University of British Columbia
Electrical and Computer Engineering
ELEC291/ELEC292

Project 1 – EFM8 board, FSM, EEPROM, and tips

Dr. Jesús Calviño-Fraga P.Eng.
Department of Electrical and Computer Engineering, UBC
Office: KAIS 3024
E-mail: jesusc@ece.ubc.ca
Phone: (604)-827-5387

February 7, 2020

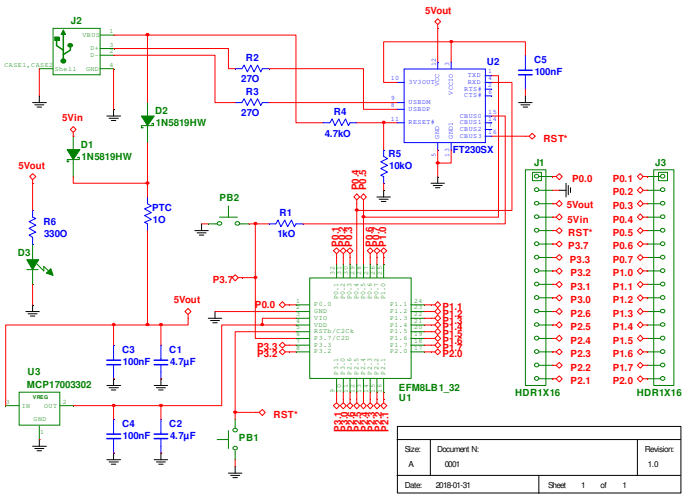
Objectives

- Introduction to the EFM8 board.
- Programming using Finite State Machines (FSMs) in assembly language.
- Using EEPROM for non-volatile variable storage and initialization.
- Extra project tips.

The EFM8 Board

- Each student should have a EFM8 board for the second half of the course.
- Each student should assemble (or try to) a EFM8 board. Stencil + Solder Paste + SMDs + TH + Testing.
- The EFM8 board needs to be soldered in an reflow oven. You need a reflow oven controller!

EFM8 circuit



EFM8 Bill of Materials (BOM)

Qty	Supplier's#	Reference	Man's #	Description
1	768-1135-1-ND	U2	FT230XS-R	IC USB SERIAL BASIC UART 16SSOP
1	MCP1700T3302ETTCT-ND	U3	MCP1700T-3302E/TT	IC REG LDO 3.3V 0.25A SOT23-3
1	336-3736-ND	U1	EFM8LB12F64E-B-QFP32	IC MCU 8BIT 64KB FLASH 32QFP
2	450-1759-1-ND	PB1, PB2	FSM4JSMATR	SWITCH TACTILE SPST-NO 0.05A 24V
2	A26509-16-ND	J1, J3	4-103741-0-16	CONN HEADR BRKWAY .100 16POS STR
1	ED2983-ND	J2	USB-B1HSB6	CONN USB TYPE B R/A BLACK
2	1N5819HW-FDICT-ND	D1, D2	1N5819HW-7-F	DIODE SCHOTTKY 40V 1A SOD123
3	399-1170-1-ND	C3, C4, C5	C0805C104K5RACTU	CAP CER 0.1UF 50V X7R 0805
2	311-22ARCT-ND	R2, R3	RC0805JR-0722RL	RES SMD 22 OHM 5% 1/8W 0805
1	160-1179-1-ND	D3	LTST-C170GKT	LED GREEN CLEAR 0805 SMD
1	311-330ARCT-ND	R6	RC0805JR-07330RL	RES SMD 330 OHM 5% 1/8W 0805
1	311-1.0KARCT-ND	R1	RC0805JR-071KL	RES SMD 1K OHM 5% 1/8W 0805
1	311-4.7KARCT-ND	R4	RC0805JR-074K7L	RES SMD 4.7K OHM 5% 1/8W 0805
2	478-8125-1-ND	C1, C2	F921A475MPA	CAP TANT 4.7UF 10V 20% 0805
1	507-1797-1-ND	PTC	0ZCJ0020FF2E	PTC RESTTBLE 0.20A 30V CHIP 1206
1	311-10KARCT-ND	R5	RC0805JR-0710KL	RES SMD 10K OHM 5% 1/8W 0805

Steps Assembling a PCB with SMDs.

- Step 1: Apply solder paste to the PCB. You will use a Mylar stencil. (The most critical step in the whole process!)
- Step 2: Place the SMT components into the PCB.
- Step 3: Reflow soldering. You will be using a toaster oven with a controller of your own design.
- Step 4: Hand soldering of TH (thru hole) components.

Testing the EFM8 Board

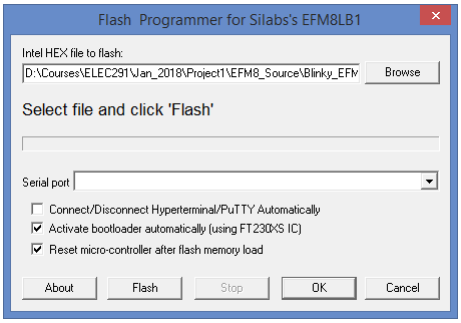
- Write a "blinky.asm" for the EFM8. Some things to take into account compared to the AT89LP51RC2 and P89LPC9351:
  - The default oscillator frequency is 6.000MHz. It can be configured for 12MHz, 24MHz, 48MHZ, and 72MHz… or many different values in between!
  - The number cycles per instruction is different.
  - The registers used to configure the ports are different. Check the datasheet!

blinky_EFM8.asm

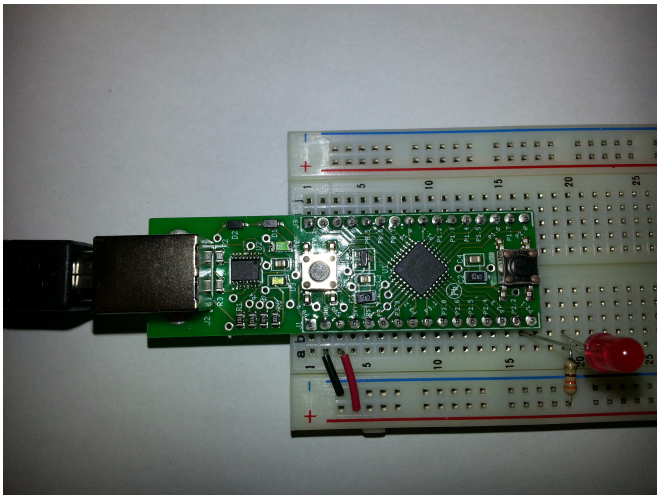
```
$MODEFM8LB1

CSEG at 0H
    ljmp main
Wait_half_second:
    ;For a 6MHz clock one machine cycle takes 1/6.0000MHz=166.666ns
    mov R2, #25
L3: mov R1, #250
L2: mov R0, #120
L1: djnz R0, L1 ; 4 machine cycles-> 4*166.666ns*120=80us
    djnz R1, L2 ; 80us*250=0.02s
    djnz R2, L3 ; 0.02s*25=0.5s
    ret
main:
    ; DISABLE WDT: provide Watchdog disable keys
    mov WDTCN, #0xDE ; First key
    mov WDTCN, #0xAD ; Second key
    mov SP, #7FH
    ; Enable crossbar and weak pull-ups
    mov XBR0, #0x00
    mov XBR1, #0x00
    mov XBR2, #0x40
    mov P2MDOUT, #0x02 ; make LED pin (P2.1) output push-pull
M0: cpl P2.1 ; Led off/on
    lcall Wait_half_second
    sjmp M0
end
```

Flashing HEX file into EFM8 Board

- In CrossIDE click fLash->Silabs EFM8LB1. Select the correct HEX file, make sure settings are like shown below, and then click 'Flash'.



Testing the board with blinky_EFM8.asm in breadboard.

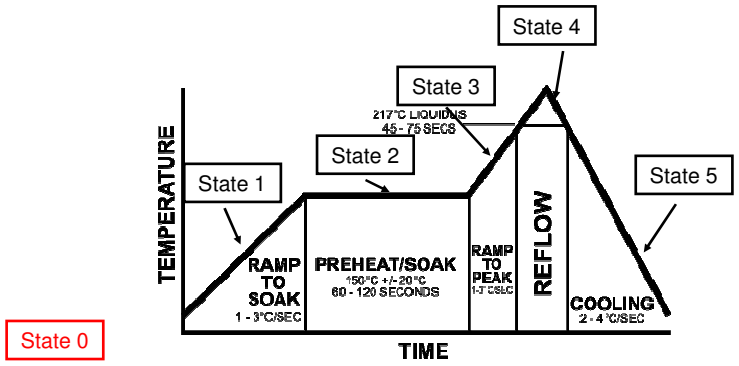


Finite State Machines in Assembly Language

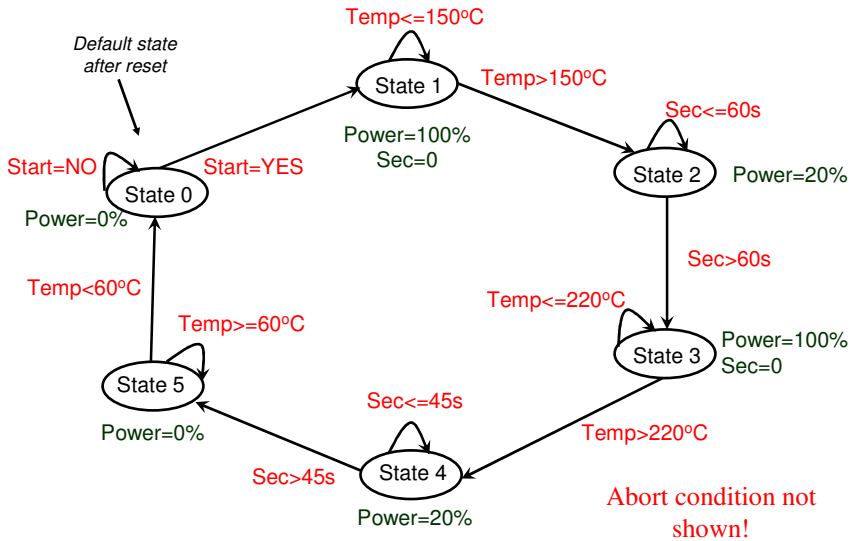
- A finite state machine (FSM) is a programming abstraction method that can be represented using a graph structure.
- We can draw the states as circles and the transitions as arrows.
- There is a finite number of states. The active state is called the current state.
- FSMs are easily implemented in assembly language!
- Many FMS can be run "concurrently". (One after another really!)
- FSM are in principle non-blocking.

Reflow Profile States

http://en.wikipedia.org/wiki/Reflow_soldering



Reflow Profile FSM



In assembly (some states only!)

```
    mov a, state

state0:
    cjne a, #0, state1
    mov pwm, #0
    jb PB6, state0_done
    jnb PB6, $ ; Wait for key release
    mov state, #1
state0_done:
    ljmp forever

state1:
    cjne a, #1, state2
    mov pwm, #100
    mov sec, #0
    mov a, #150
    clr c
    subb a, temp
    jnc state1_done
    mov state, #2
state1_done:
    ljmp forever
```

```
state2:
    cjne a, #2, state3
    mov pwm, #20
    mov a, #60
    clr c
    subb a, sec
    jnc state2_done
    mov state, #3
state2_done:
    ljmp forever
        .
        .
        .
        .
        .
        .
        .
```

In assembly (some states only!) using variables…

```
    mov a, state

state0:
    cjne a, #0, state1
    mov pwm, #0
    jb PB6, state0_done
    jnb PB6, $ ; Wait for key release
    mov state, #1
state0_done:
    ljmp forever

state1:
    cjne a, #1, state2
    mov pwm, #100
    mov sec, #0
    mov a, temp_soak
    clr c
    subb a, temp
    jnc state1_done
    mov state, #2
state1_done:
    ljmp forever
```

```
state2:
    cjne a, #2, state3
    mov pwm, #20
    mov a, time_soak
    clr c
    subb a, sec
    jnc state2_done
    mov state, #3
state2_done:
    ljmp forever
        .
        .
        .
        .
        .
DSEG ; Before the state machine!
state: ds 1
temp_soak: ds 1
Time_soak: ds 1
Temp_refl: ds 1
Time_refl: ds 1
```

Variables need to be initialized!

About Variables

- Initialize variables before using them!
- It is easy to work with binary (8-bit) variables. Use “inc”, “dec”, to increment/decrement and ‘subb’ to compare.
- Small variables are easy to save and retrieve from non-volatile memory such as EEPROM.
- If temperature measurements are too “noisy”, make several measurements and take the average!
- To convert 8-bit binary variable to decimal use either HEX2BCD (in the math32 library) or one of these 8051 subroutines:

Binary to decimal conversion of 8-bit numbers in the 8051

```
; Send eight bit number via serial port, passed in 'a'.
SendToSerialPort:
      mov b, #100
      div ab
      orl a, #0x30 ; Convert hundreds to ASCII
      lcall putchar ; Send to PuTTY/Python/Matlab
      mov a, b    ; Remainder is in register b
      mov b, #10
      div ab
      orl a, #0x30 ; Convert tens to ASCII
      lcall putchar ; Send to PuTTY/Python/Matlab
      mov a, b
      orl a, #0x30 ; Convert units to ASCII
      lcall putchar ; Send to PuTTY/Python/Matlab
      ret
```


Binary to decimal conversion of 8-bit numbers in the 8051

```
; Eight bit number to display passed in 'a'.
; Sends result to LCD
SendToLCD:
      mov b, #100
      div ab
      orl a, #0x30 ; Convert hundreds to ASCII
      lcall ?WriteData ; Send to LCD
      mov a, b    ; Remainder is in register b
      mov b, #10
      div ab
      orl a, #0x30 ; Convert tens to ASCII
      lcall ?WriteData; Send to LCD
      mov a, b
      orl a, #0x30 ; Convert units to ASCII
      lcall ?WriteData; Send to LCD
      ret
```

DIV AB

DIV AB

Function: Divide

Description: DIV AB divides the unsigned eight-bit integer in the accumulator by the unsigned eight-bit integer in register B. The accumulator receives the integer part of the quotient; register B receives the integer remainder. The carry and OV flags will be cleared. *Exception*: If B had originally contained 00H, the values returned in the accumulator and B register will be undefined and the overflow flag will be set. The carry flag is cleared in any case.

Example: The accumulator contains 251 (0FBH or 11111011B) and B contains 18 (12H or 00010010B). The instruction DIV AB will leave 13 in the accumulator (0DH or 00001101 B) and the value 17 (11H or 00010001B) in B, since 251 = (13x18) + 17. Carry and OV will both be cleared.

Operation: **DIV AB**
(A), (B) ¬ (A) / (B)

Encoding:

1	0	0	0	0	1	0	0

Non-Volatile Memory: P89LPC9351 EEPROM

18. Data EEPROM (P89LPC9351/9361)

The P89LPC9331/9341/9351/9361 has 512 bytes of on-chip Data EEPROM that can be used to save configuration parameters. The Data EEPROM is SFR based, byte readable, byte writable, and erasable (via row fill and sector fill). The user can read, write, and fill the memory via three SFRs and one interrupt:

- Address Register (DEEADR) is used for address bits 7 to 0 (bit 8 is in the DEECON register).
- Control Register (DEECON) is used for address bit 8, setup operation mode, and status flag bit (see Table 127).
- Data Register (DEEDAT) is used for writing data to, or reading data from, the Data EEPROM.

Table 127. Data EEPROM control register (DEECON address F1h) bit allocation

Bit	7	6	5	4	3	2	1	0
Symbol	EEIF	HVERR	ECTL1	ECTL0	-	EWERR1	EWERR0	EADR8
Reset	1	0	0	0	0	0	0	0

Table 128. Data EEPROM control register (DEECON address F1h) bit description

Why non-volatile memory?

- To save your reflow oven controller parameters so they are available automatically the next time you use it.
- To store other useful information, such as the last reflow profile.

EEPROM Functions Example

```
; Address to write to passed in DPTR.  Data to write passed in register 'A'
EEPROM_Write:
    clr EA ; 'Strongly recommended' in the datasheet page 133
    mov DEECON, DPH ; ECTL1/ECTL0 (DEECON[5:4]) = '00', EADR8
    mov DEEDAT, a ; Byte to write
    mov DEEADR, DPL ; Address to write to.  This initializes the write process
    ; Wait for write operation to complete
EEPROM_Write_L1:
    mov a, DEECON
    jnb acc.7, EEPROM_Write_L1 ; bit 7 of DEECON is EEIF
    setb EA
    ret

; Address to read from passed in DPTR.  Data read returned via register 'A'
EEPROM_Read:
    clr EA ; 'Strongly Recommended' in the datasheet page 133
    mov DEECON, DPH ; ECTL1/ECTL0 (DEECON[5:4]) = '00', EADR8=0
    mov DEEADR, DPL ; Address to read from.  This initializes the read process
    ; wait for read operation to complete
EEPROM_Read_L1:
    mov a, DEECON
    jnb acc.7, EEPROM_Read_L1 ; bit 7 of DEECON is EEIF
    mov a, DEEDAT
    setb EA
    ret
```

Example code in Canvas: LPC9351_EEPROM_Example.asm

Example: Writing Project Data to EEPROM

```
Save_Configuration:
    mov DPTR, #0
    ; Save variables
    mov a, temp_soak
    lcall EEPROM_Write
    inc DPTR
    mov a, time_soak
    lcall EEPROM_Write
    inc DPTR
    mov a, temp_refl
    lcall EEPROM_Write
    inc DPTR
    mov a, time_refl
    lcall EEPROM_Write
    inc DPTR
    mov a, #0x55 ; First key value
    lcall EEPROM_Write
    inc DPTR
    mov a, #0xAA ; Second key value
    lcall EEPROM_Write
    ret
```


Example: Read Project Data From EEPROM; check keys first!

```
Load_Configuration:
   mov dptr, #0x0004 ;First key value location.  Must be 0x55
   lcall EEPROM_Read
   cjne a, #0x55, Load_Defaults
   inc dptr ; Second key value location.  Must be 0xaa
   lcall EEPROM_Read
   cjne a, #0xaa, Load_Defaults
```

Continues…

Example: Read From Flash; Load Saved Values

```
; Keys are good.  Load stored values.
mov dptr, #0x0000
lcall EEPROM_Read
mov temp_soak, a
inc dptr
lcall EEPROM_Read
mov time_soak, a
inc dptr
lcall EEPROM_Read
mov temp_refl, a
inc dptr
lcall EEPROM_Read
mov time_refl, a
ret
```

Continues…

Example: Read From Flash; Load Default Values

```
Load_Defaults: ; Load defaults if 'keys' are incorrect
   mov temp_soak, #150
   mov time_soak, #45
   mov temp_refl, #225
   mov time_refl, #30
   ret
```

Extra Tips…

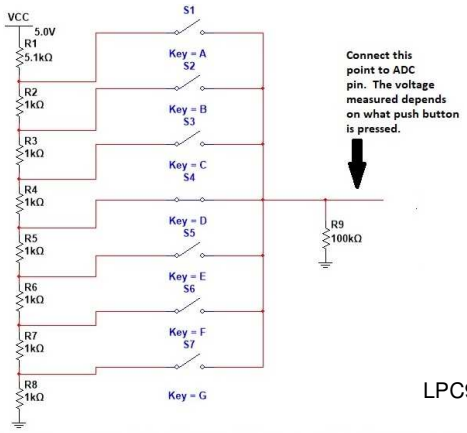
- Are you using macros yet?

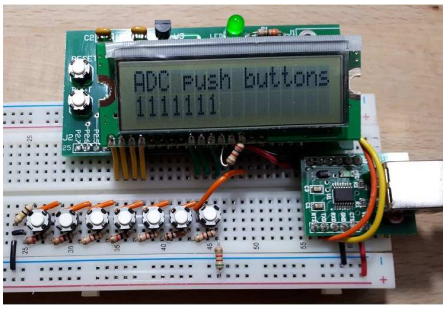
```
Change_8bit_Variable MAC
    jb %0, %2
    Wait_Milli_Seconds(#50)
    jb %0, %2
    jnb %0, $
    jb SHIFT_BUTTON, skip%Mb
    dec %1
    sjmp skip%Ma
skip%Mb:
    inc %1
skip%Ma:
ENDMAC
```

For regular pin push-buttons, for example P0.1

```
    Change_8bit_Variable(MY_VARIABLE_BUTTON, my_variable, loop_c)
    Set_Cursor(2, 14)
    mov a, my_variable
    lcall LCD_Accumulator
    lcall Save_Configuration
loop_c:
```


Push-buttons connected to voltage divider.





Example file:
LPC9351_PushButtons_Using_ADC0.asm

Extra Tips…

```
Change_8bit_Variable MAC
    lcall ADC_to_PB
    jb %0, %2
    Wait_Milli_Seconds(#50)
    lcall ADC_to_PB
    jb %0, %2
Loop%M:
    lcall ADC_to_PB
    jnb %0, Loop%M
    jb SHIFT_BUTTON, skip%Mb
    dec %1
    sjmp skip%Ma
skip%Mb:
    inc %1
skip%Ma:
ENDMAC
```

For voltage divider push-buttons connected to ADC channel as in previous slide

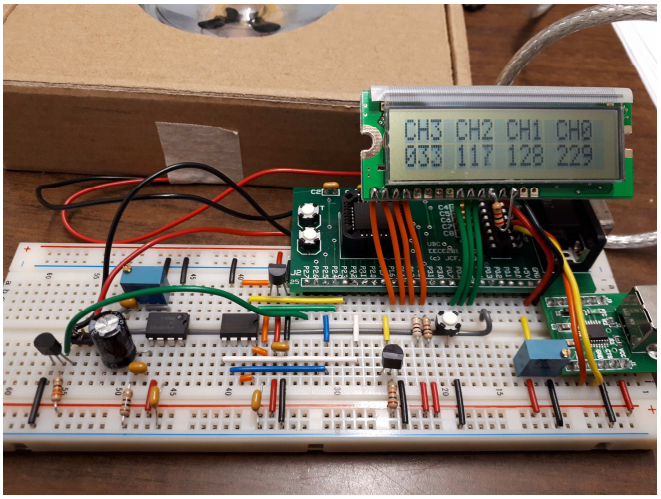
```
    Change_8bit_Variable(MY_VARIABLE_BUTTON, my_variable, loop_c)
    Set_Cursor(2, 14)
    mov a, my_variable
    lcall SendToLCD
    lcall Save_Configuration
loop_c:
```


Extra Tips: Pin management

- (RESERVED) SPI pins to access FLASH memory: P2.2, P2.3, P2.4, P2.5.
- (RESERVED) P0.4 is the DAC1 output which is used as the 'sound' output connected to the LM386 speaker amplifier.
- (RESERVED) P1.7, P0.0, P2.1, and P2.0 are the analog inputs for ADC0. Assuming we use all four channels! If a particular channel is not used, the corresponding pin is available as a general purpose I/O pin. Use register 'ADINS' to select the ADC channels to use.
- We need 7 pins for the LCD. Let arbitrarily use P0.5, P0.6, P0.7, P1.2, P1.3, P1.4, P1.6. WARNING: P1.2 and P1.3 need each a 1k ohm pull-up resistor to VCC.
- Other pins available: P3.0, P2.6, P3.1, P2.7, P0.1, P0.2, P0.3.
- Example in Canvas: 'LPC9351_Receiver_LCD_ADC0.asm'. This example uses P3.0 for a 'Play' push-button and P2.7 to 'disconnect' the speaker when not in use (otherwise we get 'clicking' noise).

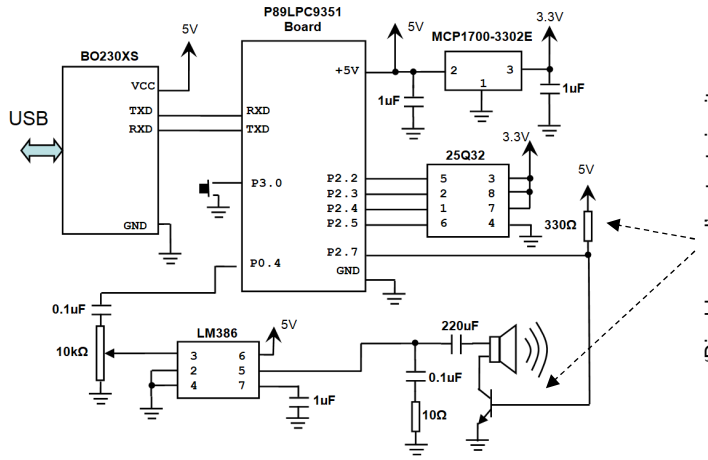
Extra Tips: Pin management

For testing: CH0 (P1.7) is connected to the LM335, CH1 (P0.0) is connected to a potentiometer, and both CH2 (P2.1) and CH3 (P2.0) are unconnected.



Extra Tips: Disable Sound Output

LCD not shown because of lack of space!



'Disable sound' output circuit.

Extra tips…

- 'Noisy' measurements? Average! (The P89LPC9351 is not noisy at all!!!)

```
Wait10us:
    mov R0, #18
    djnz R0, $
    ret

Average_AD0DAT0 :
    Load_x(0)
    mov R5, #100
Sum_loop0:
    mov y+3, #0
    mov y+2, #0
    mov y+1, #0
    mov y+0, AD0DAT0
    lcall add32
    lcall Wait10us
    djnz R5, Sum_loop0
    load_y(100)
    lcall div32
    ret
```


Extra tips…

- Op-amp has to much offset? Zero it! (Should not be needed for OP07)

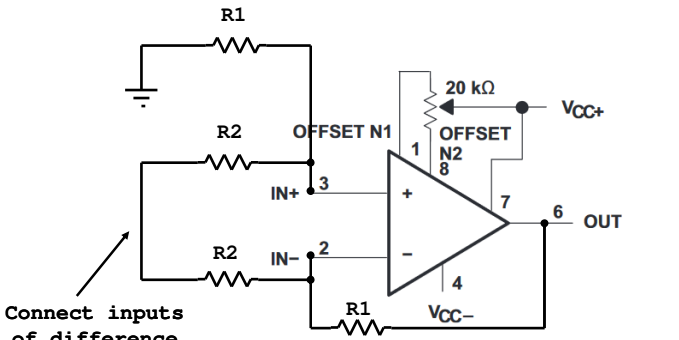


Figure 2. Input Offset-Voltage Null Circuit

Adjust pot until 'OUT' is near zero mV.

Extra tip

- There is ‘magic’ value of gain that will give you the temperature of the thermocouple (minus cold junction) directly when reading from the ADC!