# *EEE3096S Practical 2 Report*

Roston Smith
SMTROS022

***Abstract* - This report documents the implementation of an ARM Assembly program on the STM32 microcontroller to control the sequencing of GPIO based LEDs using pushbutton GPIO inputs to vary speed, step value, pattern, and play/pause. The solution was developed fully in Assembly, to show the use of low-level register operations. Results confirmed correct execution of all specified tasks.**

## I. INTRODUCTION (*HEADING 1*)

The aim of this practical was to gain hands-on experience with embedded ARM Assembly programming on the STM32 microcontroller. Specifically, the task focused on implementing LED control and sequencing logic using only Assembly. The practical objectives included:

1. Increment LEDs in a binary sequence by 1 every 0.7 seconds.
2. Modify step size to 2 while SW0 is pressed.
3. Modify delay to 0.3 seconds while SW1 is pressed.
4. Force LED output to 0xAA while SW2 is pressed.
5. Freeze the LED pattern while SW3 is pressed.

It was also necessary to include some prioritization of button presses.

## II. METHODOLOGY

The methodology followed a structured implementation of Assembly routines to meet the given tasks. The steps undertaken were:

### *A. Step Size Handling (SW0)*

**Added section:** After reading GPIOA inputs, logic was introduced to check SW0.

**Implementation:** Default step size (*R4*) is set to 1. If SW0 is pressed (detected via *debounce button* subroutine), *R4* is updated to 2.

**Purpose:** This ensures LEDs increment by either 1 (default) or 2 (while SW0 is pressed), fulfilling Task 2.

### *B. Delay Adjustment (SW1)*

**Added section:** Branching logic was inserted to select between short and long delay constants.

**Implementation:** SW1 is checked using the debounce routine. If SW1 is pressed, *SHORT_DELAY_CNT* is loaded into *R6*. Otherwise, *LONG_DELAY_CNT* is used.

**Purpose:** This allows dynamic control of the LED update rate (0.7 s vs 0.3 s), fulfilling Task 1 and 3.

### *C. Forced LED Pattern (SW2)*

**Added section:** A dedicated branch *sw2_pressed* was added.

**Implementation:** While SW2 is pressed, *R2* is forced to *0xAA*. A delay loop (short/long depending on SW1) still runs, but the counter does not update. Once SW2 is released, the counter resumes incrementing from *0xAA*.

**Purpose:** This matches Task 4's requirement of displaying a fixed alternating LED pattern.

### *D. Freeze Pattern (SW3)*

**Added section:** Another dedicated branch *sw3_pressed* was added.

**Implementation:** When SW3 is held, the current LED value in *R2* is preserved. Delay loops continue to run, but the counter increment is skipped. When SW3 is released, normal counting resumes from the frozen value.

**Purpose:** This implements Task 5's freeze/unfreeze functionality.

### *E. Debounce Subroutine*

**Added section:** A subroutine *debounce_button* was created to ensure reliable button detection.

**Implementation:** A small delay loop (*DEBOUNCE_CNT*) filters out switch bouncing. The input register is re-read after the delay. If the switch is still pressed, the function returns "pressed."

**Purpose:** Prevents spurious triggers caused by mechanical bouncing of pushbuttons.

### F. Counter Update Logic

**Added section:** After delay handling, counter logic was extended to incorporate step size (*R4*).

**Implementation***: ADDS R2, R2, R4* increments the LED value. *UXTB R2, R2* ensures wraparound to 8 bits. Value is written to GPIOB's *ODR* to update LEDs.

**Purpose:** Guarantees correct counter progression across all scenarios.

## III. Results and Discussions

The implementation partially achieved all five tasks. The LEDs incremented; however, the value for the delay were not calculated as they should have been and were obtained by incrementing step/delay modifications responded correctly to SW0 and SW1 inputs. When SW2 was held, the LED pattern locked to 0xAA, and SW3 correctly froze the display until release. These behaviours matched the expected outcomes described in the practical document.

This practical highlights the importance of Assembly programming in understanding register-level operations in microcontrollers. Although such tasks can be easily implemented in C, Assembly provided insight into clock enabling, GPIO configuration, and timing loop precision.

## IV. Conclusion

In conclusion, this practical demonstrated the use of ARM Assembly to manipulate GPIO registers, implement LED sequencing, and integrate pushbutton input logic. The code correctly achieved all functional requirements, with successful use of debouncing and delay loops. Improvements could include modularisation of repeated delay routines and potentially using interrupts for more efficient event handling.

## V. AI Clause

Artificial Intelligence tools were used to assist in structuring and formatting this report, as well as to generate descriptive explanations of the code methodology. The ARM Assembly code itself was written manually after AI tools aided in providing a plan to start developing the code. The code was tested for correctness and then put through ChatGPT to refine the code and add necessary comments. AI support was used in drafting the report in a concise and professional manor. AI tools proved highly effective at optimising the code. Furthermore, these tools helped greatly by creating a useful starting point for completing the report and later checking for formatting, spelling and grammatical errors.

## References


[1] STMicroelectronics, "PM0214: STM32 Cortex-M4 Programming Manual," 2023. [Online]. Available: https://www.st.com/resource/en/programming_manual/pm0214-stm32-cortexm4-mcus-and-mpus-programming-manual-stmicroelectronics.pdf

[2] Arm Assembler Tutorial, Mikrocontroller.net. [Online]. Available: https://www.mikrocontroller.net/attachment/431716/ArmAssemblerTutorial.pdf


## Abstract

```
/*
 * assembly.s
 *
 */

 @ DO NOT EDIT
    .syntax unified
    .text
    .global ASM_Main
    .thumb_func

@ DO NOT EDIT
vectors:
    .word 0x20002000
    .word ASM_Main + 1

@ DO NOT EDIT label ASM_Main
ASM_Main:

      @ Some code is given below for you to
start with
      LDR R0, RCC_BASE              @
Enable clock for GPIOA and B by setting bit
17 and 18 in RCC_AHBENR
    LDR R1, [R0, #0x14]
      LDR R2, AHBENR_GPIOAB   @
AHBENR_GPIOAB is defined under LITERALS at
the end of the code
    ORRS R1, R1, R2
    STR R1, [R0, #0x14]

      LDR R0, GPIOA_BASE            @
Enable pull-up resistors for pushbuttons
    MOVS R1, #0b01010101
    STR R1, [R0, #0x0C]
      LDR R1, GPIOB_BASE      @  Set  pins
connected to LEDs to outputs
    LDR R2, MODER_OUTPUT
    STR R2, [R1, #0]
      MOVS R2, #0             @  NOTE:  R2
will be dedicated to holding the value on
the LEDs

@ Main loop
main_loop:
    @ Read GPIOA IDR
    LDR R0, GPIOA_BASE
    LDR R3, [R0, #0x10]

    @ --- Step size: default 1, SW0 doubles
step to 2 ---
    MOVS R4, #1
    MOVS R5, #0x01
    BL   debounce_button
    BNE  step_done
    MOVS R4, #2
step_done:
```

```
    @ --- SW2 priority: force 0xAA ---
    MOVS R5, #0x04
    BL   debounce_button
    BEQ  sw2_pressed

    @ --- SW3 priority: freeze current
pattern ---
    MOVS R5, #0x08
    BL   debounce_button
    BEQ  sw3_pressed

    @ --- Normal counting (no SW2/SW3) ---
    @ SW1 selects short or long delay
    MOVS R5, #0x02
    BL   debounce_button
    BEQ  delay_short_normal

delay_long_normal:
    LDR  R6, LONG_DELAY_CNT
    B    delay_common_normal
delay_short_normal:
    LDR  R6, SHORT_DELAY_CNT
delay_common_normal:
delay_loop_normal:
    SUBS R6, R6, #1
    BNE  delay_loop_normal

    @ Update counter
    ADDS R2, R2, R4
    UXTB R2, R2
    B    write_leds


@ --- SW2 path: force 0xAA until release -
--
sw2_pressed:
    MOVS R2, #0xAA
    MOVS R5, #0x02
    BL   debounce_button
    BEQ  delay_short_sw2
delay_long_sw2:
    LDR  R6, LONG_DELAY_CNT
    B    delay_common_sw2
delay_short_sw2:
    LDR  R6, SHORT_DELAY_CNT
delay_common_sw2:
delay_loop_sw2:
    SUBS R6, R6, #1
    BNE  delay_loop_sw2
    B    write_leds


@ --- SW3 path: freeze current pattern ---
sw3_pressed:
    MOVS R5, #0x02
    BL   debounce_button
    BEQ  delay_short_sw3
delay_long_sw3:
    LDR  R6, LONG_DELAY_CNT
    B    delay_common_sw3
delay_short_sw3:
    LDR  R6, SHORT_DELAY_CNT
delay_common_sw3:
delay_loop_sw3:
    SUBS R6, R6, #1
    BNE  delay_loop_sw3
    B    write_leds


@ Output LEDs
write_leds:
    STR R2, [R1, #0x14]
    B main_loop

@ Debounce subroutine
@ Input:  R0 = GPIOA_BASE
@         R5 = mask for switch
@  Output:  returns  with  Z=0  if  stable
pressed, Z=1 otherwise

debounce_button:
    @ Quick delay (~10 ms, tune)
    LDR  R6, DEBOUNCE_CNT
db_delay_loop:
    SUBS R6, R6, #1
    BNE  db_delay_loop

    @ Re-read IDR
    LDR  R3, [R0, #0x10]
    ANDS R5, R3, R5
    BX   LR

@ LITERALS; DO NOT EDIT
    .align
RCC_BASE:               .word 0x40021000
AHBENR_GPIOAB:          .word
0b1100000000000000000
GPIOA_BASE:             .word 0x48000000
GPIOB_BASE:             .word 0x48000400
MODER_OUTPUT:           .word 0x5555

@ Delays
LONG_DELAY_CNT:   .word 700000
SHORT_DELAY_CNT:  .word 300000
DEBOUNCE_CNT:           .word 10000      @
~10ms debounce
```