

# 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

<b>1. Introduction</b>	1
1.1. Lab Description	1
1.2. System Requirements	2
<b>2. Background</b>	2
2.1. Equipment	2
<b>3. Implementation</b>	3
3.1.1. Software Implementation	4
3.1.2. Hardware Implementation	5
3.2. Simulation Results	6
3.3. Lab Requirements	7
3.3.1. Lab Requirement 1	7
3.3.2. Lab Requirement 2	7
3.3.3. Lab Requirement 3	7
<b>4. Conclusion</b>	9
<b>References</b>	9
<b>List Of Figures</b>	10
1. Main.c	10
<b>Appendices</b>	13
A. Main.c (requirement 2)	13
B. Main.c (requirement 3)	20

[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 are 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 were 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 (Figure 2.1) 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 perform task in a very efficient and fast manner. A USB



Figure 2.1.1 K60 Microcontroller

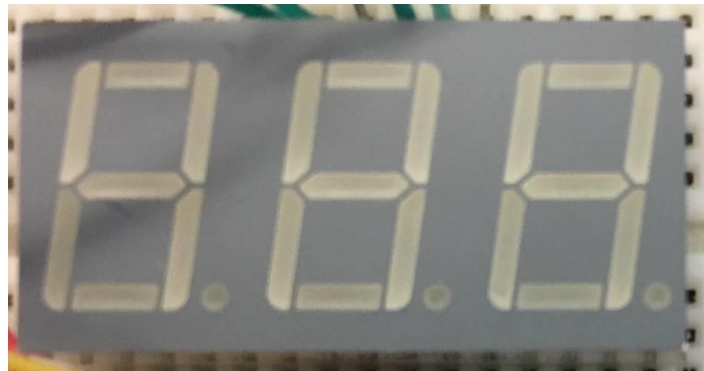
connection was used to sync the microcontroller with the Keil uVision software that was provided by the teaching assistant. Through Lab 3, The main feature that was used was the NVIC component.

- **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

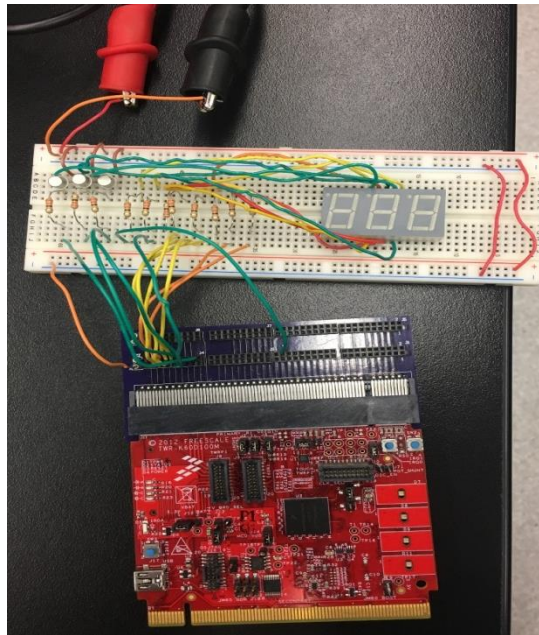


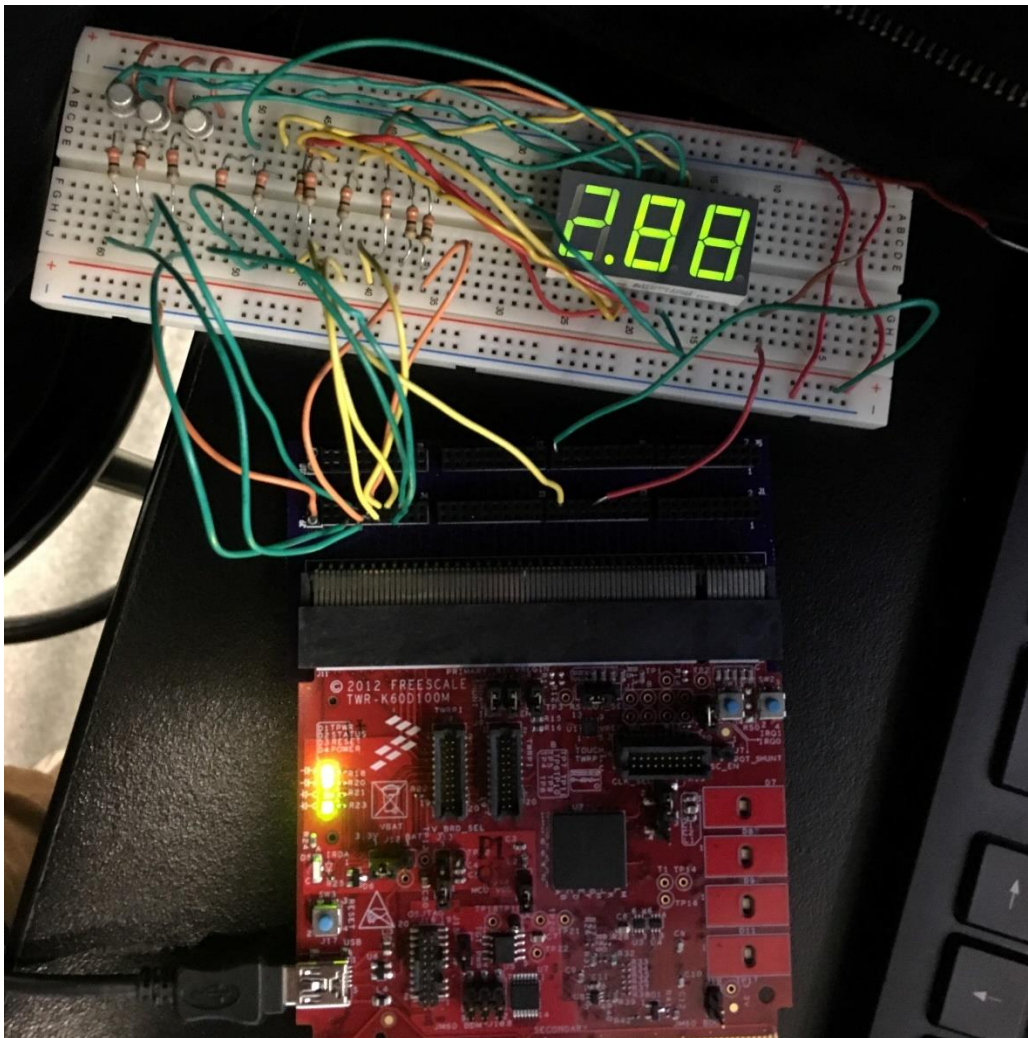
Figure 3.1.2.1 Circuit Connection

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 measuring 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

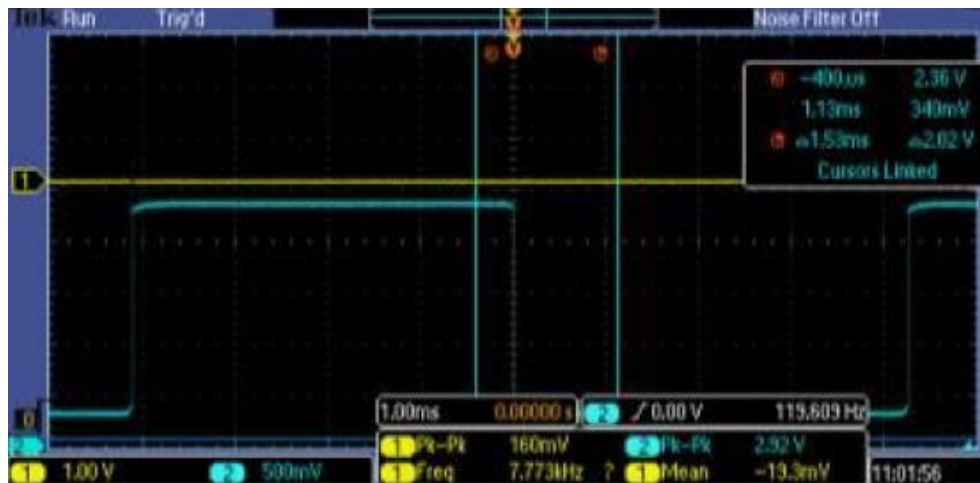


Figure 3.3.3.1: Requirement 3 time calculation



Figure 3.3.3.2: Requirement 3 Oscilloscope Result

The observed error from requirement 3 settled around  $\pm 0.05\text{V}$  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  $2.0 \times 10^{-8}$  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  $5.035 \times 10^{-5}$  volts per step. It is known that each step of voltage takes approximately one instruction cycle. Assume two volt signals occur, it would take  $7.94 \times 10^{-4}$  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: %.21f \r\n", avg);
    //     voltageavg = ((float) avg*3.30 / 65535);
    //     printf("Voltage average: %.21f", 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)

```

34 // Standard C Included Files
35 #include <stdio.h>
36 #include <string.h>
37
38 #include "board.h"
39 #include "fsl_pmc_hal.h"
40 #include "fsl_adc16_driver.h"
41 #include "fsl_debug_console.h"
42 #include <gpio_pins.h>
43 #include <fsl_pit_driver.h>
44
45
46 ///////////////////////////////////////////////////
47 // Definitions
48 ///////////////////////////////////////////////////
49
50 /*
51  * @brief These values are used to get the temperature. DO NOT MODIFY
52  * The method used in this demo to calculate temperature of chip is mapped to
53  * Temperature Sensor for the HCS08 Microcontroller Family document (Document Number: AN3031)
54  */
55 #define BOARD_PIT_INSTANCE 0
56 #define ADCR_VDD (65535U) // Maximum value when use 16b resolution
57 #define V_BG (1000U) // BANDGAP voltage in mV (trim to 1.0V)
58 #define V_TEMP25 (716U) // Typical converted value at 25 oC in mV
59 #define M (1620U) // Typical slope:uV/oC
60 #define STANDARD_TEMP (25)
61
62 #define ADC16_INSTANCE (0) // ADC instance
63 #define ADC16_VOLTAGE_CHN (kAdc16Chn10) // Temperature Sensor Channel
64 #define ADC16_BANDGAP_CHN (kAdc16Chn27) // ADC channel of BANDGAP
65 #define ADC16_CHN_GROUP (0) // ADC group configuration selection
66
67 //volatile uint32_t msTicks; // counts lms timeTicks */
68
69 ///////////////////////////////////////////////////
70 // Prototypes
71 ///////////////////////////////////////////////////
72 double ADC16_Measure(void);
73 void calibrateParams(void);
74 double get_current_voltage(uint32_t adcValue);
75
76 volatile bool pitIsrFlag[2] = {false};
77
78 ///////////////////////////////////////////////////
79 // Variables
80 ///////////////////////////////////////////////////
81 uint32_t adcValue = 0; // ADC value
82 uint32_t adcrTemp25 = 0; // Calibrated ADCR_TEMP25
83 uint32_t adcr100m = 0; // calibrated conversion value of 100mV
84 adc16_converter_config_t adcUserConfig; // structure for user config
85 ///////////////////////////////////////////////////
86 // Code

```

Figure A. 1 Main.c



```

88
89 int inttoBCD(int integer)
90 {
91     switch(integer)
92     {
93         case 1:
94             return 249UL;
95         case 2:
96             return 164UL;
97         case 3:
98             return 176UL;
99         case 4:
100             return 153UL;
101         case 5:
102             return 146UL;
103         case 6:
104             return 130UL;
105         case 7:
106             return 120UL;
107         case 8:
108             return 128UL;
109         case 9:
110             return 152UL;
111         default:
112             return 192UL;
113     }
114 }
115
116
117 /*!
118  * @brief Measures internal temperature of chip.
119  *
120  * This function used the input of user as trigger to start the measurement.
121  * When user press any key, the conversion will begin, then print
122  * converted value and current temperature of the chip.
123  */
124
125
126
127 int main(void)
128 {
129
130     int LD3 = 1024UL;
131     int LD2 = 512UL;
132     int LD1 = 256UL;
133     int dot= 127UL;
134     int i = 0;
135
136
137
138     pit_user_config_t chn0Config = {
139         .isInterruptEnabled = true,
140         .periodUs = 10u

```

Figure A. 2 Main.c

## Timers, ADC, and DAC Interfacing

```

141     };
142
143     hardware_init();
144
145     /* INTERRUPT CLOCK GATE*/
146     NVIC_EnableIRQ(PIT0_IRQn); //enable ITO timer interrupt
147     SIM->SCGC6 = (1UL << 23); //Turns on PIT timer clock logic one to bit 23)
148     SIM->SCGC5 = (1UL << 11); //Turns on PORTC gate clock (logic one to bit 11)
149
150     /* GPIO DISPLAY*/
151     for (i = 0; i<16; i++)
152     {
153         PORTC->PCR[i] = (1UL << 8); //Config pins 0-10 as general GPIO
154     }
155
156     PTC->PDDR = (2047UL); //Set pins 0-10 as output GPIO
157
158     //This above step is crucial to ensure we can output 0V or ~3.1V to each pin. Almost every pin has multiple functionalities,
159     //and therefore must be programmed for the appropriate application. The for loop programs pins 0-16 as GPIO pins.
160
161     /*-----PIT TIMER-----*/
162     PIT->MCR = (0UL << 1); //Enable PIT timer
163     PIT->CHANNEL[0].LDVAL = 100; //Load value for PIT timer for 1 second interrupts
164     PIT->CHANNEL[0].TCRRL = (3UL << 0); //Turn on interrupt and timer enable
165     PIT->CHANNEL[0].TFLG = (1UL << 0); //Clear flag
166
167     // Init hardware
168     // hardware_init();
169
170     // Initialization ADC for
171     // 12bit resolution.
172     // interrupt mode and hw trigger disabled,
173     // normal convert speed, VREFH/L as reference,
174     // disable continuous convert mode.
175     ADC16_DRV_StructInitUserConfigDefault(&adcUserConfig);
176     // Use 16bit resolution if enable.
177     #if (FSL_FEATURE_ADC16_MAX_RESOLUTION >= 16)
178     adcUserConfig.resolution = kAdc16ResolutionBitOf16;
179     #endif
180
181     #if ( defined(FRDM_KL43Z) /* CPU_MKL43Z256VLH4 */ \
182         || defined(TWR_KL43Z48M) /* CPU_MKL43Z256VLH4 */ \
183         || defined(FRDM_KL27Z) /* CPU_MKL27Z64VLH4 */ \
184     )
185     adcUserConfig.refVoltSrc = kAdc16RefVoltSrcOfVlt;
186     #endif
187
188     ADC16_DRV_Init(ADC16_INSTANCE, &adcUserConfig);
189     // Calibrate VDD and ADCR_TEMP25
190     calibrateParams();
191     PIT_DRV_StartTimer(BOARD_PIT_INSTANCE, 0); // Start channel 0 of PIT Timer
192
193

```

Figure A. 3 Main.c

## Timers, ADC, and DAC Interfacing

```
194
195 while(1)
196 {
197     ADC16_Measure();
198     double currentvolt = get_current_voltage(adcValue);
199     int int1 = 0;
200     int int2 = 0;
201     int int3 = 0;
202     int testforint = 0;
203     int num = 0;
204     int q;
205
206     currentvolt = currentvolt * 100;
207     num = (int)currentvolt;
208     int1 = num / 100;
209     testforint = num % 100;
210     int2 = testforint / 10;
211     int3 = testforint % 10;
212
213     int1 = inttoBCD(int1);
214     int2 = inttoBCD(int2);
215     int3 = inttoBCD(int3);
216     if (true == pitIsrFlag[0])
217     {
218         if(q%4 == 0) // first segment
219         {
220             PTC->PCOR = 4095UL;
221             PTC->PSOR = int1 + LD1;
222         }
223         else if(q%4 == 1)
224         {
225             PTC->PCOR = 4095UL;
226             PTC->PSOR = dot + LD1;
227         }
228         else if (q%4 == 2)
229         {
230             PTC->PCOR = 4095UL;
231             PTC->PSOR = int2 + LD2;
232         }
233         else if (q%4 == 3)
234         {
235             PTC->PCOR = 4095UL;
236             PTC->PSOR = int3 + LD3;
237         }
238         q++;
239         pitIsrFlag[0] = false;
240     }
241 }
242 }
243
244
```

Figure A. 4 Main.c

## Timers, ADC, and DAC Interfacing

```
245  /*!  
246  * @brief Parameters calibration: VDD and ADCR_TEMP25  
247  *  
248  * This function used BANDGAP as reference voltage to measure vdd and  
249  * calibrate V_TEMP25 with that vdd value.  
250  */  
251  void calibrateParams(void)  
252  {  
253      adc16_chn_config_t adcChnConfig;  
254      #if FSL_FEATURE_ADC16_HAS_HW_AVERAGE  
255          adc16_hw_average_config_t userHwAverageConfig;  
256      #endif  
257      pmc_bandgap_buffer_config_t pmcBandgapConfig = {  
258          .enable = true,  
259          #if FSL_FEATURE_PMC_HAS_BGEN  
260              .enableInLowPower = false,  
261          #endif  
262          #if FSL_FEATURE_PMC_HAS_BGBDS  
263              .drive = kPmcBandgapBufferDriveLow,  
264          #endif  
265      };  
266  
267      uint32_t bandgapValue = 0; // ADC value of BANDGAP  
268      uint32_t vdd = 0;         // VDD in mV  
269  
270      #if FSL_FEATURE_ADC16_HAS_CALIBRATION  
271          // Auto calibration  
272          adc16_calibration_param_t adcCalibrationParam;  
273          ADC16_DRV_GetAutoCalibrationParam(ADC16_INSTANCE, &adcCalibrationParam);  
274          ADC16_DRV_SetCalibrationParam(ADC16_INSTANCE, &adcCalibrationParam);  
275      #endif // FSL_FEATURE_ADC16_HAS_CALIBRATION.  
276  
277      // Enable BANDGAP reference voltage  
278      PMC_HAL_BandgapBufferConfig(PMC_BASE_PTR, &pmcBandgapConfig);  
279  
280      #if FSL_FEATURE_ADC16_HAS_HW_AVERAGE  
281          // Use hardware average to increase stability of the measurement.  
282          userHwAverageConfig.hwAverageEnable = true;  
283          userHwAverageConfig.hwAverageCountMode = kAdc16HwAverageCountOf32;  
284          ADC16_DRV_ConfigHwAverage(ADC16_INSTANCE, &userHwAverageConfig);  
285      #endif // FSL_FEATURE_ADC16_HAS_HW_AVERAGE  
286  
287      // Configure the conversion channel  
288      // differential and interrupt mode disable.  
289      adcChnConfig.chnIdx = (adc16_chn_t)ADC16_BANDGAP_CHN;  
290      #if FSL_FEATURE_ADC16_HAS_DIFF_MODE  
291          adcChnConfig.diffConvEnable = false;  
292      #endif  
293      adcChnConfig.convCompletedIntEnable = false;  
294      ADC16_DRV_ConfigConvChn(ADC16_INSTANCE, ADC16_CHN_GROUP, &adcChnConfig);  
295  
296      // Wait for the conversion to be done  
297      ADC16_DRV_WaitConvDone(ADC16_INSTANCE, ADC16_CHN_GROUP);
```

Figure A. 5 Main.c

## Timers, ADC, and DAC Interfacing

```
298
299 // Get current ADC BANDGAP value and format it.
300 bandgapValue = ADC16_DRV_GetConvValueSigned(ADC16_INSTANCE, ADC16_CHN_GROUP);
301 // Calculates bandgapValue in 16bit resolution
302 // from 12bit resolution to calibrate.
303 #if (FSL_FEATURE_ADC16_MAX_RESOLUTION < 16)
304 bandgapValue = bandgapValue << 4;
305 #endif
306 // ADC stop conversion
307 ADC16_DRV_FeuseConv(ADC16_INSTANCE, ADC16_CHN_GROUP);
308
309 // Get VDD value measured in mV
310 // VDD = (ADCR_VDD x V_BG) / ADCR_BG
311 vdd = ADCR_VDD * V_BG / bandgapValue;
312 // Calibrate ADCR_TEMP25
313 ADCR_TEMP25 = ADCR_VDD x V_TEMP25 / VDD
314 adcrTemp25 = ADCR_VDD * V_TEMP25 / vdd;
315 // Calculate conversion value of 100mV.
316 // ADCR_100M = ADCR_VDD x 100 / VDD
317 adcr100m = ADCR_VDD*100/ vdd;
318
319
320 // Disable BANDGAP reference voltage
321 pmcBandgapConfig.enable = false;
322 PMC_HAL_BandgapBufferConfig(PMC_BASE_PTR, &pmcBandgapConfig);
323 }
324
325 /*
326 * @brief Calculates the current temperature
327 *
328 * This function calculate temperatue used calibrated value as formula in reference manual.
329 *
330 * @param ADC converted value of temperature.
331 * @return current temperature in oC.
332 */
333 double get_current_voltage(uint32_t adcValue)
334 {
335     float dum = 0;
336     int lmao = adcValue;
337     float currentTemp = 0;
338     float test = 182;
339     int test1 = 0;
340     test1 = test/100;
341     printf("%d", test1);
342     if(lmao < 0){
343         dum = ((-1*adcValue));
344         currentTemp = (3.3 - (((dum)*3.30)/(65535)));
345     }
346     else{
347         currentTemp = (((float) adcValue )*3.30)/(65535);
348     }
349 }
350
```

Figure A. 6 Main.c

## Timers, ADC, and DAC Interfacing

```

350     }
351     printf("\r\n Voltage %.2lf ", currentTemp);
352     // uint2bcd(currentTemp);
353     return currentTemp;
354 }
355
356
357 /*
358  * @brief Gets current temperature of chip.
359  *
360  * This function gets conversion value, converted temperature and print them to terminal.
361  */
362 double ADC16_Measure(void)
363 {
364     adc16_chn_config_t chnConfig;
365
366     // Configure the conversion channel
367     // differential and interrupt mode disable.
368     chnConfig.chnIdx = (adc16_chn_t)ADC16_VOLTAGE_CHN;
369 #if FSL_FEATURE_ADC16_HAS_DIFF_MODE
370     chnConfig.diffConvEnable = false;
371 #endif
372     chnConfig.convCompletedIntEnable = false;
373
374     // Software trigger the conversion.
375     ADC16_DRV_ConfigConvChn(ADC16_INSTANCE, ADC16_CHN_GROUP, &chnConfig);
376
377     // Wait for the conversion to be done.
378     ADC16_DRV_WaitConvDone(ADC16_INSTANCE, ADC16_CHN_GROUP);
379
380     // Fetch the conversion value.
381     adcValue = ADC16_DRV_GetConvValueSigned(ADC16_INSTANCE, ADC16_CHN_GROUP);
382
383     // Show the current temperature value.
384     PRINTF("\r\n ADC converted value: %ld\t", adcValue );
385     // Calculates adcValue in 16bit resolution
386     // from 12bit resolution in order to convert to temperature.
387 #if (FSL_FEATURE_ADC16_MAX_RESOLUTION < 16)
388     adcValue = adcValue << 4;
389 #endif
390     double test1 = get_current_voltage(adcValue);
391     return test1;
392
393
394
395     // Pause the conversion.
396     ADC16_DRV_PauseConv(ADC16_INSTANCE, ADC16_CHN_GROUP);
397 }
398
399

```

Figure A. 7 Main.c



## B. Main.c (requirement 3)

```

1  /*
2  * Copyright (c) 2013 - 2014, Freescale Semiconductor, Inc.
3  * All rights reserved.
4  *
5  * Redistribution and use in source and binary forms, with or without modification,
6  * are permitted provided that the following conditions are met:
7  *
8  * o Redistributions of source code must retain the above copyright notice, this list
9  *   of conditions and the following disclaimer.
10 *
11 * o Redistributions in binary form must reproduce the above copyright notice, this
12 *   list of conditions and the following disclaimer in the documentation and/or
13 *   other materials provided with the distribution.
14 *
15 * o Neither the name of Freescale Semiconductor, Inc. nor the names of its
16 *   contributors may be used to endorse or promote products derived from this
17 *   software without specific prior written permission.
18 *
19 * THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS "AS IS" AND
20 * ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED
21 * WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE
22 * DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT HOLDER OR CONTRIBUTORS BE LIABLE FOR
23 * ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES
24 * (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES;
25 * LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON
26 * ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT
27 * (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS
28 * SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.
29 */
30 ///////////////////////////////////////////////////////////////////
31 // Includes
32 ///////////////////////////////////////////////////////////////////
33
34 // Standard C Included Files
35 #include <stdio.h>
36 #include <string.h>
37 #include "board.h"
38 #include "fsl_pmc_hal.h"
39 #include "fsl_adc16_driver.h"
40 #include "fsl_debug_console.h"
41 #include <gpio_pins.h>
42 #include <fsl_pit_driver.h>
43 #include "fsl_dac_driver.h"
44 #include <math.h>
45
46 ///////////////////////////////////////////////////////////////////
47 // Definitions
48 ///////////////////////////////////////////////////////////////////
49
50 /*!
51 * @brief These values are used to get the temperature. DO NOT MODIFY
52 * The method used in this demo to calculate temperature of chip is mapped to
53 * Temperature Sensor for the HCS08 Microcontroller Family document (Document Number: AN3031)
54 */
55 #define BOARD_PIT_INSTANCE 0
56 #define ADCR_VDD            (65535U)    // Maximum value when use 16b resolution
57 #define V_BG                (1000U)     // BANDGAP voltage in mV (trim to 1.0V)
58 #define V_TEMP35            (716U)      // Typical converted value at 25 oC in mV
59 #define M                    (1620U)    // Typical slope: uV/oC
60 #define STANDARD_TEMP       (25)

```

Figure B. 1 Main.c



## Timers, ADC, and DAC Interfacing

```

61
62 #define ADC16_INSTANCE          (0) // ADC instance
63 #define ADC16_VOLTAGE_CHN      (kAdc16Chn10) // Temperature Sensor Channel
64 #define ADC16_BANDGAP_CHN     (kAdc16Chn27) // ADC channel of BANDGAP
65 #define ADC16_CHN_GROUP       (0) // ADC group configuration selection
66 #define DAC_INSTANCE          BOARD_DAC_DEMO_DAC_INSTANCE
67 //volatile uint32_t msTicks; // counts lms timeTicks */
68
69 // Prototypes
70 // Prototypes
71 // Prototypes
72 double ADC16_Measure(void);
73 void calibrateParams(void);
74 double get_current_voltage(uint32_t adcValue);
75 void DACout(double ADCValue);
76 volatile bool pitIsrFlag[2] = {false};
77
78 // Variables
79 // Variables
80 // Variables
81 uint32_t adcValue = 0; // ADC value
82 uint32_t adcrTemp25 = 0; // Calibrated ADCR_TEMP25
83 uint32_t adcr100m = 0; // calibrated conversion value of 100mV
84 adc16_converter_config_t adcUserConfig; // structure for user config
85 // Code
86 // Code
87 // Code
88 int count;
89
90
91 int inttoBCD(int integer)
92 {
93     switch(integer)
94     {
95         case 1:
96             return 249UL;
97         case 2:
98             return 164UL;
99         case 3:
100             return 176UL;
101         case 4:
102             return 153UL;
103         case 5:
104             return 146UL;
105         case 6:
106             return 130UL;
107         case 7:
108             return 120UL;
109         case 8:
110             return 128UL;
111         case 9:
112             return 152UL;
113         default:
114             return 192UL;
115     }
116 }
117
118
119 /*
120 * @brief Measures internal temperature of chip.

```

Figure B. 2 Main.c

## Timers, ADC, and DAC Interfacing

```
121 *
122 * This function used the input of user as trigger to start the measurement.
123 * When user press any key, the conversion will begin, then print
124 * converted value and current temperature of the chip.
125 */
126
127
128
129 int main(void)
130 {
131
132     int LD3 = 1024UL;
133     int LD2 = 512UL;
134     int LD1 = 256UL;
135     int dot= 127UL;
136     int i = 0;
137
138
139
140     pit_user_config_t chn0Config = {
141         .isInterruptEnabled = true,
142         .periodUs = 10u
143     };
144
145     hardware_init();
146
147
148     /* INTERRUPT CLOCK GATE*/
149     NVIC_EnableIRQ(PIT0_IRQn); //enable IT0 timer interrupt
150     SIM->SCGC6 = (1UL << 23); //Turns on PIT timer clock logic one to bit 23)
151     SIM->SCGC5 = (1UL << 11); //Turns on PORTC gate clock (logic one to bit 11)
152
153     /* GPIO DISPLAY*/
154     for (i = 0; i<16; i++)
155     {
156         PORTC->PCR[i]= (1UL << 8); //Config pins 0-10 as general GPIO
157     }
158
159     PTC->PDDR = (2047UL); //Set pins 0-10 as output GPIO
160
161     //This above step is crucial to ensure we can output 0V or ~3.1V to each pin. Almost every pin has multiple functionalities,
162     //and therefore must be programmed for the appropriate application.The for loop programs pins 0-16 as GPIO pins.
163
164
165     /*-----PIT TIMER-----*/
166     PIT->MCR = (0UL << 1); //Enable PIT timer
167     PIT->CHANNEL[0].LDVAL = 50000; //Load value for PIT timer for 1 second interrupts
168     PIT->CHANNEL[0].CTRL = (3UL << 0); //Turn on interrupt and timer enable
169     PIT->CHANNEL[0].TFLG = (1UL << 0); //Clear flag
170
171
172     ADCL6_DRV_StructInitUserConfigDefault(&adcUserConfig);
173     // Use 16bit resolution if enable.
174     #if (FSL_FEATURE_ADC16_MAX_RESOLUTION >= 16)
175     adcUserConfig.resolution = kAdcl6ResolutionBitOf16;
176     #endif
177
178     #if ( defined(FRDM_KL43Z) /* CPU_MKL43Z256VLH4 */ \
179         || defined(TWR_KL43Z40M) /* CPU_MKL43Z256VLH4 */ \
180         || defined(FRDM_KL27Z) /* CPU_MKL27Z64VLH4 */ \
```

Figure B. 3 Main.c

```

181     )
182     adcUserConfig.refVoltSrc = kAdc16RefVoltSrcOfValt;
183 #endif
184     ADC16_DRV_Init(ADC16_INSTANCE, &adcUserConfig);
185     // Calibrate VDD and ADCR_TEMP25
186     calibrateParams();
187     PIT_DRV_StartTimer(BOARD_PIT_INSTANCE, 0); // Start channel 0 of PIT Timer
188     dac_converter_config_t dacUserConfig;
189     DAC_DRV_StructInitUserConfigNormal(&dacUserConfig);
190     DAC_DRV_Init(DAC_INSTANCE, &dacUserConfig);
191
192
193
194     float avg;
195     uint32_t total;
196     float voltageavg;
197
198     while(1)
199     {
200         ADC16_Measure();
201
202
203         //req2
204         double currentvolt = get_current_voltage(adcValue);
205         count++;
206         //req3
207
208
209         if (count == 1000){
210             // avg = ((float) total / 1000);
211             // printf("\r\nADC value average: %.2lf \r\n", avg);
212             // voltageavg = ((float) avg*3.30 / 65535);
213             // printf("Voltage average: %.2lf", voltageavg);
214             // avg = 0;
215             // total = 0;
216             // count = 0;
217         }
218         //endre3
219
220         //total = total + adcValue;
221         int int1 = 0;
222         int int2 = 0;
223         int int3 = 0;
224         int testforint = 0;
225         int num = 0;
226         int q;
227
228         // voltageavg = voltageavg*100;
229         // num = (int)voltageavg;
230         DACOut(adcValue);
231         currentvolt = currentvolt * 100;
232         num = (int)currentvolt;
233         int1 = num / 100;
234         testforint = num % 100;
235         int2 = testforint / 10;
236         int3 = testforint % 10;
237
238         int1 = inttoBCD(int1);
239         int2 = inttoBCD(int2);
240         int3 = inttoBCD(int3);

```

Figure B. 4 Main.c

```

241     if (true == pitIsrFlag[0])
242     {
243         if(q%4 == 0) // first segment
244         {
245             PTC->PCOR = 4096UL;
246             PTC->PSOR = int1 + LD1;
247         }
248         else if(q%4 == 1)
249         {
250             PTC->PCOR = 4096UL;
251             PTC->PSOR = dot + LD1;
252         }
253         else if (q%4 == 2)
254         {
255             PTC->PCOR = 4096UL;
256             PTC->PSOR = int2 + LD2;
257         }
258         else if (q%4 == 3)
259         {
260             PTC->PCOR = 4096UL;
261             PTC->PSOR = int3 + LD3;
262         }
263         q++;
264         pitIsrFlag[0] = false;
265     }
266 }
267 }
268
269
270 /*!
271  * @brief Parameters calibration: VDD and ADCR_TEMP25
272  *
273  * This function used BANDGAP as reference voltage to measure vdd and
274  * calibrate V_TEMP25 with that vdd value.
275  */
276 void calibrateParams(void)
277 {
278     adcl6_chn_config_t adcChnConfig;
279     #if FSL_FEATURE_ADCL6_HAS_HW_AVERAGE
280     adcl6_hw_average_config_t userHwAverageConfig;
281     #endif
282     pmc_bandgap_buffer_config_t pmcBandgapConfig = {
283         .enable = true,
284         #if FSL_FEATURE_PMC_HAS_BGEN
285         .enableInLowPower = false,
286         #endif
287         #if FSL_FEATURE_PMC_HAS_BGBDS
288         .drive = kPmcBandgapBufferDriveLow,
289         #endif
290     };
291
292     uint32_t bandgapValue = 0; // ADC value of BANDGAP
293     uint32_t vdd = 0;         // VDD in mV
294
295     #if FSL_FEATURE_ADCL6_HAS_CALIBRATION
296     // Auto calibration
297     adcl6_calibration_param_t adcCalibrationParam;
298     ADC16_DRV_GetAutoCalibrationParam(ADC16_INSTANCE, &adcCalibrationParam);
299     ADC16_DRV_SetCalibrationParam(ADC16_INSTANCE, &adcCalibrationParam);
300     #endif // FSL_FEATURE_ADCL6_HAS_CALIBRATION

```

Figure B. 5 Main.c



```

301
302 // Enable BANDGAP reference voltage
303 PMC_HAL_BandgapBufferConfig(PMC_BASE_PTR, &pmcBandgapConfig);
304
305 #if FSL_FEATURE_ADC16_HAS_HW_AVERAGE
306 // Use hardware average to increase stability of the measurement.
307 userHwAverageConfig.hwAverageEnable = true;
308 userHwAverageConfig.hwAverageCountMode = kAdc16HwAverageCountOf32;
309 ADC16_DRV_ConfigHwAverage(ADC16_INSTANCE, &userHwAverageConfig);
310 #endif // FSL_FEATURE_ADC16_HAS_HW_AVERAGE
311
312 // Configure the conversion channel
313 // differential and interrupt mode disable.
314 adcChnConfig.chnIdx = (adc16_chn_t)ADC16_BANDGAP_CHN;
315 #if FSL_FEATURE_ADC16_HAS_DIFF_MODE
316 adcChnConfig.diffConvEnable = false;
317 #endif
318 adcChnConfig.convCompletedIntEnable = false;
319 ADC16_DRV_ConfigConvChn(ADC16_INSTANCE, ADC16_CHN_GROUP, &adcChnConfig);
320
321 // Wait for the conversion to be done
322 ADC16_DRV_WaitConvDone(ADC16_INSTANCE, ADC16_CHN_GROUP);
323
324 // Get current ADC BANDGAP value and format it.
325 bandgapValue = ADC16_DRV_GetConvValueSigned(ADC16_INSTANCE, ADC16_CHN_GROUP);
326 // Calculates bandgapValue in 16bit resolution
327 // from 12bit resolution to calibrate.
328 #if (FSL_FEATURE_ADC16_MAX_RESOLUTION < 16)
329 bandgapValue = bandgapValue << 4;
330 #endif
331 // ADC stop conversion
332 ADC16_DRV_PauseConv(ADC16_INSTANCE, ADC16_CHN_GROUP);
333
334 // Get VDD value measured in mV
335 // VDD = (ADCR_VDD * V_BG) / ADCR_BG
336 vdd = ADCR_VDD * V_BG / bandgapValue;
337 // Calibrate ADCR_TEMP25
338 // ADCR_TEMP25 = ADCR_VDD * V_TEMP25 / VDD
339 adcrTemp25 = ADCR_VDD * V_TEMP25 / vdd;
340 // Calculate conversion value of 100mV.
341 // ADCR_100M = ADCR_VDD * 100 / VDD
342 adcr100m = ADCR_VDD * 100 / vdd;
343
344 // Disable BANDGAP reference voltage
345 pmcBandgapConfig.enable = false;
346 PMC_HAL_BandgapBufferConfig(PMC_BASE_PTR, &pmcBandgapConfig);
347
348 }
349
350 /*!
351 * @brief Calculates the current temperature
352 *
353 * This function calculate temperature used calibrated value as formula in reference manual.
354 *
355 * @param ADC converted value of temperature.
356 * @return current temperature in oC.
357 */
358 double get_current_voltage(uint32_t adcValue)
359 {
360     float dum = 0;

```

Figure B.6 Main.c

```

360     float dum = 0;
361     int lmao = adcValue;
362     float currentTemp = 0;
363     float test = 182;
364     int test1 = 0;
365     test1 = test/100;
366     printf("%d",test1);
367     if(lmao < 0){
368
369         dum = ((-1*adcValue));
370         currentTemp =(3.3 -(((dum)*3.30)/(65535)));
371
372     }
373     else{
374         currentTemp =(((float) adcValue )*3.30)/(65535);
375     }
376     printf("\r\n Voltage %.2lf ", currentTemp);
377     // uint2bcd(currentTemp);
378     return currentTemp;
379 }
380 double ADC16_Measure(void)
381 {
382     adc16_chn_config_t chnConfig;
383
384     // Configure the conversion channel
385     // differential and interrupt mode disable.
386     chnConfig.chnIdx = (adc16_chn_t)ADC16_VOLTAGE_CHN;
387     #if FSL_FEATURE_ADC16_HAS_DIFF_MODE
388     chnConfig.diffConvEnable = false;
389     #endif
390     chnConfig.convCompletedIntEnable = false;
391
392     // Software trigger the conversion.
393     ADC16_DRV_ConfigConvChn(ADC16_INSTANCE, ADC16_CHN_GROUP, &chnConfig);
394
395     // Wait for the conversion to be done.
396     ADC16_DRV_WaitConvDone(ADC16_INSTANCE, ADC16_CHN_GROUP);
397
398     // Fetch the conversion value.
399     adcValue = ADC16_DRV_GetConvValueSigned(ADC16_INSTANCE, ADC16_CHN_GROUP);
400
401     // Show the current temperature value.
402     PRINTF("\r\n ADC converted value: %ld\t", adcValue );
403     // Calculates adcValue in 16bit resolution
404     // from 12bit resolution in order to convert to temperature.
405     #if (FSL_FEATURE_ADC16_MAX_RESOLUTION < 16)
406     adcValue = adcValue << 4;
407     #endif
408     double test1 = get_current_voltage(adcValue);
409     return test1;
410     // Pause the conversion.
411     ADC16_DRV_PauseConv(ADC16_INSTANCE, ADC16_CHN_GROUP);
412 }
413 void DACout(double ADCValue){
414     int ADCValueInt = 0;
415     ADCValueInt = floor((ADCValue * 65535 / 3.30));
416     DAC_DRV_Output(DAC_INSTANCE, ADCValue);
417 }
418

```

Figure B.7 Main.c