ISLAMIC UNIVERSITY OF TECHNOLOGY (IUT)
ORGANISATION OF ISLAMIC COOPERATION (OIC)
DEPARTMENT OF ELECTRICAL AND ELECTRONIC ENGINEERING

EEE 4706 (Microcontroller Based System Design Lab)

Project Title: Bluetooth interfacing with Microcontroller

Student(s) Names:

NAME	ID	Group No
Farhan Abrar Feehad Kamal Md. Wahiduzzaman Shafin Ibnul Mohasin Tahmid Rahman	200021210 200021218 200021228 200021244 200021246	B2_Group_04

Submission Date: **28/04/2025.**

Bluetooth interfacing with Microcontroller

Objectives

The primary objective of this project is to design and implement a robust Bluetooth-controlled embedded system based on the 8051-microcontroller architecture that can perform multiple communication and control functions through a seamless wireless interface.

The specific technical objectives of this project include:

1) Implementation of reliable serial communication between a mobile application and the 8051-microcontroller via Bluetooth.
2) Development of a multi-modal operational framework that allows users to seamlessly transition between different functional modes.
3) Design of a comprehensive LED control system.
4) Integration of relay control capabilities.
5) Creation of a Morse code translation.
6) Development of a basic encryption/decryption system.
7) Design of an interactive Morse code decoder.
8) Implementation of a basic arithmetic calculator capable of performing fundamental mathematical operations on multi-digit numbers.

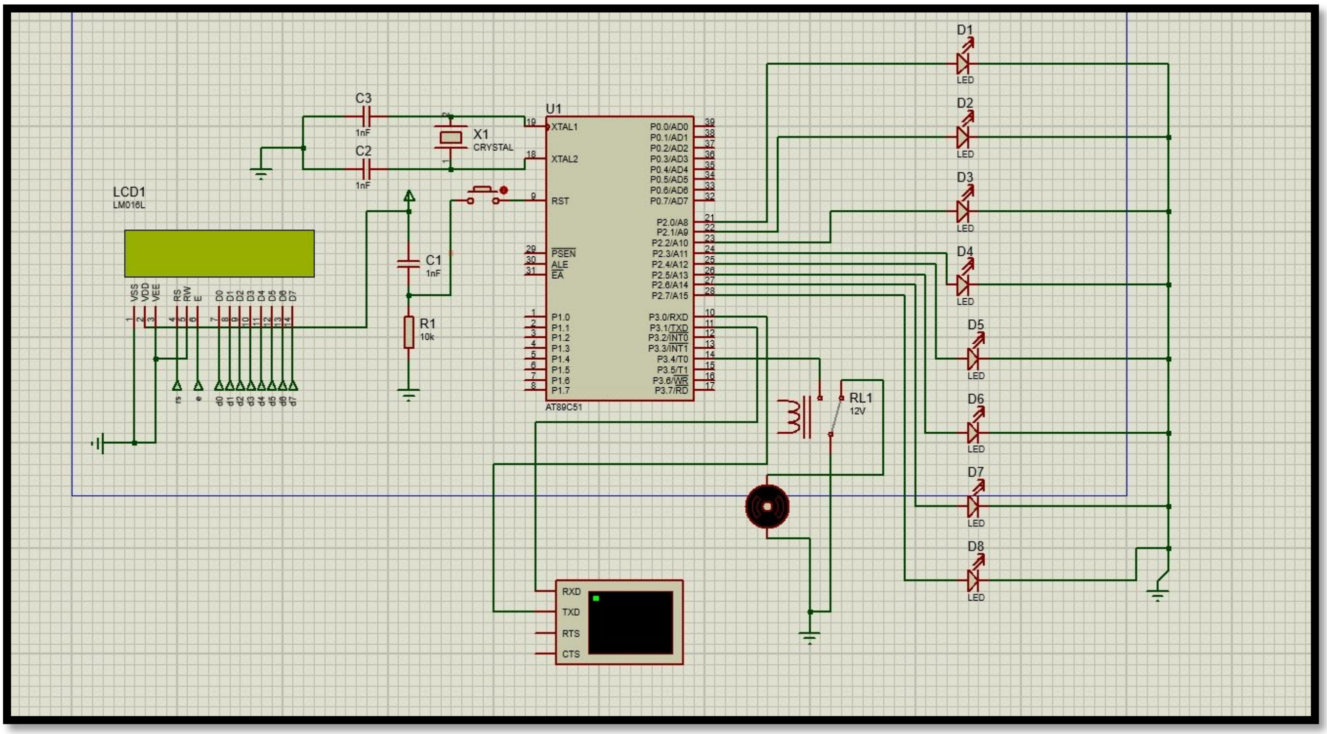
Required Components

This section provides a detailed inventory of all electronic components, modules, and accessories required for the successful implementation of the Bluetooth-interfaced 8051 microcontroller system.

Component	**Quantity**	**Price (BDT)**
8051 Microcontroller Board	1	1000
16x2 LCD Display	1	*N/A*
HC-05 Bluetooth Module	1	350
LEDs	8	*N/A*
Relay	1	70
10kΩ Resistors	8	*N/A*
33pF Capacitors	2	*N/A*
Jumper Wires	30	50
Power Supply (5V)	1	*N/A*
Total		**1,470**

Circuit Diagram

The following circuit was built using Proteus simulation software:



❖ Circuit Design and Architecture

The electronic circuit for this project represents an integrated system architecture combining digital processing, display technology, user interface elements, and communication modules. The schematic design follows engineering best practices for embedded systems, with careful consideration given to signal integrity, power distribution, and electromagnetic compatibility.

❖ Microcontroller and Core Circuitry

The central element of the circuit is the AT89S52 microcontroller, configured with the following essential connections:

1. **Clock Generation Network:**
   - The 11.0592 MHz crystal oscillator is connected between pins XTAL1 (pin 19) and XTAL2 (pin 18) of the microcontroller.
   - Two 33pF ceramic capacitors are connected from each crystal terminal to ground, forming a Pierce oscillator configuration that ensures stable clock generation.

- This specific crystal frequency was selected to minimize timing errors in serial communication, as it enables the generation of standard baud rates with minimal fractional error when used with the 8051's Timer 1.

2. **Reset Circuit:**
    - A 10kΩ pull-up resistor is connected to the RST pin (pin 9) to maintain a high logic level during normal operation.
    - A capacitor-switch combination enables manual reset functionality when needed for development or troubleshooting.

3. **Power Supply Network:**
    - The VCC (pin 40) is connected to the regulated 5V supply.
    - The GND (pin 20) is connected to the circuit common ground.
    - A 10μF electrolytic capacitor is placed near the power input of the microcontroller to filter power supply transients and reduce noise.

❖ **Serial Communication Interface**

1. **HC-05 Bluetooth Module Connections:**
    - The TXD pin of the HC-05 module is connected to the RXD (P3.0, pin 10) of the microcontroller.
    - The RXD pin of the HC-05 module is connected to the TXD (P3.1, pin 11) of the microcontroller.
    - The module's VCC and GND are connected to the system's 5V and ground lines respectively.
    - The module's state pin is connected to an LED with a current-limiting resistor to provide visual indication of the Bluetooth connection status.

2. **Logic Level Considerations:**
    - The HC-05 module operates at 3.3V logic levels, while the AT89S52 operates at 5V logic levels.
    - A voltage divider using resistors is implemented on the TXD line from the microcontroller to the Bluetooth module to ensure compatible signal levels.

- ❖ **Display Interface**

The 16x2 LCD display is integrated using an 8-bit parallel interface configuration:

1. **Data Bus Connections:**
    - The eight data lines (DB0-DB7) of the LCD are connected to Port 0 (P0.0-P0.7) of the microcontroller, enabling parallel data transfer.
    - Pull-up resistors (1kΩ) are included on the data lines due to the open-drain nature of Port 0.
2. **Control Signal Connections:**
    - The Register Select (RS) signal is connected to P2.0 to distinguish between command and data operations.
    - The Read/Write (RW) signal is connected to P2.1 to select between read and write operations.
    - The Enable (E) signal is connected to P2.2 to synchronize data transfers.
    - The contrast adjustment is implemented using a potentiometer connected to the contrast pin (V0) of the LCD.
    - The LCD's backlight is powered through a current-limiting resistor to maintain appropriate brightness levels.

- ❖ **Output Interface Circuitry**

1. **LED Array Configuration:**
    - Eight LEDs are connected to Port 1 (P1.0-P1.7) through current-limiting resistors (220Ω) to protect both the LEDs and the microcontroller output pins.
    - The common cathode configuration is utilized, with the LEDs' cathodes connected to ground and anodes connected to the port pins through the current-limiting resistors.
    - The calculated current through each LED is approximately (5V - 2V) / 220Ω = 13.6mA, which is within the safe operating range for both the LEDs and the microcontroller output pins.

2. **Relay Control Circuit:**
    - The relay module is connected to P2.5 of the microcontroller through a buffer transistor.
    - A protection diode is included in parallel with the relay coil to suppress inductive voltage spikes during switching operations.
    - The relay's normally open (NO) and common (COM) contacts are made available for controlling external devices.

Features

❖ **Mandatory Features**

Serial Communication Setup

- **Baud Rate Configuration:** The serial communication is established at 9600 baud, a standard rate that balances data throughput with reliability. This is achieved by configuring Timer 1 in Mode 2 (8-bit auto-reload) with TH1 value of 0xFD. The mathematical derivation for this value is: TH1 = 256 - [(Crystal Frequency) / (384 × Desired Baud Rate)] = 256 - [(11,059,200) / (384 × 9600)] = 256 - 30.06 ≈ 256 - 30 = 226 (0xE2) However, when SMOD=1, the formula becomes: TH1 = 256 - [(Crystal Frequency) / (192 × Desired Baud Rate)] = 256 - [(11,059,200) / (192 × 9600)] = 256 - 60.12 ≈ 256 - 60 = 196 (0xC4) The value 0xFD (253) used in the code corresponds to a different calculation approach or possibly accounts for additional system-specific factors affecting timing.
- **SCON Register Configuration:** The Serial Control (SCON) register is configured with the value 0x50 (01010000 in binary), which specifies:
    - SM0 = 0, SM1 = 1: Mode 1 operation (8-bit UART, variable baud rate)
    - SM2 = 0: Multiprocessor communication disabled
    - REN = 1: Serial reception enabled
    - TB8, RB8, TI, RI = 0: Initial clearing of transmission/reception flags
- **Bluetooth Module Integration:** The HC-05 Bluetooth module is configured as a slave device, accepting connections from smartphone applications. The module's default settings (device name, passkey) can be modified through AT commands for customization and security enhancement.

- ❖ **Multi-Mode Operation System**

   - **Mode Selection Interface:** Upon initialization, the system displays a "ENTER MODE:" prompt on the LCD and waits for user input through the Bluetooth connection. Numeric inputs (1-5) correspond to different operational modes.

   - **Modal Architecture:** The software architecture follows a modal design pattern where each mode represents an isolated functional domain with its own input processing, output generation, and state management. This approach enhances code maintainability and logical separation of concerns.

   - **Navigation Framework:** All modes implement a consistent navigation pattern where the input '0' returns the user to the mode selection menu, creating an intuitive and uniform user experience across different functionalities.

- ❖ **LED Control (Mode 1)**

   - **Individual LED Addressing:** Each of the eight LEDs connected to Port 1 (P1.0-P1.7) can be individually controlled through numeric commands (1-8) sent via Bluetooth.

   - **Toggle Operation:** Rather than simple on/off control, the system implements a toggle functionality where each command inverts the current state of the corresponding LED, allowing users to both activate and deactivate LEDs with the same command.

   - **Real-time Response:** The system provides immediate visual feedback by reflecting LED state changes without noticeable delay after receiving commands, demonstrating responsive input processing.

   - **Multiple LED Patterns:** By sequentially toggling different LEDs, users can create various patterns or configurations to visualize binary data or create signaling sequences.

- ❖ **Relay Control (Mode 2)**

   - **Safe Switching Mechanism:** The relay connected to P2.5 is controlled through a proper driver circuit that ensures electrical isolation between the microcontroller and the relay coil.

   - **State Toggling:** Similar to the LED control mode, the relay implements a toggle operation where commands alternate the relay between energized and de-energized states.

   - **Debounce Protection:** The implementation includes timing considerations to prevent rapid state changes that could damage the relay or connected devices, ensuring a minimum duration between switching operations.

   - **External Device Control:** The relay's normally open (NO) and normally closed (NC) contacts are made available for controlling external AC or DC devices, significantly expanding the system's application possibilities.

Individual Features

- ❖ **Morse Code Display (Mode 3)**

  - **Comprehensive Character Support:** The system contains pre-defined Morse code patterns for all 26 English alphabet letters (A-Z), stored efficiently in program memory using null-terminated strings.
  - **Pattern Lookup System:** When a character is received via Bluetooth, the program performs a sequential search through conditional branches to locate the corresponding Morse code pattern for the input character.
  - **Visual Representation:** The identified Morse code pattern is displayed on the LCD, showing dots (.) and dashes (-) corresponding to the international Morse code standard.

- ❖ **Simple Encryption (Mode 4)**

  - **Caesar Cipher Implementation:** The system employs a variation of the classical Caesar cipher with a fixed shift of +1, where each input character is replaced by the character that follows it in the alphabet (e.g., 'A' becomes 'B', 'B' becomes 'C').
  - **Alphabet Wrapping:** Special handling is implemented for the letter 'Z', which wraps around to 'A' when encrypted, maintaining the continuity of the encryption algorithm across the entire alphabet.

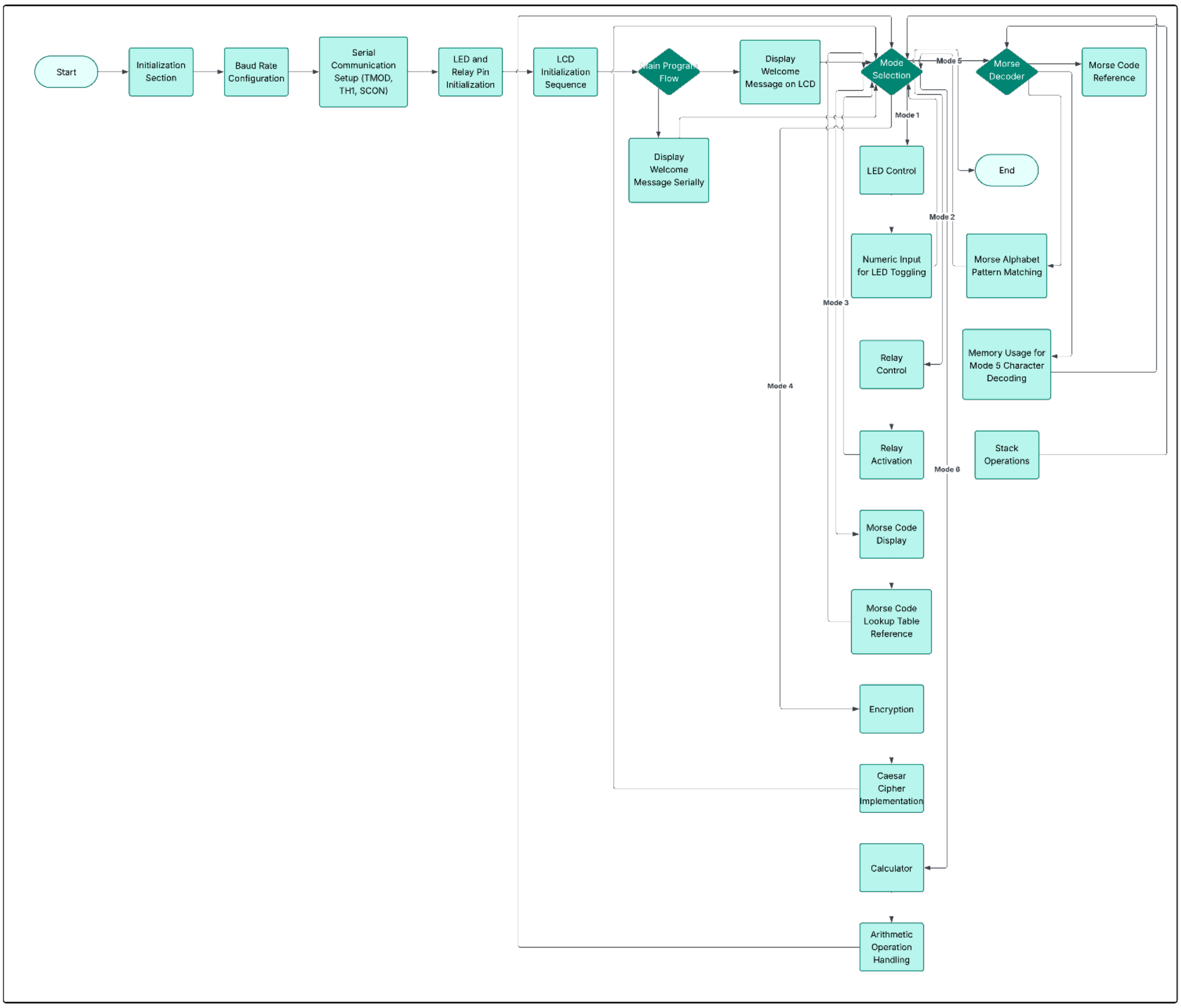
- ❖ **Morse Code Decoder with LED Visualization (Mode 5)**

  - **Interactive Morse Input:** Users can input Morse code patterns using dots (.) and dashes (-) via Bluetooth, with the system storing up to four symbols per character in registers R4-R7.
  - **Pattern Recognition Algorithm:** The system employs an extensive pattern matching algorithm that compares the input pattern against predefined Morse code sequences to identify the corresponding alphabetic character.
  - **Visual Pattern Display:** As Morse code patterns are entered, they are displayed on the LCD for visual confirmation of the input sequence.
  - **LED Visualization:** The system provides a synchronized LED display where dots are represented by short LED pulses (2 delay cycles) and dashes by longer pulses (6 delay cycles), maintaining the standard 1:3 timing ratio of international Morse code.
  - **Multi-character Message Support:** The decoded characters are accumulated and displayed on the second line of the LCD, allowing users to build complete messages. The forward slash character (/) triggers the display of the accumulated message.

- **Basic Calculator (Mode 6)**
   - **Multi-digit Number Input:** The system accepts two-digit numbers (00-99) as operands, demonstrating parsing and conversion of ASCII input to binary values for computation.
   - **Operation Selection:** Users can select between addition (+), subtraction (-), and multiplication (*) operations, providing basic calculator functionality.
   - **Result Computation and Display:** The system performs the selected arithmetic operation on the input operands and displays the result on the LCD in a formatted manner.
   - **Decimal Result Display:** The calculation results are displayed with proper decimal formatting, showing hundreds, tens, and units digits appropriately.

Working Principle

- **Flow Chart of the Algorithm:**



The flow chart illustrates the sequential and conditional execution paths of the system, highlighting:

- Initialization procedures
- Main loop structure
- Mode selection and switching logic
- Input processing mechanisms
- Output generation pathways
- Return paths to the main menu

Step-by-Step Operational Analysis

❖ **System Initialization Phase**

1. **Power-On Initialization:**
    - Upon power application, the system begins execution at the reset vector (0000H).
    - The stack pointer is initialized to a predetermined location in RAM (70H), providing sufficient space for subroutine calls and temporary variable storage.
    - The Program Status Word (PSW) is cleared to ensure a known initial state for all flags and register bank selection.
2. **Serial Communication Configuration:**
    - Timer 1 is configured in Mode 2 (8-bit auto-reload) to generate accurate baud rate timing.
    - The TH1 register is loaded with 0xFD to produce a 9600 baud rate with the 11.0592 MHz crystal.
    - The SCON register is configured for Mode 1 operation (8-bit UART) with reception enabled.
    - Timer 1 is started to begin baud rate generation.

3. **I/O Port Configuration:**
    - Port 1 pins are configured as outputs for LED control, with initial states set to OFF (logic 0).
    - Port 2 pins are appropriately configured for LCD control signals (RS, RW, ENABLE) and relay control.
    - The relay is initialized to its default state (ON in this implementation).
4. **LCD Initialization:**
    - The LCD is initialized using a standard sequence of commands: a. Function Set (38H): Configures for 2-line display with 5x7 character matrix b. Display ON/Cursor ON (0EH): Activates display and cursor visibility c. Entry Mode Set (06H): Configures for cursor movement direction and display shift d. Clear Display (01H): Clears any residual content and homes the cursor
    - A delay is inserted between commands to ensure the LCD controller has sufficient time to process each instruction.
5. **Welcome Message Display:**
    - The welcome message "B2 Group- 4" is retrieved from program memory.
    - Each character is sequentially sent to both the Bluetooth module (via SBUF) and the LCD display.
    - This dual output confirms proper operation of both communication channels.

❖ **Mode Selection Procedure**

After initialization, the system enters a menu-driven operational structure:

1. **Mode Prompt Display:**
    - The LCD is cleared to prepare for new content.
    - The message "ENTER MODE:" is retrieved from program memory and displayed on the LCD.
    - The system then waits for user input through the Bluetooth interface.

2. **Input Acquisition and Processing:**
    - The GET_IP subroutine monitors the RI (Receive Interrupt) flag to detect incoming data.
    - When data is received, it is retrieved from the SBUF register for processing.
    - The received value is compared against predefined mode numbers (1-5) using a sequence of conditional branches.
3. **Mode Transition:**
    - Based on the received command, program execution jumps to the corresponding mode subroutine.
    - Each mode begins with appropriate initialization steps, including clearing the LCD display.
    - If an invalid mode number is received, the system remains in the mode selection state, implicitly ignoring the command.

❖ **LED Control Mode (Mode 1) Operation**

1. **Mode Initialization:**
    - The LCD display is cleared to remove previous content.
    - The system prepares to receive LED control commands.
2. **Command Processing:**
    - Each received character is compared against numbers '1' through '8' to identify the targeted LED.
    - When a match is found, the corresponding LED state is toggled using the CPL (Complement) instruction on the appropriate bit address.
    - The system immediately returns to waiting for the next command, providing a responsive user interface.
3. **Mode Exit Detection:**
    - Each received character is also checked against '0' to detect an exit command.
    - When '0' is received, execution jumps back to the SELECT_MODE routine, returning to the main menu.

- **Relay Control Mode (Mode 2) Operation**

1. **Relay Toggle Operation:**
   - The relay state is toggled between active and inactive states.
   - Appropriate delays are inserted to prevent rapid switching that could damage the relay.
2. **Command Monitoring:**
   - The system continuously monitors for the '0' command to return to the mode selection menu.
   - All other inputs are effectively ignored in this simple control mode.

- **Morse Code Display Mode (Mode 3) Operation**

1. **Character Input Processing:**
   - The system waits for character input via Bluetooth using the GET_IP subroutine.
   - The received character is compared against each letter of the alphabet using a comprehensive sequence of conditional branches.
2. **Pattern Lookup and Display:**
   - When a match is found, the data pointer (DPTR) is loaded with the address of the corresponding Morse code pattern string in program memory.
   - The DISPLAY_MORSE routine is then called to retrieve and display each character of the pattern.
   - Each dot (.) and dash (-) in the pattern is displayed sequentially on the LCD, providing a visual representation of the Morse code.
3. **Return to Character Input:**
   - After displaying the complete pattern, the system returns to waiting for the next character input.
   - This creates an interactive loop where users can explore Morse code representations of different letters.
4. **Mode Exit Checking:**
   - The system continuously checks for the '0' command to exit back to the mode selection menu.

❖ **Encryption Mode (Mode 4) Operation**

1. **Input Character Processing:**
   - Characters received via Bluetooth are processed one at a time.
   - Each character is displayed on the LCD to show the original input.
2. **Encryption Algorithm Application:**
   - The standard encryption rule adds 1 to the ASCII value of the input character, effectively shifting one letter forward in the alphabet.
   - Special handling is implemented for 'Z', which wraps around to 'A' when encrypted.
3. **Encrypted Output Display:**
   - The encrypted character is displayed on the LCD immediately after the original character.
   - This provides a side-by-side view of the original and encrypted characters.
4. **Continuous Operation:**
   - The system continues processing characters in real-time as they are received.
   - This allows users to input entire messages for encryption.
5. **Exit Mechanism:**
   - The '0' character serves as the exit command, returning to the mode selection menu.

❖ **Morse Code Decoder Mode (Mode 5) Operation**

1. **Register Initialization:**
   - Registers R4, R5, R6, and R7 are cleared to prepare for storing Morse code symbols.
   - Register R3 is set to 1 to serve as a position counter for input symbols.
2. **Symbol Input and Storage:**
   - Dots (.) and dashes (-) are received via Bluetooth and displayed on the LCD.
   - Each symbol is stored in the appropriate register based on the current value of the position counter (R3).
   - The position counter is incremented after each symbol input.

3. **Pattern Recognition:**
    - When a space character is received, it signals the end of the current Morse code pattern.
    - The system then executes an extensive pattern matching algorithm that compares the stored symbols against predefined Morse code patterns for each letter.
4. **Letter Identification and Display:**
    - When a match is found, the corresponding alphabetic character is identified.
    - The MORSE_LED routine is called to visually represent the Morse pattern using LED pulses.
    - The identified letter is stored in memory at the location pointed to by register R0, building up a decoded message.
5. **Message Display:**
    - When a forward slash (/) character is received, the system displays the accumulated decoded message on the second line of the LCD.
    - This allows users to see the complete translation of multiple Morse code patterns.
6. **System Reset and Exit:**
    - The system monitors for the '0' command to return to the mode selection menu.
    - Upon receiving this command, all registers and display areas are cleared before returning.

❖ **Calculator Mode (Mode 6) Operation**

1. **Operand Input:**
    - The system accepts two-digit numbers as operands.
    - Each digit is received, displayed, and processed to build the complete operand value.
    - The first two digits form the first operand (stored in R5).
2. **Operator Input:**
    - An arithmetic operator (+, -, or *) is accepted and stored in R6.

3. **Second Operand Input:**

    - The second two-digit operand is received and processed similarly to the first, storing the result in R7.

4. **Operation Execution:**

    - Based on the operator stored in R6, the appropriate arithmetic operation is performed on the operands.
    - Addition: When R6 contains '+', the values in R5 and R7 are added using the ADD instruction. The addition is performed with consideration for potential carry generation.
    - Subtraction: When R6 contains '-', R7 is subtracted from R5 using the SUBB instruction. The operation properly accounts for borrow propagation through conditional branching.
    - Multiplication: When R6 contains '*', the MUL AB instruction is employed with appropriate register loading to produce the product of R5 and R7.

The result of the calculation is stored in registers R4 (high byte) and R5 (low byte) for subsequent display.

❖ **Result Formatting and Display:**

- The numerical result is converted from binary to decimal format using a binary-to-BCD conversion algorithm.
- The conversion process involves repeated division by 10 and extraction of remainders to identify individual decimal digits.
- Each decimal digit is converted to its ASCII representation by adding 30H before transmission to the LCD.
- The hundreds, tens, and units positions are displayed in sequence, with special handling for leading zeros.

❖ **Result Verification:**

- The system performs boundary condition checks to ensure result validity:
    - For addition and multiplication operations, potential overflow conditions are detected and flagged.
    - For subtraction operations, negative results are handled by displaying a negative sign and the absolute value.

- ❖ **Continuous Operation Support:**
  - After displaying the result, the system returns to the input state, allowing users to perform additional calculations without returning to the mode selection menu.
  - This sequential operation supports complex multi-step calculations where previous results can inform subsequent operations.
- ❖ **Exit Mechanism:**
  - The '0' command is continuously monitored during all input phases.
  - When detected, the system clears all calculation registers and returns to the mode selection menu.

CODE

```
ORG 00H  ; Origin address - program starts at memory location 00H

;==================================================================================
; SERIAL COMMUNICATION INITIALIZATION
;==================================================================================
MOV TMOD,#20H  ; Timer 1, mode 2 (8-bit auto reload timer)
MOV TH1,#0FDH  ; Set baud rate to 9600
MOV SCON,#50H  ; Configure serial port: 8 data bits, 1 stop bit, 1 start bit, REN enabled
CLR T1         ; Clear Timer 1 register
SETB TR1       ; Start Timer 1 for baud rate generation

;==================================================================================
; PIN CONFIGURATION - Define bit addresses for LEDs and control pins
;==================================================================================
LED_1     BIT P1.0  ; LEDs connected to Port 1
LED_2     BIT P1.1
LED_3     BIT P1.2
LED_4     BIT P1.3
LED_5     BIT P1.4
LED_6     BIT P1.5
LED_7     BIT P1.6
LED_8     BIT P1.7
LED_RELAY BIT P2.5  ; Relay control pin on Port 2.5

; Initialize all LEDs to OFF
CLR  LED_1
CLR  LED_2
CLR  LED_3
CLR  LED_4
CLR  LED_5
CLR  LED_6
CLR  LED_7
CLR  LED_8
SETB LED_RELAY  ; Turn relay ON initially

; LCD control pins
RS   EQU P2.0  ; Register Select: 0=Command, 1=Data
RW   EQU P2.1  ; Read/Write: 0=Write, 1=Read
ENBL EQU P2.2  ; Enable signal

;==================================================================================
; INITIAL MESSAGE DISPLAY - Send welcome message to smartphone via serial port
;==================================================================================
MOV DPTR, #MYDATA  ; Load pointer to welcome message string
GO:
    CLR A           ; Clear accumulator
    MOVC A,@A+DPTR  ; Get character from code memory
    JZ LCD          ; If byte is zero (end of string), go to LCD routine
    ACALL SEND      ; Otherwise send the byte
    SJMP GO         ; Continue with next character

; Send byte in A register via serial port
SEND:
```

```
    MOV SBUF, A       ; Load byte to serial buffer for transmission
    INC DPTR          ; Point to next byte
    HERE: JNB TI, HERE  ; Wait for transmission to complete
    CLR TI            ; Clear transmit interrupt flag
    RET

;==================================================================================
; LCD INITIALIZATION ROUTINE
;==================================================================================
LCD:
    MOV SP,#70H       ; Set stack pointer
    MOV PSW,#00H      ; Clear program status word

    ; Initialize LCD
    MOV A,#38H        ; LCD 2 lines, 5x7 character matrix
    LCALL COMMAND     ; Send command to LCD
    LCALL DELAY       ; Wait for LCD to process

    MOV A,#0EH        ; Display on, cursor on
    LCALL COMMAND
    LCALL DELAY

WELCOME:
    MOV 32H, #16      ; Set character counter for display positioning
    LCALL CLEAR_LCD ; Clear LCD display

    MOV A,#06H        ; Set entry mode: cursor moves right
    LCALL COMMAND
    LCALL DELAY

    MOV A,#80H        ; Set cursor to beginning of first line
    LCALL COMMAND
    LCALL DELAY

    ; Display welcome message on LCD
    LCALL DELAY
    MOV DPTR, #MYDATA       ; Load pointer to message
    LOOP_1: CLR A
    MOVC A,@A+DPTR          ; Get character
    JZ FINISH               ; If zero (end of string), exit
    LCALL DISPLAY           ; Display character on LCD
    LCALL DELAY
    INC DPTR                ; Next character
    LJMP LOOP_1

FINISH:
    LJMP SELECT_MODE        ; Jump to mode selection

;==================================================================================
; MORSE CODE LOOKUP TABLES - Define morse code patterns for each letter
;==================================================================================
MSG_A: DB '-.', 0     ; Morse code for A (dot-dash)
MSG_B: DB '-...', 0   ; Morse code for B
MSG_C: DB '-.-.', 0   ; Morse code for C
MSG_D: DB '-..', 0    ; Morse code for D
MSG_E: DB '.', 0      ; Morse code for E
MSG_F: DB '..-.', 0   ; Morse code for F
MSG_G: DB '--.', 0    ; Morse code for G
MSG_H: DB '....', 0   ; Morse code for H
MSG_I: DB '..', 0     ; Morse code for I
MSG_J: DB '.---', 0   ; Morse code for J
MSG_K: DB '-.-', 0    ; Morse code for K
MSG_L: DB '.-..', 0   ; Morse code for L
MSG_M: DB '--', 0     ; Morse code for M
MSG_N: DB '-.', 0     ; Morse code for N
MSG_O: DB '---', 0    ; Morse code for O
MSG_P: DB '.--.', 0   ; Morse code for P
MSG_Q: DB '--.-', 0   ; Morse code for Q
MSG_R: DB '.-.', 0    ; Morse code for R
MSG_S: DB '...', 0    ; Morse code for S
MSG_T: DB '-', 0      ; Morse code for T
MSG_U: DB '..-', 0    ; Morse code for U
MSG_V: DB '...-', 0   ; Morse code for V
MSG_W: DB '.--', 0    ; Morse code for W
MSG_X: DB '-..', '-', 0  ; Morse code for X
MSG_Y: DB '-.--', 0   ; Morse code for Y
MSG_Z: DB '--..', 0   ; Morse code for Z

; Message strings
MYDATA: DB 'B2 Group- 4',0  ; Welcome message displayed on startup
MSG1: DB 'ENTER MODE:',0    ; Mode selection prompt

;==================================================================================
; MODE SELECTION ROUTINE - Read user input to select operating mode
;==================================================================================
SELECT_MODE:
```

```
    MOV 32H, #16        ; Reset line position counter
    LCALL CLEAR_LCD     ; Clear display
    MOV DPTR, #MSG1     ; Display "ENTER MODE:" prompt
    LOOP_A: CLR A
    MOVC A,@A+DPTR
    JZ FINISH1
    LCALL DISPLAY
    LCALL DELAY
    INC DPTR
    LJMP LOOP_A

FINISH1:
    ACALL GET_IP        ; Get input from serial port (user selection)
    CJNE A, #'1',NEXT1 ; Compare with '1'
    LJMP MODE1          ; Jump to selected mode
    NEXT1: CJNE A, #'2',NEXT2
    LJMP MODE2
    NEXT2: CJNE A, #'3',NEXT3
    LJMP MODE3
    NEXT3: CJNE A, #'4',NEXT4
    LJMP MODE4
    NEXT4: CJNE A, #'5',NEXT4
    LJMP MODE5

;=====================================================================================
; MODE 1 - LED CONTROL: Toggle individual LEDs based on numeric input
;=====================================================================================
MODE1:
    MOV 32H, #16
    LCALL CLEAR_LCD

INPUT_NEW:
    LCALL GET_IP            ; Get key press from smartphone
    CJNE A,#'1', CHECK1     ; Check if key '1' was pressed
    CPL LED_1               ; Toggle LED 1 (complement current state)
    LJMP INPUT_NEW          ; Continue accepting input

; Similar checks for keys 2-8 to toggle corresponding LEDs
CHECK1: CJNE A, #'2', CHECK2
CPL LED_2
LJMP INPUT_NEW

CHECK2: CJNE A, '3', CHECK3
CPL LED_3
LJMP INPUT_NEW

CHECK3: CJNE A, #'4', CHECK4
CPL LED_4
LJMP INPUT_NEW

CHECK4: CJNE A, #'5', CHECK5
CPL LED_5
LJMP INPUT_NEW

CHECK5: CJNE A, #'6', CHECK6
CPL LED_6
LJMP INPUT_NEW

CHECK6: CJNE A, #'7', CHECK7
CPL LED_7
LJMP INPUT_NEW

CHECK7: CJNE A, #'8', CHECK8
CPL LED_8
LJMP INPUT_NEW

CHECK8: CJNE A, #'0', INPUT_NEW  ; If '0' pressed, return to mode selection
LJMP SELECT_MODE

;=====================================================================================
; MODE 2 - RELAY CONTROL: Toggle relay state
;=====================================================================================
MODE2:
    MOV 32H, #16
    LCALL CLEAR_LCD
    ACALL GET_IP        ; Get user input

    clr p2.5            ; Turn relay OFF
    acall delay

    setb p2.5           ; Turn relay back ON
    doo: cjne a, #'0', doo  ; Wait for '0' to exit

END_MODE2: LJMP SELECT_MODE

;=====================================================================================
```

```
; MODE 3 - MORSE CODE DISPLAY: Convert letters to morse code display
;==================================================================================
MODE3:
    MOV 32H, #16
    LCALL CLEAR_LCD

    ; Series of checks to determine which letter was input
    ; and jump to display corresponding morse code
CHECK_A:
    ACALL GET_IP            ; Get character input
    CJNE A, #'A', CHECK_B   ; Check if 'A' was input
    MOV DPTR, #MSG_A        ; Point to morse code for 'A'
    LJMP DISPLAY_MORSE      ; Display the morse code

CHECK_B:
    MOV A, SBUF             ; Get input character
    CJNE A, #'B', CHECK_C   ; Compare with 'B'
    MOV DPTR, #MSG_B        ; Point to morse code for 'B'
    LJMP DISPLAY_MORSE

    ; Similar checks for C through Z
    ; [code continues for each letter]

CHECK_C:
    MOV A, SBUF
    CJNE A, #'C', CHECK_D
    MOV DPTR, #MSG_C
    LJMP DISPLAY_MORSE

CHECK_D:
    MOV A, SBUF
    CJNE A, #'D', CHECK_E
    MOV DPTR, #MSG_D
    LJMP DISPLAY_MORSE

CHECK_E:
    MOV A, SBUF
    CJNE A, #'E', CHECK_F
    MOV DPTR, #MSG_E
    LJMP DISPLAY_MORSE

CHECK_F:
    MOV A, SBUF
    CJNE A, #'F', CHECK_G
    MOV DPTR, #MSG_F
    LJMP DISPLAY_MORSE

CHECK_G:
    MOV A, SBUF
    CJNE A, #'G', CHECK_H
    MOV DPTR, #MSG_G
    LJMP DISPLAY_MORSE

CHECK_H:
    MOV A, SBUF
    CJNE A, #'H', CHECK_I
    MOV DPTR, #MSG_H
    LJMP DISPLAY_MORSE

CHECK_I:
    MOV A, SBUF
    CJNE A, #'I', CHECK_J
    MOV DPTR, #MSG_I
    LJMP DISPLAY_MORSE

CHECK_J:
    MOV A, SBUF
    CJNE A, #'J', CHECK_K
    MOV DPTR, #MSG_J
    LJMP DISPLAY_MORSE

CHECK_K:
    MOV A, SBUF
    CJNE A, #'K', CHECK_L
    MOV DPTR, #MSG_K
    LJMP DISPLAY_MORSE

CHECK_L:
    MOV A, SBUF
    CJNE A, #'L', CHECK_M
    MOV DPTR, #MSG_L
    LJMP DISPLAY_MORSE

CHECK_M:
    MOV A, SBUF
    CJNE A, #'M', CHECK_N
```

```
    MOV DPTR, #MSG_M
    LJMP DISPLAY_MORSE

CHECK_N:
    MOV A, SBUF
    CJNE A, #'N', CHECK_O
    MOV DPTR, #MSG_N
    LJMP DISPLAY_MORSE

CHECK_O:
    MOV A, SBUF
    CJNE A, #'O', CHECK_P
    MOV DPTR, #MSG_O
    LJMP DISPLAY_MORSE

CHECK_P:
    MOV A, SBUF
    CJNE A, #'P', CHECK_Q
    MOV DPTR, #MSG_P
    LJMP DISPLAY_MORSE

CHECK_Q:
    MOV A, SBUF
    CJNE A, #'Q', CHECK_R
    MOV DPTR, #MSG_Q
    LJMP DISPLAY_MORSE

CHECK_R:
    MOV A, SBUF
    CJNE A, #'R', CHECK_S
    MOV DPTR, #MSG_R
    LJMP DISPLAY_MORSE

CHECK_S:
    MOV A, SBUF
    CJNE A, #'S', CHECK_T
    MOV DPTR, #MSG_S
    LJMP DISPLAY_MORSE

CHECK_T:
    MOV A, SBUF
    CJNE A, #'T', CHECK_U
    MOV DPTR, #MSG_T
    LJMP DISPLAY_MORSE

CHECK_U:
    MOV A, SBUF
    CJNE A, #'U', CHECK_V
    MOV DPTR, #MSG_U
    LJMP DISPLAY_MORSE

CHECK_V:
    MOV A, SBUF
    CJNE A, #'V', CHECK_W
    MOV DPTR, #MSG_V
    LJMP DISPLAY_MORSE

CHECK_W:
    MOV A, SBUF
    CJNE A, #'W', CHECK_X
    MOV DPTR, #MSG_W
    LJMP DISPLAY_MORSE

CHECK_X:
    MOV A, SBUF
    CJNE A, #'X', CHECK_Y
    MOV DPTR, #MSG_X
    LJMP DISPLAY_MORSE

CHECK_Y:
    MOV A, SBUF
    CJNE A, #'Y', CHECK_Z
    MOV DPTR, #MSG_Y
    LJMP DISPLAY_MORSE

CHECK_Z:
    MOV A, SBUF
    CJNE A, #'Z', NEXT5
    MOV DPTR, #MSG_Z
    LJMP DISPLAY_MORSE

NEXT5: MOV A,SBUF
CJNE A, #'0', FINISH2      ; If '0' pressed, exit to mode selection
LJMP SELECT_MODE

FINISH2: LJMP CHECK_A
```

```
; Display morse code pattern from the lookup table
DISPLAY_MORSE:
    LOOP_MORSE:
        CLR A
        MOVC A, @A+DPTR     ; Get morse character (. or -)
        JNZ NOT_ZERO_MORSE  ; If not end of string
        ZERO_MORSE: LJMP CHECK_A  ; Go back for another letter

    NOT_ZERO_MORSE:
        LCALL DISPLAY       ; Display morse character on LCD
        LCALL DELAY
        INC DPTR            ; Next character
        LJMP LOOP_MORSE

;====================================================================================
; MODE 4 - SIMPLE ENCRYPTION: Caesar cipher (shift by 1)
;====================================================================================
MODE4:
ENCRYPTION:
    MOV 32H, #16
    LCALL CLEAR_LCD

    ENCRYPTION_INPUT:
     LCALL GET_IP        ; Get character to encrypt

        CJNE A, #'Z', NOT_Z_ENCRYPTION  ; Special case for Z
        MOV A, #'A'        ; Z wraps around to A
        LCALL DISPLAY
        LCALL DELAY
        LJMP ENCRYPTION_INPUT

        NOT_Z_ENCRYPTION:
            MOV A, SBUF
            CJNE A, #'0', CONTINUE_ENCRYPTION  ; '0' exits encryption mode
            LJMP SELECT_MODE

        CONTINUE_ENCRYPTION:
            ADD A, #1     ; Shift character forward by 1 (Caesar cipher)
            LCALL DISPLAY
            LCALL DELAY
            LJMP ENCRYPTION_INPUT

;====================================================================================
; MODE 5 - MORSE CODE DECODER: Translates morse code to letters
;====================================================================================
MODE5:
    INPUT_NEW_MORSE:
    ; Initialize registers used for storing morse code pattern
    MOV R4, #0    ; First character of morse code pattern
    MOV R6, #0    ; Third character of morse code pattern
    MOV R5, #0    ; Second character of morse code pattern
    MOV R7, #0    ; Fourth character of morse code pattern

    MOV R3, #1    ; Counter for position in morse code
    MOV R0, #50H ; Memory location for storing decoded characters

    MORSE_DECRYPT_INPUT:
        LCALL GET_IP        ; Get morse code input (. or -)
        ACALL DISPLAY       ; Show on LCD
        ACALL DELAY
        CJNE A, #' ', CONTINUE_MORSE_IP  ; Space means decode the accumulated pattern
        LJMP MORSE_DECRYPT

    CONTINUE_MORSE_IP:
       CJNE A, #'/', NEXT20  ; '/' means end of character
        LJMP DISPLAY_CHAR     ; Display decoded characters

        NEXT20:
        CJNE R3, #1, NOT_1  ; Check which position we're storing
        SJMP COUNTER_1      ; Store in first position

        NOT_1:
        CJNE R3, #2, NOT_2  ; Check for second position
        SJMP COUNTER_2

        NOT_2:
        CJNE R3, #3, NOT_3  ; Check for third position
        SJMP COUNTER_3

        NOT_3:
        CJNE R3, #4, NOT_4  ; Check for fourth position
        SJMP COUNTER_4

        NOT_4:
        LJMP MODE5          ; If more than 4 characters, restart
```

```
    ; Store morse character (. or -) in respective register based on position
    COUNTER_1:
    MOV R4, A             ; Store in R4 (first position)
    INC R3                ; Increment position counter
    LJMP MORSE_DECRYPT_INPUT

    COUNTER_2:
    MOV R5, A             ; Store in R5 (second position)
    INC R3
    LJMP MORSE_DECRYPT_INPUT

    COUNTER_3:
    MOV R6, A             ; Store in R6 (third position)
    INC R3
    LJMP MORSE_DECRYPT_INPUT

    COUNTER_4:
    MOV R7, A             ; Store in R7 (fourth position)
    LJMP MORSE_DECRYPT_INPUT

; Huge series of comparisons to identify the letter from morse pattern
MORSE_DECRYPT:
    ;CHECK FOR A (.- pattern)
    CJNE R4, #'.', NOT_MORSE_A    ; First char must be .
    CJNE R5, #'-', NOT_MORSE_A    ; Second char must be -
    CJNE R6, #0, NOT_MORSE_A      ; Third char must be empty
    CJNE R7, #0, NOT_MORSE_A      ; Fourth char must be empty
    LJMP MORSE_A                  ; If pattern matches, it's A

    ;CHECK FOR B (-... pattern)
    NOT_MORSE_A:
    CJNE R4, #'-', NOT_MORSE_B
    CJNE R5, #'.', NOT_MORSE_B
    CJNE R6, #'.', NOT_MORSE_B
    CJNE R7, #'.', NOT_MORSE_B
    LJMP MORSE_B

    ; CHECK FOR C (-.-. pattern)
    NOT_MORSE_B:
    CJNE R4, #'-', NOT_MORSE_C
    CJNE R5, #'.', NOT_MORSE_C
    CJNE R6, #'-', NOT_MORSE_C
    CJNE R7, #'.', NOT_MORSE_C
    LJMP MORSE_C

    ; Similar checks for D through Z
    ; [code continues for each letter]

    ; CHECK FOR D (-.. pattern)
    NOT_MORSE_C:
    CJNE R4, #'-', NOT_MORSE_D
    CJNE R5, #'.', NOT_MORSE_D
    CJNE R6, #'.', NOT_MORSE_D
    CJNE R7, #0, NOT_MORSE_D
    LJMP MORSE_D

    ; CHECK FOR E (. pattern)
    NOT_MORSE_D:
    CJNE R4, #'.', NOT_MORSE_E
    CJNE R5, #0, NOT_MORSE_E
    CJNE R6, #0, NOT_MORSE_E
    CJNE R7, #0, NOT_MORSE_E
    LJMP MORSE_E

    ; CHECK FOR F (..-. pattern)
    NOT_MORSE_E:
    CJNE R4, #'.', NOT_MORSE_F
    CJNE R5, #'.', NOT_MORSE_F
    CJNE R6, #'-', NOT_MORSE_F
    CJNE R7, #'.', NOT_MORSE_F
    LJMP MORSE_F

    ; CHECK FOR G (--. pattern)
    NOT_MORSE_F:
    CJNE R4, #'-', NOT_MORSE_G
    CJNE R5, #'-', NOT_MORSE_G
    CJNE R6, #'.', NOT_MORSE_G
    CJNE R7, #0, NOT_MORSE_G
    LJMP MORSE_G

    ; CHECK FOR H (.... pattern)
    NOT_MORSE_G:
    CJNE R4, #'.', NOT_MORSE_H
    CJNE R5, #'.', NOT_MORSE_H
    CJNE R6, #'.', NOT_MORSE_H
```

```
CJNE R7, #'.', NOT_MORSE_H
LJMP MORSE_H

; CHECK FOR I (.. pattern)
NOT_MORSE_H:
CJNE R4, #'.', NOT_MORSE_I
CJNE R5, #'.', NOT_MORSE_I
CJNE R6, #0, NOT_MORSE_I
CJNE R7, #0, NOT_MORSE_I
LJMP MORSE_I

; CHECK FOR J (.--- pattern)
NOT_MORSE_I:
CJNE R4, #'.', NOT_MORSE_J
CJNE R5, #'-', NOT_MORSE_J
CJNE R6, #'-', NOT_MORSE_J
CJNE R7, #'-', NOT_MORSE_J
LJMP MORSE_J

; CHECK FOR K (-.- pattern)
NOT_MORSE_J:
CJNE R4, #'-', NOT_MORSE_K
CJNE R5, #'.', NOT_MORSE_K
CJNE R6, #'-', NOT_MORSE_K
CJNE R7, #0, NOT_MORSE_K
LJMP MORSE_K

; CHECK FOR L (.-.. pattern)
NOT_MORSE_K:
CJNE R4, #'.', NOT_MORSE_L
CJNE R5, #'-', NOT_MORSE_L
CJNE R6, #'.', NOT_MORSE_L
CJNE R7, #'.', NOT_MORSE_L
LJMP MORSE_L

; CHECK FOR M (-- pattern)
NOT_MORSE_L:
CJNE R4, #'-', NOT_MORSE_M
CJNE R5, #'-', NOT_MORSE_M
CJNE R6, #0, NOT_MORSE_M
CJNE R7, #0, NOT_MORSE_M
LJMP MORSE_M

; CHECK FOR N (-. pattern)
NOT_MORSE_M:
CJNE R4, #'-', NOT_MORSE_N
CJNE R5, #'.', NOT_MORSE_N
CJNE R6, #0, NOT_MORSE_N
CJNE R7, #0, NOT_MORSE_N
LJMP MORSE_N

; CHECK FOR O (--- pattern)
NOT_MORSE_N:
CJNE R4, #'-', NOT_MORSE_O
CJNE R5, #'-', NOT_MORSE_O
CJNE R6, #'-', NOT_MORSE_O
CJNE R7, #0, NOT_MORSE_O
LJMP MORSE_O

; CHECK FOR P (.--. pattern)
NOT_MORSE_O:
CJNE R4, #'.', NOT_MORSE_P
CJNE R5, #'-', NOT_MORSE_P
CJNE R6, #'-', NOT_MORSE_P
CJNE R7, #'.', NOT_MORSE_P
LJMP MORSE_P

; CHECK FOR Q (--.- pattern)
NOT_MORSE_P:
CJNE R4, #'-', NOT_MORSE_Q
CJNE R5, #'-', NOT_MORSE_Q
CJNE R6, #'.', NOT_MORSE_Q
CJNE R7, #'-', NOT_MORSE_Q
LJMP MORSE_Q

; CHECK FOR R (.-. pattern)
NOT_MORSE_Q:
CJNE R4, #'.', NOT_MORSE_R
CJNE R5, #'-', NOT_MORSE_R
CJNE R6, #'.', NOT_MORSE_R
CJNE R7, #0, NOT_MORSE_R
LJMP MORSE_R

; CHECK FOR S (... pattern)
NOT_MORSE_R:
CJNE R4, #'.', NOT_MORSE_S
```

```
        CJNE R5, #'.', NOT_MORSE_S
        CJNE R6, #'.', NOT_MORSE_S
        CJNE R7, #0, NOT_MORSE_S
        LJMP MORSE_S

        ; CHECK FOR T (- pattern)
        NOT_MORSE_S:
        CJNE R4, #'-', NOT_MORSE_T
        CJNE R5, #0, NOT_MORSE_T
        CJNE R6, #0, NOT_MORSE_T
        CJNE R7, #0, NOT_MORSE_T
        LJMP MORSE_T

        ; CHECK FOR U (..- pattern)
        NOT_MORSE_T:
        CJNE R4, #'.', NOT_MORSE_U
        CJNE R5, #'.', NOT_MORSE_U
        CJNE R6, #'-', NOT_MORSE_U
        CJNE R7, #0, NOT_MORSE_U
        LJMP MORSE_U

        ; CHECK FOR V (...- pattern)
        NOT_MORSE_U:
        CJNE R4, #'.', NOT_MORSE_V
        CJNE R5, #'.', NOT_MORSE_V
        CJNE R6, #'.', NOT_MORSE_V
        CJNE R7, #'-', NOT_MORSE_V
        LJMP MORSE_V

        ; CHECK FOR W (.-- pattern)
        NOT_MORSE_V:
        CJNE R4, #'.', NOT_MORSE_W
        CJNE R5, #'-', NOT_MORSE_W
        CJNE R6, #'-', NOT_MORSE_W
        CJNE R7, #0, NOT_MORSE_W
        LJMP MORSE_W

        ; CHECK FOR X (-..- pattern)
        NOT_MORSE_W:
        CJNE R4, #'-', NOT_MORSE_X
        CJNE R5, #'.', NOT_MORSE_X
        CJNE R6, #'.', NOT_MORSE_X
        CJNE R7, #'-', NOT_MORSE_X
        LJMP MORSE_X

        ; CHECK FOR Y (-.-- pattern)
        NOT_MORSE_X:
        CJNE R4, #'-', NOT_MORSE_Y
        CJNE R5, #'.', NOT_MORSE_Y
        CJNE R6, #'-', NOT_MORSE_Y
        CJNE R7, #'-', NOT_MORSE_Y
        LJMP MORSE_Y

        ; CHECK FOR Z (--.. pattern)
        NOT_MORSE_Y:
        CJNE R4, #'-', NOT_MORSE_Z
        CJNE R5, #'-', NOT_MORSE_Z
        CJNE R6, #'.', NOT_MORSE_Z
        CJNE R7, #'.', NOT_MORSE_Z
        LJMP MORSE_Z

        NOT_MORSE_Z:
        LJMP MODE5  ; If no match found, restart mode

        ; For each letter, load the character and blink LED in morse pattern
MORSE_A:
            MOV A, #'A'
            LJMP MORSE_LED

MORSE_B:
            MOV A, #'B'
            LJMP MORSE_LED

MORSE_C:
            MOV A, #'C'
            LJMP MORSE_LED

MORSE_D:
            MOV A, #'D'
            LJMP MORSE_LED

MORSE_E:
            MOV A, #'E'
            LJMP MORSE_LED

MORSE_F:
```

```
            MOV A, #'F'
            LJMP MORSE_LED

MORSE_G:
            MOV A, #'G'
            LJMP MORSE_LED

MORSE_H:
            MOV A, #'H'
            LJMP MORSE_LED

MORSE_I:
            MOV A, #'I'
            LJMP MORSE_LED

MORSE_J:
            MOV A, #'J'
            LJMP MORSE_LED

MORSE_K:
            MOV A, #'K'
            LJMP MORSE_LED

MORSE_L:
            MOV A, #'L'
            LJMP MORSE_LED

MORSE_M:
            MOV A, #'M'
            LJMP MORSE_LED

MORSE_N:
            MOV A, #'N'
            LJMP MORSE_LED

MORSE_O:
            MOV A, #'O'
            LJMP MORSE_LED


MORSE_P:
    MOV A, #'P'         ; Load ASCII value of 'P' into the accumulator
    LJMP MORSE_LED      ; Jump to MORSE_LED routine to display P in Morse code

MORSE_Q:
    MOV A, #'Q'         ; Load ASCII value of 'Q' into the accumulator
    LJMP MORSE_LED      ; Jump to MORSE_LED routine to display Q in Morse code

MORSE_R:
    MOV A, #'R'         ; Load ASCII value of 'R' into the accumulator
    LJMP MORSE_LED      ; Jump to MORSE_LED routine to display R in Morse code

MORSE_S:
    MOV A, #'S'         ; Load ASCII value of 'S' into the accumulator
    LJMP MORSE_LED      ; Jump to MORSE_LED routine to display S in Morse code

MORSE_T:
    MOV A, #'T'         ; Load ASCII value of 'T' into the accumulator
    LJMP MORSE_LED      ; Jump to MORSE_LED routine to display T in Morse code

MORSE_U:
    MOV A, #'U'         ; Load ASCII value of 'U' into the accumulator
    LJMP MORSE_LED      ; Jump to MORSE_LED routine to display U in Morse code

MORSE_V:
    MOV A, #'V'         ; Load ASCII value of 'V' into the accumulator
    LJMP MORSE_LED      ; Jump to MORSE_LED routine to display V in Morse code

MORSE_W:
    MOV A, #'W'         ; Load ASCII value of 'W' into the accumulator
    LJMP MORSE_LED      ; Jump to MORSE_LED routine to display W in Morse code

MORSE_X:
    MOV A, #'X'         ; Load ASCII value of 'X' into the accumulator
    LJMP MORSE_LED      ; Jump to MORSE_LED routine to display X in Morse code

MORSE_Y:
    MOV A, #'Y'         ; Load ASCII value of 'Y' into the accumulator
    LJMP MORSE_LED      ; Jump to MORSE_LED routine to display Y in Morse code

MORSE_Z:
    MOV A, #'Z'         ; Load ASCII value of 'Z' into the accumulator
    LJMP MORSE_LED      ; Jump to MORSE_LED routine to display Z in Morse code

MORSE_LED:
    MOV @R0, A          ; Store the character in memory location pointed by R0
```

```
    INC R0              ; Increment R0 to point to the next memory location

    CHECK_R4:           ; Check the Morse code pattern stored in R4
        CJNE R4, #'-', R4_NOT_DASH  ; Check if R4 contains dash
        LJMP R4_DASH                ; If it is a dash, jump to R4_DASH

        R4_NOT_DASH:
            CJNE R4, #'.', R4_NOT_DOT  ; Check if R4 contains dot
            LJMP R4_DOT                 ; If it is a dot, jump to R4_DOT

        R4_NOT_DOT:
            CJNE R4, #'0', R4_NOT_ZERO  ; Check if R4 contains '0' (end of pattern)
            LJMP R4_ZERO                 ; If it is '0', jump to R4_ZERO

        R4_NOT_ZERO:
            LJMP INPUT_NEW_MORSE         ; If none of the above, get new input

        R4_DASH:
            LCALL DASH                 ; Call DASH subroutine to display dash on LED
            LJMP CHECK_R5              ; Check next character in Morse code

        R4_DOT:
            LCALL DOT                  ; Call DOT subroutine to display dot on LED
            LJMP CHECK_R5              ; Check next character in Morse code

        R4_ZERO:
            LJMP INPUT_NEW_MORSE       ; End of pattern, get new Morse code input

    CHECK_R5:           ; Check the Morse code pattern stored in R5
        CJNE R5, #'-', R5_NOT_DASH  ; Check if R5 contains dash
        LJMP R5_DASH                ; If it is a dash, jump to R5_DASH

        R5_NOT_DASH:
            CJNE R5, #'.', R5_NOT_DOT  ; Check if R5 contains dot
            LJMP R5_DOT                 ; If it is a dot, jump to R5_DOT

        R5_NOT_DOT:
            CJNE R5, #'0', R5_NOT_ZERO  ; Check if R5 contains '0' (end of pattern)
            LJMP R5_ZERO                 ; If it is '0', jump to R5_ZERO

        R5_NOT_ZERO:
            LJMP INPUT_NEW_MORSE         ; If none of the above, get new input

        R5_DASH:
            LCALL DASH                 ; Call DASH subroutine to display dash on LED
            LJMP CHECK_R6              ; Check next character in Morse code

        R5_DOT:
            LCALL DOT                  ; Call DOT subroutine to display dot on LED
            LJMP CHECK_R6              ; Check next character in Morse code

        R5_ZERO:
            LJMP INPUT_NEW_MORSE       ; End of pattern, get new Morse code input

    CHECK_R6:           ; Check the Morse code pattern stored in R6
        CJNE R6, #'-', R6_NOT_DASH  ; Check if R6 contains dash
        LJMP R6_DASH                ; If it is a dash, jump to R6_DASH

        R6_NOT_DASH:
            CJNE R6, #'.', R6_NOT_DOT  ; Check if R6 contains dot
            LJMP R6_DOT ; If it is a dot, jump to R6_DOT

        R6_NOT_DOT:
            CJNE R6, #'0', R6_NOT_ZERO  ; Check if R6 contains '0' (end of pattern)
            LJMP R6_ZERO  ; If it is '0', jump to R6_ZERO

        R6_NOT_ZERO:
            LJMP INPUT_NEW_MORSE         ; If none of the above, get new input

        R6_DASH:
            LCALL DASH                 ; Call DASH subroutine to display dash on LED
            LJMP CHECK_R7              ; Check next character in Morse code

        R6_DOT:
            LCALL DOT   ; Call DOT subroutine to display dot on LED
            LJMP CHECK_R7     ; Check next character in Morse code

        R6_ZERO:
            LJMP INPUT_NEW_MORSE       ; End of pattern, get new Morse code input

    CHECK_R7:           ; Check the Morse code pattern stored in R7
        CJNE R7, #'-', R7_NOT_DASH  ; Check if R7 contains dash
        LJMP R7_DASH                ; If it is a dash, jump to R7_DASH

        R7_NOT_DASH:
            CJNE R7, #'.', R7_NOT_DOT  ; Check if R7 contains dot
```

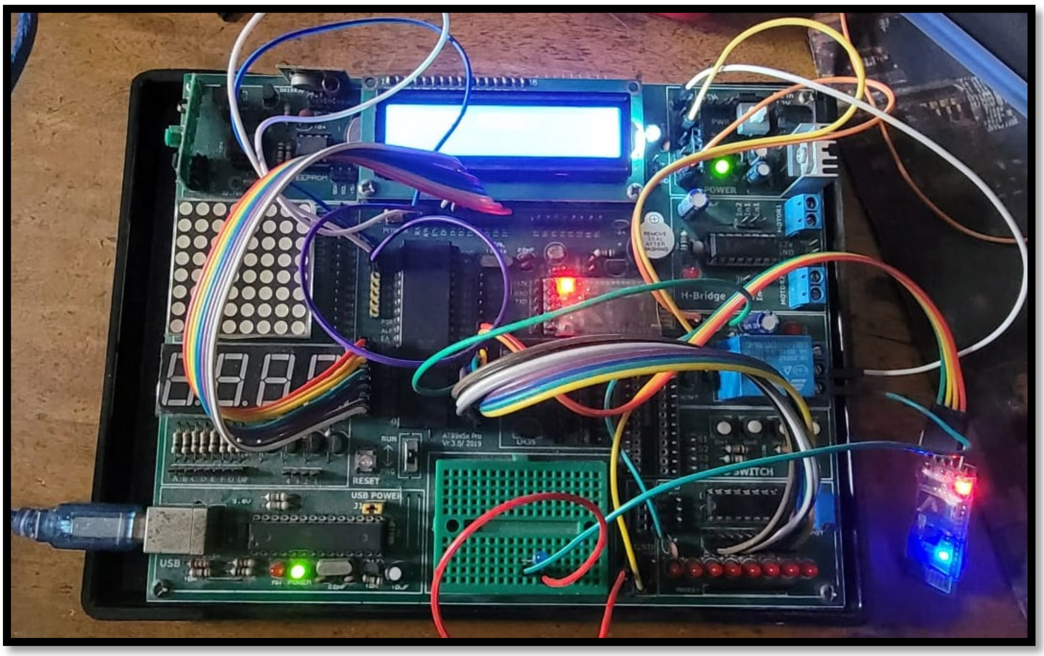
```
            LJMP R7_DOT                ; If it is a dot, jump to R7_DOT

        R7_NOT_DOT:
            CJNE R7, #'0', R7_NOT_ZERO  ; Check if R7 contains '0' (end of pattern)
            LJMP R7_ZERO                ; If it is '0', jump to R7_ZERO

        R7_NOT_ZERO:
            LJMP INPUT_NEW_MORSE        ; If none of the above, get new input

        R7_DASH:
            LCALL DASH                 ; Call DASH subroutine to display dash on LED
            LJMP INPUT_NEW_MORSE       ; Get new Morse code input

        R7_DOT:
            LCALL DOT                  ; Call DOT subroutine to display dot on LED
            LJMP INPUT_NEW_MORSE       ; Get new Morse code input

        R7_ZERO:
            LJMP INPUT_NEW_MORSE       ; End of pattern, get new Morse code input


DISPLAY_CHAR:
    MOV @R0, #00H       ; Clear memory location pointed by R0
    MOV R0, #50H        ; Set R0 to point to memory location 50H (for display buffer)

    LCALL DELAY         ; Call delay subroutine
    MOV A, #0C0H        ; Load command to move to 2nd line of LCD
    ACALL COMMAND       ; Send command to LCD
    ACALL DELAY         ; Call delay subroutine

LOOP100:
    MOV A, @R0          ; Get character from memory location pointed by R0
    JZ EXIT_MODE5       ; If character is zero, exit mode 5
    ACALL DISPLAY       ; Display character on LCD
    ACALL DELAY         ; Call delay subroutine
    INC R0              ; Increment R0 to point to next character
    SJMP LOOP100        ; Loop back to display next character

EXIT_MODE5:
    ACALL GET_IP        ; Get input from user
    CJNE A, #'0', EXIT_MODE5  ; If input is not '0', keep waiting
    MOV 32H, #16        ; Set counter for clearing LCD
    LCALL CLEAR_LCD     ; Clear LCD display
    MOV 32H, #16        ; Reset counter
    LCALL CLEAR_LCD     ; Clear LCD display again
    LJMP select_mode    ; Jump to mode selection


mode6:                  ; Calculator mode (Mode 6)
    acall GET_IP        ; Get first digit of first number
    ACALL DISPLAY       ; Display the digit
    ACALL DELAY         ; Call delay subroutine

    CLR C               ; Clear carry flag
    SUBB A, #'0'        ; Convert ASCII to numeric value
    MOV R5, A           ; Store in R5

    acall GET_IP        ; Get second digit of first number
    ACALL DISPLAY       ; Display the digit
    ACALL DELAY         ; Call delay subroutine

    CLR C               ; Clear carry flag
    SUBB A, #'0'        ; Convert ASCII to numeric value

    MOV B, #10          ; Multiply first digit by 10
    MUL AB              ; A = A * 10
    ADD A, R5           ; Add second digit
    mov r5, A           ; Store first number in R5

    acall GET_IP        ; Get operator (+, -, *)
    ACALL DISPLAY       ; Display the operator
    ACALL DELAY         ; Call delay subroutine
    MOV R6, A           ; Store operator in R6

    acall GET_IP        ; Get first digit of second number
    ACALL DISPLAY       ; Display the digit
    ACALL DELAY         ; Call delay subroutine

    CLR C               ; Clear carry flag
    SUBB A, #'0'        ; Convert ASCII to numeric value
    MOV R7, A           ; Store in R7

    acall GET_IP        ; Get second digit of second number
    ACALL DISPLAY       ; Display the digit
    ACALL DELAY         ; Call delay subroutine
```

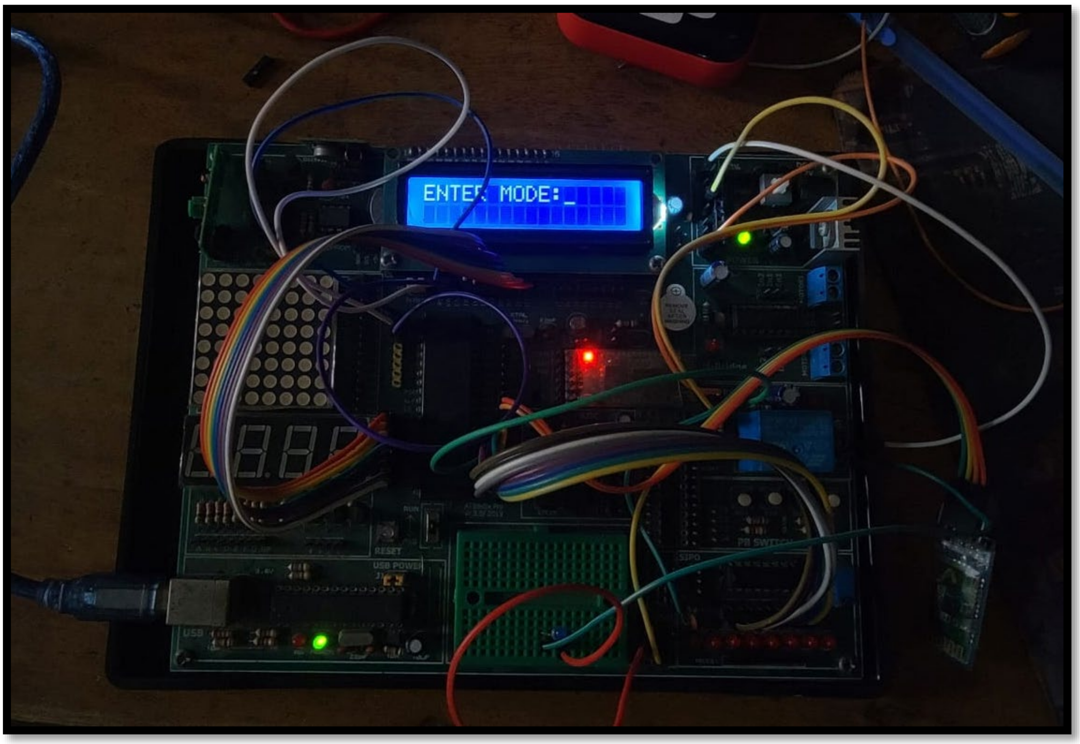
```
    CLR C               ; Clear carry flag
    SUBB A, #'0'        ; Convert ASCII to numeric value

    MOV B, #10          ; Multiply first digit by 10
    MUL AB              ; A = A * 10
    ADD A, R7           ; Add second digit
    mov r7, A           ; Store second number in R7

    MOV A, #'='         ; Load '=' character
    ACALL DISPLAY       ; Display '='
    ACALL DELAY         ; Call delay subroutine

    MOV A, R6           ; Load operator
    CJNE A, #'+', NEXT10  ; Check if operator is '+'
    MOV A, R5           ; If '+', load first number
    ADD A, R7           ; Add second number
    ACALL DISPLAY_OP    ; Display result
    LJMP END_MODE6      ; Jump to end of mode 6

NEXT10:
    CJNE A, #'-', NEXT11  ; Check if operator is '-'
    CLR C               ; Clear carry flag
    MOV A, R5           ; If '-', load first number
    SUBB A, R7          ; Subtract second number
    ACALL DISPLAY_OP    ; Display result
    LJMP END_MODE6      ; Jump to end of mode 6

NEXT11:                 ; If not '+' or '-', assume '*'
    CLR C               ; Clear carry flag
    MOV A, R5           ; Load first number
    MOV B, R7           ; Load second number
    MUL AB              ; Multiply A and B
    ACALL DISPLAY_OP    ; Display result
    LJMP END_MODE6      ; Jump to end of mode 6

END_MODE6:
    ACALL GET_IP        ; Get input from user
    CJNE A, #'0', END_MODE6  ; If input is not '0', keep waiting
    MOV 32H, #16        ; Set counter for clearing LCD
    LCALL CLEAR_LCD     ; Clear LCD display
    LJMP select_mode    ; Jump to mode selection


DISPLAY_OP:             ; Display a 3-digit number in decimal
    MOV B, #10          ; Set divisor to 10
    DIV AB              ; Divide A by 10, A = quotient, B = remainder (units place)
    MOV R0, B           ; Store units place temporarily

    MOV B, #10          ; Set divisor to 10 again
    DIV AB              ; Divide A by 10, A = hundreds place, B = tens place
    PUSH B              ; Save tens place on stack

    PUSH 0              ; Save units place on stack

    ADD A, #'0'         ; Convert hundreds place to ASCII
    ACALL DISPLAY       ; Display hundreds place
    ACALL DELAY         ; Call delay subroutine

    POP ACC             ; Retrieve tens place
    ADD A, #'0'         ; Convert to ASCII
    ACALL DISPLAY       ; Display tens place
    ACALL DELAY         ; Call delay subroutine

    POP ACC             ; Retrieve units place
    ADD A, #'0'         ; Convert to ASCII
    ACALL DISPLAY       ; Display units place
    ACALL DELAY         ; Call delay subroutine
    RET                 ; Return from subroutine


CLEAR_LCD:              ; Clear LCD display
    LCALL DELAY         ; Call delay subroutine
    MOV A, #01H         ; Load clear display command
    LCALL COMMAND       ; Send command to LCD
    LCALL DELAY         ; Call delay subroutine
    RET                 ; Return from subroutine


GET_IP:                 ; Get input from serial port
    ACALL DELAY         ; Call delay subroutine
    CLR A               ; Clear accumulator
    CLR RI              ; Clear receive interrupt flag (get ready to receive data)
    WAIT_INPUT:
        JNB RI, WAIT_INPUT  ; Wait until data is received (RI flag set)
    mov a, sbuf         ; Move received data to accumulator
    RET                 ; Return from subroutine
```

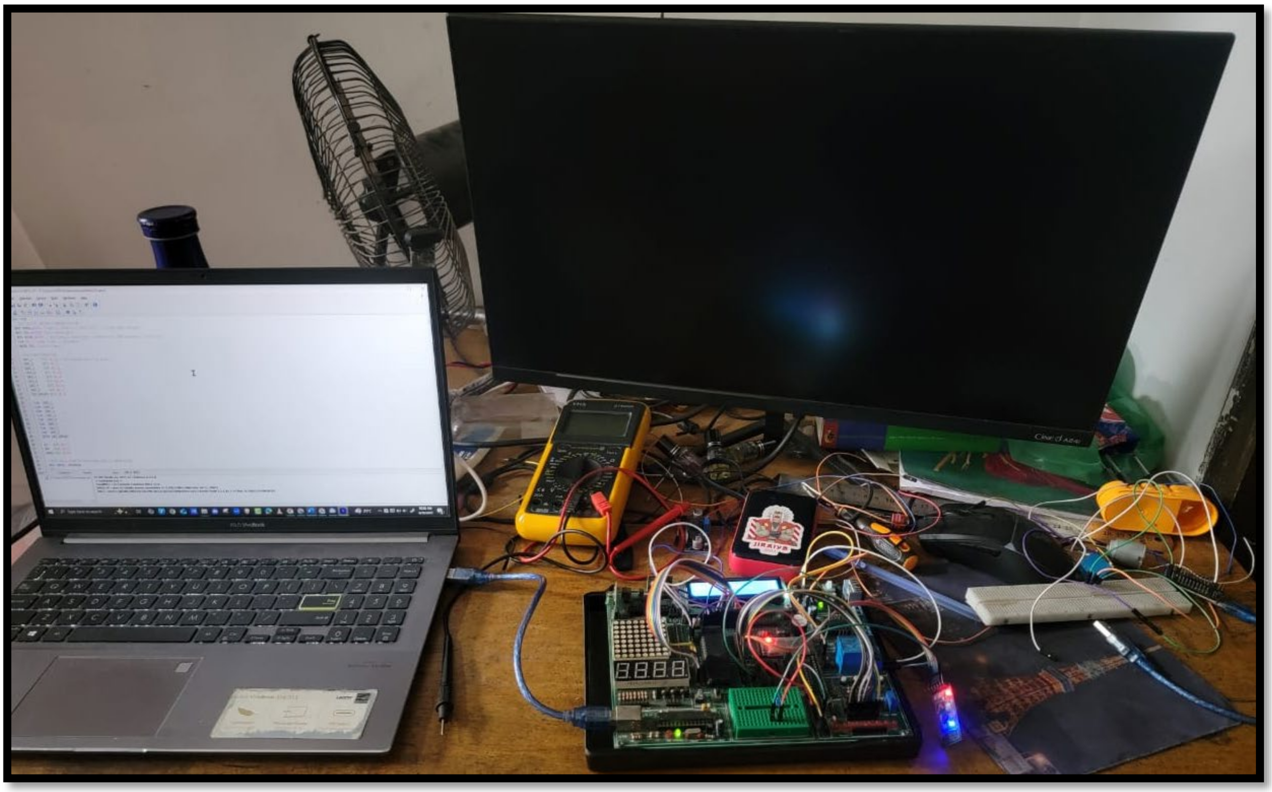
```
COMMAND:                ; Send command to LCD
    LCALL READY         ; Check if LCD is ready
    MOV P0, A           ; Output command to port 0
    CLR RS              ; Select command register (RS=0)
    CLR RW              ; Set for write operation (RW=0)
    SETB ENBL           ; Set enable bit high
    LCALL DELAY         ; Call delay subroutine
    CLR ENBL            ; Set enable bit low (H to L pulse)
    RET                 ; Return from subroutine


DISPLAY:                ; Display data on LCD
    LCALL READY         ; Check if LCD is ready
    MOV P0, A           ; Output data to port 0
    SETB RS             ; Select data register (RS=1)
    CLR RW              ; Set for write operation (RW=0)
    SETB ENBL           ; Set enable bit high
    LCALL DELAY         ; Call delay subroutine
    CLR ENBL            ; Set enable bit low (H to L pulse)
    DJNZ 32H, SAME_LINE  ; Decrement counter, if not zero stay on same line
    LJMP NEW_LINE       ; If counter is zero, move to new line

    NEW_LINE:           ; Move cursor to second line of LCD
        LCALL DELAY     ; Call delay subroutine
        MOV A, #0C0H    ; Load command to move to 2nd line
        LCALL COMMAND   ; Send command to LCD
        LCALL DELAY     ; Call delay subroutine
    SAME_LINE:          ; Continue on same line
    RET                 ; Return from subroutine

READY:                  ; Check if LCD is ready to receive commands/data
    SETB P0.7           ; Set bit 7 of port 0 as input (busy flag)
    CLR RS              ; Select command register (RS=0)
    SETB RW             ; Set for read operation (RW=1)

WAIT:                   ; Wait until LCD is not busy
    CLR ENBL            ; Set enable bit low
    ACALL DELAY         ; Call delay subroutine
    SETB ENBL           ; Set enable bit high
    JB P0.7, WAIT       ; If busy flag is set, keep waiting
    RET                 ; Return from subroutine


DASH:                   ; Display dash (long signal) on LED
    SETB LED_1          ; Turn on LED

    LCALL DELAY         ; Call delay subroutine 6 times for long signal
    LCALL DELAY
    LCALL DELAY
    LCALL DELAY
    LCALL DELAY
    LCALL DELAY
    CLR LED_1           ; Turn off LED
    LCALL DELAY         ; Short delay between signals
    RET                 ; Return from subroutine


DOT:                    ; Display dot (short signal) on LED
    SETB LED_1          ; Turn on LED

    LCALL DELAY         ; Call delay subroutine 2 times for short signal
    LCALL DELAY
    CLR LED_1           ; Turn off LED

    LCALL DELAY         ; Short delay between signals
    RET                 ; Return from subroutine

DELAY:                  ; Delay subroutine using nested loops
    MOV R1, #1          ; Set outer loop counter
    AGAIN_3:
        MOV R3, #220    ; Set middle loop counter
        AGAIN_2:
            MOV R4, #220  ; Set inner loop counter
            AGAIN:
                DJNZ R4, AGAIN     ; Decrement inner counter until zero
            DJNZ R3, AGAIN_2       ; Decrement middle counter until zero
        DJNZ R1, AGAIN_3           ; Decrement outer counter until zero
    RET                  ; Return from subroutine

EXIT_2: SJMP EXIT_2     ; Infinite loop for program end
END                     ; End of program
```

Hardware Implementation

Snapshots of our Project:







Problems Faced

1. **Serial Communication Timing Issues**
   - **Problem**: Initially, the serial communication was unstable, with garbled characters appearing on both the smartphone app and LCD display.
   - **Solution**: We implemented the precise 11.0592 MHz crystal oscillator to ensure accurate baud rate generation. Additionally, we verified the TH1 register value (0xFD) for generating exactly 9600 baud rate with Timer 1 in Mode 2. This eliminated the timing discrepancies.
2. **LCD Initialization Sequence**
   - **Problem**: The LCD would occasionally show random characters or not display anything at all.
   - **Solution**: We refined the LCD initialization sequence by adding proper delays between commands and implementing the busy flag checking mechanism (READY subroutine) to ensure the LCD controller was ready before sending new commands.

3. **Morse Code Timing Challenges**
   - **Problem**: The visual representation of Morse code using LEDs was either too fast to distinguish or inconsistent.
   - **Solution**: We calibrated the timing by creating specific DOT and DASH subroutines with standardized durations. The dot was set to 2 delay cycles, while the dash was set to 6 delay cycles, maintaining the standard 1:3 ratio for Morse code timing.
4. **Input Buffer Management**
   - **Problem**: When receiving multiple characters rapidly via Bluetooth, some characters would be missed or processed incorrectly.
   - **Solution**: We implemented proper handling of the RI flag in the GET_IP subroutine, ensuring that each character is fully received before processing the next one. This improved the reliability of multi-character inputs, especially in the calculator and encryption modes.
5. **Mode Switching Logic**
   - **Problem**: Occasionally, the system would enter an undefined state when switching between modes rapidly.
   - **Solution**: We standardized the return-to-menu logic by implementing consistent checks for the '0' command across all modes. Additionally, we added proper initialization steps at the beginning of each mode to reset relevant variables and clear the display.

Conclusion

This project successfully demonstrated the integration of Bluetooth communication with the 8051 microcontroller to create a versatile multi-mode system. The implementation of various features—from basic LED control to more complex operations like Morse code translation and encryption—showcases the flexibility and capability of the 8051 architecture despite its relatively limited resources.

Key achievements of this project include:

1. Establishing reliable wireless communication between a smartphone and microcontroller
2. Implementing multiple operational modes with distinct functionalities
3. Demonstrating practical applications of microcontroller programming concepts
4. Integrating multiple peripheral devices including LEDs, LCD, and relay

Potential improvements for future iterations could include:

- Implementing bidirectional communication for enhanced smartphone app integration
- Adding more advanced encryption algorithms
- Improving power management for battery-powered operation

Overall, the project successfully meets all initial objectives and provides a solid foundation for further exploration in embedded systems development and wireless communication protocols.

xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx