# TEAM:

Harish Krishna-CB.EN.U4CSE22417

Nitish Krishna-CB.EN.U4CSE22428

Naveen Kumar-CB.EN.U4CSE22436

Yuva Praneeth-CB.EN.U4CSE22459

# CODE:

#include "stm32f4xx.h"

void USART1\_Init(void);

void USART1\_Write(char ch);

char USART1\_Read(void);

void SysTick\_Handler(void);

volatile int blink\_C14 = 0; // Blinking state for C14

volatile int blink\_C15 = 0; // Blinking state for C15

volatile int led\_on = 0;

volatile int toggle\_led = 0;

volatile int counter = 599;

volatile int static\_counter = 599 ; // Secondary counter for displaying when PB10 is pressed

volatile int pause = 0; // Flag to pause both counters

const uint8\_t segment\_map[10] = {

0b00111111, // 0

0b00000110, // 1

0b01011011, // 2

0b01001111, // 3

0b01100110, // 4

0b01101101, // 5

0b01111101, // 6

0b00000111, // 7

0b01111111, // 8

0b01101111 // 9

};

void display2(int digit) {

uint8\_t segments = segment\_map[digit];

// Clear all segment bits first (for segments connected to GPIOA and GPIOB)

GPIOA->ODR &= ~(1U << 7); // Clear PA7 (segment G)

GPIOB->ODR &= ~(1U << 0 | 1U << 1 | 1U << 2 | 1U << 7 | 1U << 8 | 1U << 9); // Clear PB0 to PB2 and PB7 to PB9

// Set each segment based on the bit values in segments

if (segments & 0b00000001) GPIOB->ODR |= (1U << 7); // Segment a (mapped to PB7)

if (segments & 0b00000010) GPIOB->ODR |= (1U << 8); // Segment b (mapped to PB8)

if (segments & 0b00000100) GPIOB->ODR |= (1U << 9); // Segment c (mapped to PB9)

if (segments & 0b00001000) GPIOB->ODR |= (1U << 2); // Segment d (mapped to PB2)

if (segments & 0b00010000) GPIOB->ODR |= (1U << 1); // Segment e (mapped to PB1)

if (segments & 0b00100000) GPIOB->ODR |= (1U << 0); // Segment f (mapped to PB0)

if (segments & 0b01000000) GPIOA->ODR |= (1U << 7); // Segment g (mapped to PA7)

}

void display1(int digit) {

uint8\_t segments = segment\_map[digit];

// Clear all segment bits first (for segments connected to GPIOA and GPIOB)

GPIOA->ODR &= ~(1U << 11 | 1U << 12 | 1U << 15);

GPIOB->ODR &= ~(1U << 3 | 1U << 4 | 1U << 5 | 1U << 6);

// Set each segment based on the bit values in segments

if (segments & 0b00000001) GPIOA->ODR |= (1U << 11); // Segment a

if (segments & 0b00000010) GPIOA->ODR |= (1U << 12); // Segment b

if (segments & 0b00000100) GPIOA->ODR |= (1U << 15); // Segment c

if (segments & 0b00001000) GPIOB->ODR |= (1U << 3); // Segment d

if (segments & 0b00010000) GPIOB->ODR |= (1U << 4); // Segment e

if (segments & 0b00100000) GPIOB->ODR |= (1U << 5); // Segment f

if (segments & 0b01000000) GPIOB->ODR |= (1U << 6); // Segment g

}

void display(int digit) {

uint8\_t segments = segment\_map[digit];

GPIOC->ODR = (GPIOC->ODR & ~((1 << 14) | (1 << 15))) // Clear a, b

| ((segments & 0b00000001) << 14) // Segment a

| ((segments & 0b00000010) << 14);

GPIOB->ODR = (GPIOB->ODR & ~((1 << 12) | (1 << 13) | (1 << 14) | (1 << 15)))

| ((segments & 0b00000100) << 10) // Segment c

| ((segments & 0b00001000) << 10) // Segment d

| ((segments & 0b00010000) << 10) // Segment e

| ((segments & 0b00100000) << 10); // Segment f

GPIOA->ODR = (GPIOA->ODR & ~(1 << 8))

| ((segments & 0b01000000) << 2); // Segment g

}

void SysTick\_Handler(void) {

if (blink\_C14) {

GPIOC->ODR ^= (1U << 13); // Toggle LED on C14

}

if (blink\_C15) {

if (!pause) { // Only update if not paused

if (toggle\_led) {

counter--;

display((counter%60)/10 % 10);

display1((counter%60)% 10);

display2(counter/60 % 10);

} else {

static\_counter--;

display((static\_counter%60)/10 % 10);

display1((static\_counter%60) % 10);

display2(static\_counter/60 % 10); // Display static counter when toggling is off

}

}

}

}

void delay(volatile int count) {

while (count--) {

\_\_NOP(); // No Operation (NOP) for delay

}

}

// EXTI9\_5 interrupt handler (PA6 is on EXTI line 6)

void EXTI9\_5\_IRQHandler(void) {

if (EXTI->PR & (1 << 6)) {

EXTI->PR |= (1 << 6); // Clear interrupt flag for PA6

delay(30000); // Basic debounce

toggle\_led = 1;

}

}

// EXTI15\_10 interrupt handler (PB10 is on EXTI line 10, PC13 on line 13)

void EXTI15\_10\_IRQHandler(void) {

if (EXTI->PR & (1 << 10)) {

EXTI->PR |= (1 << 10); // Clear interrupt flag for PB10

delay(30000); // Basic debounce

toggle\_led = 0; // Disable toggling

}

if (EXTI->PR & (1 << 13)) {

EXTI->PR |= (1 << 13); // Clear interrupt flag for PC13

delay(30000); // Basic debounce

pause ^= 1; // Toggle pause flag (1 to pause, 0 to resume)

}

}

int main(void) {

// Enable GPIOC clock for LED control

RCC->AHB1ENR |= (1U << 0); // Enable clock for GPIOA

RCC->AHB1ENR |= (1U << 1); // Enable clock for GPIOB

RCC->AHB1ENR |= (1U << 2); // Enable clock for GPIOC

GPIOC->MODER &= ~(0x3 << 28); // Clear mode bits for PC14

GPIOC->MODER |= (0x1 << 28); // Set PC14 to output mode (01)

// Segment 'b' on PC15

GPIOC->MODER &= ~(0x3 << 30); // Clear mode bits for PC15

GPIOC->MODER |= (0x1 << 30); // Set PC15 to output mode (01)

// Segment 'c' on PB12

GPIOB->MODER &= ~(0x3 << 24); // Clear mode bits for PB12

GPIOB->MODER |= (0x1 << 24); // Set PB12 to output mode (01)

// Segment 'd' on PB13

GPIOB->MODER &= ~(0x3 << 26); // Clear mode bits for PB13

GPIOB->MODER |= (0x1 << 26); // Set PB13 to output mode (01)

// Segment 'e' on PB14

GPIOB->MODER &= ~(0x3 << 28); // Clear mode bits for PB14

GPIOB->MODER |= (0x1 << 28); // Set PB14 to output mode (01)

// Segment 'f' on PB15

GPIOB->MODER &= ~(0x3 << 30); // Clear mode bits for PB15

GPIOB->MODER |= (0x1 << 30); // Set PB15 to output mode (01)

// Segment 'g' on PA8

GPIOA->MODER &= ~(0x3 << 16); // Clear mode bits for PA8

GPIOA->MODER |= (0x1 << 16); // Set PA8 to output mode (01)

GPIOA->MODER &= ~((0x3 << 22) | (0x3 << 24) | (0x3 << 30)); // Clear mode bits

GPIOA->MODER |= ((0x1 << 22) | (0x1 << 24) | (0x1 << 30)); // Set to output mode (01)

// Set GPIOB pins B3, B4, B5, and B6 as output

GPIOB->MODER &= ~((0x3 << 6) | (0x3 << 8) | (0x3 << 10) | (0x3 << 12)); // Clear mode bits

GPIOB->MODER |= ((0x1 << 6) | (0x1 << 8) | (0x1 << 10) | (0x1 << 12)); // Set to output mode (01)

GPIOB->MODER &= ~((0x3 << 0) | (0x3 << 2) | (0x3 << 4) | (0x3 << 14) | (0x3 << 16) | (0x3 << 18));

GPIOA->MODER &= ~(0x3 << 14); // Clear mode bits for PA7

// Set mode bits for each segment pin to output mode (01)

GPIOB->MODER |= ((0x1 << 0) | (0x1 << 2) | (0x1 << 4) | (0x1 << 14) | (0x1 << 16) | (0x1 << 18));

GPIOA->MODER |= (0x1 << 14); // Set PA7 to output mode

// Enable USART1 clock

RCC->APB2ENR |= (1U << 4); // Enable clock for USART1

// Configure C14 and C15 as outputs (LEDs)

GPIOC->MODER &= ~(3U << (13 \* 2) | 3U << (15 \* 2)); // Clear mode bits

GPIOC->MODER |= (1U << (13 \* 2)) | (1U << (15 \* 2)); // Set C14 and C15 to output

GPIOA->MODER &= ~(0x3 << 12); // Set PA6 to input mode

GPIOB->MODER &= ~(0x3 << 20); // Set PB10 to input mode

GPIOC->MODER &= ~(0x3 << 26); // Set PC13 to input mode

// Enable internal pull-up resistors for PA6, PB10, and PC13

GPIOA->PUPDR &= ~(0x3 << 12); // Clear pull-up/pull-down bits for PA6

GPIOA->PUPDR |= (0x1 << 12); // Set pull-up for PA6

GPIOB->PUPDR &= ~(0x3 << 20); // Clear pull-up/pull-down bits for PB10

GPIOB->PUPDR |= (0x1 << 20); // Set pull-up for PB10

GPIOC->PUPDR &= ~(0x3 << 26); // Clear pull-up/pull-down bits for PC13

GPIOC->PUPDR |= (0x1 << 26); // Set pull-up for PC13

// 4- Enable SYSCFG clock

RCC->APB2ENR |= (1U << 14); // Enable SYSCFG clock

// 5- Configure EXTI for PA6, PB10, and PC13

SYSCFG->EXTICR[1] &= ~(0xF << 8); // Clear EXTI6 configuration (PA6)

SYSCFG->EXTICR[1] |= (0x0 << 8); // Set PA6 as the source input for EXTI6

SYSCFG->EXTICR[2] &= ~(0xF << 8); // Clear EXTI10 configuration (PB10)

SYSCFG->EXTICR[2] |= (0x1 << 8); // Set PB10 as the source input for EXTI10

SYSCFG->EXTICR[3] &= ~(0xF << 4); // Clear EXTI13 configuration (PC13)

SYSCFG->EXTICR[3] |= (0x2 << 4); // Set PC13 as the source input for EXTI13

// 6- Enable the Interrupt Mask register

EXTI->IMR |= (1 << 6); // Unmask EXTI line 6 (PA6)

EXTI->IMR |= (1 << 10); // Unmask EXTI line 10 (PB10)

EXTI->IMR |= (1 << 13); // Unmask EXTI line 13 (PC13)

// 7- Select the Interrupt Trigger

EXTI->FTSR |= (1 << 6); // PA6 as Falling trigger

EXTI->FTSR |= (1 << 10); // PB10 as Falling trigger

EXTI->FTSR |= (1 << 13); // PC13 as Falling trigger

// 8- NVIC Enable

NVIC\_EnableIRQ(EXTI9\_5\_IRQn); // Enable EXTI9\_5 interrupt

NVIC\_EnableIRQ(EXTI15\_10\_IRQn); // Enable EXTI15\_10 interrupt

USART1\_Init(); // Initialize UART for Bluetooth communication

// Configure SysTick to interrupt every 500 ms (assuming 16 MHz clock)

SysTick->LOAD = 16000000 - 1;

SysTick->VAL = 0;

SysTick->CTRL = 7; // Enable SysTick with interrupt

while (1) {

char command = USART1\_Read(); // Read a command from Bluetooth

if (command == '1') {

blink\_C14 = !blink\_C14; // Toggle blink state for C14

} else if (command == '2' || command=='3') {

if(command=='3'){

counter = 300;

static\_counter = 300 ;

}

blink\_C15 = !blink\_C15; // Toggle blink state for C15

GPIOC->ODR &= ~(1U << 15); // Ensure LED starts in off state

}

}

}

void USART1\_Init(void) {

// Enable GPIOA clock for USART1 pins

RCC->AHB1ENR |= (1U << 0); // Enable GPIOA clock

// Configure PA9 (TX) and PA10 (RX) for USART1

GPIOA->MODER &= ~(3U << (9 \* 2) | 3U << (10 \* 2));

GPIOA->MODER |= (2U << (9 \* 2)) | (2U << (10 \* 2)); // Set PA9 and PA10 to alternate function

// Set pull-up for UART pins

GPIOA->PUPDR &= ~(3U << (9 \* 2) | 3U << (10 \* 2)); // Clear bits

GPIOA->PUPDR |= (1U << (9 \* 2)) | (1U << (10 \* 2)); // Set pull-up

// Correct AFR configuration

GPIOA->AFR[1] &= ~(0xFFU << ((9-8) \* 4) | 0xFFU << ((10-8) \* 4)); // Clear bits

GPIOA->AFR[1] |= (7U << ((9-8) \* 4)) | (7U << ((10-8) \* 4)); // Set AF7 for PA9 and PA10

// Configure USART1

USART1->BRR = 0x683; // 9600 baud @ 16MHz

USART1->CR1 = 0; // Clear all settings first

USART1->CR1 |= (1U << 13); // Enable USART

USART1->CR1 |= (1U << 3); // Enable transmitter

USART1->CR1 |= (1U << 2); // Enable receiver

}

void USART1\_Write(char ch) {

while (!(USART1->SR & (1U << 7))); // Wait for TXE flag

USART1->DR = ch;

while (!(USART1->SR & (1U << 6))); // Wait for TC flag

}

char USART1\_Read(void) {

while (!(USART1->SR & (1U << 5))); // Wait until RX buffer has data

return USART1->DR;

}

# Detailed Explanation of STM32 Code for GPIO, USART, and SysTick Functionality:

This document provides a comprehensive breakdown of the STM32 code provided, which demonstrates GPIO control, USART communication, and SysTick usage. The code is written in C for an STM32 microcontroller and is structured into several functions that facilitate GPIO pin manipulation, serial communication, and timed operations.

## Includes and Variable Declarations:

The program begins with including the STM32 standard peripheral library header file 'stm32f4xx.h'. This file defines registers, peripheral addresses, and other key components needed to control the STM32. The code then declares various volatile variables used to manage the blinking state of LEDs on GPIO pins C14 and C15, and flags to track the counter and toggle states. Variables like 'pause' help control the timing logic for blinking, while arrays like 'segment\_map' provide bitwise representations of numbers for seven-segment display purposes.

## USART1 Initialization Function:

The 'USART1\_Init' function configures the USART1 peripheral for serial communication. It first enables the GPIOA clock to allow access to the GPIOA pins, then sets PA9 and PA10 as alternate function pins (AF7) required for USART1 communication. It also configures the baud rate for 9600 bps and enables the USART, transmitter, and receiver, allowing it to handle serial communication.

## GPIO Pin Configuration:

The code configures GPIO pins for output or input based on their intended usage in controlling LEDs or segments. This involves setting GPIO modes for pins in ports A, B, and C. The configuration is detailed for each pin, and certain pins are explicitly set for the purpose of controlling the seven-segment display.

## Display Functions for Seven-Segment Control:

The functions 'display', 'display1', and 'display2' manage the seven-segment display by taking an integer digit (0-9) and mapping it to specific GPIO pins to represent segments (a to g). The 'segment\_map' array holds binary representations of each number, and bitwise operations in each function set or clear GPIO pins to display the digit accordingly.

## SysTick Interrupt Handler:

The 'SysTick\_Handler' function serves as an interrupt handler, triggered at regular intervals to update the state of LEDs. The 'blink\_C14' and 'blink\_C15' flags toggle LEDs, while the counter variables are updated when toggling is enabled. By using the counter, the function achieves precise timing for LED blinks and controls the display values on the seven-segment display based on the countdown timer.

## EXTI9\_5\_IRQHandler and EXTI15\_10\_IRQHandler Functions:

These two functions handle interrupts triggered by external buttons connected to GPIO pins PA6, PB10, and PC13. When an interrupt occurs (button press), the program clears the interrupt flag, applies a simple debounce by adding a delay, and toggles the 'toggle\_led' or 'pause' flags. These flags control whether the counter updates and whether it pauses, enabling start-stop or reset-like functionality.

## Main Function Execution:

The ‘main’ function initializes GPIO, USART, and SysTick functionalities and enters an infinite loop where it reads commands via USART. Depending on the received command (‘1’, ‘2’, or ‘3’), it toggles the blink state of LEDs connected to GPIO C14 and C15. Command ‘3’ resets the counter values to 300, setting a custom countdown value.

## Additional Explanations of Core Concepts:

1. The volatile Keyword: Variables used within interrupt handlers or hardware registers are marked as volatile, informing the compiler not to optimize their access since their values can change unexpectedly.

2. Bitwise Operations: Bitwise AND, OR, and shift operations are essential for controlling individual bits in registers, which is how GPIO pins are manipulated without affecting others.

3. Timing Control: SysTick provides an efficient way to implement delays and periodic tasks, ensuring accurate timing without relying on delay loops, which can be inaccurate at higher clock speeds.

## Conclusion:

This STM32 code provides a comprehensive approach to handling basic GPIO operations, timed tasks, and serial communication, allowing a microcontroller to control LEDs and seven-segment displays based on user inputs and system timing. By understanding the code structure and modular approach, one can modify or extend functionality for similar applications.