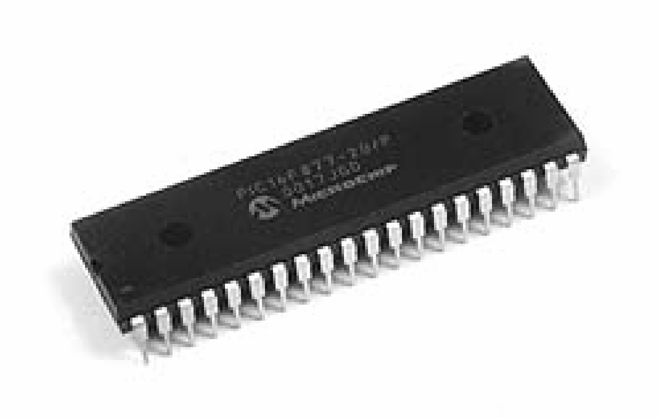
Name: Honnet Cedric
Student Number: 0531984
Group Code: ME

Workshop Session n°2 - PIC16F877:

Analog to Digital Conversion, Pulse Width Modulation



1 Purpose

The purpose of this lab was to make us understand the concepts of Analog to Digital Conversion and Pulse Width Modulation using the PIC microcontroller and its development board. Our programming skills with assembly language, the use of timer and interrupt service routines were to be greatly improved. On another hand, Real time interfacing and other concrete applications with the PIC16F877 microcontroller were to be achievable.

To meet this challenge, we are given templates to fill with our understanding of the corresponding comments. But I wanted to do a little more to finalize this project, I realized a voltmeter coupled to our pulse width modulator, its particularity is to display the average of the voltage given.

To explain what I understood, I will simply try to comment it as if you, dear lector, didn't know anything about this project. But as my English writing is still not efficient enough, I will also use pictures and video to complete my explanations (a CD with videos and .asm files is joined).

2 Analysis for the Tasks

ADC, main functionalities:

-Sample analog input values

-Sample and hold capacitor

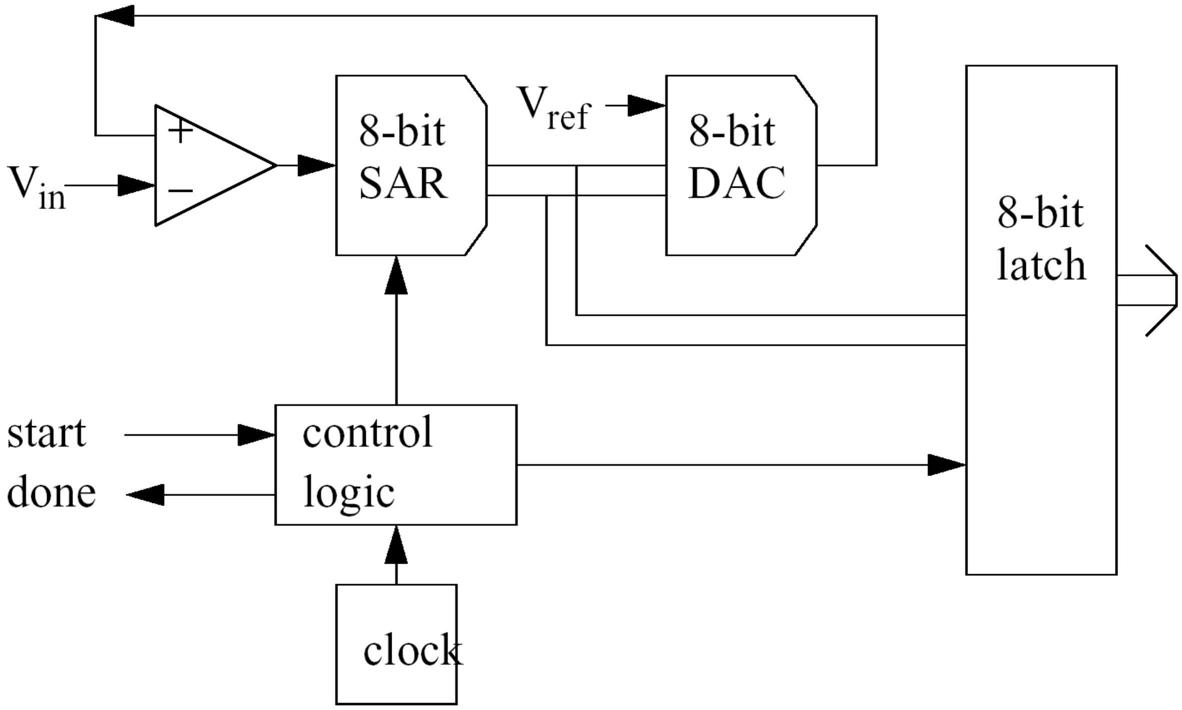
-Compare with current approximation (from DAC) – DAC starts with maximum possible analog voltage output.

-SAR – successive approximation register. Holds current bit high, if comparator output is high (Vin <= current approximation), then it resets the bit and moves to the next less significant bit by setting it high. If low (Vin > current approximation) then the bit is left high and the SAR moves to the next less significant bit by setting it high.

-DAC just converts the value in the latch(current approximation) back in to an analog signal to compare with the input in the comparator.

-The latch stores the value when the LSB is complete.

-The control logic counts the number of bits and then when all counted tells the latch to hold and store the value.



Almost all the fundamental components of the PIC that were vital to understand and to write the program code were found in the PIC16F877 data sheet:

1. The ADCON0 special function register: p111
    a. Contains the settings for the AD conversion clock select, which sets the Fosc ratio.
    b. The analog channel select bits: 001 selects channel 1 which is the potentiometer on AN1/RA1, or 000 selects channel 0 which is the Light dependent resistor…
    c. There is a bit for the GO/ , which is reset when the conversion is complete, and can be set when the user requires the ADC to start.
    d. And finally, ADON is about the operating state of the A/D converter module.

2. The OPTION_REG register: p22
    a. This contains settings about the timer and Watchdog pre-scalars, and pre-scalar assignment(to either WDT or TMR0).
    b. Post-scalar settings and
    c. TMR0 source setting(High – transition on RA4 pin, Low – Internal instruction clock signal).

3. The ADCON1 register: p112
    a. This contains settings for the justification of the result in to ADRESH:ADRESL. High this sets the 10 bit converted value in to the right 10 bits of the concatenated registers, Low this sets the 10 bits in to the left 10 bits. We use this set low and discard the 2 bits in the ADRESL register.
    b. The only other 3 bits select the configuration of the I/O ports for ADC (PORTS A/E). In this I used the setting 000 to set all A/D ports as Analog, therefore disabling any digital input or output (Digital output to PORT B are not concerned).

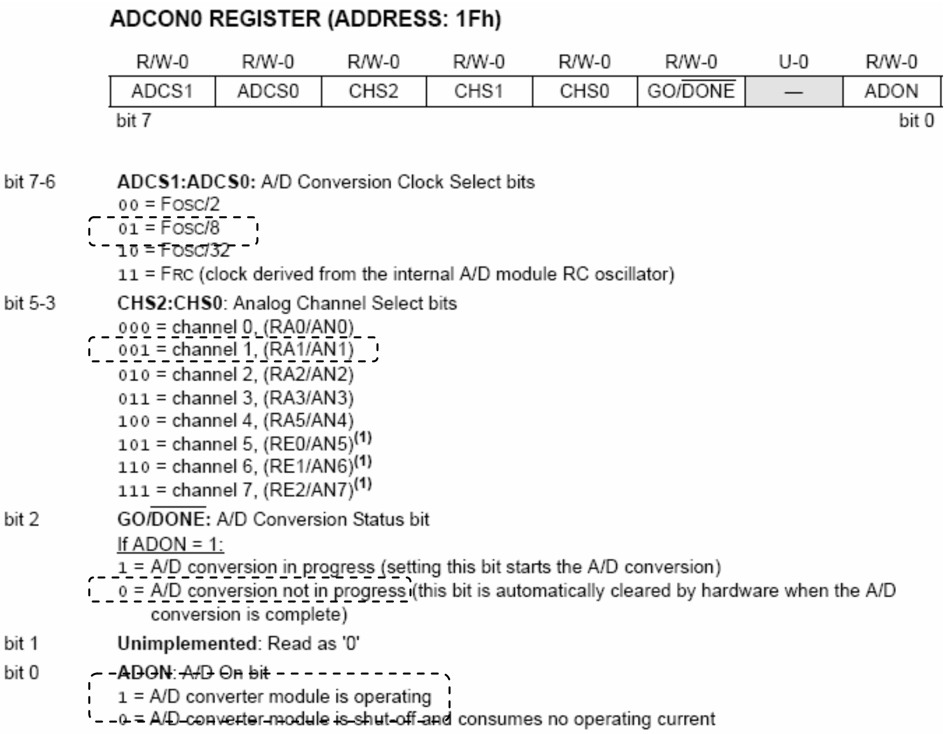
4. The INTCON register: p20
    a. This contains the settings for enabling unmasked global and peripheral interrupts. GIE, PEIE.
    b. Also the setting to enable the TMR0 Overflow interrupt is TOIE (high = enabled).
    c. The overflow interrupt flag bit TOIF, (high = overflowed). This can be cleared or polled in software.

5. The PIR1 register: p22
    a. This mainly contains the Flag for A/D Converter Interrupt* (Conversion completion)
    b. And the other bits are not going to be used in this project.

These are the main SPR’s that understanding of is required in the program. Also previous understanding of basic I/O controls using TRISA/B and PORTA/B is assumed.
With this basic knowledge we can proceed to the code.

*Knowledge of the ADIE bit (A/D converter interrupt enable) in PIE1 SPR to enable the A/D interrupt, and ADIF (A/D converter interrupt flag) in PIR1 SPR to test for the A/D conversion completion is required.

2.1 Task 1



ADCON0 REGISTER (ADDRESS: 1Fh)

R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	U-0	R/W-0
ADCS1	ADCS0	CHS2	CHS1	CHS0	GO/$\overline{\text{DONE}}$	—	ADON
bit 7							bit 0

bit 7-6 **ADCS1:ADCS0:** A/D Conversion Clock Select bits
00 = Fosc/2
01 = Fosc/8
10 = Fosc/32
11 = FRC (clock derived from the internal A/D module RC oscillator)

bit 5-3 **CHS2:CHS0**: Analog Channel Select bits
000 = channel 0, (RA0/AN0)
001 = channel 1, (RA1/AN1)
010 = channel 2, (RA2/AN2)
011 = channel 3, (RA3/AN3)
100 = channel 4, (RA5/AN4)
101 = channel 5, (RE0/AN5)(1)
110 = channel 6, (RE1/AN6)(1)
111 = channel 7, (RE2/AN7)(1)

bit 2 **GO/$\overline{\text{DONE}}$:** A/D Conversion Status bit
If ADON = 1:
1 = A/D conversion in progress (setting this bit starts the A/D conversion)
0 = A/D conversion not in progress (this bit is automatically cleared by hardware when the A/D conversion is complete)

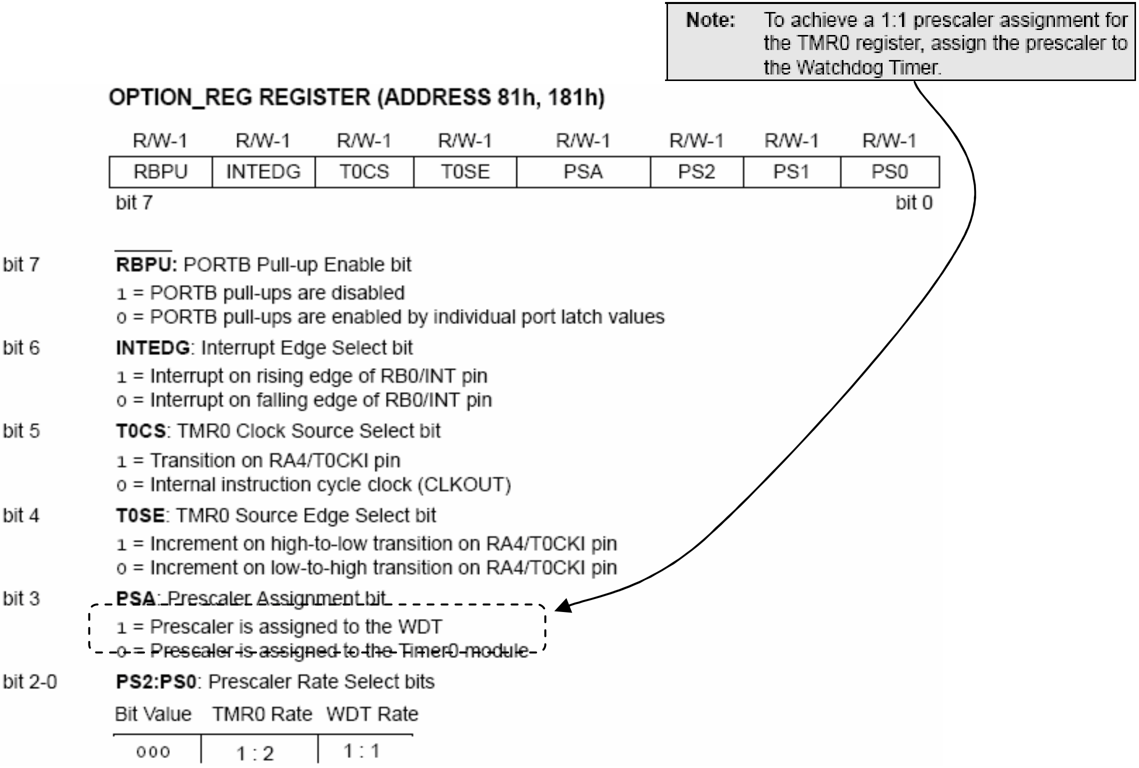
bit 1 **Unimplemented**: Read as '0'

bit 0 **ADON**: A/D On bit
1 = A/D converter module is operating
0 = A/D converter module is shut-off and consumes no operating current

For the first blank to fill in is then :

```
movlw       B'01001001'          ; Setup A/D to read the Potential Meter on RA1
movwf       ADCON0               ; with the parameters include Fosc/8, A/D operating, Sample Channel 1
```

For the next instruction we need option_reg:



Note: To achieve a 1:1 prescaler assignment for the TMR0 register, assign the prescaler to the Watchdog Timer.

OPTION_REG REGISTER (ADDRESS 81h, 181h)

R/W-1	R/W-1	R/W-1	R/W-1	R/W-1	R/W-1	R/W-1	R/W-1
$\overline{\text{RBPU}}$	INTEDG	T0CS	T0SE	PSA	PS2	PS1	PS0
bit 7							bit 0

bit 7 **$\overline{\text{RBPU}}$:** PORTB Pull-up Enable bit
1 = PORTB pull-ups are disabled
0 = PORTB pull-ups are enabled by individual port latch values

bit 6 **INTEDG**: Interrupt Edge Select bit
1 = Interrupt on rising edge of RB0/INT pin
0 = Interrupt on falling edge of RB0/INT pin

bit 5 **T0CS**: TMR0 Clock Source Select bit
1 = Transition on RA4/T0CKI pin
0 = Internal instruction cycle clock (CLKOUT)

bit 4 **T0SE**: TMR0 Source Edge Select bit
1 = Increment on high-to-low transition on RA4/T0CKI pin
0 = Increment on low-to-high transition on RA4/T0CKI pin

bit 3 **PSA**: Prescaler Assignment bit
1 = Prescaler is assigned to the WDT
0 = Prescaler is assigned to the Timer0 module

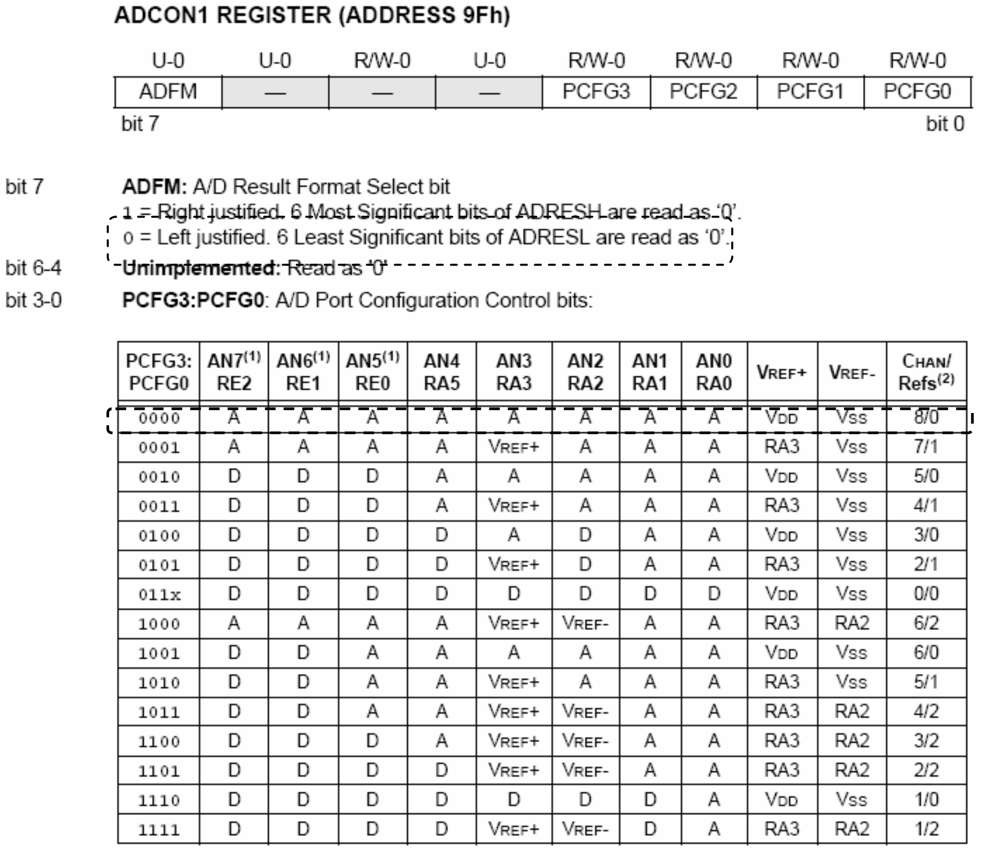
bit 2-0 **PS2:PS0**: Prescaler Rate Select bits

Bit Value	TMR0 Rate	WDT Rate
000	1 : 2	1 : 1

The corresponding code is then :

```
movlw       B'00001000'  ; To set TMR0 with prescale value of 1:1 we have to assign the prescaler to
movwf       OPTION_REG   ; the watch dog timer (see note p.19)

movlw       B'00000011'  ; Set RA0, RA1 as Analog (1)nput, and the rest of PORTA as (0)utput (obvious)
```

Next register used:



ADCON1 REGISTER (ADDRESS 9Fh)

U-0	U-0	R/W-0	U-0	R/W-0	R/W-0	R/W-0	R/W-0
ADFM	—	—	—	PCFG3	PCFG2	PCFG1	PCFG0
bit 7							bit 0

bit 7 **ADFM:** A/D Result Format Select bit
1 = Right justified. 6 Most Significant bits of ADRESH are read as '0'.
0 = Left justified. 6 Least Significant bits of ADRESL are read as '0'.

bit 6-4 **Unimplemented:** Read as '0'

bit 3-0 **PCFG3:PCFG0:** A/D Port Configuration Control bits:

PCFG3: PCFG0	AN7[1] RE2	AN6[1] RE1	AN5[1] RE0	AN4 RA5	AN3 RA3	AN2 RA2	AN1 RA1	AN0 RA0	VREF+	VREF-	CHAN/ Refs[2]
0000	A	A	A	A	A	A	A	A	VDD	VSS	8/0
0001	A	A	A	A	VREF+	A	A	A	RA3	VSS	7/1
0010	D	D	D	A	A	A	A	A	VDD	VSS	5/0
0011	D	D	D	A	VREF+	A	A	A	RA3	VSS	4/1
0100	D	D	D	D	A	D	A	A	VDD	VSS	3/0
0101	D	D	D	D	VREF+	D	A	A	RA3	VSS	2/1
011x	D	D	D	D	D	D	D	D	VDD	VSS	0/0
1000	A	A	A	A	VREF+	VREF-	A	A	RA3	RA2	6/2
1001	D	D	A	A	A	A	A	A	VDD	VSS	6/0
1010	D	D	A	A	VREF+	A	A	A	RA3	VSS	5/1
1011	D	D	A	A	VREF+	VREF-	A	A	RA3	RA2	4/2
1100	D	D	D	A	VREF+	VREF-	A	A	RA3	RA2	3/2
1101	D	D	D	D	VREF+	VREF-	A	A	RA3	RA2	2/2
1110	D	D	D	D	D	D	D	A	VDD	VSS	1/0
1111	D	D	D	D	VREF+	VREF-	D	A	RA3	RA2	1/2

The corresponding code is then :

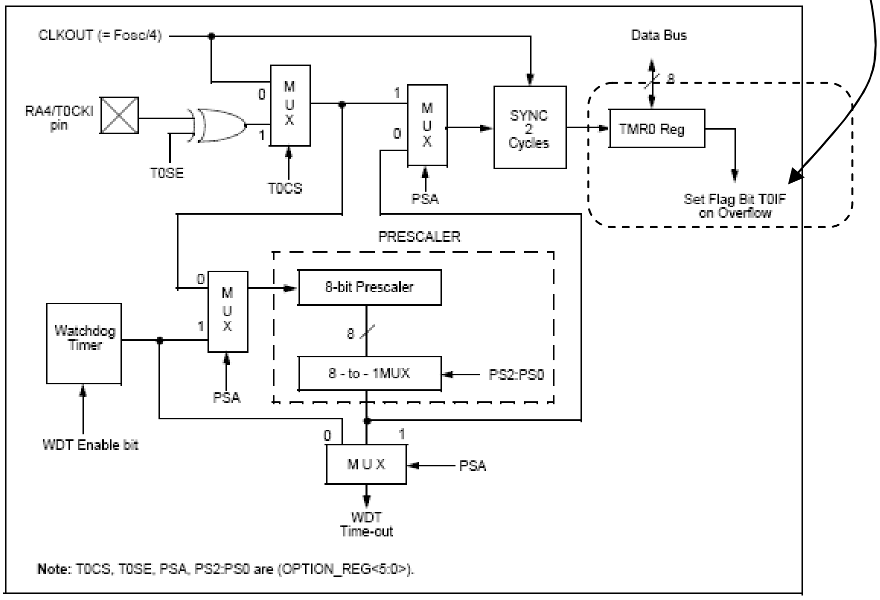
```
movlw       B'00000000'             ; Set A/D result to be left justified and enables all A/D channel
movwf       ADCON1                  ; with Vref+ = VDD and Vref- = VSS references
```

To setup TMR0 we need to know an important detail: (p.130)

12.10.2 TMR0 INTERRUPT

An overflow (FFh → 00h) in the TMR0 register will set flag bit T0IF (INTCON<2>).

FIGURE 5-1: BLOCK DIAGRAM OF THE TIMER0/WDT PRESCALER

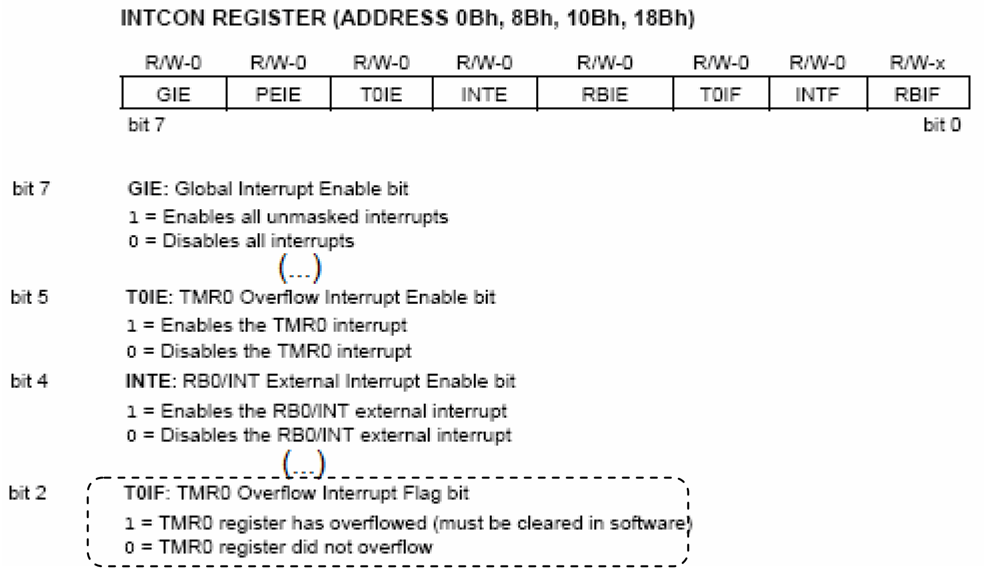


Note: T0CS, T0SE, PSA, PS2:PS0 are (OPTION_REG<5:0>).

The corresponding code is then :

```
Main        movlw       H'EC'       ; 256 - 20 = 236 = 0xEC => 20 Tosc timer.
            movwf       TMR0        ; Setup TMR0  to implement settling time of 20us for the A/D
            bcf         INTCON,T0IF ; Clear TMR0 overflow Interrupt (T0IF) SEE NEXT PAGE
```

…and we can continue:



INTCON REGISTER (ADDRESS 0Bh, 8Bh, 10Bh, 18Bh)

R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-x
GIE	PEIE	T0IE	INTE	RBIE	T0IF	INTF	RBIF
bit 7							bit 0

bit 7 **GIE**: Global Interrupt Enable bit
1 = Enables all unmasked interrupts
0 = Disables all interrupts

(…)

bit 5 **T0IE**: TMR0 Overflow Interrupt Enable bit
1 = Enables the TMR0 interrupt
0 = Disables the TMR0 interrupt

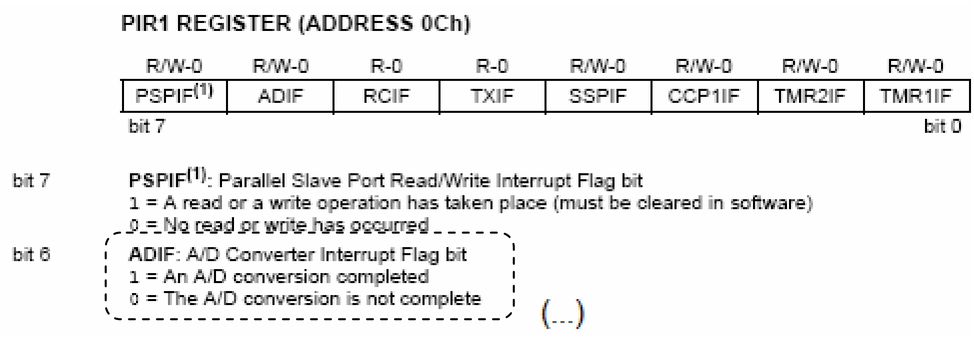
bit 4 **INTE**: RB0/INT External Interrupt Enable bit
1 = Enables the RB0/INT external interrupt
0 = Disables the RB0/INT external interrupt

(…)

bit 2 **T0IF**: TMR0 Overflow Interrupt Flag bit
1 = TMR0 register has overflowed (must be cleared in software)
0 = TMR0 register did not overflow

The corresponding code is then :

```
Loop          btfss INTCON,T0IF          ; Timer0 counter expire? skip next instruction if yes (expired=0)
              goto  Loop                 ;
              bcf   INTCON,T0IF          ; Clear TMR0 overflow Interrupt (T0IF)
              bsf   ADCON1,GO_DONE       ; Start A/D conversion
```



PIR1 REGISTER (ADDRESS 0Ch)

R/W-0	R/W-0	R-0	R-0	R/W-0	R/W-0	R/W-0	R/W-0
PSPIF[1]	ADIF	RCIF	TXIF	SSPIF	CCP1IF	TMR2IF	TMR1IF
bit 7							bit 0

bit 7 **PSPIF**[1]: Parallel Slave Port Read/Write Interrupt Flag bit
1 = A read or a write operation has taken place (must be cleared in software)
0 = No read or write has occurred

bit 6 ADIF: A/D Converter Interrupt Flag bit
1 = An A/D conversion completed
0 = The A/D conversion is not complete

(…)

The corresponding code is then :

```
Wait          btfss       PIR1,ADIF    ; Wait conversion complete, skip next instruction if
              goto        Wait         ; it's completed (=TMRO overflow)
              movf        ADRESH,W     ; Get the 8 MSB of 10-bit value, and write the
              movwf       PORTB        ; A/D result (MSB) to PORTB LEDs.
              bcf         PIR1,ADIF    ; Clear A/D completion flag
              goto        Loop         ; Do it again
```

The complete code is also used in the task2 (but the electronic version is available in the cd).

Task1 conclusion :

Using the potentiometer, the PORTB LED's increases from 0 to 255. The resolution of the A/D conversion is 10bits, but only the most significant 8 bits are displayed on PORTB.

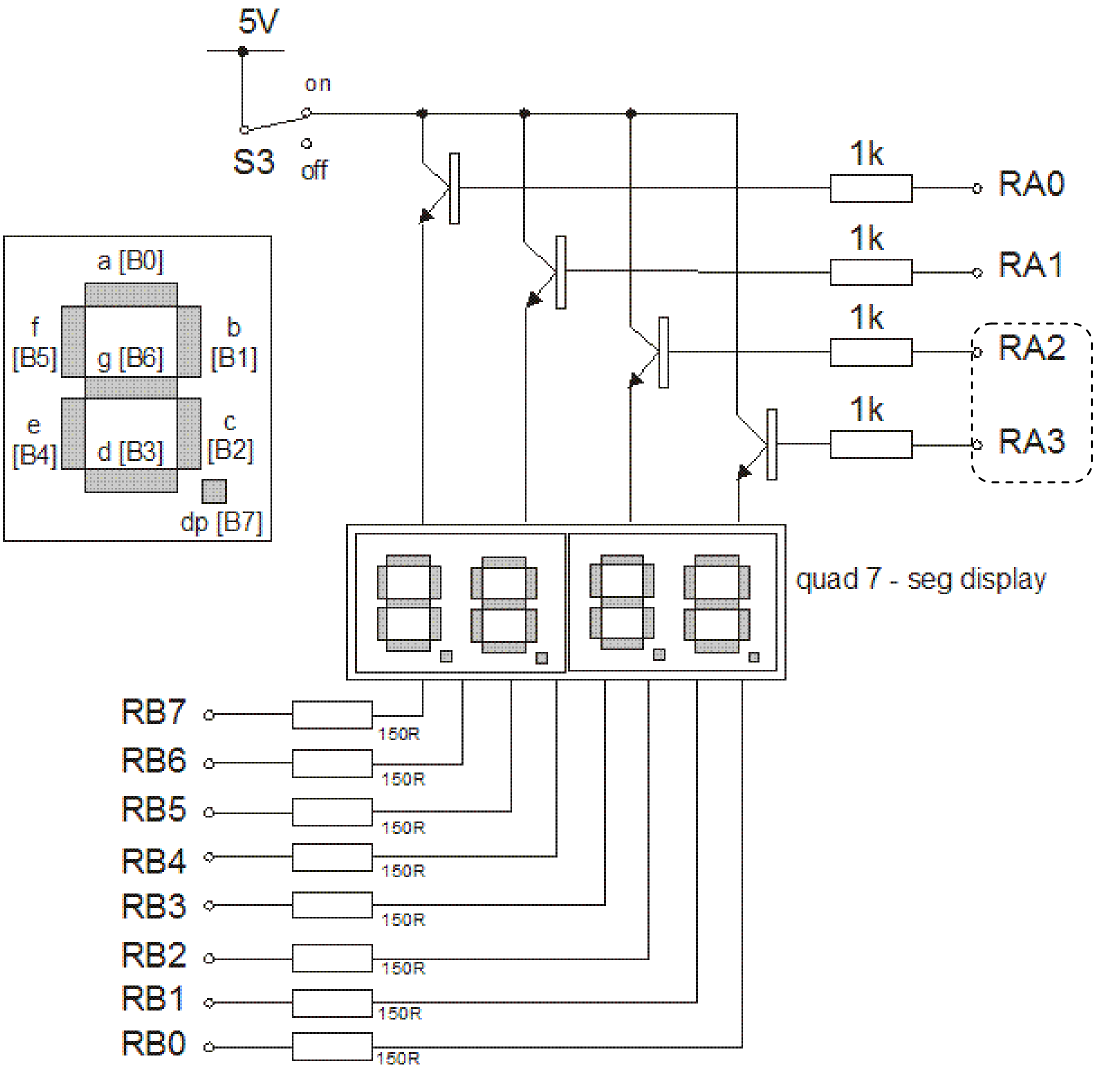
The total voltage displayable is 5v, and the maximum value of the digital equivalent displayed is FF therefore 2.5v displayed 7F. The resolution of quantization levels is $5v/2^8 \approx 20mV$, this is the minimum incrementation possible.

2.2 Task 2

The idea:

The value is displayed on the 7 segment LED, the top and bottom nibbles of ADRESH are displayed on 2 separate displays (determined by RA2 and RA3 respectively), and are switched between fast (frequency of TMR0) to give illusion that they are both on. The reason both displays are not displayed using separate outputs is to minimize I/O pin use.

The TMR0 rate comes in useful here as this delays the time that it takes for the A/D to start again.



The code:

After the operation of the task 1, the value in ADRESH is moved to Temp via the Working register, and the complement is made. This is ANDED with 0F to keep only the bottom nibble.

This value is then added to the PCL in the call to subroutine Seven_seg, and the seven segment code relating to this value is returned to the working register, then output on to PORTB and to the display by setting up PORTA to output value in PORTB to seven segments.

Note:

The delay loop does not have a return command after it, therefore runs through to the seven_seg service routine and then returns in to the loop label with a value in working register and rewrites over the working register with ADRESH.

This task required me to take the previously created code and combine it with the template for task 2:

```
Temp  EQU 0x20
count EQU 0x21
      ORG     0x00
      goto    start

start BANKSEL PORTA       ; User "BANKSEL" on any PORT will goto the right memory page
      clrf    PORTA       ; Clear PORTA
      clrf    PORTB       ; Clear PORTB
      movlw   B'01000001' ; Setup A/D to read the Potential Meter on RA0
      movwf   ADCON0      ; with the parameters include Fosc/8, A/D enabled, Sample Channel 0,

      BANKSEL OPTION_REG  ; Select right memory page
      movlw   B'00001000'
      movwf   OPTION_REG  ; Set TMR0 with prescale value of 1:1
      movlw   B'00000011' ; Set RA0, RA1 as Analog Input, and the rest of PORTA as output
      movwf   TRISA
      clrf    TRISB       ; Set PORTB as output
      movlw   B'00001000' ; To set TMR0 with prescale value of 1:1 we have to assign the prescaler to
      movwf   OPTION_REG  ; the watch dog timer (see note p.19)

      BANKSEL PORTB
Main  movlw   B'11101100' ; 256 - 20 = 236 // counter for TMR0 - Sampling rate
      movwf   TMR0        ; Setup TMR0  to implement settling time of 20us for the A/D
      bcf     INTCON,2    ; Clear TMR0 Interrupt

Loop  btfss   INTCON,2    ; Wait for Timer0 counter to expire, skip next instruction if it’s expired;
      goto    Loop
      bcf     INTCON,2    ; Clear TMR0 overflow Interrupt
      bsf     ADCON0,2    ; Start A/D conversion

Wait  btfss   PIR1,ADIF   ; Wait for conversion to complete, skit next instruction if it’s completed
      goto    Wait
      movf    ADRESH,W    ; Get MSB of 10-bit value (see PIC16F877 datasheet page-116), and write
      movwf   Temp
      comf    Temp        ; complement the value
      movlw   0x0F
      andwf   Temp,W      ; obtain the bottom nibble
      call    Seven_seg   ; get the value from subroutine, move to PORTB LED's
      movwf   PORTB
      movlw   B'00001000' ; This turns on the 7 seg display output (RA3) connecting to one display.
      movwf   PORTA;
      movlw   .200        ; allow to generate delay (to stall before outputting on other display)
      movwf   count
      call    delay
      swapf   Temp,F      ; swapp top and bottom nibble
      movlw   0x0F
      andwf   Temp,W      ; obtain the top nibble
      call    Seven_seg
      movwf   PORTB
      movlw   B'00000100' ; This sets the output to be on the display connected to RA2
      movwf   PORTA;
      movlw   .200        ; instruction generated delay again.
      movwf   count
      call    delay
      bcf     PIR1,ADIF   ; Clear A/D completion flag
      goto    Loop

delay nop
      decfsz  count
      goto    delay

Seven_seg ; table lists 7 seg pins as dp, g, f, e, d, c, b, a
      andlw   0x0F
      addwf   PCL,F
      retlw   B'11000000' ;0
      retlw   B'11111001' ;1
      retlw   B'10100100' ;2
      retlw   B'10110000' ;3
      retlw   B'10011001' ;4
      retlw   B'10010010' ;5
      retlw   B'10000011' ;6
      retlw   B'11111000' ;7
      retlw   B'10000000' ;8
      retlw   B'10011000' ;9
      retlw   B'10100000' ;a
      retlw   B'10000011' ;b
      retlw   B'10100111' ;c
      retlw   B'10100001' ;d
      retlw   B'10000110' ;e
      retlw   B'10001110' ;f

      END                 ; End of program
```

PROVES OF CODE EFFICIENCE ARE GIVEN IN TASK 3

2.3 Task 3

In the 3rd part the aim was to create an .asm file which would create a program enabling the development board to read the voltage level on a potentiometer on the input RV3, convert it to a digital equivalent using ADC, and create a changing output d.c voltage using PWM.
The input voltage between 0-5v is read in to the ADC, converted to a 0-255 digital equivalent, and then interpreted in to a duty cycle ‘0’ being 0% duty cycle and ‘255’ being 100% duty cycle.

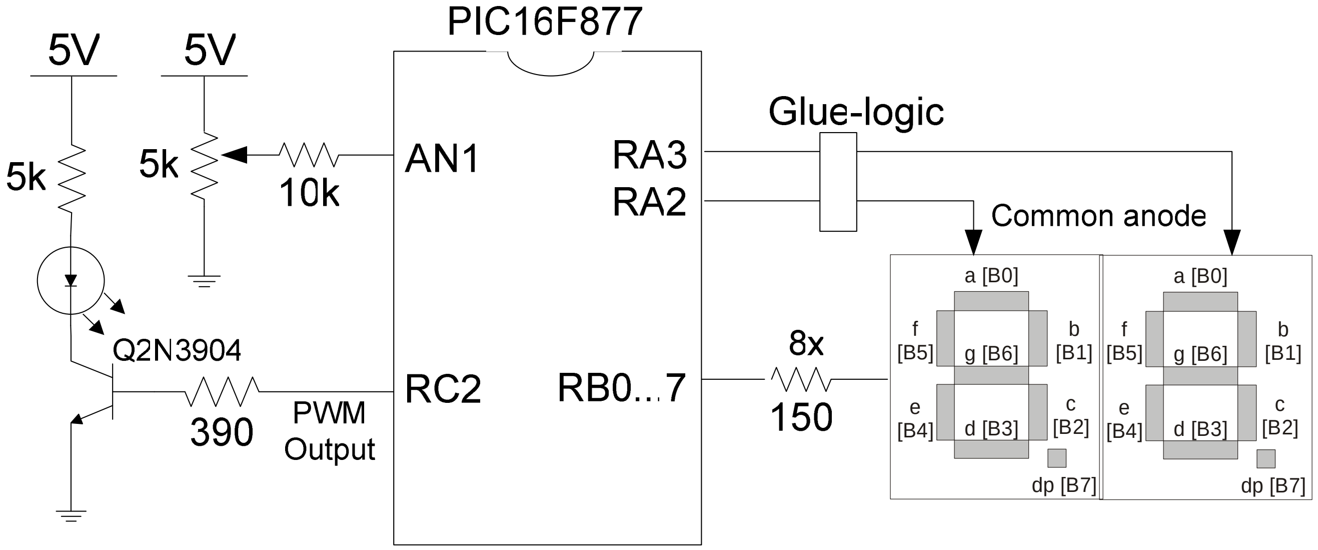
The period of the PWM can be determined using the equation:

PWM period = [(PR2) + 1] * 4 * Tosc * (TMR2 pre-scale value)

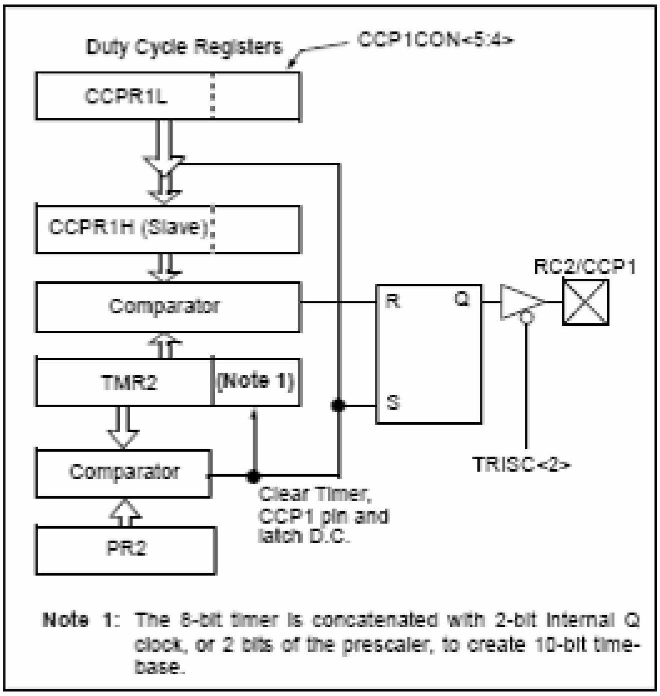
The PR2 value is 254 (because of the lost cycle in , as this is the maximum value to be held in the PWM register (associated with TMR2), the Tosc being 4Mhz and the pre-scalar being 1:1, the PWM period is 255us.
The value needed in PR2 is the number required to represent the intermediate values of the duty cycle. 255 is maximum (100%) and 0 is minimum (0%), therefore from the equation the maximum value of the PWM period should be 255us, this is when PR2 + 1 * 1us = 255us therefore the size needed in PR2 is 254. No duty cycle is representable as ‘off’ voltage and 100% duty cycle is represented as ‘on’ voltage.
Therefore by manually changing the input voltage on the potentiometer, the output voltage level is changed using PWM to digitally alter the value of the output d.c. through use of the transistor.

When development board with the program was connected to a spectrum analyser, the PWM output was observed, and when the duty cycle was half of the PWM period, the digital value on the PIC showed 128 = 2.5v.

The process of part 3 is as follows:



Note: some picture are given later for efficiency prove.

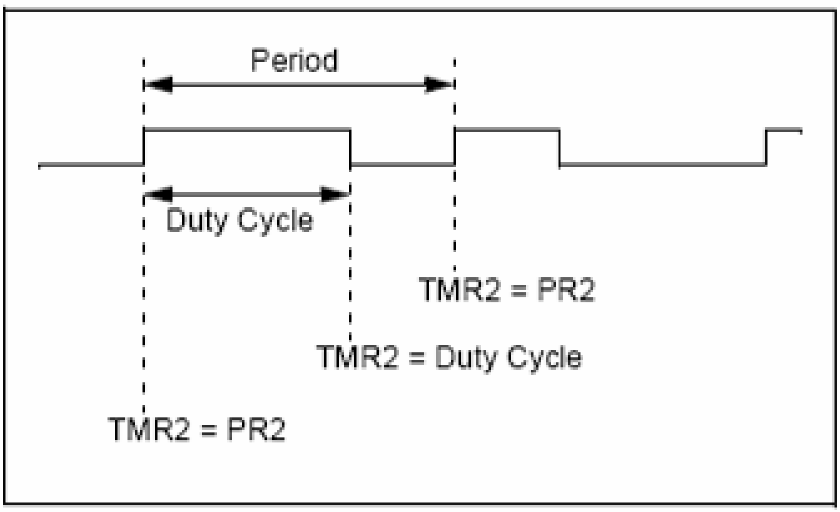


This diagram shows the method of pulse width modulation occuring. The CCPR1L is loaded to CCPR1H at start, then it is compared with the value of TMR2 as TMR2 is incremented until value equals that of CCPR1H, when this occurs, output of flipflop goes low. This is the duty cycle. When the value in TMR2 equals that of PR2 then the timer is reset and the flipflop set and the value in CCPR1L is latched in to CCPR1H. This is the end of the PWM period.

TrisC controls the output of the PWM.
We do not use the fractional part of the conversion.

The basic flow of the program involves setting the registers involved with conversion and input and output, then setting up the timer and PWM duty cycle and period.
Enabling interrupts, starting TMR2, checking for overflow, excecuting PWM ISR, writing result when finished A/D, updating PWM duty cycle/ intensity.
Therefore the program is continually checking the potentiometer input to update the duty cycle for the PWM.

FIGURE 8-4: PWM OUTPUT



Task2&3 Conclusion:

This experiment has been designed for demonstrational purposes and the application has relatively little practical use (as we are adjusting the voltage manually on the input).
It illustrates the point of being able to adjust the voltage digitally, showing that this can be automated and the voltage can be automatically altered using PWM on the digitalized input voltage.

Note: the code is given later with a little improvement (decimal display).

2.4 BONUS ! ! ! (sorry for your time)

I was a little frustrated to finish like that then I decided to improve the last code a little. I made a conversion to allow seeing the output voltage in decimal (more relevant than hexadecimal).

The principle:

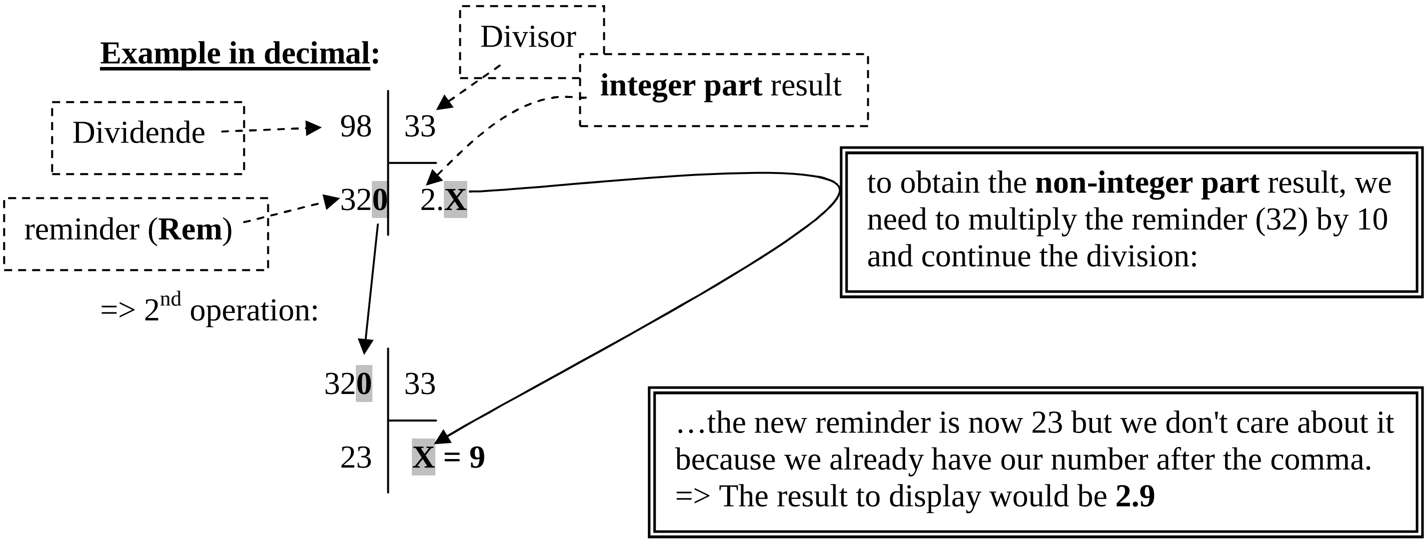
-The maximum value extracted from the ADC is 11111111_B = 255 => corresponds to Vref = 5V
(and 00000000_B => corresponds obviously to 0V)

-Hence, to "normalize" the display, a solution can be to proportionally map [0;255] in [0;5] thus the operation is a division : 255/5 = 51 = $\mathbf{33_H}$

-To achieve this conversion I used an algorithm given in lecture that divides an 8bits value by another 8bits value (the result being also in 8bits for the integer part and for the reminder).

=> My problem was that after the 1st division, I had to multiply the reminder by 10_H and divide again this result by $\mathbf{33_H}$.

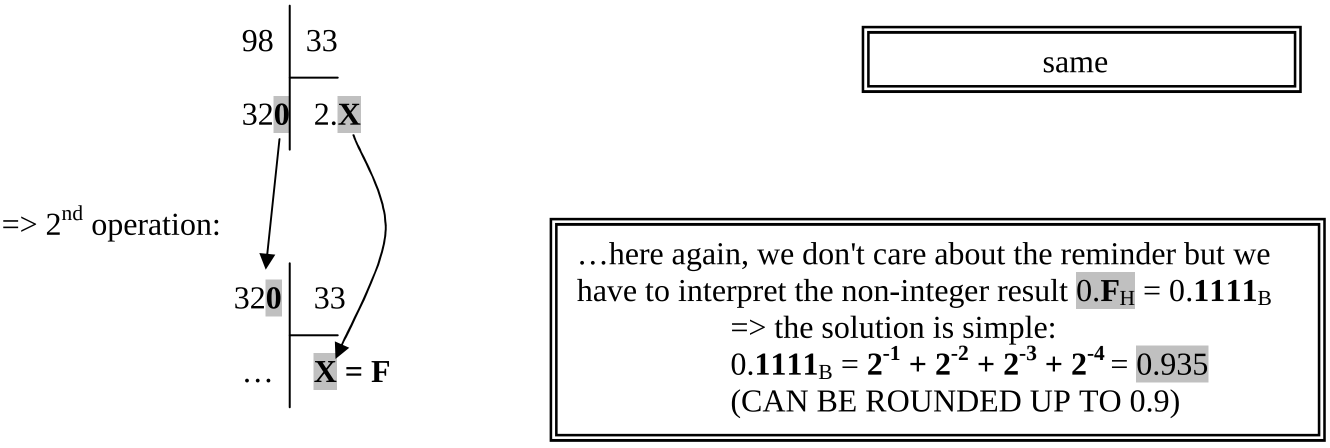
Example in decimal:



to obtain the **non-integer part** result, we need to multiply the reminder (32) by 10 and continue the division:

…the new reminder is now 23 but we don't care about it because we already have our number after the comma.
=> The result to display would be **2.9**

Same in HEXAdecimal:



…here again, we don't care about the reminder but we have to interpret the non-integer result $0.\mathbf{F}_H = 0.\mathbf{1111}_B$
=> the solution is simple:
$0.\mathbf{1111}_B = \mathbf{2^{-1} + 2^{-2} + 2^{-3} + 2^{-4}} = 0.935$
(CAN BE ROUNDED UP TO 0.9)

To implement these operations, we could use an algorithm to do division by 33_H, another for the multiplication by 10_H and do the division by 33_H again, but the code becomes really big.
As the ADC is already not completely exact, we can use a quite good approximation:

=> For the 2nd operation, instead of: $\text{Rem} \times \frac{10}{33} = \text{Rem} \times \frac{1}{33/10}$ => I approximate by: $\text{Rem} \times \frac{1}{3}$

Approximation calculation:
In the worst case: 32_H /3 = 10_H = 1 0000_B = 16 => BUT 320_H /33_H = F_H = 1111_B = 15

…thus we get an error of 1/16 = 6.25 % (not very small but tolerable)

The code:

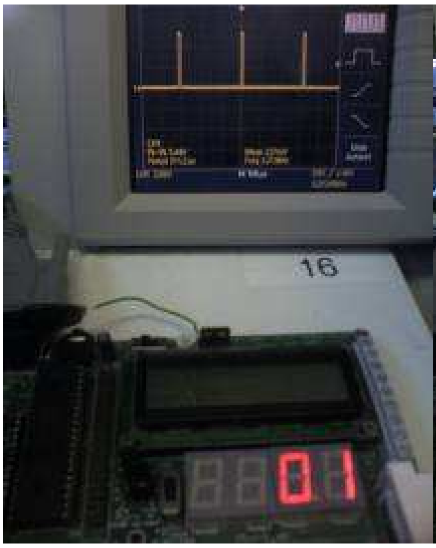
```
; all files declarations are not written here but are obviously in the .asm file

      ORG    0x00
      goto   Init
      ORG    0x04
      goto   ISR

Init  bcf    INTCON, GIE
      btfsc  INTCON,GIE
      goto   Init

      BANKSEL PORTA           ; User BANKSEL on any PORT will goto the right memory page
      clrf   PORTA            ; Clear PORTA
      clrf   PORTB            ; Clear PORTB
      movlw  b'01000001'      ; Setup A/D to read the Potential from the LDR (Channel 0)
      movwf  ADCON0           ; with the parameters include Fosc/8, A/D enabled, Analog Channel 0

      BANKSEL  OPTION_REG                                     ; Select right memory page
      bsf    PIE1,ADIE        ; Enable A/D Interrupt
      movlw  b'00001000'
      movwf  OPTION_REG       ; Set TMR0 with prescale value of 1:1
      bsf    INTCON,T0IE      ; Unmask Timer Interrupt
      movlw  B'00000011'      ; Set RA0, RA1 as Analog Input, and the rest of PORTA as output
      movwf  TRISA;
      clrf   TRISB            ; Set PORTB as output
      bcf    TRISC,2          ; Setup PWM frequency output
      movlw  B'00000000'      ; Set A/D result to be left justified (the 8 MSB result goes in ADRESH)
      movwf  ADCON1           ; and enables all A/D channel with Vref+ = VDD and Vref- = VSS ref
      movlw  b'11111110'      ; Setup PWM frequency at 254 because of the "lost cycle"
      movwf  PR2              ; work out a PR2 (8-bit register) value so that 255 = 100% duty cycle.

      BANKSEL PORTB
      clrf   CCPR1L           ; initialise the duty cyle size at 0
      clrf   CCP1CON
      movlw  B'00000100'
      movwf  T2CON            ; Turn on TMR2 with prescaler of 1:1 and postscale of 1:1
      movlw  B'00001100'
      movwf  CCP1CON          ; Set the Capture/Compare/PWM (CCP) module to just PWM mode
      clrf   Intensity
      bsf    INTCON,PEIE      ; Unmask Peripheral Interrupt
      bsf    INTCON,GIE       ; Unmask Global Interrupt


Loop  movf   Intensity,W      ; Put the content of the variable called Intensity (result of the ADC)
      movwf  CCPR1L           ; in CCP1RL which is the register to modify the PWM duty cycle
      call   Display
      goto   Loop


;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;; Interrupt Service Routine ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

ISR   movwf  w_temp           ; Save W content into w_temp
      movf   STATUS,W         ; Save STATUS content into status_temp before server the interrupt
      movwf  status_temp

Poll  btfsc  INTCON,T0IF      ; Check if Timer interrupting for expired counter?
      call   AD_Start         ; If YES, start A/D conversion
      btfsc  PIR1,ADIF        ; Check if A/D has completed the conversion?
      call   AD_Done          ; If YES, get the A/D result and put "Intensity" in PORTB

      movf   status_temp,W
      movwf  STATUS           ; Restore STATUS content
      swapf  w_temp,F         ; Restore W content
      swapf  w_temp,W

      retfie                  ; Return where the program is interrupted

AD_Start
      bsf    ADCON0,GO        ; Start A/D conversion
      bcf    INTCON,T0IF      ; Clear TMR0 overflow Interrupt

      return                  ; Return to the program where the call is made

AD_Done
      movf   ADRESH,W         ; get MSB of 10-bit value (see PIC16F877 datasheet page-116), and
      movwf  Intensity        ; put the result into variable called Intensity
      bcf    PIR1,ADIF        ; Clear A/D completion flag

      return                  ; Return to the program where call is made
```

```
Display ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
      call   convert          ; this subroutine converts hexa-display in decimal display.
      movf   IntDispl,W       ; prepare the integer result part to be displayed

      call   Seven_seg_int    ; use the appropriate table to display the float part.
      movwf  PORTB            ; send the "coded" valur to PORTB (to be displayed on the LEDs)
      movlw  B'00000100'      ; turn on the 7 seg connected to RA2 to display the integer part.
      movwf  PORTA;

      movlw  .200             ; generate delay (to stall before outputting on other display)
      movwf  count
      call   delay

      movf   FltDispl,W       ; prepare the float result part to be displayed
      call   Seven_seg_flt    ; use the appropriate table to display the float part.
      movwf  PORTB            ; send to PORTB
      movlw  B'00001000'      ; turn on the 7 seg connected to RA3 to display the float part.
      movwf  PORTA;

      movlw  .200             ; delay again...
      movwf  count
      call   delay
      return

delay nop
      decfsz count
      goto   delay
      return


convert ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
; this subroutine "converts" a binary value between 00000000 and 11111111 in "decimal"
; considering that 11111111 = 5V then divide by .51 = 0x33 and use 2 tables to display.
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
      movlw  .51
      movwf  Divisor          ; and put (decimal)51 = 0x33 in the Divisor

      call   DIV8by8          ; call the division subroutine

      movf   Int,W
      movwf  IntDispl         ; save the integer part resulted from the division by 51

; continue the division but divide by 3 (because it's roughly = to multiply by 0x10 and divide by 0x33)
      movlw  3
      movwf  Divisor          ; then set the divisor to 3
      movf   Rem,W
      movwf  Dividend         ; finish to prepare the division: set the dividend to Rem

      call   DIV8by8          ; effectuate it : Rem / 3

      movf   Int,W
      movwf  FltDispl         ; save result of last division to be able to display the float part
      return


DIV8by8 ;;;;;;;;;;;;;;;;;;;;;;;; source : Web CT ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
; division of an 8bit dividende by an 8bit divisor => result: 8bit Integer part and 8bit Reminder
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
      movf   Dividend,W
      movwf  Int              ; final Integer part will be in Int
      clrf   Rem              ; final remainder will be in Rem
      movlw  8
      movwf  count

branch
      bcf    STATUS,C
      rlf    Int,F
      rlf    Rem,F
      movf   Divisor,W
      subwf  Rem,W
      btfss  STATUS,C         ; if we did not borrow then carry is set
      goto   chk              ; is clear and we do not want to store Rem
      movwf  Rem              ; is set and we need to store Rem and change Int_0
      bsf    Int,0

chk   decfsz count,F          ; check the count
      goto   branch

      return
```

```
Seven_seg_int ;;;;;;;;;;;;;;;;;;;;;;; table lists 7 segments for integer part ;;;;;;;;;;;;;;;;;;;;;;;;;;

      andlw   B'00000111'      ; to be sure not to go out of the table => no need to "and" the
      addwf   PCL,F            ; PC with 0x0F because the max value is supposed to be 5 (on 3bits)

                               ; display:
      retlw   B'01000000'      ; 0.
      retlw   B'01111001'      ; 1.
      retlw   B'00100100'      ; 2.
      retlw   B'00110000'      ; 3.
      retlw   B'00011001'      ; 4.
      retlw   B'00010010'      ; 5. => as the maximum voltage is 5V we don't need more

      retlw   B'10000110'      ; display: "E" in case of Error
      retlw   B'10000110'      ; display: "E" in case of Error
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

Seven_seg_flt ;;;;;;;;;;;;;;;;;;;;;;; table lists 7 segments for float part ;;;;;;;;;;;;;;;;;;;;;;;;;;;;

      btfsc   FltDispl,4       ; due to the approximation, the result can be 10000 instead of 1111
      retlw   B'10011000'      ; we thus display 9 (because 1111 => 0.935  CAN BE ROUNDED UP TO 0.9)

      andlw   0x0F             ; ...to be sure not to go out of the table
      addwf   PCL,F
                               ; display:
      retlw   B'11000000'      ; 0
      retlw   B'11111001'      ; 1 => because 0001 => 0.0625 CAN BE ROUNDED UP TO 0.1
      retlw   B'11111001'      ; 1 => because 0010 => 0.125  CAN BE ROUNDED UP TO 0.1
      retlw   B'10100100'      ; 2 => because 0011 => 0.1875 CAN BE ROUNDED UP TO 0.2
      retlw   B'10110000'      ; 3 => because 0100 => 0.25   CAN BE ROUNDED UP TO 0.3
      retlw   B'10110000'      ; 3 => because 0101 => 0.3125 CAN BE ROUNDED UP TO 0.3
      retlw   B'10011001'      ; 4 => because 0110 => 0.375  CAN BE ROUNDED UP TO 0.4
      retlw   B'10011001'      ; 4 => because 0111 => 0.44   CAN BE ROUNDED UP TO 0.4
      retlw   B'10010010'      ; 5 => because 1000 => 0.5
      retlw   B'10000011'      ; 6 => because 1001 => 0.5625 CAN BE ROUNDED UP TO 0.6
      retlw   B'10000011'      ; 6 => because 1010 => 0.625  CAN BE ROUNDED UP TO 0.6
      retlw   B'11111000'      ; 7 => because 1011 => 0.687  CAN BE ROUNDED UP TO 0.7
      retlw   B'10000000'      ; 8 => because 1100 => 0.75   CAN BE ROUNDED UP TO 0.8
      retlw   B'10000000'      ; 8 => because 1101 => 0.8125 CAN BE ROUNDED UP TO 0.8
      retlw   B'10011000'      ; 9 => because 1110 => 0.8725 CAN BE ROUNDED UP TO 0.9
      retlw   B'10011000'      ; 9 => because 1111 => 0.935  CAN BE ROUNDED UP TO 0.9
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

      END
```

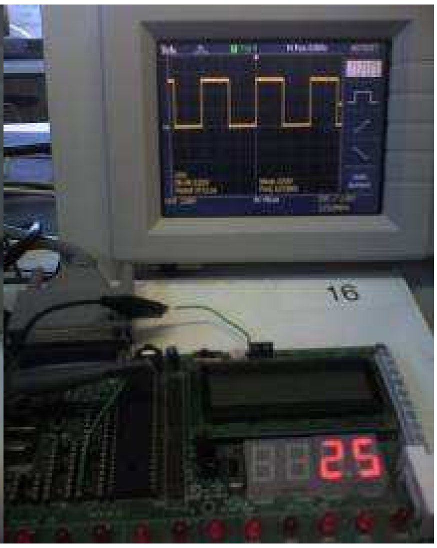
The pictures:

I've had the chance to have better than these pictures as evidence, I've had a witness (you).

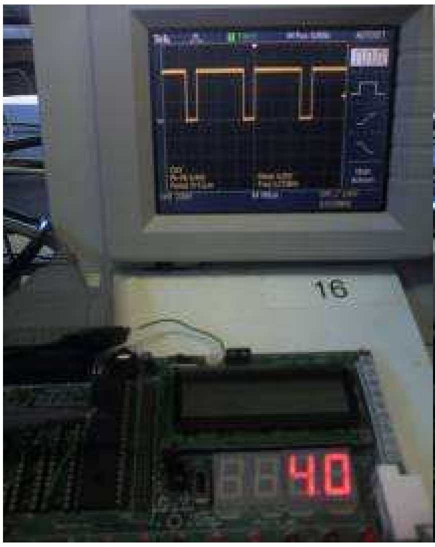


0.1V

1.0V



2.5V



4.0V

4.9V

5.0V

Note: I've also done pictures for task1, 2 and 3 (+ hexadecimal display) but it's not interesting because now we've got the decimal display. As all this lab was oriented on the final experiment, I finally didn't really explain all register and bit used, but I think that you don't really need me to prove you that I know how to read a data sheet, copy it and paste it (I've done it in task 1).

3 Conclusion

This has many applications in industry by being able to digitally control the output DC without altering the voltage supply. This application of the PIC is typical, such a small chip being able to perform as glue logic to perform “fixes” to digital systems without having to redesign the whole circuitry. When the board was connected to a small motor, it was observed that the speed of the motor increased as the voltage applied was increased. It was also observed that the motor created background Electro Magnetic Force noise which created distortion on the waveform observable. This is due to Lenz's effect of the movement of the motor and the conflicting magnetic and electrical forces inducing current in the system. The method in which the PWM was connected to the motor/LED was to connect the PWM output to the base of a transistor via a resistor. This transistor averages the voltage of the PWM because the transistor is unable to switch within the same period as the PWM. Also the transistor controls the current flowing from collector to the emitter and therefore the voltage over the transistor. This enables the user to control the voltage over the LED or Motor. This technique could be utilised in cruise control in automotive industry to control the fuel injected in to the car, it could be used in cutting edge “intelligent carpet technology”, where the carpet senses where people are in the room and therefore alters the intensity of light in various areas of the room, central heating controls, curtain controls etc.. This would also be viable to use for security control systems.

Finally, in this lab I have learnt the concepts of A/D conversion, interfacing and the applications of the PIC877 series, the use of 7 segment displays and PWM using the PIC development board. My programming skills with PIC16F877 assembly code have improved and my ability and familiarity with the use of timer capabilities and interrupt service routines have also improved greatly. I am now fairly confident in the use of PIC assembly code to create modular programs calling and returning from subroutines. I can see where and how the PIC microcontroller can be used as glue and fix logic to various large and complex digital designs to save money from not having to reconstruct these large designs, and am confident in how to go about creating a PIC solution.