ADC - DAC Implementation on an ARM Processor using C programming Language

EE2016 Microprocessors

Sanjeev Subrahmaniyan S B

EE23B102

Abstract

ARM processors are the buzzword in the microprocessor industry now, with most devices having processors built on the ARM RISC architecture. They are particulary common in embedded and handheld devices like microcontrollers and smartphones. This experiment uses the LPC ARM chip, which is suited for low power applications. Using the C programming language to program ARM based microcontrollers is a very useful skill to be picked up from this course. The experiment involves the implementation of ADC and DAC on the LPC development kit provided in the laboratory. The ADC is implemented by converting the analog value in a potentiometer into binary numbers displayed on the LED array, while DAC is used to generate common functions like square, triangle, ramp and sine, which are observed on an oscilloscope.

1 Objectives

1. Use the DAC of the development kit to generate square, triangle, ramp and sine waveforms, and observe them on an oscilloscope.
2. Use the ADC to display the analog value(which is varied by tuning the knob of the trimpot on the development board) converted into a digital number on the LED array.

2 Programs and Results

2.1 DAC Implementation

2.1.1 Square Wave

The program for square wave was given as an example on moodle. The program is given here because the other programs simply have a change in the looping function alone.

```
#include "LPC214x.H"                              /* LPC214x
   definitions */

#define DAC_BIAS              0x00010000
void mydelay(int);

void DACInit( void )
{
    /* setup the related pin to DAC output */
    PINSEL1 &= 0xFFF3FFFF;
    PINSEL1 |= 0x00080000;  /* set p0.25 to DAC output */
    return;
}

int main (void)
{
    DACInit();

//SQUARE WAVE
while(1)
{
    DACR=0|DAC_BIAS;
      mydelay(0x0F);
      DACR=(0x3FF<<6)|DAC_BIAS;
      mydelay(0x0f);
}
    return 0;

}
void mydelay(int x)
{
int j,k;
for(j=0;j<=x;j++)
{
    for(k=0;k<=0xFF;k++);
```

```
}
}
```

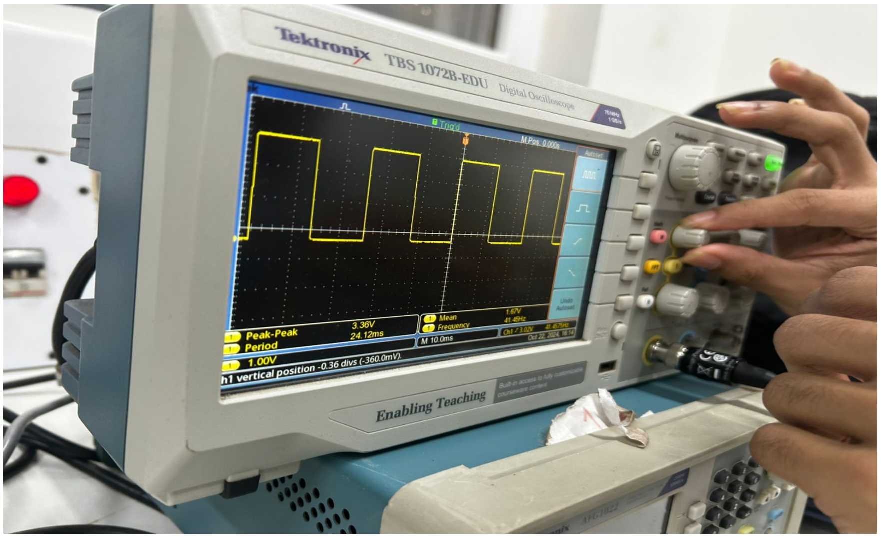

Figure 1: Square Wave generated and observed on the oscilloscope

2.1.2 Ramp Wave

The part of the code inside the main function corresponding to the wave generation is alone presented here. The remaining parts are the same as that of the program for the square wave.

```
while(1)
{
    DACR=0|DAC_BIAS;
      mydelay(0x01);


      for (int i = 0; i <= 0x3FF; i++){
            DACR = (i<<6) | DAC_BIAS;
            mydelay(0x01);
        }
```

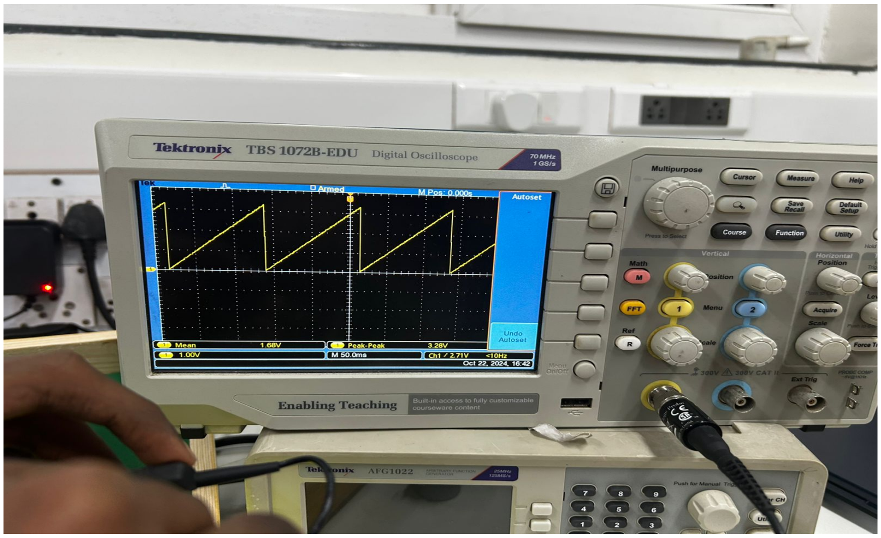

Figure 2: Oscilloscope image of the generated ramp wave

2.1.3 Triangle Wave

```
while(1)
{
    DACR=0|DAC_BIAS;
      mydelay(0x01);


      for (int i = 0; i <= 0x3FF; i++){
            DACR = (i<<6) | DAC_BIAS;
            mydelay(0x01);
        }
        for (int i=0x3FF; i>=0; i--){
            DACR = (i<<6) | DAC_BIAS;
            mydelay(0x01);
        }
```

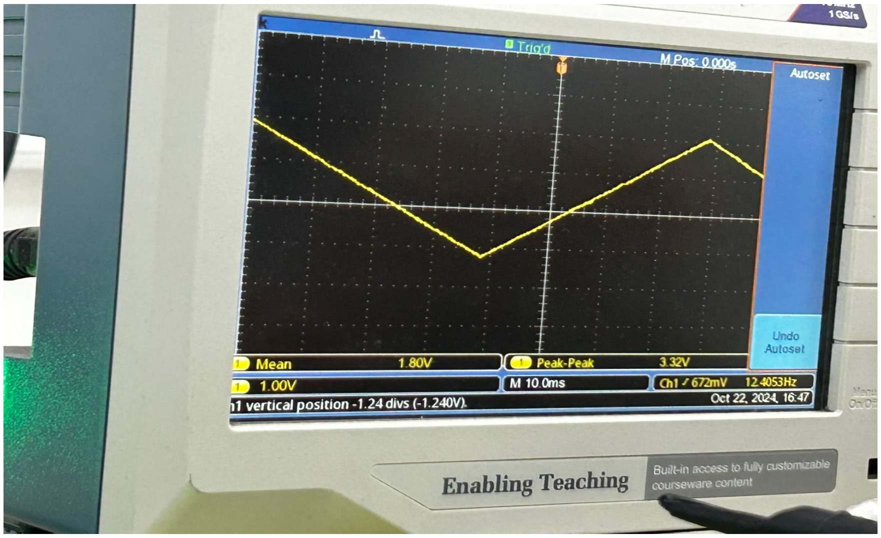

Figure 3: Oscilloscope image of the generated triangle wave

2.1.4 Sine Wave

```
while(1)
{
    DACR=0|DAC_BIAS;
      mydelay(0x01);


      for (int i = 0; i < 256; i++){

            DACR = ((sineLookupTable[i])<<6) | DAC_BIAS;
            mydelay(0x0F);
        }


}
```

Note that for the sine function generation, we have used a lookup table that we generated using a python script and attached it to the program code before building.

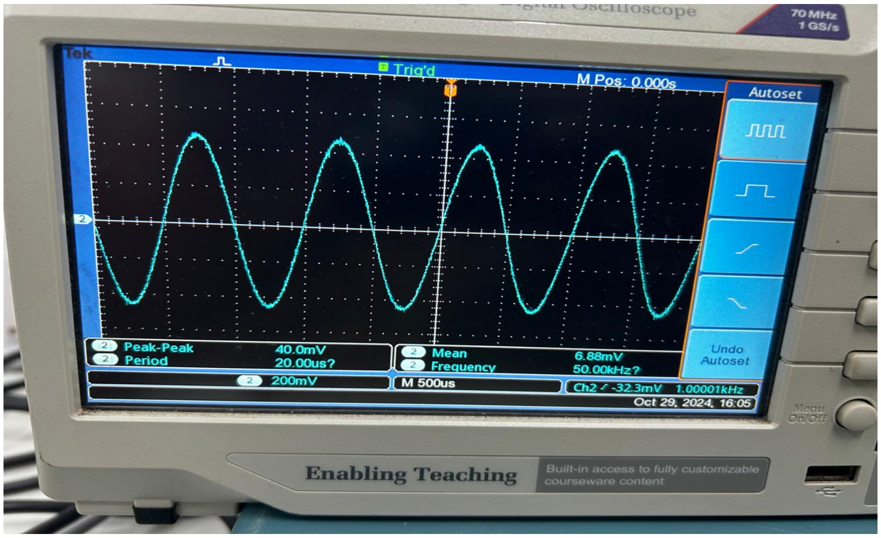

Figure 4: Oscilloscope image of the generated sine wave

2.2 ADC Conversion

The fixed code to implement the ADC on the development kit is done by adding the led turn off feature to the part of the program that triggers the corresponding leds. Appropriate definitions for the LEDOFF functions are also made.

```
#include <LPC214x.H>                                /* LPC214x
   definitions */

//#include "delay.h"

unsigned long Read_ADC0(unsigned char); /*
void Init_ADC0(unsigned char);

#ifndef __ADC_H
#define __ADC_H


#define CHANNEL_0    0
#define CHANNEL_1    1
#define CHANNEL_2    2
#define CHANNEL_3    3
#define CHANNEL_4    4
#define CHANNEL_5    5
#define CHANNEL_6    6
#define CHANNEL_7    7

```

```

/* Crystal frequency,10MHz~25MHz should be the same as actual status. */
#define Fosc          12000000    /* 12 MHz is the operational frequency of o/p dgtlClk */
#define ADC_CLK       1000000     /* set to 1Mhz */

/* A/D Converter 0 (AD0) */
#define AD0_BASE_ADDR       0xE0034000
#define ADC_INDEX           4

#define ADC_DONE            0x80000000
#define ADC_OVERRUN         0x40000000


#define ADC_FullScale_Volt  3.3 // 3.3V - ADC Referance Voltage
#define ADC_FullScale_Count 1024    // 2^10 - 10 bit ADC
#define LED_IOPIN       IO0PIN
#define BIT(x)  (1 << x)

#define LED_D0  (1 << 10)       // P0.10 mapping same as in Exp7 switch LED
#define LED_D1  (1 << 11)       // P0.11
#define LED_D2  (1 << 12)       // P0.12
#define LED_D3  (1 << 13)       // P0.13

#define LED_D4  (1 << 15)       // P0.15
#define LED_D5  (1 << 16)       // P0.16
#define LED_D6  (1 << 17)       // P0.17
#define LED_D7  (1 << 18)       // P0.18
#define LED_DATA_MASK           ((unsigned long)((LED_D7 | LED_D6 | LED_D5 | LED_D4 | LED_D3 | LED_D2 | LED_D1 | LED_D0 )))

#define LED1_ON     LED_IOPIN |= (unsigned long)(LED_D0);       // LED1 ON
#define LED2_ON     LED_IOPIN |= (unsigned long)(LED_D1);       // LED2 ON
#define LED3_ON     LED_IOPIN |= (unsigned long)(LED_D2);       // LED3 ON
#define LED4_ON     LED_IOPIN |= (unsigned long)(LED_D3);       // LED4 ON
#define LED5_ON     LED_IOPIN |= (unsigned long)(LED_D4);       // LED5 ON
```

```
#define LED6_ON     LED_IOPIN |= (unsigned long)(LED_D5);       // LED6 ON
#define LED7_ON     LED_IOPIN |= (unsigned long)(LED_D6);       // LED7 ON
#define LED8_ON     LED_IOPIN |= (unsigned long)(LED_D7);       // LED8 ON


#define LED1_OFF       LED_IOPIN |= 0;                          // LED1 OFF
#define LED2_OFF       LED_IOPIN |= 0;                          // LED2 OFF
#define LED3_OFF       LED_IOPIN |= 0;                          // LED3 OFF
#define LED4_OFF       LED_IOPIN |= 0;                          // LED4 OFF
#define LED5_OFF       LED_IOPIN |= 0;                          // LED5 OFF
#define LED6_OFF       LED_IOPIN |= 0;                          // LED6 OFF
#define LED7_OFF       LED_IOPIN |= 0;                          // LED7 OFF
#define LED8_OFF       LED_IOPIN |= 0;                          // LED8 OFF

#endif
#ifndef LED_DRIVER_OUTPUT_EN
#define LED_DRIVER_OUTPUT_EN (1 << 5)   // P0.5
#endif
//LED definitions





int main (void)
{

    unsigned long ADC_val;

    Init_ADC0(CHANNEL_1);
    Init_ADC0(CHANNEL_2);

    delay_mSec(100);

```

```
    IO0DIR |= LED_DATA_MASK;            // GPIO Direction control -> pin is output
    IO0DIR |= LED_DRIVER_OUTPUT_EN;     // GPIO Direction control -> pin is output
    IO0CLR |= LED_DRIVER_OUTPUT_EN;


    while(1)
    {
        //ADC_val = Read_ADC0(CHANNEL_1);
        ADC_val = Read_ADC0(CHANNEL_2);
        ADC_val=(ADC_val>>2);
        delay_mSec(5);
LED8 = (ADC_val & BIT(0)) ? LED8_ON : LED8_OFF;
LED7 = (ADC_val & BIT(1)) ? LED7_ON : LED7_OFF;
LED6 = (ADC_val & BIT(2)) ? LED6_ON : LED6_OFF;
LED5 = (ADC_val & BIT(3)) ? LED5_ON : LED5_OFF;

LED4 = (ADC_val & BIT(4)) ? LED4_ON : LED4_OFF;
LED3 = (ADC_val & BIT(5)) ? LED3_ON : LED3_OFF;
LED2 = (ADC_val & BIT(6)) ? LED2_ON : LED2_OFF;
LED1 = (ADC_val & BIT(7)) ? LED1_ON : LED1_OFF;


    }



    return 0;
}

void Init_ADC0(unsigned char channelNum)
{
    if(channelNum == CHANNEL_1)
        PINSEL1 = (PINSEL1 & ~(3 << 24)) | (1 << 24);       // P0.28 -> AD0.1

    if(channelNum == CHANNEL_2)
        PINSEL1 = (PINSEL1 & ~(3 << 26)) | (1 << 26);       // P0.29 -> AD0.2

    if(channelNum == CHANNEL_3)
        PINSEL1 = (PINSEL1 & ~(3 << 28)) | (1 << 28);       // P0.30 -> AD0.3

```

```

    AD0CR = ( 0x01 << 1 ) |                         // SEL=1, select channel 0, 1 to 4 on ADC0
            (( Fosc / ADC_CLK - 1 ) << 8 ) |      // CLKDIV = Fpclk / 1000000 - 1
            ( 0 << 16 ) |                           // BURST = 0, no BURST, software controlled
            ( 0 << 17 ) |                           // CLKS = 0, 11 clocks/10 bits
            ( 1 << 21 ) |                           // PDN = 1, normal operation
            ( 0 << 22 ) |                           // TEST1:0 = 00
            ( 0 << 24 ) |                           // START = 0 A/D conversion stops
            ( 0 << 27 );                            /* EDGE = 0 (CAP/MAT singal falling,trigger A/D conversion) */
}
unsigned long Read_ADC0( unsigned char channelNum )
{
    unsigned long regVal, ADC_Data;

    /* Clear all SEL bits */
    AD0CR &= 0xFFFFFF00;
    /* switch channel, start A/D convert */
    AD0CR |= (1 << 24) | (1 << channelNum);


    /* wait until end of A/D convert */
    while ( 1 ) {

//        regVal = *(volatile unsigned long *)(AD0_BASE_ADDR + ADC_INDEX);
        regVal = AD0GDR;

        if ( regVal & ADC_DONE ){
            break;
        }
    }

    /* stop ADC now */
    AD0CR &= 0xF8FFFFFF;
    /* save data when it's not overru otherwise, return zero */

    if ( regVal & ADC_OVERRUN ) {
```

```
        return ( 0 );
    }
    ADC_Data = ( regVal >> 6 ) & 0x3FF;
    /* return A/D conversion value */
    return ( ADC_Data );

}

void delay_mSec(int dCnt)       // pr_note:~dCnt mSec
{
  int j=0,i=0;

  while(dCnt--)
  {
      for(j=0;j<1000;j++)
      {
        /* At 60Mhz, the below loop introduces
        delay of 10 us */
        for(i=0;i<10;i++);
      }
  }
}
```

The above mentioned changes can be found in the fully completed code given above.

The video of us tuning the potentiometer and the led values accordingly changing can be found at this url `https://drive.google.com/file/d/1XbuXvfKBSVxo4XKUrx44M7j3JxKhy15k/view?usp=drive_link`

There is a question on what happens when the number of bits used in the ADC is decreased. This would mean the resolution of the ADC decreases, and the number of leds used to show the full range also decreases. This is because the same range of analog values is now captured in a lesser number of bits.

The quantisation error is related to the number of bits as 6.02 * n dB. This means, decreasing the number of error would decrease the magnitude of the quantisation error, but increases the net error as the quantisation error is usually measured below the signal.

3 Conclusion

From this experiment, I learned the key concepts related to signal conversion, and gained an understanding of their implementation on an ARM based processor. I also gained familiarity with the operation of an oscilloscope to measure generated waveforms.