JAYARAJ ANNAPACKIAM C S I COLLEGE OF ENGINEERING

(Approved by AICTE, New Delhi and Affiliated to Anna University, Chennai)

MARGOSCHIS NAGAR, NAZARETH – 628 617

DEPARTMENT OF ELECTRONICS AND COMMUNICATION ENGINEERING

2024-2025 (EVEN SEMESTER)

REGULATIONS 2021

COURSE IN-CHARGE	**: R. JEYA PREEDA**
DESIGNATION	**: AP / ECE**
COURSE CODE	**: EC369**
COURSE NAME	**: IOT PROCESSORS**
CLASS	**: III ECE B**
SEMESTER	**: VI**

PREPARED BY	**VERIFIED BY**	**APPROVED BY**
R. JEYA PREEDA	**H.O.D**	**PRINCIPAL**

VISION

Impart the technology of electronics and communication along with ethical values transforming the learners into global achievers

MISSION

- Offer high quality education with hope, confidence and moral values.
- Equip the students with hands-on training on new tools.
- Motivate students for self-learning.
- Outfit the students to be compatible with recent trends in electronic industries.
- Organize students for professional careers and advanced studies.

PROGRAM EDUCATIONAL OBJECTIVES (PEOs)

PEO1: To provide the students with a strong foundation in the required sciences in order to pursue studies in Electronics and Communication Engineering.

PEO2: To gain adequate knowledge to become good professional in electronic and communication engineering associated industries, higher education and research.

PEO3: To develop attitude in lifelong learning, applying and adapting new ideas and technologies as their field evolves.

PEO4: To prepare students to critically analyze existing literature in an area of specialization and ethically develop innovative and research-oriented methodologies to solve the problems identified.

PEO5: To inculcate in the students a professional and ethical attitude and an ability to visualize the engineering issues in a broader social context.

PROGRAM SPECIFIC OUTCOMES (PSOs)

PSO1: Design, develop and analyze electronic systems through application of relevant electronics, mathematics and engineering principles

PSO2: Design, develop and analyze communication systems through application of fundamentals from communication principles, signal processing, and RF System Design & Electromagnetics.

PSO3: Adapt to emerging electronics and communication technologies and develop innovative solutions for existing and newer problem.

PROGRAM OUTCOMES (POs)

PO1 Engineering knowledge: Apply the knowledge of mathematics, science, engineering fundamentals, and an engineering specialization to the solution of complex engineering problems.

PO2 Problem analysis: Identify, formulate, review research literature, and analyze complex engineering problems reaching substantiated conclusions using first principles of mathematics, natural sciences, and engineering sciences.

PO3 Design/development of solutions: Design solutions for complex engineering problems anddesign system components or processes that meet the specified needs with appropriate consideration for the public health and safety, and the cultural, societal, and environmental considerations.

PO4 Conduct investigations of complex problems: Use research-based knowledge and research methods including design of experiments, analysis and interpretation of data, and synthesis of the information to provide valid conclusions.

PO5 Modern tool usage: Create, select, and apply appropriate techniques, resources, and modern engineering and IT tools including prediction and modeling to complex engineering activities with an understanding of the limitations.

PO6 The engineer and society: Apply reasoning informed by the contextual knowledge to assess societal, health, safety, legal and cultural issues and the consequent responsibilities relevant to the professional engineering practice.

PO7 Environment and sustainability: Understand the impact of the professional engineering solutions in societal and environmental contexts, and demonstrate the knowledge of, and need forsustainable development.

PO8 Ethics: Apply ethical principles and commit to professional ethics and responsibilities and norms of the engineering practice.

PO9 Individual and team work: Function effectively as an individual, and as a member or leader in diverse teams, and in multidisciplinary settings.

PO10 Communication: Communicate effectively on complex engineering activities with the engineering community and with society at large, such as, being able to comprehend and write effective reports and design documentation, make effective presentations, and give and receive clear instructions.

PO11 Project management and finance: Demonstrate knowledge and understanding of the engineering and management principles and apply these to one's own work, as a member and leader in a team, to manage projects and in multidisciplinary environments.

PO12 Life-long learning: Recognize the need for, and have the preparation and ability to engage in independent and life-long learning in the broadest context of technological change.

CEC369 IOT PROCESSORS

L	T	P	C
2	0	2	3

COURSE OBJECTIVES:

- Learn the architecture and features of ARM.
- Study the exception handling and interrupts in CORTEX M3
- Program the CORTEX M3
- Learn the architecture of STM 32L15XXX ARM CORTEX M3/M4 microcontroller.
- Understand the concepts of System – On – Chip (SoC)

PRACTICAL EXERCISES:

ARM Assembly Programming

1. Write a program to add two 32-bit numbers stored in r0 and r1 registers and write the result to r2. The result is stored to a memory location. a) Run the program with breakpoint and verify the result b) Run the program with stepping and verify the content of registers at each stage.
2. Write ARM assembly to perform the function of division. Registers r1 and r2 contain the dividend and divisor, r3 contains the quotient, and r5 contains the remainder.

Embedded C Programming on ARM Cortex M3/M4 Microcontroller

1. Write a program to turn on green LED (Port B.6) and Blue LED (Port B.7) on STM32L- Discovery by configuring GPIO.
2. Transmit a string "Programming with ARM Cortex" to PC by configuring the registers of USART2. Use polling method.

ARM Cortex M3/M4 Programming with CMSIS

1. Write a program to toggle the LEDs at the rate of 1 sec using standard peripheral library. Use Timer3 for Delay.
2. Transmit a string "Programming with ARM Cortex" to PC by using standard peripheral library with the help of USART3. Use polling method.

30 PERIODS

COURSE OUTCOMES:

On successful completion of this course, the student will be able to

CO1: Explain the architecture and features of ARM.

CO2: List the concepts of exception handling.

CO3: Write a program using ARM CORTEX M3/M4.

CO4: Learn the architecture of STM32L15XXX ARM CORTEX M3/M4.

CO5: Design an SoC for any application.

EXP. NO :01 **DATE :**	**ARM ASSEMBLY PROGRAM TO ADD TWO NUMBERS**

AIM:

To write a program to add two 32-bit numbers stored in R0 and R1 registers and write the result to R2.

APPARATUS REQUIRED:

1. Keil uvision5 with CortexM0 package

PROCEDURE:

Run the Program with Breakpoint and Verify the Result

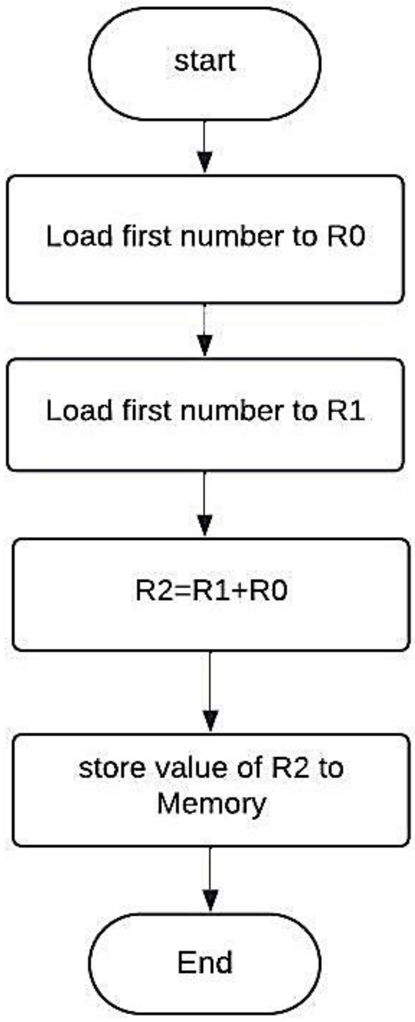
1. **Set a breakpoint** at the line where the ADD instruction is located. This is where r2 will get the result of r0 + r1.
2. **Run the program** in your debugger.
3. When the program hits the **breakpoint**, verify the values of r0, r1, and r2:
    - r0 should contain the first operand.
    - r1 should contain the second operand.
    - r2 should have the result of the addition.
4. **Continue execution** until the program finishes.
5. Verify that the memory location RESULT contains the sum of the two numbers.

PROGRAM:

```
 .global main

main:
  MOVS r0, #0x05     @ Load data into r0
  MOVS r1, #0x03     @ Load data into r1
  ADDS r2, r1, r0    @ r2 = r1 + r0 (Addition)
  B .
```

FLOW DIAGRAM:



TABULATION:

INPUT		OUTPUT	
ADDRESS	**DATA**	**ADDRESS**	**DATA**

RESULT:

Thus, the ARM assembly to perform the function of addition was written and executed successfully.

EXP. NO : 02 DATE :	ARM ASSEMBLY TO DIVISION TWO NUMBERS

AIM:

To write ARM assembly to perform the function of division. Registers r1 and r2 contain the dividend and divisor, r3 contains the quotient, and r5 contains the remainder.

APPARATUS REQUIRED:

2. Keil uvision5 with CortexM4 package

PROCEDURE:

ARM Assembly with UDIV Instruction

UDIV r3, r1, r2:

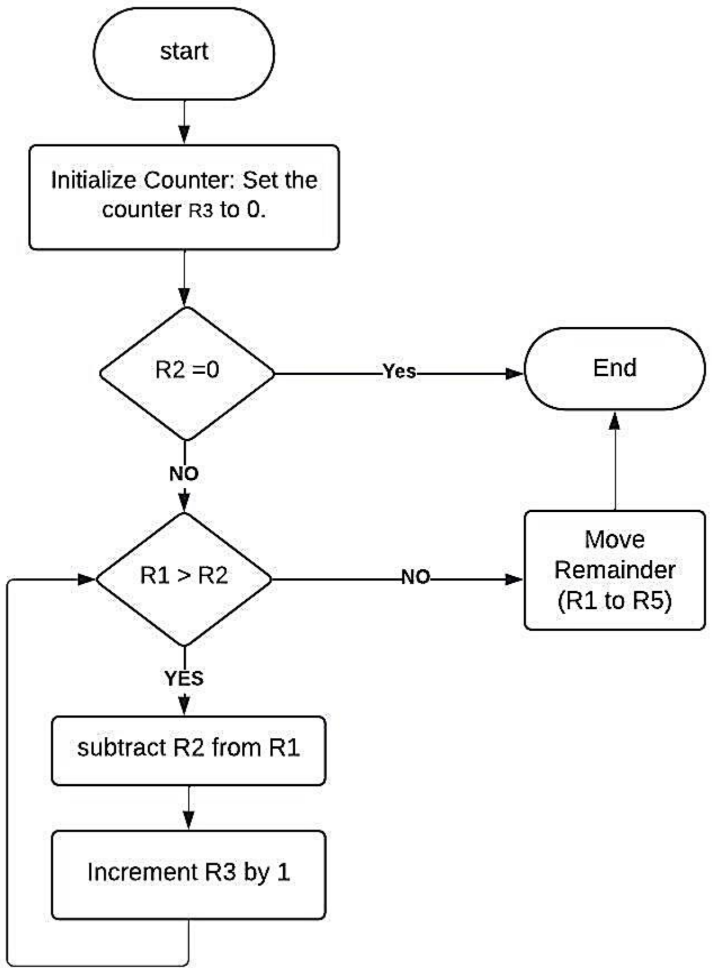
- UDIV performs an unsigned integer division of r1 by r2 and stores the quotient in r3.
- Example: If r1 = 09 and r2 =04, then r3 will contain 2.
- UMULL r7, r6, r2, r3 and SUB r5, r1, r7 (Multiply and Subtract) calculates the remainder.
- It multiplies r3 (the quotient) by r2 (the divisor) and subtracts the result from r1 (the dividend).
- This stores the remainder in r5.
- Example: r5 = 9 - (4 * 2) = 1.

PROGRAM:

```
        .global main

main:
        MOV R1, #0x09
        MOV R2, #0x04
        UDIV R3,R1,R2
        UMULL R7,R6,R2,R3
        SUB R5,R1,R7
        B .
```

FLOW CHART:


start
Initialize Counter: Set the counter R3 to 0.
R2 =0
Yes
End
NO
R1 > R2
NO
Move Remainder (R1 to R5)
YES
subtract R2 from R1
Increment R3 by 1

Tabulation:

INPUT		OUTPUT	
ADDRESS	**DATA**	**ADDRESS**	**DATA**

RESULT:

Thus, the ARM assembly to perform the function of division was written and executed successfully.

EXP. NO: 03 DATE :	PROGRAM TO TURN ON LED USING ON STM32

AIM:

To write a program for turn on green LED (Port B.6) and Blue LED (Port B.7) on STM32L Discovery by configuring GPIO.

APPARATUS REQUIRED:

S.NO	APPRATAUS NAME	RANGE	QUANTITY
1	STM32	-	01

PROCEDURE:

Set Up the Development Environment

1. Install a development environment such as STM32CubeIDE or Keil uVision.
2. Ensure you have the appropriate STM32 HAL/LL or CMSIS libraries installed.

Include the Necessary Header File

- Include the header file corresponding to your STM32 series, e.g., stm32f4xx.h.

Enable GPIO Clock for Port D

- GPIO peripherals are not enabled by default. Use the RCC register to enable the clock for Port D:

Configure GPIO Pins for LEDs

- Each GPIO pin has a mode register (MODER) for selecting the function:

1. Clear the 2 bits corresponding to the desired pin.
2. Set the bits to configure the pin as an output.

Turn On the LEDs

- Use the BSRR (Bit Set Reset Register) to set the pins high

Write the Infinite Loop

- Ensure the program keeps running to maintain the LEDs in the ON state

Compile and Load the Code

1. Compile the program in your IDE or Keil uVision.
2. Connect the STM32L/STM32F to your PC using a USB cable.
3. Flash the compiled program onto the board using a debugger like ST-Link

Program:

Turn On Green and Blue LEDs on STM32L or STM32F Discovery

```
#include "stm32f4xx.h"
int main(void)
{

     // Configure LEDs

                  RCC->AHB1ENR |= RCC_AHB1ENR_GPIODEN;        // Enable the clock of port D of the GPIO

     GPIOD->MODER |= GPIO_MODER_MODER12_0;                    // Green LED, set pin 12 as output
     GPIOD->MODER |= GPIO_MODER_MODER15_0;                    // Blue LED, set pin 15 as output

     GPIOD->BSRR = 1<<12;                      // Set the BSRR bit 12 to 1 to turn respective LED on green
     GPIOD->BSRR = 1<<15;                       // Set the BSRR bit 15 to 1 to turn respective LED on blue
  while (1)
  {}
}
```

OUTPUT:

The Green LED and blue LED are turned on.

RESULT:

Thus, the program for turn on green LED (Port B.6) and Blue LED (Port B.7) on STM32L Discovery by configuring GPIO was written and executed successfully.

EXP. NO: 04 DATE :	PROGRAM TO TRANSMIT A STRING ON STM32 USING USART2

AIM:

To write a program for Transmit a string “Programming with ARM Cortex” to PC by configuring the registers of USART2.Use polling method.

APPARATUS REQUIRED

S.NO	APPRATAUS NAME	RANGE	QUANTITY
1	STM32	-	01

PROCEDURE:

Enable Necessary Clocks

- Enable the clocks for **GPIOA**, **USART2**, and **GPIOD** by configuring the RCC->AHB1ENR and RCC-
>APB1ENR registers.

Configure PA2 as USART2TX

- Set **PA2** to alternate function mode in GPIOA->MODER and assign **AF7** (USART2) in GPIOA->AFR*0+.

Initialize USART2

- Set the baud rate to **9600 bps** using USART2->BRR.
- Enable the transmitter (USART2->CR1 |= (1 << 3)) and turn on USART2 (USART2->CR1 |= (1 << 13)).

Set Up LEDs

- Configure **PD12** (green LED) and **PD13** (orange LED) as output by setting their mode in GPIOD->MODER.

Write Function for USART2 Transmission

- Implement a function (usart2_write) to send a single character. Wait until the transmit buffer is empty (TXE set), then write the character to USART2->DR.

Write Function to Transmit Strings

- Create a function (usart2_write_string) to iterate over a string and transmit each character using usart2_write. Call the LED blink function after transmitting each character.

Blink PD13 During Transmission

- Write a function (led_blink) to toggle **PD13** on and off with a short delay for visible blinking.

Turn on PD12 After Transmission

- Implement a function (led_on) to set the output data register for **PD12**, turning it on to indicate the transmission is complete.

Write the Main Function

- Initialize USART2 and LEDs using usart2_init() and led_init().
- Transmit the string "Programming with ARM Cortex" using usart2_write_string.
- Call led_on() to turn on PD12 after the transmission.

Compile, Flash, and Test

- Compile the program, flash it onto the STM32F407 board, and connect the board to a terminal application via a USB-to-serial adapter.
- Verify:
    - **PD13 blinks** during string transmission.
    - **PD12 remains lit** after the transmission complete

Program:

Transmit a string using on Stm32

```
#include "stm32f4xx.h"     // Include the header file for STM32F407
void usart2_init(void);
void usart2_write(char ch);
void usart2_write_string(const char *str);
void led_init(void);
void led_on(void); void led_blink(void);

int main(void) {
// Initialize USART2 and LED
usart2_init();
led_init();
      for (volatile int i = 0;i < 100000; i++);          // Simple delay loop

// Transmit the string with LED blinking during transmission
usart2_write_string("Programming with ARM Cortex");
      for (volatile int i = 0;i < 100000; i++);          // Simple delay loop

// After transmission, turn on the PD12 LED
led_on();
while (1) {}
// Infinite loop to keep the PD12 LED on
}
void usart2_init(void) {
// Enable clock for GPIOA, GPIOD, and USART2
RCC->AHB1ENR |= (1 << 0); // Enable GPIOA clock (for USART2 TX)
RCC->AHB1ENR |= (1 << 3); // Enable GPIOD clock (for LEDs)
RCC->APB1ENR |= (1 << 17); // Enable USART2 clock

// Configure PA2 (USART2 TX) as Alternate Function
GPIOA->MODER &= ~(0x3 << 4); // Clear mode bits for PA2
GPIOA->MODER |= (0x2 << 4); // Set PA2 to Alternate Function mode
GPIOA->AFR[0] |= (0x7 << 8); // Set AF7 (USART2) for PA2

// Configure USART2
USART2->BRR = 0x0683;                    // Set baud rate to 9600 (assuming 16 MHz clock)
USART2->CR1 |= (1 << 3);                 // Enable transmitter
USART2->CR1 |= (1 << 13);                // Enable USART2
}

void usart2_write(char ch) {
while (!(USART2->SR & 0x80));            // Wait until TXE (Transmit data register empty) is set
USART2->DR = ch;                         // Write character to data register
led_blink();
}

void usart2_write_string(const char *str){
      while (*str) {
              usart2_write(*str++);      // Transmit each character
```

```
        }
        // Blink LED during transmission
}
// LED Initialization Function
void led_init(void) {
// Configure PD12 and PD13 as Output for LEDs
GPIOD -> MODER &=~ (3 << 24);               // Clear mode bits for PD12
GPIOD -> MODER |= (1<<24);                  // Set PD12 to Output mode
GPIOD -> MODER &= ~(3 << 26);               // Clear mode bits for PD13
GPIOD -> MODER |= (1 << 26);                // Set PD13 to Output mode
}

// LED ON Function for PD12
void led_on(void) {
GPIOD->ODR |= (1 << 12);                     // Set PD12 high to turn on the LED
}

// LED Blink Function for PD13
void led_blink(void) {
GPIOD->ODR ^= (1 << 13);                    // Toggle PD13 to blink the LED
for (volatile int i = 0;i < 100000; i++);   // Simple delay loop
GPIOD->ODR ^= (1 << 13); // Toggle PD13 back
for (volatile int i = 0; i < 100000; i++);   // Simple delay loop
        GPIOD->ODR ^= (1 << 13);            // Toggle PD13 to blink the LED
for (volatile int i = 0;i < 100000; i++);   // Simple delay loop
GPIOD->ODR ^= (1 << 13); // Toggle PD13 back
for (volatile int i = 0; i < 100000; i++);  // Simple delay loop
}
```

<u>**OUTPUT:**</u>

The string “Programming with ARM Cortex” was transmitted.

<u>**RESULT:**</u>

Thus, the program for Transmit a string “Programming with ARM Cortex” to PC by configuring the registers of USART2 using polling method was written and executed successfully.

EXP. NO : 05 DATE :	PROGRAM TO TOGGLE THE LED USING ON STM32

AIM:

To write a program for toggle the LEDs at the rate of 1 sec using standard peripheral library. Use Timer3 for Delay.

APPARATUS REQUIRED:

S.NO	APPRATAUS NAME	RANGE	QUANTITY
1	STM32	-	01

PROCEDURE:

1. Enable the Clock for Timer 2 and GPIOD
   Enable Timer 2 by setting the 0th bit in RCC->APB1ENR
2. Configure Timer 2 for a 1 ms Delay
   Set the prescaler to divide the clock down to 1 kHz (1 ms per tick)
3. Configure GPIO Pins for LEDs
   Set PD12, PD13, PD14, and PD15 as output by configuring the GPIOD->MODER register:
4. Create the Timer-Based Delay Function
   Implement a delay function using Timer 2:
5. Toggle LEDs in the Main Loop
   Use the GPIOD->ODR register to toggle the state of the LEDs
6. Insert a Delay in the Loop
   Call the delay function after toggling the LEDs to control the blinking speed
7. Run the Infinite Loop
   Continuously toggle LEDs with the delay function in the while loop

➢ **Includes and GPIO Configuration:**

- Include stm32l1xx.h for STM32L series (adjust the header file if using a different series).
- GPIO_Config() sets PB6 and PB7 as output pins for the green and blue LEDs.

➢ **Timer3 Configuration:**

- Timer3_Config() enables the clock for Timer3 using RCC->APB1ENR.
- Configures the prescaler (TIM3->PSC) to divide the 16 MHz clock by 16,000, resulting in a 1 kHz clock (1 ms tick).

- Sets the auto-reload register (TIM3->ARR) to 1000, giving a 1-second delay.
- Enables the Timer3 update interrupt with TIM3->DIER |= TIM_DIER_UIE.
- Enables Timer3 and configures it to start counting.

 ➢ **Interrupt Service Routine (ISR):**

- TIM3_IRQHandler() is triggered whenever Timer3 reaches the ARR value (1 second).
- Checks the update interrupt flag (TIM3->SR & TIM_SR_UIF) to ensure the interrupt is triggered by the timer update.
- Toggles the LED states using GPIOB->ODR ^= (1 << 6) for the green LED and GPIOB->ODR ^= (1 << 7) for the blue LED.
- Clears the interrupt flag after toggling the LEDs.

 ➢ **Main Loop:**

- The while(1) loop is empty because the LED toggling is managed by the Timer3 interrupt.

PROGRAM:
Toggle the LED using on Stm32

```
#include "stm32f4xx.h"
int main(void)
{
    // Configue LEDs
    RCC->AHB1ENR |= RCC_AHB1ENR_GPIODEN;        // Enable the clock of port D of the GPIO

    GPIOD->MODER |= GPIO_MODER_MODER12_0;      // Green LED, set pin 12 as output
    GPIOD->MODER |= GPIO_MODER_MODER13_0;      // Orange LED, set pin 13 as output
    GPIOD->MODER |= GPIO_MODER_MODER14_0;      // Red LED, set pin 14 as output
    GPIOD->MODER |= GPIO_MODER_MODER15_0;      // Blue LED, set pin 15 as output

    uint32_t i=0;

  /* Infinite loop */
  /* USER CODE BEGIN WHILE */
  while (1)
  {

    /* USER CODE END WHILE */

              // Turn on LEDs
              GPIOD->BSRR = 1<<12;          // Set the BSRR bit 12 to 1 to turn respective LED on green
              GPIOD->BSRR = 1<<13;           // Set the BSRR bit 13 to 1 to turn respective LED on orange
              GPIOD->BSRR = 1<<14;           // Set the BSRR bit 14 to 1 to turn respective LED on red
              GPIOD->BSRR = 1<<15;           // Set the BSRR bit 15 to 1 to turn respective LED on blue

              // Delay
              i = 0;
              while(i < 2000000)
              { i++;} // Loop repeats 2,000,000 implementing a delay

              // Turn off LEDs
              GPIOD->BSRR = 1<<(12+16); // Set the BSRR bit 12 + 16 to 1 to turn respective LED off
              GPIOD->BSRR = 1<<(13+16);  // Set the BSRR bit 13 + 16 to 1 to turn respective LED off
              GPIOD->BSRR = 1<<(14+16); // Set the BSRR bit 14 + 16 to 1 to turn respective LED off
              GPIOD->BSRR = 1<<(15+16);  // Set the BSRR bit 15 + 16 to 1 to turn respective LED off

              // Delay
              for(i = 0; i < 2000000; i++){};   // Loop repeats 2,000,000 implementing a delay
              i=0;
              while(i < 2000000)
              { i++;} // Loop repeats 2,000,000 implementing a delay

  }

}
```

OUTPUT:

The LEDs are toggled (On and Off).

RESULT:

Thus, the program for toggle the LEDs at the rate of 1 sec using standard peripheral library using Timer3 for delay was written and executed successfully.

EXP. NO: 06 DATE :	**PROGRAM TO TRANSMIT A STRING ON STM32 USING USART3**

AIM:

To write a program for Transmit a string “Programming with ARM Cortex” to PC by configuring the registers of USART3.Use polling method.

APPARATUS REQUIRED

S.NO	APPRATAUS NAME	RANGE	QUANTITY
1	STM32	-	01

PROCEDURE:

1. Enable and Configure USART3
2. Implement USART3 Write Function
3. Implement USART3 String Transmission
4. Initialize LEDs on GPIOD 5. Create LED Control Functions
6. Implement a Simple Delay Function
7. initialize USART3 and LEDs, then send data and control LEDs

Program:

Transmit a string using on Stm32

```
#include <stm32f405xx.h>
#include <string.h>
void USART3_init(void);
void USART3_write(uint8_t ch);
void USART3_write_string(const char* str);
void delayMs(int);
void LED_init(void);
void LED_on_red(void);
void LED_off_red(void);
void LED_on_orange(void);
void LED_off_orange(void);
int main(void)
{
 USART3_init();                    /* Initialize USART3 */
 LED_init();                        /* Initialize LEDs */
 // Transmit the string over USART3
 LED_on_red();                     // Turn on red LED to indicate transmission
 USART3_write_string("Programming with ARM Cortex\n");          // Send string
 LED_off_red();                    // Turn off red LED after transmission
 LED_on_orange();                   // Turn on orange LED after transmission
 while (1)
 {
```

```
 // Main loop can do other tasks
 delayMs(1000);                          // Delay to avoid busy looping
 }
}
void USART3_init(void)
{
 RCC->AHB1ENR |= (1 << 2);                         // Enable GPIOC clock
 RCC->APB1ENR |= (1 << 18);                        // Enable USART3 clock
 // Configure PC10 as TX for USART3
 GPIOC->MODER &= ~(3 << 20);                        // Clear mode for PC10
 GPIOC->MODER |= (1 << 21);                        // Set PC10 to alternate function mode
 GPIOC->AFR[1]&= ~(0xF << 8);                      // Clear alternate function for PC10
 GPIOC->AFR[1]|= (7 << 8); // Set AF7 for PC10
 // Configure PC11 as RX for USART3
 GPIOC->MODER &= ~(3 << 22);                        // Clear mode for PC11
 GPIOC->MODER |= (1 << 23);                         // Set PC11 to alternate function mode
 GPIOC->AFR[1] &= ~(0xF << 12);                     // Clear alternate function for PC11
 GPIOC->AFR[1] |= (7 << 12);                        // Set AF7 for PC11
 // USART3 settings
 USART3->BRR = 0x0683;                              // 9600 baud rate at 16MHz
 USART3->CR1 |= (0xC << 0);                        // Enable RE and TE bits
 USART3->CR1 |= (1 << 6);                          // Enable RX
 USART3->CR2 = 0;                                  // 1 stop bit
 USART3->CR3 = 0;                                   // Default settings
 USART3->CR1 |= (1 << 13);                         // Enable USART3
}
void USART3_write(uint8_t ch)
{
 // Write the character to USART data register
 USART3->DR = ch;
 while (!(USART3->SR & (1 << 7)));                  // Wait until TXE is set
}
void USART3_write_string(const char* str)
{
 while (*str) // Transmit until null terminator
 {
 USART3_write(*str++);                              // Send each character
 }
}
void LED_init(void)
{
 RCC->AHB1ENR |= (1 << 3);                          // Enable GPIOD clock
 // Configure PD14 (Red LED) as output
 GPIOD->MODER &= ~(3 << 28);                       // Clear mode for PD14
 GPIOD->MODER |= (1 << 28);                        // Set PD14 to output mode
 // Configure PD13 (Orange LED) as output
 GPIOD->MODER &= ~(3 << 26);                        // Clear mode for PD13
 GPIOD->MODER |= (1 << 26);                    // Set PD13 to output mode
}
void LED_on_red(void)
{
 GPIOD->ODR |= (1 << 14);                      // Set PD14 high to turn on red LED
}
void LED_off_red(void)
    {
 GPIOD->ODR &= ~(1 << 14);                     // Set PD14 low to turn off red LED
    }
void LED_on_orange(void)
    {
 GPIOD->ODR |= (1 << 13);                      // Set PD13 high to turn on orange LED
    }
void LED_off_orange(void)
    {
 GPIOD->ODR &= ((1 << 13));                     // Set PD13 low to turn off orange LED
}
void delayMs(int n)
{
 int i;
 for (; n > 0; n--)
 for (i = 0; i < 2000; i++);                    // Simple delay loop
}
```

OUTPUT:

The string “Programming with ARM Cortex” was transmitted.

RESULT:

Thus, the program for Transmit a string “Programming with ARM Cortex” to PC by configuring the registers of USART3 using polling method was written and executed successfully.

EC369 IOT PROCESSORS
VIVA QUESTIONS

1. What is an IoT processor, and how does it differ from a general-purpose processor?
An **IoT processor**—typically a microcontroller (MCU) or embedded microprocessor—is optimized for **low power**, **small size**, and **real-time control**, often integrating CPU, RAM, flash, and I/O on a single chip. General-purpose processors (like desktop CPUs) focus on high performance and multitasking, lacking integrated peripherals and consuming much more power.

2. Role of microcontrollers in IoT devices
MCUs are ubiquitous in IoT: they run sensor/actuator logic, manage communications (UART, I²C, SPI), perform signal processing, and stay in energy-saving sleep modes to conserve battery.

3. Key features of ARM Cortex-M series for IoT
- **Low Power**: Sleep modes, clock gating, microamp-level idle power
- **Thumb-2 Code Density**: Combines 16-bit/32-bit instructions for compact code
- **DSP & FPU Options** (M4, M33): SIMD, MAC, optional floating-point
- **Interrupt & RTOS Support**: NVIC, dual stack pointers, systick timer
- **Security**: MPU, TrustZone-M (in M33), fault handlers

4. Microprocessor vs. Microcontroller in IoT
MCUs include CPU + RAM + flash + I/O on one chip—ideal for compact, low-power tasks. MPUs have only CPU, need external RAM/peripherals, and are suited for higher computation but consume more energy

5. Significance of low-power consumption
IoT devices often run on batteries or energy harvesting. Low power maximizes lifespan, reduces EMI (beneficial for wireless), and lowers overall costs.

6. Main components of an IoT processor architecture
- **CPU core** (32-bit like ARM Cortex-M)
- **Memory** (Flash, SRAM)
- **Peripherals** (timers, ADCs)
- **Communication interfaces** (UART, SPI, I²C)
- **Power/clock management**
- **Security modules** (MPU, TrustZone)
-

7. System-on-Chip (SoC) in IoT
An SoC combines CPU, memory, analog, wireless radios (Wi-Fi, BLE), and MPUs on a single chip—compact, power-efficient, ideal for connected devices.

8. Role of DSPs in IoT
DSP blocks in cores like Cortex-M4/M33 perform signal filtering, FFTs, sensor processing (audio, vibration) efficiently with SIMD/MAC acceleration

9. Harvard architecture benefits
Separate instruction/data buses enable parallel fetch/execute, higher throughput, better deterministic behavior, and improved real-time performance.

10. Importance of RTOS
RTOS support enables task scheduling, real-time interrupts, resource management—enhanced by features like NVIC, dual stack pointers, MPU, and systick timer

11. Common communication protocols
MCUs support **UART**, **SPI**, **I^{2}C**, **CAN**, and wireless: **Bluetooth LE**, **Zigbee**, **Wi-Fi**, **LoRa**, etc.

12. Handling multiple comm interfaces
Multiple hardware peripherals + DMA controllers facilitate concurrent communication, while software libraries/RTOS manage protocol stacks.

13. Role of UART, SPI, I^{2}C

- **UART**: Serial console/long-range comm
- **SPI**: High-speed data exchange with sensors
- **I^{2}C**: Low-speed bus for everyday ICs like EEPROM, sensors

14. Significance of edge computing
IoT processors perform local analytics (e.g., compression, anomaly detection), reducing latency and bandwidth needs before sending data to the cloud

15. Managing data transmission to cloud services
MCUs buffer/process; optional hardware crypto secures; run stacks (MQTT, HTTP) on RTOS; interfaces (Wi-Fi, LTE-M) handle transport to cloud.

16. Integrated security features

- **MPU**: Memory domain isolation
- **Secure boot**: Verified firmware at startup
- **Crypto accelerators** (AES)
- **TrustZone-M**: Hardware separation of secure/not-secure

17. Secure boot
Firmware signature check at boot ensures only authenticated code runs, protecting device integrity.

18. Hardware security modules
MPUs and TrustZone separate code/data domains, reducing vulnerability impact and enabling safe execution environments.

19. Common vulnerabilities and mitigation
Issues: buffer overflows, weak crypto, side-channel attacks. Mitigation: use MPU/TrustZone, secure coding, regular firmware updates

20. Firmware updates importance
Patch security vulnerabilities and bug fixes; secure delivery mechanisms (TLS, encrypted updates, rollback protection) are essential.

21. Balancing performance and power
Techniques used include:

- **DVFS**: Dynamic voltage/frequency scaling
- **Sleep modes**: Sleep, deep sleep using WFI/WFE instructions

22. DVFS explained
Adjust CPU voltage/frequency in real-time based on load, optimizing energy efficiency vs processing speed.

23. Sleep modes in IoT processors
Modes like Sleep, Deep Sleep, Shutoff allow disabling CPU or peripheral clocks to save energy, with fast wake-up (<2 µs)

24. Thermal management
IoT MCUs operate at low frequency (<200 MHz) with passive cooling. Power management features limit heat by reducing voltage and clock speeds.

25. Challenges optimizing processing speed
Limited memory, strict power budgets, real-time constraints, and the need to fit both firmware and communication stacks.

26. Give applications of IOT processors.

- **Wearables**: e.g. smartwatches using Cortex-M4/M33 (DSP, FPU, low power)
- **Smart home**: MCU + Wi-Fi/BLE SoC control sensors, lighting through local automation stacks.
- **Industrial automation**: Cortex-M33 used in motor control, timing precision, safety – TrustZone adds security.
- **Agricultural monitoring**: Sensor nodes use low-power MCUs (e.g., Cortex-M0+) for temperature, moisture collection before transmitting via LoRa.
- **Healthcare**: Medical IoT (glucose monitors, patient wearables) use MCUs with ultra-low power consumption and security like secure boot and MPU

27. What is an STM32 microcontroller?
STM32 is a family of 32-bit microcontrollers from STMicroelectronics, based on ARM Cortex-M cores. They're known for high performance, low power, and rich peripheral sets

28. Which core architectures are used in STM32?
They use ARM Cortex-M0/M0+, M3, M4(F), M7, M33, and M55 cores, depending on the series and application needs

29. Name the main STM32 series and their purposes.

- **F-series**: general purpose (F0, F1, F3, F4, F7, H7)
- **L-series**: low-power (L0, L1, L4, L5)
- **G-series**: mainstream value/performance (G0, G4)
- **W/LB/WB/WL**: wireless & secure embedded (e.g. BLE, LoRa)

30. What architecture do STM32 MCUs use?
They follow a Harvard architecture, separate instruction and data buses, with Cortex-M cores, memory, debug, and peripherals

31. Explain the role of RCC in STM32.
The Reset and Clock Control (RCC) unit manages clock sources like HSI/HSE, PLL setup, prescalers, and peripheral resets

32. What is purpose of NVIC?
Nested Vectored Interrupt Controller manages interrupt prioritization and handling with low latency

33. What is SysTick timer used for?
A 24-bit system timer typically used for RTOS ticks or time base generation for delays

34. Compare HAL and LL libraries.

- **HAL (Hardware Abstraction Layer)**: high-level, easy to use
- **LL (Low Layer)**: register-level, faster, more efficient

35. What power modes does STM32 support?
Sleep, Stop, and Standby modes reduce power consumption, useful for battery-powered applications

36. How many memory buses are in STM32?
Three main buses: AHB (system), APB1/APB2 (peripherals), plus instruction/data buses internal to CPU

37. What interfaces are provided for programming?
Standard bootloaders support UART, SPI, USB, I²C etc. SWD/JTAG is used for debugging/programming

38. Explain memory-mapped I/O.
Peripherals are accessed via specific memory addresses; reading/writing those registers controls hardware.

39. What is STM32CubeMX and STM32CubeIDE?
STM32CubeMX: graphical tool to configure MCU and generate init code. CubeIDE: ST's integrated development environment for coding, compiling, and debugging.

40. What is the function of DMA?
Direct Memory Access allows peripherals to transfer data to/from memory without CPU, improving efficiency