Timers, ADC and DAC Interfacing

Course: ENGG*3640 Microcomputer Interfacing

Instructor: Radu Muresan

Student Names/Numbers:

Bilal Ayyache (0988616)

Emeka Madubuike (0948959)

Mohammed Al-Fakhri (0982745)

Date: 14/11/2018

Contents

OBJ

1. Introduction

The main objective of lab 5 is to understand how the Analog to Digital converter (ADC) and the Digital to analog converter (DAC) interface functions using the K60 Microcontroller. Lab 5 also introduces the concept behind using timers to control the ADC module. Understanding how the ADC and DAC module function is a great asset for any hardware developer to have as these modules is heavily used in the today's tech industry. The 16-bit analog to digital converter found in the K60 microcomputer is a successive approximation ADC module designed for operation within an integrated microcontroller system on chip. This module obtains great features that can be used to develop cutting edge hardware. Features to the ADC module include:

- Linear successive approximation algorithm with up to 16-bit resolution
- Input clock selectable from up to four sources
- Operation in Low-Power modes for lower noise
- Programmable Gain Amplifier (PGA) with up to x64 gain
- Temperature sensor
- Output format in 2's complement 16-bit sign extended for differential modes

ADC modules are used in most digital voltmeters for their linearity and flexibility. To understand this concept further, a voltmeter was developed using the ADC and DAC modules.

1.1. Lab Description

Using the 3-digit 7 segment display circuit developed in lab 4, the main task of lab 5 is to develop a voltmeter that measures the mean voltage of an input square wave generated by a DC power supply. After voltage level is obtained, the measured voltage is then displayed on the 7 segment display developed in the previous lab. The voltmeter developed can measure a DC voltage between 0 to 3.3V. The design samples an analog line in which the DC voltage is inputted. To cycle the 7- Segment digits the PIT Timer was used. The lab implementation uses interrupts for the PIT timer and the ADC0 module (further explanation of lab implementation can be found in section 3 of Lab 5.) After voltmeter was developed, output from a function generator was connected to the voltmeter. The ADC measurements was then inputted to the DAC module to measure the output signal of the DAC. Output signal was measured using an oscilloscope.

1.2. System Requirements

During this lab, the tools and equipment that were used are enumerated below:

- Kiel Uvision Program was used to create instructions to the K60 Microcontroller
- FreeScale TWR-K60D100M Microcontroller was used to implement instructions sent
- A Hi Speed USB 2.0 Connection cable between PC and board was used to connect the Microcontroller to the PC.
- Putty displayed the results by receiving commands through the COM port in which the USB 2.0 was connected to. Kiel Uvision software receives hardware input using the serial window. The Serial window accepts serial input and output data streams. The window displays serial output data received from a simulated CPU, while characters typed into a serial window are input to the simulated CPU. This allows testing a UART interface prior to having the target hardware.
- To implement the circuit design, a 7 Segment display, transistors, resistors, and a breadboard was used. (More information of equipment can be found in section 2.1).

2. Background

2.1. Equipment

- **Kiel Uvision Program**: The Kiel Uvision program is an IDE that combines project management, source code editing, program debugging, and run-time environment into a single powerful environment. Using this environment, the user is able to easily and efficiently test, verify, debug, and optimize the code developed. During the third lab, the debug functionality helped in understanding how the code is acting.

- **K60 Microcontroller**: The K60 Microcontroller (Figure2.1) from NXP contains a low power MCU core ARM Cortex-M4 that features an analog integration, serial communication, USB 2.0 full-speed OTG controller and 10/100 Mbps Ethernet MAC. These characteristics makes this Microcontroller suitable to preform task in a very efficient and fast manner. A USB

Figure 2.1.1 K60 Microcontroller

connection was used to sync the microcontrollerwith the Keil uVision software that that was provided by the teaching assistant. Through Lab 3, The main feature that was used was the NVIC component.

- **BreadBoard:** A breadboard is a construction base for prototyping of electronics. In this lab we used a breadboard to implement the logic functions by connecting our IC Chips' pins to satisfy the objective of this lab. IC Chips were placed between the board's valley and pins were connected using jumper wires.
- **3 digit 7 segment display:** consists of seven LEDs (hence its name) arranged in a rectangular fashion as shown. Each of the seven LEDs is called a segment because when illuminated the segment forms part of a numerical digit (both Decimal and Hex) to be displayed. The displays common pin is generally used to identify which type of 7-segment display it is. As each LED has two connecting pins, one called the "Anode" and the other called the "Cathode", there are therefore two types of LED 7-segment display called: Common Cathode (CC) and Common Anode (CA).
- **Resistors:** A resistor is a circuit element used to model the current resisting behavior of a material. For the purpose of constructing circuits, resistors are usually made from metallic alloys and carbon compounds
- **Transistors:** a transistor is a "nonlinear" component that has three leads. Transistors can be used for switching and amplifying signals.
- **DC Power Supply:** A DC power supply is an electronic device that supplies electricity to a circuit. The power supply job is to convert electric current from the source to the voltage and current required.
- **Oscilloscope:** An Oscilloscope is an electronic device that is used to view oscillation of current or voltage, by displaying the signal on a digital screen.
- **Function Generator:** A function generator is an electronic device that produces different types of electric waveforms over large variety of frequencies. It can generate a sine wave, square wave, or a triangular wave.

3. Implementation

As an implementation overview of this lab, it was required to develop a voltmeter that shows the input voltage on a 7-segment LED display screen. The implementation of this lab was approached from two different engineering perspectives. The first part is software implementation part that discusses how the code implementation was done including the configuration of all the registers. Many obstacles were encountered in this part and to overcome these, an Engineering approach was considered. The code outline was written on a paper and debugged before typing into uVision Keil for execution. The second part is basically discussing the hardware components used in the implementation of this application like transistors, resistors, jumper wires, oscilloscope, DC output generator, and 7-segment LED. This approach to lab 5 ensured success of experiment.

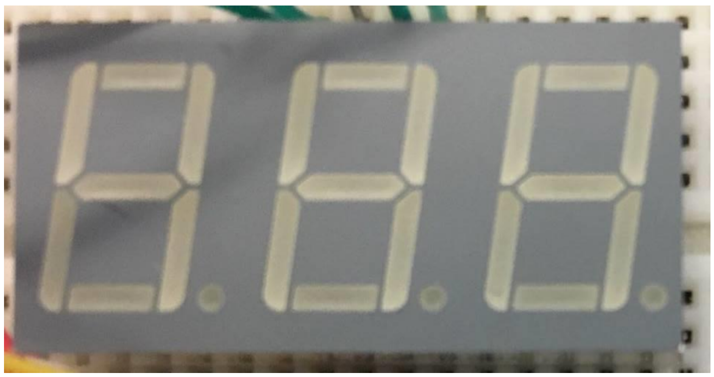

Figure 3.1 Running the Code

Figure 3.1 represents an overview of the implementation of this application without any input or output displayed, it shows the initial condition of this project before any changes happen, neither on the 7-segment LED display as an output nor in the code as an input.

3.1.1. Software Implementation

Software implementation of was performed using C- Programming. To start implementation, the ADC was first set up to initialize the required control registers. This implementation can be seen in figure B3 and B4 at line 172 to 184. After the ADC module was set up, the SIM_SCGC6 register bit 27 was set to 1. The main purpose of setting bit 27 to 1 is to enable the clock gate control to the ADC. The following action can be seen in line 151. In the same line Port A pin 23 was set to receive analog input. ADC0 configuration register is then set to low power mode as seen in line 284 and 285. The CFG2 was then set to the default setting with long sample time.

In Figure B.7., the hardware trigger is chosen as the conversion trigger in status control register 2. After that action was completed, interrupt and continuous conversion were enabled. Implementation can be seen in lines 392 to 400. The PDB counter is later reset by setting the software trigger value to 1 as seen in line 411.

For part 2 of the lab, The DAC clock gate was enabled using the SIM_SCGC2. This register was set to enable in lines 188 to 190. The upper bound limit was set to limit the DAC, then at line 230, the DAC system was enabled using the software trigger. This was completed by selecting the DAC trigger and setting the DAC reference in an appropriate manner. From lines 413 to 417 in the DACout function. The values obtained were loaded into the ADC0_IRQHandler. Only the first 12 bits of the ADC output was used because of the 12-bit accuracy feature in the DAC.

3.1.2. Hardware Implementation

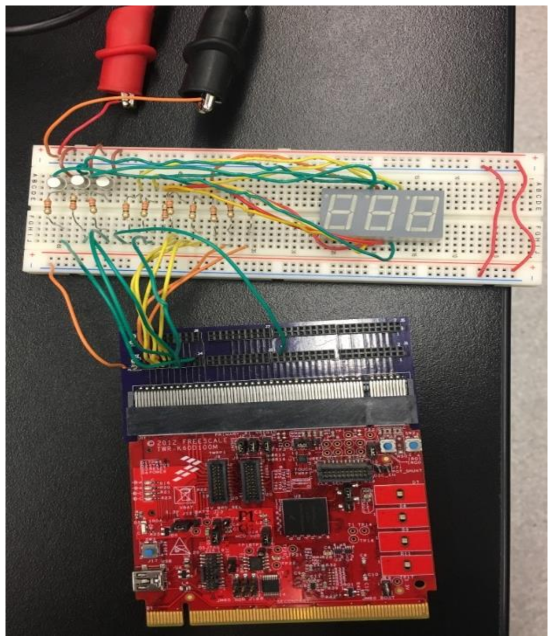

Figure 3.1.2.1 Circuit Connection

As shown in the circuit, the circuit was implemented using transistors, resistors, and a 3- Digit 7 segment. Voltage was supplied by the DC Output generator for part 2, and a function generator for part 3. Voltage goes from output source into transistor in which the GPIO pins in the microcomputer is connected to the base. Voltage from base allows voltage flow into the 3-digit 7 segment display.

when the user resets the microcomputer, the program runs. The 3-digit LED simply works when voltage is applied, so if the power source applies high on all segments and the microcomputer also applies high, the segments will not light up because the flow on both sides is the same which will cause the flow to stop and no voltage is going through the circuit. If the user inputs a number for example “1”, the board will set segments “f” and “e” low and the remaining segments to high this will cause segments “f” and “e” to light up.

3.2. Simulation Results

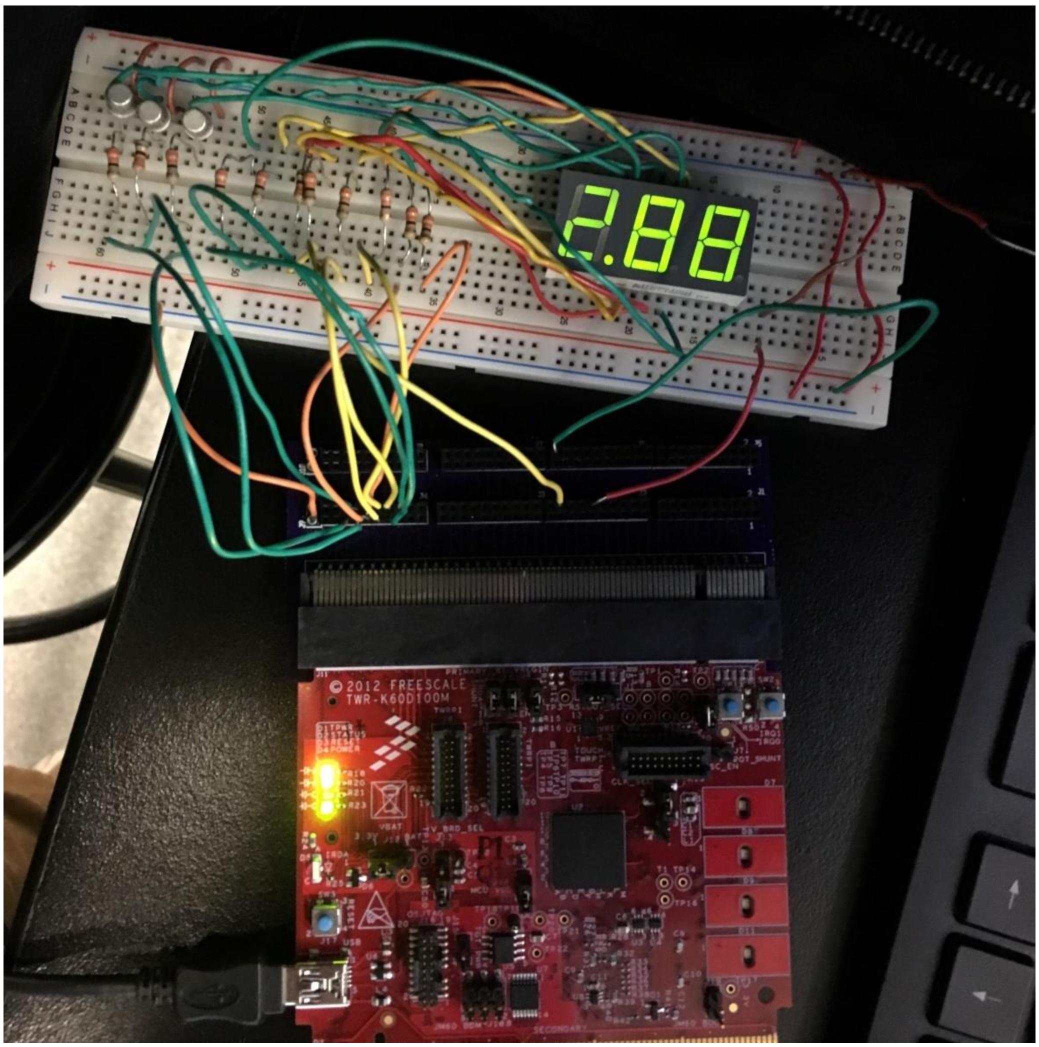

Figure 3.2.1. Results of running code with inputs

As shown in figure 3.2.1, after the code is uploaded on the board. The 3 digit LED will then display the voltage value that was inputted into the circuit and keeps displaying it until the voltage value or the program gets terminated.

3.3. Lab Requirements

3.3.1. Lab Requirement 1

For Lab requirement 1, please refer to the software implementation for detailed explanation. The software implementation can be found in section 3.1.1.

3.3.2. Lab Requirement 2

This requirement asks us to develop a voltmeter. Here we take in the DC voltage values onto an analog line, pass that into the ADC for conversions and present the converted ADC value on the seven-segment display. This lab was implemented in C hence this was done by configuring the ADC and calling the function ADC16_Measure to convert the analog value from the function generator to a digital value that can be used. Then the digital value is passed into a function that converts that to the final voltage value. In our calculations the voltage values were printed to putty and this allowed us to see that when rmeasuring values above 1.6 volts, the ADC value was becoming negative. Upon thorough research it was discovered that this was happening because overflow was occurring due to our use of 16 bit representation and this overflow caused the appearance of negative numbers after 1.6 volts. This was dealt with using logic that adjusted the ADC value back to positive when the overflow occurred. Then the measured voltage values were sent to and displayed on the seven-segment. The proof of this can be seen in Figure 3.2.1.

3.3.3. Lab Requirement 3

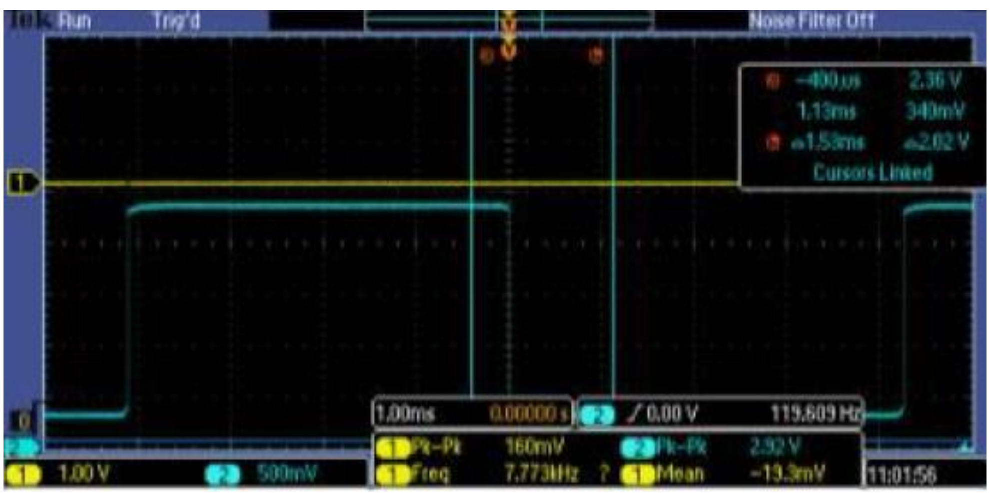

Figure 3.3.3.1: Requirement 3 time calculation

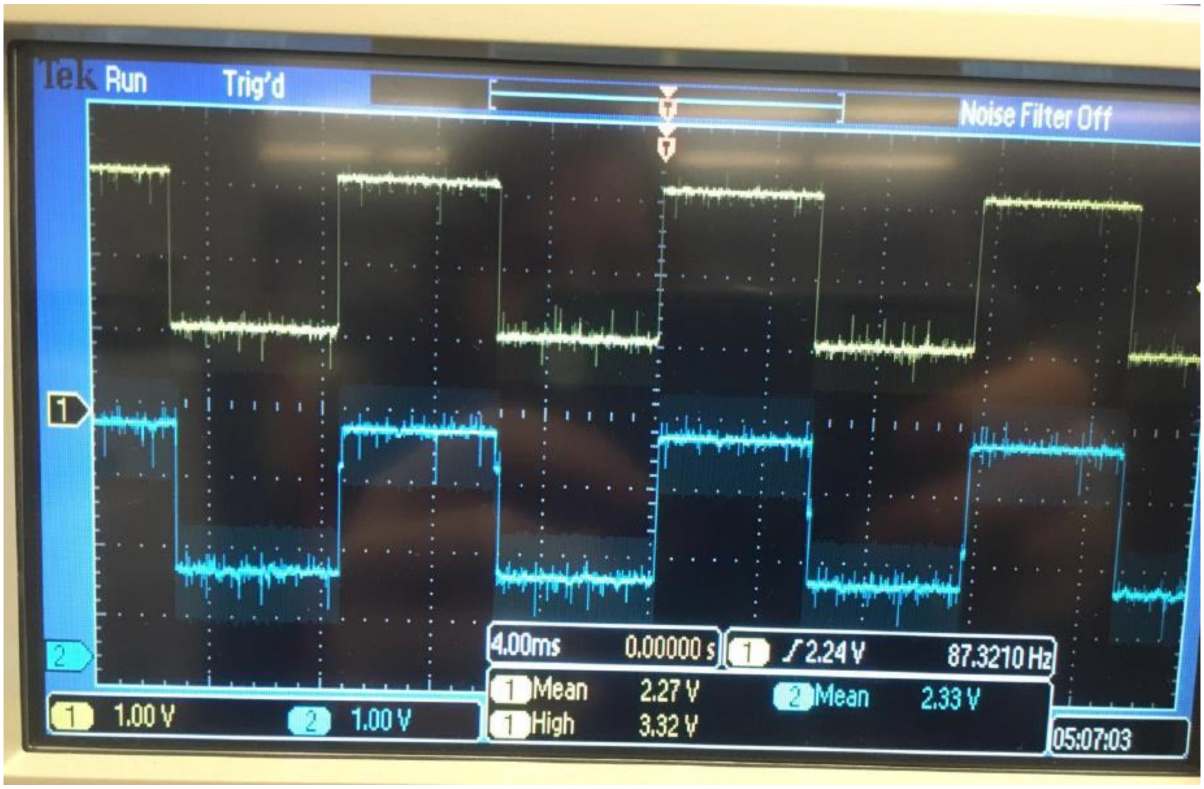

Figure 3.3.3.2: Requirement 3 Osciloscope Result

The observed error from requirement 3 settled around +/- 0.05V of the actual value. Several Factors can explain why this happened. The resolution in which the signal was sampled plays a big role in the output signal. The minimum step size was the smallest error achieved. Using a higher bit DAC's such as bit 18-20 will cut down this error but wouldn't eliminate it completely. Another factor is measurement error, in which changing the resolution of the scope changes the reported voltage value. Having a very high scaling factor (ex: using 10v per division) will result in poor measurement results as it increases the percentage error. Another source of error could be caused due to different couplings between the measured signals and equipment.

There are two methods of measuring the conversion time required by the digital to analog converter module:

Method 1: The conversion time required can be measured manually using an oscilloscope via the cursor setting. The conversion time required can be measured using the Math function on the oscilloscope. By monitoring the difference in time between the rising edge of both the input signal and the output signal, user can calculate how many microseconds the conversion took to perform. This can be seen in figure 3.3.3.1 above.

Method 2: The conversion time required can be measured using calculations in which the frequency and the step count is considered. The K60 Microcomputer is running at 100MHz. Converting this value to the time domain, a value of 2.0 *10^-8 seconds/ instruction was calculated. By referring back to the introduction section, we know that the DAC module used was running at 16 bits with a maximum reference voltage of 3.3 V. Using the following information, we get 5.035 * 10^-5 volts per step. It is known that each step of voltage takes approximately one instruction cycle. Assume two volt signals occur, it would take 7.94*10^-4 seconds to complete the calculations and send out the signal. This can be seen in figure 3.3.3.2 above.

4. Conclusion

To conclude the experiment from lab 5, the K60 Microcomputer was able to successfully output the mean value of the AC voltage input applied to the board. Results of this lab proves that the functionality of the experiment is evident and shows that the voltmeter implemented works properly. Results of lab 5 can be found in the simulation results section (section 3.3).

Through lab 5, the interface between DAC and ADC module was implemented using timers. As explained through the implementation section (section 3), this experiment was implemented using interrupts for the PIT and interrupts for the ADC0 module which is used through PTB0 timer. The ADC0 is programmed for 16-bit conversion, interrupts enable and hardware trigger with continuous trigger from the PTB0 timer. The ADC module was used to convert an analog voltage input to a digital value. To display results, configurations from lab 4 was used to utilize the three-digit seven segment display to successfully. output the mean voltage of the input signal supplied the DC supply. The conversion of the value back to analog using the DAC module and displaying it on the oscilloscope was also completed successfully.

References

[1] Radu Muresan, "ENGG3640: Microcomputer Interfacing Laboratory Manual, Version 2", University of Guelph, July 2016.

List Of Figures

1. Main.c

```
double get_current_voltage(uint32_t adcValue)
{
    float dum = 0;
    int lmao = adcValue;
    float currentTemp = 0;
    float test = 182;
    int test1 = 0;
    test1 = test/100;
    printf("%d",test1);
    if(lmao < 0){

        dum = ((-1*adcValue));
        currentTemp =(3.3 -(((dum)*3.30)/(65535)));

    }
    else{
    currentTemp =(((float) adcValue )*3.30)/(65535);
    }
        printf("\r\n Voltage %.21f ", currentTemp);
   // uint2bcd(currentTemp);
    return currentTemp;
}
```

Figure A. 1 ADC to Voltage function

```
void DACout(double ADCValue){
            int ADCValueInt = 0;
    ADCValueInt = floor((ADCValue * 65535 / 3.30));
    DAC_DRV_Output(DAC_INSTANCE, ADCValue);
}
```

Figure A. 2 DAC output function

```
int inttoBCD(int integer)
{
    switch(integer)
    {
       case 1:
           return 249UL;
       case 2:
           return 164UL;
       case 3:
           return 176UL;
       case 4:
           return 153UL;
       case 5:
           return 146UL;
       case 6:
           return 130UL;
       case 7:
           return 120UL;
       case 8:
           return 128UL;
       case 9:
           return 152UL;
       default:
           return 192UL;
    }
}
```

Figure A. 3 Integer to BCD function

```
    while(1)
    {
      ADC16_Measure();
            double currentvolt = get_current_voltage(adcValue);
            count++;
//            if (count == 1000){
//                avg = ((float) total / 1000);
//                printf("\r\nADC value average: %.2lf \r\n", avg);
//                voltageavg = ((float) avg*3.30 / 65535);
//                printf("Voltage average: %.2lf", voltageavg);
//                avg = 0;
//                total = 0;
//                count = 0;
//            }
          //total = total + adcValue;
            int int1 = 0;
            int int2 = 0;
            int int3 = 0;
            int testforint = 0;
            int num = 0;
            int q;
//            voltageavg = voltageavg*100;
//            num = (int)voltageavg;
            DACout(adcValue);
            currentvolt = currentvolt * 100;
            num = (int)currentvolt;
            int1 = num / 100;
            testforint = num % 100;
            int2 = testforint / 10;
            int3 = testforint % 10;

   int1 = inttoBCD(int1);
   int2 = inttoBCD(int2);
   int3 = inttoBCD(int3);
  if (true == pitIsrFlag[0])
        {
            if(q%4 == 0) // first segment
      {
       PTC->PCOR = 4095UL;
       PTC->PSOR = int1 + LD1;
      }
      else if(q%4 == 1)
      {
       PTC->PCOR = 4095UL;
       PTC->PSOR = dot + LD1;
      }
      else if (q%4 == 2)
      {
       PTC->PCOR = 4095UL;
       PTC->PSOR = int2 + LD2;
      }
      else if (q%4 == 3)
      {
       PTC->PCOR = 4095UL;
       PTC->PSOR = int3 + LD3;
      }
      q++;
      pitIsrFlag[0] = false;
    }
    }
    }
```

Figure A. 4 Main while loop for printing and outputting

Appendices

A. Main.c (requirement 2)

```c
// Standard C Included Files
#include <stdio.h>
#include <string.h>

#include "board.h"
#include "fsl_pmc_hal.h"
#include "fsl_adc16_driver.h"
#include "fsl_debug_console.h"
#include <gpio_pins.h>
#include <fsl_pit_driver.h>


///////////////////////////////////////////////////////////////////////////////
// Definitions
///////////////////////////////////////////////////////////////////////////////

/*!
 * @brief These values are used to get the temperature. DO NOT MODIFY
 * The method used in this demo to calculate temperature of chip is mapped to
 * Temperature Sensor for the HCS08 Microcontroller Family document (Document Number: AN3031)
 */
#define BOARD_PIT_INSTANCE  0
#define ADCR_VDD                (65535U)    // Maximum value when use 16b resolution
#define V_BG                    (1000U)     // BANDGAP voltage in mV (trim to 1.0V)
#define V_TEMP25                (716U)      // Typical converted value at 25 oC in mV
#define M                       (1620U)     // Typical slope:uV/oC
#define STANDARD_TEMP           (25)

#define ADC16_INSTANCE                (0)   // ADC instacne
#define ADC16_VOLTAGE_CHN          (kAdc16Chn10) // Temperature Sensor Channel
#define ADC16_BANDGAP_CHN             (kAdc16Chn27) // ADC channel of BANDGAP
#define ADC16_CHN_GROUP               (0)   // ADC group configuration selection

//volatile uint32_t msTicks;                           /* counts 1ms timeTicks */

///////////////////////////////////////////////////////////////////////////////
// Prototypes
///////////////////////////////////////////////////////////////////////////////
double ADC16_Measure(void);
void calibrateParams(void);
double get_current_voltage(uint32_t adcValue);

volatile bool pitIsrFlag[2] = {false};

///////////////////////////////////////////////////////////////////////////////
// Variables
///////////////////////////////////////////////////////////////////////////////
uint32_t adcValue = 0;               // ADC value
uint32_t adcrTemp25 = 0;             // Calibrated ADCR_TEMP25
uint32_t adcr100m = 0;               // calibrated conversion value of 100mV
adc16_converter_config_t adcUserConfig;   // structure for user config
///////////////////////////////////////////////////////////////////////////////
// Code
```

Figure A. 1 Main.c

```c

int inttoBCD(int integer)
{
    switch(integer)
    {
        case 1:
            return 249UL;
        case 2:
            return 164UL;
        case 3:
            return 176UL;
        case 4:
            return 153UL;
        case 5:
            return 146UL;
        case 6:
            return 130UL;
        case 7:
            return 120UL;
        case 8:
            return 128UL;
        case 9:
            return 152UL;
        default:
            return 192UL;
    }
}


/*!
 * @brief Measures internal temperature of chip.
 *
 * This function used the input of user as trigger to start the measurement.
 * When user press any key, the conversion will begin, then print
 * converted value and current temperature of the chip.
 */



int main(void)
{

  int LD3 = 1024UL;
  int LD2 = 512UL;
  int LD1 = 256UL;
  int dot= 127UL;
  int i = 0;



    pit_user_config_t chn0Confg = {
      .isInterruptEnabled = true,
      .periodUs = 10u
```

Figure A. 2 Main.c

```c
  };

hardware_init();


  /* INTERRUPT CLOCK GATE*/
  NVIC_EnableIRQ(PIT0_IRQn); //enable IT0 timer interrupt
  SIM->SCGC6 = (1UL << 23); //Turns on PIT timer clock logic one to bit 23)
  SIM->SCGC5 = (1UL << 11); //Turns on PORTC gate clock (logic one to bit 11)

  /* GPIO DISPLAY*/
  for (i = 0; i<16; i++)
  {
    PORTC->PCR[i]= (1UL << 8);   //Config pins 0-10 as general GPIO
  }

  PTC->PDDR = (2047UL);  //Set pins 0-10 as output GPIO

  //This above step is crucial to ensure we can output 0V or ~3.1V to each pin. Almost every pin has multiple functionalities,
  //and therefore must be programmed for the appropriate application.The for loop programs pins 0-16 as GPIO pins.


  /*---------------------PIT TIMER-----------------------*/
  PIT->MCR = (0UL << 1); //Enable PIT timer
  PIT->CHANNEL[0].LDVAL = 100; //Load value for PIT timer for 1 second interrupts
  PIT->CHANNEL[0].TCTRL =(3UL << 0); //Turn on interrupt and timer enable
  PIT->CHANNEL[0].TFLG = (1UL << 0); //Clear flag

    // Init hardware
   // hardware_init();

    // Initialization ADC for
    // 12bit resolution.
    // interrupt mode and hw trigger disabled,
    // normal convert speed, VREFH/L as reference,
    // disable continuous convert mode.
    ADC16_DRV_StructInitUserConfigDefault(&adcUserConfig);
    // Use 16bit resolution if enable.
#if (FSL_FEATURE_ADC16_MAX_RESOLUTION >= 16)
    adcUserConfig.resolution = kAdc16ResolutionBitOf16;
#endif

#if (  defined(FRDM_KL43Z)   /* CPU_MKL43Z256VLH4 */ \
    || defined(TWR_KL43Z48M) /* CPU_MKL43Z256VLH4 */ \
    || defined(FRDM_KL27Z)   /* CPU_MKL27Z64VLH4  */ \
    )
    adcUserConfig.refVoltSrc = kAdc16RefVoltSrcOfValt;
#endif
    ADC16_DRV_Init(ADC16_INSTANCE, &adcUserConfig);
    // Calibrate VDD and ADCR_TEMP25
    calibrateParams();
    PIT_DRV_StartTimer(BOARD_PIT_INSTANCE, 0);  // Start channel 0 of PIT Timer

```

Figure A. 3 Main.c

```c

    while(1)
    {
      ADC16_Measure();
      double currentvolt = get_current_voltage(adcValue);
      int int1 = 0;
      int int2 = 0;
      int int3 = 0;
      int testforint = 0;
      int num = 0;
      int q;

      currentvolt = currentvolt * 100;
      num = (int)currentvolt;
      int1 = num / 100;
      testforint = num % 100;
      int2 = testforint / 10;
      int3 = testforint % 10;

   int1 = inttoBCD(int1);
   int2 = inttoBCD(int2);
   int3 = inttoBCD(int3);
  if (true == pitIsrFlag[0])
       {
           if(q%4 == 0) // first segment
      {
       PTC->PCOR = 4095UL;
       PTC->PSOR = int1 + LD1;
      }
      else if(q%4 == 1)
      {
       PTC->PCOR = 4095UL;
       PTC->PSOR = dot + LD1;
      }
      else if (q%4 == 2)
      {
       PTC->PCOR = 4095UL;
       PTC->PSOR = int2 + LD2;
      }
      else if (q%4 == 3)
      {
       PTC->PCOR = 4095UL;
       PTC->PSOR = int3 + LD3;
      }
      q++;
      pitIsrFlag[0] = false;
    }
    }
    }

```

Figure A. 4 Main.c

```c
/*!
 * @brief Parameters calibration: VDD and ADCR_TEMP25
 *
 * This function used BANDGAP as reference voltage to measure vdd and
 * calibrate V_TEMP25 with that vdd value.
 */
void calibrateParams(void)
{
    adc16_chn_config_t adcChnConfig;
#if FSL_FEATURE_ADC16_HAS_HW_AVERAGE
    adc16_hw_average_config_t userHwAverageConfig;
#endif
    pmc_bandgap_buffer_config_t pmcBandgapConfig = {
        .enable = true,
#if FSL_FEATURE_PMC_HAS_BGEN
        .enableInLowPower = false,
#endif
#if FSL_FEATURE_PMC_HAS_BGBDS
        .drive = kPmcBandgapBufferDriveLow,
#endif
    };

    uint32_t bandgapValue = 0;  // ADC value of BANDGAP
    uint32_t vdd = 0;           // VDD in mV

#if FSL_FEATURE_ADC16_HAS_CALIBRATION
    // Auto calibration
    adc16_calibration_param_t adcCalibraitionParam;
    ADC16_DRV_GetAutoCalibrationParam(ADC16_INSTANCE, &adcCalibraitionParam);
    ADC16_DRV_SetCalibrationParam(ADC16_INSTANCE, &adcCalibraitionParam);
#endif // FSL_FEATURE_ADC16_HAS_CALIBRATION.

    // Enable BANDGAP reference voltage
    PMC_HAL_BandgapBufferConfig(PMC_BASE_PTR, &pmcBandgapConfig);

#if FSL_FEATURE_ADC16_HAS_HW_AVERAGE
    // Use hardware average to increase stability of the measurement.
    userHwAverageConfig.hwAverageEnable = true;
    userHwAverageConfig.hwAverageCountMode = kAdc16HwAverageCountOf32;
    ADC16_DRV_ConfigHwAverage(ADC16_INSTANCE, &userHwAverageConfig);
#endif // FSL_FEATURE_ADC16_HAS_HW_AVERAGE

    // Configure the conversion channel
    // differential and interrupt mode disable.
    adcChnConfig.chnIdx                  = (adc16_chn_t)ADC16_BANDGAP_CHN;
#if FSL_FEATURE_ADC16_HAS_DIFF_MODE
    adcChnConfig.diffConvEnable          = false;
#endif
    adcChnConfig.convCompletedIntEnable  = false;
    ADC16_DRV_ConfigConvChn(ADC16_INSTANCE, ADC16_CHN_GROUP, &adcChnConfig);

    // Wait for the conversion to be done
    ADC16_DRV_WaitConvDone(ADC16_INSTANCE, ADC16_CHN_GROUP);
```

Figure A. 5 Main.c

```c

    // Get current ADC BANDGAP value and format it.
    bandgapValue = ADC16_DRV_GetConvValueSigned(ADC16_INSTANCE, ADC16_CHN_GROUP);
    // Calculates bandgapValue in 16bit resolution
    // from 12bit resolution to calibrate.
#if (FSL_FEATURE_ADC16_MAX_RESOLUTION < 16)
    bandgapValue = bandgapValue << 4;
#endif
    // ADC stop conversion
    ADC16_DRV_PauseConv(ADC16_INSTANCE, ADC16_CHN_GROUP);

    // Get VDD value measured in mV
    // VDD = (ADCR_VDD x V_BG) / ADCR_BG
    vdd = ADCR_VDD * V_BG / bandgapValue;
    // Calibrate ADCR_TEMP25
    // ADCR_TEMP25 = ADCR_VDD x V_TEMP25 / VDD
    adcrTemp25 = ADCR_VDD * V_TEMP25 / vdd;
    // Calculate conversion value of 100mV.
    // ADCR_100M = ADCR_VDD x 100 / VDD
    adcr100m = ADCR_VDD*100/ vdd;


    // Disable BANDGAP reference voltage
    pmcBandgapConfig.enable = false;
    PMC_HAL_BandgapBufferConfig(PMC_BASE_PTR, &pmcBandgapConfig);
}

/*!
 * @brief Calculates the current temperature
 *
 * This funcion calculate temperatue used calibrated value as formula in reference manual.
 *
 * @param ADC convered value of temperature.
 * @return current temperature in oC.
 */
double get_current_voltage(uint32_t adcValue)
{
  float dum = 0;
  int lmao = adcValue;
  float currentTemp = 0;
  float test = 182;
  int test1 = 0;
  test1 = test/100;
  printf("%d",test1);
  if(lmao < 0){

    dum = ((-1*adcValue));
    currentTemp =(3.3 -(((dum)*3.30)/(65535)));

  }
  else{
    currentTemp =(((float) adcValue )*3.30)/(65535);
  }
```

Figure A. 6 Main.c

```c
  }
    printf("\r\n Voltage %.2lf ", currentTemp);
   // uint2bcd(currentTemp);
    return currentTemp;
}


/*!
 * @brief Gets current temperature of chip.
 *
 * This function gets convertion value, converted temperature and print them to terminal.
 */
double ADC16_Measure(void)
{
    adc16_chn_config_t chnConfig;

    // Configure the conversion channel
    // differential and interrupt mode disable.
    chnConfig.chnIdx     = (adc16_chn_t)ADC16_VOLTAGE_CHN;
#if FSL_FEATURE_ADC16_HAS_DIFF_MODE
    chnConfig.diffConvEnable = false;
#endif
    chnConfig.convCompletedIntEnable  = false;

    // Software trigger the conversion.
    ADC16_DRV_ConfigConvChn(ADC16_INSTANCE, ADC16_CHN_GROUP, &chnConfig);

    // Wait for the conversion to be done.
    ADC16_DRV_WaitConvDone(ADC16_INSTANCE, ADC16_CHN_GROUP);

    // Fetch the conversion value.
    adcValue = ADC16_DRV_GetConvValueSigned(ADC16_INSTANCE, ADC16_CHN_GROUP);

    // Show the current temperature value.
    PRINTF("\r\n ADC converted value: %ld\t", adcValue );
    // Calculates adcValue in 16bit resolution
    // from 12bit resolution in order to convert to temperature.
#if (FSL_FEATURE_ADC16_MAX_RESOLUTION < 16)
    adcValue = adcValue << 4;
#endif
double test1 = get_current_voltage(adcValue);
return test1;



    // Pause the conversion.
    ADC16_DRV_PauseConv(ADC16_INSTANCE, ADC16_CHN_GROUP);

}
```

Figure A. 7 Main.c

B. Main.c (requirement 3)

```c
/*
 * Copyright (c) 2013 - 2014, Freescale Semiconductor, Inc.
 * All rights reserved.
 *
 * Redistribution and use in source and binary forms, with or without modification,
 * are permitted provided that the following conditions are met:
 *
 * o Redistributions of source code must retain the above copyright notice, this list
 *   of conditions and the following disclaimer.
 *
 * o Redistributions in binary form must reproduce the above copyright notice, this
 *   list of conditions and the following disclaimer in the documentation and/or
 *   other materials provided with the distribution.
 *
 * o Neither the name of Freescale Semiconductor, Inc. nor the names of its
 *   contributors may be used to endorse or promote products derived from this
 *   software without specific prior written permission.
 *
 * THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS "AS IS" AND
 * ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED
 * WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE
 * DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT HOLDER OR CONTRIBUTORS BE LIABLE FOR
 * ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES
 * (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES;
 * LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON
 * ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT
 * (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS
 * SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.
 */
///////////////////////////////////////////////////////////////////////////////
//  Includes
///////////////////////////////////////////////////////////////////////////////

// Standard C Included Files
#include <stdio.h>
#include <string.h>
#include "board.h"
#include "fsl_pmc_hal.h"
#include "fsl_adc16_driver.h"
#include "fsl_debug_console.h"
#include <gpio_pins.h>
#include <fsl_pit_driver.h>
#include "fsl_dac_driver.h"
#include <math.h>

///////////////////////////////////////////////////////////////////////////////
// Definitions
///////////////////////////////////////////////////////////////////////////////

/*!
 * @brief These values are used to get the temperature. DO NOT MODIFY
 * The method used in this demo to calculate temperature of chip is mapped to
 * Temperature Sensor for the HCS08 Microcontroller Family document (Document Number: AN3031)
 */
#define BOARD_PIT_INSTANCE  0
#define ADCR_VDD                (65535U)    // Maximum value when use 16b resolution
#define V_BG                    (1000U)     // BANDGAP voltage in mV (trim to 1.0V)
#define V_TEMP25                (716U)      // Typical converted value at 25 oC in mV
#define M                       (1620U)     // Typical slope:uV/oC
#define STANDARD_TEMP           (25)
```

Figure B. 1 Main.c

```c

#define ADC16_INSTANCE                  (0)   // ADC instacne
#define ADC16_VOLTAGE_CHN          (kAdc16Chn10) // Temperature Sensor Channel
#define ADC16_BANDGAP_CHN               (kAdc16Chn27) // ADC channel of BANDGAP
#define ADC16_CHN_GROUP                 (0)   // ADC group configuration selection
#define DAC_INSTANCE           BOARD_DAC_DEMO_DAC_INSTANCE
//volatile uint32_t msTicks;                          /* counts 1ms timeTicks */

///////////////////////////////////////////////////////////////////////////////
// Prototypes
///////////////////////////////////////////////////////////////////////////////
double ADC16_Measure(void);
void calibrateParams(void);
double get_current_voltage(uint32_t adcValue);
void DACout(double ADCValue);
volatile bool pitIsrFlag[2] = {false};

///////////////////////////////////////////////////////////////////////////////
// Variables
///////////////////////////////////////////////////////////////////////////////
uint32_t adcValue = 0;              // ADC value
uint32_t adcrTemp25 = 0;            // Calibrated ADCR_TEMP25
uint32_t adcr100m = 0;              // calibrated conversion value of 100mV
adc16_converter_config_t adcUserConfig;   // structure for user config
///////////////////////////////////////////////////////////////////////////////
// Code
///////////////////////////////////////////////////////////////////////////////
 int count;


int inttoBCD(int integer)
{
    switch(integer)
    {
        case 1:
            return 249UL;
        case 2:
            return 164UL;
        case 3:
            return 176UL;
        case 4:
            return 153UL;
        case 5:
            return 146UL;
        case 6:
            return 130UL;
        case 7:
            return 120UL;
        case 8:
            return 128UL;
        case 9:
            return 152UL;
        default:
            return 192UL;
    }
}


/*!
 * @brief Measures internal temperature of chip.
```

Figure B. 2 Main.c

```c
 *
 * This function used the input of user as trigger to start the measurement.
 * When user press any key, the conversion will begin, then print
 * converted value and current temperature of the chip.
 */



int main(void)
{

    int LD3 = 1024UL;
    int LD2 = 512UL;
    int LD1 = 256UL;
    int dot= 127UL;
    int i = 0;



        pit_user_config_t chn0Confg = {
        .isInterruptEnabled = true,
        .periodUs = 10u
    };

hardware_init();


    /* INTERRUPT CLOCK GATE*/
  NVIC_EnableIRQ(PIT0_IRQn); //enable IT0 timer interrupt
  SIM->SCGC6 = (1UL << 23); //Turns on PIT timer clock logic one to bit 23)
    SIM->SCGC5 = (1UL << 11); //Turns on PORTC gate clock (logic one to bit 11)

  /* GPIO DISPLAY*/
  for (i = 0; i<16; i++)
    {
        PORTC->PCR[i]= (1UL << 8);    //Config pins 0-10 as general GPIO
    }

    PTC->PDDR = (2047UL);  //Set pins 0-10 as output GPIO

    //This above step is crucial to ensure we can output 0V or ~3.1V to each pin. Almost every pin has multiple functionalities,
    //and therefore must be programmed for the appropriate application.The for loop programs pins 0-16 as GPIO pins.


    /*---------------------PIT TIMER-----------------------*/
    PIT->MCR = (0UL << 1); //Enable PIT timer
    PIT->CHANNEL[0].LDVAL = 50000; //Load value for PIT timer for 1 second interrupts
    PIT->CHANNEL[0].TCTRL =(3UL << 0); //Turn on interrupt and timer enable
    PIT->CHANNEL[0].TFLG = (1UL << 0); //Clear flag


    ADC16_DRV_StructInitUserConfigDefault(&adcUserConfig);
    // Use 16bit resolution if enable.
#if (FSL_FEATURE_ADC16_MAX_RESOLUTION >= 16)
    adcUserConfig.resolution = kAdc16ResolutionBitOf16;
#endif

#if (  defined(FRDM_KL43Z)   /* CPU_MKL43Z256VLH4 */ \
    || defined(TWR_KL43Z48M) /* CPU_MKL43Z256VLH4 */ \
    || defined(FRDM_KL27Z)   /* CPU_MKL27Z64VLH4  */ \
```

Figure B. 3 Main.c

```
    }
    adcUserConfig.refVoltSrc = kAdc16RefVoltSrcOfValt;
#endif
    ADC16_DRV_Init(ADC16_INSTANCE, &adcUserConfig);
    // Calibrate VDD and ADCR_TEMP25
    calibrateParams();
    PIT_DRV_StartTimer(BOARD_PIT_INSTANCE, 0);    // Start channel 0 of PIT Timer
    dac_converter_config_t dacUserConfig;
    DAC_DRV_StructInitUserConfigNormal(&dacUserConfig);
    DAC_DRV_Init(DAC_INSTANCE, &dacUserConfig);



float avg;
uint32_t total;
float voltageavg;

    while(1)
    {
      ADC16_Measure();


      //req2
            double currentvolt = get_current_voltage(adcValue);
            count++;
      //req3


//            if (count == 1000){
//                avg = ((float) total / 1000);
//                printf("\r\nADC value average: %.2lf \r\n", avg);
//                voltageavg = ((float) avg*3.30 / 65535);
//                printf("Voltage average: %.2lf", voltageavg);
//                avg = 0;
//                total = 0;
//                count = 0;
//            }
//endre3

          //total = total + adcValue;
            int int1 = 0;
            int int2 = 0;
            int int3 = 0;
            int testforint = 0;
            int num = 0;
            int q;

//            voltageavg = voltageavg*100;
//            num = (int)voltageavg;
            DACout(adcValue);
            currentvolt = currentvolt * 100;
            num = (int)currentvolt;
            int1 = num / 100;
            testforint = num % 100;
            int2 = testforint / 10;
            int3 = testforint % 10;

   int1 = inttoBCD(int1);
   int2 = inttoBCD(int2);
   int3 = inttoBCD(int3);
```

Figure B. 4 Main.c

```c
    if (true == pitIsrFlag[0])
        {
            if(q%4 == 0) // first segment
        {
         PTC->PCOR = 4095UL;
         PTC->PSOR = int1 + LD1;
        }
        else if(q%4 == 1)
        {
         PTC->PCOR = 4095UL;
         PTC->PSOR = dot + LD1;
        }
        else if (q%4 == 2)
        {
         PTC->PCOR = 4095UL;
         PTC->PSOR = int2 + LD2;
        }
        else if (q%4 == 3)
        {
         PTC->PCOR = 4095UL;
         PTC->PSOR = int3 + LD3;
        }
        q++;
        pitIsrFlag[0] = false;
      }
    }
    }


/*!
 * @brief Parameters calibration: VDD and ADCR_TEMP25
 *
 * This function used BANDGAP as reference voltage to measure vdd and
 * calibrate V_TEMP25 with that vdd value.
 */
void calibrateParams(void)
{
    adc16_chn_config_t adcChnConfig;
#if FSL_FEATURE_ADC16_HAS_HW_AVERAGE
    adc16_hw_average_config_t userHwAverageConfig;
#endif
    pmc_bandgap_buffer_config_t pmcBandgapConfig = {
        .enable = true,
#if FSL_FEATURE_PMC_HAS_BGEN
        .enableInLowPower = false,
#endif
#if FSL_FEATURE_PMC_HAS_BGBDS
        .drive = kPmcBandgapBufferDriveLow,
#endif
    };

    uint32_t bandgapValue = 0;  // ADC value of BANDGAP
    uint32_t vdd = 0;           // VDD in mV

#if FSL_FEATURE_ADC16_HAS_CALIBRATION
    // Auto calibration
    adc16_calibration_param_t adcCalibraitionParam;
    ADC16_DRV_GetAutoCalibrationParam(ADC16_INSTANCE, &adcCalibraitionParam);
    ADC16_DRV_SetCalibrationParam(ADC16_INSTANCE, &adcCalibraitionParam);
#endif // FSL_FEATURE_ADC16_HAS_CALIBRATION.
```

Figure B. 5 Main.c

```c

    // Enable BANDGAP reference voltage
    PMC_HAL_BandgapBufferConfig(PMC_BASE_PTR, &pmcBandgapConfig);

#if FSL_FEATURE_ADC16_HAS_HW_AVERAGE
    // Use hardware average to increase stability of the measurement.
    userHwAverageConfig.hwAverageEnable = true;
    userHwAverageConfig.hwAverageCountMode = kAdc16HwAverageCountOf32;
    ADC16_DRV_ConfigHwAverage(ADC16_INSTANCE, &userHwAverageConfig);
#endif // FSL_FEATURE_ADC16_HAS_HW_AVERAGE

    // Configure the conversion channel
    // differential and interrupt mode disable.
    adcChnConfig.chnIdx                  = (adc16_chn_t)ADC16_BANDGAP_CHN;
#if FSL_FEATURE_ADC16_HAS_DIFF_MODE
    adcChnConfig.diffConvEnable          = false;
#endif
    adcChnConfig.convCompletedIntEnable  = false;
    ADC16_DRV_ConfigConvChn(ADC16_INSTANCE, ADC16_CHN_GROUP, &adcChnConfig);

    // Wait for the conversion to be done
    ADC16_DRV_WaitConvDone(ADC16_INSTANCE, ADC16_CHN_GROUP);

    // Get current ADC BANDGAP value and format it.
    bandgapValue = ADC16_DRV_GetConvValueSigned(ADC16_INSTANCE, ADC16_CHN_GROUP);
    // Calculates bandgapValue in 16bit resolution
    // from 12bit resolution to calibrate.
#if (FSL_FEATURE_ADC16_MAX_RESOLUTION < 16)
    bandgapValue = bandgapValue << 4;
#endif
    // ADC stop conversion
    ADC16_DRV_PauseConv(ADC16_INSTANCE, ADC16_CHN_GROUP);

    // Get VDD value measured in mV
    // VDD = (ADCR_VDD x V_BG) / ADCR_BG
    vdd = ADCR_VDD * V_BG / bandgapValue;
    // Calibrate ADCR_TEMP25
    // ADCR_TEMP25 = ADCR_VDD x V_TEMP25 / VDD
    adcrTemp25 = ADCR_VDD * V_TEMP25 / vdd;
    // Calculate conversion value of 100mV.
    // ADCR_100M = ADCR_VDD x 100 / VDD
    adcr100m = ADCR_VDD*100/ vdd;


    // Disable BANDGAP reference voltage
    pmcBandgapConfig.enable = false;
    PMC_HAL_BandgapBufferConfig(PMC_BASE_PTR, &pmcBandgapConfig);
}

/*!
 * @brief Calculates the current temperature
 *
 * This funcion calculate temperatue used calibrated value as formula in reference manual.
 *
 * @param ADC convered value of temperature.
 * @return current temperature in oC.
 */
double get_current_voltage(uint32_t adcValue)
{
    float dum = 0;
```

Figure B.6 Main.c

```c
    float dum = 0;
    int lmao = adcValue;
    float currentTemp = 0;
    float test = 182;
    int test1 = 0;
    test1 = test/100;
    printf("%d",test1);
    if(lmao < 0){

        dum = ((-1*adcValue));
        currentTemp =(3.3 -(((dum)*3.30)/(65535)));

    }
    else{
    currentTemp =(((float) adcValue )*3.30)/(65535);
    }
        printf("\r\n Voltage %.2lf ", currentTemp);
   // uint2bcd(currentTemp);
    return currentTemp;
}
double ADC16_Measure(void)
{
    adc16_chn_config_t chnConfig;

    // Configure the conversion channel
    // differential and interrupt mode disable.
    chnConfig.chnIdx     = (adc16_chn_t)ADC16_VOLTAGE_CHN;
#if FSL_FEATURE_ADC16_HAS_DIFF_MODE
    chnConfig.diffConvEnable = false;
#endif
    chnConfig.convCompletedIntEnable  = false;

    // Software trigger the conversion.
    ADC16_DRV_ConfigConvChn(ADC16_INSTANCE, ADC16_CHN_GROUP, &chnConfig);

    // Wait for the conversion to be done.
    ADC16_DRV_WaitConvDone(ADC16_INSTANCE, ADC16_CHN_GROUP);

    // Fetch the conversion value.
    adcValue = ADC16_DRV_GetConvValueSigned(ADC16_INSTANCE, ADC16_CHN_GROUP);

    // Show the current temperature value.
    PRINTF("\r\n ADC converted value: %ld\t", adcValue );
    // Calculates adcValue in 16bit resolution
    // from 12bit resolution in order to convert to temperature.
#if (FSL_FEATURE_ADC16_MAX_RESOLUTION < 16)
    adcValue = adcValue << 4;
#endif
double test1 = get_current_voltage(adcValue);
return test1;
    // Pause the conversion.
    ADC16_DRV_PauseConv(ADC16_INSTANCE, ADC16_CHN_GROUP);
}
void DACout(double ADCValue){
           int ADCValueInt = 0;
    ADCValueInt = floor((ADCValue * 65535 / 3.30));
    DAC_DRV_Output(DAC_INSTANCE, ADCValue);
}
```

Figure B.7 Main.c