9 Lab 9 - Analogue to Digital Conversion Module

9.1 Aim

This laboratory practical will investigate methods for configuring and using the analogue-to-digital convertor (ADC) module on the PIC16F882 to convert voltages to digital values. The relationship between converted digital values to physical quantities such as voltage will be investigated.

9.2 Learning Outcomes

After this laboratory practical you will:

- Understand how a physical quantity can be represented by a digital value through the use of an analogue transducer and the process of analogue to digital conversion.
- Understand the capabilities and limitations of the 10 bit ADC module on the PIC16F882.
- Be able to configure the PIC16F882 ADC module correctly.
- Be able to create useful code to monitor analogue voltages using polling and interrupts.
- Implement a library file of useful ADC functions.

9.3 Background

9.3.1 Introduction

An analogue-to-digital convertor (ADC) is a system that converts an analogue signal (voltage) that is continuous in both time and magnitude, into a digital representation. In order to do this, the signal is first *sampled* before being *quantised.* Sampling is the act of taking a 'snapshot' of the voltage at a point in time. Quantisation is the process of converting a sampled voltage into a discrete digital value. An ADC converts an analogue signal into a digital signal that is discrete in both time and amplitude.

In most applications ADC conversion takes place at a regular rate defined by the *sampling rate.* The Nyquist-Shannon sampling theorem stipulates that the sampling rate must be at least double the highest frequency contained in a band-limited continuous-time signal. A failure to adhere condition can lead to *aliasing*, where high frequency content appears incorrectly as lower frequency content in the digital signal. Aliasing can be avoided by band-limiting the analogue signal using an *anti-aliasing* filter.

There are many different types of ADCs e.g. flash, sigma-delta, successive approximation, pipelined. Each design has its own strengths and weaknesses in terms of key parameters such as speed, complexity, resolution, linearity etc. The type of ADC design chosen in any give applications will depend on the requirements of that application.

Commercial ADCs generally come as dedicated ICs and may require other external components such as voltage references to operate. ADCs are also often found integrated into MCUs to allow the conversion of analogue signals from transducers in embedded systems applications. A transducer is a device that converts one form of physical quantity to another. Some examples of common transducers are microphones (sound), temperature sensors, light-level sensors, potentiometers (linear/rotary position).

9.3.2 PIC16F882 ADC module

The PIC16F882 has a single on board ADC module which has a resolution of 10 bits. This means that it can convert an analogue voltage into one of $2^{10} = 1024$ separate digital values ranging from 0-1023. In front of the ADC is an 16-channel multiplexer (MUX) which allows the selection of any one of 11 channels (AN5-7 not implemented in 28 pin variant of PIC16F882) connected to external pins on the PIC16F882. The MUX allows for several separate analogue inputs to be converted using a single ADC unit, although only one input can be converted at any one time. This means that the maximum possible sampling rate is reduced in proportion to the number of multiplexed channels converted e.g. sampling 4 channels will reduce the maximum sampling rate by a factor of 4.

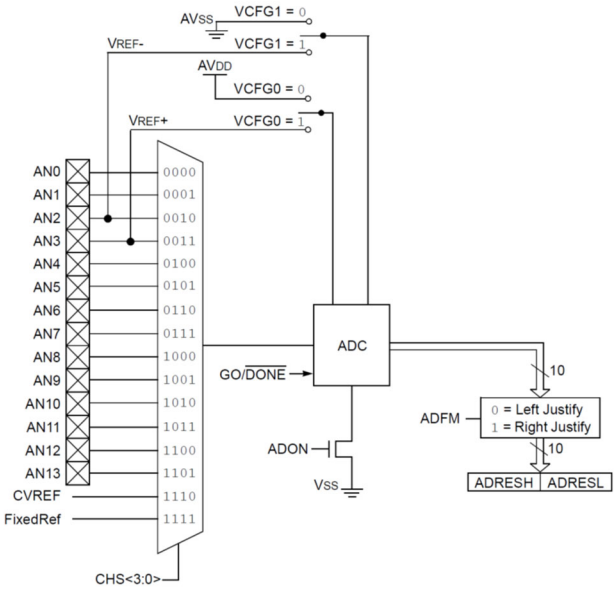

Figure 9-1 Schematic of the PIC16F882 ADC module

The ADC can supply its own voltage reference internally, or if required external voltage references can be supplied. The reference voltages (*VREF+* and *VREF-*) are important because they define the voltage conversion range of the ADC. Generally speaking the voltage resolution of an ADC is defined by,

$$\text{ADC Resolution} = \frac{V_{REF+} - V_{REF-}}{2^N}$$

We will be using the on board ADC module in conjunction with internally generated voltage references, which are sourced from the power supply rails. The internal voltage references are thus,

$$V_{REF+} = V_{DD} = 5\text{ V}$$
$$V_{REF-} = V_{SS} = 0\text{ V}$$

So the ADC voltage resolution available is,

$$\text{ADC Resolution} = \frac{V_{REF+} - V_{REF-}}{2^N} = \frac{5-0}{2^{10}} = 4.88\text{ mV}$$

In general, it is important to match the conversion range of an ADC with the anticipated operating range of the analogue signal you are monitoring. If the range is too small you will experience clipping

of the converted digital signal as the analogue voltage goes out of range. Conversely, if the range is too large you will reduce the effective resolution at which the ADC is operating.

It is worth noting that other parameters can affect the effective resolution of a real-world ADC including linearity, missing codes and non-monotonicity. However, such detailed additional considerations are outside of the scope of this laboratory practical.

9.3.3 ADC conversion clock requirements

The PIC16F882 ADC module works on the principle of *successive-approximation.* This means that each ADC conversion requires a number of conversion cycles to arrive at the final converted digital value. In total 11 conversion cycles are required (one for each bit of resolution plus one for overhead). The minimum conversion cycle time (T_{AD}) is 1.6 µs if the ADC is to achieve its specified accuracy. The conversion cycle time is determined by the *ADC conversion clock.*

The ADC conversion clock can be supplied internally from the crystal oscillator driving the MCU via a prescaler or via a dedicated RC oscillator (for low power applications). There are 4 software selectable options in total:

- $F_{OSC}/2$
- $F_{OSC}/8$
- $F_{OSC}/32$
- F_{RC} dedicated internal oscillator)

Because we are using a 4 MHz crystal to drive the PIC16F882, in order to maintain the criteria for a 1.6 µs conversion clock interval then we need to select the conversion clock source as $F_{OSC}/8$ which yields a conversion clock interval, $T_{AD} = 2\ \mu s$. The total conversion time per sample is then $11T_{AD} = 22\ \mu s$.

9.3.4 ADC acquisition time requirements

The analogue voltage input to the ADC module is sampled before conversion by a sample-and-hold circuit, which comprises a capacitor and switch. It is necessary to allow enough time for the capacitor to charge and for the amplifier that buffers the signal to the ADC to settle if the sampled analogue voltage is to be accurate enough to provide true 10 bit resolution on conversion. This so called *acquisition time* (T_{ACQ}) also depends on ambient temperature and the source impedance of the analogue input being sampled. Detailed information on the calculation of acquisition time can be found in the PIC16F882 datasheet. A worst case value of T_{ACQ} = 5 µs is assumed, which will ensure operation of the ADC to the specified accuracy when the ADC channel is switched.

9.3.5 Configuring the PIC16F882 ADC module

The ADC module is configured through two SFRs, *ADCON1* and *ADCON2.* These registers contain bitfields to do the following,

- Set the ADC conversion clock source.
- Select which MUX channel to convert.
- Start an ADC conversion.
- Turn the ADC module on/off.

- Set the justification of the conversion result stored in the in the *ADCRESH* and *ADCRESL* registers.
- Selection of internal/external voltage reference sources.

The contents of the *ADCON0* and *ADCON1* SFRs are summarised in figure 9-2 and figure 9-3. This looks quite complicated at first, but luckily the ADC should be configured in a fixed manner for correct operation in these laboratory practicals. Only the MUX channel selection bitfield (*CHS2:0*) needs to be changed for your application.

R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0
ADCS1	ADCS0	CHS3	CHS2	CHS1	CHS0	GO/$\overline{\text{DONE}}$	ADON
bit 7							bit 0

Legend:			
R = Readable bit	W = Writable bit	U = Unimplemented bit, read as '0'	
-n = Value at POR	'1' = Bit is set	'0' = Bit is cleared	x = Bit is unknown

bit 7-6 **ADCS<1:0>:** A/D Conversion Clock Select bits
00 = Fosc/2
01 = Fosc/8
10 = Fosc/32
11 = FRC (clock derived from a dedicated internal oscillator = 500 kHz max)

bit 5-2 **CHS<3:0>: Analog Channel Select bits**
0000 = AN0
0001 = AN1
0010 = AN2
0011 = AN3
0100 = AN4
0101 = AN5
0110 = AN6
0111 = AN7
1000 = AN8
1001 = AN9
1010 = AN10
1011 = AN11
1100 = AN12
1101 = AN13
1110 = CVREF
1111 = Fixed Ref (0.6V fixed voltage reference)

bit 1 **GO/$\overline{\text{DONE}}$:** A/D Conversion Status bit
1 = A/D conversion cycle in progress. Setting this bit starts an A/D conversion cycle.
This bit is automatically cleared by hardware when the A/D conversion has completed.
0 = A/D conversion completed/not in progress

bit 0 **ADON:** ADC Enable bit
1 = ADC is enabled
0 = ADC is disabled and consumes no operating current

Figure 9-2 ADCON0 SFR

REGISTER 9-2: ADCON1: A/D CONTROL REGISTER 1

R/W-0	U-0	R/W-0	R/W-0	U-0	U-0	U-0	U-0
ADFM	—	VCFG1	VCFG0	—	—	—	—
bit 7							bit 0

Legend:			
R = Readable bit	W = Writable bit	U = Unimplemented bit, read as '0'	
-n = Value at POR	'1' = Bit is set	'0' = Bit is cleared	x = Bit is unknown

bit 7 **ADFM:** A/D Conversion Result Format Select bit
1 = Right justified
0 = Left justified

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

bit 5 **VCFG1:** Voltage Reference bit
1 = VREF- pin
0 = VSS

bit 4 **VCFG0:** Voltage Reference bit
1 = VREF+ pin
0 = VDD

bit 3-0 **Unimplemented:** Read as '0'

Figure 9-3 ADCON1 SFR

Table 9-1 summarises the configuration requirements for the ADC.

Table 9-1 Summary of ADC configuration requirements

Bit field	Value	Reason
ADCS1:0	1	Need to set the conversion clock rate to 8 F_{OSC}/8 which gives T_{AD}=2 µs when MCU is clocked at 4 MHz. This ensures T_{AD}>1.6 µs (minimum requirement)
CHS3:0	9	Set AN9 as the default MUX channel for conversion. This is connected to the POT on the tutorial board.
ADON	1	The ADC module should be turned on ready for use
ADFM	1	Set results justification to right-justified. This allows easy combination of *ADRESH* and *ADRESL* to yield a 10 bit result.
VCFG1:0	0	Use only internal voltage references: V_{REF-} = 0 V, V_{REF+} = +5V

The required configuration for ADCON0 and ADCON1 is summarised in the tables below (X represents a 'don't care' state).

Table 9-2 ADCON0 Configuration

ADCS1	ADCS0	CHS3	CHS2	CHS1	CHS0	GO/DONE	ADON
0	1	1	0	0	1	X	1

Table 9-3 ADCON1 Configuration

ADFM	-	VCFG1	VCFG0	-	-	-	-
1	X	0	0	X	X	X	X

This configuration can be achieved by writing the following values to the ADC configuration SFRs (don't care' states are written as zeroes in this instance).

```
ADCON0 = 0x64;
ADCON1 = 0x80;
```

9.3.6 Starting an ADC conversion

Once the ADC module has been configured correctly and turned on, starting an ADC conversion requires the following two actions,

- Wait the required minimum acquisition time (5 µs) has elapsed to allow the analogue voltage to stabilise in the sample and hold circuit.
- Set the Go/Done bit in the ADCON0 SFR to command the conversion process to start.

These actions can easily be implemented using the code below

```
__delay_us(5);              // Wait for acquisition time (worst case 5 us)

ADCON0bits.GO = 1;           // Set GO/DONE Bit to start conversion
```

When the conversion is complete then the Go/Done bit is automatically reset by the MCU. This condition can be checked by polling. The following code sets up a do-nothing while loop that polls the GO/DONE bit, and exits only when it has been reset.

```
while(ADCON0bits.GO==1);      // Wait for GO bit to clear (conversion complete)
```

When the result is ready it is stored in the ADC results SFRs, *ADRESH* and *ADRESL.*

9.3.7 Retrieval and storage of ADC results

It is necessary to use two SFRs to store the 10 bit result of a conversion by the ADC module. This is because of the maximum 8 bit register width set by the PIC16 family architecture. The ADC result registers are called *ADRESH* and *ADRESL.* Combining both of these 8 bit SFRs provides a 16 bit wide result register. The ADC result itself is only 10 bits wide so there is an option provided to set the justification of the result in the register.

Results can be stored in either a left-justified or right justified manner depending on the state of the *ADFM* bit in the *ADCON1* SFR. Regardless of the selected justification type, the unused bits are padded with zeroes.

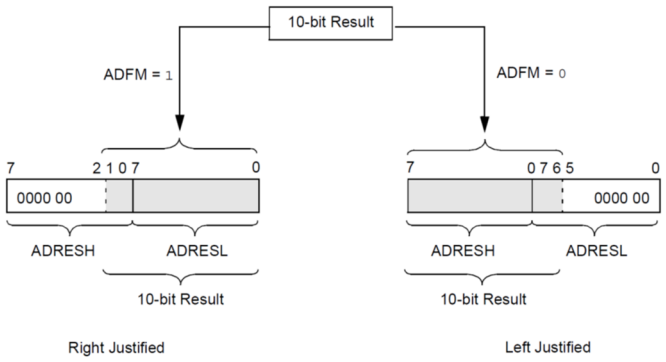

Figure 9-4 Justification of the ADC result

The simplest way of retrieving the result of the 10 bit ADC conversion is to configure right-justification (*ADFM=1*) and use a bitwise shift to combine the registers before assigning the result to a variable of type *unsigned int,*

```
result = (ADRESH<<8) + ADRESL;   // Combine to produce final 10 bit result
```

Alternatively, if the full resolution of the ADC is not important then it is possible use left-justification (*ADFM=0*) and simple retrieve the high byte as follows,

```
result =  ADRESH;              // Retreive 8 bit resolution result directly
```

This will provide an effective ADC resolution of 8 bits because the 2 LSBs of the result (stored in the ADRESL register) are ignored.

9.3.8 ADC conversion complete interrupt

There is a peripheral interrupt available that can be triggered whenever an ADC conversion is completed. You can interact with this by using the *ADIE/ADIF* bits in the *PIE1/PIR1* SFRs. Do not forget to enable peripheral interrupts by setting the *PEIE* bit in the *INTCON* SFR if you wish to use this facility. Using interrupts allows the MCU to get on with other useful tasks while an ADC conversion takes place. See the PIC16F882 datasheet for more information on using the ADC conversion complete interrupt.

9.4 Procedure

9.4.1 Exercise 1 – Configuring and using the ADC module

In this exercise we will create some basic ADC functions and investigate their operation, utilising the LCD to provide output.

3) Create a new project in the MPLAB X IDE, then create an empty source (.c) file called *ADClib882.c* and populate it with the code listing below.

```
// Filename:      ADClib882.c
// Version:       1.0
// Date:   <Insert current date>
// Author: <Insert your name>
// Purpose     : Provides functions to easily use PIC16F882 ADC
#include <xc.h>
#define _XTAL_FREQ 4000000  // Required for the __delay_us and __delay_ms macros

/* ADC_Config Configures the ADCON1 register for Fosc/8 and enables the ADC
 * function once before prior to using any of the LCD display functions
 * contained in this library. */

void ADC_initialise(void)
{
    TRISB  |= 0x18;   // Set RB3&4 as inputs as required for ADC lines
    ANSELH |= 0x0A;   // Set RB4 and RB4 as analogue inputs AN9 and AN11
    ADCON0 = 0x64;      // Set ADC to: Channel AN9, conversion clock = Fosc/8
    ADCON1 = 0x80;      // Set ADC to: Internal Vref, right justified results
    ADCON0bits.ADON = 1;    // Turn ADC On
}

/* ADC_Read reads the current analogue reading channel selected. It starts the
 * conversion by setting the Go/Done bit. Conversion is complete when the bit
 * is cleared by the MCU so a polling loop is set up detect this. After
 * conversion the ADRESH and ADCRESL are combined to provide a 10 bit result.
```

```
 */

unsigned int ADC_read(void)
{
    unsigned int result;
    __delay_us(5);               // Wait for acquisition time (worst case 5 us)
    ADCON0bits.GO = 1;            // Set GO Bit to start conversion

    while(ADCON0bits.GO==1);      // Wait for GO bit to clear=conversion complete

    result =(ADRESH<<8)+ADRESL;   // Combine to produce final 10 bit result
    return(result);
}

/* ADC_channel_select() selects the current channel for conversion.
* On the E-block sensor board:
* Channel 0 (AN0) is connected to the LDR.
* Channel 1 (AN1) is connected to the potentiometer.
*/

void ADC_channel_select(unsigned char channel)
{
    ADCON0bits.CHS=channel;       // Select ADC analogue input (AN9 or AN11)
}
```

4) Next add a new header file called *ADClib882.h* to the project and and populate it with the following code listing:

```
// Filename:     ADClib882.h
// Version:      1.0
// Date:   <Insert current date>
// Author: <Insert your name>
// Purpose: Provides functions to easily use PIC16F882 ADC

#define POT 9               // Define channel for POT on tutorial board (AN9)
#define EXT 11              // Define channel for EXT ADC on tutorial board (AN11)

//Function Prototypes

void ADC_initialise(void);
unsigned int ADC_read(void);
void ADC_channel_select(unsigned char channel);
```

5) Study the code carefully and note the purpose of each of the functions. Pay attention the parameter and return data types. The *ADClib882.h* and *ADClib882.c* files form a library which provides you with 3 functions you can deploy in your projects to utilise the ADC.

6) In the next task we will use the new library to develop some simple ADC routines. To do this we need to create a main function that does something useful. Create a new source file called e.g. *Lab9Ex1.c* and populate it with the following code listing:

```
// Filename:     Lab9Ex1.c
// Version:      1.0
// Date:   <Insert current date>
// Author: <Insert your name>
// Purpose: Test ADC library

// PIC16F882 Configuration Bit Settings
```

```
#pragma config FOSC = INTRC_NOCLKOUT // Oscillator Selection bits
#pragma config WDTE = OFF       // Watchdog Timer Enable bit
#pragma config PWRTE = OFF      // Power-up Timer Enable bit
#pragma config MCLRE = ON       // RE3/MCLR pin function select bit
#pragma config CP = OFF         // Code Protection bit
#pragma config CPD = OFF        // Data Code Protection bit
#pragma config BOREN = OFF      // Brown Out Reset Selection bits
#pragma config IESO = OFF       // Internal External Switchover bit
#pragma config FCMEN = OFF      // Fail-Safe Clock Monitor Enabled bit
#pragma config LVP = OFF        // Low Voltage Programming Enable bit
#pragma config BOR4V = BOR40V   // Brown-out Reset Selection bit
#pragma config WRT = OFF        // Flash Program Memory Self Write Enable bits

#include <xc.h>             // Required by compiler, PIC specific definitions
#include "LCDdrive882.h"    // Header file needed to access to LCD custom library
#include "ADClib882.h"   // Header file for ADC library

#define _XTAL_FREQ 4000000  // MCU clock speed - required for delay macros

void main(void) {
    ANSEL = 0x00;   // Set all pins to digital IO mode
    ANSELH = 0x00;
    TRISA = 0x00;   // Set all PORTA to outputs (for LEDs)
    PORTA = 0x00;
    TRISC = 0x00;   // Set all PORTC to outputs (for LCD)
    PORTC = 0x00;

    LCD_initialise();
    ADC_initialise();

    while(1)
    {
        LCD_cursor(0,0);
        LCD_putsc("ADC: POT =      ");
        ADC_channel_select(9);
        LCD_cursor(11,0);
        LCD_display_value(ADC_read());
        __delay_ms(500);
    }
}
```

7) Write down a flow chart describing the operation of the main() function using the subroutine call box (see figure 8-5).

8) Write another flowchart describing the operation of the ADC_read() function.

9) MPLAB provides a useful feature, *Call Graph* for mapping the interconnectivity of functions in C code. This can be useful for generating hierarchy diagrams for software design reports. You can access the call graph by right clicking on a function in the editor and selecting *Show Call Graph* from the context menu.

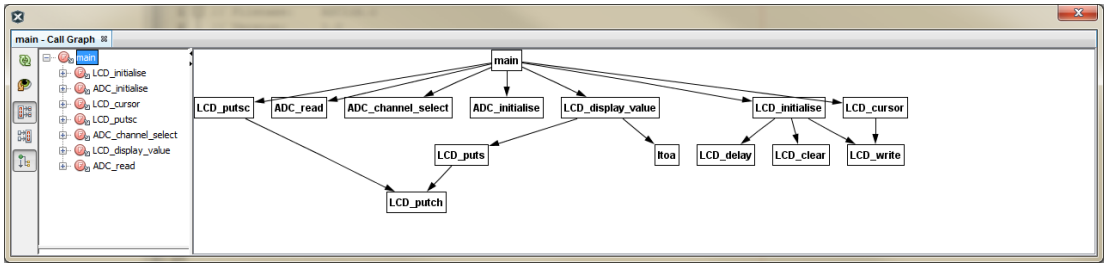

Figure 9-5 Call Graph in MPLAB X

Right click on the following line and select *Show Call* Graph, to produce a call graph of the main() function

```
void main(void)
```

You can right click on the boxes representing functions and select *Expand Callees* from the resulting context menu to see further levels of function calls. Boxes can also be dragged around to create a neat result.

Arrange the boxes in the Call Graph output window until you have a hierarchical structure similar to that shown in figure 9-5. Right click on whitespace in the window and select Export… to bring up a dialog window which allows you to save the result as a .png file.

10) Take a local copy of the two files *ACDlib882.h* and *ADClib882.c.* These files make up a custom library which can can deploy in your projects in the same manner as the LCD library. For the rest of the exercises in this section you should use the custom ADC library.

9.4.2 Exercise 2 - Calculating physical quantities from converted ADC values

This exercise requires you to deploy the custom ADC library in order to perform a simple ADC task. We will utilise our newly created ADC functions library to investigate the conversion of ADC values to represent actual physical quantities such as voltage.

1) Create a new project and incorporate the custom ADC library from the last exercise by adding copies of *ADClib882.h* and *ACDlib882.c*. You will also need to add a new source file and add the following directive to the top of the main code listing.

```
#include “ADClib882.h”
```

2) Devise a programme that will convert the voltage from the tutorial board potentiometer on ADC channel AN9 and display the digital value on the top line of the LCD at a rate of 1 Hz.

3) Expand the code so that the current potentiometer voltage in millivolts is displayed on the bottom line of the LCD display. The value should be appended with the correct units (mV).

The calculation of the ADC resolution in volts for one LSB is described in section 9.3.2. Multiply the digital value obtained from the ADC by the voltage resolution (in mV) to calculate the converted voltage. You will need to use a *float* data type to store the result or you may not get an accurate calculation result.

The reason for displaying the converted voltage in millivolts is that is avoids the need to display floating point values using the *LCD_display_float()* function which is costly in terms both computation cycles and memory.

You may view the correct operation for your developed code in this task by uploading the HEX file *ADC_Voltage.hex* (available on Brightspace).

9.4.3 Exercise 3 – Adding functionality to our ADC acquisitions

1) Create a new project in the MPLAB X IDE, then create an empty source file and populate it with the code listing below. **Note:** You will need to place a copy of the LCD and ADC library files in the project directory in order to build the project successfully.

   Create an empty source file and insert the code from the previous exercise. Expand your code solution so that the following functionality is implemented.

   - Display only the converted voltage in millivolts on the top line
   - Display the minimum and maximum voltage values converted since startup on the bottom line of the LCD.
   - Provide a reset function so that when SW0 is pressed, the *minimum* voltage value is reset to the current value. Write a high level flow chart representation of your ISR.
   - Provide a reset function so that when SW1 is pressed, the *maximum* voltage value is reset to the current value. Write a high level flow chart representation of your ISR.

   You should use the PORTB interrupt-on-change function in to implement functionality associated with buttons SW0 and SW1.

   You may view the correct operation for your developed code in this task by uploading the HEX file ADC_Voltage_MinMax.hex (available on Brightspace).