LAB 02

CO4203 – MICROCONTROLLERS

NAME : J.R. REMILTAN

REGISTRATION NO: EN102828

INDEX NO : 21/ENG/48

SUBMISSION DATE: 11/12/2025

II. Practical Work

1. Setting up Keil µVision for Embedded C Programming

4) **Briefly discuss the requirement for a STARTUP.A51 file citing its functions.**

- The header file reg51.h includes predefined declarations for the Special Function Registers (SFRs) of the 8051 microcontroller. In situations where this file is unavailable, each SFR must be manually declared using the sfr keyword together with its corresponding memory address. For instance, Port 0 resides at address 0x80, Port 1 at 0x90, and Port 2 at 0xA0 within the 8051 SFR address space. Therefore, the programmer would need to explicitly define these registers, such as using sfr P0 = 0x80, before accessing them in the program. Although this manual method is functional, using reg51.h is more convenient and reliable, as it ensures consistent, error-free register definitions and improves code portability across various 8051 microcontroller versions

6) **Create the program file**

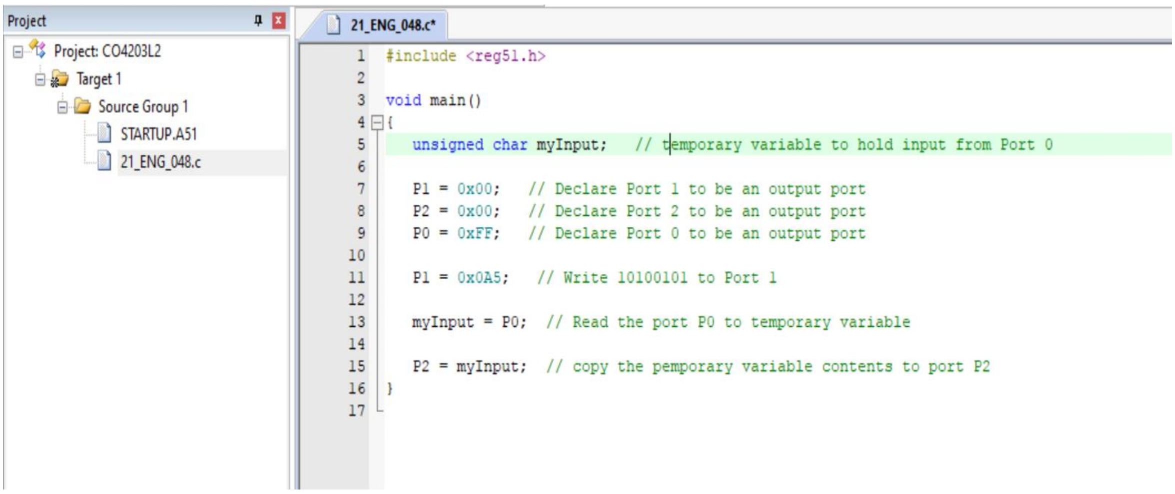


Figure 01: Create the program file

7) **Explain how the above code should have been written if the file was not available.**

- If the reg51.h header file is not available, the programmer must manually define all the Special Function Registers (SFRs) required by the 8051 microcontroller. This is done using the sfr keyword along with their respective memory-mapped addresses.

  For example, ports and control registers must be declared as follows before they are used in the program

  sfr P0 = 0x80; // Port 0

  sfr P1 = 0x90; // Port 1

  sfr P2 = 0xA0; // Port 2

sfr TMOD = 0x89;   // Timer Mode Register

sfr SCON = 0x98;   // Serial Control Register

These definitions allow the compiler to recognize and access the hardware registers directly. Without these declarations, the program will not be able to control the microcontroller ports or peripherals.

12)

Initail

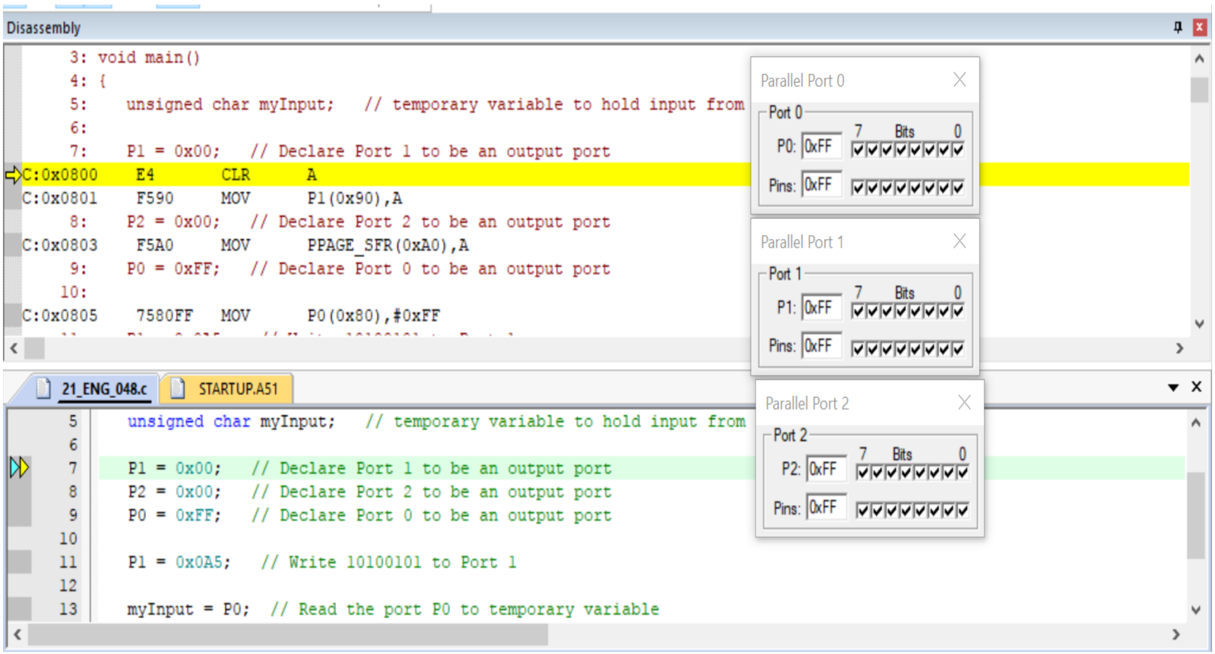


Figure 02: Before execution all ports = 0xFF

Step 01

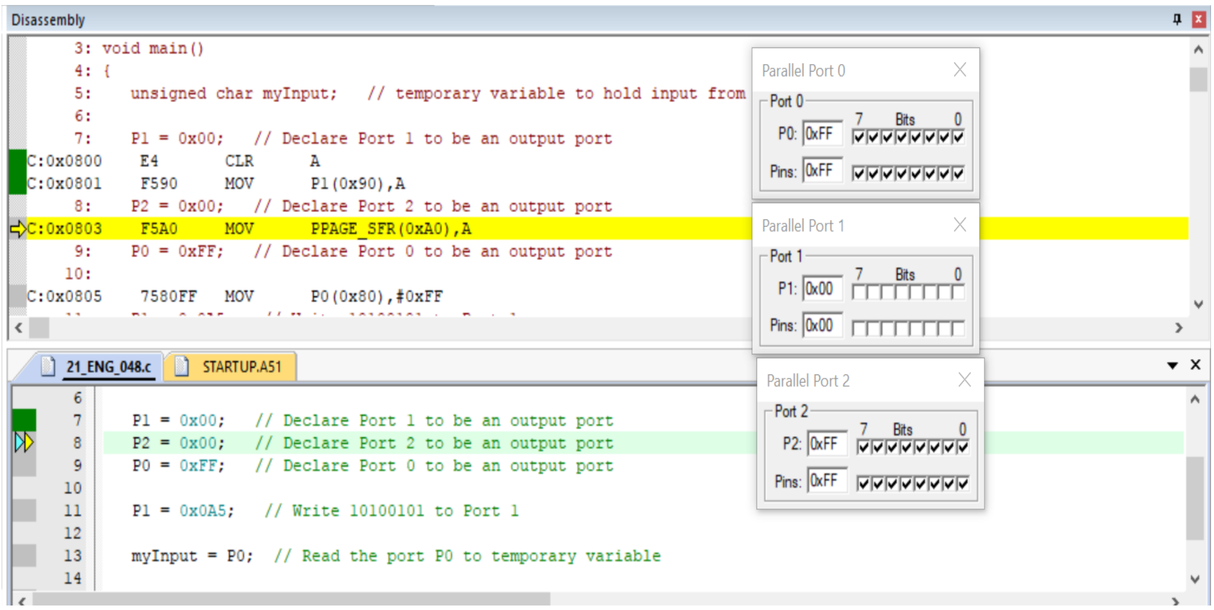


Figure 03: Screenshot of Port 1, 2, 3 after step 1 (port1 = 0x00)

Step 02

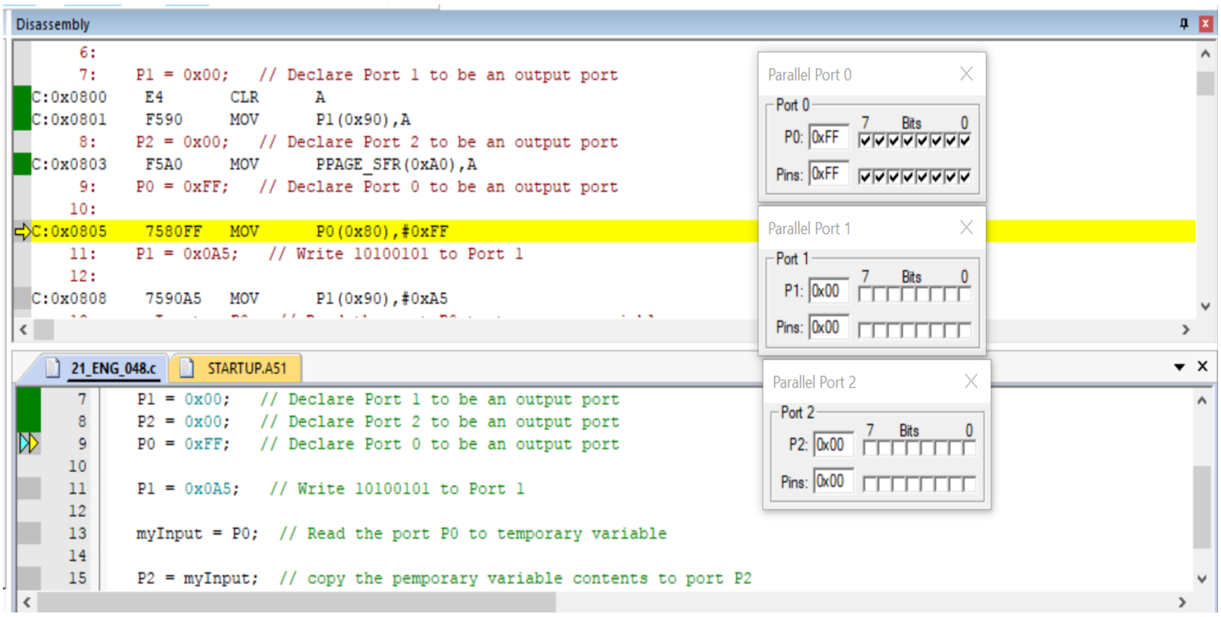


Figure 04: Screenshot of Port 1, 2, 3 after step 2 (port1 = 0x00, port2 = 0x00)

Step 03

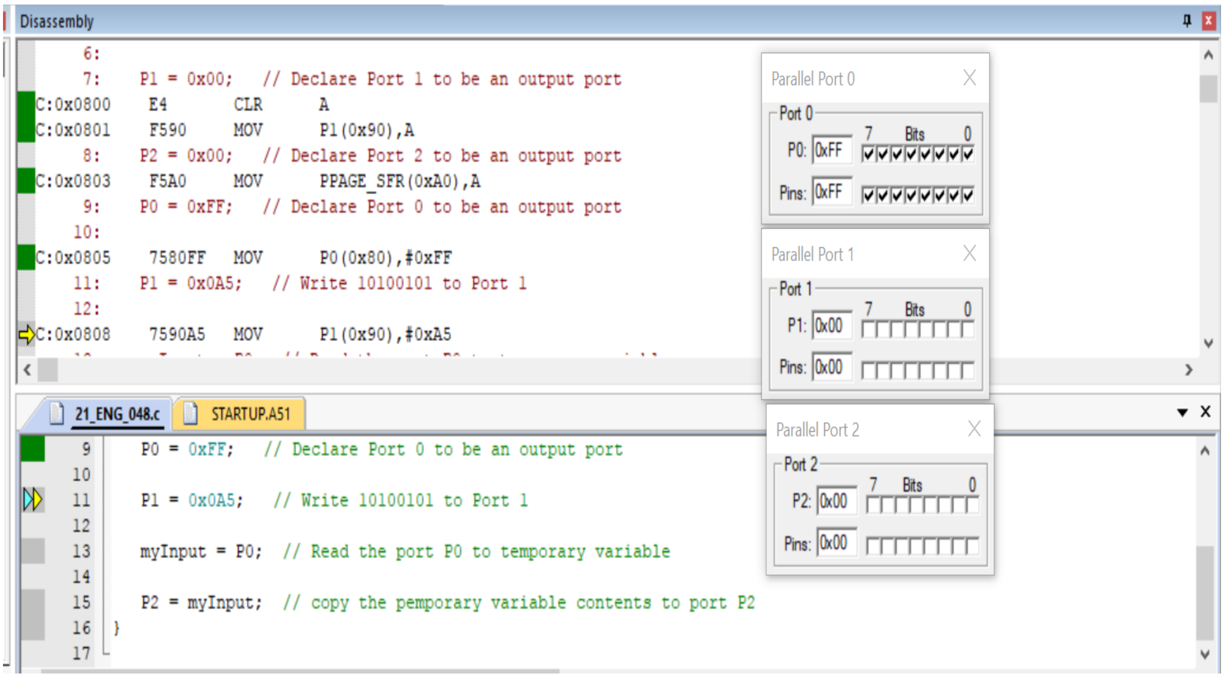


Figure 05: Screenshot of Port 1, 2, 3 after step 3 (port1 = 0x00, port2 = 0x00)

Step 04

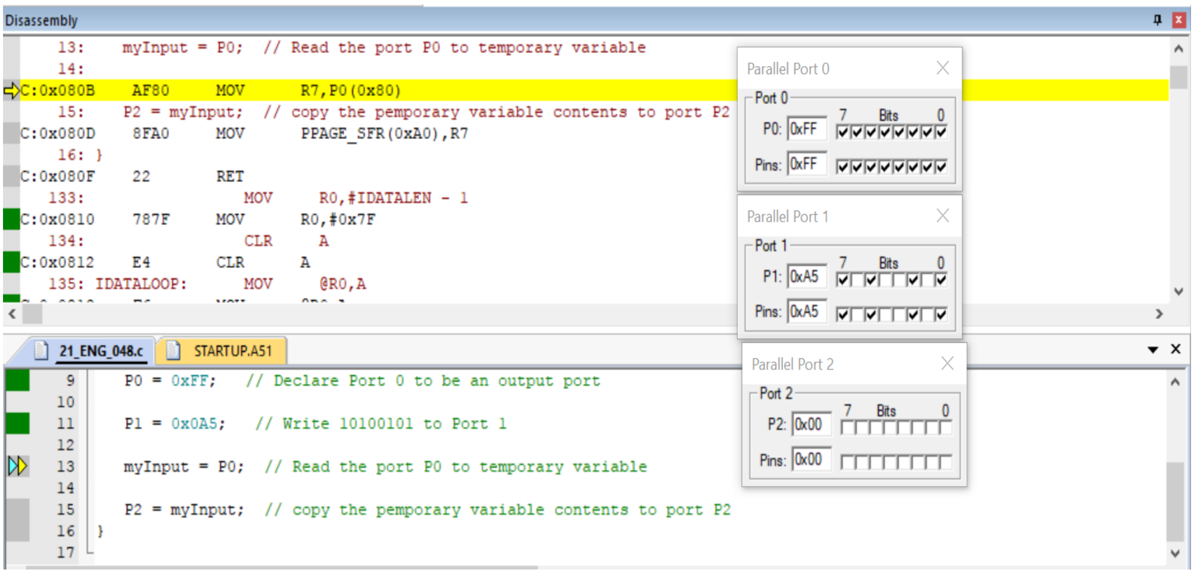
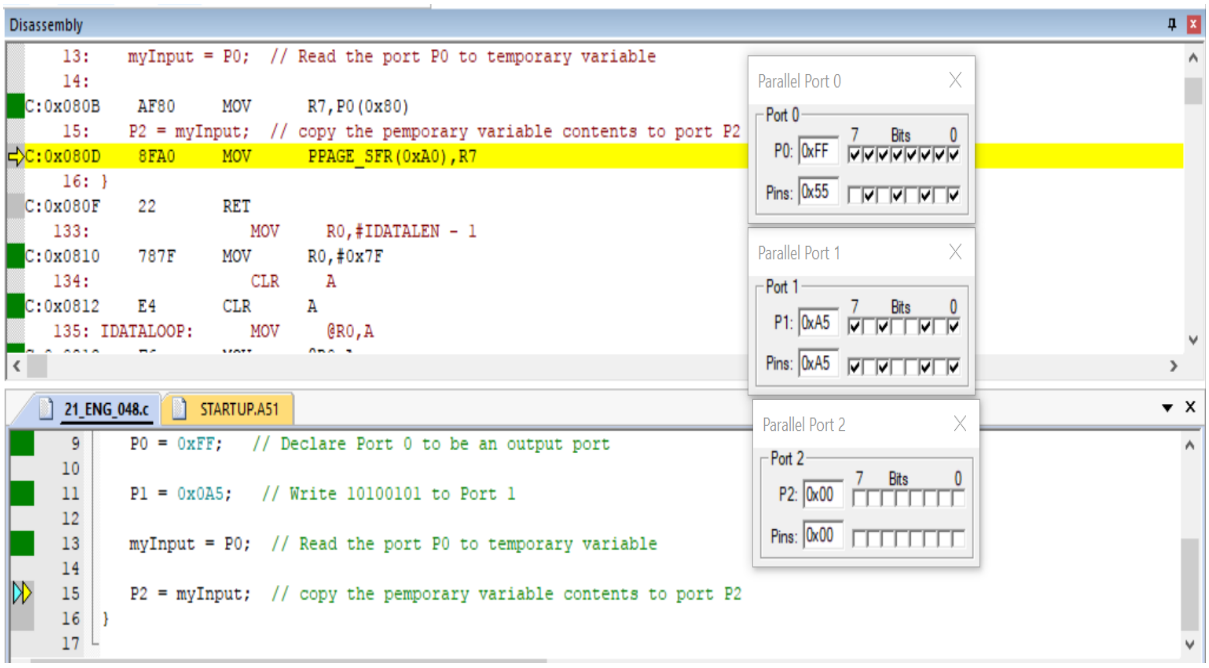


Figure 06: Screenshot of Port 1, 2, 3 after step 4 (port1 = 0xA5, port2 = 0x00)

Step 05



Figire 07: Configure Pins of the Port 0

Step 06

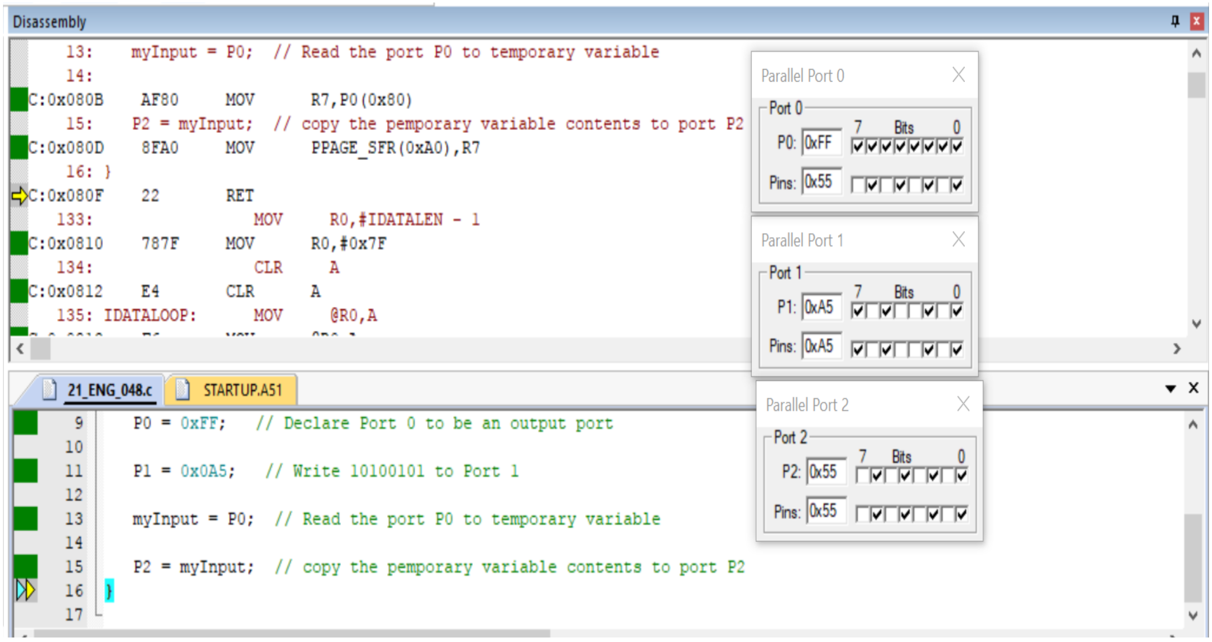


Figure 08: The P0 input has been sent to the P2 output

16)

a. Why the temporary variable myInput is of type unsigned char?

- The variable myInput is declared as an unsigned char because Port 0 of the 8051 microcontroller outputs an 8-bit digital value that can range from 0 to 255. Using an unsigned data type allows the program to correctly store all possible 8-bit values, from 00H to FFH, without misinterpreting any of them as negative numbers. This ensures that the data read from the port is preserved accurately and reflects the true binary state of the input pins.

b. What would happen if the data type of myInput is changed to signed char?

- If myInput is changed to signed char, values greater than 127 (7FH) will be treated as negative numbers due to two’s complement representation. For example, a value like 0xFF (255) would be interpreted as -1. This can lead to incorrect or unexpected results when the value is processed or displayed.

c. What is the reason behind the observations in (i) and (ii)?

The difference occurs because:

- ✓ unsigned char stores values from **0 to 255**
- ✓ signed char stores values from **-128 to +127**

When the most significant bit (MSB) is 1, a signed char treats the value as negative, whereas an unsigned char treats it as a normal positive value. Since microcontroller ports deal with raw binary data, using unsigned types prevents misinterpretation of the data.

d. How you could modify the program without a temporary variable.

- The program can be modified by directly assigning the value of Port 0 to Port 2 using the statement P2 = P0. This removes the need for a temporary variable, reducing memory usage and the number of instructions.

  As a result, the program becomes simpler, more efficient, and executes faster. A temporary variable is only necessary if the data needs to be processed or changed before being sent to Port 2.

2. Serial Communication

17)

21_ENG_048_02.c* | STARTUP.A51

```c
#include <reg51.h>

void main() {
    // Step a: Configure TMOD register in auto reload mode
    TMOD = 0x20;

    // Step b: Configure TH1 register to generate baud rate of 9600
    TH1 = 0xFD;

    // Step c: Configure SCON register (8 bits, 1 stop bit, REN enabled)
    SCON = 0x50;

    // Step d: Start Timer 1
    TR1 = 1;

    while(1) {
        // Step e: Load ASCII character 'C' to SBUF register
        SBUF = 'C';

        // Step f: Pause until complete byte is sent
        while(TI == 0);  // Wait for TI flag to set

        // Step g: Clear the TI bit
        TI = 0;
    }
}
```

Figure 09: Embedded c programme for transmit ASCII character

19) Now press F5 or to run the entire program in debug mode.

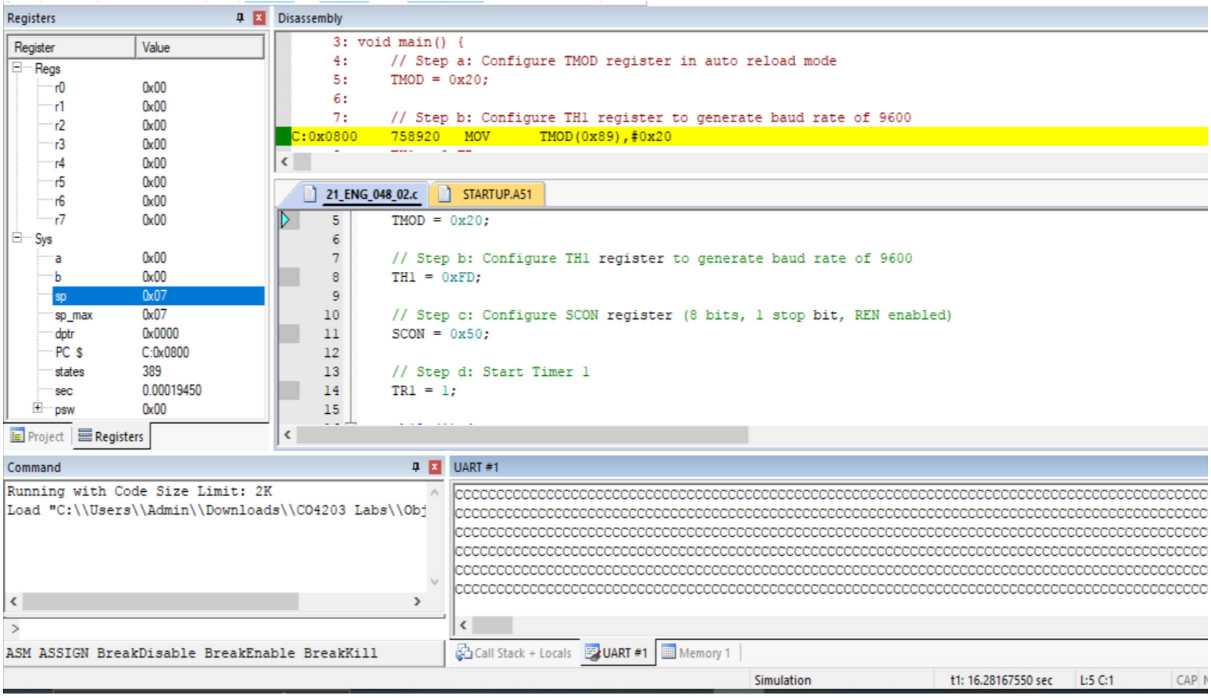


Figure 10: Entire program in debug mode.

20) Take a screenshot of UART #1 window.

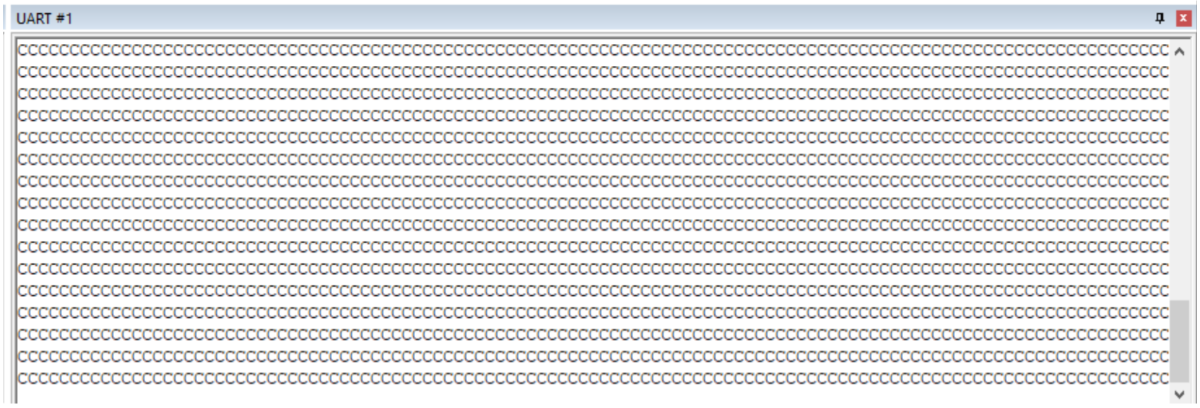


Figure 11: Screenshot with UART#1 window

- ✓ The program successfully transmits the ASCII character **'C' (0x43)** continuously through the UART. The **TMOD register** is set to **0x20**, configuring Timer 1 in auto-reload mode, which ensures a stable and consistent baud rate by automatically reloading the timer value after each overflow. The value **0xFD** loaded into **TH1** produces a baud rate of **9600 bps** when using an **11.0592 MHz crystal oscillator**.
- ✓ The **SCON register** is configured with the value **0x50**, enabling 8-bit UART mode with one stop bit and activating the receiver by setting the REN bit. In the UART #1 window, the character 'C' is observed being transmitted repeatedly, confirming that the serial communication has been correctly configured and that the polling method is functioning properly for detecting transmission completion.

21)

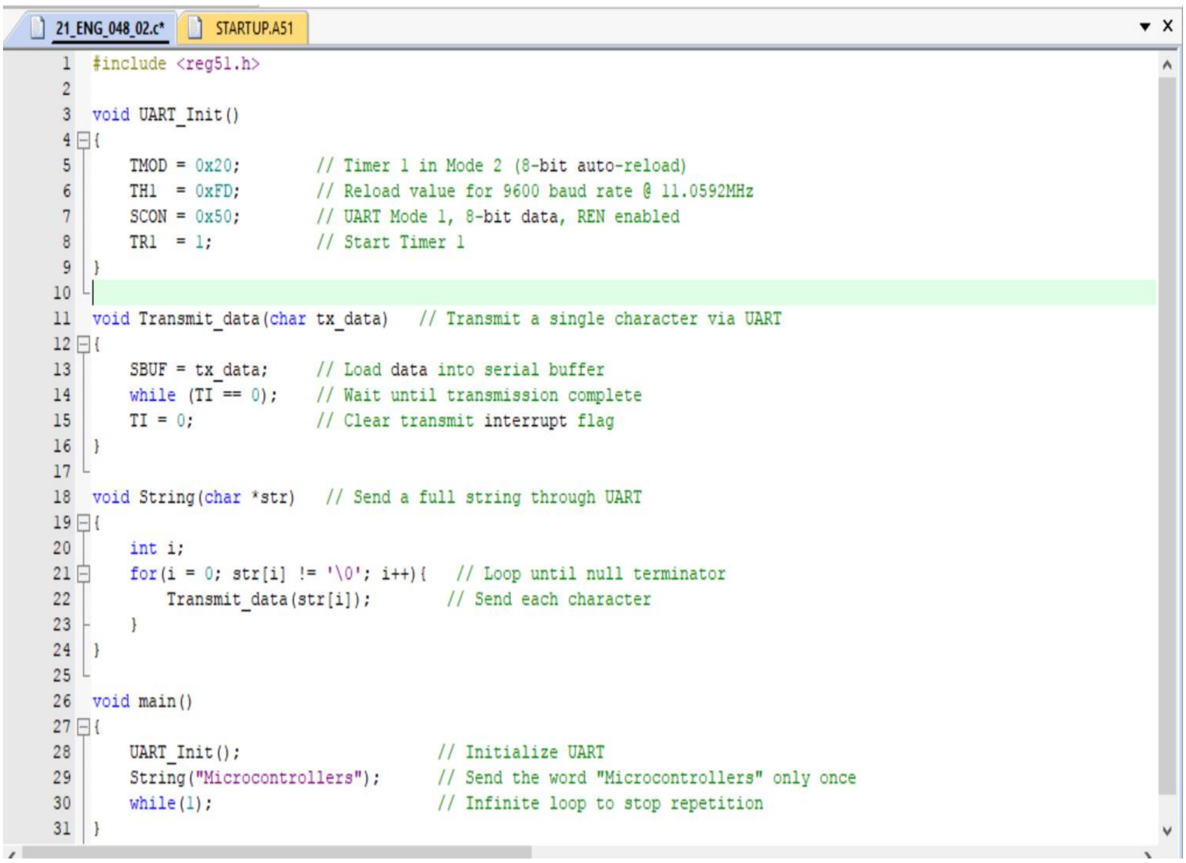


Figure 12: Code for print Microcontreoller

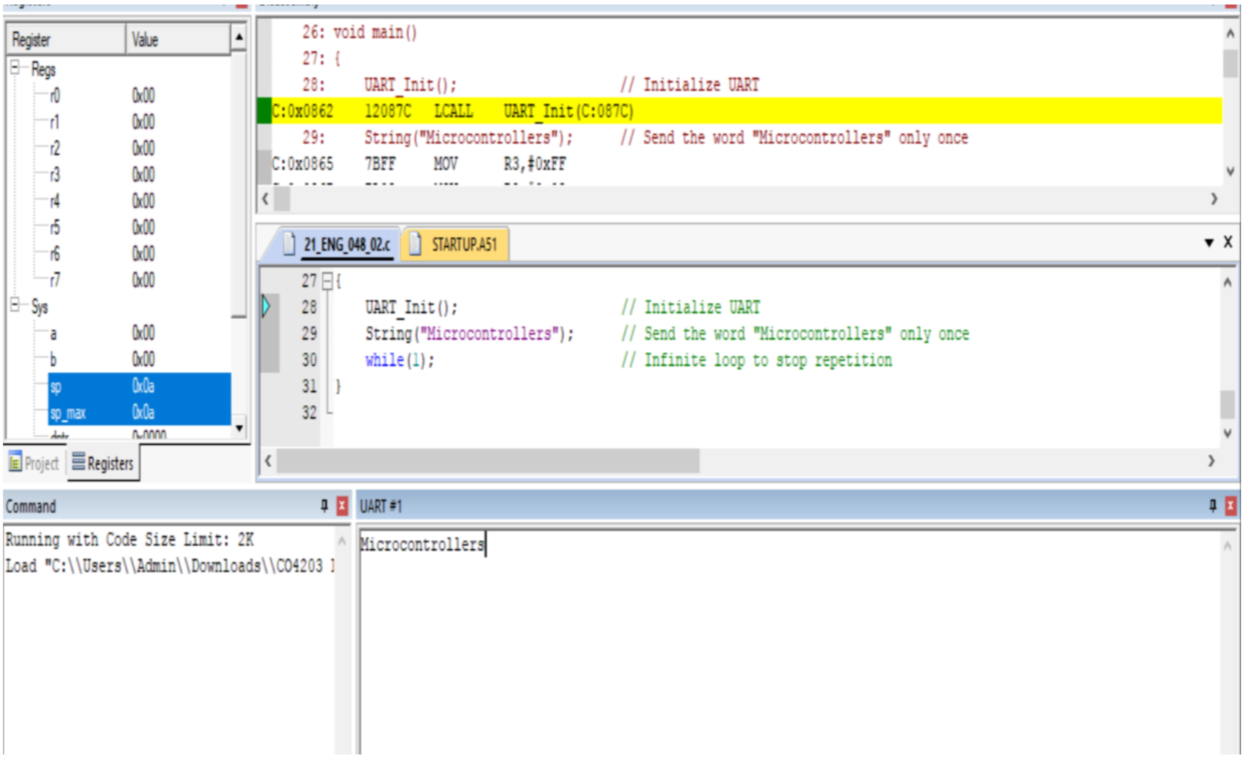


Figure 13: UART output only print “Microcontroller”

- ✓ The program transmits the string "Microcontrollers" once via UART. The UART_Init() function sets up Timer 1 in auto-reload mode for 9600 baud rate and configures 8-bit UART with the receiver enabled. The Transmit_data() function sends one character at a time, ensuring each is fully transmitted before sending the next. The String() function loops through the string until the null terminator, sending each character sequentially.This modular approach keeps the code clear and reusable. The UART #1 window shows the string only once, confirming correct transmission without repetition.

3. Timers

23)

21_ENG_048_03.c*

```c
#include <reg51.h>

sbit WAVE_PIN = P1^0; // Output pin declaration (P1.0 used as square wave output)

void delay() {

    // a. Set Timer 0 to Mode 0 (13-bit mode), keep Timer 1 unchanged
    TMOD = (TMOD & 0xF0);

    // b. Calculate the value that should be loaded onto TH0 and TL0 registers.
    // Load timer values for 1ms delay (TH0 = 0xE3, TL0 = 0x13)

    TH0 = 0xE3;         // c. Load high byte of timer
    TL0 = 0x13;         // d. Load low byte of timer
    TR0 = 1;            // e. Start Timer 0

    while (TF0 == 0);   // f. Wait until Timer 0 overflows (TF0 becomes 1)
    TR0 = 0;            // g. Stop Timer 0
    TF0 = 0;            // h. Clear overflow flag TF0
}

void main() {
    WAVE_PIN = 0;       // Initialize output pin to LOW
    while (1) {
        WAVE_PIN = ~WAVE_PIN;    // Toggle pin to generate square wave
        delay();                  // Call delay to control frequency
    }
}
```

Figure 14: Code for Timers question

25)

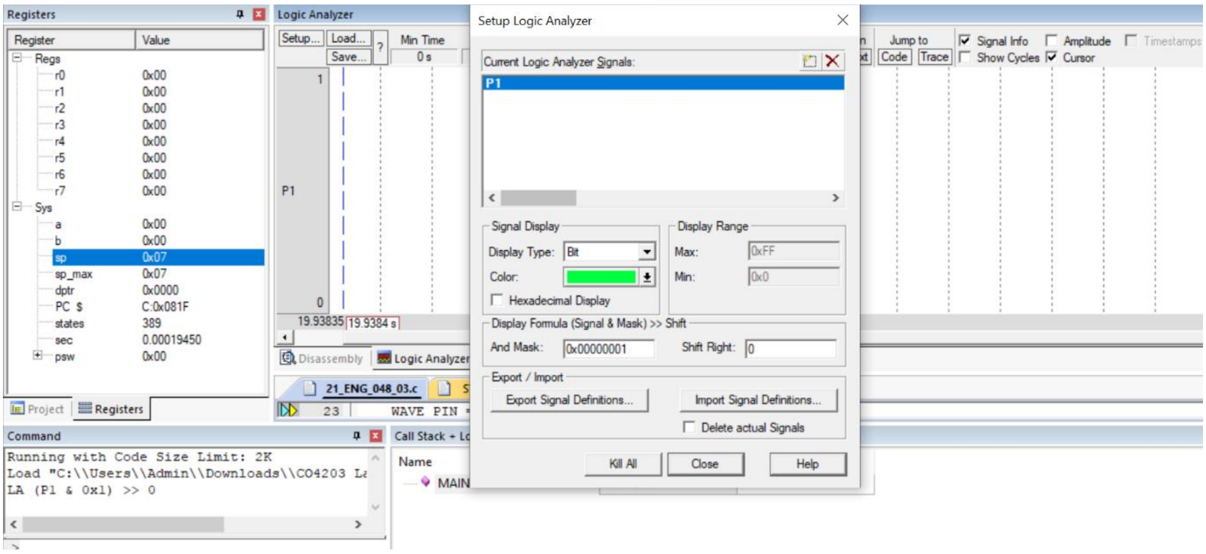


Figure 15: Configure Logic Analyzer setup

29)

Figure 16: Screenshot of Logic Analyzer window

- ✓ The program generates a 2 ms period square wave (1 ms HIGH, 1 ms LOW) on P1.0 using Timer 0 in 13-bit mode (Mode 0). The machine cycle time is calculated as 12 / 11.0592 MHz ≈ 1.085 µs, so a 1 ms delay requires 922 machine cycles**.** In 13-bit mode, the timer counts from 0 to 8191, so the initial count is 8192 – 922 = 7270 (0x1C66)**.**
- ✓ The program loads TH0 = 0xE3 and TL0 = 0x13**,** starts the timer, and waits for the TF0 overflow flag**.** After each 1 ms delay, P1.0 is toggled**,** creating the square wave. The process repeats in an infinite loop. The logic analyser confirms a 2 ms period with equal HIGH and LOW durations**,** showing correct timing and waveform generation.

30)

To generate the same 2 ms period square wave using Timer 0 in Mode 1 (16-bit timer), the initial timer value must be recalculated. In Mode 1, the timer counts from 0 to 65535. For a 1 ms delay requiring 922 machine cycles, the initial count is:

- ✓ 65536 – 922 = 64614 (0xFC66)

The TMOD register is set to 0x01 to select Timer 0 in Mode 1. The 16-bit initial value is loaded as TH0 = 0xFC (high byte) and TL0 = 0x66 (low byte). Timer 1 is started, and the TF0 flag is polled as in Mode 0.

Mode 1 provides higher resolution, easier calculations, and a wider timing range, making it more precise than Mode 0 while the rest of the program structure (starting, polling, and stopping the timer) remains the same.