Practical 4: Assembly Language Programming with STM32

Georgene de Wet[†] and Arend JAcques du Preez[‡]
EEE3096S **Group 36**
University of Cape Town
South Africa
[†]DPRARE001 [‡]DWTGEO002

***Abstract*—This report covers the implementation of an assembly language program on the STM32 micro controller. The tasks involved incrementing LED values, implementing specific LED patterns, introducing different delays, and freezing the LED display when necessary, which was all successfully achieved. The results of this practical demonstrate the program's ability to perform these functions by using software delay loops. The system met all requirements, with the potential to improve in areas such as optimizing delays with hardware timers being discussed.**

I. Introduction

Assembly language programming is essential in understanding the low-level operations of embedded systems. While C programming (being a high level language) is used for abstracting hardware details, assembly gives direct control over the hardware components, such as micro controller registers and peripherals.By understanding and using assembly,one is exposed to register manipulation, bit-wise operations, and control flow, all being fundamental to low-level programming. Furthermore, understanding assembly is critical for scenarios where precise control and optimization are required, especially in resource constrained environments.

In this practical the objective is to demonstrate how assembly can be used to manage peripherals through direct register access. This is done by manipulating the STM32 micro controller's GPIO ports for controlling LEDs and reading push button inputs.

The tasks of this practical involved programming the STM32 to increment LEDs by 1 or 2 based on button presses, set specific LED patterns (like 0xAA), and freeze the LED display when required. Through this, a deeper understanding of hardware interfacing and timing mechanisms using software-based delay loops is gained.

The source code for this project can be found on GitHub: EmbeddedSystems$_{G}roup36$.

[1]

II. System Overview

A. Hardware Setup

The practical involves interfacing an STM32 microcontroller with four pushbuttons (SW0-SW3) and eight LEDs connected to GPIO pins. The pushbuttons are connected to GPIOA, configured as input pins with pull-up resistors to register button presses as low states. The LEDs are connected to GPIOB, which is configured as output to display different patterns based on the logic in the assembly program. Each button press triggers a specific function such as modifying the LED increment, adjusting the delay, or freezing the current LED state.

The pull-up configuration for the pushbuttons ensures that, in the absence of a press, the inputs are read as high, while pressing a button pulls the input low. This simple yet effective hardware setup allows the assembly code to detect button presses and control the LED behavior through direct manipulation of the GPIO registers.

B. Software Implementation

The software for this practical is written entirely in assembly, providing low-level control of the STM32 microcontroller's hardware. The core of the program involves setting up the clock for the GPIO ports, configuring GPIOA as input for the buttons and GPIOB as output for the LEDs, and implementing a main loop that continuously monitors button presses and updates the LED patterns accordingly. The full assembly code can be referred to in the appendix of this report.A

Clock Setup and GPIO Initialization

To enable the GPIO ports, the clock must first be configured by setting specific bits in the AHBENR (Advanced High-Performance Bus Enable Register). The following assembly code sets up the clock for both GPIOA and GPIOB:

Listing 1: Clock and GPIO Initialization

```
14  LDR R0, RCC_BASE          @ Enable clock for GPIOA and GPIOB
15  LDR R1, [R0, #0x14]
16  LDR R2, AHBENR_GPIOAB
17  ORRS R1, R1, R2
18  STR R1, [R0, #0x14]
```

After enabling the clock, the GPIO ports are configured. GPIOA is set as input to detect button presses, while GPIOB is set as output to control the LEDs:

Listing 2: GPIOA and GPIOB Configuration

```
20  LDR R0, GPIOA_BASE        @ Configure pull-up resistors for pushbuttons
21  MOVS R1, #0b01010101
22  STR R1, [R0, #0x0C]
24  LDR R1, GPIOB_BASE  @ Set LED pins to output mode
25  LDR R2, MODER_OUTPUT
26  STR R2, [R1, #0]
```

This setup ensures that the program can read the state of the push-buttons and update the LEDs by modifying the appropriate GPIO registers.

Main Loop and Button Functionality

The main loop checks the state of the push-buttons and updates the LEDs based on the detected inputs. The default behavior is to increment the LED value by 1 every 0.7 seconds, but pressing specific buttons modifies this behavior:

- **SW0**: (PA0) increases the increment to 2 (lines 50-53).
- **SW1**:(PA1) reduces the delay to 0.3 seconds (lines 57-60).
- **SW2**: (PA2) sets the LED pattern to 0xAA (lines 63-66).
- **SW3**: (PA3) freezes the LED state (lines 70-73).

Below is a snippet of the main loop, which checks the button states:

Listing 3: Main Loop for LED Control

```
30  main_loop:
31      LDR R0, GPIOA_BASE        @ Load GPIOA base address
32      LDR R4, [R0, #0x10]       @ Check pushbutton state
33      MOVS R6, #0b1111          @ Mask for PA0-PA3
34      ANDS R4, R4, R6           @ Apply mask
35      CMP R4, #0b1111           @ Check if no buttons are pressed
36      BEQ increment_default      @ Branch to default increment
```

The program uses a combination of bit-wise operations and conditional branching to determine which button is pressed and update the LED state accordingly. For instance, if SW0 is pressed, the program jumps to the ‘increment_by_two‘ section to increase the LED value by 2, while pressing SW3 causes the current LED state to freeze until the button is released.

III. Results

The assembly code was tested on the STM32 microcontroller to verify the LED responses to button inputs were correct. Button press triggers and their expected changes in LED patterns were checked, as well as all timing delays matching the practical's requirements. Table I summarizes the LED behavior for different button presses.

A. Behavior Analysis

TABLE I: Button Functionality and LED Behavior

Button Pressed	Expected LED Behavior	Time Delay
None	Increment by 1	0.7 seconds
SW0	Increment by 2	0.7 seconds
SW1	Increment by 1	0.3 seconds
SW0 + SW1	Increment by 2	0.3 seconds
SW2	Set LED pattern to 0xAA	N/A
SW3	Freeze current LED state	N/A

As shown in Table I, the system behaved as expected. When no buttons were pressed, the LEDs incremented by 1 every 0.7 seconds. SW0 caused increments of 2 with the same delay, while SW1 reduced the delay to 0.3 seconds. Pressing both SW0 and SW1 together resulted in increments of 2 with the shorter delay.

Further, SW2 displayed the pattern 0xAA, and SW3 froze the current LED state, with normal operation resuming after the button was released.

B. Testing Observations

The system performed exactly as specified. LEDs incremented by 1 every 0.7 seconds when no buttons were pressed, with SW0 altering the increment to 2. SW1 shortened the delay to 0.3 seconds, and pressing both SW0 and SW1 together confirmed the correct implementation of bit wise operations and timing changes.

SW2 immediately displayed the 0xAA pattern, pausing the counting sequence, and SW3 froze the current LED state, both functioning as designed.

C. Timing Delay Verification

The long (0.7 seconds) and short (0.3 seconds) delays were confirmed by using a stopwatch. These delays were accurately implemented by the system, as specified in the assembly code, with no discrepancies observed.

IV. Conclusion

This practical successfully demonstrated the use of assembly language to interface with hardware peripherals on the STM32 microcontroller. The assembly program met all specified requirements, including incrementing LED values, adjusting the timing delay, setting specific LED patterns, and freezing the LED state based on pushbutton inputs.

The practical provided a deeper understanding of how low-level programming can be used to manipulate hardware registers directly, as well as the importance of careful timing control in embedded systems. Although the current implementation is functional, further improvements, such as using hardware timers and implementing interrupt-based logic, could enhance system efficiency and performance in future designs.

Overall, this practical highlighted the power and complexity of assembly language in embedded systems and reinforced the value of understanding low-level hardware interactions for developing efficient and optimized embedded applications.

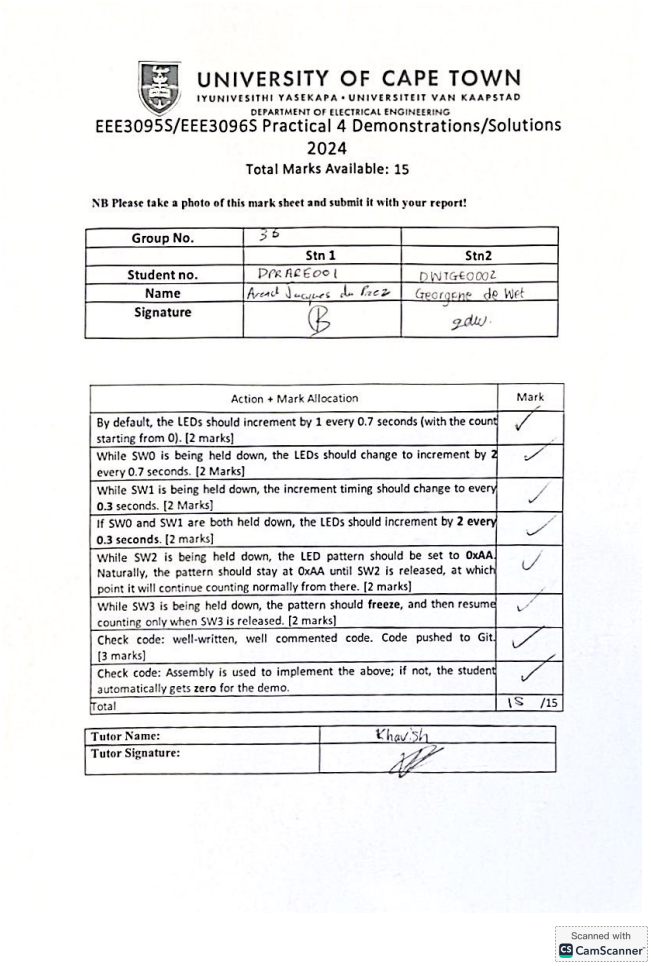
[2]

References

[1] U. o. C. T. Department of Electrical Engineering, “Eee3096s course handout,” Technical Report, January 2024.

[2] “Grammarly,” https://www.grammarly.com, accessed: 2023-08-27.

APPENDIX



UNIVERSITY OF CAPE TOWN
IYUNIVESITHI YASEKAPA • UNIVERSITEIT VAN KAAPSTAD
DEPARTMENT OF ELECTRICAL ENGINEERING

EEE3095S/EEE3096S Practical 4 Demonstrations/Solutions
2024
Total Marks Available: 15

NB Please take a photo of this mark sheet and submit it with your report!

Group No.	35	
	Stn 1	Stn2
Student no.	PRRAEE001	DWTGE002
Name	Arend Jacques du Toit	Georgene de Wet
Signature		

Action + Mark Allocation	Mark
By default, the LEDs should increment by 1 every 0.7 seconds (with the count starting from 0). [2 marks]	✓
While SW0 is being held down, the LEDs should change to increment by 2 every 0.7 seconds. [2 Marks]	✓
While SW1 is being held down, the increment timing should change to every 0.3 seconds. [2 Marks]	✓
If SW0 and SW1 are both held down, the LEDs should increment by 2 every 0.3 seconds. [2 marks]	✓
While SW2 is being held down, the LED pattern should be set to 0xAA. Naturally, the pattern should stay at 0xAA until SW2 is released, at which point it will continue counting normally from there. [2 marks]	✓
While SW3 is being held down, the pattern should freeze, and then resume counting only when SW3 is released. [2 marks]	✓
Check code: well-written, well commented code. Code pushed to Git. [3 marks]	✓
Check code: Assembly is used to implement the above; if not, the student automatically gets zero for the demo.	✓
Total	15 /15

Tutor Name:	Khav Sh
Tutor Signature:	

Scanned with CamScanner

Fig. 1: Lab Sign off

A. Main Assembly Code

The complete assembly code used for this practical is provided below. This code implements the functionality described in the report, including controlling LED patterns based on pushbutton inputs, adjusting the increment and delay, and handling special cases like freezing the LED state.

Listing 4: Main Assembly Code

```
/*
 * assembly.s
 */
.syntax unified
.text
.global ASM_Main
.thumb_func

vectors:
    .word 0x20002000
    .word ASM_Main + 1

ASM_Main:
    LDR R0, RCC_BASE          @ Enable clock for GPIOA and GPIOB
    LDR R1, [R0, #0x14]
    LDR R2, AHBENR_GPIOAB
    ORRS R1, R1, R2
    STR R1, [R0, #0x14]

    LDR R0, GPIOA_BASE        @ Configure pull-up resistors for pushbuttons
    MOVS R1, #0b01010101
    STR R1, [R0, #0x0C]

    LDR R1, GPIOB_BASE        @ Set LED pins to output mode
    LDR R2, MODER_OUTPUT
    STR R2, [R1, #0]

    MOVS R2, #0              @ LED value stored in R2

```

```
main_loop:
    LDR R0, GPIOA_BASE          @ Load GPIOA base address
    LDR R4, [R0, #0x10]         @ Check pushbutton state
    MOVS R6, #0b1111            @ Mask for PA0-PA3
    ANDS R4, R4, R6             @ Apply mask
    CMP R4, #0b1111             @ Check if no buttons are pressed
    BEQ increment_default       @ Branch to default increment

    MOVS R6, #0b1111            @ Mask for PA0
    ANDS R4, R4, R6             @ Check if PA0 (SW0) is pressed
    CMP R4, #0b1110
    BEQ increment_by_two        @ Increment by 2 if PA0 pressed

    MOVS R6, #0b1111            @ Mask for PA1 (SW1)
    ANDS R4, R4, R6
    CMP R4, #0b1101
    BEQ call_short_delay        @ Short delay if PA1 pressed

    MOVS R6, #0b1111            @ Mask for PA2 (SW2)
    ANDS R4, R4, R6
    CMP R4, #0b1011
    BEQ set_led_to_aa           @ Set LED pattern to 0xAA if PA2 pressed

    MOVS R6, #0b1111            @ Mask for PA3 (SW3)
    ANDS R4, R4, R6
    CMP R4, #0b0111
    BEQ check_freeze            @ Freeze LED if PA3 pressed

increment_by_two:
    ADDS R2, R2, #2             @ Increment by 2
    BL long_delay               @ Long delay
    B mask_leds                 @ Apply LED mask

call_short_delay:
    ADDS R2, R2, #1             @ Increment by 1
    BL short_delay              @ Short delay
    B mask_leds

set_led_to_aa:
    MOVS R3, #0xAA              @ Set LED to 0xAA
    LDR R0, GPIOB_BASE
    STR R3, [R0, #0x14]
    B main_loop

increment_default:
    ADDS R2, R2, #1             @ Default increment by 1
    BL long_delay
    B mask_leds

mask_leds:
    MOVS R3, #0xFF              @ Mask LED output to 8 bits
    ANDS R2, R2, R3
    B write_leds

write_leds:
    LDR R0, GPIOB_BASE
    STR R2, [R0, #0x14]         @ Update LED state
    B main_loop

check_freeze:
    LDR R0, frozen_state        @ Check frozen state
    LDR R1, [R0]
    CMP R1, #0                  @ If not frozen, update LEDs
    BEQ mask_leds
    B main_loop

long_delay:
    LDR R3, LONG_DELAY_CNT      @ Load long delay count
delay_loop:
    SUBS R3, R3, #1             @ Decrement counter
    BNE delay_loop
    BX LR                       @ Return to main loop

short_delay:
    LDR R3, SHORT_DELAY_CNT     @ Load short delay count
short_delay_loop:
    SUBS R3, R3, #1
    BNE short_delay_loop
    BX LR

.align
RCC_BASE:           .word 0x40021000
AHBENR_GPIOAB:      .word 0b1100000000000000000
GPIOA_BASE:         .word 0x48000000
GPIOB_BASE:         .word 0x48000400
MODER_OUTPUT:       .word 0x5555
LONG_DELAY_CNT:     .word 1400000
SHORT_DELAY_CNT:    .word 600000
frozen_state:       .word 0
```