Analog Output with a Digital to Analog Converter

Matthew Beckler
beck0778@umn.edu
EE2361 Lab 007

April 5, 2006

Abstract

Without help, microcontrollers can have great trouble creating analog signals. Approximations using rectifiers and pulse-width-modulation schemes work to a degree, but the best method of creating an analog signal is to use a Digital-to-Analog converter. These relatively simple, but sometimes expensive, devices work very nicely for this task. In this lab, we create a rudimentary triangle-wave generator with a PIC microprocessor and a 12-bit DAC.

Introduction

To create a triangle waveform, the signal's voltage should increase linearly from the minimum voltage to the maximum voltage, then linearly decrease back down to the minimum voltage. The process then repeats. We can approximate a triangle wave by updating a DAC device's voltage at a regular time interval, using the smallest voltage increment the DAC supports. We have set up a dedicated timer to generate regular interupts, where we either increment or decrement two output data registers. These two registers are connected to the twelve input pins of our DAC. When the maximum or minimum voltage has been reached, the program switches direction.

Observations

The progamming for this project is relatively simple compared to other projects. Timer two has been set up for operation with no prescaler and no postscaler. The period register `PR2` has been set to 243, which produces interrupts every 24.4 microseconds. The interrupt handling routines have returned from previous labs, and we are again using only the high priority interrupt handler. Two global variables have been created, *value*, and *direction*. The current number being sent to the DAC is stored in *value*, while the current slope ($\pm$ 1) is stored in *direction*. Each time the processor runs the interrupt handler, the value of *direction* is added to the current *value*. The processor then checks if the current value has reached either end of the continum, at either `4095` or `0`. If one of these conditions is detected, *direction* is multipled by negative 1, which flips it

between +1 and -1. The values of `PORTC` and `PORTD` are updated to reflect the new current *value*, and the DAC's write pin is toggled, signaling the DAC to update its internal data latches.

For accuracy calculations, we will consider both the voltage and time errors. Using reference voltages of 0V and +5V, there are 4096 voltage increments between those two voltages, since this is a 12-bit DAC, and 2^12 is 4960. This produces a voltage resolution of:

$$\frac{(5-0)}{2^{12}} = 0.001220703125 \frac{Volts}{Division}$$

This means that the output voltage should be the calculated voltage plus or minus this resolution factor. We can write formulas for translating between a binary value (N) and an analog voltage (V):

$$V = \frac{5}{2^{12}} * N \pm \frac{1}{2} * \frac{5}{2^{12}} = \left(N \pm \frac{1}{2}\right) * \frac{5}{2^{12}}$$

For calculating the accuracy of the signal frequency, we need to look at the accuracy of the timing interrupts. With the Timer 2 Period Register set to 243, Timer 2 will generate an interrupt once every $244 * 100ns = 24.4\mu s$. There are 4096 interrupts needed for one half of a period, so 8192 interrupts per waveform period. The frequency of the waveform, from trough to trough, is $8192 * 24.4\mu s = 0.1998848$ seconds. This equates to a period of 5.0028816 Hz. The waveform generator would need to run for at least 4736 periods, which corresponds to 347 seconds (5:47), before the waveform would be one period off from the 'correct' waveform.

Conclusion

Altogether, this project was a success. The osciloscope trace appeared to be perfect, but as we have calculated, it is not quite perfect. The waveform gains one period once every 5:47, which is not good enough for most projects, but as a proof-of-concept project, it works quite well. Learning to interface with a high quality DAC is a useful skill to have for future projects. Using the parallel IO in combination with a control bit will be useful for other types of devices, such as an LCD display or parallel IO controller. Continuing to develop our skills with the timers, we have devised a very accurate usage of Timer 2, using the automatically resetting period register.

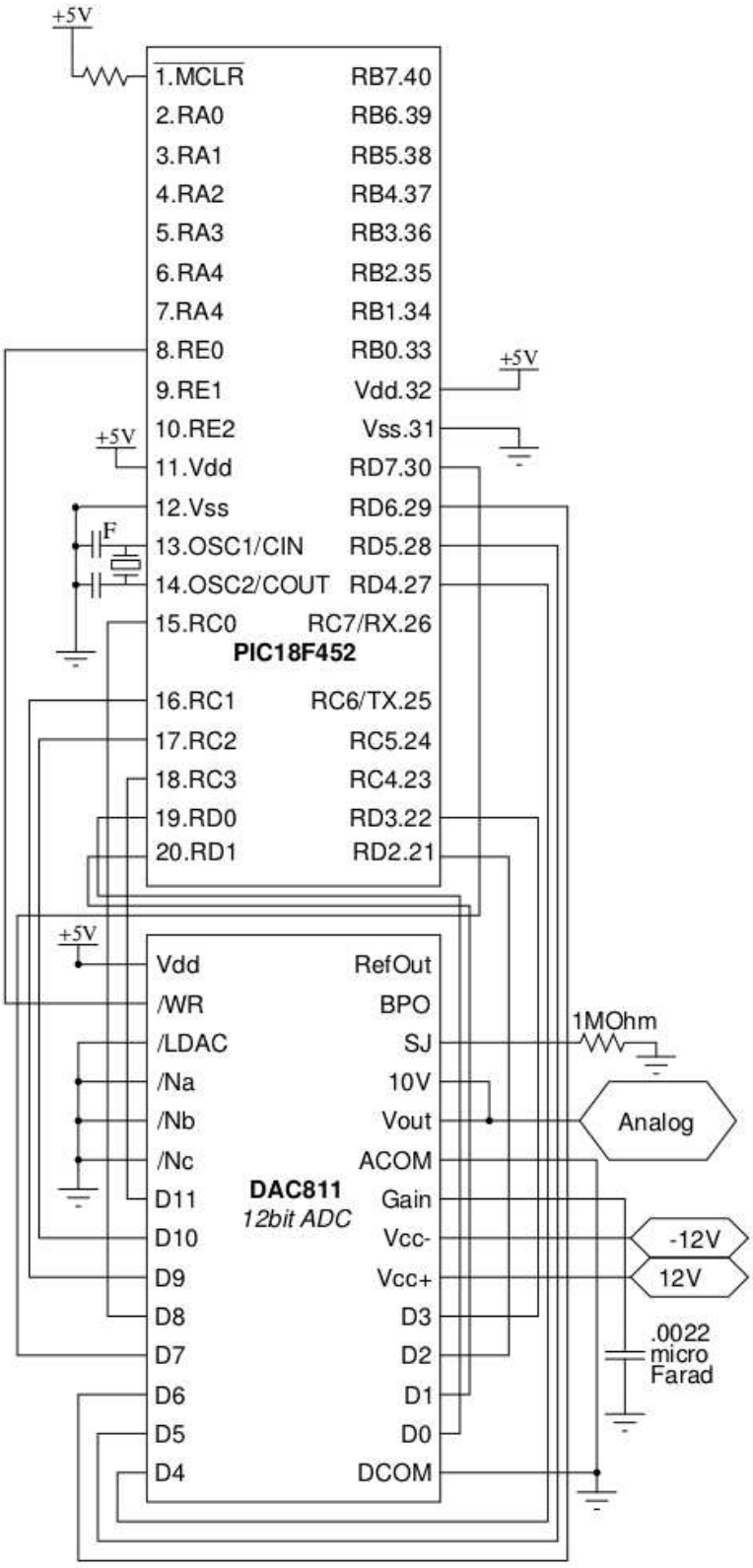

Figure 1: Circuit Schematic - Courtesy Professor Rennolet

C Language Program

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

void low_isr(void);
void high_isr(void);

static volatile int value = 0;
//value is the current output
//and is between 0 and 4095

static volatile int direction = 1;
//1 is increasing, -1 is decreasing

#pragma code high_isr_entry=0x8
void high_isr_entry(void)
{
    _asm GOTO high_isr _endasm
}

#pragma code low_isr_entry=0x18
void low_isr_entry(void)
{
    _asm GOTO low_isr _endasm
}

#pragma code

#pragma interrupt high_isr
void high_isr(void)
{
    PIR1bits.TMR2IF = 0;
    value += direction;
    if ((value == 4095) || (value == 0))
    {
        direction *= -1;
    }

    //update PORTC and PORTD
    PORTD = value & 0xFF;
    PORTC = value >> 8;

    PORTEbits.RE0 = 0;
    //wait at least 200ns
    //for the DAC to update
    _asm nop _endasm
    _asm nop _endasm
    PORTEbits.RE0 = 1;
}
```

```
#pragma interrupt low_isr
void low_isr(void){}


void main(void)
{
    TRISC = 0; //RC0-RC3 are DB8-DB11
    TRISD = 0; //RD0-RD7 are DB0-DB7
    TRISE = 0; //RE0 is the not-enable bit
    PORTC = 0;
    PORTD = 0;
    PORTEbits.RE0 = 1;

    INTCONbits.GIE = 1;   //enable global interrupts
    INTCONbits.PEIE = 1;  //enable peripheral interrupts
    PIE1bits.TMR2IE = 1;  //enable Timer2 interrupt
    PIR1bits.TMR2IF = 0;  //clear Timer2 interrupt flag
    T2CON = 0b00000100;   //Timer2: No pre- or post-scaler
    PR2 = 243;            //Timer2 set to 24.4 microseconds

    while(1);
}
```