Microprocessor and Microcontroller Based Design Project

Wahaaj Nasir, Syed Adnan Aijaz Bukhari
Department of Computer & Software Engineering
College of E&ME, NUST, Rawalpindi
Email: *wahaajnasir1@gmail.com*

***Abstract*—This report documents the design and implementation of an Encoder and Decoder using the PIC18F452 Microcontroller. The system supports two Caesar Cipher encoders/decoders and a Morse Code encoder/decoder, demonstrating the effective application of the microcontroller.**

I. Introduction

The PIC microcontroller, introduced by Microchip Technologies in 1993, has been a staple in embedded systems design. This project was an opportunity to consolidate knowledge of programming and hardware utilization through a practical implementation. The objectives included designing encoders and decoders using the PIC18F452, initially targeting Caesar Cipher with shifts of 3 and 5, and later extending to Morse Code encoding.

II. Objectives

The Project initially involved using the PIC18F452 Microcontroller to create an Encoder and Decoder of various types. There were only 2 types of encoder and decoder at first, a Caesar Cipher with shift 3 and one with Shift 5, but we ended up incorporating Morse Code into it as well.

III. Planning

A. Software Used

We used the following programs:

- MikroC
- PICKit 3 v3.10
- MPLab 8.88

IV. Hardware Used

- PIC18F452
- PICKit3
- DIP-Switch (8 Switches)
- Potentiometer – 10K
- 7-Segment Displays
- Crystal Oscillator 8MHz
- Capacitors – 22 μF
- Resistors

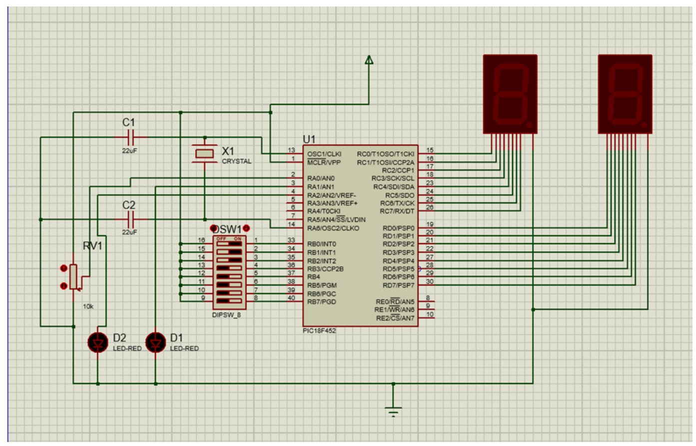

Fig. 1. The entire Proteus Design, showing all components connected to the PIC18F452.

V. Proteus Design

VI. Key Components

A. Crystal Oscillator

Configuration: Connected to OSC1 and OSC2 with grounding.
Purpose: Ensures proper frequency as the internal oscillator proved unreliable.

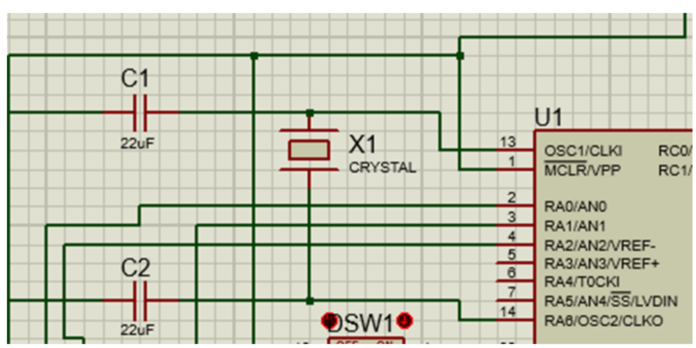

Fig. 2. Crystal Oscillator connected to OSC1 and OSC2. Its proximity to the microcontroller ensures stable oscillation.

B. DIP Switch

Connection: PORTB
Purpose: Provides input for encoding/decoding selection. It

also determines the cipher function to execute based on user input.

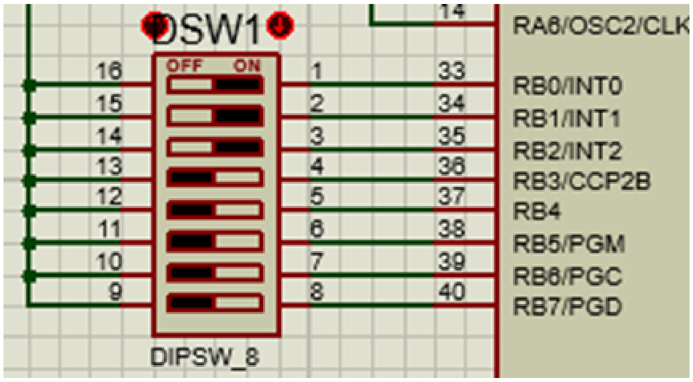

Fig. 3. The DIP Switch inputs are read by PORTB to determine encoding or decoding mode and function type.

C. Potentiometer

Connection: RA0
Purpose: Analog reading is used to select the encoding/decoding function dynamically.

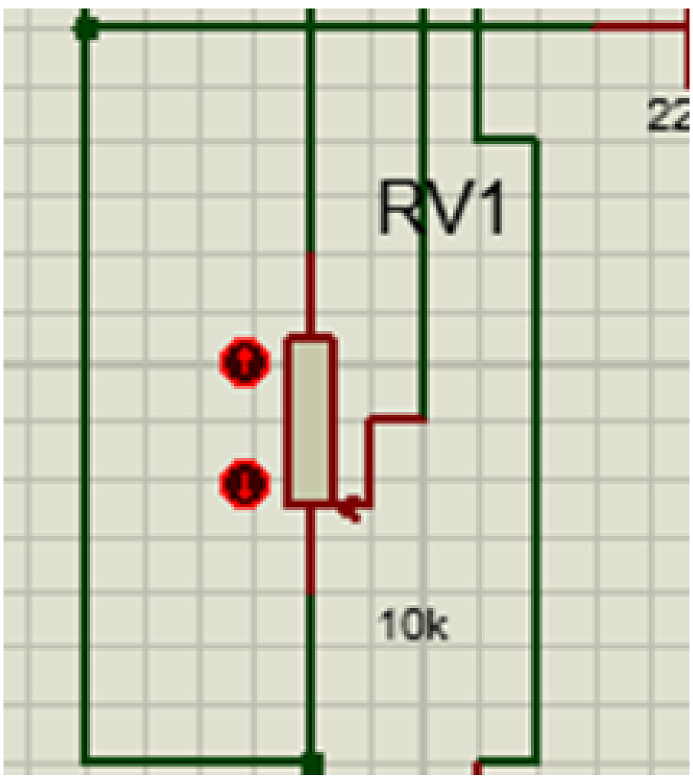

Fig. 4. The Potentiometer connected to RA0 provides analog input, dividing ADC values into functional ranges.

D. LEDs

Connection: RA1 and RA2
Purpose: Indicate the selected cipher function.
The LEDs visually represent the encoding or decoding function in use, based on their binary state:

- **0 0:** Caesar Cipher with Shift 3.
- **0 1:** Caesar Cipher with Shift 5.
- **1 0:** Morse Code Cipher.
- **1 1:** Combination of Morse Code and Caesar Cipher with Shift 3.

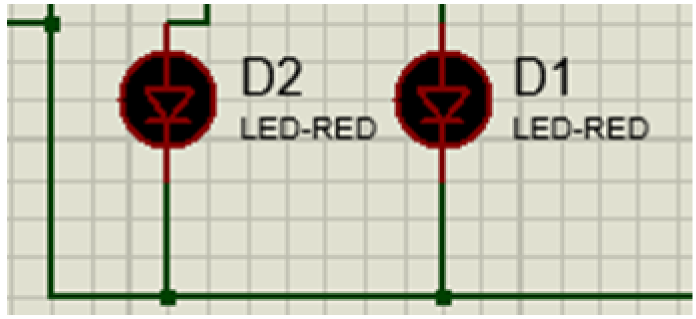

Fig. 5. LEDs connected to RA1 and RA2 indicate the active cipher function.

E. 7-Segment Displays

Connection: PORTC (input display), PORTD (output display)
Purpose: Displays the input and output values visually.

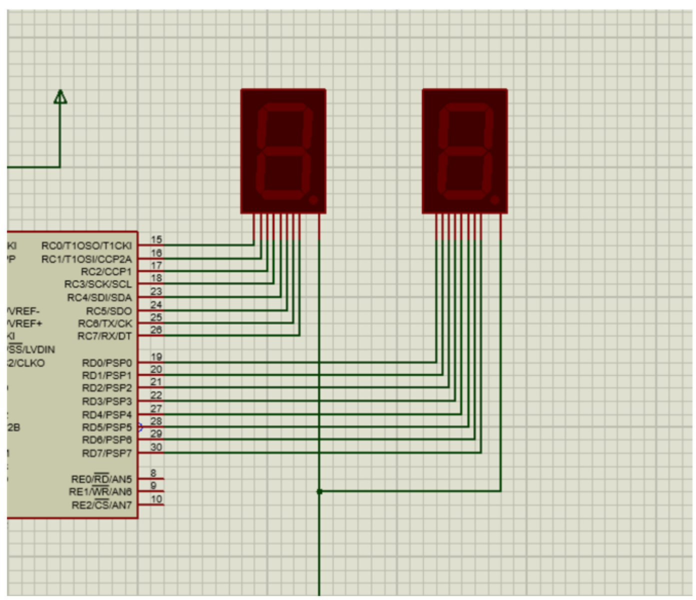

Fig. 6. 7-Segment Displays showing input on PORTC and output on PORTD.

VII. Code Implementation

A. 7 Segment Displays

Listing 1. 7 Segment Display Values

```
unsigned char segmentValues[] =
{
	0xBF, //0
	0x86, //1
	0xDB, //2
```

```
        0xCF, //3
        0xE6, //4
        0xED, //5
        0xFD, //6
        0x87, //7
        0xFF, //8
        0xE7, //9
        0x77, //A
        0x7C, //b
        0x39, //C
        0x5E, //d
        0x79, //E
        0x71, //F
        0x6F, //g
        0x76, //H
        0x06, //I
        0x1F, //J
        0x75, //k
        0x38, //L
        0x55, //m
        0x54, //n
        0x3F, //O
        0x73, //P
        0x67, //q
        0x50, //r
        0x6D, //S
        0x78, //t
        0x1C, //u
        0x2A, //V
        0x6A, //W
        0x36, //X
        0x6E, //Y
        0x5B, //Z
};

unsigned char get7SegVal(unsigned char input_num)
{
    switch(input_num)
    {
        case 0b000000: return 0; // 0
        case 0b000001: return 1; // 1
        case 0b000010: return 2; // 2
        case 0b000011: return 3; // 3
        case 0b000100: return 4; // 4
        case 0b000101: return 5; // 5
        case 0b000110: return 6; // 6
        case 0b000111: return 7; // 7
        case 0b001000: return 8; // 8
        case 0b001001: return 9; // 9
        case 0b001010: return 10; // A
        case 0b001011: return 11; // B
        case 0b001100: return 12; // C
        case 0b001101: return 13; // D
        case 0b001110: return 14; // E
        case 0b001111: return 15; // F
        case 0b010000: return 16; // G
        case 0b010001: return 17; // H
        case 0b010010: return 18; // I
        case 0b010011: return 19; // J
        case 0b010100: return 20; // K
        case 0b010101: return 21; // L
        case 0b010110: return 22; // M
        case 0b010111: return 23; // N
        case 0b011000: return 24; // O
        case 0b011001: return 25; // P
        case 0b011010: return 26; // Q
        case 0b011011: return 27; // R
        case 0b011100: return 28; // S
        case 0b011101: return 29; // T
        case 0b011110: return 30; // U
        case 0b011111: return 31; // V
        case 0b100000: return 32; // W
        case 0b100001: return 33; // X
        case 0b100010: return 34; // Y
        case 0b100011: return 35; // Z
        default: return 255; // Return 255 (or an
            error value) for invalid input
    }
}
```

B. Caesar Cipher Functions

Listing 2. Caesar Cipher Function

```
unsigned char ceaserCipher(unsigned char inputChar,
    unsigned char shift, unsigned char en_de)
{
    unsigned char result;

    if (en_de == 0) // Encoding
    {
        result = (inputChar + shift) % 36; // Shift
            forward and wrap around
    }
    else // Decoding
    {
        result = (inputChar + 36 - shift) % 36; //
            Shift backward and wrap around
    }

    return result;
}
```

C. Morse Code

Listing 3. Morse Code Encoding

```
char morseCodes[36][6] = {
    "-----", // 0
    ".----", // 1
    "..---", // 2
    "...--", // 3
    "....-", // 4
    ".....", // 5
    "-....", // 6
    "--...", // 7
    "---..", // 8
    "----.", // 9
    ".-",    // A
    "-...",  // B
    "-.-.",  // C
    "-..",   // D
    ".",     // E
    "..-.",  // F
    "--.",   // G
    "....",  // H
    "..",    // I
    ".---",  // J
    "-.-",   // K
    ".-..",  // L
    "--",    // M
    "-.",    // N
    "---",   // O
    ".--.",  // P
    "--.-",  // Q
    ".-.",   // R
    "...",   // S
    "-",     // T
    "..-",   // U
    "...-",  // V
    ".--",   // W
    "-..-",  // X
    "-.--",  // Y
    "--.."   // Z
};
```

D. Morse Code Encoding

Listing 4. Caesar Cipher Function

```c
void morseEncode(unsigned char inputChar)
{
    unsigned char i;
    unsigned char index = get7SegVal(inputChar); // Convert input to an index (0-35)
    if (index >= 36) return; // Invalid input, ignore

    // Traverse the Morse code for the character
    for (i = 0; i < 6; i++)
    {
        char symbol = morseCodes[index][i];
        if (symbol == '\0') break; // End of Morse code for this character

        if (symbol == '.')
        {
            LATD = 0x80; // Represent dot (e.g., LED ON)
            Delay_ms(500); // Duration for dot
        }
        else if (symbol == '-')
        {
            LATD = 0x08; // Represent dash (e.g., LED ON)
            Delay_ms(500); // Duration for dash
        }

        LATD = 0x00; // Turn off LED between signals
        Delay_ms(250); // Inter-element gap
    }

    Delay_ms(300); // Gap between characters
}
```

E. Morse Code Decoding

Listing 5. Caesar Cipher Function

```c
unsigned char getMorseCodeBinary(unsigned char index)
{
    unsigned char morseInput = 0;

    // Map the index to Morse code using binary patterns
    switch(index) {
        case 0: morseInput = 0b000000; break;  // 0
        case 1: morseInput = 0b000001; break;  // 1
        case 2: morseInput = 0b000011; break;  // 2
        case 3: morseInput = 0b000111; break;  // 3
        case 4: morseInput = 0b001111; break;  // 4
        case 5: morseInput = 0b011111; break;  // 5
        case 6: morseInput = 0b100000; break;  // 6
        case 7: morseInput = 0b100001; break;  // 7
        case 8: morseInput = 0b100011; break;  // 8
        case 9: morseInput = 0b100111; break;  // 9
        case 10: morseInput = 0b001000; break; // A
        case 11: morseInput = 0b100010; break; // B
        case 12: morseInput = 0b101000; break; // C
        case 13: morseInput = 0b101001; break; // D
        case 14: morseInput = 0b101010; break; // E
        case 15: morseInput = 0b101011; break; // F
        case 16: morseInput = 0b101100; break; // G
        case 17: morseInput = 0b101101; break; // H
        case 18: morseInput = 0b110000; break; // I
        case 19: morseInput = 0b110001; break; // J
        case 20: morseInput = 0b110010; break; // K
        case 21: morseInput = 0b110011; break; // L
        case 22: morseInput = 0b110100; break; // M
        case 23: morseInput = 0b110101; break; // N
        case 24: morseInput = 0b110110; break; // O
        case 25: morseInput = 0b110111; break; // P
        case 26: morseInput = 0b111000; break; // Q
        case 27: morseInput = 0b111001; break; // R
        case 28: morseInput = 0b111010; break; // S
        case 29: morseInput = 0b111011; break; // T
        case 30: morseInput = 0b111100; break; // U
        case 31: morseInput = 0b111101; break; // V
        case 32: morseInput = 0b111110; break; // W
        case 33: morseInput = 0b111111; break; // X
        case 34: morseInput = 0b000100; break; // Y
        case 35: morseInput = 0b000101; break; // Z
        default: morseInput = 255; break;  // Invalid Morse code
    }

    return morseInput;
}

unsigned char morseDecode(unsigned char morseInput)
{
    unsigned char i;
    unsigned char decodedChar;  // Default to 255 (invalid) if not found
    decodedChar = 255;

    // Traverse all the Morse code representations (36 characters)
    for (i = 0; i < 36; i++)
    {
        // Compare the 6-bit input with the corresponding binary pattern
        if (morseInput == getMorseCodeBinary(i))
        {
            decodedChar = i;  // Return the character index if a match is found
            break;  // Exit loop once the character is found
        }
    }

    return decodedChar;
}

}
```

F. Main Code

Listing 6. Main Function

```c
void main()
{
    unsigned int adc_value;  // Variable to store ADC result
    unsigned char inputChar;
    unsigned char en_de;
    unsigned char orig_char;
    unsigned char coded_char;
    ADCON1 = 0x0E;  // Configure AN0 as analog, others as digital
    TRISA = 0x01;   // Set RA0 as input
    TRISB = 0xFF;   // Configuring PORTB as input
    TRISC = 0x00;   // Set PORTC as output
    TRISD = 0x00;   // Set PORTD as output
    PORTC = 0x00;   // Clear PORTC
    while (1)
    {
        adc_value = ADC_Read(0);  // Read analog value from AN0 (RA0)
        inputChar = PORTB & 0x3F;
        en_de = PORTB & 0x40;
```

```c
        //Divide ADC range (0-1023) into 4 sections
        if (adc_value < 256)
        {
            Delay_ms(100);
            asm{
          BCF LATA, 1
          BCF LATA, 2
          }
            Delay_ms(100);
            orig_char = get7SegVal(inputChar);
            LATC = segmentValues[orig_char];

            coded_char = ceaserCipher(inputChar, 3, en_de);
            LATD = segmentValues[coded_char];
        }
        else if (adc_value < 512)
        {
            Delay_ms(100);
            asm{
          BCF LATA, 1
          BSF LATA, 2
          }
            Delay_ms(100);

            orig_char = get7SegVal(inputChar);
            LATC = segmentValues[orig_char];

            coded_char = ceaserCipher(inputChar, 5, en_de);
            LATD = segmentValues[coded_char];
        }
        else if (adc_value < 768)
        {
          Delay_ms(100);
          asm{
          BSF LATA, 1
          BCF LATA, 2
          }
          Delay_ms(100);

          if (en_de == 0) // Encoding mode
          {
              orig_char = get7SegVal(inputChar); // Get the original character index (0-35)
              LATC = segmentValues[orig_char];  // Display original character on PORTC
              morseEncode(inputChar); // Encode the character into Morse code
          }
          else // Decoding mode
          {
              // Decode Morse code received from PORTB as a 6-bit binary input
              LATC = 0x00;
              coded_char = morseDecode(inputChar); // Decode the Morse code

              if (coded_char != 255) // If valid decoded character
              {
                  LATD = segmentValues[coded_char]; // Display decoded character on PORTD
              }
              else
              {
                  LATD = 0x00; // Invalid Morse code, display nothing
              }
          }
        }
        else
        {
            Delay_ms(100);
            asm {
            BSF LATA, 1
            BSF LATA, 2
            }
            Delay_ms(100);

            if (en_de == 0) // Encoding mode
            {
                  // Step 1: Get the original character index
                  orig_char = get7SegVal(inputChar);
                  LATC = segmentValues[orig_char]; // Display original character on PORTC
                  Delay_ms(100);

                  // Step 2: Apply Caesar cipher (e.g., shift by 3)
                  coded_char = ceaserCipher(orig_char, 3, 0); // Encode with Caesar cipher

                  // Step 3: Encode the shifted character into Morse code
                  morseEncode(coded_char); // Display Morse code via LEDs
              }
              else // Decoding mode
              {
                  LATC = 0x00;
                  Delay_ms(100);
                  // Step 1: Decode Morse code received from PORTB
                  coded_char = morseDecode(inputChar); // Decode Morse input
                  if (coded_char != 255) // Valid Morse code decoded
                  {
                      // Step 2: Reverse the Caesar cipher (e.g., shift back by 3)
                      orig_char = ceaserCipher(coded_char, 3, 1); // Decode with Caesar cipher

                      // Step 3: Display the decoded original character on PORTD
                      LATD = segmentValues[orig_char];
                  }
                  else
                  {
                      LATD = 0x00; // Invalid Morse code, display nothing
                  }
              }
         }


        Delay_ms(100);  // Add a small delay for stability
    }
}
```

VIII. Challenges

- **Pulldown Resistors:** Each value at the inputs should be properly pulled down. This is done so that the inputs when set to high are properly set to high.
- **MCLR Pin:** The MCLR Pin is the first pin of the PIC18F452. This needs to be properly pulled up with a resistor between the 5V rail and the pin. This is done because otherwise the microcontroller would go into a constant reset state.
- **Crystal Oscillator:** The crystal oscillator should be set as close as possible to the 13 and 14 pins of the microcontroller. If it is too far away, it will not oscillate properly.
- **Corrupted Hex Files:** The MPLAB IDE (Both X and Standard), have the tendency to produce corrupted hex files. These hex files run on the simulator as the simulated PIC in Proteus but not on hardware, as the hex file corruption leads to corruption of memory. Thus, mikroC was utilized to create the hex file, and the PICKit3 programming software.

IX. YouTube Link

A demonstration of the project can be viewed at the following YouTube link:

- **Project Demonstration:** Click here to watch the video.

X. Conclusion

The project successfully implemented multiple encoding and decoding schemes using the PIC18F452 microcontroller, demonstrating integration of hardware and software for practical applications.

References