EE 337: MICROPROCESSOR LAB

Lab No. 09 - B (part2)

Yaswanth Ram Kumar
23B1277

Dept. of Electrical Engineering
Indian Institute of Technology Bombay

Aim

In this part of the lab, we extend the board-to-board communication system implemented in Part-1 by introducing UART communication to accept dynamic inputs from a laptop and integrating assembly-level computation into the C code. The Master board receives two numbers from a laptop using UART, forwards them to the Slave via SPI, and then transmits the response back to the laptop via UART. Additionally, the arithmetic computation is offloaded to an external assembly routine for better performance. Finally, as a bonus, the addition operation is replaced by a MAC (Multiply and Accumulate) operation using previously implemented assembly code.

SPCON Register Configuration

From the AT89C5131A datasheet `https://ww1.microchip.com/downloads/en/DeviceDoc/doc4337.pdf`, the SPCON register is configured with the hexadecimal value `0x7F` for Master and `0x6F` for Slave, based on the following bit settings:

Bit	Name	Description	Master Value	Slave Value
7	SPR2	Reserved (must be 0)	0	0
6	SPEN	1 = Enable SPI	1	1
5	SSDIS	1 = Disable Slave Select (SSBAR)	1	1
4	MSTR	1 = Set MCU as SPI Master	1	0
3	CPOL	1 = SCK idle high	1	1
2	CPHA	1 = Data sampled on the second edge	1	1
1–0	SPR1:0	11 = SPI Clock = $F_{clk}/16$	11	11

Master Configuration:
Binary Representation: `0b01111111`
Hexadecimal Value: `0x7F`

Slave Configuration:
Binary Representation: `0b01101111`
Hexadecimal Value: `0x6F`

Code Files and SPI Register Explanation

The SPI communication setup in this project was implemented from scratch based on the AT89C5131 datasheet. Two separate files were created: `master_SPI.c` and `slave_SPI.c`, which contain the full implementation for the SPI Master and SPI Slave respectively. Instead of using the provided `SPI.h` file, we directly accessed and configured the necessary Special Function Registers (SFRs) for SPI, allowing for better understanding and finer control over the protocol.

SPDAT (SPI Data Register): This is the data register used for both transmitting and receiving a byte via SPI. Writing to `SPDAT` initiates the transmission, and once the transfer is complete, the received data from the slave (or master, depending on the mode) is available in the same register.

SPSTA (SPI Status Register): This register contains status flags related to SPI transmission. The most important bit used in our code is:

- `SPIF (SPI Interrupt Flag) { Bit 7`: This bit is automatically set by hardware when a data transfer is complete, indicating that the `SPDAT` register now contains valid received data. Our code continuously checks this bit (using `SPSTA & 0x80`) to determine when it's safe to read from `SPDAT`.

Overall, this approach ensures tight control of SPI communication and provides valuable experience in low-level embedded system programming.

1 SPI Master Code Explanation

The SPI Master code is designed to interface with the SPI Slave and communicate prime numbers. The main operations are:

1.1 UART Communication Setup

In the 'uart_init()' function, Timer 1 is set up to generate the required baud rate for serial communication. The baud rate is calculated based on the microcontroller's crystal frequency (24 MHz) and the desired baud rate (1200 bps). The 'SCON' register is configured for 8-bit serial communication. Timer 1 is started to generate the clock for UART transmission, and the serial interrupt is enabled.

```
void uart_init(void)
{
    TMOD = 0x20;                // Set Timer 1 Mode 2 (8-bit auto-reload)
    TH1 = 0xCC;                 // Load TH1 for 1200 bps with 24 MHz crystal
    SCON = 0x50;                // Configure UART for 8-bit data transfer
    TR1 = 1;                    // Start Timer 1
    ES = 1;                     // Enable Serial Interrupt
    EA = 1;                     // Enable Global Interrupt
}
```

1.2 Receiving and Sending Data

The 'receive_num()' function reads numeric data from the UART input until the user presses Enter. The function processes the received characters, constructs the number, and returns it as an unsigned 8-bit integer. The 'spi_send()' function sends a byte of data over SPI and waits for the transfer to complete.

```
unsigned char receive_num(void)
{
    char buffer[4];  // Max 3 digits + null terminator
    unsigned char i = 0;
    unsigned char num = 0;
    char ch;

    while (1)
    {
```

```
        ch = receive_char();        // Get character from UART
        transmit_char(ch);          // Echo back

        if (ch == '\r')             // Enter key (Carriage Return)
        {
            buffer[i] = '\0';       // Null-terminate the string
            break;
        }

        if (ch >= '0' && ch <= '9' && i < 3)
        {
            buffer[i++] = ch;
        }
    }

    for (i = 0; buffer[i] != '\0'; i++)
    {
        num = num * 10 + (buffer[i] - '0');
    }

    return num;  // Will be in range 0{255
}
```

The master sends two prime numbers via SPI, waits for a response, and displays the result on the LCD.

```
void main(void)
{
    unsigned char prime1 = 7, prime2 = 13;
    uart_init();      // Initialize UART for communication
    lcd_init();       // Initialize LCD
    master_greet_lcd();

    transmit_string("Enter the prime number 'a': ");
    prime1 = receive_num();
    transmit_string("Enter the prime number 'b': ");
    prime2 = receive_num();

    SPCON = 0x7F;  // Master mode, Fclk/16, CPOL=1, CPHA=1, SSDIS=1
    spi_send(prime1);
    spi_send(prime2);
    // Wait for response from the slave
    response = spi_send(0x00);

    if (response == 1)
    {
        lcd_write_string("Sum is Prime");
```

```
    }
    else if (response == 2)
    {
        lcd_write_string("Sum Not Prime");
    }
    else
    {
        lcd_write_string("Unexpected Value");
    }
    while (1);
}
```

2 SPI Slave Code Explanation

The SPI Slave code receives the two prime numbers sent by the master, computes their sum using assembly, and checks if the sum is prime. It then sends the result back to the master.

2.1 Includes and Setup

Similar to the master code, the slave code includes necessary headers and initializes the LCD display.

```
#include <at89c5131.h>
#include "lcd.h"
```

The ‘lcd_init()‘ and ‘slave_greet_lcd()‘ functions set up the LCD to display status messages.

2.2 Assembly Code Integration

The slave uses an assembly routine ‘ADD_ASM()‘ to add the two received numbers. The assembly code is written in ‘add.asm‘ and performs the addition by accessing the values stored at memory locations ‘0x30‘ and ‘0x31‘, and storing the result at ‘0x32‘.

```
extern void ADD_ASM(void);
unsigned char *addr_30 = (unsigned char *)0x30;
unsigned char *addr_31 = (unsigned char *)0x31;
unsigned char *addr_32 = (unsigned char *)0x32;
unsigned int sum_asm;
```

```
; add.asm
; Adds values from 0x30 and 0x31, stores result in 0x32

PUBLIC ADD_ASM

CSEG AT 0100H    ; Start from address 0x100

ADD_ASM:
```

```
    MOV A, 30H
    ADD A, 31H
    MOV 32H, A
    RET
```

2.3 Receiving Data and Calculating Sum

The slave code receives the first prime number and stores it in ‘prime1‘, then receives the second prime number and stores it in ‘prime2‘. The assembly routine is called to calculate the sum, which is checked for primality.

```
void main() {
    unsigned char received;
    unsigned char sum;
    lcd_init();
    slave_greet_lcd();

    SPCON = 0x6F;  // Slave mode, Fclk/16, CPOL=1, CPHA=1, SSDIS=1

    while (1) {
        while (!(SPSTA & 0x80));  // Wait for transfer complete
        received = SPDAT;

        if (!received_first) {
            prime1 = received;
            received_first = 1;
        } else {
            prime2 = received;
            a = (int)prime1;
            b = (int)prime2;
            *addr_30 = a;
            *addr_31 = b;
            *addr_32 = 123;  // Set it manually first
            ADD_ASM();  // Perform the addition
            sum_asm = *addr_32;
            sum = (unsigned char)sum_asm;

            if (is_prime(sum)) {
                SPDAT = 1;  // Prime
            } else {
                SPDAT = 2;  // Not prime
            }}}}
```

2.4 Prime Check

The function ‘is_prime()‘ checks if the sum is prime by testing divisibility.

BONUS

In this system, the Master and Slave communicate to perform a Multiply-Accumulate (MAC) operation, followed by a prime check on the result. The Master receives six numbers (three from arrays a and three from arrays b) and computes the sum of their pairwise products:

$$\text{Result} = a_1b_1 + a_2b_2 + a_3b_3$$

The result is then sent to the Slave, which checks whether the computed result is a prime number.

3 Master Code

The Master receives six numbers, a_1, a_2, a_3 and b_1, b_2, b_3, from the PC through UART, computes the MAC operation, and sends the result to the Slave. Below is the step-by-step explanation of the Master code.

3.1 UART Communication

The Master communicates with the PC via UART to receive six numbers. The function 'receive_num()' handles the reception of each number from the PC:

```
unsigned char receive_num(void)
{
    char buffer[4];
    unsigned char num = 0;
    char ch;
    while (1)
    {
        ch = receive_char();       // Receive character via UART
        transmit_char(ch);         // Echo back
        if (ch == '\r')            // Carriage return (Enter key)
        {
            buffer[i] = '\0';      // Null terminate
            break;
        }
        if (ch >= '0' && ch <= '9' && i < 3)
        {
            buffer[i++] = ch;
        }
    }
    for (i = 0; buffer[i] != '\0'; i++)
    {
        num = num * 10 + (buffer[i] - '0');
    }
    return num;  // Returns number between 0 and 255
}
```

This function reads each character, stores it, and converts it into an integer value for further processing.

3.2 MAC Operation

After receiving the six numbers, the Master sends them to the Slave via SPI. The MAC operation is performed by the Master using the following code:

```
SPCON = 0x7F;          // Configure SPI in master mode
spi_send(num1);        // Send num1 to slave
spi_send(num2);        // Send num2 to slave
// Repeat for all six numbers

// Wait for slave response
response = spi_send(0x00);
```

Here, the Master sends the numbers $a_1, a_2, a_3, b_1, b_2, b_3$ to the Slave. After sending the data, it waits for the Slave's response to indicate whether the result is prime or not.

4 Slave Code

The Slave receives the six numbers from the Master, performs the MAC operation, and checks whether the result is prime.

4.1 Receiving Data

The Slave receives each of the six numbers from the Master using SPI:

```
num1 = SPDAT;  // Read first number from SPI data register
num2 = SPDAT;  // Read second number from SPI data register
```

The Slave reads the values and stores them in memory for further processing.

4.2 MAC Operation

Once the six numbers are received, the Slave executes the MAC operation. The formula for the MAC operation is as follows:

$$\text{Result} = a_1b_1 + a_2b_2 + a_3b_3$$

The Slave calls the MAC function (explained below) to compute this result.

4.3 Prime Checking

After the MAC operation, the Slave checks if the result is a prime number using the 'is_prime()' function. If the result is prime, it sends a success signal to the Master, otherwise, it sends an error signal:

```
if (is_prime(result)) {
    SPDAT = 1;  // Success signal
} else {
    SPDAT = 0;  // Error signal
}
```

This function determines whether the result is prime and notifies the Master accordingly.

5 MAC Assembly Code

The MAC operation is implemented in assembly language. The assembly code multiplies corresponding elements from arrays a_1, a_2, a_3 and b_1, b_2, b_3, and accumulates the result.

```
PUBLIC MAC
CSEG AT 0100H

MAC:
    MOV R0, #70H        ; pointer to a1, a2, a3
    MOV R1, #73H        ; pointer to b1, b2, b3

    CLR A
    MOV 51H, A          ; clear MSB (high byte of accumulator)
    MOV 52H, A          ; clear LSB (low byte of accumulator)
    MOV 50H, A          ; temporary carry storage

    MOV R2, #03H        ; loop counter

MAC_LOOP:
    MOV A, @R0          ; A = a_i
    MOV B, @R1          ; B = b_i
    MUL AB              ; A*B => A(low), B(high)

    ; Add A (LSB of product) to 52H
    MOV C, 0            ; clear carry
    ADD A, 52H          ; add A (low byte) to accumulator
    MOV 52H, A          ; store result in 52H

    ; Add B (MSB of product) + carry to 51H
    MOV A, B            ; move MSB of product into A
    ADDC A, 51H         ; add with carry to accumulator
    MOV 51H, A          ; store result in 51H
```

```
    INC R0              ; increment pointer to next a_i
    INC R1              ; increment pointer to next b_i
    DJNZ R2, MAC_LOOP   ; decrement loop counter and repeat if not zero

    RET                 ; return from subroutine

END
```

5.1 Explanation of the MAC Assembly Code

1. **Pointers**:
   - `R0` is initialized to the memory address of the first a_i value.
   - `R1` is initialized to the memory address of the first b_i value.
   - These registers point to the memory locations where the a and b values are stored.
2. **Accumulator Setup**:
   - `51H` and `52H` are used to store the result of the MAC operation (high and low bytes of the accumulator).
   - `50H` is used to store the carry bit temporarily.
3. **MAC Operation Loop**:
   - The loop runs three times (since there are three elements in each array).
   - In each iteration:
     - The value of a_i is loaded into register `A`.
     - The value of b_i is loaded into register `B`.
     - The `MUL AB` instruction multiplies the two values, placing the low byte in `A` and the high byte in `B`.
     - The product is added to the accumulator (`52H` for the low byte, `51H` for the high byte) with carry.
4. **Loop Counter**:
   - `R2` is the loop counter, initialized to 3, as there are three pairs of numbers to multiply.
   - The loop continues until all pairs have been processed.
5. **Return**:
   - After completing the MAC operation, the function returns (`RET`).

Overall Flow

1. **Master**:
    - Receives six numbers from the user via UART.
    - Sends these numbers to the Slave via SPI for further processing.
2. **Slave**:
    - Receives the six numbers from the Master.
    - Performs the MAC operation to compute the result $a_1b_1 + a_2b_2 + a_3b_3$.
    - Checks if the result is prime and sends a success or error signal to the Master.
3. **Prime Checking**:
    - The Slave checks if the result is a prime number and communicates the result back to the Master.

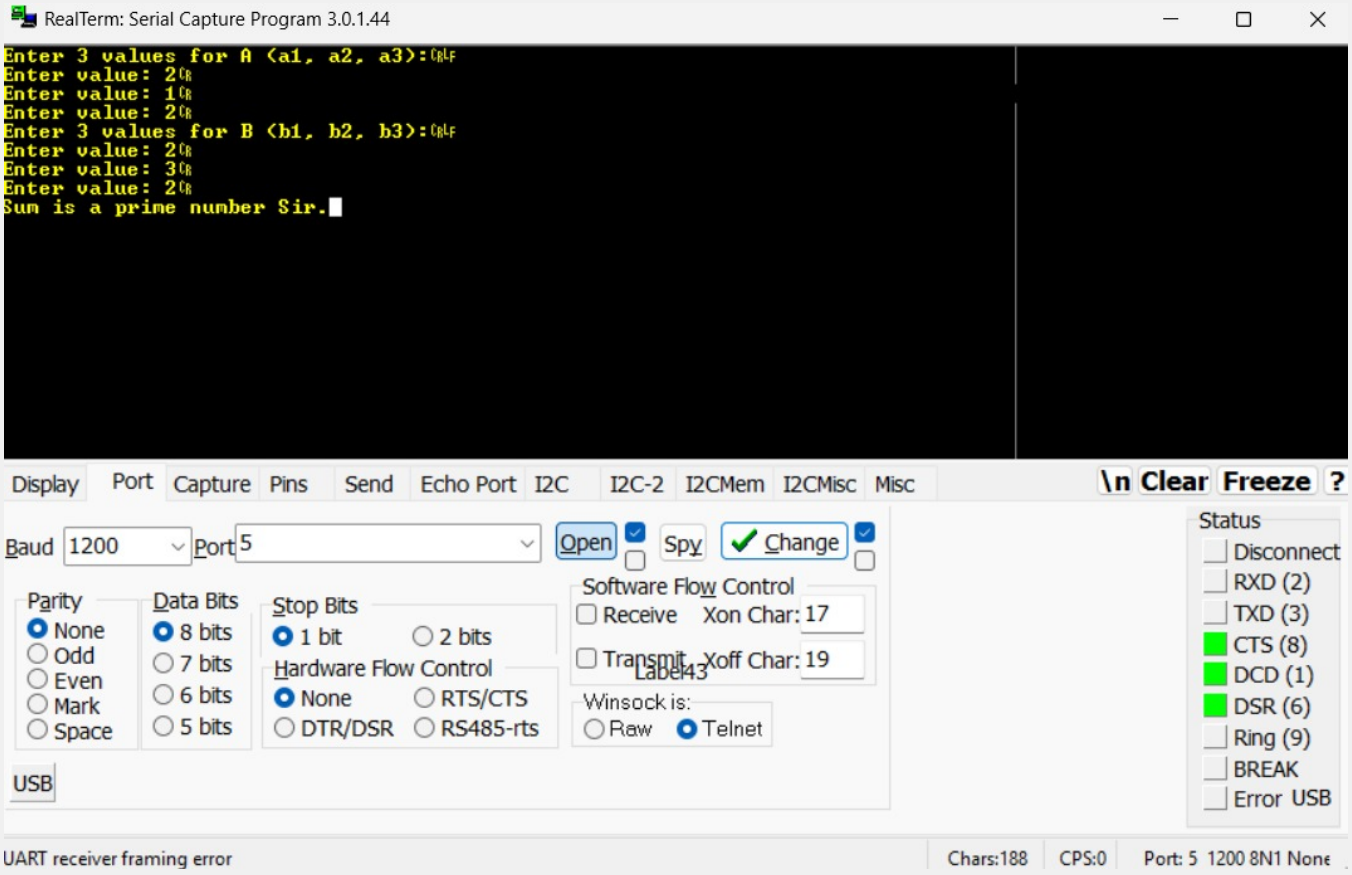

Figure 1: Realterm Output for MAC Operation

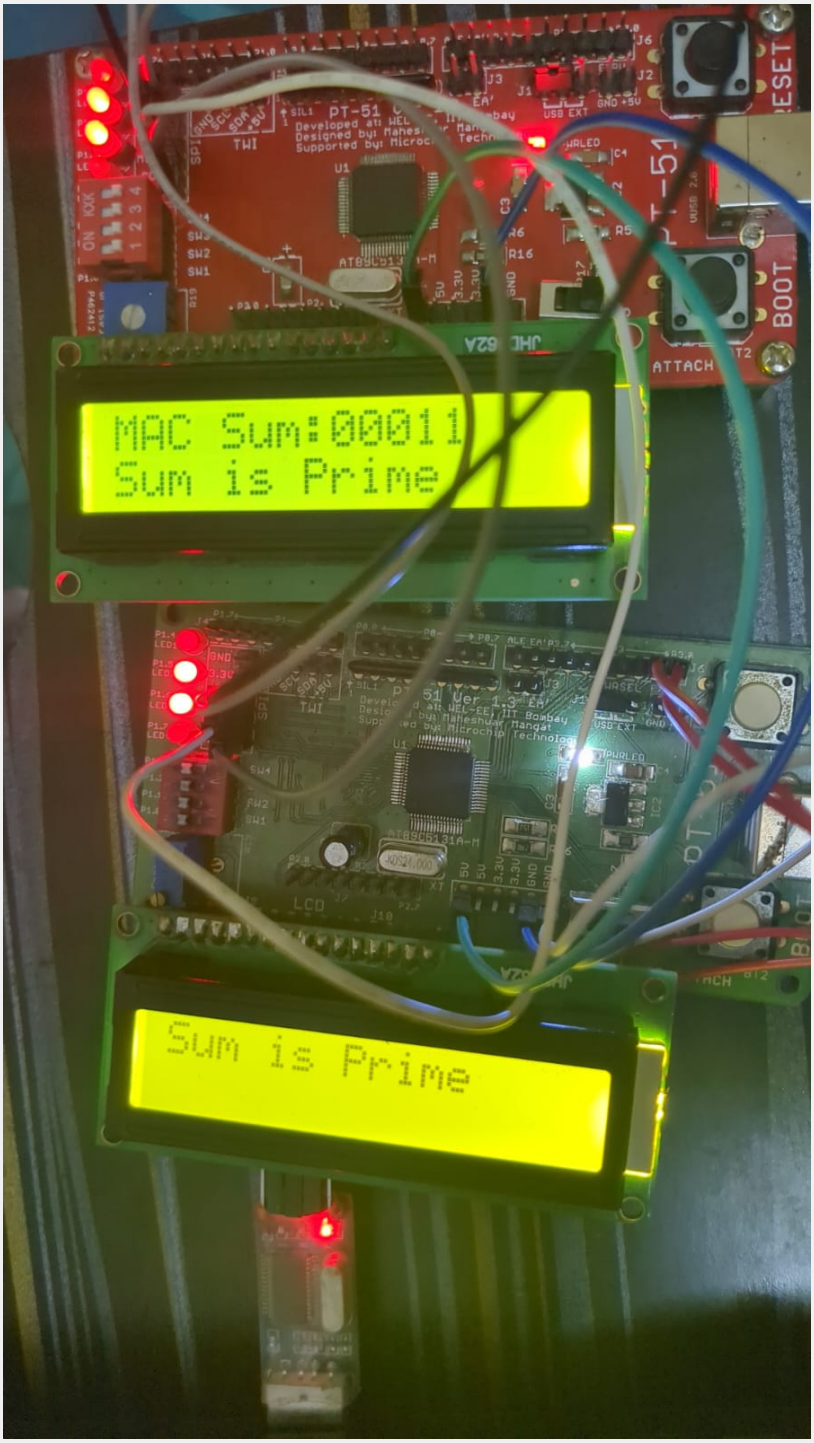

Figure 2: LCD Output for MAC Operation

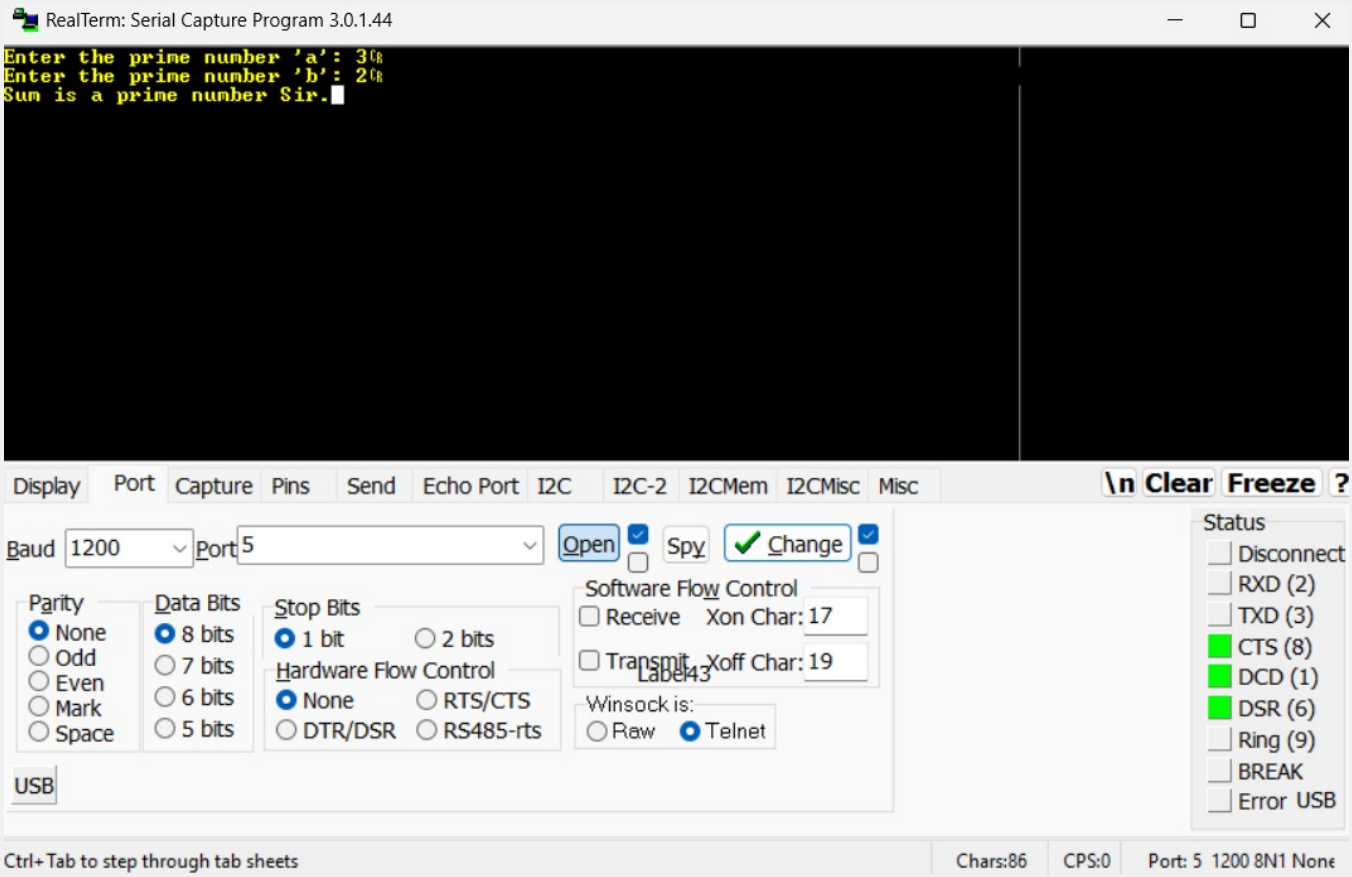

Figure 3: Realterm Output for Two Prime Numbers

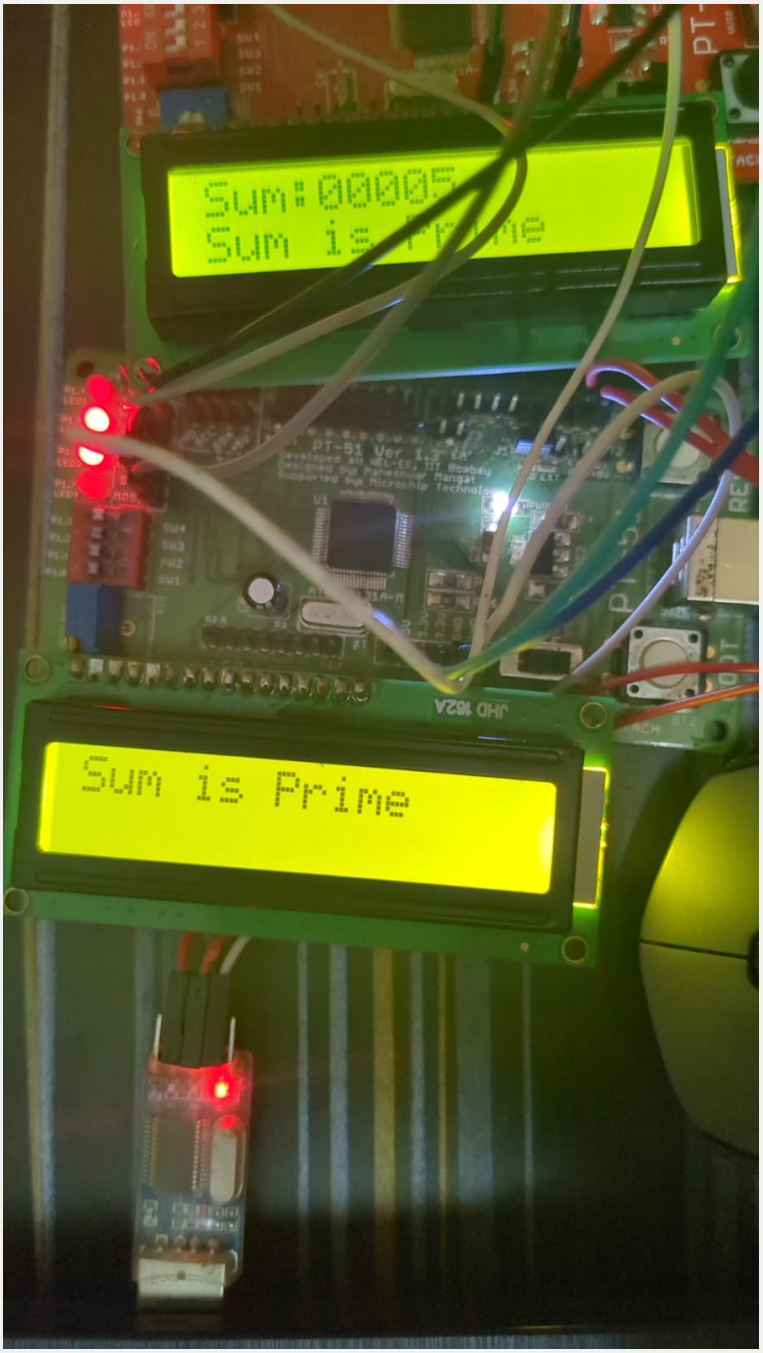

Figure 4: LCD Output for Two Prime Numbers

Thank You

23B1277