Embedded Systems : Analog to Digital Conversion and High-level Programming

Date: 04/04/24

Name: Alfie Antony

Student ID: 22504619

Objective:

The goal of this experiment is to make a digital to analog converter and a thermometer. We will learn how the C programming language works in order to perform these conversions. From this, we will look at the benefits of the compiler of higher-programming level languages and learn the process of higher-level programming. The expectations of this project is that we will be able to use a potentiometer to increment the digital conversion and a LM35 temperature sensor to measure the temperature of the room.

Procedure:

The procedure for this lab exercise has the following parts: 1(a), 1(b) and 2. Initially, we will set up the digital to analog converter. Hence we use the following code template below for 1(a) in terms of software:

```
01:  #include <xc.h>
02:
03:  #define _XTAL_FREQ 4000000 // 4Mhz
04:
05:  // Set config word
06:  #pragma config FOSC=INTRCIO, WDTE=OFF, PWRTE=OFF, MCLRE=ON, \
07:  CP=OFF, CPD=OFF, BOREN=OFF, IESO=OFF, FCMEN=OFF
08:
09:  int get7SegmentCode(int value) {
10:       switch(value) {
11:            case 0: return 0b00111111;
12:            case 1: return 0b00000110;
13:            case 2: return 0b01011011;
14:            case 3: return 0b01001111;
15:            case 4: return 0b01100110;
16:            case 5: return 0b01101101;
17:            case 6: return 0b01111101;
18:            case 7: return 0b00000111;
19:            case 8: return 0b01111111;
20:            case 9: return 0b01101111;
21:            case 72: return 0b01110110; // H
22:            case 73: return 0b00000110; // I
23:            case 76: return 0b00111000; // L
24:            case 79: return 0b00111111; // O
25:       }
26:  }
27:
28:  void displayNumber(int number) {
29:       int tens, units;
30:       if(number<0){tens=76; units=79;}
31:       else if(number>99){tens=72; units=73;}
32:       else {tens = number/10; units = number%10;}
33:
34:       RA4 = 0; // units
35:       RA5 = 1; // tens
36:       int unitsCode = get7SegmentCode(units);
37:       PORTC = unitsCode & 0b00000001; __delay_ms(1);
38:       PORTC = unitsCode & 0b00000010; __delay_ms(1);
39:       PORTC = unitsCode & 0b00000100; __delay_ms(1);
40:       PORTC = unitsCode & 0b00001000; __delay_ms(1);
41:       PORTC = unitsCode & 0b00100000; __delay_ms(1);
42:       PORTC = unitsCode & 0b01000000; __delay_ms(1);
43:       PORTC = unitsCode & 0b10000000; __delay_ms(1);
44:       PORTC = 0x00;
45:
46:       RA4 = 1; // units
47:       RA5 = 0; // tens
48:       int tensCode = get7SegmentCode(tens);
49:       PORTC = tensCode & 0b00000001; __delay_ms(1);
50:       PORTC = tensCode & 0b00000010; __delay_ms(1);
51:       PORTC = tensCode & 0b00000100; __delay_ms(1);
52:       PORTC = tensCode & 0b00001000; __delay_ms(1);
53:       PORTC = tensCode & 0b00100000; __delay_ms(1);
54:       PORTC = tensCode & 0b01000000; __delay_ms(1);
55:       PORTC = tensCode & 0b10000000; __delay_ms(1);
56:       PORTC = 0x00;
57:  }
58:
59:
60:  void main(void) {
61:
62:       // Task 1: Configure the ports
63:       TRISA = 0;
64:       TRISC = 0;
65:       ANSEL = 0;
66:       ANSELH = 0;
67:
68:       // Task 2: Initialise the AD converter
69:
70:       while(1){
71:
72:            // Task 3: Carry out conversion and update display
73:            int result = 34;
74:            displayNumber(result);
75:       }
76:  }
```

We use this template in order to display the numbers on the 7 segment display. As you can see we use a case statement-switch in order to switch between the different numbers displayed in binary. This switches from the numbers (0-9) to the letter O. In the second function, displayNumber() will go through the numbers 0-99. We use an else-if statement so that when the number is < 99 it will be LO and HI if > 99. However, if the number is between 0-99 it will divide this by 10 for the units and tens. We see that when the function has the units argument, RA4 is set to 0 and RA5 is set to 1 which means that the tens segment is displayed whereby each bit of PORTC is moved through with a delay of 1 ms. After this is done we clear PORTC. The same process is repeated for the tens argument with the opposite result occurring. We modify the code in order to get the following result:

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

 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);
 int result = ((ADRESH<<8)+ADRESL);
 float conversion = 100.0 / 1023.0;
 int percentage = result * conversion;
 displayNumber(percentage);
 }
}
```

From this code we will be able to obtain the analog-digital conversion. We will change PORTA and PORTC to digital whereby all the pins in PORTC and PORTA are set to an output. The ANSEL port will set the input to analog. ANSELH will be set to analog by default and read the input as an analog input. After this, we will set the TRISA2 bit in the TRISA register to 1 in order to set it as an analog input. We do the same for the ANS2 bit in the ANSEL register. In Task 2, we will configure the ADFM bit in order to select the right justified format. We will set VCFG so we can configure reference voltage. ADCS sets the A/D clock-period. We will make the AN2 pin select channel 2. Finally, we will turn on the ADC module. After this, we will make an infinite while loop with a delay of 5 milliseconds. From here we will switch on the analog to digital converter. We make a while loop in order to check the conversion. We will use the result variable to combine the 10 bits of the conversion. This will allow us to make the conversion of floating factors and multiply with the result. In the end, we will display the percentage.

In terms of hardware, we had to connect up PORTC pins (0-6) for both the tens and units 7-segment display. Each pin of the PORTC will go to (a-g) of the 7-segment display. We connect up the common cathode of both the tens and units 7-segment display to RA4 and RA5. Finally, we connect pin 17 to the 2nd pin of the potentiometer. We will give its positive terminal to pin 1 of the potentiometer and negative to pin 3.

Next, we will look at part 1(b). For this we have to make the A/D conversion into a temperature value. For this, we will keep our result as normal by combining 10 bits of the conversion. However, instead of using the percentage conversion. We will use temperature instead. Since, I obtained a value of 5.12 V when I measured the voltage of the circuit. I convert this to a floating point conversion. Thereby, I will divide this conversion by 0.01 in order to make the conversion mV/Celsius. This will allow me to multiply the result variable and obtain the value for temperature in Celsius. The temperature reading I obtained from my embedded system was 16.6 degrees Celsius.
Whereas in terms of hardware all we had to do was switch the potentiometer with the LM35 temperature sensor.

```
 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);
 int result = ((ADRESH<<8)+ADRESL);
 float conversion = 5.12 / 1023.0;
 int temperature = result * (conversion)/0.01;
 displayNumber(temperature);
 }
}
```

```
#include <xc.h>

#define _XTAL_FREQ 4000000 // 4Mhz

#pragma config FOSC=INTRCIO, WDTE=OFF, PWRTE=OFF, MCLRE=ON, CP=OFF, \
                CPD=OFF, BOREN=OFF, IESO=OFF, FCMEN=OFF

void main(void) {

    unsigned char multiplicand = 10;
    unsigned char multiplier = 5;
    int product = 0;

    while(multiplier > 0) {
        product += multiplicand;
        multiplier--;
    }
}
```

In terms of Part 2, we analyse each line of the C code above in terms of assembly language representation. This first line of code from the resulting assembly corresponds to the C code as it is describing the header file associated with the XC compiler that contains all the functions necessary. This is related to C code as it is usually at the top of a program to indicate that we will be using the functions related to a programming language. This is the same for assembly as we would also use the <xc.inc> file. In the second line of the code we are defining the oscillatory frequency. This doesn't relate much to C code or assembly language. Next, we look at another feature of the code which is pragma that will allow us to use different functionality of the microcontroller and be able to control it. This will allow us to specify the configuration bits in the microcontroller using C code. However, this feature also doesn't relate much to assembly language representation. This has a similar functionality when using custom linker options, -Pres_vect=0h whereby we are telling the code how it should be handled. In the next line, there is a connection between the assembly language as it is looking at the main() and has a void as an argument and data type preceding it. When it comes to assembly we also use MAIN which will help us start the instructions we want. Afterwards, in terms of C code we are initialising a variable 'multiplicand' to have the value 10. We would do something like this assembly, except we would use the move instruction and move this value from a working register into a specific file register. In the line after, we would do the same steps as it is like moving a value to the working register and to the file register. This is the same when we initialise a product. When we look at the while loop we would also use loop in assembly language which would also have a condition beside it e.g. nop. After that, if we look at the line that adds the variable product it is similar to the augend operation we use in assembly. After this we will decrement the multiplicand variable. Once this happens we exit the loop. In assembly, we also have common outcomes such as when we type goto Loop which will go back to the loop and then exit the MAIN.

Results:

In terms of the results for the first experiment, we are able to demonstrate that there was an analog to digital conversion that could be used as a counter. The initial part of the first experiment allowed us to increment from (0-99) using a potentiometer. This result was to be expected as we made conditions that would meet the requirements. As well as that, we configured the ports correctly, initialised the AD converter and updated the display. In the final part, we set up a thermometer. Instead of the potentiometer, we used a LM35 temperature sensor. We modified the code so that it could take temperature in the following format: mV/Celsius. This caused us to create a temperature variable which read the temperature of the environment that the embedded system was placed in.
The result of the second part was that we compared the resulting assembly representation to the corresponding C code. In addition to that, we came to the conclusion that: a) the program assembly language representation of the C program is smaller compared to the version in the notes. This is because it took around 4 lines of code in version 1 of the notes compared to 1 line of code in the version we viewed in the lab document. b) However, we came to the conclusion that the program from version 1 of the notes probably runs faster than the C code presented as it takes less time to compile and run. As well as that, there are specific memory locations allocated for how each operation of the program should move to. This isn't specifically defined in the assembly language representation of the C code. Finally, we also saw the benefits and drawbacks of programming with a high-level language. The benefit of programming with a high-level language is that it is more readable compared to an assembly language as some of the functions in assembly take a while to understand. Whereas, we use ordinary human language to process higher-level programming languages. Another benefit of high-level languages is that it takes far less lines of code in order to perform operations compared to one operation in assembly language taking many lines of code. However, a drawback associated with

higher-level languages is that compiling is slower and less efficient. This can be seen as we are compiling the code ourselves in assembly and can make it faster. Whereas, compiling in a higher-level language cannot be compiled by us and is compiled by the compiler which depends on how efficient the creator of the compiler makes it. This makes it open to ambiguity giving an erratic compiling time.

Conclusions:

In conclusion, I learned how to program C code (higher-level languages) in order to enable interrupts within my embedded system. This can be seen in the initial setup to the first experiment whereby I increment a counter (0-99) and create a thermometer by changing the variables to suit the embedded system. Finally, I was able to relate assembly to C code and see how both have their pros and cons. Overall, I encountered extremely minor errors which were usually in incorrect syntax within my code.