

LAB 02

CO4203 – MICROCONTROLLERS

NAME : J.R. REMILTAN
REGISTRATION NO: EN102828
INDEX NO : 21/ENG/48
SUBMISSION DATE: 11/12/2025

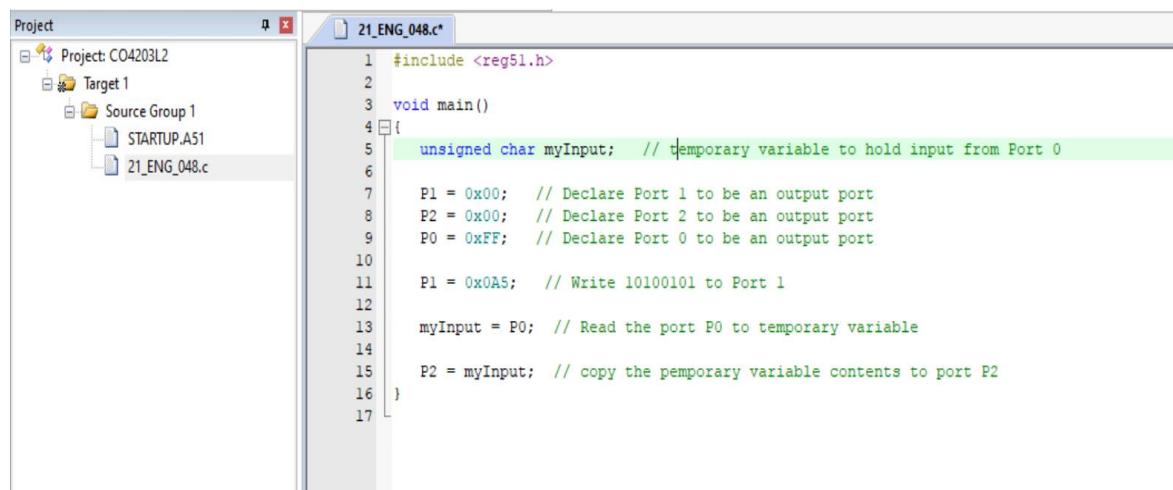
II. Practical Work

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



```
Project 21_ENG_048.c*
File Project: CO4203L2
  Target 1
    Source Group 1
      STARTUP.A51
      21_ENG_048.c

21_ENG_048.c*
1 #include <reg51.h>
2
3 void main()
4 {
5     unsigned char myInput; // temporary variable to hold input from Port 0
6
7     P1 = 0x00; // Declare Port 1 to be an output port
8     P2 = 0x00; // Declare Port 2 to be an output port
9     P0 = 0xFF; // Declare Port 0 to be an output port
10
11    P1 = 0x0A5; // Write 10100101 to Port 1
12
13    myInput = P0; // Read the port P0 to temporary variable
14
15    P2 = myInput; // copy the temporary variable contents to port P2
16 }
17
```

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

Disassembly

```
3: void main()
4: {
5:     unsigned char myInput; // temporary variable to hold input from
6:
7:     P1 = 0x00; // Declare Port 1 to be an output port
C:0x0800 E4 CLR A
C:0x0801 F590 MOV P1(0x90),A
8:     P2 = 0x00; // Declare Port 2 to be an output port
C:0x0803 F5A0 MOV PPAGE_SFR(0xA0),A
9:     P0 = 0xFF; // Declare Port 0 to be an output port
10:
C:0x0805 7580FF MOV P0(0x80),#0xFF
```

21_ENG_048.c STARTUP.A51

```
5:     unsigned char myInput; // temporary variable to hold input from
6:
7:     P1 = 0x00; // Declare Port 1 to be an output port
8:     P2 = 0x00; // Declare Port 2 to be an output port
9:     P0 = 0xFF; // Declare Port 0 to be an output port
10:
11:    P1 = 0xA5; // Write 10100101 to Port 1
12:
13:    myInput = P0; // Read the port P0 to temporary variable
```

Parallel Port 0

Port 0	7 Bits	0
P0:	0xFF	████████████████
Pins:	0xFF	████████████████

Parallel Port 1

Port 1	7 Bits	0
P1:	0xFF	████████████████
Pins:	0xFF	████████████████

Parallel Port 2

Port 2	7 Bits	0
P2:	0xFF	████████████████
Pins:	0xFF	████████████████

Figure 02: Before execution all ports = 0xFF

Step 01

Disassembly

```
3: void main()
4: {
5:     unsigned char myInput; // temporary variable to hold input from
6:
7:     P1 = 0x00; // Declare Port 1 to be an output port
C:0x0800 E4 CLR A
C:0x0801 F590 MOV P1(0x90),A
8:     P2 = 0x00; // Declare Port 2 to be an output port
C:0x0803 F5A0 MOV PPAGE_SFR(0xA0),A
9:     P0 = 0xFF; // Declare Port 0 to be an output port
10:
C:0x0805 7580FF MOV P0(0x80),#0xFF
```

21_ENG_048.c STARTUP.A51

```
6:
7:     P1 = 0x00; // Declare Port 1 to be an output port
8:     P2 = 0x00; // Declare Port 2 to be an output port
9:     P0 = 0xFF; // Declare Port 0 to be an output port
10:
11:    P1 = 0xA5; // Write 10100101 to Port 1
12:
13:    myInput = P0; // Read the port P0 to temporary variable
```

Parallel Port 0

Port 0	7 Bits	0
P0:	0x00	████████████████
Pins:	0x00	████████████████

Parallel Port 1

Port 1	7 Bits	0
P1:	0x00	████████████████
Pins:	0x00	████████████████

Parallel Port 2

Port 2	7 Bits	0
P2:	0xFF	████████████████
Pins:	0xFF	████████████████

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

Step 02

The screenshot shows the Keil MDK-ARM IDE interface. On the left, the Disassembly window displays assembly code for the 21_ENG_048.c project. The code initializes ports P0, P1, and P2 to 0x00, then writes 0xA5 to P1 and reads it into a temporary variable myInput. Finally, it copies myInput to P2. The assembly code is as follows:

```
6:    P1 = 0x00; // Declare Port 1 to be an output port
C:0x0800  E4      CLR     A
C:0x0801  F590    MOV     P1(0x90),A
8:    P2 = 0x00; // Declare Port 2 to be an output port
C:0x0803  F5A0    MOV     PPAGE_SFR(0xA0),A
9:    P0 = 0xFF; // Declare Port 0 to be an output port
10:
C:0x0805  7580FF  MOV     P0(0x80),#0xFF
11:   P1 = 0xA5; // Write 10100101 to Port 1
12:
C:0x0808  7590A5  MOV     P1(0x90),#0xA5
13:
14:
15:
```

On the right, three parallel port configuration windows are open: Parallel Port 0, Parallel Port 1, and Parallel Port 2. Each window shows a bit mask for the port and a pin configuration table. For Parallel Port 0, the bit mask is 0xFF and pins are all checked (0xFF). For Parallel Port 1, the bit mask is 0x00 and pins are all unchecked (0x00). For Parallel Port 2, the bit mask is 0x00 and pins are all unchecked (0x00).

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

Step 03

The screenshot shows the Keil MDK-ARM IDE interface. The assembly code has been modified to declare Port 0 as an output port before writing to it. The code is as follows:

```
6:
7:    P1 = 0x00; // Declare Port 1 to be an output port
C:0x0800  E4      CLR     A
C:0x0801  F590    MOV     P1(0x90),A
8:    P2 = 0x00; // Declare Port 2 to be an output port
C:0x0803  F5A0    MOV     PPAGE_SFR(0xA0),A
9:    P0 = 0xFF; // Declare Port 0 to be an output port
10:
C:0x0805  7580FF  MOV     P0(0x80),#0xFF
11:   P1 = 0xA5; // Write 10100101 to Port 1
12:
C:0x0808  7590A5  MOV     P1(0x90),#0xA5
13:
14:
15:   myInput = P0; // Read the port P0 to temporary variable
16:
17:   P2 = myInput; // copy the temporary variable contents to port P2
}
```

The parallel port configuration windows remain the same as in Figure 04, showing Port 0 with bit mask 0xFF and pins 0xFF, Port 1 with bit mask 0x00 and pins 0x00, and Port 2 with bit mask 0x00 and pins 0x00.

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

Step 04

The screenshot shows the Keil MDK-ARM IDE interface. On the left, the Disassembly window displays assembly code for a program named STARTUP.A51. The code includes instructions to read from Port 0, write to Port 1, and copy temporary variables between them. On the right, three parallel port configuration windows are open: Parallel Port 0, Parallel Port 1, and Parallel Port 2. Parallel Port 0 is set to P0: 0xFF and Pins: 0xFF. Parallel Port 1 is set to P1: 0xA5 and Pins: 0xA5. Parallel Port 2 is set to P2: 0x00 and Pins: 0x00.

```
Disassembly
13: myInput = P0; // Read the port P0 to temporary variable
14:
C:0x080B AF80 MOV R7,P0(0x80)
15: P2 = myInput; // copy the temporary variable contents to port P2
C:0x080D 8FA0 MOV PPAGE_SFR(0xA0),R7
16: }
C:0x080F 22 RET
133: MOV R0,#IDATALEN - 1
C:0x0810 787F MOV R0,#0x7F
134: CLR A
C:0x0812 E4 CLR A
135: IDATALOOP: MOV @R0,A
C:0x0814 22 RET
136: }

21_ENG_048.c STARTUP.A51
9: P0 = 0xFF; // Declare Port 0 to be an output port
10:
11: P1 = 0xA5; // Write 10100101 to Port 1
12:
13: myInput = P0; // Read the port P0 to temporary variable
14:
15: P2 = myInput; // copy the temporary variable contents to port P2
16: }
```

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

Step 05

The screenshot shows the Keil MDK-ARM IDE interface. The assembly code remains largely the same as in Step 04, with the addition of a new instruction at address C:0x080D: 8FA0 MOV PPAGE_SFR(0xA0),R7. The parallel port configurations are identical to Step 04: Parallel Port 0 (P0: 0xFF, Pins: 0xFF), Parallel Port 1 (P1: 0xA5, Pins: 0xA5), and Parallel Port 2 (P2: 0x00, Pins: 0x00).

```
Disassembly
13: myInput = P0; // Read the port P0 to temporary variable
14:
C:0x080B AF80 MOV R7,P0(0x80)
15: P2 = myInput; // copy the temporary variable contents to port P2
C:0x080D 8FA0 MOV PPAGE_SFR(0xA0),R7
16: }
C:0x080F 22 RET
133: MOV R0,#IDATALEN - 1
C:0x0810 787F MOV R0,#0x7F
134: CLR A
C:0x0812 E4 CLR A
135: IDATALOOP: MOV @R0,A
C:0x0814 22 RET
136: }

21_ENG_048.c STARTUP.A51
9: P0 = 0xFF; // Declare Port 0 to be an output port
10:
11: P1 = 0xA5; // Write 10100101 to Port 1
12:
13: myInput = P0; // Read the port P0 to temporary variable
14:
15: P2 = myInput; // copy the temporary variable contents to port P2
16: }
```

Figire 07: Configure Pins of the Port 0

Step 06

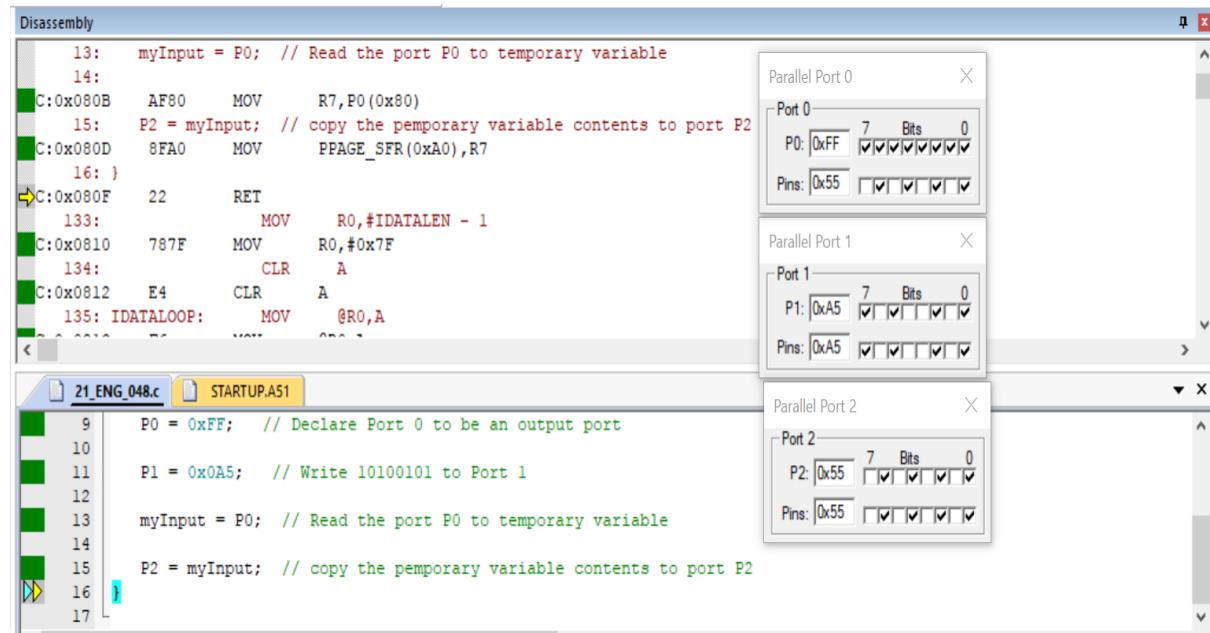


Figure 08: The P0 input has been sent to the P2 output

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
1 #include <reg51.h>
2
3 void main() {
4     // Step a: Configure TMOD register in auto reload mode
5     TMOD = 0x20;
6
7     // Step b: Configure TH1 register to generate baud rate of 9600
8     TH1 = 0xFD;
9
10    // Step c: Configure SCON register (8 bits, 1 stop bit, REN enabled)
11    SCON = 0x50;
12
13    // Step d: Start Timer 1
14    TR1 = 1;
15
16    while(1) {
17        // Step e: Load ASCII character 'C' to SBUF register
18        SBUF = 'C';
19
20        // Step f: Pause until complete byte is sent
21        while(TI == 0); // Wait for TI flag to set
22
23        // Step g: Clear the TI bit
24        TI = 0;
25    }
26}
```

Figure 09: Embedded c programme for transmit ASCII character

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

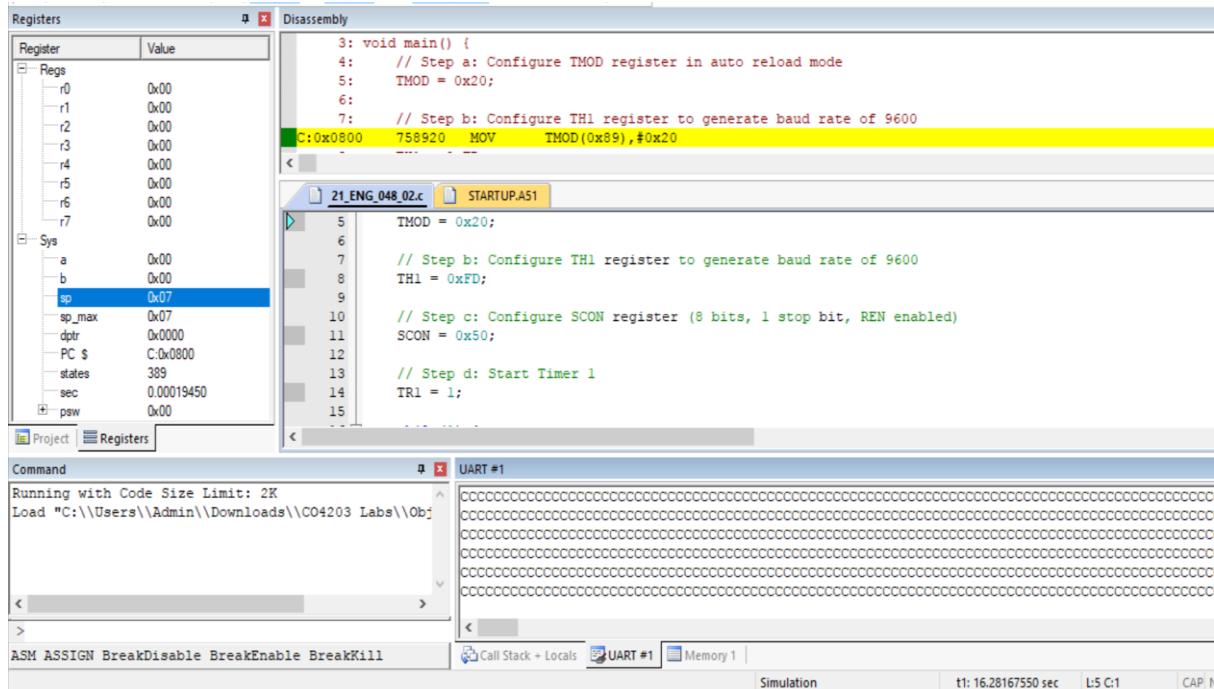


Figure 10: Entire program in debug mode.

20) Take a screenshot of UART #1 window.

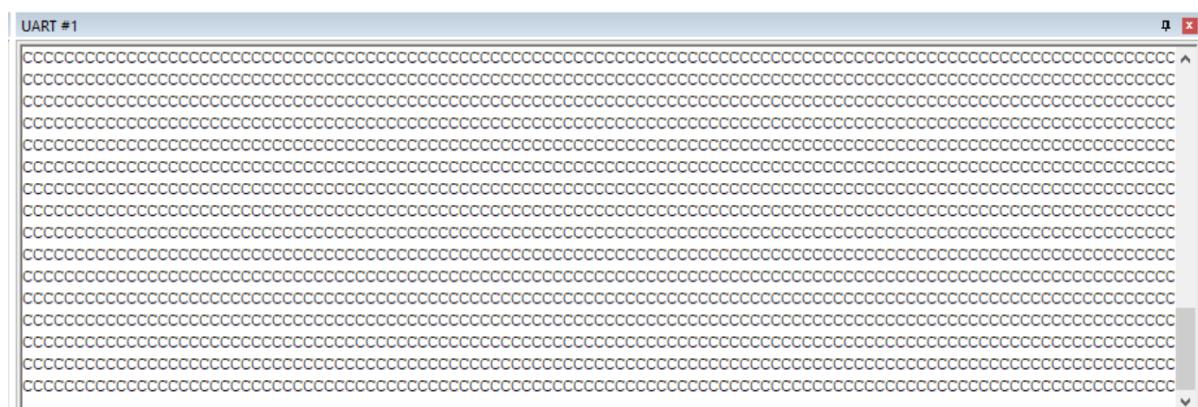


Figure 11: Screenshot with UART#1 window

- ✓ 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)

The screenshot shows a code editor window with two tabs: "21_ENG_048_02.c*" and "STARTUP.A51". The code in "21_ENG_048_02.c" is as follows:

```
1 #include <reg51.h>
2
3 void UART_Init()
4 {
5     TMOD = 0x20;          // Timer 1 in Mode 2 (8-bit auto-reload)
6     TH1 = 0xFD;           // Reload value for 9600 baud rate @ 11.0592MHz
7     SCON = 0x50;          // UART Mode 1, 8-bit data, REN enabled
8     TR1 = 1;              // Start Timer 1
9 }
10
11 void Transmit_data(char tx_data) // Transmit a single character via UART
12 {
13     SBUF = tx_data;        // Load data into serial buffer
14     while (TI == 0);       // Wait until transmission complete
15     TI = 0;                // Clear transmit interrupt flag
16 }
17
18 void String(char *str) // Send a full string through UART
19 {
20     int i;
21     for(i = 0; str[i] != '\0'; i++) { // Loop until null terminator
22         Transmit_data(str[i]);      // Send each character
23     }
24 }
25
26 void main()
27 {
28     UART_Init();            // Initialize UART
29     String("Microcontrollers"); // Send the word "Microcontrollers" only once
30     while(1);               // Infinite loop to stop repetition
31 }
```

Figure 12: Code for print Microcontreoller

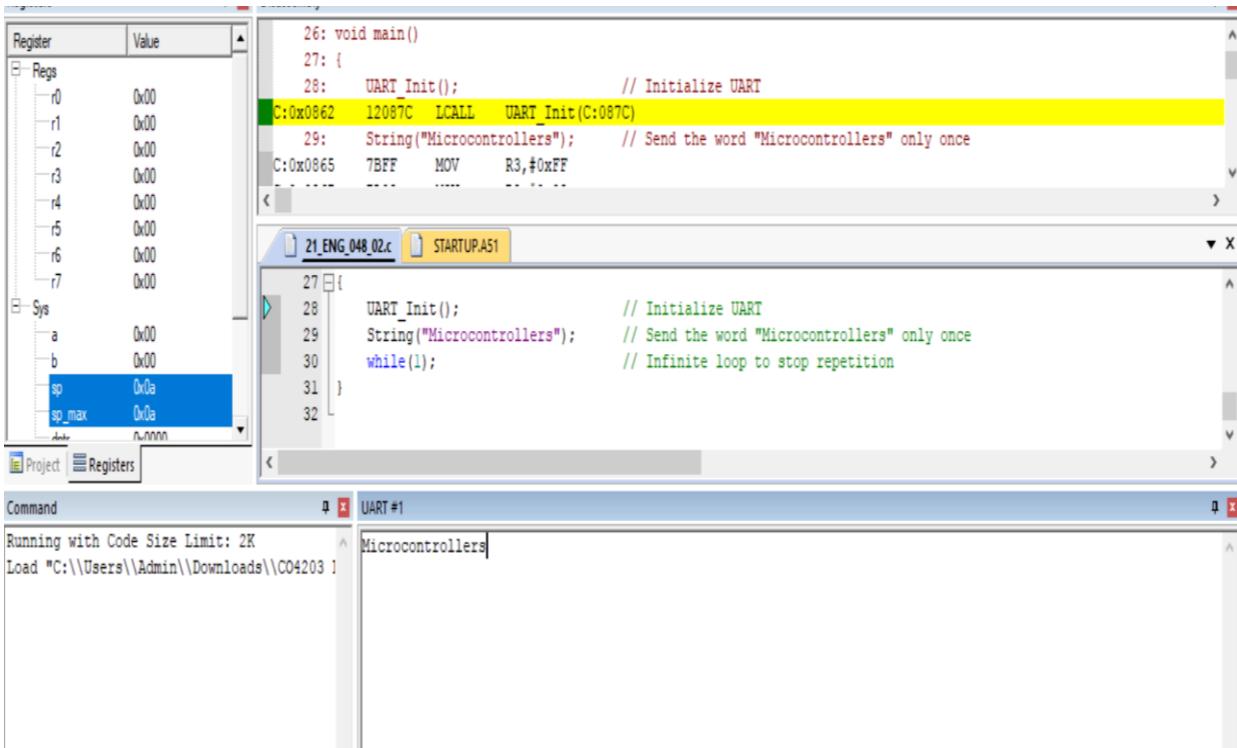


Figure 13: UART output only print “Microcontroller”

- ✓ 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)

```

1  #include <reg51.h>
2
3  sbit WAVE_PIN = P1^0; // Output pin declaration (P1.0 used as square wave output)
4
5 void delay() {
6
7      // a. Set Timer 0 to Mode 0 (13-bit mode), keep Timer 1 unchanged
8      TMOD = (TMOD & 0xF0);
9
10     // b. Calculate the value that should be loaded onto TH0 and TL0 registers.
11     // Load timer values for 1ms delay (TH0 = 0xE3, TL0 = 0x13)
12
13     TH0 = 0xE3;           // c. Load high byte of timer
14     TL0 = 0x13;           // d. Load low byte of timer
15     TR0 = 1;              // e. Start Timer 0
16
17     while (TF0 == 0);    // f. Wait until Timer 0 overflows (TF0 becomes 1)
18     TR0 = 0;              // g. Stop Timer 0
19     TF0 = 0;              // h. Clear overflow flag TF0
20 }
21
22 void main() {
23     WAVE_PIN = 0;          // Initialize output pin to LOW
24     while (1) {
25         WAVE_PIN = ~WAVE_PIN; // Toggle pin to generate square wave
26         delay();             // Call delay to control frequency
27     }
28 }
29

```

Figure 14: Code for Timers question

25)

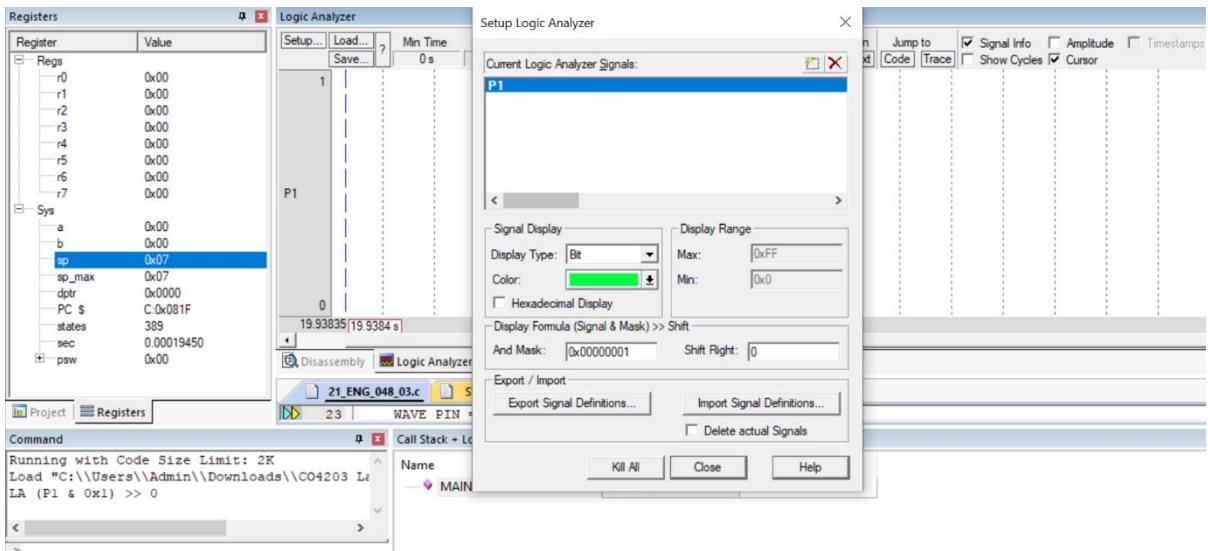


Figure 15: Configure Logic Analyzer setup

29)

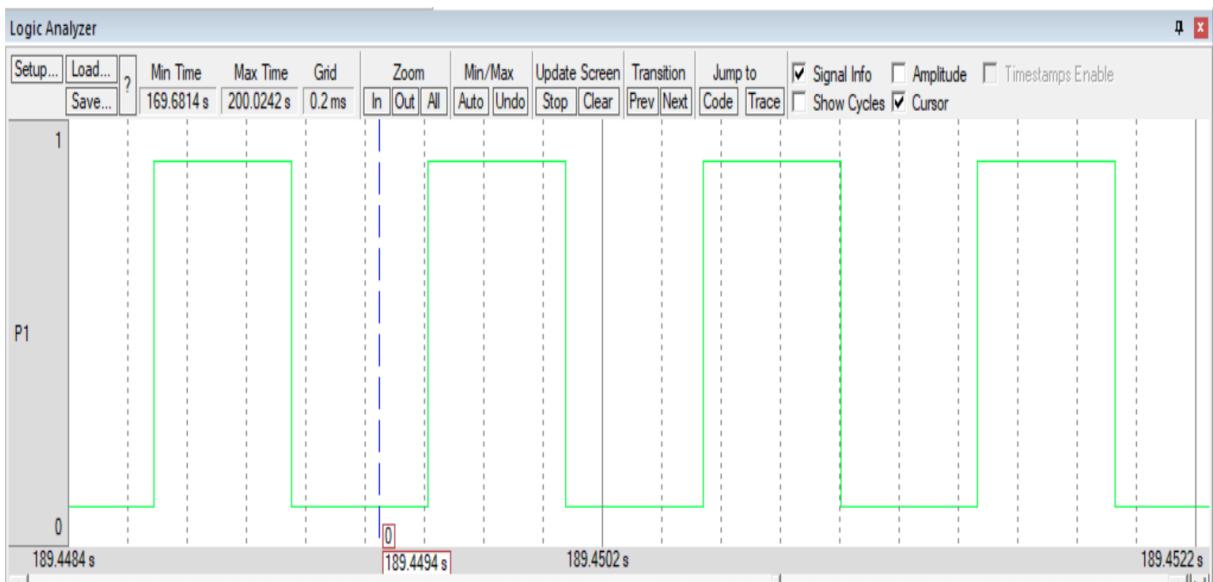


Figure 16: Screenshot of Logic Analyzer window

- ✓ 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 $12 / 11.0592 \text{ MHz} \approx 1.085 \mu\text{s}$, 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.