Faculty of Engineering

Ain Shams University

2020-2021

Microprocessor Based Systems

CSE 413

MCT 2021

Project Report #: 1

Title: ***Light Dimmer System Code Analysis***

Made by: Team 13 **Section:** 1

Ahmed Magdy Ahmed Bayoumi	16E0211
Ahmed Mohamed Gamal Farag	1600165
Hamdy Osama Mohamed Elsayed	16E0062
Salem Majed Salem Isleem	16E0079

Table of Contents

1. Overview

The project required simulates the effect of a light dimmer module that controls the intensity of a smart home lighting system. To prototype this system, we will be using the FRDM-KL25Z128 NXP board that will be the microcontroller of the system. The board has an ADC Module that will be used to interface a potentiometer used as user input method to control the intensity of the light. To have a variable voltage control over the LED, we will use the DAC module integrated within the microcontroller.

Each module used in the project has its own driver that facilitates the user interface with the microcontroller without referring to the datasheet for convenience. In this report, we will discuss the drivers used in the project as well as the logic behind the main program.

2. Project Drivers

The drivers running the project prototype include:

<u>GPIO Driver</u> -> used to configure each pin on the microcontroller as well as initialize the module it is connected to by the PCR MUX

<u>ADC Driver</u> -> used to configure the ADC0 Module and return captured and converted digital data of the selected ADC pin.

<u>DAC Driver</u> -> used to configure and initialize the DAC0 Module that enables variable voltage output to vary the intensity of the connected LED.

Note that all drivers have 4 files each:

Program.c -> has all the function definitions and logic running the driver.
(user should not alter this file)

Private.h -> has private definitions and values that should not be used by the user.

Interface.h -> has all the functions available for the user declared and a description of how to use them and with what arguments.

Config.h -> has configurable definitions that should be configured as stated in the comments of each definition. If not altered, default configurations are used.

2.1. GPIO Driver

This driver has functions for the following operations:

1- Initialize clock for selected port from SIM SCGC5.
2- Set pin as GPIO Mode from PCR MUX and select Pull configuration.
3- Set pin direction (input/output/Analog).
4- Write a state on an output GPIO Pin.
5- Toggle state of an output GPIO Pin.
6- Read the state of an input GPIO Pin.

```
void MGPIO_voidInit (uint8_t Port); //Enable Port Clock and Config Properties for port selected pins
void MGPIO_voidPinControl(uint8_t Port, uint8_t Pin, uint8_t Pull); //Enable Pin GPIO Mode + Pull up/down/off
void MGPIO_voidPinDirection (uint8_t Port, uint8_t Pin, uint8_t Directon); //Input (Analog/Dgital) or Output Mode (Digital)
void MGPIO_voidPinDigitalWrite (uint8_t Port, uint8_t Pin, uint8_t State); //Write Output Value
void MGPIO_voidPinToggle(uint8_t Port, uint8_t Pin);//Toggle pin output
uint8_t MGPIO_u8PinDigitalRead (uint8_t Port, uint8_t Pin); //Read Input Value (Digital)
```

```
//u8 Port Macros
#define  A       9
#define  B       10
#define  C       11
#define  D       12
#define  E       13

//u8 Pin Macros
// Enter numbers from 0 to 31 based on pin number on mcu

//u8 Pull Macros
#define PULL_UP      1
#define PULL_DOWN    0
#define PULL_OFF     2

//u8 Direction Macros
#define INPUT        0
#define OUTPUT       1
#define ANALOG       2

//U8 State Macros
#define HIGH         1
#define LOW          0
```

Note about Analog configuration -> it sets the PCR MUX to 000 (analog mode) and sets pull configuration as PULL_OFF, so the function MGPIO_voidPinControl() usage is redundant as it will be overwritten by setting the voidPinDirection as ANALOG.

2.2. ADC Driver

This driver has functions that does the following:

1- Initialize the ADC0 clock from SIM SCGC6 as well as initialize configurations stated in the config file. The configurations available include:

```
/*Configurations available for the user to modify to get required ADC Performance

  1- Interrupt Enabling           (0/1)
  2- Differential Mode            (0/1)
  3- Low Power Mode               (0/1)
  4- Clock Prescaler              (1/2/4/8)
  5- Sample Time                  (ADC_SHORT / ADC_LONG)
  6- Resolution                   (8/10/12/16)
  7- ADC MUX Channels             (ADC_A / ADC_B)
  8- Conversion Trigger           (ADC_SW / ADC_HW)
  9- DMA Enable                   (0/1)
 10- Continuous Conversion        (0/1)
*/
```

2- Start ADC Conversion by writing the channel to be read in SC1 register and return the converted digital value after conversion complete flag is set.

```
void ADC_voidInit(uint8_t ADC_Module);      //Initialize selected ADC Module with configurations made in config file
int16_t ADC_u16PinRead(uint8_t ADC_Pin);   //Get data from analog pin (after successful conversion)

/*Note: this read function does not support 2 successive reads from SC1A, SC1B
For successive reads, call function twice or enable continuous conversion in config file
Utilizing A,B SC1 Registers will be added in a future version*/
```

```
//ADC_Module Options   //FRDM MKL25Z4 Board has ADC0 only
#define ADC_0      0

//ADC_Pin Options
/*NOTE:
    The following definitions are critical to successful mapping
    between board pins and ADC Module channels.
    PLEASE DO NOT MODIFY THE FOLLOWING DEFINITIONS
*/
#define ADC_PE20    0
#define ADC_PE22    3
#define ADC_PE21    4   //When ADC_A is configured only
#define ADC_PB29    4   //When ADC_B is configured only
#define ADC_PD1     5   //When ADC_B is configured only
#define ADC_PD5     6   //When ADC_B is configured only
#define ADC_PE23    7   //When ADC_A is configured only
#define ADC_PD6     7   //When ADC_B is configured only
#define ADC_PB0     8
#define ADC_PB1     9
#define ADC_PC2     11
#define ADC_PB2     12
#define ADC_PB3     13
#define ADC_PC0     14
#define ADC_PC1     15
#define ADC_PE30    23

#define ADC_TEMP    26
```

Note that the user is only required to specify the pin he is using as analog input only, while the driver handles the channel to be opened that corresponds to the pin the user specifies. This makes it easier for the user to use the ADC Module without knowing the mapping between driver channels and GPIO Pins.

2.3. DAC Driver

This driver has functions that performs the following:

1- Initialize DAC0 Clock from SIM SCGC6 as well as port E clock from SIM SCGC5; then set MUX of PCR[30] of port E as analog (000) to set the pin as DAC pin. Enable configuration options of the DAC Module found in config file including the following:

```
/*This file contains all the configurations possible for main DAC activation
  Future updates to the driver would include stand-alone functions that enables
  these features for multiple DACs simultaneously
*/
/*Version 1 available features:
    1- DAC Reference Select                       (VREFH / VDDA)
    2- DAC Buffer Trigger Select                     (HW / SW)
    3- DAC Power Mode                                (HIGH / LOW)
    4- DAC Interrupt Enable (top and bottom)          (0 / 1)*
    5- DAC DMA Enable                                 (0 / 1)*
    6- DAC Buffer Enable                              (0 / 1)
    7- DAC Buffer Mode                           (NORMAL / ONE_SCAN)
    8- DAC Buffer Upper Limit                         (0 / 1)
 * -> Do not enable DMA and Interrupts simultaneously
 */
```

2- Set the DAC value output to correspond to a voltage level output on pin 30 port E.
3- Disable the DAC Module (Pin 30 Port E is floating).
4- 4- Get the current value set on the DAC Output. (can be used in verification)

```
void DAC_voidInit(uint8_t DAC_number);
//Used to initialize the DAC Module used. for FRDMKL25Z -> DAC_0 is used
void DAC_voidSetOutput(uint8_t DAC_number,uint16_t DAC_value);
//Used to set output voltage on DAC Module... DAC_value from 0 to 4095 indicates voltage level
void DAC_voidDisableDAC(uint8_t DAC_number);
//Close DAC Module if not used to save power
uint16_t DAC_u16GetDACvalue(uint8_t DAC_number);
//Return the value (0 to 4095) output found currently on DAC Module


//DAC_number Options
#define DAC_0    0

//DAC_value Options
// FROM 0 TO 4095
```

3. Main Program flow

```
#include <MKL25Z4.h>            //Include the header file of the MCU Pins
#include "DAC_interface.h"      //Include the DAC Driver
#include "ADC_interface.h"      //Include the ADC Driver
#include "GPIO_interface.h"     //Include the GPIO Driver

int main(){
  DAC_voidInit(DAC_0);   //Initialize DAC0 Module (Pin PE30 as output for LED Included)

  ADC_voidInit(ADC_0);       //Initialize ADC0 Module

  //Initializing a GPIO Pin to be Analog (Potentiometer input)
  MGPIO_voidInit(C);
  MGPIO_voidPinDirection(C, 2, ANALOG);

  volatile uint16_t value = 0;  //Variable to store analog read and be used in DAC input
  while(1){
  value = ADC_u16PinRead(ADC_PC2);   //Reading analog pin
  //value = value * 4095/65335;        //use this formula if the ADC is 16 bit resolution to scale down the value to suitable range for DAC
  DAC_voidSetOutput(DAC_0,value);    //Setting DAC Output to a voltage level equal to that read on analog pin
  }
}
```

3.1. Initializing LED Output Pin (DAC)

Necessary steps to set pin 30 on port E as output (DAC not GPIO)

1- SIM SCGC5 to enable port E clock.
2- SIM SCGC6 to enable DAC0 clock.
3- PORTE->PCR [30] MUX as 000 for analog mode (DAC).
4- DAC0 -> C0 to set voltage reference selection and trigger source.
5- Set data registers (Low and High) to the corresponding voltage level to be written on pin 30 port E.
6- Enable DAC Module from DAC0 -> C0.

```
void DAC_voidInit(uint8_t DAC_number){
  //Check DAC_Number
  if (DAC_number==DAC_0){
    SIM->SCGC6 |= (1<<31);                       // Clock activation for DAC0
    SIM->SCGC5 |= (1<<13);                       // Clock activation for PORTE
    PORTE->PCR[30] &=~ (7<<8);                   // Pin 30 on PORTE is analog

    //Activate configuration options of DAC Module
    DAC0->C0 = 0;                               //Reset C0 register
    DAC0->C0 |= (DAC_REF<<6);                   //Set reference voltage source
    DAC0->C0 |= (DAC_BUFFER_TRIGGER<<5);        //Set trigger source
    DAC0->C0 |= (DAC_POWER<<3);                 //Set DAC Power Mode
    DAC0->C0 |= (DAC_TOP_INTERRUPT<<1);         //Set DAC Top Interrupt
    DAC0->C0 |= (DAC_BOTTOM_INTERRUPT<<0);      //Set DAC Bottom Interrupt

    DAC0->C1 = 0;                               //Reset C1 register
    DAC0->C1 |= (DAC_DMA<<7);                   //Set DMA configuration
    DAC0->C1 |= (DAC_BUFFER<<0);                //Set Buffer Configuration
    DAC0->C1 |= (DAC_BUFFER_MODE<<2);           //Set Buffer Mode

    DAC0->C2 = 0;                               //Reset C2 register
    DAC0->C2 |= (DAC_BUFFER_UPPER_LIMIT<<0);    //Set Upper limit of buffer pointer

    //Reset data registers
    DAC0->DAT[0].DATL=0;
    DAC0->DAT[0].DATH =0;
```

3.2. Initializing Analog Input Pin for potentiometer

Any analog pin found in the interface file of the ADC Driver can be used here. In this code, pin 2 on port C is used.

Necessary steps to use an analog input pin:

1- SIM SCGC5 to enable port C clock (found in MGPIO_voidInit function)
2- PORTC->PCR [2] MUX is set to 000 (Analog) and Pull Mode is Disabled (floating) (found in MGPIO_voidPinDirection function)
3- Initializing ADC0 Module clock from SIM SCGC6. (The rest of the configurations can be skipped if the defaults will be used (8bit resolution)

```
void ADC_voidInit(uint8_t ADC_Module){
  if (ADC_Module == ADC_0){    //Check for ADC Module

    SIM ->SCGC6 |= (1<<27);    //Clock Enabling for ADC0 Module

    // Set Configurations of the selected ADC Module
    ADC0 -> SC1[0] = ADC_OFF;  //Reset SC1A Register (and disable ADC)
    ADC0 -> SC1[0] |=
    (ADC_INTERRUPT<<6)|        //Interrupt Configuration
    (ADC_DIFF_MODE<<5);        //Single Ended / Differential Configuration

    ADC0 -> SC1[1] = ADC_OFF;  //Reset SC1B Register (and disable ADC)
    ADC0 -> SC1[1] |=
    (ADC_INTERRUPT<<6)|        //Interrupt Configuration
    (ADC_DIFF_MODE<<5);        //Single Ended / Differential Configuration

    ADC0 -> CFG1 = 0;          //Reset CFG1 Register
    ADC0 -> CFG1 |=
    (ADC_LOW_POW<<7)|          //Low Power Mode Configuration
    (ADC_SAMPLE<< 4);          //ADC Sampling Time configuration

    //Setting clock divider
    #if (ADC_CLK_DIV==1)
      ADC0 -> CFG1 |= ADC_CLK_1;
    #elif (ADC_CLK_DIV==2)
      ADC0 -> CFG1 |= ADC_CLK_2;
    #elif (ADC_CLK_DIV==4)
      ADC0 -> CFG1 |= ADC_CLK_4;
    #elif (ADC_CLK_DIV==8)
      ADC0 -> CFG1 |= ADC_CLK_8;
    #endif

    //Setting ADC Resolution
    #if (ADC_RESOLUTION==8)
      ADC0 -> CFG1 |= ADC_RES_8;
    #elif (ADC_RESOLUTION==10)
      ADC0 -> CFG1 |= ADC_RES_10;
    #elif (ADC_RESOLUTION==12)
      ADC0 -> CFG1 |= ADC_RES_12;
    #elif (ADC_RESOLUTION==16)
      ADC0 -> CFG1 |= ADC_RES_16;
    #endif

    ADC0 -> CFG2 = 0;         //Reset CFG2 Register
```

All macro definitions that the user uses to configure settings in config file can be found in the private file of the driver. Macro definitions should not be altered by the user as they are directly used to manipulate module registers as seen in the above code. (All project files are attached with the report assignment)

3.3. While loop logic

```
volatile uint16_t value = 0;  //Variable to store analog read and be used in DAC input
while(1){
value = ADC_u16PinRead(ADC_PC2);    //Reading analog pin
//value = value * 4095/65335;         //use this formula if the ADC is 16 bit resolution to scale down the value to suitable range for DAC
DAC_voidSetOutput(DAC_0,value);    //Setting DAC Output to a voltage level equal to that read on analog pin
}
```

A variable is used to store the value read from the ADC Module on the potentiometer pin (Pin 2 Port C).

This variable is then scaled to match the range of the DAC Module. (in the above code, the ADC is running at 12bit resolution as the DAC module, so no scaling is required, and the line is commented out).

After taking the value of RA register from ADC0 and storing it in variable “value”, the variable is used to write its content into the data registers of the DAC module using the function DAAC_voidSetOutput. This loop is repeated indefinitely in the while (1) loop.

Another approach would be to utilize the interrupts of both the ADC module and DAC module and keep calling each other out, or by using the ADC in continuous mode and use its interrupt handler to write the ADC RA value into the DAC data lines. Both approaches are viable but would keep the MCU processor busy like polling as they keep pending interrupts continuously. A novel approach would be to activate the ADC only if its value changed to free the processor to other tasks in the system, but because this prototype only involves dimming a LED, the current polling method is accepted.

```
void DAC_voidSetOutput(uint8_t DAC_number,uint16_t DAC_value){
  //Check DAC_Number
  if (DAC_number==DAC_0){
    DAC0->DAT[0].DATL = (DAC_value&0xFF);        //Write lowest 8 bits of dac output value
    DAC0->DAT[0].DATH = ((DAC_value>>8)&0xF);    //Write highest 4 bits of dac output value

    DAC0->C0 |= (1<<7);                          //Enable DAC Module
  }
  else{
    //Error
  }
}

int16_t ADC_u16PinRead(uint8_t ADC_Pin){
  if (ADC_Pin<24 || ADC_Pin==26){                                        //Check for input channel
    ADC0 -> SC1[0] =(ADC_INTERRUPT<<6)|(ADC_DIFF_MODE<<5)|ADC_Pin;      //Begin Conversion of selected channel
    while (COCO_BIT != 1){                                             //Wait till conversion is complete

    }
    return (ADC0 -> R[0]);                                             //Return converted data (This clears COCO)
  }
  else{
    //Error
    return -1;
  }
}
```