Building a Digital Voltmeter with Serial Communication

Matthew Beckler
beck0778@umn.edu
EE2361 Lab 007

April 18, 2006

Abstract

Using the PIC18F452's digital to analog converter (DAC), we have created a very simple digital voltmeter. It utilizes the built-in serial capabilities of the 18F452 to transmit the voltage to a host computer using standard RS232 serial communications. Multiple samples are taken of the input voltage, averaged, and transmitted to the computer. A MAX202 charge pump chip is used to boost standard TTL logic levels to serial port logic levels.

Introduction

This project combines two important devices, the analog to digital converter, and the serial port USART. A potentiometer is used to provide an easily variable voltage input. The potentiometer was connected between +5V and ground, meaning that possible output values are between 0 and 5V. The processor samples this input voltage 128 times, and averages the recorded values. The format for the output is one digit before the decimal point, and two digits after. The software calculates the correct value for each digit, and transmits the data to the computer at 9600 baud, using a standard serial cable. Using two simple function for all serial communications makes interfacing very easy from the main line of the program. We have not used any timers in this application, nor have we used any interrupts. We are using simple polled I/O to wait for each ADC conversion.

Observations

The overall program control for this application is very simple. The first task is to configure the devices' configuration registers. This involves selecting the baud rate of 9600, enabling 8-bit transmission with the USART, and initializing the ADC for $F_{osc}/64$ operation. A welcome message of "Digital Voltmeter" is then transmitted to the client. After that, the processor starts a loop of 128 samples of the analog input, summing their values into a variable. It is important to note that before each analog to digital conversion is started, the processor must wait for at least $13.8\mu s$ for the aquisition time. After 128 samples have been

added, the processor averages and scales the variable into a number between 0 and 500. This allows for an easy translation into the three digits that are needed for the output.

The `putch` function is used to put a charater to the serial port. It waits until the transmit buffer is not filled, and can be written to. This buffer is double buffered, meaning it can be written to twice before filling. Nevertheless, it is a good idea to wait until the buffere isn't filled before writing another byte. The `putch` function encapsulates this very nicely. The `delay` function provides at $13.8\mu s$ delay for the ADC aquisition time. It is interesting to not that the while loop contained inside the `delay` function only runs 3 times to produce a delay of $13.8\mu s$. This gives a glimpse into the MCC18's compiler in-efficiencies, because those three loops use approximately 1380 instructions.

For translating the average, scaled voltage into three digits for serial transmission, a bit of tricky programming is used. Remember, the output voltage is a number between 0 and 500, so we only have to split this number into the three digits. The least significant digit can be found by taking the voltage modulo 10. The modulo operation find the remainder when the voltage is divided by 10, leaving the value of the least significant digit. To find the next digit, we first must use integer division to divide the voltage by 10. Since integer division truncates the decimal part of the quotient, we are left with the proper value. For example, if we start with a voltage value of 427, using integer division, 427 / 10 is truncated to 42. If we perform the modulo 10 operation on this result, we get the second digit of the final answer. The same process can be repeated for the most significant digit, by dividing by 100 instead of dividing by 10. These three digits are stored into three variables, `d0, d1, d2`, and are sent out the serial port, with a decimal point after the first digit. After each value is transmitted to the computer, the processor sends a carriage return, ASCII character `0x0D`, which resets the computer terminal's cursor to the start of the current line, ready to overwrite the transmitted voltage. Since this happens very quickly, at a rate to be calculated in the next section, a human will not notice a delay. In fact, the values are provided much more quickely than the computer's monitor can update itself, so the processor's sample and transmission rate is not a source of delay.

When considering the sampling rate of the digital voltmeter, a number of factors come into play. The analog to digital conversion takes a certain amount of time, composed of two parts, the *aquisition* time and the *successive approximation* time. The aquisition time is a constant $13.8\mu s$ per sampling. The ADC uses a *sample and hold* scheme for the conversion, and there are capacitors involved which must charge to the proper levels before an accurate sample can be taken. The time for the successive approximation is based on the basic idea of the processor's ADC method. The PIC has a seperate clock for the ADC, called T_{AD}, which we have set to $F_{osc}/64$ for this project. The term, *Successive approximation*, means that for each T_{AD} clock, one bit of the final value is found, starting with the most significant. Since the ADC in our PIC has 10-bit resolution, it takes, at the least, 10 T_{AD} clocks to complete the conversion. The actual value is 12 T_{AD} clocks. The period of the 40 MHz F_{osc} is 25ns. The period of $T_{AD} = Fosc/64 = 625KHz$ is $1.6\mu s$. The sampling rate for each 10-bit analog to digital conversion takes:

$$13.8\mu s + 12 \cdot 1.6\mu s = 33\mu s$$

For all 128 samples per transmission, the processor needs:

$$33\mu s \cdot 128 = 4.2ms$$

When this time period is compared to the period of the USART's 9600 baud transmission, you notice that the time period required for the 128 ADC samples is much longer than the time period for one transmission. The ADC's sampling is completed in 4.2ms, while the time necessary to transmit five bytes, (three digits, one decimal point, and a carriage return), is:

$$5 \cdot \left(\frac{1}{9600}s\right) = 5 \cdot 104.167\mu s = 520.833\mu s$$

You could transmit the data over 8 times in the time it takes to complete all 128 ADC samples.

An analog input of 2.3 volts would be converted into a 10-bit binary number according to the following formula:

$$V = \frac{B}{2^{10}} \cdot 5 = \frac{5 \cdot B}{1024} \Rightarrow B = \frac{V}{5} \cdot 1024$$

Where V is the analog voltage input, and B is the 10-bit binary number received as output. For the input voltage of 2.3 volts:

$$B = \frac{V}{5} \cdot 1024 = \frac{2.3}{5} \cdot 1024 = 471$$

The resolution of the 10-bid ADC is simply the smallest voltage difference between adjacent binary numbers. For this setup, the basic resolution is:

$$\frac{1}{2^{10}} = \frac{1}{1024} = 0.000977 = 977\mu s$$

However, this number is the resolution of the ADC when measuring voltages between 0 volts and 1 volt. For this project, we need to scale the resolution:

$$Resolution = 0.00977 * 5Volts = 0.004833 = 4833\mu s$$

Conclusion

Altogether, this project was very much a success. We have successfully interfaced a PIC processor with a standard x86 computer through the standard serial port. We have refined our ADC interfacing skills, and practiced working with decimal numbers without the use of the floating point number package on the PIC. The voltages measured and transmitted to the computer are accurate to approximately one part in 200, and are generated at a quicker rate than the computer, or a human, can use.

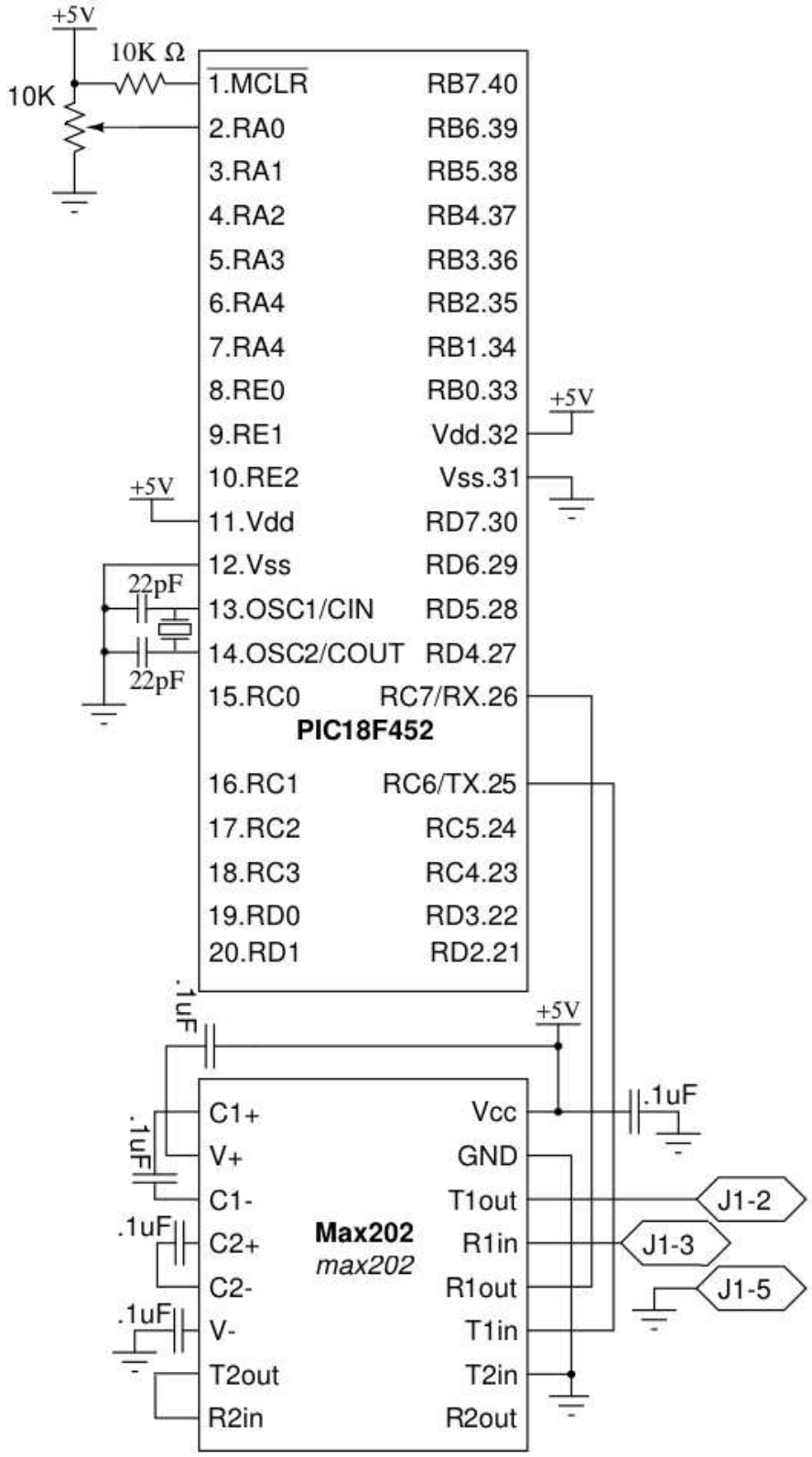

Figure 1: Circuit Schematic - Courtesy Professor Rennolet

C Language Program

```
#include <p18f452.h>
#pragma config OSC=HSPLL, WDT=OFF, BOR=OFF, PWRT=ON

void putch(char byte)
{
    while (!PIR1bits.TXIF);
    TXREG = byte;
}

void delay(void)
{
    int x = 3;
    while(x--);
}

void main(void)
{
    char welcomeMessage[] = "Digital Voltmeter";
    unsigned long int i, average, d0, d1, d2;
    unsigned long int values;
    unsigned long int scaleFactor = 130944; //130944 = 1023*128

    //initialize serial registers
    TXSTAbits.TXEN = 1;
    RCSTAbits.SPEN = 1;
    RCSTAbits.CREN = 1;
    SPBRG = 0x40;

    //initialize the ADC registers
    ADCON0 = 0b10000001;
    ADCON1 = 0b11000000;

    for (i = 0; welcomeMessage[i] != '\0'; i++)
    {
        putch(welcomeMessage[i]);
    }
    putch(0x0D);
    putch(0x0A);

    while(1)
    {
        values = 0;
        average = 0;
        for (i = 0; i < 128; i++)
        {
            delay(); //wait 13.8 us for acquisition
            ADCON0bits.GO = 1; //start AD conversion
            while (ADCON0bits.GO); //wait for AD to finish
```

```
            values += ADRES;
        }

        average = (values * 500) / (scaleFactor);

        d0 = average \% 10;
        d1 = (average / 10) \% 10;
        d2 = (average / 100) \% 10;
        putch('0' + d2);
        putch('.');
        putch('0' + d1);
        putch('0' + d0);

        putch(0x0D);
    }
}
```