CSE211s: Introduction to Embedded Systems

Project Report

Team Members	
Name	**ID**
Mazen Ashraf Mohamed	2200822
Yasser Mohamed Abdallah	2200732

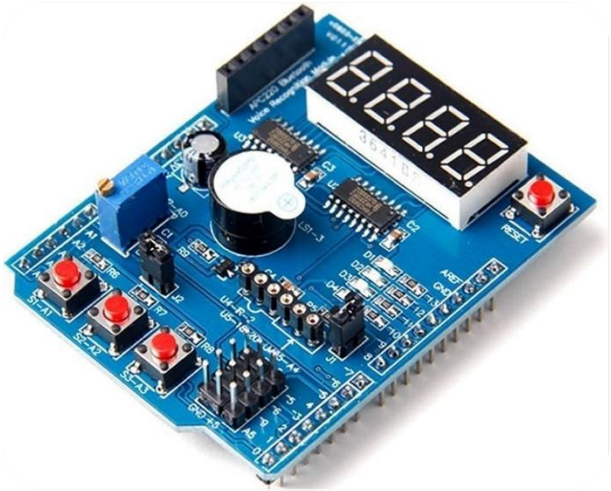
1. Introduction:

This document presents a detailed explanation of a project that uses the Arduino Multi-Function Shield with the NUCLEO-F401RE board. The shield includes a 4-digit 7-segment display, buttons, and a potentiometer. The implemented project includes a real-time clock (RTC) that displays elapsed time and can show an analog voltage reading when a button is pressed.

2. Hardware:

1.1. The Arduino Multi-Function Shield used in this project includes the following components:

- **4-digit 7-segment display (used to show time and voltage)**
- **Onboard potentiometer (used to input analog voltage)**
- **Tactile push buttons S1–S4 (S1 used for reset, S3 to toggle voltage view)**
- **Buzzer and LED (not used in this project)**
- **Two shift registers to control display digits via serial communication**



4-digit 7-segment display and two shift registers to control display digits via serial communication:

Group of Four 7-Segment LED Displays Controlled by 74HC595

- **Latch Write Control: Connected to pin 4**
- **Start write: set LOW**
- **End write: set HIGH**
- **Clock Pulse (Shift Clock): Connected to pin 7**
- **Serial Data Input: Connected to pin 8 Data is written as two 8-bit values (2 bytes)**
- **First Byte: Controls which segments light up**
  Segment ON LOW, Segment OFF HIGH Bits correspond to individual segments (a-g + DP).
- **Second Byte: Selects the active digit (display position)**
  - **Digit 1 (leftmost): 0001000**
  - **Digit 2: 0000100**
  - **Digit 3: 0000010**
  - **Digit 4 (rightmost): 0000001**

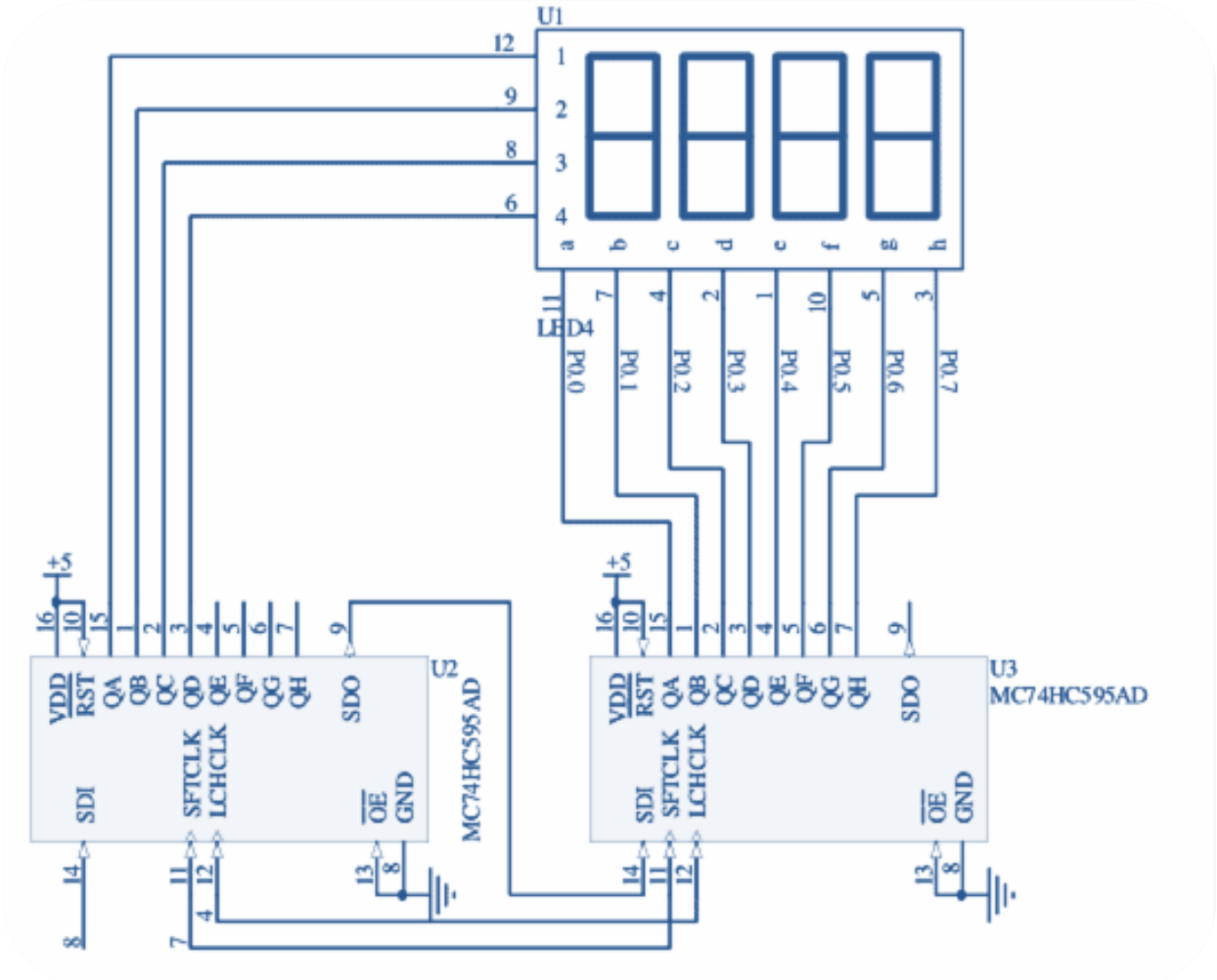


Fig.1 : Schematic diagram of the Multi-Function Shield
(4-digit 7-segment and two shift registers connections)

Onboard potentiometer:

10kΩ Potentiometer (Trimmer Resistor) -Connected between +5V (reference voltage) and GND, with the wiper connected to pin A0

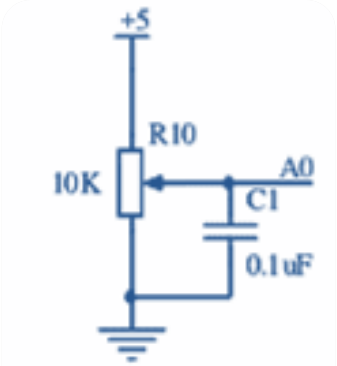


Fig.2 : Schematic diagram of the Multi-Function Shield (Onboard potentiometer connections)

Tactile push buttons S1–S4:

(S1 used for reset, S3 to toggle voltage view)
Input Buttons-Connected to pins A1-A3 (pressed = LOW signal)
Button Description (from left to right):

- **Button S1 pin A1**
- **Button S2 pin A2**
- **Button S3 pin A3**

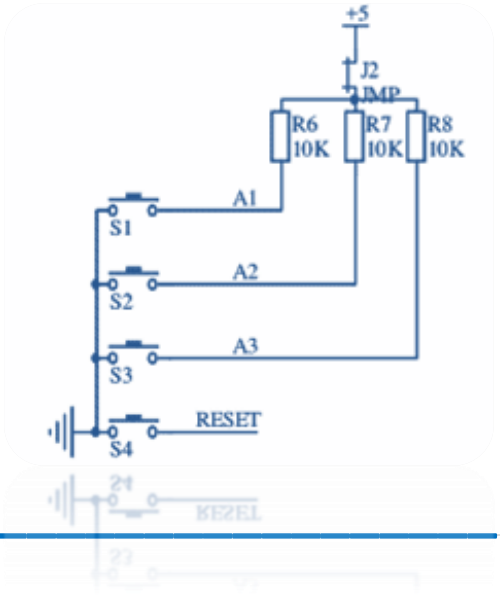
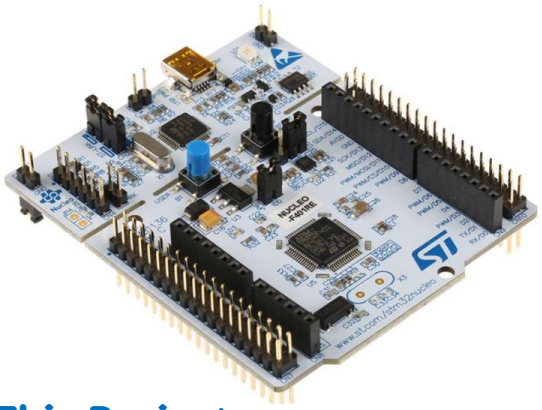


Fig.3 : Schematic diagram of the Multi-Function Shield (Tactile push buttons S1–S4connections)

1.2. NUCLEO-F401RE Board Overview:

The NUCLEO-F401RE is a development board from STMicroelectronics, built around the STM32F401RE microcontroller featuring a 32-bit ARM Cortex-M4 core running at 84 MHz. In this project, it functions as the main controller, interfacing with the Arduino Multi-Function Shield to manage display and input functionality.



Relevant Features for This Project:

❖ Analog Input Pins:

The board offers multiple analog input channels. In this setup:

1. **A0 reads the potentiometer voltage**
2. **A1 is connected to Switch 1, used to reset the real-time clock**
3. **A2 is connected to Switch 3, used to toggle between time and voltage display modes**

❖ Digital I/O Pins:

Used to control a 74HC595 shift register, which drives the 4-digit 7-segment display. Specifically:

1. **D4 → Data (SER)**
2. **D7 → Clock (SRCLK)**
3. **D8 → Latch (RCLK)**

- **High-Speed Processing:**
  The 84 MHz CPU ensures precise timekeeping, responsive inputs, and smooth display updates.
- **Mbed OS Compatibility:**
  Fully supported in Mbed Studio, enabling easy development using modern C++ APIs.
- **USB Interface:**
  The board is powered through USB, which also facilitates serial communication for debugging and monitoring.

3. Code Explanation:

1.3. Overview of the Startup Code :

The startup file for a microcontroller like the STM32F401XE defines the interrupt vector table, which maps exception names to their corresponding handlers. It also includes the reset handler, which is responsible for performing basic system initialization tasks and eventually jumping to the main() function.

Key Tasks Performed:

- **Setting up the stack pointer:** **The initial stack pointer is assigned to the top of the stack memory.**
- **Reset handler:** **The Reset_Handler function initializes the system, sets up peripherals, and jumps to the main() function.**
- **Interrupt vector table:** **The table is created, mapping each interrupt or exception to its respective handler.**

Detailed Breakdown of the Code:

❖ Header Information

At the top of the file, the following header comment section provides metadata about the code:

```
;******************************************************************************
;* File Name          : startup_stm32f401xe.s
;* Author             : MCD Application Team
;* Description        : STM32F401xe devices vector table for MDK-ARM toolchain.
;*                      This module performs:
;*                      - Set the initial SP
;*                      - Set the initial PC == Reset_Handler
;*                      - Set the vector table entries with the exceptions ISR address
;*                      - Branches to __main in the C library (which eventually
;*                        calls main()).
;*                      After Reset the CortexM4 processor is in Thread mode,
;*                      priority is Privileged, and the Stack is set to Main.
;******************************************************************************
;* @attention
;*
;* <h2><center>&copy; Copyright (c) 2017 STMicroelectronics.
;* All rights reserved.</center></h2>
;*
;* This software component is licensed by ST under BSD 3-Clause license,
;* the "License"; You may not use this file except in compliance with the
;* License. You may obtain a copy of the License at:
;*                        opensource.org/licenses/BSD-3-Clause
;*
;******************************************************************************
;* <<< Use Configuration Wizard in Context Menu >>>
```

This is just metadata and licensing information from STMicroelectronics, and is part of their system startup files.

❖ Stack Pointer and Vector Table Initialization:

```
AREA    RESET, DATA, READONLY
EXPORT  __Vectors
EXPORT  __Vectors_End
EXPORT  __Vectors_Size
```

- ❖ **AREA RESET, DATA, READONLY:** Defines the RESET section as data that is readable but not writable.
- ❖ **EXPORT:** Exports the vector table to make it accessible from other parts of the program. The three labels __Vectors, __Vectors_End, and __Vectors_Size are exported to be used in other parts of the project.

❖ The next part of the code defines the vector table, which is a list of addresses pointing to various handlers for exceptions and interrupts:

```
DCD    |Image$$ARM_LIB_STACK$$ZI$$Limit|          ; Top of Stack
DCD    Reset_Handler            ; Reset Handler
DCD    NMI_Handler              ; NMI Handler
DCD    HardFault_Handler        ; Hard Fault Handle
DCD    MemManage_Handler        ; MPU Fault Handler
DCD    BusFault_Handler         ; Bus Fault Handler
DCD    UsageFault_Handler       ; Usage Fault Handler
DCD    0                        ; Reserved
DCD    0                        ; Reserved
DCD    0                        ; Reserved
DCD    0                        ; Reserved
DCD    SVC_Handler              ; SVCall Handler
DCD    DebugMon_Handler         ; Debug Monitor Handler
DCD    0                        ; Reserved
DCD    PendSV_Handler           ; PendSV Handler
DCD    SysTick_Handler          ; SysTick Handler
```

- **DCD (Define Constant Doubleword): Defines a 32-bit value. Each entry here is a 32-bit pointer to the respective interrupt or exception handler.**
- **The first entry points to the top of the stack (|Image$$ARM_LIB_STACK$$ZI$$Limit|), which is where the stack pointer (SP) will point to on reset.**
- **The following entries point to various handlers, including:**
  - **Reset handler: The first handler, Reset_Handler, will execute after a reset.**
  - **Exception handlers: For NMI, HardFault, BusFault, UsageFault, etc.**
  - **stem handlers: Such as SVC_Handler, PendSV_Handler, and SysTick_Handler.**

External Interrupt Handlers:

These are additional interrupt handlers for external events (peripherals and other system events):

```
DCD    WWDG_IRQHandler              ; Window WatchDog
DCD    PVD_IRQHandler             ; PVD through EXTI Line detection
DCD    TAMP_STAMP_IRQHandler          ; Tamper and TimeStamps through the EXTI line
DCD    RTC_WKUP_IRQHandler          ; RTC Wakeup through the EXTI line
DCD    FLASH_IRQHandler            ; FLASH
DCD    RCC_IRQHandler             ; RCC
DCD    EXTI0_IRQHandler            ; EXTI Line0
DCD    EXTI1_IRQHandler           ; EXTI Line1
DCD    EXTI2_IRQHandler            ; EXTI Line2
DCD    EXTI3_IRQHandler            ; EXTI Line3
DCD    EXTI4_IRQHandler            ; EXTI Line4
```

- These handlers correspond to various external interrupts triggered by peripherals like timers, communication interfaces, or sensors.
- Each entry in the vector table points to a handler function that will execute when a corresponding interrupt is triggered.

Reset_Handler Function:

The Reset_Handler is the first function that gets executed when the microcontroller is reset. It typically performs system initialization tasks, such as configuring the clock system, setting up memory, and preparing the environment for the main application. After initialization, it jumps to the main() function.

```
Reset_Handler   PROC
                EXPORT  Reset_Handler          [WEAK]
        IMPORT  SystemInit
        IMPORT  __main

                LDR     R0, =SystemInit
                BLX     R0
                LDR     R0, =__main
                BX      R0
                ENDP
```

- LDR R0, =SystemInit: Loads the address of SystemInit into register R0.
- BLX R0: Branches to SystemInit, a function that is usually provided in the startup code of the STM32 HAL, responsible for setting up the system clock and other hardware configurations.
- LDR R0, =__main: Loads the address of the main() function into register R0.
- BX R0: Branches to the main() function.

❖ Dummy Handlers:

The handlers for exceptions like NMI_Handler, HardFault_Handler, etc., are dummy handlers in this case. These are defined as infinite loops (B .), which means if an interrupt or exception occurs and there's no specific handler, the processor will enter an infinite loop. These are placeholders:

```
NMI_Handler     PROC
                EXPORT  NMI_Handler             [WEAK]
                B       .                       ; Infinite loop if NMI occurs
                ENDP
```

❖ Default Handlers:

The Default_Handler section is another set of handlers for interrupts that might be triggered but don't have a custom implementation. These are also infinite loops:

```
Default_Handler PROC
                EXPORT  WWDG_IRQHandler         [WEAK]
                EXPORT  PVD_IRQHandler          [WEAK]
                EXPORT  TAMP_STAMP_IRQHandler   [WEAK]
                ; More handlers for interrupts
                B       .  ; Infinite loop
                ENDP
```

❖ Interrupts and Exceptions:

- **❖ System Exceptions:** These include the Hard Fault, Bus Fault, Usage Fault, etc. These are critical system exceptions, and when they occur, the program is halted (usually in an infinite loop).
- **❖ External Interrupts:** These are generated by peripheral devices (e.g., timers, GPIO pins) and handled by interrupt service routines (ISRs) like EXTI0_IRQHandler, TIM1_IRQHandler, etc.

❖ End of File:

```
END
```

This directive marks the end of the file. All code and data (like the vector table) up to this point will be included in the compiled binary.

2.3. Code

Here is a structured breakdown and detailed explanation of your Mbed OS project code, block by block. The code uses a 74HC595 shift register to control a 4-digit 7-segment display, showing either a real-time clock (minutes:seconds) or an analog voltage when a button is pressed.

❖ Header and Pin Definitions

```
#include "mbed.h"

// Shift register pins
DigitalOut serPin(D8);    // Serial data
DigitalOut clkPin(D7);    // Clock
DigitalOut latchPin(D4);  // Latch

// Inputs
AnalogIn voltagePin(A0);
DigitalIn s1Switch(A1);  // Reset
DigitalIn s3Switch(A3);  // Show voltage
```

Explanation:

- Initializes the mbed.h library.
- Sets digital pins for 74HC595 shift register: SER, CLK, LATCH.
- Configures analog pin A0 to read potentiometer voltage.
- Configures digital inputs for two pushbuttons:
  - S1 (A1) to reset time.
  - S3 (A3) to show voltage instead of time.

❖ Segment and Digit Data

```
const uint8_t seg_digits[] = {
    0b11000000, // 0
    0b11111001, // 1
    0b10100100, // 2
    0b10110000, // 3
    0b10011001, // 4
    0b10010010, // 5
    0b10000010, // 6
    0b11111000, // 7
    0b10000000, // 8
    0b10010000  // 9
};

const uint8_t digit_sel[4] = {
    0b00000001,  // D0 (rightmost)
    0b00000010,  // D1
    0b00000100,  // D2
    0b00001000   // D3 (leftmost)
};
```

Explanation:

- **seg_digits[]:** Each 8-bit binary represents segment patterns for numbers 0–9 on a common anode 7-segment display. Active LOW means 0 turns ON a segment.
- **digit_sel[]:** Activates each digit individually using the digit enable line of the display (via the second 74HC595 chip).

❖ shiftOut16(): Send data to shift register

```
void shiftOut16(uint8_t seg_data, uint8_t digit_mask) {
    latchPin = 0;

    for (int i = 7; i >= 0; i--) {
        serPin = (seg_data >> i) & 0x01;
        clkPin = 1;
        clkPin = 0;
    }

    for (int i = 7; i >= 0; i--) {
        serPin = (digit_mask >> i) & 0x01;
        clkPin = 1;
        clkPin = 0;
    }

    latchPin = 1;
}
```

Explanation:

- ❖ **Sends 16 bits to two daisy-chained 74HC595 chips:** 8 for segments, 8 for digit selection.
- ❖ **seg_data:** Which segments light up.
- ❖ **digit_mask:** Which digit is activated.
- ❖ **latchPin:** Ensures data appears only after full transmission.

❖ displayVoltPot(): Show voltage

```
void displayVoltPot(int volt_pot) {
    int d[4] = {
        (volt_pot / 1000) % 10,
        (volt_pot / 100) % 10,
        (volt_pot / 10) % 10,
        volt_pot % 10
    };

    for (int i = 0; i < 4; i++) {
        uint8_t seg = seg_digits[d[i]];
        if (i == 1) seg &= ~(1 << 7);  // Add decimal point at 2nd digit
        shiftOut16(seg, digit_sel[i]);
        wait_us(2000);
    }
}
```

Explanation:

- ❖ Displays volt_pot (voltage × 100) as 4 digits.
- ❖ Places a decimal point after the first digit (for format like 3.30V).
- ❖ Loops through each digit and sends segment + digit data.

❖ getStableVoltage(): Read smoothed voltage

```
float getStableVoltage() {
    const int sampleCount = 50;
    float sum = 0;

    for (int i = 0; i < sampleCount; i++) {
        sum += voltagePin.read();  // 0.0 to 1.0
        wait_us(100);
    }

    return (sum / sampleCount) * 5.0f;
}
```

Explanation:

- ❖ Reads 50 samples from analog pin.
- ❖ Averages them to reduce noise and fluctuation.
- ❖ Scales the result to real voltage (assuming 5V reference).

❖ displayTime(): Show clock

```
void displayTime(int min, int sec) {
    int d[4] = {
        (min / 10) % 10,
        min % 10,
        (sec / 10) % 10,
        sec % 10
    };

    for (int i = 0; i < 4; i++) {
        uint8_t seg = seg_digits[d[i]];
        shiftOut16(seg, digit_sel[i]);
        wait_us(2000);
    }
}
```

Explanation:

- Splits min and sec into digits.
- Displays in MMSS format (without colon or decimal).
- Each digit is displayed briefly and repeatedly for persistence of vision.

❖ Global variables

```
long long start_time = 0;
int timesec = 0;
int timemin = 0;
```

Explanation:

- **start_time:** Used to track the passage of real-time using get_ms_count().
- **timesec, timemin:** Clock counters incremented every second.

❖ main(): Core logic

```
int main() {
    start_time = get_ms_count();

    while (true) {
        long long now = get_ms_count();

        // Update time every second
        if (now - start_time >= 1000) {
            start_time = now;
            timesec++;
            if (timesec >= 60) {
                timesec = 0;
                timemin++;
            }
        }

        // Show voltage when S3 is pressed
        if (s3Switch == 0) {
            float voltage = getStableVoltage();
            int volt_int = (int)(voltage * 100);
            displayVoltPot(volt_int);
        } else {
            displayTime(timemin, timesec);
        }

        // Reset on S1 or when minutes reach 100
        if (s1Switch == 0 || timemin >= 100) {
            timemin = 0;
            timesec = 0;
        }
    }
```

Explanation:

- Uses get_ms_count() to track time.
- Increments seconds and minutes based on elapsed time.
- If S3 is pressed (LOW), it displays voltage.
- Otherwise, displays the running clock.
- Resets time when S1 is pressed or 100 minutes pass.

❖ Final Notes

- we did not use ISR (Interrupt Service Routine): All button reading and time handling are done in polling-style inside the main loop. This is simple and acceptable for such lightweight logic, though interrupts could improve responsiveness.
- we using multiplexing to control all digits with limited pins through the shift register.
- Timing (refresh rate) is maintained via wait_us(2000), which helps ensure visible and flicker-free display.

4. Links:

- **GitHub Repository Link :**
  https://github.com/Mazen-Elborhamy/Embedded-Project-.git

- **Video Link:**
  https://drive.google.com/file/d/1s7I7npO2QUiUPna9hscxqAzC01U5uYPd/view?usp=drive_link

- **Startup Code Link :**
  https://github.com/Mazen-Elborhamy/Embedded-Project-/blob/main/Code%20%26%20Startup%20Code/startup%20code.txt

- **Code Link :**
  https://github.com/Mazen-Elborhamy/Embedded-Project-/blob/main/Code%20%26%20Startup%20Code/Code.txt