
# University of California, Riverside

# Lab #3 Report

# Interrupt Handling and ADC

EE 128 Section 21, Fall 2025

Name: Muneer Al Jufout

Student ID: 862410258

Lab Partner: Sameer Anjum

Lab Partner ID: 862422332

TA: Johnson Zhang

## Introduction

The objective of this lab is to develop an understanding using the FRDM-K64F development board and implement an interrupt-driven bi-directional counter and ADC voltage display system. Interrupt Service Routine (ISR) that toggles a clock tick indicator on a two 7-segment display, which can update either the ADC value or counter value depending on the switch.

## Experiment System Specification

For this lab, we used Port C for one 7-segment display (excluding pin 6), Port D for the other 7-segment display, and Port A pins 1 and 2 as digital inputs from the DIP switches to control the ADC value and counter value. We also used ADC0 to read the analog voltage from the potentiometer.

## Flowchart

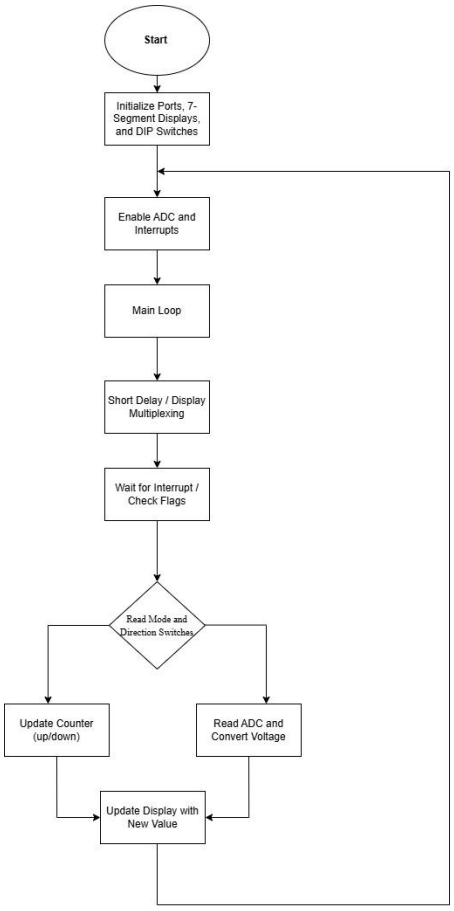
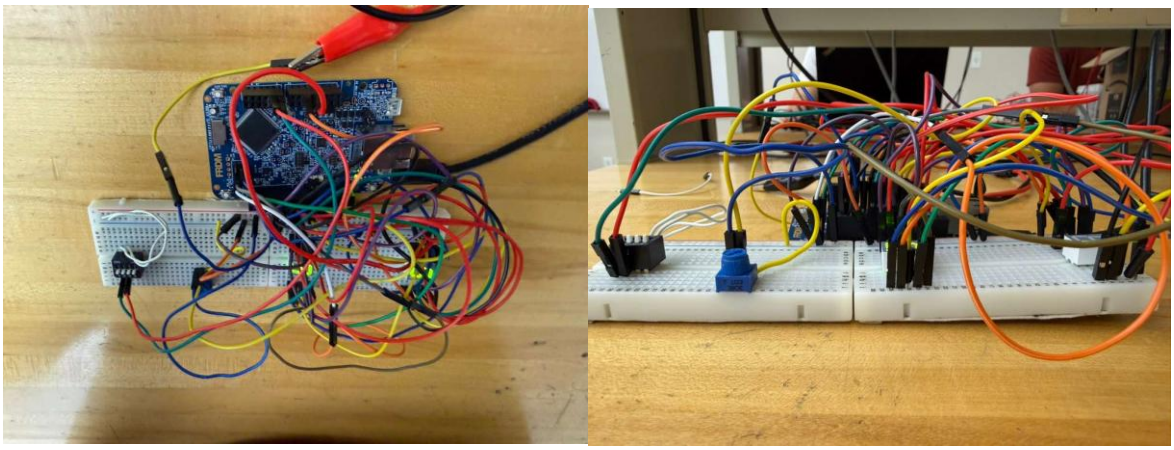



Fig. 1: The Program Flowchart

## Hardware Design

7 Segment displays connected to Port C, and another 7 Segment displays connected to Port D. DIP switch connected to Port A pins 1 and 2 for direction control. Potentiometer connected to ADC0.



a) b)

Fig. 2: Photos of the Setup

## Schematic Diagram

This schematic included:

- MK64FN1M0VLL12 microcontroller
- DIP switch (SW_DIP_4)
- Two 7 Segment displays
- Potentiometer Resistor (R)
- Ground (GND)

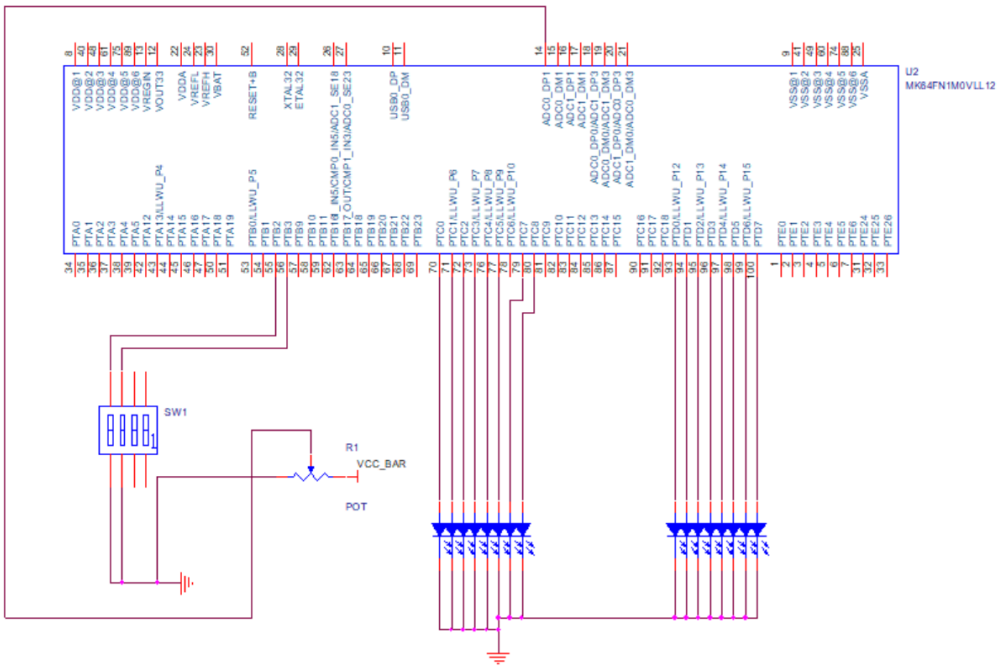



Fig. 3: Schematic Diagram

## High-Level Description

The system uses two 7-segment displays, with one connected to Port C and the other connected to Port D. A DIP switch is connected to Port A pins 1 and 2 to control the counting direction and operating mode. A potentiometer is connected to ADC0 to provide an analog input that is converted into a digital value and displayed on the 7-segment displays.

## Program Listing (Main Sections)

```
#include "fsl_device_registers.h"

volatile int counter = 0;
volatile int clock_tick = 0;
volatile int mode_state = 0;

// These are the references that we call for showing numbers 0-9 on both our 7-segment displays
const unsigned char dict[10] = {
   0x7E, 0x30, 0x6D, 0x79, 0x33,
   0x5B, 0x5F, 0x70, 0x7F, 0x7B
};

/* Show one digit on Port D 7-segment display
  Turns on the decimal point sometimes if clock_tick is set. Supposed to show
```

```
   if voltage is 3.2 first digit is 3 followed by a blinking decimal point */
void displayDigitPortD(int value) {
   uint32_t pattern = dict[value];
   GPIOD_PCOR = 0xFF; // clear all segments

   if (clock_tick) {
      pattern |= 0x80; // turn on decimal point
   } else {
      pattern &= ~0x80; // turn it off
   }

   GPIOD_PSOR = pattern; // light up the segments
}

/* Show one digit on Port C 7-segment display
  Handles the segments and decimal point */
void displayDigitPortC(int value) {
   uint32_t pattern = dict[value];
   GPIOC_PCOR = 0x1BF; // clear all segments
   GPIOC_PSOR = (pattern & 0x3F); // turn on segments A-F

   if (pattern & 0x40) {
      GPIOC_PSOR |= (1 << 7); // turn on segment G if needed
   }
}

/* Show a two-digit number using both displays.
   tiny delay so you can actually see the numbers */
void displayTwoDigits(int value) {
   int tens = (value / 10) % 10;
   int ones = value % 10;

   displayDigitPortD(tens);
   for (volatile int i = 0; i < 300; i++) {} // small delay

   displayDigitPortC(ones);
   for (volatile int i = 0; i < 300; i++) {} // small delay
}

/* Read the analog value from ADC_DP0 pin
  Just grabs the ADC value and waits until it's ready */
static inline uint32_t adc_read_dp0() {
   ADC0->SC1[0] = ADC_SC1_ADCH(0); // start conversion
   while (!(ADC0->SC1[0] & ADC_SC1_COCO_MASK)) {} // wait till done
   return ADC0->R[0]; // return the number
}

/* This runs when the falling edge occurs on PA1
  It flips clock_tick, checks dip switch connected to PB2/PB3, and updates counter */
void PORTA_IRQHandler() {
   PORTA_ISFR = (1 << 1); // clear interrupt flag
```

```
    clock_tick = !clock_tick; // flip decimal point on/off

    uint32_t portb = GPIOB_PDIR;
    mode_state = (portb >> 2) & 1; // read mode button
    uint32_t dir = (portb >> 3) & 1; // read direction button

    if (mode_state == 0) { // ADC mode
        uint32_t raw = adc_read_dp0();
        uint32_t mV = (raw * 3300UL) / 65535UL; // convert to mV
        counter = (int)((mV + 50) / 100); // round to nearest 100mV

        if (counter > 33) counter = 33; // limit max
        if (counter < 0) counter = 0;   // limit min
    } else { // button mode
        if (dir == 0) { // up button
            counter++;
            if (counter > 99) counter = 0;
        } else { // down button
            counter--;
            if (counter < 0) counter = 99;
        }
    }
}

/* Set up some pins on a port as regular GPIO
   mask = which pins you want to change */
void setupPort(volatile uint32_t *port, uint32_t mask) {
    for (int i = 0; i < 32; i++) {
        if (mask & (1 << i)) {
            port[i] = PORT_PCR_MUX(1); // make it GPIO
        }
    }
}

int main() {
    // Turn on clocks for all the ports we need and the ADC
    SIM_SCGC5 |= SIM_SCGC5_PORTA_MASK |
                 SIM_SCGC5_PORTB_MASK |
                 SIM_SCGC5_PORTC_MASK |
                 SIM_SCGC5_PORTD_MASK;
    SIM_SCGC6 |= SIM_SCGC6_ADC0_MASK;

    // Set up the 7-segment displays
    setupPort(PORTD_BASE_PTR, 0xFF);
    GPIOD_PDDR = 0xFF; // set all as output
    GPIOD_PCOR = 0xFF; // turn everything off

    setupPort(PORTC_BASE_PTR, 0x1BF);
    GPIOC_PDDR |= 0x1BF;
    GPIOC_PCOR = 0x1BF;
```

```
    // Set up pushbuttons with pull-ups
    PORTB_PCR2 = PORT_PCR_MUX(1) | PORT_PCR_PE_MASK | PORT_PCR_PS_MASK;
    PORTB_PCR3 = PORT_PCR_MUX(1) | PORT_PCR_PE_MASK | PORT_PCR_PS_MASK;
    GPIOB_PDDR &= ~((1 << 2) | (1 << 3)); // make inputs

    // Set up PA1(clock pin) interrupt
    PORTA_PCR1 = PORT_PCR_MUX(1) | PORT_PCR_PE_MASK |
            PORT_PCR_PS_MASK | PORT_PCR_IRQC(0xA);
    GPIOA_PDDR &= ~(1 << 1);
    PORTA_ISFR = (1 << 1);

    NVIC_EnableIRQ(PORTA_IRQn);
    __enable_irq();

    // Configure and calibrate the ADC(Potentiometer)
    ADC0->CFG1 = ADC_CFG1_MODE(3) | ADC_CFG1_ADICLK(0) | ADC_CFG1_ADLSMP_MASK;
    ADC0->SC1[0] = ADC_SC1_ADCH(31);
    ADC0->SC3 = ADC_SC3_CAL_MASK;
    while (ADC0->SC3 & ADC_SC3_CAL_MASK) {} // wait for calibration

    // Loop forever
    while (1) {
      displayTwoDigits(counter);
    }
}
```

## Technical Problems Encountered and Solutions

During the lab, we initially had trouble with the 7-segment display. At first, we assumed the issue was in our code because it was displaying incorrect numbers. However, after carefully testing the connections, we discovered that it was a pin issue; we did not plug them in correctly. We fixed the problem by putting the cables into the correct pins.

## Questions

1. Measure the time duration from the falling edge of the CLK signal to the change of the decimal point (from on to off, or vice versa) on a 7-segment display. (Use oscilloscope)

   The measured time from the falling edge of the clock signal to the change of the decimal point on the 7-segment display was approximately 2 milliseconds.

2. For this lab, we use a falling-edge generated interrupt. If you use level-sensitive interrupts, what extra steps (in hardware and/or software) would you need? (don’t write the entire code or schematic; answer in a brief manner)

The interrupt flag would stay active as long as the input level remains low. To handle that issue using the code, we need to clear the interrupted flag inside the ISR. In hardware, we need to be released before the interrupt is re-armed.

3. What are the factors contributing to the interrupt latency? Answer briefly.

   The main things that cause interrupt delay are the time the CPU takes to find the interrupt, the time to start and finish the ISR, delays inside the CPU, and if other interrupts are happening at the same time.

4. What is the resolution (minimum distinguishable input voltage) of the ADC implemented in this lab?

   The resolution can be calculated by dividing 3.3 volts by $\mathbf{2^{16}}$. This means the smallest voltage difference the ADC can detect is 50 μV.

## Conclusions

In the laboratory, we were able to gain experience in using the FRDM-K64F microcontroller. We correctly installed and configured the software tools, created and constructed a basic I/O program, and performed circuit connections via 7-segment displays and DIP switches. We implemented a two-digit counter and an ADC voltage display, switched their direction through switches, and observed their functionality through the 7-segment display outputs. We also used a function generator to see the behavior of the system.