Lab 2: Interrupts

ECE342, Winter 2026

Department of Electrical and Computer Engineering

University of Toronto

Introduction

This lab revisits the concept of interrupts, first introduced in ECE243, and extends your practical experience with hardware interaction. In the first part, you will configure and use interrupts to detect button presses on the development board. In the second part, you will interface with a 4×3 matrix keypad to input data and explore how interrupts streamline user interaction. Together, these exercises reinforce your understanding of event-driven programming in embedded systems. Learning objectives include:

- Explain the role of external interrupts in embedded systems
- Implement GPIO-based interrupt service routines
- Interface with and program matrix keypads
- Apply UART communication for data transfer

1. Part I: Button Interrupt

1.1 Overview

In this part of the lab, interrupts will be configured to detect when the blue pushbutton (labeled $B1$ on the development board) is pressed. Because the button is an external device connected to the microcontroller, pressing it generates an external interrupt request that must be handled by the processor.

1.2 Technical Background

The STM32F446ZE microcontroller supports up to 23 external interrupt sources. To conserve chip area and simplify routing, multiple sources are grouped into shared interrupt lines. For example, external interrupt sources 10–15 are combined into a single line, `EXTI15_10`. When any of these sources triggers, the shared interrupt handler executes, and the software must determine which specific source caused the interrupt.

1.3 Implementation Steps

1. **Setup:** Download the provided `Lab_2_kit.zip` file, extract it into a new folder, and open the project in your IDE.
2. **Locate the Interrupt Handler:** In the Project pane, open `stm32f4xx_it.c`. This file contains default interrupt handlers for the board. Scroll to the bottom to find the `EXTI15_10_IRQHandler()` function.
3. **Examine the Handler:** The provided code includes the following template:

```
// Check if interrupt was triggered by the pushbutton
if (__HAL_GPIO_EXTI_GET_FLAG(USER_Btn_Pin)) {
 // TODO: Add your code here

 // Clear the interrupt flag
 __HAL_GPIO_EXTI_IRQHandler(USER_Btn_Pin);
}
```

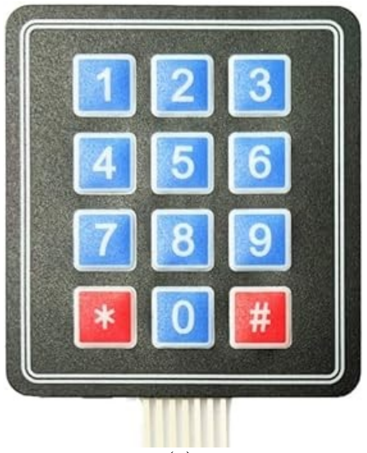
4. **Key Functions:**
   - `__HAL_GPIO_EXTI_GET_FLAG(USER_Btn_Pin)`: Returns 1 if the pushbutton interrupt was triggered.
   - `__HAL_GPIO_EXTI_IRQHandler(USER_Btn_Pin)`: Clears the interrupt flag, allowing future interrupts to be detected.
5. **Your Task:** Add code inside the handler to toggle the green LED each time the button is pressed.

Try it yourself: Modify your Lab 0 code so that each button press cycles the LED blink rate. For example, loop through delays of 0.5 s, 1 s, and 2 s between blinks.

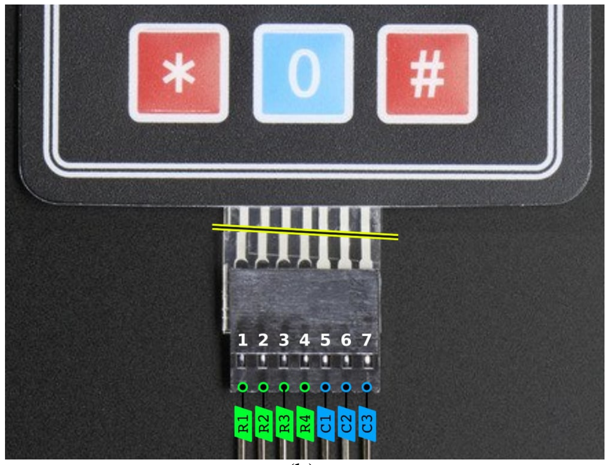
2. Part II: Keypad

2.1 Overview

In this part of the lab, you will interface with an AdaFruit Membrane Matrix Keypad (Figure 1(a)) as an input device for your microcontroller system.



(a)



(b)

Figure 1. (a) Front view of the 3×4 keypad showing button layout. (b) Back view showing pinout labels R1–R4 and C1–C3.

2.2 Keypad Hardware Architecture

A keypad consists of pushbuttons arranged in a grid. To minimize wiring and reduce manufacturing complexity, most keypads use a matrix architecture (Figure 2) rather than dedicating a wire to each button. The matrix design offers several advantages:

- Reduces wire count from $N \times M$ to $N + M$ (for an $N \times M$ grid).
- Simplifies PCB routing and connector requirements.
- Lowers manufacturing costs.

The trade-off is that matrix keypads can reliably detect only one key press at a time and require scanning algorithms to identify pressed keys.

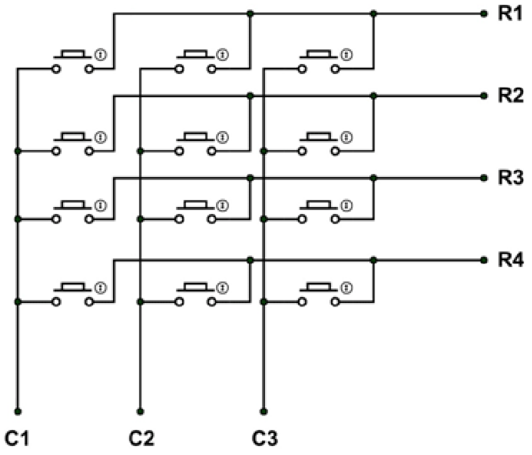


Figure 2. Internal matrix structure of a 3×4 keypad module, showing row and column connections with pushbuttons at intersections. More details are available at `https://www.adafruit.com/product/3845`.

2.3 Detection Algorithm

Keypad detection follows a row-by-row scanning process:

1. **Initial Configuration:**
   - Configure all columns as inputs with pull-down resistors.
   - Configure all rows as outputs, initially set LOW.
   - All columns should read LOW due to the pull-downs.
2. **Scanning Process:**
   - Drive one row HIGH while keeping the others LOW.
   - Read all column inputs.
   - A column reading HIGH indicates a button press at that row–column intersection.
   - Repeat for each row sequentially.
3. **Key Identification:**
   - The active row combined with the responding column uniquely identifies the pressed key.

2.4 Hardware Connection

Follow these steps to connect the keypad to the NUCLEO-F446ZE board:

1. **Physical Connections** Connect male-to-female jumper wires from your kit to the identified header pins.
2. **Software Configuration** The project is pre-configured with labels `ROW1--4` and `COL1--3`. Verify pin assignments in `main.h`. For example:

   ```
   #define ROW1_Pin GPIO_PIN_8
   #define ROW1_GPIO_Port GPIOC
   ```

   This means `ROW1` is connected to pin `PC_8`, located on the `CN8` header of the NUCLEO-F446ZE board (Figure 3). Use this approach to identify the remaining keypad pins.
3. **Board Pin Mapping** Using the CN8 and CN9 headers pinout (Figure 3), connect each keypad jumper wire to the appropriate NUCLEO-F446ZE pin.

2.5 Software Implementation

The keypad interface requires two complementary software components: (1) a scanning loop in `main.c` to drive rows and check columns, and (2) interrupt handlers in `stm32f4xx_it.c` to respond quickly when a key press is detected.

Main Loop (`main.c`)

Implement a continuous scanning routine that:

1. Iterates through all rows (`ROW1--4`).
2. Sets one row HIGH at a time, keeping the others LOW.
3. Waits briefly (a few microseconds) to allow signals to settle.
4. Reads the state of all column inputs.
5. Repeats indefinitely to ensure all key presses are detected.

This loop ensures that the active row is known whenever a column interrupt occurs.

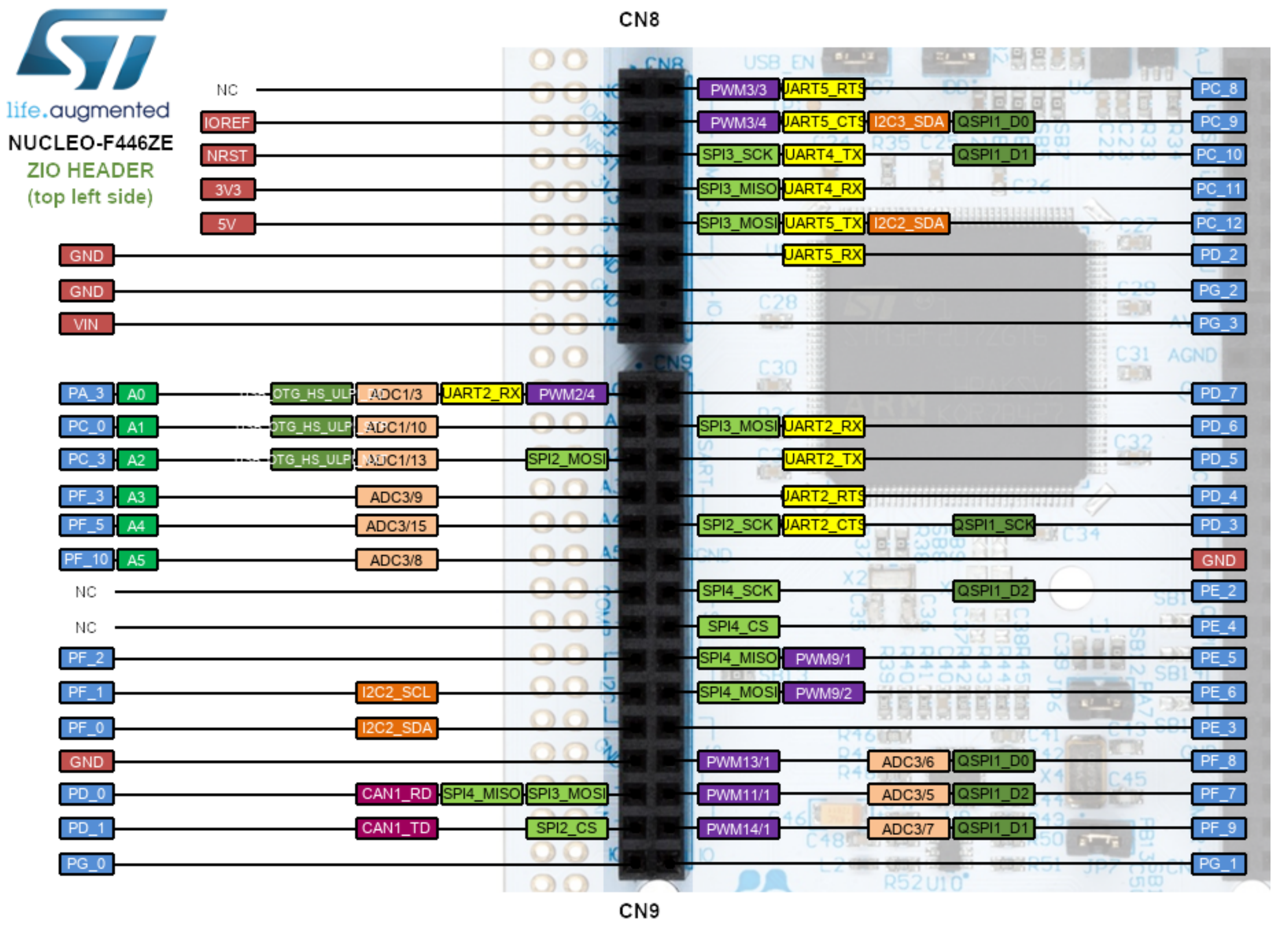


Figure 3. Nucleo F446ZE CN8 and CN9 headers pinout. *Note: Complete board documentation is available at* *`https://os.mbed.com/platforms/ST-NucleoF446ZE/`*..

Interrupt Handlers (`stm32f4xx_it.c`)

Column lines are configured to generate interrupts when a key press pulls them HIGH. Two handlers are required:

- `EXTI9_5_IRQHandler()`: Handles interrupts from `COL1--3`.

Handler responsibilities:

1. Identify which column triggered the interrupt.
2. Pass this column information to the main loop (e.g., via a shared variable or flag).
3. Combine the active row (from the scanning loop) with the detected column to determine the pressed key.
4. Clear the interrupt flag to allow future detections.
5. Send the key value to the serial port using UART.

The main loop then combines `activeRow` and `activeColumn` to map the pressed key (e.g., row 2 + column 1 = key '5').

Try it yourself: Key presses may sometimes register multiple times due to *bouncing*, a mechanical effect in pushbuttons. This settles within a few milliseconds. How could you add simple software debouncing to make detection more reliable?

Try it yourself: In order the keep the CPU free for other tasks, interrupts should be kept as short as possible. Printing to the serial port can be slow. How could you modify your keypad implementation to minimize the work done in the interrupt handlers? (Hint: consider using a buffer or flag to defer processing to the main loop.)

Try it yourself: Build a basic calculator using the keypad for input and the serial port for output. Use # for addition (+), and * for multiplication (×). Display results via UART.

Try it yourself: For easier debugging, you can add a timestatmp to each message printed to the serial port. How could you implement this feature? (Hint: consider using the SysTick timer.)

3. Deliverable

Display the correct serial output for every key on the keypad. Each key press should produce its corresponding character or assigned function over UART.