Lab 5

Formal Laboratory Report

EE202 – Embedded Systems

Student: Adelin Denis Diac

Student Number: 20337941

Date: 24/03/2022

Lecturer: Robert Sadleir

Faculty of Engineering & Computing
DCU Glasnevin Campus
Glasnevin
Dublin 9

Contents

Objective

The aims of part 1 was to understand how Analog to Digital Conversion is done by the PIC16F690 and using this tool to make a small thermometer system. Part 2 showed the benefits and drawbacks of using High Level Programming when used with Embedded Systems.

Procedure & Results

Part 1a – Analogue to Digital Conversion (Initial Setup)

In this part, it was required to set up the circuit as per the given diagram and then complete the given code to display the percentage voltage on the two 7-segment displays.

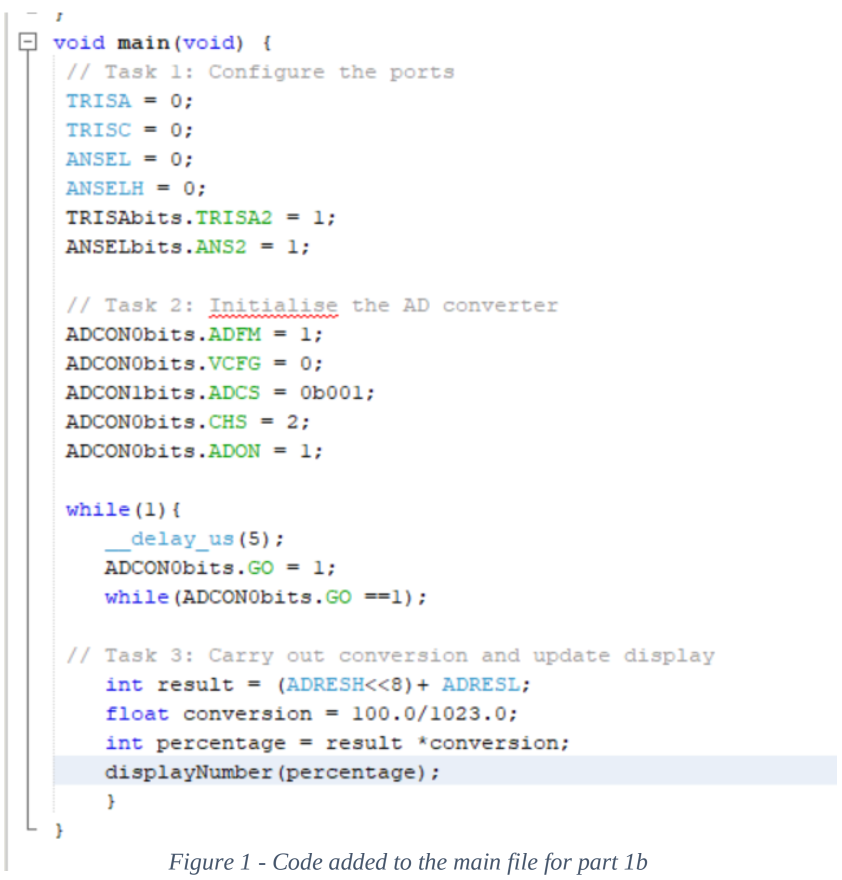
Link to see final circuit working

- The PICKIT was connected to the PIC16F690, and the two seven segment displays were connected as per the diagram. In this set up, there was no need for as many resistors since a different method of displaying the numbers was used. In this case only one segment was lit up at a time, however the delay between them was so short and to the human eye it looked like they were all lit up at once. This was implemented mainly via the software.
- Once the circuit was set up, it was required to upload the given code, and update it so that it can read an analogue value from the AN2 pin and begin an analogue-to-digital conversion.
- Firstly, pins RA<5:4> and RC<7:0> were set as digital outputs. This was done by setting TRISA, TRSIC, ANSEL and ANSELH to 0.
- Pin AN2 then had to de initiated as an analogue input pin. This was done by setting TRISA bit 2 and ANSEL bit ANS2 to high. The variable names defined in the xc.h file were used to initialise these values.
- The AD module was then configured so that the digital value is stored in right justified format. This was done by setting the ADFM bit of ADCON0.
- The VCFG bit was the cleared, so that VDD was used as the reference voltage.
- The ADCS bits of the ADCON1 file register was given the value 0b001, so that the clock period is 2µs.
- It was also required to use Channel 2, since this is the one that corresponds to the AN2 pin. This was done by giving the CHS bits of ADCON0 register the value 2.
- The final initialisation was to turn on the Analogue to Digital converter module. To do this, the ADON bit of ADCON0 had to be set.

These initialisations were required for the PIC to store the final converted digital value in the format which we desire. Next, it was programmed so that the code would run in a continuous loop, and in each iteration, it would convert the analogue value on the AN2 pin into a digital value.

- Firstly, before a conversion started, a delay of 5µs was placed. To start a conversion, the GO bit of the ADCON0 register was set. This bit is automatically cleared when the conversion ends, so it was required to keep the program at a stand still while this bit is set. Once this was complete, the 10-bit value was stored in two different registers. The high register was shifted 8 bits to the left and combined with the low address and stored in a variable called result. This was divided by 1023 (which is the 10-bit value that would represent the maximum voltage) and multiplied by 100 to give a percentage.
- The percentage was the displayed on the 7-segment display.

The final code implemented in this section is seen below in figure 1.



```
void main(void) {
 // Task 1: Configure the ports
 TRISA = 0;
 TRISC = 0;
 ANSEL = 0;
 ANSELH = 0;
 TRISAbits.TRISA2 = 1;
 ANSELbits.ANS2 = 1;

 // Task 2: Initialise the AD converter
 ADCON0bits.ADFM = 1;
 ADCON0bits.VCFG = 0;
 ADCON1bits.ADCS = 0b001;
 ADCON0bits.CHS = 2;
 ADCON0bits.ADON = 1;

 while(1){
    __delay_us(5);
    ADCON0bits.GO = 1;
    while(ADCON0bits.GO ==1);

 // Task 3: Carry out conversion and update display
    int result = (ADRESH<<8)+ ADRESL;
    float conversion = 100.0/1023.0;
    int percentage = result *conversion;
    displayNumber(percentage);
    }
}
```

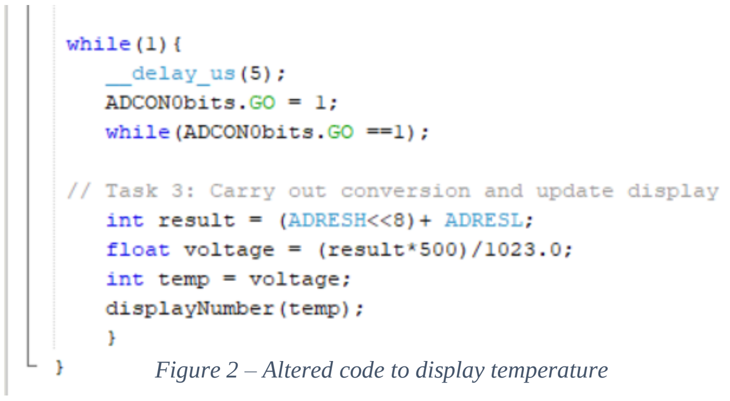
Figure 1 - Code added to the main file for part 1b

This final embedded system works by reading the voltage value on pin AN2. It then converts this to a digital value and this value is turned into a percentage of the input voltage via the last few lines of the code. It then displays the number on the two displays. As the potentiometer is rotated, the voltage increases and the value on the display changes.

Part 1b – Analogue to Digital Conversion (Temperature Measurement)

For this section the embedded system was slightly changed so that the display shows the temperature instead of a percentage value.

- For this section, the circuit was slightly changed, instead of the potentiometer the LM35 was installed. The middle pin of this was connected to the AN2 pin and then the code from Figure 1 was slightly altered so that the display shows the temperature of the room.
- The code was only changed in the "Task 3" section, since everything else had to be the same in order to read the voltage on pin AN2, where the middle pin of the LM35 was connected. The code can be seen in figure 2 below.



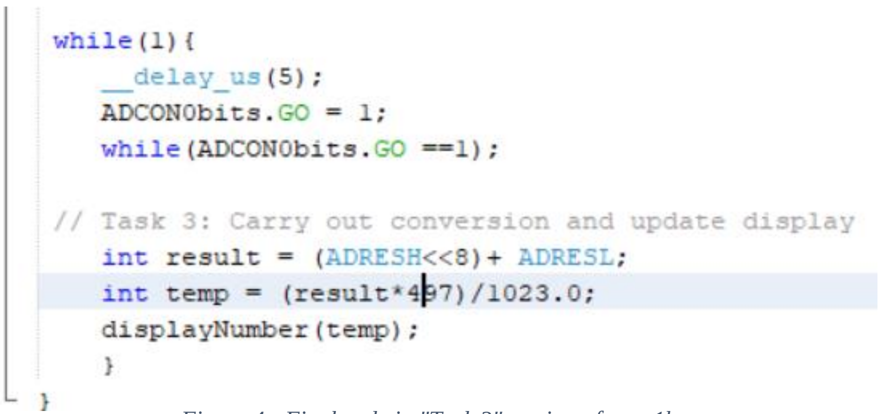
```
 while(1){
    __delay_us(5);
    ADCON0bits.GO = 1;
    while(ADCON0bits.GO ==1);

 // Task 3: Carry out conversion and update display
    int result = (ADRESH<<8)+ ADRESL;
    float voltage = (result*500)/1023.0;
    int temp = voltage;
    displayNumber(temp);
    }
}
```

Figure 2 – Altered code to display temperature

- The LM35 gives off 10mV per degree Celsius. So, in order to get the voltage its giving off, the 10-bit digital value generated after the conversion was found as a fraction of the max value (1023) and then it was multiplied by the input voltage, 5V. However, to convert from Volts to degrees Celsius, it was then required to multiply by one hundred.
- This was implemented differently in figure 2 to allow for as accurate of a result as possible. The multiplication was done first and then the division by 1023.
- This value was then displayed on the two seven segment displays.

This was found to give a slightly larger result, and the code was changed to try and rectify this. The voltage was measured, and it was found that it was around 4.8 volts, however it did ted to fluctuate slightly. The fluctuations were more than likely due to the power source used. The power rails were connected directly to the plug which was set at 5V. This isn't a very precise component, and it has caused problems in the past. The connections to this were also hard to keep in perfect contact. The cables were connected with crocodile clips, as can be seen in the video linked in part 1a.

The power source was changed into the breadboard power supply (BBP-32701). The voltage across all power rails were tested, and it was found the voltage no longer fluctuated. This gave a consistent 4.97 volts throughout, so the code was also changed to satisfy this. The final code can be seen in figure 3, and the final circuit set up in figure 4.



```
while(1){
    __delay_us(5);
    ADCON0bits.GO = 1;
    while(ADCON0bits.GO ==1);

// Task 3: Carry out conversion and update display
    int result = (ADRESH<<8)+ ADRESL;
    int temp = (result*497)/1023.0;
    displayNumber(temp);
    }
}
```

Figure 4 - Final code in "Task 3" section of part 1b

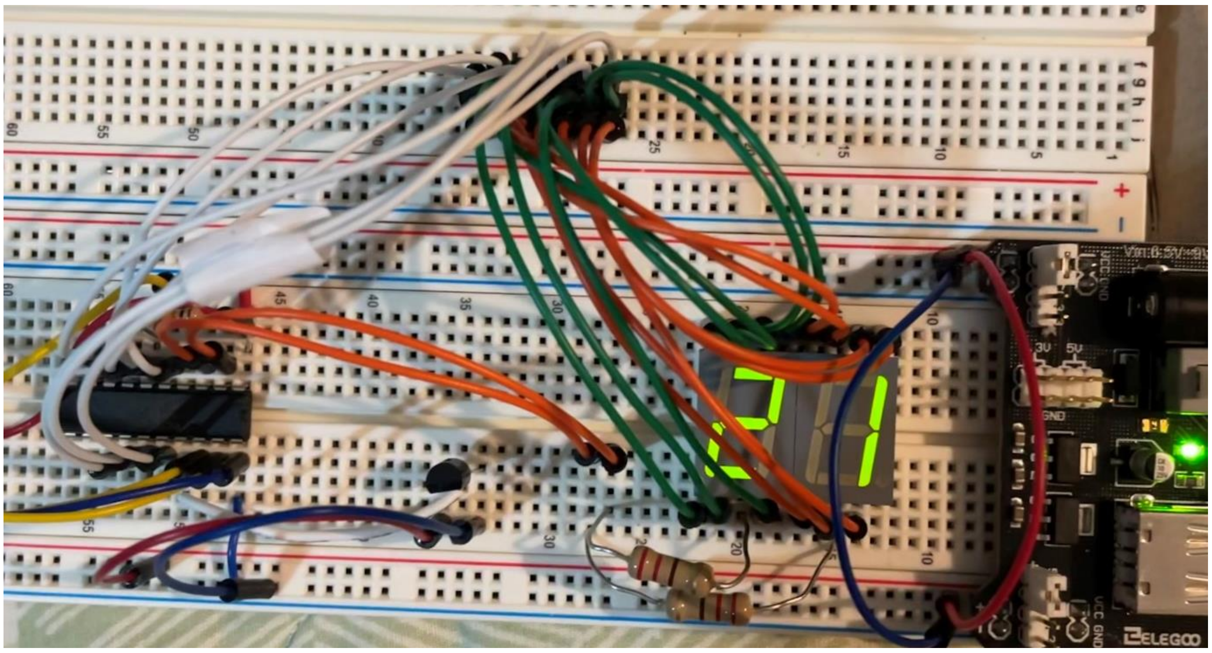
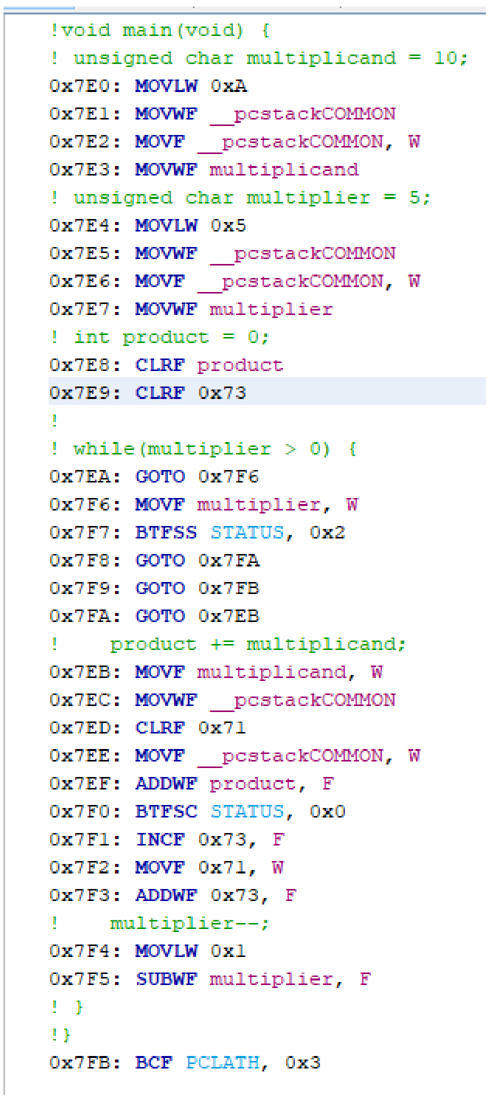


Figure 3 - Final circuit set up for part 1b - including the new BBP-32701

Part 2 – Understanding the output of the Compilation process

- The first thing which can be seen in the disassembly example is that when an integer/character is declared and initialised, the compiler first stores it in the "_pcstackCOMMON" file and then stores it in the address with the name of the variable. The pcstack address in this example was register 0x70. This seems a little unnecessary, as it uses 4 instructions when it could be done in only 2.
- It also does not write out the instructions in their executed order. As can be seen at instruction 0x7F6, the previous instruction would be expected to be 0x7F5, however this is down near the bottom of the code. If it was done by a programmer, this might've been made more obvious to allow for easier readability.
- At instruction 0x7F8, it calls a "GOTO" instruction which goes to another "GOTO" instruction which then goes onto the required part of the code. Again, this isn't necessary as the first "GOTO" could have not been implemented.
- The one section that it does similarly to the example from the notes is the one at instruction 0x7F0. Here, after the multiplier has been added to the product, it checks if a carry has occurred (if our value overflowed) and increments the most significant byte of the product if necessary.
- From the values of the "product" and the MSB of the product, it can be seen that the values are stored in "Little Endian Byte Ordering".
- After this, it must decrement the multiplier. In the notes example, this was done using the "decf" instruction which decrements one from the specified file, however the compiler did it slightly different. It moved the value 1 to the working register, then subtracting it from the multiplier file.
- Both the notes and compiler then check to see if the multiplier is zero by checking if the zero flag is set. If it is, it jumps to the end of the code where it would clear the 3rd bit in the PCLATH register.



```
!void main(void) {
! unsigned char multiplicand = 10;
0x7E0: MOVLW 0xA
0x7E1: MOVWF __pcstackCOMMON
0x7E2: MOVF __pcstackCOMMON, W
0x7E3: MOVWF multiplicand
! unsigned char multiplier = 5;
0x7E4: MOVLW 0x5
0x7E5: MOVWF __pcstackCOMMON
0x7E6: MOVF __pcstackCOMMON, W
0x7E7: MOVWF multiplier
! int product = 0;
0x7E8: CLRF product
0x7E9: CLRF 0x73
!
! while(multiplier > 0) {
0x7EA: GOTO 0x7F6
0x7F6: MOVF multiplier, W
0x7F7: BTFSS STATUS, 0x2
0x7F8: GOTO 0x7FA
0x7F9: GOTO 0x7FB
0x7FA: GOTO 0x7EB
!    product += multiplicand;
0x7EB: MOVF multiplicand, W
0x7EC: MOVWF __pcstackCOMMON
0x7ED: CLRF 0x71
0x7EE: MOVF __pcstackCOMMON, W
0x7EF: ADDWF product, F
0x7F0: BTFSC STATUS, 0x0
0x7F1: INCF 0x73, F
0x7F2: MOVF 0x71, W
0x7F3: ADDWF 0x73, F
!    multiplier--;
0x7F4: MOVLW 0x1
0x7F5: SUBWF multiplier, F
! }
!}
0x7FB: BCF PCLATH, 0x3
```

Figure 5 - Assembly code generated when debugging C code

Overall, the code generated by the compiler is longer, and it also requires longer amount of time to complete. The programme would be more efficient if it was all done using assembly language, however there are also some clear advantages of using higher level languages.

<u>Advantages of using Higher Level Language</u>

- Easier to understand since it is closer to human language in comparison to the assembly language.
- It also allows for more complex algorithmic problems to be easily solved with the use of its loops and control statements.
- Although the code itself isn't as efficient, it takes a lot less time to write the same code in Higher Level Languages.

<u>Disadvantages of using Higher Level Language</u>

- As can be seen from the examples above, compilers don't come up with the most efficient code. It can sometimes do things in an awkward way in comparison to how a programmer would do it.
- The longer codes can also take longer time to compile them and turn into assembly and then to binary.
- There is also no direct control over hardware. The programmer can't really select where to save certain variables etc.

Conclusion

By completing this lab session, the following lessons were learned:

- How to programme the PIC16F690 to read analogue values and convert them to a 10-bit digital value.
- Using the digital value to display the percentage voltage at the pin being read.
- Using the Analogue to Digital converter of the PIC16F690 to build a small Embedded System which displays the temperature outside.
- Understanding the advantages and disadvantages of high-level programming and seeing how the compiler converts C code into Assembly.