	lds r18, TRIM
	cp r16, r18
	brsh adc_hh
	cp r16, r18
	brlo adc_ll
adc_sample_continue:
	rcall ms2_wait
	inc r17
	cpi r17, $FF
	brne adc_sample_loop
	pop r18
	pop r17
	pop r16
	ret

adc_hh:
	push r16
	lds r16, HH
	inc r16
	sts HH, r16
	pop r16
	rjmp adc_sample_continue

adc_ll:
	push r16
	lds r16, LL
	inc r16
	sts LL, r16
	pop r16
	rjmp adc_sample_continue

ms2_wait:			;Accurate 2ms counter
	push r16
	mov r16, r24
```

```
       inc r16
ms2_wait_loop:
       cp r16, r24
       brne ms2_wait_loop
       pop r16
       ret

input_getbuff:                          ;The input will be stored in R16, therefore it
is a MUST to push and pop R16 around the sbr call to this
       push r18
       ldi r29, high(input_buff)
       ldi r28, low(input_buff)
input_getbuff_loop:
       ld r18, Y+
       cpi r18, CR
       breq input_getbuff_exit
       mov r16, r18
       rjmp input_getbuff_loop
input_getbuff_exit:
       pop r18
       ret

input_check:                    ;Must store R16 around this function call
       push r16
       lds r16, FLAG3                   ;In case of invalid substate flag, this will
default to $E0
       cpi r16, $0
       brne input_check_continue
       rcall input_getbuff
       cpi r16, $73             ;This is where the menu options go to be branched into
their code portions.
       breq set_flag_S
       cpi r16, $61
       breq set_flag_D
       cpi r16, $74
       breq set_flag_T
input_check_continue:
       pop r16
       ret

set_flag_D:                     ;This function is triggered by the input_check sbr. It
will call the function to sample the ADC
       sts HH, r0               ;values and also makes the hex to ASCII conversion for
ease of access later in the main display loop
       sts LL, r0
       rcall newl
       ldi   R30, low(adc_start<<1)
       ldi   R31, high(adc_start<<1)
    rcall puts
       rcall adc_sample
       rcall newl
       ldi   R30, low(adc_stop<<1)
       ldi   R31, high(adc_stop<<1)
    rcall puts
       rcall set_flag_A

       rjmp  input_check_continue
```

```
set_flag_T:
       push r18
       ldi r18, $E0                              ;No T value in hex :/
       sts FLAG2, r18
       pop r18
       rjmp input_check_continue

set_flag_S:
       push r18
       ldi r18, $B0
       sts FLAG2, r18
       pop r18
       cli
       push r16
       push r17
       push r18
       push r19
       clr r18
       clr r16
       clr r19
       ldi r31, HIGH(sample_buff)
       ldi r30, LOW(sample_buff)

sample_cal:
       ld r17, Z+
       add r18, r17
       adc r19, r0
       inc r16
       brne sample_cal

       mov r16, r19
       rcall hex2asc
       sts samplesumHH, r17
       sts samplesumHL, r16
       mov r16, r18
       rcall hex2asc
       sts samplesumLH, r17
       sts samplesumLL, r16

       pop r19
       pop r18
       pop r17
       pop r16

       sei

       rjmp input_check_continue

set_flag_A:                          ;This function is triggered by the input_check sbr. It
will call the function to sample the ADC
       rcall newl
       ldi   R30, low(adc_display<<1)
       ldi   R31, high(adc_display<<1)
    rcall puts
       push r16
       push r15
       push r17
```

```
        push r18
        lds r16, HH
        rcall hex2asc
        sts HHmsb, r17
        sts HHlsb, r16
        lds r16, LL
        rcall hex2asc
        sts LLmsb, r17
        sts LLlsb, r16
        pop r18
        pop r17
        pop r15
        pop r16

        push r18
        ldi r18, $D0
        sts FLAG2, r18
        pop r18
        ret

seg7puts:
        ldi r31, HIGH(seg7code << 1)
        ldi r30, LOW(seg7code << 1)
        add r30, r16
        push r16
        push r17
        lpm r16, Z
        rcall hex2asc
        push r16
        mov r16, r17
        rcall putchar
        pop r16
        rcall putchar
        pop r17
        pop r16
        ret

input_check_sup:                        ;Must store R16 around this function call
        cli
        push r16
        lds r16, FLAG3                  ;In case of invalid substate flag, this will
default to $E0
        cpi r16, $0
        brne input_check_continue_sup
        rcall input_getbuff
//      rcall newl
//      rcall putchar
//      rcall newl
//      cpi r16, $72              ;This is where the menu options go to be branched into
their code portions.
//      brne set_flag_B
        rcall sup_loop_R
input_check_continue_sup:
        pop r16
        sei
        ret

.include "termio.inc"
```

```
.include "lcd.inc"
.include "numio.inc"
.exit
```

termin.io

```
;
; Filename: termio.inc
;
; The following code supplied to the Fall 2018 243-513-DW students for educational/study
; purposes. The use of these routines, in whole or in part, without proper reference to
; origin is a violation of ISEP regarding Academic Integrity and Plagerism.
;
;Description: Subroutine framework for use in the project (do not modify)
;
;Author: Mr. Markou

.equ  UBRR  = 25   ;see p.138 (important)
.equ  FRAME = $86      ;8N1 standard frame
.equ  TXE = $18        ;Transmit & receive enable
.equ  LF = $0A                ;ASCII line feed
.equ  CR = $0D                ;ASCII carriage return
.equ  NUL = 0        ;string terminating value
.equ  EOL = 0        ;string terminating value
.equ  EOT = $04             ;string terminating value

;
; Standard USART init which logically belongs here
;
init_uart:
       ldi R16, 0           ;always zero (mostly)
       out UBRRH, R16
       ldi R16, UBRR
       out UBRRL, R16     ;config. the rate of data tx
       ldi R16, TXE
       out UCSRB, R16     ;enable port tx (see p.158)
       ldi R16, FRAME     ;defined in calling
       out UCSRC, R16     ;config. frame elements
       ret
;
;*gets - asm workalike C routine which inputs from UART RxD
; Entry: Z reg must point to buffer
;        R17 must contain the size of the buffer.
gets:
       dec   R17            ;leave room for NUL w/o exceeding buffer limit
gtx:
    rcall getche
       st       Z+, R16
       cpi      R16, CR      ;check for returm
       breq  gty
       dec      R17
       brne  gtx
```

```
gty:
       ldi   R16, NUL     ;place end of string
       st    Z, R16
       ret
;
;*getch - asm workalike C routine to receive char from UART
; Exit:  R16 contain rx char
getch:
       push  r16
       lds      r16, FLAG3
       cpi      r16, 0
       brne  getch_ret
       pop   r16
       in    R16,UCSRA
       andi  R16, $80
       breq  getch             ;poll status for key press
       in       R16,UDR      ;get char from UART
       ret
;
;*getche - like above but w/echo
; Exit:  R16 contain rx char
getche:
       rcall getch
       push  R16          ;save R16 on stack
       rcall putchar
       pop  R16           ;and retrieve it back
       ret
;
;*putchar - char tx UART routine
; Entry: R16  char. to send
putchar:
       cli
    out  UDR,R16        ;txmt char. out the TxD
putc1:
    in   R16, UCSRA    ;poll status reg
       andi R16, $20      ;check for tx complete
       breq putc1
       sei
       ret
;
;*puts - asm workalike routine to puts() in C
; Entry: Z index points to an NUL terminated string in cseg or dseg
puts:
       cli
       push r16
puts_loop:
       lpm   R16, Z+      ;get char from memory
       cpi   R16, NUL
       breq  px               ;this is the end, my only friend, the end
       rcall putchar      ;send it to uart
       rjmp  puts_loop
px:
       pop r16
       sei
       ret
;
;*newl - issues a new line (CR&LF) which comes in handy
; Entry: R16
```

```
newl:
       cli
       push r16
       ldi   R16, LF     ;nothing new here
    rcall putchar
       ldi   R16, CR
    rcall putchar
       pop r16
       sei
       ret

getch_ret:
       pop r16
       ret

;----------
getdata:                    ;getdata sbr inspired by the one done in class.
       push r16
       ldi   R17, MAX              ;leave room for a final CR
       ldi   R30, low(input_buff)  ;point Z-reg and buffer
       ldi   R31, high(input_buff)
       rcall gets
       pop r16
       ret

;.include "termio.inc"   ;append library subroutines from same folder
;Dedicated space for string buffers for both the LCD and UART. LCD strings are terminated
by $A0 and UART ones, by a NUL
;LED
;The following code is programmed for a 7seg display connected in PORTC
;with the following order: g, f, a, b, c, d, e
seg7code: .db $3f, $0c, $5b, $5e, $6c, $76, $77, $1c, $7f, $7c, $7d, $67, $33, $4f, $73,
$71, $0d, 0

;LCD
buff_lcd: .db "SCADA Mon 1.3v ", $A0
buff_lcd0: .db "ID: 1834747", $A0
buff_lcd1: .db "Supervisor ", $A0
buff_lcd2: .db "Node ", $A0
buff_lcd3: .db "Mode Standby ", $A0
buff_lcd4:    .db "Mode Stdby ", $A0
buff_nodeS: .db " S:", $A0
buff_nodeT: .db " T=", $A0
buff_nodeL: .db " L:", $A0
buff_nodeH: .db "H:", $A0, 0

;UART
banner: .db "SCADA Mon 1.3v", LF, CR, "ID: 1834747", LF, CR, NUL
buff_utx1: .db "Svr#", NUL, 0
buff_utx2: .db "Node [a, s, t]>", NUL
adc_start: .db "-----Sampling Voltages, please wait----- ", LF, CR, NUL
adc_stop: .db "-----Sampling done!----- ", LF, CR, NUL
adc_display: .db "-----Displaying Statistics to LCD-----", LF, CR, NUL, 0
adc_trim: .db "Insert new trim level in HEX. (USE UPPERCASE ONLY)", LF, CR,"[$00 >= x >=
$FF]> ", NUL
adc_trim_error: .db "!!!INVALID INPUT, RETRY!!! ", LF, CR, NUL
sample_set: .db "-----Displaying Sample Set Sum to LCD-----", LF, CR, NUL, 0
```

numio.inc

```
;
; Filename: numio.inc
;
; The following code supplied to the Fall 2018 243-513-DW students for educational/study
; purposes. The use of these routines, in whole or in part, without proper reference to
; origin is a violation of ISEP regarding Academic Integrity and Plagerism.
;
;Description: Subroutine framework for use in the project (do not modify)
;
;Author: Mr. Markou
;
;Changelog:
;1v0 11/10/17 - finalized and commented version
;2v0 11/11/18 - forked into this version with extra routine
;2.1 11/12/19 - added hex2bcd8
;
;----------------
;*hex2asc - converts an 8 bit hex value into a valid ASCII characters by
;*         masking and shifting ASCII character into a valid binary form
; Entry: R16 should contain the hex number to convert
; Usage: R15,R17,R18 **save before calling rtn if used
; Exit:  R17:R16 contain msd:lsd respectively
hex2asc:
        cli
     clr   R17
     mov   R15,R16      ;make copy of hex number
     lsr   R16          ;shift ms nyble to ls
     lsr   R16
     lsr   R16
     lsr   R16
h2a:
     andi  R16,$0F     ;mask off upper nyble
     ldi   R18,$30
     add   R16,R18     ;add $30 to adjust to ascii 0 - 9
     cpi   R16,$3A     ;check if valid bcd range
     brlo  h2b
     ldi   R18,7       ;adjust to hex char into 'A' - 'F'
     add   R16,R18
h2b:
     push  R16                ;save msd ascii
     mov   R16,R15
     inc   R17
     cpi   R17,1
     breq  h2a
        pop  R16           ;R17:R16 contain msd:lsd ascii numbers
     pop  R17
        sei
     ret

;*asc2hex - converts a pair of ASCII characters into an 8bit number
;                   by subtracting from ASCII values and adding both individual digits
;                   together
; Entry: Z should contain the ASCII pair to convert
; Usage: R15,R17,R18 **save before calling rtn if used
```

```
; Exit:  R16 will contain the 8bit hex value

asc2hex:
     clr   R17
a2h:
        lds r16, FLAG3
        cpi r16, $0
        brne hex_exit
        ld r16, Z+
        ldi r18, $30
        sub r16, r18
        cpi r16, $A
        brlo b2h
        ldi r18, 7
        sub r16, r18
b2h:
     inc   R17
     cpi   R17,2
        breq  c2h
        lsl r16
        lsl r16
        lsl r16
        lsl r16
        push r16
        rjmp a2h
c2h:
        pop r17
        add r16, r17
     ret

;---------------
;* hex2BCD8 - 8-bit Binary to BCD rtn which converts a hex number
;             from $00 - $63 to $00 - $99 in bcd
; Entry: R16 should contain the hex number to convert
; Exit: R16 contains converted number
hex2bcd8:
       cli
       push    R17      ;save on stack in case of zombies
       clr    R17       ;clear R16 reg
msd:
       push r16
       lds    r16, FLAG3
       cpi    r16, $0
       brne hex_exit
       pop r16
       subi   R16,10    ;input = input - 10
       brcs   lsd       ;abort if carry set
       subi   R17,-$10   ;R17 = R17 + 10
       rjmp   msd       ;loop again
lsd:
       subi   R16,-10          ;compensate extra subtraction
       add           R16,R17
       pop     R17      ;restore it back
hex_exit:
       sei
       ret

;---------------
```

```
;* hex2binary - 16bit hex number convertion rtn that utilizes above routines
;                           to translate hex into *displayable* binary numbers.
;                           This rtn is only useful for displaying, and should never be
used
;                           for arithmetic.
; Entry: R16 should contain the hex number to convert
; Exit: R17:R16 contain the converted binary characters
hex2binary:
       rcall hex2bcd8
       rcall hex2asc
       ret

;---------------
;* hex2binary - 16bit hex number convertion rtn that utilizes above routines
;                           to translate hex into *displayable* binary numbers.
;                           This rtn is only useful for displaying, and should never be
used
;                           for arithmetic.
; Entry: R16 should contain the hex number to convert
; Exit: R17:R16 contain the converted binary characters
hex2seg:
       rcall hex2bcd8
       rcall bcd82seg
       ret

bcd82seg:
       cli
       push r16
       lds    r16, FLAG3
       cpi    r16, $0
       brne hex_exit
       pop r16
       push r0
       push r1
       push r20
       ldi r20, $10

       mov r17, r16 ;copying value for alteration
       andi r16, $0F ;isolating lower nibble
       mul r17, r20 ;shifting by 4 bits to the left.
                           ;we are only interested in the first byte of thespecial mul
pair
                           ;therefore, only r0
       mov r17, r1

       pop r20
       pop r1
       pop r0
       sei
       ret
;
;//insert following two directives to main program a line before .exit
;.nolist
;.include "numio.inc"   ;append library subroutines from same folder
```

lcd.inc

```
;Header file for anything LCD related. This comes prebuilt with the LCD initialization
routine, as well as some special delays and different output sbrs.

lcd_start:                  ;This sbr will init the LCD in 8bit, 2 line mode, and will
clear the display as well.
       push r16                     ;Pushing r16 to save its state in case it is important
       rcall delay40ms
       ldi r16, $38
       sts    $2000, r16            ;Over here, our memory has been split for memory
address $2000, where
                                           ;$2000 to $20FF is the DR and $2100 to 21FF is
IR
       rcall delay37us
       ldi r16, $38
       sts    $2000, r16

       rcall delay37us
       ldi r16, $0F
       sts    $2000, r16 ;DISPLAY ON

       rcall delay37us
       ldi r16, $01
       sts    $2000, r16 ;CLEAR DISPLAY

       rcall delay1_52ms
       ldi r16, $06
       sts    $2000, r16 ;ENTRY MODE SET

       rcall delay37us
       pop r16
       ret

delay37us:                  ;since each clockcycle is taken into account to make a 37us
delay,
       NOP                          ;I counted the clocks of rcall and ret, and padded with
NOPs :D lazy i know
       NOP                          ;but it works perfectly!
       NOP
       NOP
       NOP
       NOP
       NOP
       NOP
       NOP
       NOP
       NOP
       NOP
       NOP
       NOP
       NOP
       NOP
       NOP
       NOP
       NOP
       NOP
```

```
	NOP
	NOP
	NOP
	NOP
	NOP
	NOP
	NOP
	NOP
	NOP
	NOP

	ret

;Below are some precise timing delay functions that are not only required for LCD init
;but could also come in use for timing

delay1_52ms:
	push r18
	clr r18
delay_loop1_52ms:
	rcall delay37us
	inc r18
	cpi r18, 41		;1520us/37us is roughly 41
	breq fini_1_52ms
	rjmp delay_loop1_52ms
fini_1_52ms:
	pop r18
	ret

delay40ms:
	push r18
	clr r18
delay_loop40ms:
	rcall delay1_52ms
	inc r18
	cpi r18, 26		;40ms/1.5ms is roughly 26
	breq fini_40ms
	rjmp delay_loop40ms
fini_40ms:
	pop r18
	ret

delay1ms:
	push r18
	clr r18
delay_loop1ms:
	rcall delay37us
	inc r18
	cpi r18, 27		;1000us/37us is roughly 27
	breq fini_1ms
	rjmp delay_loop1ms
fini_1ms:
	pop r18
	ret

;clcd_outs1 will output a message to the first LCD line while keeping the second intact
;The following message buffer templates must be followed for it to work in CSEG
;***Before calling it, use Z index for the message buffer***
```

```
;
;       buff1: .db "First LCD line", $A0
;
;clcd_outs2 will output a message to the second LCD line while keeping the first intact
;The following message buffer template must be followed for it to work in CSEG
;***Before calling it, use Z index for the message buffer***
;
;       buff2: .db "Scnd LCD line", $A0
;
;$A0 is the termination character
;
clcd_outs1:
        push r17
        ldi r17, $80
        sts $2000, r17
        rcall delay1ms
clcd_outs_loop1:
        lpm r17, Z+
        sts $2100, r17
        rcall delay1ms
        cpi r17, $A0
        brne clcd_outs_loop1
        pop r17
        ret

clcd_outs2:
        push r17
        ldi r17, $C0
        sts $2000, r17
        rcall delay1ms
clcd_outs_loop2:
        lpm r17, Z+
        sts $2100, r17
        rcall delay1ms
        cpi r17, $A0
        brne clcd_outs_loop2
        pop r17
        ret

clcd_outs_manual:                   ;Must Push R17 with appropriate custom LCD control
signals to use this funtion
        rcall delay1ms
clcd_outs_manual_loop:
        lpm r17, Z+
        sts $2100, r17
        rcall delay1ms
        cpi r17, $A0
        brne clcd_outs_manual_loop
        ret

clcd_outs_xmanual:                  ;This function will output a single char

        rcall delay1ms                      ;Must Push R17 & Load R17 with appropriate
custom LCD control signals to use this funtion
clcd_outs_xmanual_loop:
        sts $2100, r17
        rcall delay1ms
        ret
```

```
delay_4s:
	push r18
	ldi R18, 10
second1:
	ldi r25, $ff
	ldi r24, $ff
second2:
	sbiw r25:r24, 1
	brne second2
	dec r18
	breq sec_fini
	rjmp second1
sec_fini:
	pop r18
	ret


.exit
```

AVRtimer.asm

```
.cseg
reset:  rjmp init0

init0:
	ldi R16, low(RAMEND)    ;first things first, init stack pointer
	out SPL, R16
	ldi R16, high(RAMEND)
	out SPH, R16
	ldi r17, $FF
	ldi r18, $00

	out DDRD, r17

main:
	out PORTD, r18
	rcall delay1ms
	out PORTD, r17
	rcall delay1ms
	rjmp main

delay37us:                  ;since each clockcycle is taken into account to make a 37us
delay,
	NOP                         ;I counted the clocks of rcall and ret, and padded with
NOPs :D lazy i know
	NOP                         ;but it works perfectly!
	NOP
	NOP
	NOP
```

```
        NOP
        NOP
        NOP
        NOP
        NOP
        NOP
        NOP
        NOP
        NOP
        NOP
        NOP
        NOP
        NOP
        NOP
        NOP
        NOP
        NOP
        NOP
        ret


delay1ms:
        push r18
        clr r18
delay_loop1ms:
        rcall delay37us
        inc r18
        cpi r18, 27                ;1000us/37us is roughly 27
        breq fini_1ms
        rjmp delay_loop1ms
fini_1ms:
        pop r18
        ret
```

RESULTS

The whole design of this circuit could not be produced to completion, below is a list of the requirements and their completion:

FEATURE	COMPLETION
Displaying power-up banner on both LCD and Terminal	Completed
Asynchronous switching between Supervisor and Node modes (in less than two seconds)	Completed
Node mode Reading from ADC and displaying statistics	Completed
Node mode changing TRIM level	Completed
Node mode calculating and displaying SUM of recorded ADC levels	Partially completed (see Discussion)
Supervisor mode asynchronous timekeeping	Completed
Supervisor mode displaying time on LCD	Completed
Supervisor mode displaying time on LED	Partially completed (see Discussion)
Supervisor mode Reset	Completed
Supervisor mode Pause	Not completed
Supervisor mode Toggle	Not completed
Overall input validation/error detection	Completed

Table 0: Feature completion list

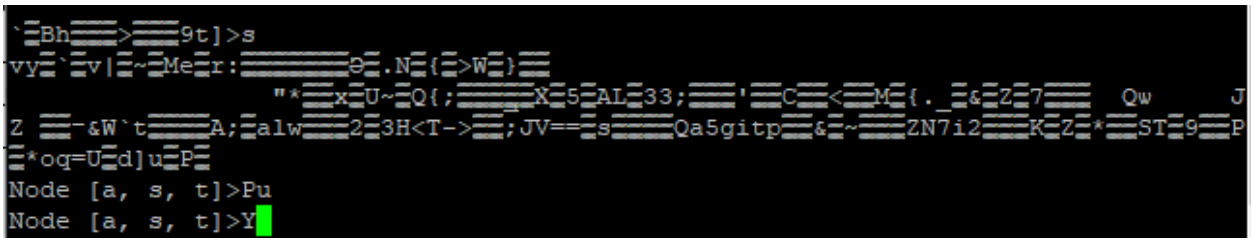

Fig 1: Sample calculation error first run

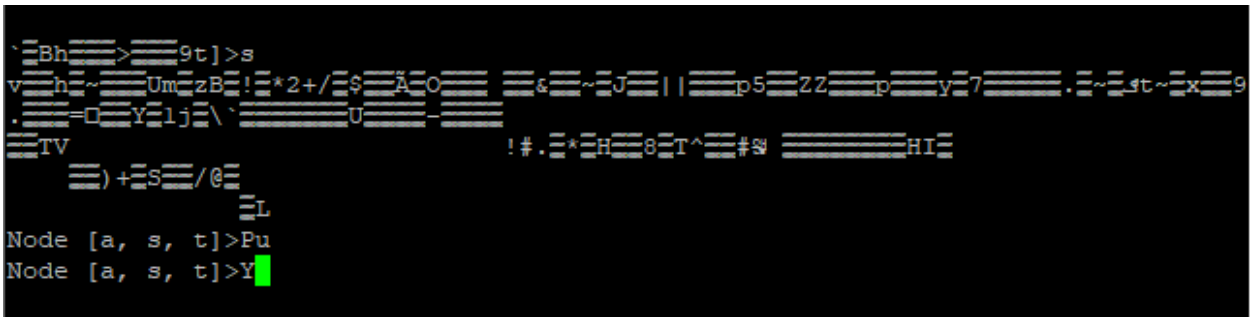

Fig 2: Sample calculation error second run

Fig 3: SVR input error

DISCUSSION

Register/Memory Summary

Register/Memory Location	Purpose
BUSON_INT0	Used to define bus and interrupt behavior
FLAG1 ($60)	Used to define global mode: Supervisor or Node
FLAG2 ($61)	Used to define sub-mode (user input)
FLAG3 ($62)	Used to indicate to sbrs that a global mode has changed, and they should exit so the program state could be re-evaluated. This flag appears in sbrs contained in the include files as well!
HH ($63)	Used to store Higher-than-threshold counter from ADC readings
LL ($64)	Used to store Higher-than-threshold counter from ADC readings
HHmsb ($65)	HH ASCII convertion (HIGH)
HHlsb ($66)	HH ASCII convertion (LOW)
LLmsb ($67)	LL ASCII convertion (HIGH)
LLlsb ($68)	LL ASCII convertion (LOW)
TRIM ($69)	Used to store TRIM level in hex
MAX ($70 - $72)	Macro to define maximum allowed number of characters input through the Terminal

input_buff ($75)	Buffer the size of MAX to store input
seconds ($80)	Used to store seconds in hex
minutes ($81)	Used to store minutes in hex
hours ($82)	Used to store hours in hex
seg_secondH ($83)	Used to store seconds hex conversion to 7 segment display code (HIGH)
seg_secondL ($84)	Used to store seconds hex conversion to 7 segment display code (LOW)
sample_buff ($100 - $1FF)	Used to store the 256 samples acquired from the ADC in hex
samplesumHH ($205)	Used to store ASCII value of the sample sum (HIGH byte HIGH nyble)
samplesumHL ($206)	Used to store ASCII value of the sample sum (HIGH byte LOW nyble)
samplesumLH ($207)	Used to store ASCII value of the sample sum (LOW byte HIGH nyble)
samplesumLL ($208)	Used to store ASCII value of the sample sum (LOW byte LOW nyble)
R19	Used for asynchronous count of seconds
R20	Used for asynchronous count of minutes
R21	Used for asynchronous count of hours
R25:R24	Special register pair used to count **isr0** pulses coming from the AVR.
$2000	LCD memory mapped device
$3000	LED memory mapped device
$6000	ADC memory mapped device

Table 1: Register/Memory Summary

Preamble

Welcome to my project's dissection! I've been eager to present the way I implemented my design since last year.

Firstly, to talk about my program, I must first introduce the notion of a **finite state machine**.

A finite state machine is a system that, as the name implies, utilizes a finite number of states to perform its functions. It goes through those states being guided by variables named **flags** that determine what code should be executed given an event.

The code will begin with default flag states, then an input will come disrupt that loop. An event handler takes care of setting the appropriate flags, performing some computation, then jumping back to the top of the code, where the previously set flags will indicate which instructions should be executed next, and so on and so forth.

I started my design with the intention of making a pure finite state machine, but with the complexity of the code increasing, I had to perform some actions given events that do not set any flags.

However, this design is highly modular, and allows anybody to simply check for a new character input, associate it to a new flag state, and create a new sbr to handle that input. The code will do the job of interpreting the states and updating them asynchronously.

It all starts at the beginning of the code, where I allocate some memory in the internal RAM for three different types of flags:

- FLAG1 is a value that determines which **mode** the program is in, and is changed by the input on PORTB(0) in response to an external logical switch.
- FLAG2 is a value that determines which **sub-mode** the program is in, and is changed by inputs originating from the USART, created by the user in a Terminal window.
- FLAG3 is a value that determines a **state change**, and is changed by any input belonging to this finite state machine architecture. The purpose of this flag is to tell subroutines that they should quit their operation and allow the code to re-evaluate the flags.

The rest of the program will be explained bellow, starting with the interrupt INT0.

Isr0

Before tacking the program, I must explain the functionality of the interrupt INT0 since the whole code is dependent on it. The rest of the major sbrs explained here will be detailed by order of appearance. **isr0** is an exception.

The interrupt subi, called **isr0**, is an interrupt triggered on the falling edge of an input signal, as determined by the macro BUSON_INT0 with a value of $82 which configures the config. reg. MCUCR.

This interrupt is called at the frequency determined by AVR1 (**see Program Code**) and contains an internal counter that will, through arithmetic, calculate a precise **second**, and at that moment it will call a special subroutine called **delay1s**.

The **delay1s** sbr firstly updates the memory locations for hours, minutes and seconds with the current values contained in the timekeeping registers. This subroutine will then call another one called **clock_update**. Keep in mind this is all happening asynchronously independent of mode or user input. **clock_update** will then change the values of registers R21:R20:R19 according to our conventional 24-hour time scheme. These registers translate directly to the timekeeping convention HH:MM:SS. This also means that running time is always being kept as long as interrupts have not been disabled.

The code then returns and calls the next sbr called **mode_switch**. This sbr will check if, from the last time it was called, the switch on PORTB(0) changed values. If it didn't, it will simply return. But if it did, this sbr will update the current state into FLAG1, reset FLAG2 and set FLAG3, so that it can tell sbrs to exit their loops and return to main, so the program can start a new mode.

Then finally, this is where the code stops being purely state driven when it calls the **sup_modecheck** sbr. I had to find a way to efficiently update the LCD and LED at every second, and so I decided to check for FLAG2 to determine which output mode the user had requested, and immediately calling a sbr to perform the displaying, without returning to main so that the flag could be checked.

This worked well enough and would've given me the opportunity to Toggle by outputting values differently to the LCD and LED, as well as Resetting the clock. This could not be done because of **SUP_INPUT_ERROR**.

It is important to note that all sbrs called by the interrupt this way must always find a way to return with **RETI**.

Init0

The label **init0** is the first one to be called, and it is called by the **reset** interrupt signal. It contains code to initialize the Stack, as well as the various registers and memory locations (such as flags) before the start of execution.

Init_bus

The label **init_bus** contains code to initialize the bus and the external interrupt settings.

Init_serial

The label **init_serial** calls the **init_uart** function found in **termio.inc** file to start the USART.

Init_lcd

The label **init_lcd** calls the **lcd_start** function found in **lcd.inc** file to start the LCD. This process is identical from Lab 9 and utilizes timing delay functions that are independent from the interrupt.

Init_banner

The **init_banner** label outputs the banner messages seen on both the Terminal and the LCD. The output to the LCD is done through the functions **clcd_outs1** and **clcd_outs2** which are found in the **lcd.inc** file.

The output to the Terminal is done through the functions **newl** and **puts** which are found in the **termio.inc** file.

There is a delay of 4 seconds before the execution of the following code.

Init_vars

This label contains other variable initialization code.

Main

The **main** label is the label that is returned to at the end of any input checking operation or state change.

The code in it is short, all it does is clear the FLAG3 flag in case main was executed from a state change, and then proceeds to checking FLAG1 to determine which mode the program is in.

If FLAG1 = $01, then the program is in Node mode, and it will jump to **main_node1**.

Otherwise, the program is in Sup mode, and it will continue to **main_sup**.

Main_sup

In this label, the program evaluates flag FLAG2 to determine which displaying mode it should jump to. This would allow me to use a pure state machine architecture, however shortcomings (see **SUP_INPUT_ERROR**) made me have to use an event handler instead.

This loop requires a default FLAG2 value to output the clock asynchronously, and so I set it to value **$E0**.

Sup_loop

In this label, a pattern for displaying is shown for the first time. The sbr takes care of displaying the text corresponding to the Supervisor mode to both the Terminal and the LCD.

It then calls the **getdata** sbr, which is used to retrieve data from the user.

After that, an sbr called **input_check_sup** is called to evaluate the input from the user and change states accordingly.

The loop will then automatically return to **main** if no state change occurs.

Sup_loop_T

This sbr contains code to display the clock on the LCD and LEDs. This sbr is only accessible through a function call that happens every second and is described in the

sup_modecheck sbr. The values for hours, minutes and seconds are stored as indicated by the Register Summary. The hours and minutes are translated into ASCII chars by the **hex2binary** sbr which is found in the **numio.inc** file. The seconds are translated by the **hex2seg** sbr which is also found in the same file.

The output to the LCD has been discussed previously. The output to the LEDs happens through the **update_led** sbr contained in the **main** file.

Sup_loop_P

This sbr would contain code to Pause the clock.

Sup_loop_R

This sbr contains code to Reset the clock by means of clearing the timekeeping registers R21:R20:R19. The **delay1s**, as you will see, is responsible for updating the .DSEG memory of these new values.

Update_led

This sbr will send the value stored in R16 to the memory mapped LED @ address $3000. It does this by receiving the value of R16 as a decimal number, loading the Z memory addresser with the address to memory location **seg7code**, then using it to offset the Z pointer to the correct digit to be displayed. The **seg7code** contains 17 values, 16 of them describing the 8bit hex value that will make a 7 segment LED display a character from 0 to F.

This sbr is incomplete and would've needed to call the $3000 range twice to multiplex between the two LEDs intended for the design. However, I only had time to install and test one LED.

Main_node

Similar to sup_node, this sbr will read the value for FLAG2 and then will enter the appropriate loop. If FLAG2 is not defined (like on mode switch), then the default label accessed is called **node_loop**.

Node_loop

This sbr outputs the default Node standby text on both the Terminal and the LCD.

The sbr will then wait for user input and return to main.

Node_loop_D

This sbr is called in response to the user inputting the letter “a” on the Terminal while in Node mode. After some operations done asynchronously, the flag FLAG2 will be loaded with $D0 and **main_node** will jump straight here.

In this sbr, statistics about the TRIM level and the ADC readings are displayed on the LCD. This is done through extensive use of functions defined in lcd.inc.

The sbr will then wait for user input and return to main.

Node_loop_T

This sbr is called in response to the user inputting the letter “t” on the Terminal while in Node mode. After some operations done asynchronously, the flag FLAG2 will be loaded with $E0 and **main_node** will jump straight here. Note that this flag state is also used in Supervisor mode, but the code will never mix up the two since the flag validation process is hierarchical.

In this sbr, the Terminal prompts the user for a new TRIM value.

The program will then cascade to the sbrs labeled **node_T_** that perform input error detection by seeing if the value inputted is within the hex boundaries of $00 to $FF.

The sbr will then return to main.

Node_loop_S

This sbr is called in response to the user inputting the letter “s” on the Terminal while in Node mode. After some operations done asynchronously, the flag FLAG2 will be loaded with $B0 and **main_node** will jump straight here.

In this sbr, the program will output the SUM calculation that is performed right after the input.

The sbr will then wait for user input and return to main.

Sup_modecheck

This sbr is called every second and is responsible for updating the information required by Supervisor mode to display accurate timing asynchronously. It is called every second by the **delay1s** sbr.

Here we can see FLAG3 being used to exit the sbr in case the program state has changed, avoiding any unintentional asynchronous operations.

This sbr will asynchronously check the state of FLAG2 and jump to the appropriate label.

This sbr would allow the sup mode to perform a different displaying sequence depending on the current value of the Toggle function.

In the code that was demonstrated, it only managed to call **sup_loop_T** repeatedly to display the clock on the LCD and LED.

The modularity of the code can be seen in this sbr, as it would be trivial to check for a new sub-mode flag state and execute the appropriate new code.

Clock_update

This is an sbr mentioned in the **isr0** description. It contains various increment, clear and comparison operations that efficiently update the timekeeping registers with respect to the 24 hour convention system.

Mode_switch

This is an sbr mentioned in the **isr0** description. It contains code that will read PORTB pin 0, and decide if it has changed since the last reading. If it didn't, the sbr will simply return.

But if it does, the sbr jumps to the **mode_switch_chk** label, which does 3 things:

- Stores new mode state reading into FLAG1 address.
- Changes the value of FLAG3 to a non-zero value to indicate a state change.
- Clears FLAG2 and allows the main functions to either enter default sub-state value in it, or to wait for user input.

The sbr will then return.

display_seconds

This sbr is absent from the program loop and is used to debug the correct operation of both the isr0 and clock_update sbrs.

Will display seconds to the Terminal whenever called.

Adc_sample

This sbr is called in response to the user inputting the letter "a" on the Terminal while in Node mode by the **set_flag_D** label. This performs the asynchronous operation of reading 256 voltage samples from the ADC at address $6000.

The **adc_** labels will output a "." to the Terminal at every sample, as well as creating a pulse on PORTE that lasts 2ms on HIGH.

Before anything, the sbr also stores the value read into **sample_buff** to be used for sample calculation.

They well then immediately compare the value read from the ADC and compare it to the TRIM level. Depending on the magnitude of the value read, it will either store it in memory location **HH** or **LL**.

Ms2_wait

This sbr will wait 2ms before returning. It works asynchronously with the interrupt clock pulse.

Input_getbuff

This sbr will read the values stored in the input buffer and return the last value entered before CR was entered. That value is stored in R16 to be used in the **input_check** label family.

Input_check

This sbr works with the finite state machine architecture to produce different flags depending on user input, and to trigger other asynchronous sbrs.

Depending on the input stored in the buffer, it can jump to different labels that are defined below itself. This sbr should work for both modes Node and Supervisor, but due to **SUP_INPUT_ERROR**, I dedicated this only to the Node mode.

Set_flag_D

This sbr is called in response to the user inputting the letter “a” on the Terminal while in Node mode. It outputs status messages to the Terminal to let the user know the beginning and end of operations.

It also calls sbr **set_flag_A** which is used to display statistics to the LCD.

Set_flag_T

This sbr is called in response to the user inputting the letter “t” on the Terminal while in Node mode. It simply sets the flag FLAG2 with value $E0 for **main_node** to check.

Set_flag_S & sample_cal

The **set_flag_S** label is jumped to in response to the user inputting the letter "s" on the Terminal while in Node mode. It sets the flag FLAG2 with value $B0 to direct **main** to the appropriate display loop. It also stops any interrupts to guarantee an accurate calculation without errors.

It prepares some registers for the calculations done by label **sample_cal**.

The label **sample_cal** fetches the values stored in **sample_buff** and performs a sum calculation of all values in it. It then performs a conversion of the final sum into ASCII characters by using the **hex2asc** function found in **numio.inc**.

It then stores those characters properly into four different memory locations of the **samplesum** family.

Unfortunately, due to **NODE_SUM_ERROR**, this doesn't seem to work.

Set_flag_A

This sbr is responsible for placing the statistics necessary for displaying the ADC values as well as TRIM. It sends a message to the Terminal indicating the start of the statistics being displayed to the LCD.

It then sets flag FLAG2 with value $D0 to direct **main** to the appropriate display loop.

Seg7puts

This sbr is used for debugging, and it will output to the Terminal the 7 segment code of a value stored in R16. BE CAREFUL!!, this could easily overflow.

Input_check_sup

This sbr is a copy of the **input_check** one, but used only in Supervisor mode because of the **SUP_INPUT_ERROR**. Due to this error, I only had time to implement one function for input detection, which is to Reset the timekeeping counters.

ERRORS

As it has been mentioned throughout this report, there were some errors that could not be fixed in time for the demonstration. The ones I could spot are detailed below:

ERRORS	Explanation
SUP_INPUT_ERROR	If we observe Fig 3, we can see the result of the interaction of the **getdata** and **input_getbuffer** sbrs during Supervisor mode go wrong. It seems that somehow the value inside **input_buff** don't get modified all the time when the **getdata** sbr is called during Supervisor mode. This made it impossible for me to write any functionality for this mode aside from Reset, which triggers at just about any input. My theory is that this issue comes from the fact that **sup_loop_T** gets called every second to update the clock.
NODE_SUM_ERROR	If we observe Fig 1 and Fig 2 we can see the debug output of the value found in R18 after every iteration of the Sample SUM. The accuracy of the values displayed doesn't matter, but this serves to show that the values differ from one run to the next, with the exception of those "PuY" ones. The relevance of this is that the SUM value at the end always comes up to a weirdly specific and most definitely inaccurate result. For instance, both of these calculations resulted in the decimal value 6330 being displayed. I do not know where this error is coming from, but my theory is that: 1. Something is altering the buffer data 2. The conversion is being messed up I disabled the interrupts for the SUM routine, but it didn't seem to fix the problem.

TERMINAL_DISPLAY_ERROR	This error was observed during the presentation of the project. Basically, for whatever reason, the USART sometimes did not sync to the Terminal correctly and, as such, it would not display any text. The weird thing was, the program still worked on the background and input could still be sent to the AVR, and processed by it. I have no clue as to why, and the solution to this problem was only found a few minutes before the demonstration. It basically came down to shorting VCC and Ground to force the AVR to reset with a single touch of the wire, until output could be seen on the Terminal.

Table 2: Error/bug description table

CONCLUSION

This project was a big step-up from the activities done in class. To accomplish all, if not most, of the requirements of it, immense creativity was required of us.

In the end, this project taught me a few things:

- Reliable electronics and wiring make the job of making a product much more pleasant and hassle free.
- 4.5-year-old breadboards are not meant for this type of project ☹
- There are many ways to accomplish the same task in electronics, and my approaches wildly differed from my peers. This made independent work a lot more necessary, and without the help of my peers this project took considerably longer than any previous lab.
- Modularity comes at the cost of complexity.

The biggest obstacle at completing this project is detailed at the **TERMINAL_DISPLAY_ERROR** entry of Table 2. Had it not been for that, or had I found the solution to it faster, this report would've been done earlier and the circuit would've been more complete.