Abstract:
This application note will cover how to use a digital to analog convertor in symphony with a PIC micro processor: more specifically how to communicate between the two devices without using a pre-existing serial transfer protocol, such as SPI, UART, or I2C, as well as code for easy use as a function generating system using c code. The devices used in this note will be the dsPIC30f2020 microprocessor and the AD7303 serial input 8-bit digital to analog convertor.

Keywords:
dsPIC30f2020, AD7303, digital to analog convertor, function generation, serial communication
**Introduction:**
The dsPIC30f2020 is a powerful microprocessor with many signal processing functions built in. However it does not have the capability of digital to analog conversion, so an external chip is necessary for this function. One such chip is the AD7303 DAC. This chip is an 8 bit convertor meaning that it has 256 ($2^8$) distinct steps between vdd and ground. This device also has two analog outputs meaning that is essentially two DAC’s on a single chip controlled by the same serial connection but in this application note only one will be used.

**Objective:**
This application note will go through the set up and corresponding code needed to implement a digital to analog convertor with PIC microprocessor. The goal of the code is to make it easy produce a voltage waveform out of the DAC. This code will also not rely on the use of a serial protocol such as SPI, UART, or I2C as the ports necessary for these protocols may already be used by other communications in the system.

**Implementation:**
First the microprocessor and the DAC need to be wired together and powered. This can be done as shown in Fig. 1.

![Fig. 1](image)

This configuration in Fig. 1 has Vdd = 5V and has connected port b bit 5 to SYNC’, port e bit 7 to DIN, and port b bit 7 to SCLK. The SYNC’, DIN, and SCLK pins of the DAC are what control the serial communication in to the DAC.

The DIN is the actual data that is read in to the DAC. This data is made up of eight control bits and eight data bits. The data bits are just what they sound like they contain the voltage step information so that the desired output is achieved. The control bits tell the device to power on or off the individual DACs as well as which DAC the data is for along with information on where the DAC should load data from, the shift register or the input register. The use of the input register is so that both DACA and DACB can be
changed at the same time. A table of the CR control bits can be seen below (Fig. 2). The control bits also control the use of the internal reference or the external reference. The external reference can be used to change the range of the analog output. The internal reference is set to half of Vdd where as the external reference can be set anywhere from half of Vdd to 1 volt. So setting the external reference can be used to decrease the output range but increase the resolution (step size in volts) of the DAC.

<table>
<thead>
<tr>
<th>CONTROL BITS</th>
<th>CR0</th>
<th>CR1</th>
<th>CR0/CR1</th>
<th>Function Implemented</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>X</td>
<td>0</td>
<td>0</td>
<td>Both DAC registers loaded from shift register.</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>Update DAC A input register from shift register.</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>Update DAC B input register from shift register.</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>Update DAC A DAC register from input register.</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>Update DAC B DAC register from input register.</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>Update DAC A DAC register from shift register.</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>Update DAC B DAC register from shift register.</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>X</td>
<td>X</td>
<td>Load DAC A input register from shift register and update both DAC A and DAC B DAC registers.</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>X</td>
<td>X</td>
<td>Load DAC B input register from shift register and update both DAC A and DAC B DAC registers outputs.</td>
</tr>
</tbody>
</table>

Fig. 2

SCLK is the clock signal that is common between the microprocessor’s data transfer rate and the DAC. This signal is used to latch the data into the shift register of the DAC. The AD7303 latches the data on the rising edge of the clock meaning that whatever value that is present on the DIN pin when the SCLK goes high is saved into the shift register.

SYNC’ is the pin that lets the device know when the data on DIN is to be saved in the shift register. This pin is active low which means that it should be kept high until the data on DIN is wanted to be latched into the shift register. This allows multiple devices to share the save data line from the microprocessor.

Now that the devices are connected and powered, coding can begin. First a function is made that needs to be passed a value corresponding to a desired output voltage. That will make the creation of signal generation easier later. This function first needs to set the sclk and the sync’ low this will signal the DAC that data is coming. This is done using the following lines of code, where wait is a function that waits for a specified number of cycles of the microprocessor’s clock before executing the next instruction.

```c
PORTBbits.RB7 = 0; //set clock low
wait(1);
PORTBbits.RB5=0; //Set sync low
```

The reason for the wait is so that the port has time to execute to the change; otherwise the next port bit change may be ignored due to the first instruction not being completed before the second instruction is executed.

The DAC is now ready for its first control bit to be transmitted.

```c
wait(1);
PORTEbits.RE7=0;
```
Again a wait sequence is needed to insure the port change. Next the SCLK needs to be set high in order to latch the data the DAC’s input shift register.

\[ \text{wait}(1); \]
\[ \text{PORTBbits.RB7} = 1; // \text{set clock high} \]

This is the process that will be used for the remainder of the control and data bits including the wait sequence between each changing port bit while leaving the SYNC’ low. If the SYNC’ is put low before the communication is complete the device will ignore all the data that has already been latched. Similarly if SYNC’ is left low to long and to much data is latched the device will also ignore all of the previously latched data. The control bit values used here are as follows: 0,0,0,1,0,0,1,1. This configuration tells the device to use the internal reference, power down DAC B, and the data is to be loaded into the DAC A register from the shift register. After the data bits have been transmitted the SYNC’ need to be set back high to close the communication and change the analog output.

Before sending the data bits the floating value that was passed to the function needs to be converted into binary. This can be done using the following code.

```c
int i,c,modval=1;
int bin[8];
double set;
set=(255*v)/5;
for(i=0;i<8;i++)
{
    modval = 1;
    for(c=7;c>i;c--)
    {
        modval=2*modval;
        if(set/modval>=1)
        {
            bin[i] = 1;
            set=set-modval;
        }
        else
        {
            bin[i] = 0;
        }
    }
}
```

The final code for controlling the DAC including all of the necessary microprocessor setup and initialization of the ports and internal oscillator can be found in the appendix of the application note.
Conclusion:
This application note has presented a solution to digital to analog conversion using the dsPIC30f2020 and the AD7303 digital to analog convertor. The resulting code also made it easy to implement voltage waveforms as the c function used for controlling the DAC needs only a voltage input in the form of a float value. This input could be based on mathematical functions, such as sine, cosine, or even a line whose slope changes from positive to negative based on predetermined time period resulting in a triangle wave. This code was also beneficial in the fact that it did not require using a predefined serial port, which might already be being used for communications in the final system, such as communication with a pc and/or a serial LCD display.
Appendix:

```
#include "p30f2020.h"

_FOSC(CSW_FSCM_OFF & FRC_HI_RANGE & OSC2_IO); //Oscillator Configuration
_FOSCSEL(FRC_PLL); //Oscillator Selection
_FWDT(FWDTEN_OFF); //Turn off WatchDog Timer
_FGS(CODE_PROT_OFF); //Turn off code protect
_FPOR( PWRT_OFF ); //Turn off power up time

void wait(int x)
{
    int q;
    for(q=0;q<x;q++); //Delay
};

int DACA(float v)
{
    int i,c,modval=1;
    double set;
    set=(255*v)/5; //Convert voltage to step level
    int check = set;
    int bin[8]={1,1,1,1,1,1,1,1};
    int cont[8]={0,0,0,1,0,0,1,1};

    //Round converted voltage value to nearest step

    if((set-check)>.5)
    {
        set = check + 1;
    }
    else
    {
        set = check;
    }

    for(i=0;i<8;i++)
    {
        modval = 1;
        for(c=7;c>i;c--)
            modval=2*modval;
        if(set/modval>=1)
        {
            bin[i] = 1;
            set=set-modval;
        }
    }

```
else
    bin[i] = 0;
}
PORTBbits.RB7 = 0; //Set clock low
wait(1);
PORTBbits.RB5=0; //Set sync’ low
wait(1);
//set command and selection bits
PORTEbits.RE7=cont[0];
wait(1);
PORTBbits.RB7 = 1; //Set clock high
wait(1);
PORTBbits.RB7 = 0; //Set clock low
wait(1);
PORTEbits.RE7=cont[1];
wait(1);
PORTBbits.RB7 = 1; //Set clock high
wait(1);
PORTBbits.RB7 = 0; //Set clock low
wait(1);
PORTEbits.RE7=cont[2];
wait(1);
PORTBbits.RB7 = 1; //Set clock high
wait(1);
PORTBbits.RB7 = 0; //Set clock low
wait(1);
PORTEbits.RE7=cont[3];
wait(1);
PORTBbits.RB7 = 1; //Set clock high
wait(1);
PORTBbits.RB7 = 0; //Set clock low
wait(1);
PORTEbits.RE7=cont[4];
wait(1);
PORTBbits.RB7 = 1; //Set clock high
wait(1);
PORTBbits.RB7 = 0; //Set clock low
wait(1);
PORTEbits.RE7=cont[5];
wait(1);
PORTBbits.RB7 = 1; //Set clock high
wait(1);
PORTBbits.RB7 = 0; //Set clock low
wait(1);
PORTEbits.RE7=cont[6];
wait(1);
PORTBbits.RB7 = 1; //Set clock high
wait(1);
PORTBbits.RB7 = 0; //Set clock low
wait(1);
PORTEbits.RE7=cont[7];
wait(1);
PORTBbits.RB7 = 1; //Set clock high
wait(1);
//data
PORTBbits.RB7 = 0; //Set clock low
wait(1);
PORTEbits.RE7=bin[0];
wait(1);
PORTBbits.RB7 = 1; //Set clock high
wait(1);
PORTBbits.RB7 = 0; //Set clock low
wait(1);
PORTEbits.RE7=bin[1];
wait(1);
PORTBbits.RB7 = 1; //Set clock high
wait(1);
PORTBbits.RB7 = 0; //Set clock low
wait(1);
PORTEbits.RE7=bin[2];
wait(1);
PORTBbits.RB7 = 1; //Set clock high
wait(1);
PORTBbits.RB7 = 0; //Set clock low
wait(1);
PORTEbits.RE7=bin[3];
wait(1);
PORTBbits.RB7 = 1; //Set clock high
wait(1);
PORTBbits.RB7 = 0; //Set clock low
wait(1);
PORTEbits.RE7=bin[4];
wait(1);
PORTBbits.RB7 = 1; //Set clock high
wait(1);
PORTBbits.RB7 = 0; //Set clock low
wait(1);
PORTEbits.RE7=bin[5];
wait(1);
PORTBbits.RB7 = 1; //Set clock high
wait(1);
PORTBbits.RB7 = 0; //Set clock low
wait(1);
PORTEbits.RE7 = bin[6];
wait(1);
PORTBbits.RB7 = 1; //Set clock high
wait(1);
PORTBbits.RB7 = 0; //Set clock low
wait(1);
PORTEbits.RE7 = bin[7];
wait(1);
PORTBbits.RB7 = 1; //Set clock high
wait(1);
PORTBbits.RB7 = 0; //Set clock low
wait(1);
PORTBbits.RB5 = 1; //Set sync’ high
wait(1);
PORTBbits.RB7 = 1; //Set clock high
return 1;

int main(void)
{
    ADPCFG = 0xffff;  //Make ADC pins all digital
    TRISA = 0;        //Make all ports all outputs
    TRISB = 0;
    TRISD = 0;
    TRISE = 0;
    TRISF = 0;
    PORTBbits.RB5 = 1; //Set sync' high

    while(1){

        //Desired voltage wave function implemented here
        //Value of function can be sent to DAC

        DACA(1.1); //Set DACA output to 1.1 Volts or desired value
    }
}

This code was developed by design team 2 with the help of years of experience and the information gathered in the datasheets for both the PIC and the DAC.
Reference: