ELEC422

Microprocessor Systems
Assignment 1
Microprocessor Systems Development with MBED and Kinetis SDK

Khanthapak Thaipakdee

Department of Electrical Engineering and Electronics,

University of Liverpool,

Brownlow Hill, Liverpool L69 3GJ, UK

Table of Contents

Contents **Page**

Table of Contents (Cont.)

Contents **Page**

List of Figures

Figure **Page**

1. Introduction

This project consists of three parts, with Part A and Part B aimed at enhancing skills in embedded systems programming and assembly language proficiency on the FRDM-K64F development board using the MBED platform. In Part A, the task is to develop a program that interacts with the Application Shield to display the user's name and ID on an LCD screen. Additionally, the program should allow the user to increment or decrement a displayed number using cursor keys. Part B focuses on improving assembly language programming for the Cortex M4 processor by implementing two subroutines to calculate the square root of a positive integer. These subroutines will be integrated into the Part A code, with one utilizing the Floating-Point Unit (FPU) and the other not. Performance comparisons will be made between the two implementations to determine the faster method.

While Part A and Part B concentrate on MBED libraries, Part C introduces the implementation of peripheral access using the Manufacturers' provided Kinetis Software Development Kit (KSDK). This section focuses on reading values from the analogue-to-digital converter (ADC) inputs connected to Pot 1 and Pot 2 of the Application Shield and transmitting these values to a PC via the serial port for display in a terminal program. Pot 1's value is sampled at a user-defined rate, while Pot 2's value is sampled upon pressing the joystick up. This part of the assignment emphasizes the importance of leveraging manufacturer-provided Freescale libraries for efficient peripheral access and communication in embedded systems development.

2. Top-down structure diagram: Part A & Part B

The top-down structure of the program is illustrated in Figure 1, showcasing the relationships between key components. At its core lies the main program, which interacts with modules such as DigitalIn, LCD, Timer, Printf, Sleep, and random. These modules encapsulate functionalities such as reading digital inputs, displaying information on the LCD screen, managing timing operations, printing formatted output, enabling sleep functionality, and generating random numbers, respectively.

Additionally, the program interfaces with assembly language functions represented by simple_sqrt assembly and square_root_fpu assembly, each serving specific square root computation purposes. Data and control flow between these components, facilitating the execution of tasks within the program. Overall, this structure organizes the program's architecture, delineating the roles and relationships of its constituent elements.

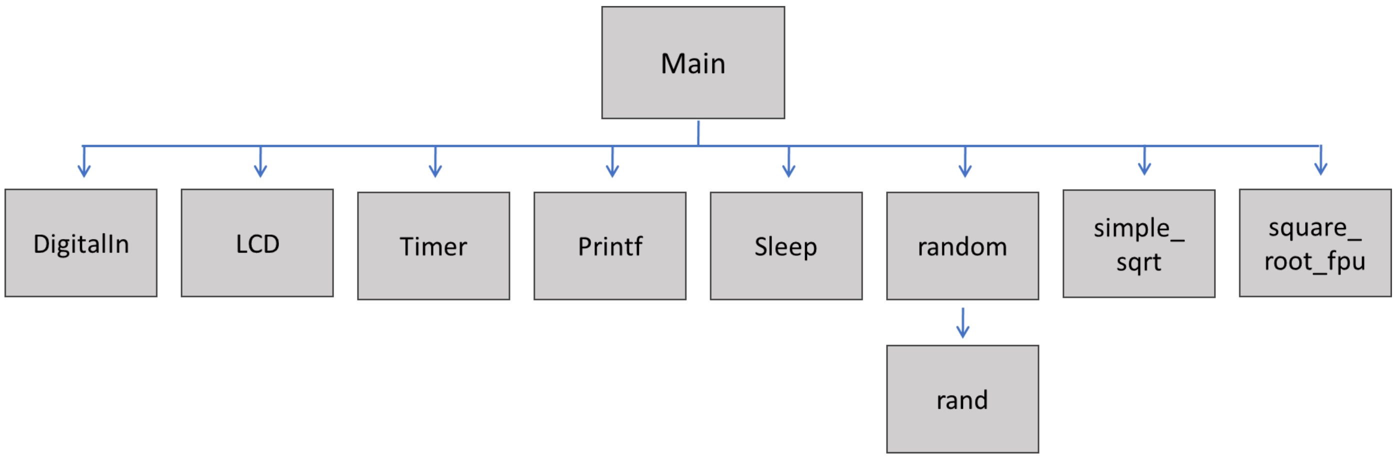

Figure 1 Top-down structure diagram: Part A & Part B

3. Flow charts of overall system: Part A & Part B

The flowchart (Figure 2) delineates the sequence of actions within the program. It commences with the inclusion of all necessary header files and proceeds to initialize the DigitalIn and LCD display components. Following this, the external assembly language function is declared, and a random function is created.

In the main program, a while loop ensures continuous program execution. Within this loop, the program utilizes the "read" function from the DigitalIn class to ascertain the pin status. Initially, the program checks the status of the SW_UP button. If the button is pressed, indicating a change in the status of pin A2, the program invokes the random function to generate a random number based on the current ID. This number is subsequently added to the current ID to yield the new ID. Additionally, if the new ID exceeds 999,999,999, the ID is set to 999,999,999, and a message "Value cannot > 999999999" is displayed on the LCD display. The program then pauses for 10 seconds to allow the user to view the message before clearing the LCD screen. If the new ID does not exceed 999,999,999, no action is taken.

The program continues to check for the status of the SW_DOWN button in both situations. If the SW_DOWN button is pressed, causing a change in the status of pin A3, the program follows a similar process. The random function is invoked to generate a random number based on the current ID, which is then subtracted from the current ID to obtain the new ID. Additionally, if the new ID is lower than 1, indicating an invalid value, the ID is set to 1, and a message "Value cannot < 1" is displayed on the LCD display. The program then pauses for 10 seconds to allow the user to view the message before clearing the LCD screen. If the new ID is not less than 1, no action is taken. The program continues to check for the status of the left button in both situations.

For the left button, if it's pressed, the program calls the simple_sqrt function with the current ID value as the input parameter, returning the square root of the ID to the sqrt_val variable. Subsequently, the program checks if the right button is pressed. If so, it invokes the square_root_fpu function, which utilizes the floating-point unit to calculate the square root of the current ID. The result is then assigned to the sqrt_val variable.

After checking all switches, the program proceeds to locate the LCD cursor at position 0,0 and displays the name “Khanthapak Thaipakdee”. Then, it locates the cursor at position 0,10 to display the current ID in the form “ID: current ID value”, and at position 0,20 to display the square root of the ID from the sqrt_val variable in the form "Square root of ID is sqrt_val”. Finally, the program loops back to check for switch presses, ensuring continuous operation.

To measure the execution time of the simple_sqrt function and square_root_fpu function, a timer object is added at the top of the program. Before calling each function, the timer is started using the start method. After execution, the timer is stopped using the stop method. The elapsed time is then displayed on the serial port, enabling performance analysis. Finally, the timer is reset using the reset method for subsequent timing analysis. This approach is depicted by the green box in Figure 2.

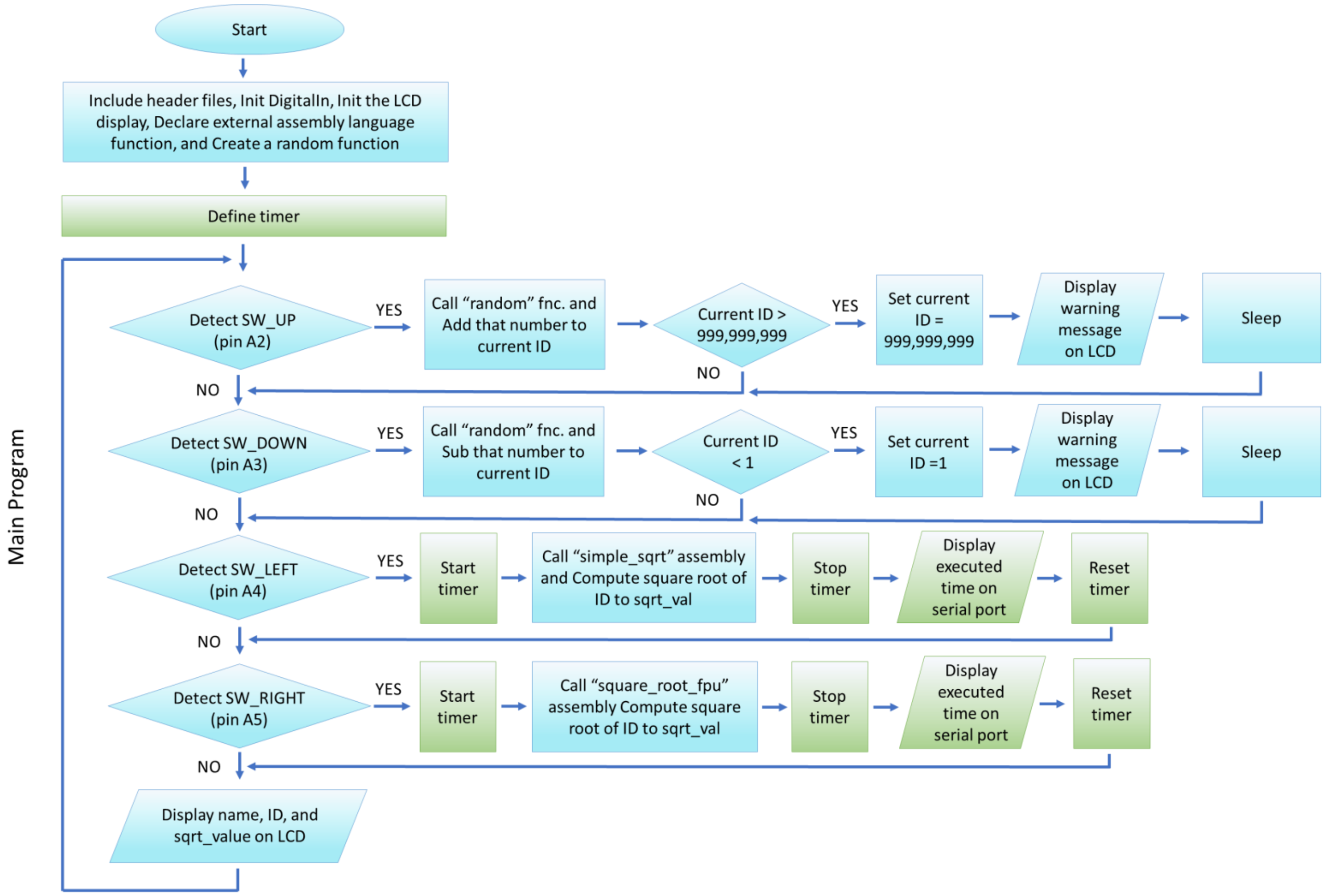

Figure 2 Flow chart of overall system: Part A & Part B

Figure 3 illustrates the flowchart of the random function, detailing the process of generating pseudo-random numbers within a specified range. The function accepts an integer input parameter and employs conditional logic to categorize the input into distinct ranges. Within each range, a random number is generated using the rand() function or a similar mechanism. This approach ensures that the output adheres to the desired range while providing variation within that range. By employing if-else statements or similar constructs, the code iterates through different conditions based on the input value, enabling the generation of random numbers tailored to specific requirements.

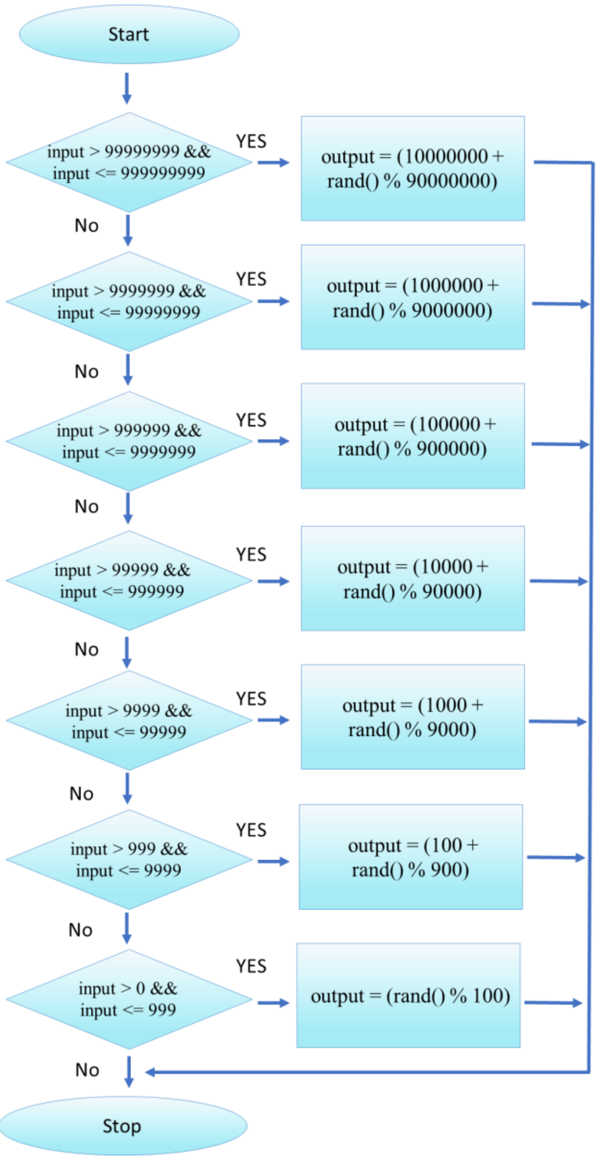

Figure 3 The flowchart of random function

4. Flow chart of simple_sqrt function

Figure 4 indicates flowchart of simple_sqrt function. The algorithm initializes two registers, R1 and R2, with predefined values, where R1 holds 0x8000 (32768) and R2 is initialized to 0. It then enters a loop where R2 is iteratively adjusted based on comparisons with the input value in R0. Within each iteration, R2 is incremented or decremented by the value in R1, and the result is squared and compared with the input value. If the squared value exceeds the input, R2 is adjusted accordingly, and R1 is halved. This process continues until R1 becomes 0. Finally, the resulting approximation of the square root is stored in R0, and the function returns.

Table 1 illustrates the values of R1, R2, and R3 in each iteration of the program. The values denoted by R1* and R2* signify the initial values of R1 and R2 at the start of each iteration, while R1# and R2# indicate the updated values of R1 and R2 at the end of each loop. In this scenario, the input is assumed to be 122 (0x007A). The program attempts to find the largest number by iteratively changing bits using logical right shift operations, such that the square is less than the input value (R0), which is 8 (0x0008) in loop 13.

Then it tries to use a smaller mask to increase the value of R2. As you can see, 4 (which is half of 8) is added to R2, making R2 equal to 12 (0x000C). However, 144 (0x0090) is greater than 122, so R2 returns to its previous value, which is 8 (as seen in loop 14). Then it tries to use a smaller mask than 4 (which is 2). 2 is added to R2, so R2 equals 10 (0x000A), and 100 (0x0064) is less than 122, so R2 is updated to 10.

After that, it tries to use a smaller mask than 2 (which is 1). 1 is added to R2, so R2 equals 11 (0x00B). 121 (0x0079) is smaller than 122, so it tries to use a smaller mask than 1 (which we don't have any smaller integer than 1). The mask (R0) is then equal to 0, and the loop exits. R2's stored value is 11.

The return value from R2 to R0 is then sent to the C code. Hence, the approximate square root of 122 is 11. Additionally, when the condition of R3 > R0 is false, as indicated by the red square, R2 at the end of the loop is updated, which is consistent with the behavior shown in the flowchart.

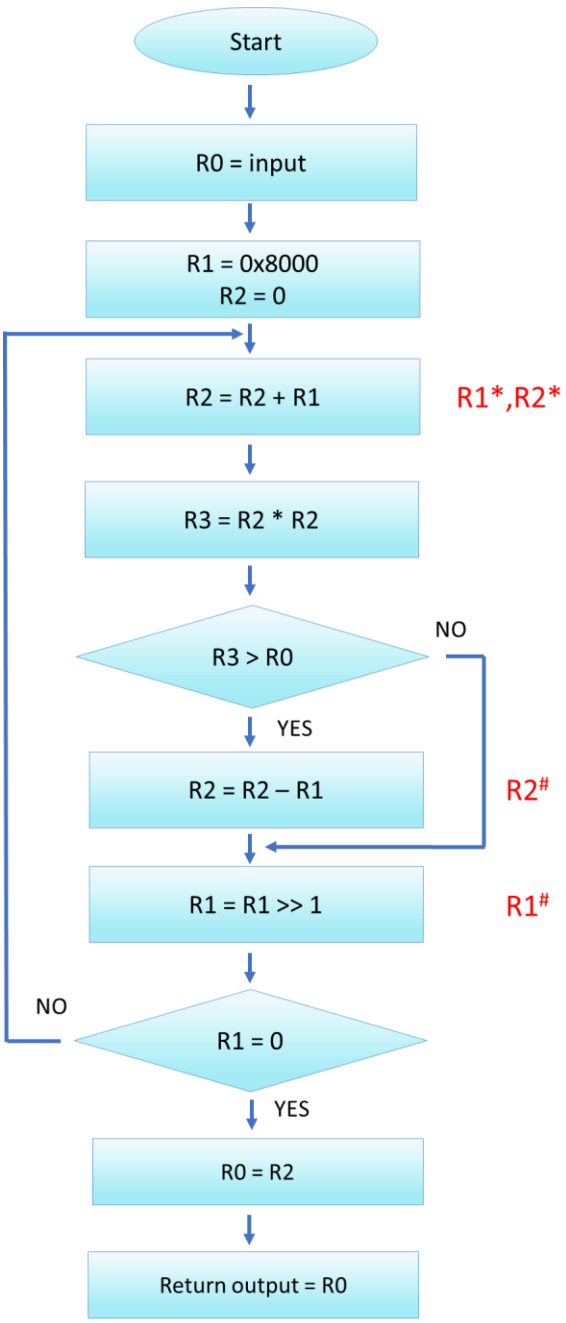

Figure 4 Flow chart of simple_sqrt function

Table 1 Table represent value of R1, R2, and R3 in simple_sqrt function

R0 = 0x007A	R1*	R2*	R3	Condition (R3 > R0)	R2[#]	R1[#]
Loop 1	0x8000	0x8000	0x40000000	(Yes)	0	0x4000
Loop 2	0x4000	0x4000	0x10000000	(Yes)	0	0x2000
Loop 3	0x2000	0x2000	0x4000000	(Yes)	0	0x1000
Loop 4	0x1000	0x1000	0x1000000	(Yes)	0	0x0800
Loop 5	0x0800	0x0800	0x400000	(Yes)	0	0x0400
Loop 6	0x0400	0x0400	0x100000	(Yes)	0	0x0200
Loop 7	0x0200	0x0200	0x40000	(Yes)	0	0x0100
Loop 8	0x0100	0x0100	0x10000	(Yes)	0	0x0080
Loop 9	0x0080	0x0080	0x4000	(Yes)	0	0x0040
Loop 10	0x0040	0x0040	0x1000	(Yes)	0	0x0020
Loop 11	0x0020	0x0020	0x0400	(Yes)	0	0x0010
Loop 12	0x0010	0x0010	0x0100	(Yes)	0	0x0008
Loop 13	0x0008	0x0008	0x0040	(No)	0x0008	0x0004
Loop 14	0x0004	0x000C	0x0090	(Yes)	0x0008	0x0002
Loop 15	0x0002	0x000A	0x0064	(No)	0x000A	0x0001
Loop 16	0x0001	0x000B	0x0079	(No)	0x000B	0x0000

5. Description of each module:
Part A & Part B

5.1 DigitalIn

The code is setting up digital inputs for switches connected to pins A2, A3, A4, and A5, and then reading the status of these switches. The pin name of each switch is the pin name on the Arduino R3 on the application shield, as shown in the Figure 5. Each DigitalIn object represents a digital input pin that can be read as either high (1) or low (0), indicating whether the switch connected to that pin is pressed (high) or not pressed (low). The status of these switches will be used in conditional statements after reading their status to determine what action the program should take, as outlined in the provided flowchart.

If the SW_UP switch is pressed, the 'id' variable is incremented by a random value based on its current value. If the incremented 'id' exceeds 999999999, it is set to 999999999, and a message indicating that the value cannot exceed this limit is displayed on the LCD screen for 10 seconds before clearing the screen.

If the SW_DOWN switch is pressed, the 'id' variable is decremented by a random value based on its current value. If the decremented 'id' becomes negative, it is set to 1, and a message indicating that the value cannot be less than 1 is displayed on the LCD screen for 10 seconds before clearing the screen.

If the SW_LEFT switch is pressed, the square root of the 'id' variable is calculated using the 'simple_sqrt' function.

If the SW_RIGHT switch is pressed, the square root of the 'id' variable is calculated using the 'square_root_fpu' function.

```
// Define digital inputs for switches connected
to pins A2, A3, A4, and A5
DigitalIn SW_UP(A2);
DigitalIn SW_DOWN(A3);
DigitalIn SW_LEFT(A4);
DigitalIn SW_RIGHT(A5);

// Read the status of each switch and use that
status for condition

if (SW_UP.read()){
    id = id + random(id);
    lcd.cls();
            if (id > 999999999){
               id = 999999999;
               lcd.locate(0,20);
               lcd.printf("Value cannot >
999999999");
               ThisThread::sleep_for(10s);
               lcd.cls();
            }
}

if (SW_DOWN.read()){
    id = id - (random(id));
    lcd.cls();
            if (id < 0){
               id = 1;
               lcd.locate(0,20);
               lcd.printf("Value cannot < 1");
               ThisThread::sleep_for(10s);
               lcd.cls();
            }
}

if (SW_LEFT.read()){
    sqrt_val = simple_sqrt(id);
}

if (SW_RIGHT.read(){
    sqrt_val = square_root_fpu(id);
}
```

5.2 LCD

The provided code snippet includes a header file, initializes the LCD display, clears it, locate and prints a text at specified location.

#include "C12832.h": This line includes a header file named "C12832.h" which likely contains declarations necessary for working with the C12832 LCD display.

The line C12832 lcd(D11, D13, D12, D7, D10); initializes an object named lcd of type C12832, which represents the LCD display. This object constructor requires parameters in the form C12832 (PinName mosi, PinName sck, PinName reset, PinName a0, PinName cs). Referring to Figure 5, we can see that D11 corresponds to the MOSI pin, D13 corresponds to the SCK pin, D12 corresponds to the reset pin, D7 corresponds to the A0 pin, and D10 corresponds to the chip select (CS) pin. Therefore, these parameters specify the pin connections necessary for communication with the LCD display.

lcd.cls();: This function clears the display, effectively wiping out any content previously displayed on it.

lcd.locate();: The locate() function sets the cursor position on the display where the next text will be printed.

lcd.printf();: The printf() function is used to print formatted text onto the LCD display.

```
#include "C12832.h"

// Initialize the LCD display with the specified pins
C12832 lcd(D11, D13, D12, D7, D10)

lcd.cls();

int id = 201750406;

// Set the cursor position to (0,0) and print the name "Khanthapak Thaipakdee"
lcd.locate(0,0);
lcd.printf("Khanthapak Thaipakdee");

// Set the cursor position to (0,10) and print the text "ID: %d" followed by the value of 'id'
lcd.locate(0,10);
lcd.printf("ID: %d", id);
```

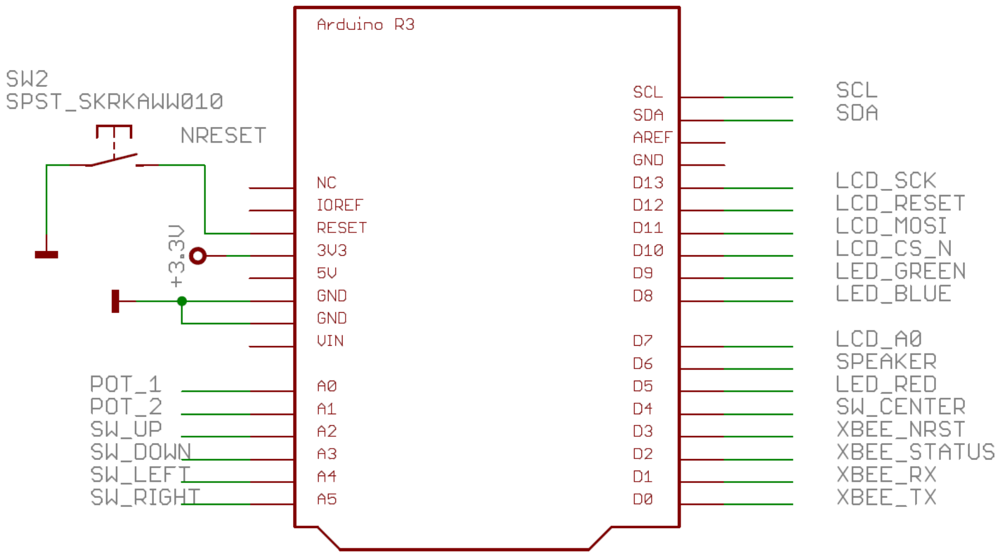

Figure 5 Arduino R3 Pin Configuration for Application Shield.

5.3 Timer

In this code snippet, a timer object named timer1 is defined to measure the execution time of a computational operation represented by the function simple_sqrt(x). The timer is started before the computation begins and stopped afterward to capture the elapsed time. The duration of the computation is then extracted using the elapsed_time() function, and the result is stored in the variable simple_sqrt_time. Finally, the timer is reset, preparing it for potential future use. This code segment enables the evaluation of the performance of the simple_sqrt function by quantifying the time taken for its execution. Additionally, this coding approach can be extended to measure the execution time of the square_root_fpu function, facilitating a comparative analysis of performance, a discussion of which will follow.

```
Timer timer1; // Define a timer object named
timer1

timer1.start(); // Start the timer

sqrt_val = simple_sqrt(id); // Calculate the
square root of id

timer1.stop(); // Stop the timer after the
computation is finished

// Get the elapsed time from the timer and store
it in the variable simple_sqrt_time

simple_sqrt_time =
timer1.elapsed_time().count();

timer1.reset(); // Reset the timer for potential
future use
```

5.4 Printf

In Mbed-5, when using printf for serial communication (UART), the serial port and associated pins need to be defined explicitly. This includes specifying the UART_TX and UART_RX pins for communication. However, in Mbed-6, the default "printf" port is predefined, allowing for standard C/C++ usage, such as printf("Started\n\r"). It means that in Mbed-6, printf can be directly used for serial communication without specifying the UART port.

The printf function is commonly used for debugging purposes, allowing developers to output messages to a serial terminal for monitoring program behavior or diagnosing issues. The default baud rate for UART communication in the Mbed library is typically set to 9600 bits per second (bps).

5.5 Sleep

The line ThisThread::sleep_for(); refers a sleep function call within the Mbed OS environment, directing the current thread to pause or sleep for a specified duration of time. Specifically, when ThisThread::sleep_for(10s); is executed within a program, the thread halts execution for a period of 10 seconds before proceeding with subsequent instructions.

5.6 random

The random function takes an integer input parameter and generates a pseudo-random integer based on the input value. It categorizes the input into different ranges and assigns a random number within each range.

For input values between 100 million and 999 million, it generates a random number between 10 million and 100 million.

For input values between 10 million and 99 million, it generates a random number between 1 million and 10 million.

For input values between 1 million and 9.9 million, it generates a random number between 100 thousand and 1 million.

For input values between 100 thousand and 999 thousand, it generates a random number between 10 thousand and 100 thousand.

For input values between 10 thousand and 99 thousand, it generates a random number between 1 thousand and 10 thousand.

For input values between 1 thousand and 9.9 thousand, it generates a random number between 100 and 1 thousand.

For input values between 1 and 999, it generates a random number between 0 and 99.

The function utilizes the rand() function along with modulo operations to limit the generated numbers within the predefined ranges. Importantly, it's designed to maintain a specific constraint: if the input number comprises 8 digits, the resulting random number will consist of 7 digits. This design facilitates the identification of recommended numbers for quick increment and decrement operations, which we'll discuss later.

```
int random(int input)
{
   int output = 0;

   if (input > 99999999 && input <= 999999999){
      output = (10000000 + rand() % 90000000);
   }
   else if (input > 9999999 && input <= 99999999){
      output = (1000000 + rand() % 9000000);
   }
   else if (input > 999999 && input <= 9999999){
      output = (100000 + rand() % 900000);
   }
   else if (input > 99999 && input <= 999999){
      output = (10000 + rand() % 90000);
   }
   else if (input > 9999 && input <= 99999){
      output = (1000 + rand() % 9000);
   }
   else if (input > 999 && input <= 9999){
      output = (100 + rand() % 900);
   }
   else if (input > 0 && input <= 999){
      output = (rand() % 100);
   }


   return output;

}
```

5.7 simple_sqrt

This code snippet declares an external assembly language function named simple_sqrt, which is defined in a separate assembly file with a .s extension. The function takes an integer parameter value and returns an integer result, presumably the square root of the input value. The extern "C" declaration ensures that the function uses C linkage, allowing the C++ compiler to properly interface with the assembly function.

Later in the code, the function simple_sqrt is called with an integer argument "id", and the result is assigned to the variable "sqrt_val". This means that the square root of the "id" value stored in "sqrt_val" is calculated using the assembly language implementation provided in the external file.

```
// declare external assembly language function
(in a *.s file)
extern "C" int simple_sqrt(int value);

sqrt_val = simple_sqrt(id);
```

```
   AREA asm_func, CODE, READONLY

; Export my asm function location so that C
compiler can find it and link
   EXPORT simple_sqrt

   ALIGN

simple_sqrt FUNCTION
   ; Input : R0
   ; Output : R0 (square root result)
   MOVW R1, #0x8000 ; R1 = 0x00008000
   MOVS R2, #0 ; Initialize result

simple_sqrt_loop
   ADDS R2, R2, R1 ; M = (M + N)
   MUL R3, R2, R2 ; R3 = M^2
   CMP R3, R0 ; If M^2 > Input
   IT HI ; Greater Than
   SUBHI R2, R2, R1 ; M = (M - N)
   LSRS R1, R1, #1 ; N = N >> 1
   BNE simple_sqrt_loop
   MOV R0, R2 ; Copy to R0 and return
   BX LR ; Return
ENDFUNC
```

5.8 square_root_fpu

The provided assembly code defines a function named square_root_fpu, designed to compute the square root of an input value using the Floating-Point Unit (FPU). Initially, the code enables the FPU by configuring the Control Access Port Control Register (CPACR). It then loads the input value from register r0 into a single-precision floating-point register s0. Subsequently, it converts the unsigned 32-bit integer in s0 to a single-precision floating-point number, calculates the square root of the floating-point number, and converts it back to an unsigned 32-bit integer. Finally, it moves the result from floating-point register s0 back to general-purpose register r0 and returns from the function. This code provides a hardware-accelerated method for computing square roots using the FPU, enhancing efficiency compared to software-based approaches.

Later in the code, the function square_root_fpu is called with an integer argument "id", and the result is assigned to the variable "sqrt_val". This means that the square root of the "id" value stored in "sqrt_val" is calculated using the assembly language implementation provided in the external file.

```
// declare external assembly language function
(in a *.s file)
extern "C" int square_root_fpu(int value);

sqrt_val = square_root_fpu(id);
```

```
   AREA asm_func, CODE, READONLY
   THUMB

   ; Export my asm function location so that C
compiler can find it and link
   EXPORT square_root_fpu

square_root_fpu FUNCTION

   ; Enable FPU
   LDR r1, =0xE000ED88

   ; Read CPACR
   LDR r2, [r1]
   ORR r2, r2, #(0xF << 20)
   STR r2, [r1]
   DSB
   ISB
   VMOV s0, r0
   VCVT.F32.U32 s0, s0
   VSQRT.F32 s0, s0
   VCVT.U32.F32 s0, s0
   VMOV r0, s0
   BX LR

   ENDFUNC
   END
```

6. Testing scheme: Part A & Part B

- random function

The provided code snippet utilizes the printf function to test the random function, showcasing its behavior with various input numbers. Each printf statement displays two values: the input number x and the corresponding random number generated by the random function with the same input. Starting with large input values such as 999999999 and gradually decreasing to smaller values like 1, the code evaluates how the random function behaves across different ranges.

```
  printf("\n x= %d random =
%d\n",999999999, random(999999999));
   printf("\n x= %d random =
%d\n",100000000, random(100000000));
   printf("\n x= %d random = %d\n",10000000,
random(10000000));
   printf("\n x= %d random = %d\n",1000000,
random(1000000));
   printf("\n x= %d random = %d\n",100000,
random(100000));
   printf("\n x= %d random = %d\n",10000,
random(10000));
   printf("\n x= %d random = %d\n",1000,
random(1000));
   printf("\n x= %d random = %d\n",100,
random(100));
   printf("\n x= %d random = %d\n",10,
random(10));
   printf("\n x= %d random = %d\n",1,
random(1));
```

Figure 6 illustrates a screenshot of the random function, where the random output number is clearly associated with the input parameter, mirroring the structure of the provided code snippet. Additionally, it is apparent that when the input consists of 8 digits, the generated random number consists of 7 digits. This design choice facilitates ease of adjustment for IDs, providing a straightforward method to quickly modify the ID as needed.

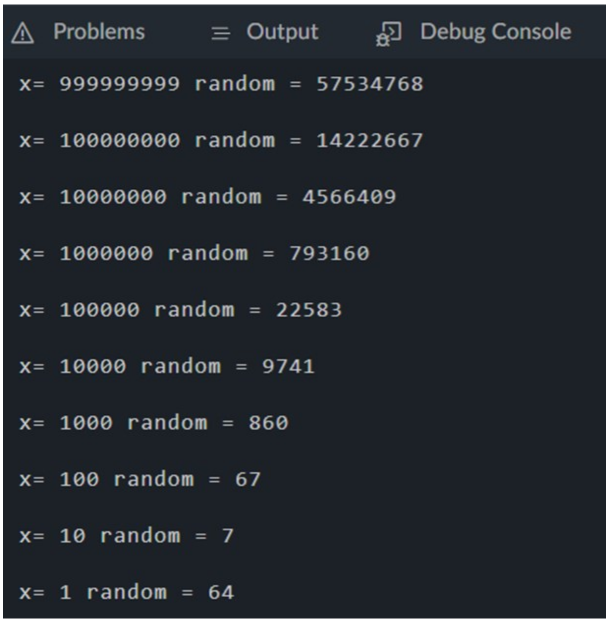

Figure 6 Screenshot of random function

- simple_sqrt

The provided code snippet effectively employs the printf function to evaluate the simple_sqrt function, presenting the square root values of diverse input parameters. Each printf statement illustrates both the input number and its corresponding square root value computed by the simple_sqrt function. This systematic approach commences with testing large input values, such as 999999999, and systematically decreases to smaller values like 2. By examining input parameters ranging from 9 digits to 1 digit, developers gain insight into how the simple_sqrt function performs across a wide spectrum of numerical ranges.

```
   printf("\n square root of %d is
%d",999999999, simple_sqrt(999999999));
    printf("\n square root of %d is
%d",100000000, simple_sqrt(100000000));
    printf("\n square root of %d is
%d",10000000, simple_sqrt(10000000));
    printf("\n square root of %d is %d",100000,
simple_sqrt(100000));
    printf("\n square root of %d is %d",10000,
simple_sqrt(10000));
    printf("\n square root of %d is %d",5000,
simple_sqrt(5000));
    printf("\n square root of %d is %d",900,
simple_sqrt(900));
    printf("\n square root of %d is %d",122,
simple_sqrt(122));
    printf("\n square root of %d is %d",10,
simple_sqrt(10));
    printf("\n square root of %d is %d",4,
simple_sqrt(4));
    printf("\n square root of %d is %d",2,
simple_sqrt(2));
```

Figure 7 provides a screenshot of the simple_sqrt function, showcasing the association between input parameters and their corresponding square root values, which aligns with the structure of the provided code snippet. Furthermore, the computed square root values accurately reflect the input IDs. For instance, 31622 is the square root of 999,999,999, 10000 is the square root of 100,000,000, 3162 is the square root of 10,000,000, and so forth. The simple_sqrt function produces correct results, demonstrating its effectiveness in accurately calculating square roots. Notably, the function accurately handles cases where the square root is an integer (e.g., the square root of 4 yields 2) and where it approximates the square root (e.g., the square root of 122 yields approximately 11, which is the closest integer below the exact square root).

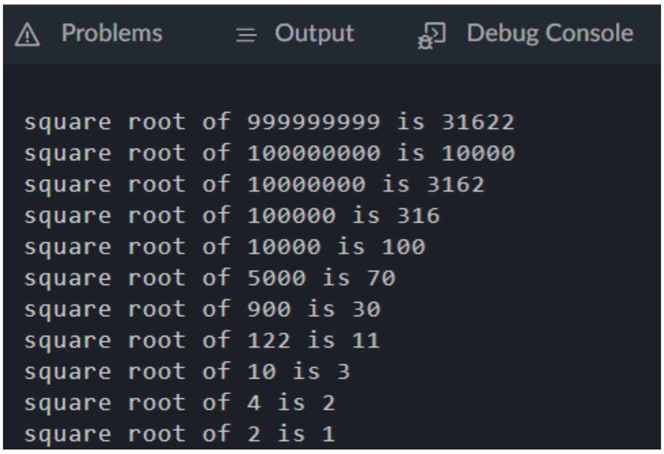

Figure 7 Screenshot of simple_sqrt function

- square_root_fpu

The provided code snippet utilizes the printf function to test the square_root_fpu function, displaying the square root values of various input parameters. Each printf statement showcases two values: the input number and the corresponding square root value computed by the square_root_fpu function with the same input. Starting with larger input values such as 999999999 and gradually decreasing to smaller values like 2, the code evaluates how the square_root_fpu function behaves across different input ranges.

```
printf("\n square root fpu of %d is
%d",999999999, square_root_fpu(999999999));
   printf("\n square root fpu of %d is
%d",100000000, square_root_fpu(100000000));
   printf("\n square root fpu of %d is
%d",10000000, square_root_fpu(10000000));
   printf("\n square root fpu of %d is %d",100000,
square_root_fpu(100000));
   printf("\n square root fpu of %d is %d",10000,
square_root_fpu(10000));
   printf("\n square root fpu of %d is %d",5000,
square_root_fpu(5000));
   printf("\n square root fpu of %d is %d",900,
square_root_fpu(900));
   printf("\n square root fpu of %d is %d",122,
square_root_fpu(122));
   printf("\n square root fpu of %d is %d",10,
square_root_fpu(10));
   printf("\n square root fpu of %d is %d",4,
square_root_fpu(4));
   printf("\n square root fpu of %d is %d",2,
square_root_fpu(2));
```

Figure 8 provides a screenshot of the square_root_fpu function, showcasing the association between input parameters and their corresponding square root values, which aligns with the structure of the provided code snippet. By examining the output of each printf statement, we can assess the accuracy and reliability of the square root values generated by the square_root_fpu function.

From Figure 9, the simple_sqrt function was executed five times, with recorded execution times of 6, 5, 6, 5, and 5 microseconds (µs) respectively while the square_root_fpu function was also executed five times, with recorded execution times of 4, 4, 5, 5, and 5 microseconds (µs) respectively.

These execution times confirm that the square_root_fpu function generally exhibits slightly faster execution times compared to the simple_sqrt function. This systematic approach aids in understanding the behaviours and effectiveness of the function in accurately calculating square roots using the Floating-Point Unit (FPU) over simple method, providing insights into its performance and suitability for various applications.

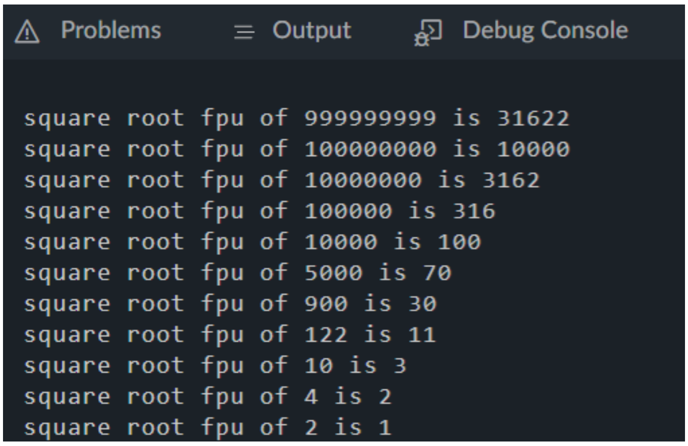

Figure 8 Screenshot square_root_fpu function

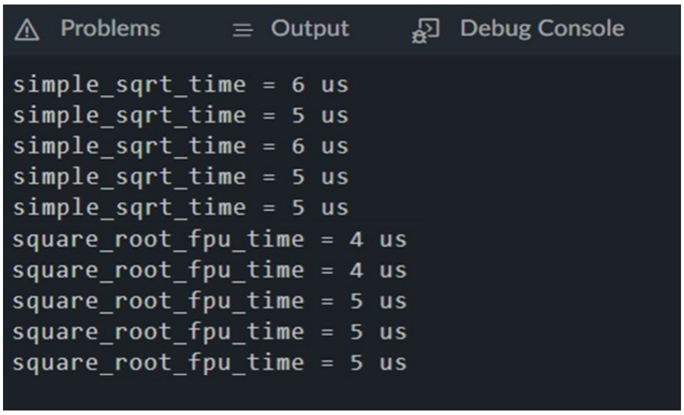

Figure 9 Screenshot executed time of simple_sqrt compared by square_root_fpu function.

- Overall system: Part A & Part B

From Figure 10, it is evident that functions such as DigitalIn, LCD, and sleep have been successfully tested using an LCD display on the microcontroller. Figure 10A illustrates the initial LCD display, presenting the name, student ID number, and the square root number of that ID. Subsequent figures, such as Figure 10B, demonstrate the functionality of the application as it displays the ID at 869445100 along with its corresponding square root value. This showcases the application's ability to increment the number based on the pressed SW_UP button and accurately display the square root value of the changed ID.

Similarly, Figure 10C and 10D showcase the functionality of the SW_DOWN button, as the ID decreases to 5950 and 92 respectively, with the LCD display correctly updating to show the square root value of the changed ID. Additionally, Figure 10E illustrates the scenario where the ID exceeds 999,999,999, while Figure 10F demonstrates the application's behavior when the ID goes below 1, which is beyond the expected range for this assignment.

Overall, these test cases highlight the successful implementation and functionality of the DigitalIn, LCD, and sleep functions in the microcontroller application.

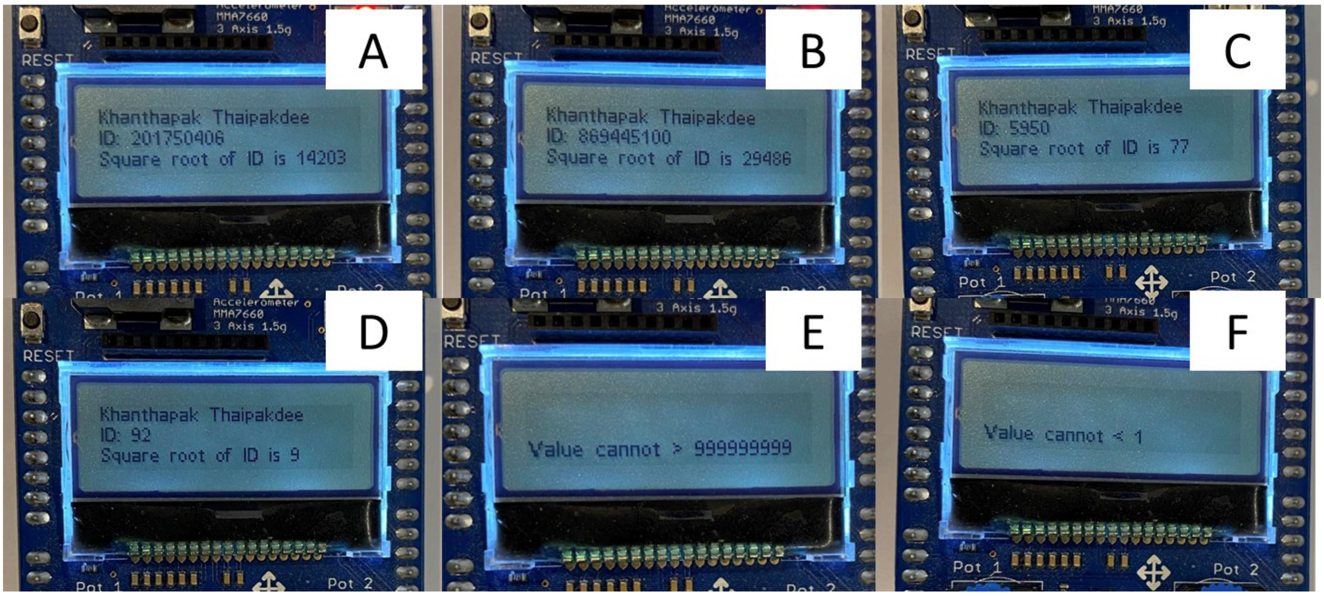

Figure 10 Capture of LCD display.

7. Top-down structure diagram: Part C

The top-down structure diagram (refers Figure 11) outlines the hierarchical organization of functions within the microcontroller application. Part C encompasses essential initialization and configuration functions, starting with BOARD_InitBootPins() for initializing board-specific pins, followed by BOARD_InitBootClocks() to set up system clocks, and BOARD_InitDebugConsole() to enable debug console functionality.

Moreover, the diagram includes BOARD_SW_IRQ_HANDLER() to handle interrupts from the switch inputs, GPIO_PinInit() for general-purpose input/output pin initialization, PORT_SetPinInterruptConfig() to configure pin interrupts, and functions like EnableIRQ() and DisableIRQ() to control interrupt handling.

Furthermore, Part C incorporates ADC16 initialization and configuration functions, such as ADC16_Init(), ADC16_GetDefaultConfig() , and ADC16_EnableHardwareTrigger(), along with calibration and channel configuration functions like ADC16_DoAutoCalibration(), ADC16_SetChannelConfig(), and ADC16_GetChannelStatusFlags().

The diagram also includes ADC16_GetChannelConversionValue() to retrieve ADC conversion results. Additionally, PRINTF() is included for output, along with delay() for timing control. This structured approach ensures systematic initialization, configuration, and operation of the microcontroller's peripherals and functionalities, facilitating efficient and reliable operation of the overall system.

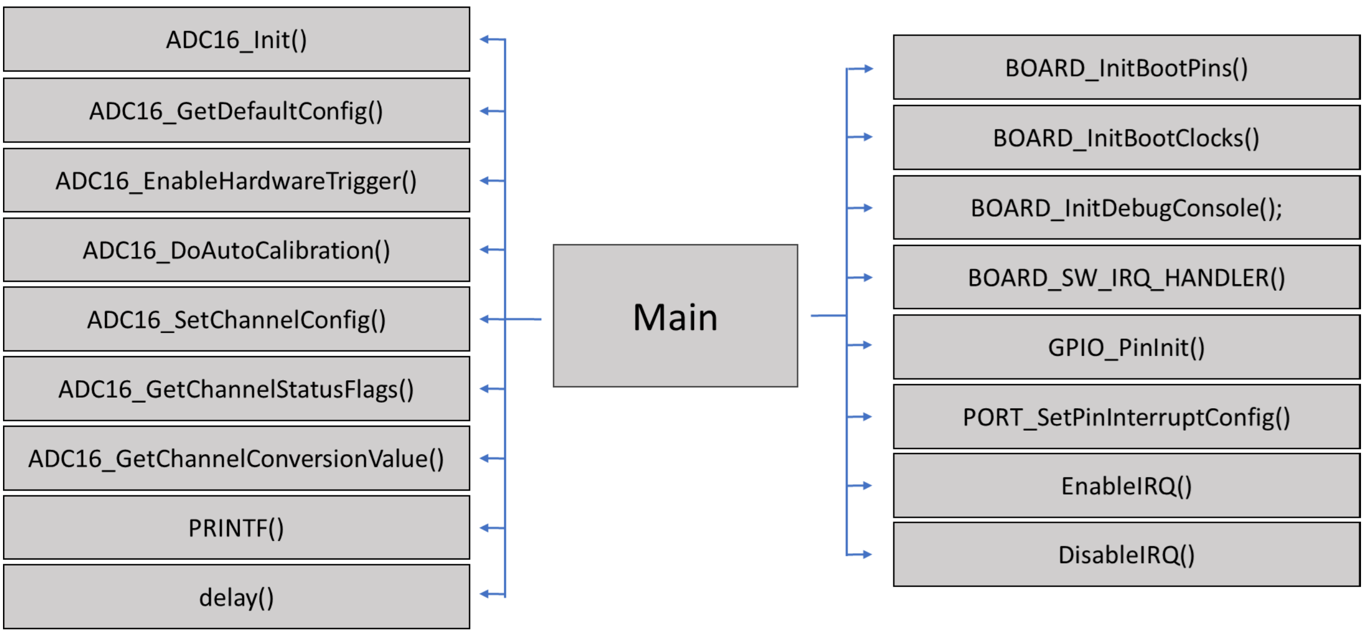

Figure 11 Top-down structure diagram: Part C

8. Flow charts of overall system: Part C

The flowchart (Figure 12) outlines the sequence of operations within the program, starting with the definition of constants for configuring a switch input, including specifications such as GPIO port, pin number, interrupt details, and switch name. Following this, the flowchart proceeds to define the initialization structure for the input switch pin. Subsequently, it initializes board pins, clocks, and the debug console. The next steps involve initializing the input switch GPIO and setting pin interrupts, particularly for the SW_UP switch. Following this, the flowchart illustrates the definition of the ADC configuration structure and the initialization of the ADC16 module. It then proceeds to perform ADC auto-calibration, displaying either "ADC16_DoAutoCalibration() Done" or "ADC16_DoAutoCalibration() Failed" based on the success of the calibration process. Finally, the flowchart concludes with the enabling of IRQs for interrupt handling. This delineation provides a clear overview of the program's initialization and configuration steps, ensuring proper setup for subsequent operations.

In the main loop of the program, the first action is to configure the ADC user channel to ADC0_SE12 to read the analog input from Pot_1. Subsequently, the program disables interrupts associated with the switch input (BOARD_SW_IRQ) to prevent interruptions during the ADC conversion process. Following this, the program triggers the conversion if in software trigger mode and waits for the flag that confirms the completion of the conversion. Once the conversion is done, the program reads the value from Pot_1 and stores it in a variable. After that, interrupts are enabled again to allow for switch input handling. The calculated voltage value from Pot_1 is then displayed on the debug console, formatted as "Pot_1: 1.15 V", indicating the voltage measured from Pot_1. Finally, a delay is introduced to control the sampling rate, and the loop repeats to continuously sample the analog input from Pot_1 and print its corresponding voltage value. This process ensures that the value from Pot_1 is printed at regular sampling intervals.

In the Interrupt Service Routine (ISR), the first action is to clear the external interrupt flag to acknowledge the interrupt. Following this, the ISR defines the ADC configuration structure and initializes the ADC16 module. Subsequently, it sets the configuration for channel_2 of ADC16 (ADC0_SE13, Pot 2). The ISR then disables interrupts to ensure uninterrupted execution of the ADC conversion process. It triggers the conversion using ADC16_SetChannelConfig and waits for the flag that confirms the completion of the conversion. Once the conversion is done, the ISR reads the value from Pot_2 and stores it in a variable. Interrupts are then enabled again to resume normal interrupt handling. Later, the ISR displays a message indicating that "SW_UP is pressed" and prints the calculated value in terms of voltage, formatted as "Pot_2: 1.15 V". Finally, the ISR exits, allowing the program to continue execution.

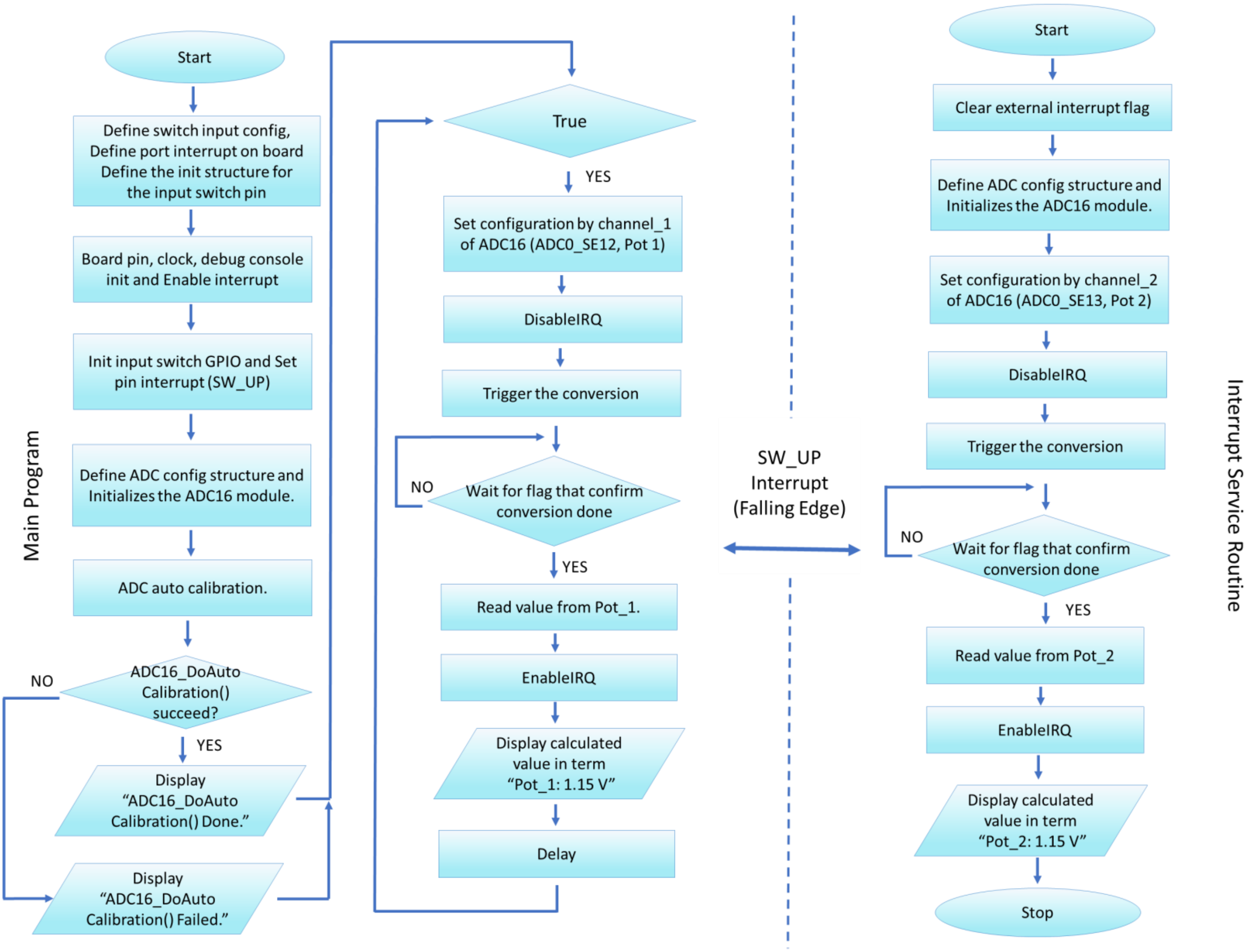

Figure 12 Flow charts of overall system: Part C

9. Description of each module: Part C

9.1 BOARD_InitBootPins()

The BOARD_InitBootPins() function is a boot-time initialization function responsible for setting up the pins of the microcontroller. This initialization typically involves configuring the pins for their intended use, such as input, output, or alternate function modes, and setting any required pin attributes or characteristics, such as pull-up or pull-down resistors, drive strength, or slew rate control. It achieves this by calling the BOARD_InitPins() function, which is the actual pin initialization routine.

Within BOARD_InitPins(), the Port B clock gate is enabled to ensure that the clock to Port B is enabled. Then, specific pins are configured for various functions:

PORTB16 (pin 62) and PORTB17 (pin 63) are configured as UART0_RX and UART0_TX, respectively, for UART communication.

PORTB2 (pin 55) and PORTB3 (pin 56) are configured as analog input pins for Pot_1 and Pot_2, respectively, for ADC conversion.

PORTB10 (pin 58) is configured as a GPIO pin for SW_UP functionality.

These configurations are crucial for the operation of the corresponding peripherals and features, as indicated in Figure 13 for reference.

9.2 BOARD_InitBootClocks()

BOARD_InitBootClocks() function is a boot-time initialization function responsible for configuring the clock settings of the microcontroller. This function ensures that the microcontroller's clock system is properly configured to meet the requirements of the application.

```
void BOARD_InitPins(void)
{
/* Port B Clock Gate Control: Clock enabled */
  CLOCK_EnableClock(kCLOCK_PortB);

/* PORTB16 (pin 62) is configured as UART0_RX */
  PORT_SetPinMux(PORTB, 16U, kPORT_MuxAlt3);

/* PORTB17 (pin 63) is configured as UART0_TX */
  PORT_SetPinMux(PORTB, 17U, kPORT_MuxAlt3);

/* Pot_1 */
/* PORTB2 (pin 55) is configured as ADC0_SE12 */
    PORT_SetPinMux(PORTB, 2U,
  kPORT_PinDisabledOrAnalog);

/* Pot_2 */
/* PORTB3 (pin 56) is configured as ADC0_SE13 */
    PORT_SetPinMux(PORTB, 3U,
  kPORT_PinDisabledOrAnalog);

/* SW_UP */
/* PORTB10 (pin is configured as PTB10 */
    PORT_SetPinMux(PORTB, 10U,
  kPORT_MuxAsGpio);

/* Mask bits to zero which are setting */
/* UART 0 transmit data source select:
UART0_TX pin. */

  SIM->SOPT5 = ((SIM->SOPT5 &
  (~(SIM_SOPT5_UART0TXSRC_MASK)))
          | SIM_SOPT5_UART0TXSRC
  (SOPT5_UART0TXSRC_UART_TX));
}
```

9.3 BOARD_InitDebugConsole()

The BOARD_InitDebugConsole() function is a boot-time initialization function responsible for setting up the debug console interface on the microcontroller. This function initializes the UART interface used for debugging purposes, allowing the microcontroller to communicate with a host computer for debugging and monitoring purposes. In the board.h file, the baud rate is set to 115200.

```
BOARD_DEBUG_UART_BAUDRATE
#define BOARD_DEBUG_UART_BAUDRATE
115200
#endif /* BOARD_DEBUG_UART_BAUDRATE
*/
```

9.4 GPIO_PinInit()

The function GPIO_PinInit() initializes a digital input pin on a GPIO port specified by the base pointer base and the pin number pin, according to the configuration provided in the config pointer.

The GPIO port, port number, and pin number are defined as GPIOB, PORTB, and 10U respectively. Then, an initialization structure sw_config is defined with the configuration parameters for the digital input pin, specifying it as a digital input (kGPIO_DigitalInput) with no additional settings (0).

```
#define BOARD_SW_GPIO       GPIOB
#define BOARD_SW_PORT       PORTB
#define BOARD_SW_GPIO_PIN    10U


/* Define the init structure for the input switch pin
*/
   gpio_pin_config_t sw_config = {
      kGPIO_DigitalInput,
      0,
   };

  GPIO_PinInit(BOARD_SW_GPIO,
  BOARD_SW_GPIO_PIN, &sw_config);
```

9.5 PORT_SetPinInterruptConfig()

PORT_SetPinInterruptConfig() function configures the interrupt settings for a specific pin on a GPIO port. It takes the following parameters:

port: GPIO peripheral base pointer (e.g., GPIOA, GPIOB, GPIOC, etc.).

pin: GPIO port pin number.

config: Pointer to a configuration structure specifying the interrupt settings for the pin.

In this code snippet, the input switch GPIO pin is configured to generate interrupts when a falling edge is detected. The port and pin parameters are set according to the GPIO port and pin number of "SW_UP", respectively, as illustrated in Figure 13.

```
#define BOARD_SW_GPIO       GPIOB
#define BOARD_SW_PORT       PORTB
#define BOARD_SW_GPIO_PIN    10U


/* Set input switch GPIO as Pin Interrupt. */
#if
(defined(FSL_FEATURE_PORT_HAS_NO_INTER
RUPT) &&
FSL_FEATURE_PORT_HAS_NO_INTERRUPT)
   GPIO_SetPinInterruptConfig(BOARD_SW_GPIO,
BOARD_SW_GPIO_PIN,
kGPIO_InterruptFallingEdge);
#else

PORT_SetPinInterruptConfig(BOARD_SW_PORT,
BOARD_SW_GPIO_PIN,
kPORT_InterruptFallingEdge);
#endif
```

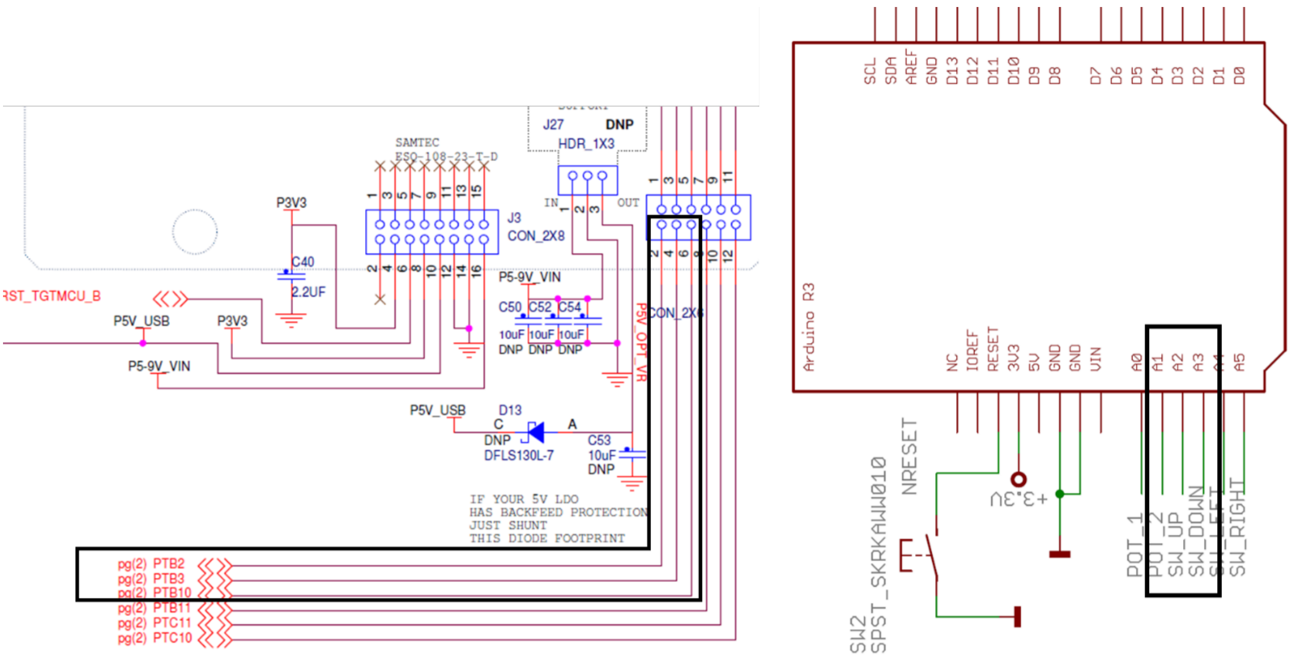

Figure 13 Pin schematics.

9.6 ADC16_GetDefaultConfig ()

ADC16_GetDefaultConfig() function retrieves the default configuration settings for the ADC16 module. The retrieved default configuration settings are stored in the adc16ConfigStruct variable of type adc16_config_t. These default settings can later be modified as needed before initializing the ADC16 module

9.7 ADC16_Init ()

ADC16_Init() function initializes the ADC16 module with the specified configuration settings. This function initializes the ADC hardware according to the provided configuration, enabling the module to perform analog-to-digital conversions. The ADC16_Init() function is invoked to initialize the ADC16 module with the provided configuration settings stored in adc16ConfigStruct.

9.8 ADC16_EnableHardwareTrigger ()

ADC16_EnableHardwareTrigger() function enables the hardware trigger mode for ADC conversions. By enabling hardware triggering, the ADC will wait for an external signal to start each conversion.

```
/* ADC Config */
  adc16_config_t adc16ConfigStruct;
  adc16_channel_config_t
  adc16ChannelConfigStruct;

  ADC16_GetDefaultConfig(&adc16ConfigStr
  uct);

/* Initializes the ADC16 module. */
   ADC16_Init(DEMO_ADC16_BASE,
  &adc16ConfigStruct);

  ADC16_EnableHardwareTrigger(DEMO_AD
  C16_BASE, false); /* Make sure the software
  trigger is used. */
```

9.9 ADC16_DoAutoCalibration ()

The ADC16_DoAutoCalibration() function performs automatic calibration for the ADC module. The calibration adjusts the gain on both the positive and negative sides of the ADC's input range automatically. It's crucial to execute this calibration process before utilizing the ADC converter to ensure accurate and reliable conversion of analog signals to digital values.

```
/* Calibration */
#if
defined(FSL_FEATURE_ADC16_HAS_CALIBRATION)
&& FSL_FEATURE_ADC16_HAS_CALIBRATION
   if (kStatus_Success ==
ADC16_DoAutoCalibration(DEMO_ADC16_BASE))
  {
  PRINTF(" ADC16_DoAutoCalibration() Done.\r\n");
  }
   else
  {
   PRINTF(" ADC16_DoAutoCalibration() Failed.\r\n");
  }
#endif /* FSL_FEATURE_ADC16_HAS_CALIBRATION
*/
```

9.10 ADC16_SetChannelConfig ()

The ADC16_SetChannelConfig() operation triggers the conversion when in software trigger mode. It takes the following parameters:

base: The base address of the ADC peripheral.

channelGroup: The index of the ADC channel group.

config: A pointer to a configuration structure containing the settings for the ADC channel.

The ADC channel configuration is set up. The DEMO_ADC16_USER_CHANNEL specifies the specific channel number (12U in this case), which corresponds to the physical pin PTB2 (ADC0_SE12) associated with Pot1. The adc16ChannelConfigStruct is initialized with this channel number and with interrupt-on-conversion completion disabled.

```
#define DEMO_ADC16_BASE         ADC0
#define DEMO_ADC16_CHANNEL_GROUP 0U
#define DEMO_ADC16_USER_CHANNEL  12U /*
PTB2, ADC0_SE12 */ /* Pot_1*/

/* Set Channel_1 of ADC16 */

  adc16ChannelConfigStruct.channelNumber
= DEMO_ADC16_USER_CHANNEL;

  adc16ChannelConfigStruct.enableInterruptOnConv
ersionCompleted = false;

ADC16_SetChannelConfig(DEMO_ADC16_BASE,
DEMO_ADC16_CHANNEL_GROUP,
&adc16ChannelConfigStruct);
```

9.11 ADC16_GetChannelStatusFlags ()

The ADC16_GetChannelStatusFlags() function is used to retrieve the status flags associated with a specific channel within an ADC module. It takes the following parameters:

base: The base address of the ADC peripheral.

channelGroup: The index of the ADC channel group.

```
#define DEMO_ADC16_BASE          ADC0
#define DEMO_ADC16_CHANNEL_GROUP 0U
#define DEMO_ADC16_USER_CHANNEL  12U
/* PTB2, ADC0_SE12 */ /* Pot_1*/

while (0U ==
(kADC16_ChannelConversionDoneFlag &

ADC16_GetChannelStatusFlags(DEMO_ADC16_B
ASE, DEMO_ADC16_CHANNEL_GROUP)))
    {

    }
```

9.12 ADC16_GetChannelConversionValue ()

ADC16_GetChannelConversionValue () function is used to retrieve the conversion result of a specific channel within an ADC module. It takes the following parameters:

base: The base address of the ADC peripheral.

channelGroup: The index of the ADC channel group.

This function returns the conversion value of the specified channel as a 16-bit unsigned integer. The conversion value represents the analog input voltage converted to a digital value.

The result is stored in the variable named g_Adc16ConversionValue_ch1 for further display.

```
#define DEMO_ADC16_BASE          ADC0
#define DEMO_ADC16_CHANNEL_GROUP 0U
#define DEMO_ADC16_USER_CHANNEL  12U /*
PTB2, ADC0_SE12 */ /* Pot_1*/

int g_Adc16ConversionValue_ch1;
g_Adc16ConversionValue_ch1 =
ADC16_GetChannelConversionValue(DEMO_ADC
16_BASE, DEMO_ADC16_CHANNEL_GROUP);
```

9.13 PRINTF ()

The PRINTF() function is used to print formatted output to the debug console. The macro PRINTF is defined as DbgConsole_Printf in fsl_debug_console.h.

The PRINTF() function will be utilized in the main program to display messages and converted values from Pot_1, as well as in the BOARD_SW_IRQ_HANDLER() function to display messages and converted values from Pot_2 to a serial terminal. Exactly, by defining PRINTF_FLOAT_ENABLE as 1, the PRINTF function becomes capable of handling floating-point numbers, enabling the printing of float values using PRINTF().

```
/* Print value of Pot_1 to serial terminal*/
      PRINTF(" Pot_1: %.2f V\r\n\n",
3.3/4096*g_Adc16ConversionValue_ch1);

void BOARD_SW_IRQ_HANDLER(void)
{

 /* Print value of Pot_2 to serial terminal*/

PRINTF(" %s is pressed \r\n Pot_2 = %.2f V \r\n\n",
BOARD_SW_NAME,3.3/4096*g_Adc16Conversion
Value_ch2);
}
```

9.14 GPIO_PortClearInterruptFlags ()

The GPIO_PortClearInterruptFlags() function clears the interrupt flags for a specific GPIO port. It takes two arguments: the GPIO peripheral base pointer and a bitmask representing the pins whose interrupt flags should be cleared.

9.15 delay ()

The snippet code defines a delay function that loops for a predetermined number of iterations to create a delay. Inside the function, a volatile uint32_t variable named i is declared and initialized to 0. Then, a for loop iterates from 0 to 59999999, incrementing the value of i in each iteration. Within the loop, the __asm("NOP") statement is used as a no-operation instruction, effectively creating a delay.

```
void delay(void)
{
   volatile uint32_t i = 0;
   for (i = 0; i < 60000000; ++i)
   {
      __asm("NOP"); /* delay */
    }
}
```

9.16 EnableIRQ ()

The EnableIRQ() function is used to enable interrupts for a specific interrupt source. It takes the interrupt number or priority level as its argument and enables the interrupt accordingly. In the provided code snippet, EnableIRQ() is used to enable interrupts for the switch input, allowing the microcontroller to respond to button presses by triggering the corresponding interrupt service routine (ISR) which is BOARD_SW_IRQ_HANDLER ().

9.17 DisableIRQ ()

The DisableIRQ() function is employed to disable interrupts for a specific interrupt source.

DisableIRQ() is utilized to temporarily disable interrupts while configuring the ADC and reading analog input values.

9.18 BOARD_SW_IRQ_HANDLER ()

The BOARD_SW_IRQ_HANDLER() function serves as the interrupt service routine (ISR) responsible for managing interrupts generated by the input switch. Within this ISR, the first action taken is to clear the external interrupt flag associated with the input switch GPIO pin, ensuring proper handling of subsequent interrupts. Subsequently, configuration structures for the analog-to-digital converter (ADC) are defined. The ADC is then configured to read from ADC0_SE13 (PORTB3: pin 56) , representing Pot_2, and interrupts are momentarily disabled to maintain atomicity during this configuration process. Following this, the ADC channel configuration is set, and a conversion is triggered, with the function subsequently waiting until the ADC conversion is completed. Once completed, interrupts are re-enabled, and the conversion value from Pot_2 is retrieved. Utilizing the PRINTF function, a message is displayed to indicate that the switch is pressed, alongside the calculated voltage value corresponding to Pot_2.

Finally, the SDK_ISR_EXIT_BARRIER macro is employed to exit the ISR, ensuring proper handling of subsequent interrupt events.

```
#define DEMO_ADC16_BASE          ADC0
#define DEMO_ADC16_CHANNEL_GROUP 0U
#define DEMO_ADC16_USER_CHANNEL  12U
/* PTB2, ADC0_SE12 */ /* Pot_1*/
#define DEMO_ADC16_USER_CHANNEL_2 13U
 /* PTB3, ADC0_SE13 */ /* Pot_2*/
#define BOARD_SW_IRQ        PORTB_IRQn
#define BOARD_SW_IRQ_HANDLER
PORTB_IRQHandler

void BOARD_SW_IRQ_HANDLER(void)
{
#if
(defined(FSL_FEATURE_PORT_HAS_NO_INTERR
UPT) &&
FSL_FEATURE_PORT_HAS_NO_INTERRUPT)

GPIO_GpioClearInterruptFlags(BOARD_SW_GPIO,
1U << BOARD_SW_GPIO_PIN);
#else
   /* Clear external interrupt flag. */
   GPIO_PortClearInterruptFlags(BOARD_SW_GPIO,
1U << BOARD_SW_GPIO_PIN);
#endif

    /* ADC Config */
   adc16_channel_config_t adc16ChannelConfigStruct;

    /* Set Channel_2 of ADC16 */
    adc16ChannelConfigStruct.channelNumber
= DEMO_ADC16_USER_CHANNEL_2;

  adc16ChannelConfigStruct.enableInterruptOnConver
sionCompleted = false;
```

```
  DisableIRQ(BOARD_SW_IRQ);

  ADC16_SetChannelConfig(DEMO_ADC16_BAS
E, DEMO_ADC16_CHANNEL_GROUP,
&adc16ChannelConfigStruct);
    while (0U ==
(kADC16_ChannelConversionDoneFlag &

  ADC16_GetChannelStatusFlags(DEMO_ADC16_
BASE, DEMO_ADC16_CHANNEL_GROUP)))
          {
          }

    int g_Adc16ConversionValue_ch2;

    /* Read value from Pot_2*/
    g_Adc16ConversionValue_ch2 =
ADC16_GetChannelConversionValue(DEMO_AD
C16_BASE,
DEMO_ADC16_CHANNEL_GROUP);

    EnableIRQ(BOARD_SW_IRQ);

    PRINTF(" %s is pressed \r\n Pot_2 = %.2f V
\r\n\n",
BOARD_SW_NAME,3.3/4096*g_Adc16Conversio
nValue_ch2);
   SDK_ISR_EXIT_BARRIER;
}
```

10. Testing scheme: Part C

The test scheme for Part C involves evaluating the functionality and performance of the system implemented using the Manufacturers' supplied SDK (KSDK) for accessing Microcontroller peripherals. The FRDM-K64F board is connected to a PC via the serial port. Terminal software, such as PuTTY, is used on the PC to receive and display the data transmitted from the FRDM-K64F board.

Testing Procedure: The program is initiated on the FRDM-K64F board, with the necessary configurations set up to enable ADC sampling and serial communication. The user interacts with the system by pressing the SW_UP button to trigger the display of values from Pot 2. The system continuously samples values from Pot 1 at a predefined rate.

Observations and Findings: Figure 14: Putty Screenshot illustrating the program's interface. At the top of the screenshot, there is a general description of the program's functionalities, such as ADC16 Pot 1 and Pot 2, along with the instruction to press SW_UP to read and display the value from Pot 2. Additionally, the continuous display of values from Pot 1 is evident, while pressing SW_UP triggers the display of values from Pot 2.

Throughout the testing process, particular attention was paid to the accuracy of ADC readings. It was observed that at maximum input, the displayed voltage readings fluctuated between 3.30V and 3.29V. This variability indicates the presence of noise in the received input signals.

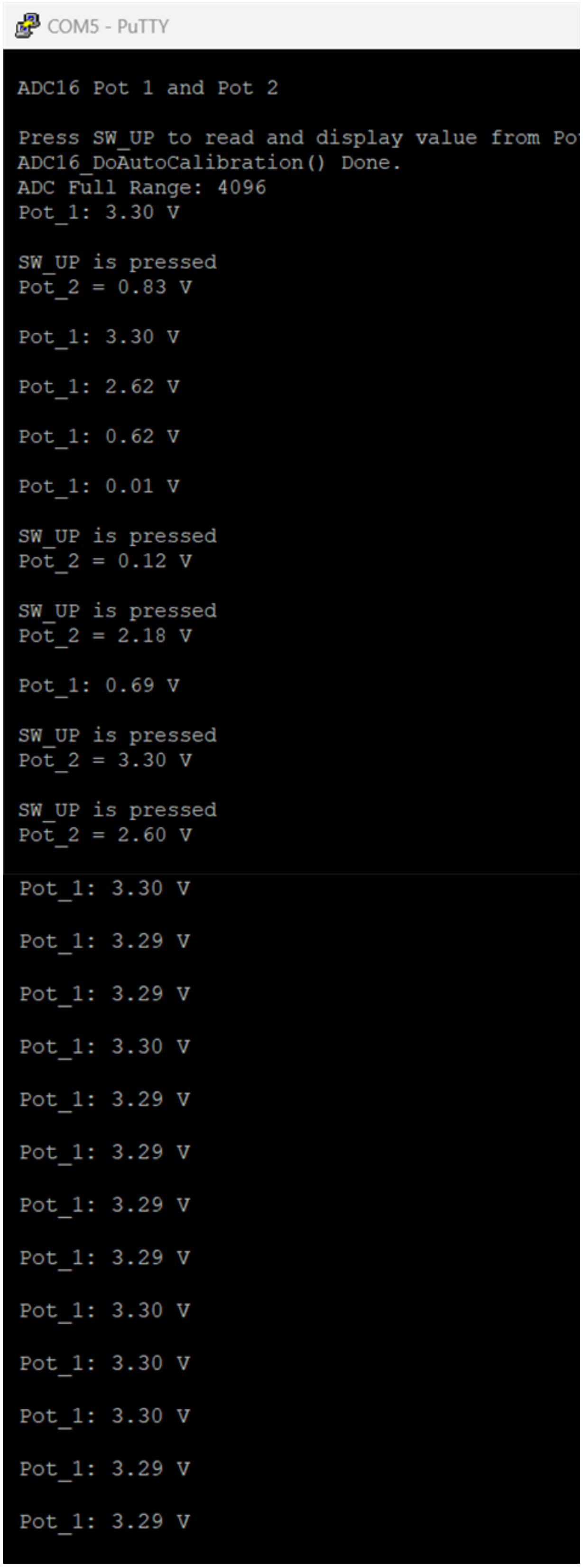

Figure 14 Putty Screenshot

11. Discussion

This response addresses the challenges and suggests improvements to enhance the system:

Utilization of Multi-Core Microprocessors: Integrating a multi-core microprocessor can optimize system performance by assigning specific tasks to individual cores. For instance, one core can handle ADC sampling accurately, while another can manage user input detection and display the pot value on the serial monitor. This approach ensures efficient multitasking, leading to improved overall performance.

Implementation of Real-Time Operating System (RTOS): Deploying an RTOS can enhance system responsiveness and efficiency through task scheduling, priority management, and inter-task communication. With an RTOS, tasks such as ADC sampling and LCD display updates can be executed in a timely manner, ensuring smooth operation. This ensures that critical tasks are prioritized, leading to improved system reliability and performance.

Button Debouncing Mechanism: Incorporating a pulse detection mechanism to eliminate button debounce issues can enhance the reliability and responsiveness of user input detection. By filtering out debounce signals, false button presses can be prevented, improving the accuracy of user interactions. This ensures that user inputs are accurately detected, enhancing overall system usability.

Signal Filtering Techniques: Implementing signal filtering techniques, such as digital filters or averaging algorithms, can mitigate noise and improve the accuracy of ADC readings. By filtering out noise signals, the system can achieve more precise, particularly in scenarios where only the MSB capacitor bank is calibrated. This leads to enhanced overall performance and reliability of the system, ensuring accurate data acquisition.

12. Conclusion

Part A focuses on getting acquainted with the FRDM-K64F board and MBED platform using the Application Shield. The task involves creating a program that showcases the user's name and ID on the LCD screen. Furthermore, the program should enable the user to adjust a displayed number by incrementing or decrementing it through cursor key inputs.

Part B aims to enhance assembly language programming skills for the Cortex M4 processor by implementing two subroutines to calculate the square root of a given number. These subroutines should determine the largest integer closest to, but not exceeding, the square root of the input. They will be integrated into the Part A code, with one subroutine utilizing the FPU for computation and the other not. These two parts provide experience in software development using the MBED library.

In Part C, the objective is to utilize the Manufacturers' supplied SDK (KSDK) for accessing Microcontroller peripherals, specifically the ADC inputs connected to Pot 1 and Pot 2 of the Application Shield. This part offers hands-on experience with Freescale libraries and involves working with interrupts. Unlike previous parts, MBED libraries are not permitted here; only Manufacturers' libraries are allowed. The ultimate goal is to read ADC values from Pot 1 and Pot 2 and display them on a PC terminal program using the FRDM-K64F board through the serial port. Pot 1 values are sampled at regular intervals, while Pot 2 values are sampled when the joystick is pressed up.

Appendix A: Code Implementation for Part A & Part B

```
#include "mbed.h"
#include <stdio.h>
#include <stdlib.h>
#include "C12832.h"

// Define digital inputs for switches connected to pins A2, A3, A4, and A5
DigitalIn SW_UP(A2);
DigitalIn SW_DOWN(A3);
DigitalIn SW_LEFT(A4);
DigitalIn SW_RIGHT(A5);

// Define timer object for
Timer timer1;


// Initialize the LCD display with the specified pins
// C12832(PinName mosi, PinName sck, PinName reset, PinName a0, PinName cs)
C12832 lcd(D11, D13, D12, D7, D10);

// Declare external assembly language function (in a *.s file)
extern "C" int simple_sqrt(int value);
extern "C" int square_root_fpu(int value);


int random(int input)
{
    int output = 0;
    if (input > 99999999 && input <= 999999999){
        output = (10000000 + rand() % 90000000);
    }
    else if (input > 9999999 && input <= 99999999){
        output = (1000000 + rand() % 9000000);
    }
    else if (input > 999999 && input <= 9999999){
        output = (100000 + rand() % 900000);
    }
    else if (input > 99999 && input <= 999999){
        output = (10000 + rand() % 90000);
    }
    else if (input > 9999 && input <= 99999){
        output = (1000 + rand() % 9000);
    }
    else if (input > 999 && input <= 9999){
        output = (100 + rand() % 900);
    }
    else if (input > 0 && input <= 999){
        output = (rand() % 100);
    }

    return output;
}
```

```
// main() runs in its own thread in the OS
int main()
{

    lcd.cls();
    int id = 201750406;
    int sqrt_val = 0;
    long long unsigned  simple_sqrt_time,square_root_fpu_time;


    /* // Uncomment to test random function
    printf("\n x= %d random = %d\n",999999999, random(999999999));
    printf("\n x= %d random = %d\n",100000000, random(100000000));
    printf("\n x= %d random = %d\n",10000000, random(10000000));
    printf("\n x= %d random = %d\n",1000000, random(1000000));
    printf("\n x= %d random = %d\n",100000, random(100000));
    printf("\n x= %d random = %d\n",10000, random(10000));
    printf("\n x= %d random = %d\n",1000, random(1000));
    printf("\n x= %d random = %d\n",100, random(100));
    printf("\n x= %d random = %d\n",10, random(10));
    printf("\n x= %d random = %d\n",1, random(1));
    // */

    /* //Uncomment to test simple_sqrt assembly code
    printf("\n square root of %d is %d",999999999, simple_sqrt(999999999));
    printf("\n square root of %d is %d",100000000, simple_sqrt(100000000));
    printf("\n square root of %d is %d",10000000, simple_sqrt(10000000));
    printf("\n square root of %d is %d",100000, simple_sqrt(100000));
    printf("\n square root of %d is %d",10000, simple_sqrt(10000));
    printf("\n square root of %d is %d",5000, simple_sqrt(5000));
    printf("\n square root of %d is %d",900, simple_sqrt(900));
    printf("\n square root of %d is %d",122, simple_sqrt(122));
    printf("\n square root of %d is %d",10, simple_sqrt(10));
    printf("\n square root of %d is %d",4, simple_sqrt(4));
    printf("\n square root of %d is %d",2, simple_sqrt(2));
    // */

    /* //Uncomment to test square_root_fpu assembly code
    printf("\n square root fpu of %d is %d",999999999, square_root_fpu(999999999));
    printf("\n square root fpu of %d is %d",100000000, square_root_fpu(100000000));
    printf("\n square root fpu of %d is %d",10000000, square_root_fpu(10000000));
    printf("\n square root fpu of %d is %d",100000, square_root_fpu(100000));
    printf("\n square root fpu of %d is %d",10000, square_root_fpu(10000));
    printf("\n square root fpu of %d is %d",5000, square_root_fpu(5000));
    printf("\n square root fpu of %d is %d",900, square_root_fpu(900));
    printf("\n square root fpu of %d is %d",122, square_root_fpu(122));
    printf("\n square root fpu of %d is %d",10, square_root_fpu(10));
    printf("\n square root fpu of %d is %d",4, square_root_fpu(4));
    printf("\n square root fpu of %d is %d",2, square_root_fpu(2));
    // */
```

```
while (true) {
        if (SW_UP.read()){
            id = id + random(id);
            lcd.cls();
                if (id > 999999999){
                    id = 999999999;
                    lcd.locate(0,20);
                    lcd.printf("Value cannot > 999999999");
                    ThisThread::sleep_for(10000); //10s
                    lcd.cls();
                }
        }
        if (SW_DOWN.read()){
            id = id - (random(id));
            lcd.cls();
                if (id < 0){
                    id = 1;
                    lcd.locate(0,20);
                    lcd.printf("Value cannot < 1");
                    ThisThread::sleep_for(10000); //10s
                    lcd.cls();
                }
        }

        if (SW_LEFT.read()){
        // Uncomment to test exceuted time //
        // timer1.start(); //
        sqrt_val = simple_sqrt(id);
        // timer1.stop(); //
                  // simple_sqrt_time = timer1.elapsed_time().count();
        // timer1.reset(); //
        // printf("\n simple_sqrt_time = %llu us",simple_sqrt_time); //
        lcd.cls();
        }

        if (SW_RIGHT.read()){
        // timer1.start(); //
        sqrt_val = square_root_fpu(id);
        // timer1.stop(); //
        // square_root_fpu_time = timer1.elapsed_time().count();
        // timer1.reset(); //
        // printf("\n square_root_fpu_time = %llu us",square_root_fpu_time); //
        lcd.cls();
        }

        lcd.locate(0,0);
        lcd.printf("Khanthapak Thaipakdee");
        lcd.locate(0,10);
        lcd.printf("ID: %d", id);

        lcd.locate(0,20);
        lcd.printf("Square root of ID is %d", sqrt_val);
    }
}
```

Appendix B: Code Implementation for Part C

```
#include "fsl_debug_console.h"
#include "fsl_port.h"
#include "fsl_gpio.h"
#include "fsl_common.h"
#include "fsl_adc16.h"
#include "pin_mux.h"
#include "clock_config.h"
#include "board.h"
/*******************************************************************************
 * Definitions
 ******************************************************************************/
#define BOARD_SW_GPIO        GPIOB
#define BOARD_SW_PORT        PORTB
#define BOARD_SW_GPIO_PIN    10U
#define BOARD_SW_IRQ         PORTB_IRQn
#define BOARD_SW_IRQ_HANDLER PORTB_IRQHandler
#define BOARD_SW_NAME        "SW_UP"

#define DEMO_ADC16_BASE          ADC0
#define DEMO_ADC16_CHANNEL_GROUP 0U
#define DEMO_ADC16_USER_CHANNEL  12U /* PTB2, ADC0_SE12 */ /* Pot_1*/
#define DEMO_ADC16_USER_CHANNEL_2  13U /* PTB3, ADC0_SE13 */ /* Pot_2*/


/*******************************************************************************
 * Prototypes
 ******************************************************************************/

 /*******************************************************************************
 * Variables
 ******************************************************************************/

volatile uint32_t g_Adc16InterruptCounter;
const uint32_t g_Adc16_12bitFullRange = 4096U;

/*******************************************************************************
 * Code
 ******************************************************************************/

 void delay(void)
{
    volatile uint32_t i = 0;
    for (i = 0; i < 60000000; ++i)
    {
        __asm("NOP"); /* delay */
    }
}
```

```
*!
 * @brief Interrupt service function of switch.
 *
 */
void BOARD_SW_IRQ_HANDLER(void)
{
#if (defined(FSL_FEATURE_PORT_HAS_NO_INTERRUPT) &&
FSL_FEATURE_PORT_HAS_NO_INTERRUPT)
   /* Clear external interrupt flag. */
   GPIO_GpioClearInterruptFlags(BOARD_SW_GPIO, 1U << BOARD_SW_GPIO_PIN);
#else
   /* Clear external interrupt flag. */
   GPIO_PortClearInterruptFlags(BOARD_SW_GPIO, 1U << BOARD_SW_GPIO_PIN);
#endif

   /* Increament interrupt count */
   g_Adc16InterruptCounter++;

    /* ADC Config */
    adc16_config_t adc16ConfigStruct;
   adc16_channel_config_t adc16ChannelConfigStruct;

    /* Set Channel_2 of ADC16 */
    adc16ChannelConfigStruct.channelNumber                    = DEMO_ADC16_USER_CHANNEL_2;
    adc16ChannelConfigStruct.enableInterruptOnConversionCompleted = false;

    DisableIRQ(BOARD_SW_IRQ);

    ADC16_SetChannelConfig(DEMO_ADC16_BASE, DEMO_ADC16_CHANNEL_GROUP,
&adc16ChannelConfigStruct);
    while (0U == (kADC16_ChannelConversionDoneFlag &
                              ADC16_GetChannelStatusFlags(DEMO_ADC16_BASE,
DEMO_ADC16_CHANNEL_GROUP)))
            {
            }

    int g_Adc16ConversionValue_ch2;

    /* Read value from Pot_2*/
    g_Adc16ConversionValue_ch2 = ADC16_GetChannelConversionValue(DEMO_ADC16_BASE,
DEMO_ADC16_CHANNEL_GROUP);

    EnableIRQ(BOARD_SW_IRQ);

    PRINTF(" %s is pressed \r\n Pot_2 = %.2f V\r\n",
BOARD_SW_NAME,3.3/4096*g_Adc16ConversionValue_ch2);
   SDK_ISR_EXIT_BARRIER;
}
```

```
*!
 * @brief Main function
 */
int main(void)
{
   /* Define the init structure for the input switch pin */
   gpio_pin_config_t sw_config = {
      kGPIO_DigitalInput,
      0,
   };

   BOARD_InitBootPins();
   BOARD_InitBootClocks();
   BOARD_InitDebugConsole();
    EnableIRQ(BOARD_SW_IRQ);

   /* Print a note to terminal. */
   PRINTF("\r\n GPIO Driver example\r\n");
   PRINTF("\r\n Press %s to turn on/off a LED \r\n", BOARD_SW_NAME);

    /* Init input switch GPIO. */
#if (defined(FSL_FEATURE_PORT_HAS_NO_INTERRUPT) &&
FSL_FEATURE_PORT_HAS_NO_INTERRUPT)
   GPIO_SetPinInterruptConfig(BOARD_SW_GPIO, BOARD_SW_GPIO_PIN,
kGPIO_InterruptFallingEdge);
#else
   PORT_SetPinInterruptConfig(BOARD_SW_PORT, BOARD_SW_GPIO_PIN,
kPORT_InterruptFallingEdge);
#endif

   GPIO_PinInit(BOARD_SW_GPIO, BOARD_SW_GPIO_PIN, &sw_config);

    /* ADC Config */
    adc16_config_t adc16ConfigStruct;
   adc16_channel_config_t adc16ChannelConfigStruct;

    ADC16_GetDefaultConfig(&adc16ConfigStruct);
#ifdef BOARD_ADC_USE_ALT_VREF
   adc16ConfigStruct.referenceVoltageSource = kADC16_ReferenceVoltageSourceValt;
#endif

    /* Initializes the ADC16 module. */
   ADC16_Init(DEMO_ADC16_BASE, &adc16ConfigStruct);
   ADC16_EnableHardwareTrigger(DEMO_ADC16_BASE, false); /* Make sure the software trigger is
used. */
```

```
    /* Calibration */
#if defined(FSL_FEATURE_ADC16_HAS_CALIBRATION) &&
FSL_FEATURE_ADC16_HAS_CALIBRATION
   if (kStatus_Success == ADC16_DoAutoCalibration(DEMO_ADC16_BASE))
   {
      PRINTF(" ADC16_DoAutoCalibration() Done.\r\n");
   }
   else
   {
      PRINTF(" ADC16_DoAutoCalibration() Failed.\r\n");
   }
#endif /* FSL_FEATURE_ADC16_HAS_CALIBRATION */

   PRINTF(" ADC Full Range: %d\r\n", g_Adc16_12bitFullRange);


#if defined(FSL_FEATURE_ADC16_HAS_DIFF_MODE) &&
FSL_FEATURE_ADC16_HAS_DIFF_MODE
   adc16ChannelConfigStruct.enableDifferentialConversion = false;
#endif /* FSL_FEATURE_ADC16_HAS_DIFF_MODE */

   g_Adc16InterruptCounter = 0U;

   while (1)
   {               /* Set Channel_1 of ADC16 */
                   adc16ChannelConfigStruct.channelNumber                    =
DEMO_ADC16_USER_CHANNEL;
                   adc16ChannelConfigStruct.enableInterruptOnConversionCompleted = false;

                   DisableIRQ(BOARD_SW_IRQ);

                   ADC16_SetChannelConfig(DEMO_ADC16_BASE,
DEMO_ADC16_CHANNEL_GROUP, &adc16ChannelConfigStruct);

      while (0U == (kADC16_ChannelConversionDoneFlag &
                ADC16_GetChannelStatusFlags(DEMO_ADC16_BASE,
DEMO_ADC16_CHANNEL_GROUP)))
      {
      }

                   int g_Adc16ConversionValue_ch1;
                   g_Adc16ConversionValue_ch1 =
ADC16_GetChannelConversionValue(DEMO_ADC16_BASE, DEMO_ADC16_CHANNEL_GROUP);

                   EnableIRQ(BOARD_SW_IRQ);

                   /* Read value from Pot_1*/
      PRINTF(" Pot_1: %.2f V\r\n\n", 3.3/4096*g_Adc16ConversionValue_ch1);

                   delay();
   }
}
```