ECEN301 : Embedded Systems
Lab 2 Submission

Daniel Eisen : 300447549

August 12, 2020

1 Objectives

This lab demonstrated the use of the analogue to digital convertor onboard the 8051 as well as displaying data to a discrete LCD display. Using that as a base we covered analogue sensor input (in the form of a thermistor) as well as calibration, linearization and unit conversion. Side effects of this was encountering the limitations that an embedded engineer must overcome, such as small stack size, peripheral type limitations etc.

2 Methodology

2.1 Introduction

This lab requires the 8051, the LCD display (control to P4, data to P0), and breadboard module with a variable voltage divider module attached. As with the last lab, the programs were developed and compiled to a hex file with Atmel Studio and the included 8051 library and lab modules library. Initially LCD control was tested with basic IO, then familiarising ourselves with the ADC by designing a basic voltmeter, then finally, with more complexity a temperature measurement program was designed and calibrated.

2.2 LCD Display

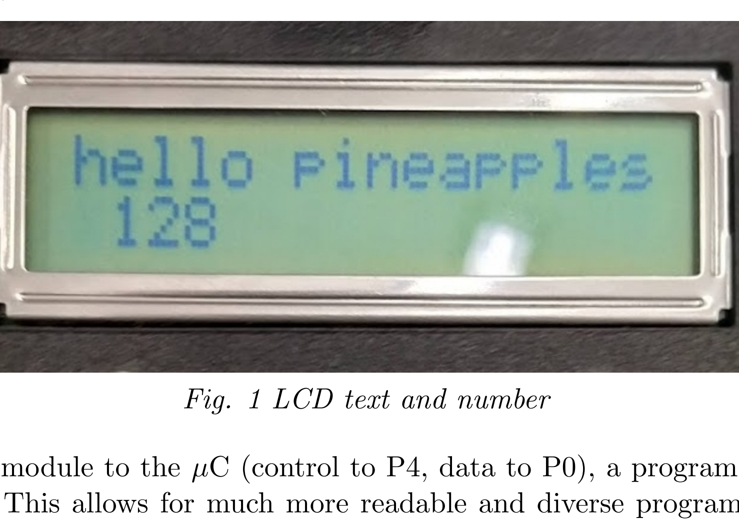


Fig. 1 LCD text and number

First connecting the module to the μC (control to P4, data to P0), a program can then be written to interface with it. This allows for much more readable and diverse program output that can be read with out via the 7-seg or LEDs on the IO module. To be able to write to the LCD panel, the ECEN301LibSDCC.h library is used to abstract the interfacing specifics away from the main control flow of the program.

To be able to write multiple data types to the LCD the std.io library can be used, specifically the `sprintf` function for formatting an char pointer variable to be printed later.

On the subject of this char pointer, this is a significant cause of stack memory usage so during the lab it was found that anything over a size of 32 was uncompilable so keeping to 16 was necessary for larger programs. Though a size 32 array was needed for a 2 lines display.

To prevent flashing a delay used to hold the current state within the loop.

```c
/*
    main.c

    Created : 7/28/2020 10:03:53 AM
    Author  : eisendani
*/

/*
Port0: LCD Data
Port1: PWM output (Same port as PORTADC)
Port2: Keypad or I/O Module
Port3: DAC
Port4: LCD Control
PortADC: Motor Encoder input - Thermister input

*/
#include "AT89C51AC3.h"
#include "ECEN301LibSDCC.h"
#include <stdio.h>

void main(void)
{
    char str[32];
    sprintf(str, "hello pineapples %d", 128);

    initLCD();
    writeLineLCD(str);

    unsigned int i = 0;
    while (1) {
        clearLCD();
        i++;
        sprintf(str, "%d", i);
        writeLineLCD(str);
        delay(10000);
    }
}
```

2.3 ADC

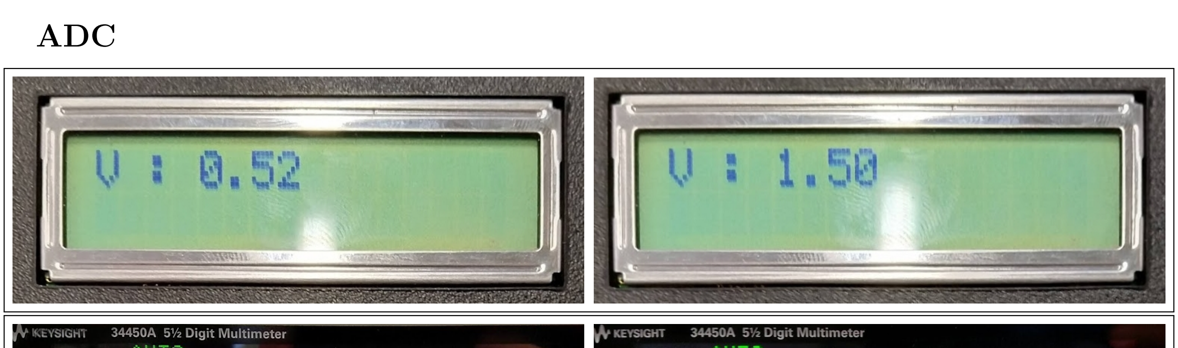
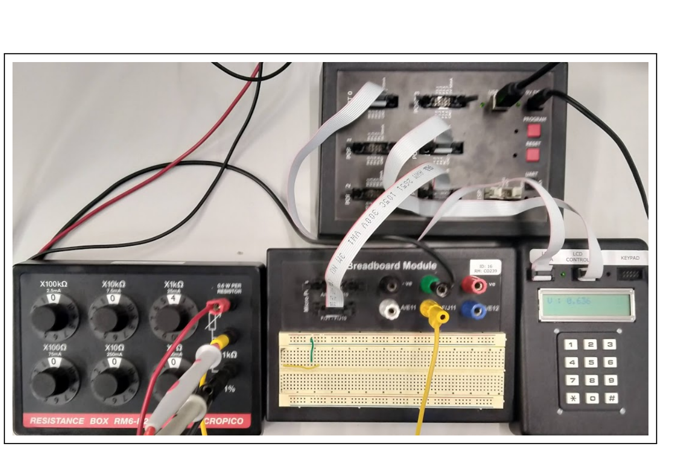


Fig. 2 Comparison between reading and "true" value

The main purpose of the analog converters is to convert conditioned analog signals into digital data so that the microcontroller system can process them for display, calculations, and analysis etc. To do this in the 8051, this is controlled via the ADCON and ADCF special function registers registers.



The setup for our use case is a voltage meter, utilising a voltage divider of the reference voltage as the analog input signal. This is seen above. The voltage is varied by changing R_{total} with the variable resistance module. To setup the ADC for take and convert and external input, ADCF is used to select input channel, then ADCON to enable ADC, set CPU into pseudo idle, and start the conversion. Pseudo Idle mode, is a reduces the CPU noise to a state where the last 2 bit of the 10bit ADC are usable. After this, the end of conversion flag must be reset in software, and the resulting conversion can be read from ADDH and ADDL.

The procedure of calibrating this digital reading to a displayable voltage value is as follows.

$$voltage = ADC_{sample} * V_{ref}/2^{10}$$

This can then be displayed to LCD, but due to being unable to writing floats to a string the reading must be split across the decimal place and displayed "separately".

```c
/*
    main.c

    Created : 7/28/2020 10:41:52 AM
    Author  : eisendani
*/
#include "AT89C51AC3.h"
#include "ECEN301LibSDCC.h"
#include <math.h>
#include <stdio.h>

/***  Function to sample an analog voltage  *******************************/

unsigned int Sample_ADC()
{
    static unsigned int sample = 0;
    ADCF = 0b00000001; //select input
    ADCON = 0b00000000; //clear control
    ADCON |= 0b01100000; //set psidle and enable ADC
    ADCON |= 0b00001000; //start

    ADCON &= 0b1101111; //bitmask clear the end of conversion flag
    sample = (ADDH << 2) + ADDL;
    return sample;
}

/**  Main Function  *****************************************************/

void main()
{
    char str[16];
    initLCD();
    while (1) {
        float reading = (Sample_ADC() * 3.3 / 1023.0);
        unsigned int upper = floorf(reading);
        unsigned int lower = floorf((reading - (float)upper) * 1000.0);
        sprintf(str, "V : %d.%d", upper, lower);
        writeLineLCD(str);
        ;
        delay(100000);
        clearLCD();
    }
}
```

2.4 Temperature

For a more complex and practical demonstration of the ADC, we implemented a Temperature sensor. Using a thermistor is as a temperature dependant resistance as an analog sensor input to the ADC. From then it can be calibrated with known temperatures and characterised.

Setup

Leaving the modules in same configuration as before, with the breadboard suppling the ADC input, with the same LCD output except in this case replacing the voltage divider the thermistor and matched resistance. At room temperature the nominal resistance was measured at 10k so this is what was used as the second resistance.

Calibration

The value of the thermistors resistance cannot be directly as temperature, so its it must be characterised and translated into a temperature.
Using a thermometer and range of temperatures I correlated the ADC value and the measure value. This is seen to be non linear so to make it more usable linearization is applied to spilt the curve into two regions and fitting a linear curve to them to get an equation to translate the ADC reading to a temperature

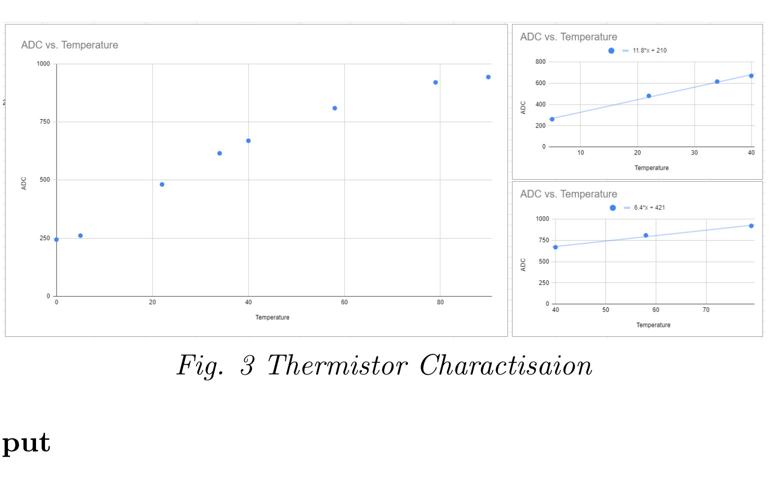
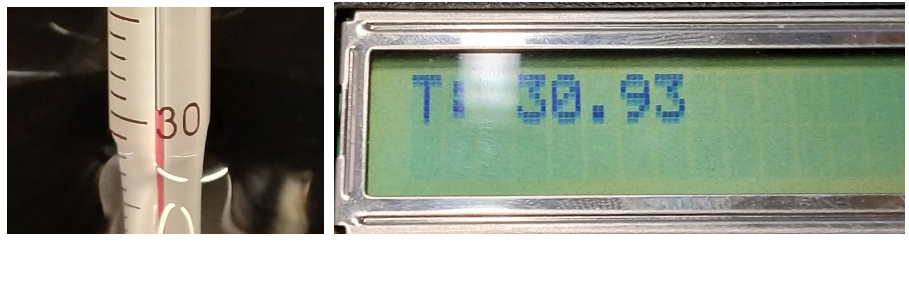


Fig. 3 Thermistor Charactisaion

Program and output



As seen above, the characterised equations when implemented into the program work to a reasonable accuracy with a normal operating range.
In the implemented program, there is a check to switch from one region to the other.
When extending the functionality, I encountered multiple issue regard stack memory space. Due to the lack of floating point support I found myself having to store more and more temp variables for each side of the decimal place. The sliding average worked well and is implemented but the necessary array didn't help with stack space. This is why I didn't get SD and min max working.
How I would implement it now would, instead of floats, work with scaled (x100) ints for the value then use a modulus method to split them. ie temp = 3042: temp/100 would be 30 and temp%100 would be 42 and print those either side of the decimal point.
This would enable a much better usage of stack space, clean up the readability and increase performance.

```c
/*
    main.c

    Created : 7/28/2020 10:41:52 AM
    Author  : eisendani
*/
#include "AT89C51AC3.h"
#include "ECEN301LibSDCC.h"
#include <math.h>
#include <stdio.h>

#define VREF 3.0

/***  Function to sample an analog voltage  *******************************/
char str[16];
unsigned int arrNumbers[5] = { 0 };
unsigned int sample, upper, lower;
//unsigned int upperS, lowerS;
unsigned int sum = 0;
unsigned short pos = 0;
//int min = 1000, max=-1000;
float SD;

unsigned int movingAvg(unsigned int* ptrArrNumbers, unsigned int* ptrSum, unsigned
    short pos, unsigned int len, unsigned int nextNum)
{
    //Subtract the oldest number from the prev sum, add the new number
    *ptrSum = *ptrSum - ptrArrNumbers[pos] + nextNum;
    //Assign the nextNum to the position in the array
    ptrArrNumbers[pos] = nextNum;
    //return the average
    return *ptrSum / len;
}

unsigned int Sample_ADC()
{
    static unsigned int sample = 0;
    ADCF = 0b0000001; //select input
    ADCON = 0b1101000; //clear control (0),set psidle and enable ADC(110...). Start
    (00010...)
    ADCON &= 0b1101111; //bitmask clear the end of conversion flag
    sample = (ADDH << 2) + ADDL;
    return sample;
}

/**  Main Function  *****************************************************/

void main()
{
    initLCD();
    while (1) {
        sample = Sample_ADC();
        sample = movingAvg(arrNumbers, &sum, pos, 5, sample);
        pos++;
        if (pos >= 5) {
            pos = 0;
        }

        if (sample >= 669) {
            upper = (unsigned int)((float)sample - 421.0) / 6.4;
            lower = ((((float)sample - 421.0) / 6.4) - (float)upper) * 100.0;
        } else {
            upper = (unsigned int)((float)sample - 210.0) / 11.8;
            lower = ((((float)sample - 210.0) / 11.8) - (float)upper) * 100.0;
        }

        //for (int i=0; i<5 ; ++i){
        //SD += powf(arrNumbers[i] - sample, 2.0);
        //}
        //SD = SD / 10.0;
        //
        //upperS = (unsigned int)SD;
        //lowerS = (unsigned int)(SD-upperS)*100.0;

        sprintf(str, "T: %d.%d", upper, lower);
        ;
        writeLineLCD(str);
        ;
        delay(100000);
        clearLCD();
    }
}
```

3 Questions

1. *Explain what all of the bits in the ADCON and ADCF registers mean?*

    - ADCF is the 'ADC Configuration' register. Setting one of the bits in this register enable the corresponding pin on Port 1 for the input to the ADC, (Clearing sets it back to standard IO).
    - ADCON is the control register for the ADC:
        - Bit 7 is reserved
        - Bit 6, set this to enable "pseudo idle mode." Basically putting the CPU into sort of sleep/low actively mode that allow the full 10bits of data from ADC to be used. This is due to the reduction in CPU noise.
        - Bit 5 enables/standbys the ADC. If code doesn't use the ADC clearing this can save power.
        - Bit 4 is a flag that is set when the ADC and finished doing a reading, this needs to be cleared by your code.
        - Bit 3 is used to start the conversion, this cleared afterward by the hardware.
        - Bit 2:0 are used to select between the 8 input channels