Development of a Microprocessor-Based Telegraph

Ziyao Xiong

***Abstract*—This project implements an interactive telegraph system for encoding and decoding Morse code, developed using assembly language. A PIC18 microprocessor is integrated with a push button, an LCD, and a keypad. Users input Morse sequences via the push button, where short and long presses represent dots (.) and dashes (-), respectively. An indicator helps distinguish between press durations: if it flashes, the press is considered long; otherwise, it is short. The sequences are displayed on the LCD in real time. Pressing 'D' on the keypad decodes the stored Morse sequence using a predefined lookup table, displaying the corresponding characters or numbers on the LCD, while pressing 'C' clears the patterns. Extensive testing demonstrated 100% decoding accuracy for press durations between 0.1 s and 267 s. The delay for both pattern display and the indicator's flashing were within the limits of human reaction times, demonstrating the system's reliability and practicality for real-world use.**

I. Introduction

MORSE code, invented by Samuel Morse in the 1830s, revolutionized long-distance communication by encoding messages into dots (.) and dashes (-), transmitted as binary-like electrical pulses over telegraph lines [1]. Its simplicity made it a cornerstone of 19th and early 20th-century global communication. However, Morse code's reliance on skilled operators for manual encoding and decoding required extensive training, limiting both accessibility and transmission speed [2].

This project modernizes Morse code using the PIC18F87K22 microcontroller to automate encoding and decoding, addressing Morse code's historical challenges. With built-in flash memory, random-access memory (RAM), and timer modules, the PIC18 efficiently stores Morse patterns and distinguishes short and long presses via precise timing mechanisms. Peripherals such as a push button for input, an LCD for real-time display, and a keypad for user commands ensure seamless interaction. Users can input Morse sequences using the push button, with an indicator providing feedback on input duration. The stored patterns are decoded when the 'D' key is pressed, and result is displayed on the LCD. The system offers an intuitive interface, allowing users to explore Morse code without prior training, demonstrating its principles interactively and efficiently.

This exploration highlights the potential of modern embedded systems to breathe new life into historical tools. By providing a user-friendly, hands-on platform for engaging with Morse code, the project bridges the gap between past and present, fostering a deeper appreciation for the technologies that shaped global communication and ensuring their legacy and ingenuity remain valued by new generations.

II. High Level Design

The system is built around the PIC18F87K22 microcontroller, which serves as the central processing unit. This microcontroller is equipped with 128 KB of Flash memory [3], which is used to store the predefined Morse code patterns for the 26 letters of the alphabet and 10 numerical digits. These patterns remain permanently available for decoding operations. The 4 KB of RAM is utilized for temporary data handling, including storing user-input Morse patterns, intermediate results, and other variables required for real-time processing. The microcontroller's versatile I/O ports interface the system's peripherals with its internal data bus. The push button is connected to Pin 0 of PORTJ, configured for digital input to detect user presses. The keypad utilizes PORTE to capture user commands such as decoding or clearing patterns, while the LCD is connected through the output pins of PORTB to display real-time input patterns and decoded results. Additionally, an indicator, controlled via PORTD, provides visual feedback by flashing to distinguish between short and long presses, ensuring intuitive user interaction. The system also relies on the PIC18's Timer0 to time input durations. Fig. 1 illustrates the system's high-level structure, detailing component connections and data flow. The high-level top-down modular diagram is shown in Fig.7.

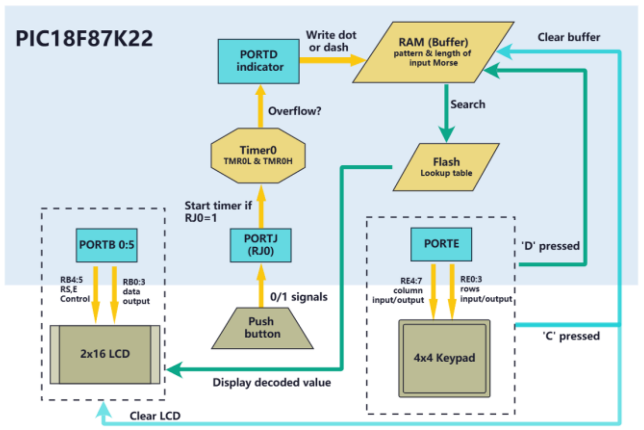

Fig.1. High-level design of the telegraph system. Dark green blocks represent the peripherals, blue blocks indicate the input data bus pins of the ports, and yellow blocks highlight the built-in functionalities and data space of the PIC microcontroller. Data flow is depicted with directional arrows.

III. Software And Hardware Design

A. *Push Button*

The RJSGME1-Y-12V-D [4] push button serves as the primary input device for Morse code signal generation in the system. It connects to the PIC18 via the RJ0 pin. Users input Morse codes by pressing the button for a short duration to represent dots or a longer duration to represent dashes. The button's COM terminal is connected to a 5V source, while its NO terminal is connected to RC0 and grounded through a 10 kΩ pull-down resistor, as shown in Fig. 2(b). RJ0 is selected since PORTJ pins are designed with built-in Schmitt Trigger buffers [5]. Compared to traditional Transistor-Transistor Logic (TTL) buffers used by many other ports, ST buffers offer superior noise immunity and ensure clean, stable state transitions. This eliminates the need for additional debouncing circuitry, simplifying the circuit design [6].

Configured as a digital input, RJ0 detects the button's state based on the microcontroller's defined input high voltage (VIH) threshold, which is $VIH \geq 0.8\times Vdd$ [7], where Vdd is the microcontroller's supply voltage. With Vdd=5 V, the VIH threshold is 4V. Connecting the button's COM terminal to 5 V ensures the pin reliably detects a logic-high signal (1) when the button is pressed, as the input voltage exceeds the VIH threshold. In practice, the finite input impedance of microcontroller pins (typically 10 MΩ [8]) and stray capacitance can cause voltage instability when the pin is left floating, leading to unreliable logic detection. A pull-down resistor addresses this issue by providing a low-resistance path to ground, stabilizing the pin at a logic-low (0) when the button is not pressed. The 10 kΩ pull-down resistor is chosen for its significantly lower resistance compared to RJ0's input impedance, ensuring RJ0 receives only ~1/1000 of the resistor's voltage (Kirchhoff's law) and maintains a stable logic-low when the button is unpressed. Additionally, it draws only 500 μA at 5 V, ensuring low power consumption. Thus, This design guarantees both stability and efficiency.

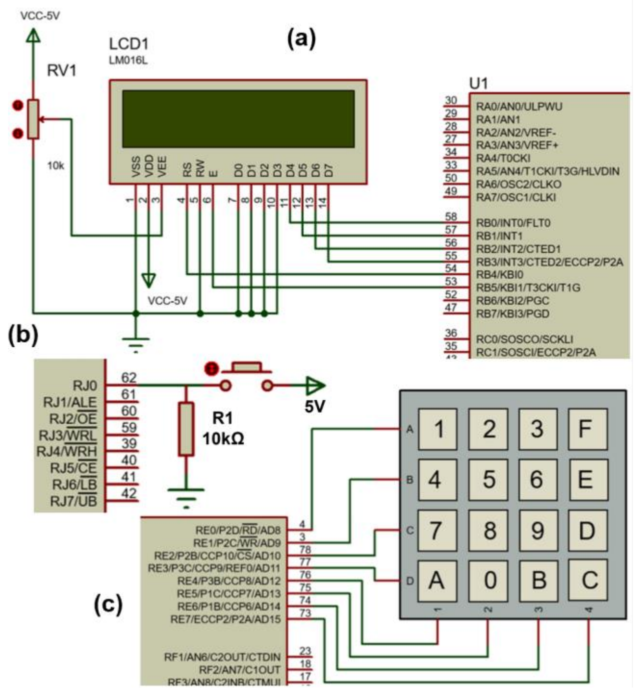

Fig. 2. Hardware configuration of the circuit. The diagram zooms in on the peripherals connected to the PIC18. Ports not shown in the figure are unconnected. (a): LCD connected to PORTB. (b): Push button connected to RJ0. (c): Keypad connected to PORTE.

B. *Timer*

Historically, Morse code follows a 1:3 timing ratio, where dots (T) and dashes (3T) are differentiated by their relative durations [9]. Considering the average human reaction time of 200–250 ms [10], a short press (dot) naturally falls in the range of 350–500 ms, while a long press (dash) spans approximately 1.05–1.50 seconds. By setting the threshold approximately at 1 second, the design maintains the historical 1:3 ratio while providing users adequate time to differentiate between dots and dashes without requiring excessive precision, ensuring both usability and reliability.

In the PIC18 microcontroller, there are 3 internal oscillators designed to count clock cycles or external events [11]. These oscillators increment their count register on each clock cycle, and when the count exceeds its maximum value, an overflow occurs, which can trigger an interrupt. In this design, Timer0 was selected for measuring the duration of button presses due to its versatility and simplicity. Timer0 is a general-purpose timer capable of operating in both 8-bit and 16-bit modes [12], along with prescaler options to adjust the counting rate. We configured it in 16-bit mode; thus, it offers a counting range of 2^{16}=65,5362 ticks, minimizing interrupt frequency compared to 8-bit mode (2^8=256 ticks). A 1:256 prescaler is selected, providing an increment interval of 16 μs, which results in an overflow time of:

$$t_{overflow} = 16\,\mu s \cdot 2^{16} = 1.0485s. \qquad (1)$$

The timer starts when RJ0 is logical high and stops when RJ0 is logical low. If the button press duration exceeds $t_{overflow}$, an interrupt flag is set, causing the microcontroller to suspend execution and jump to the Interrupt Vector at address 0x08 and execute the Interrupt Service Routine (ISR) [13]. In ISR, the interrupt flag is cleared, and the overflow count is incremented. The microcontroller then resumes the main program from the interrupted point.

We notice that $t_{overflow}$ is approximately 1s, making it a practical choice for the threshold. In 16-bit mode, Timer0 stores count in two 8-bit registers, TMR0H (high byte) and TMR0L (low byte). Reading the timer value requires combining these registers and handling potential carry-over. To simplify, our design avoids explicit use of timer registers and instead counts overflow events. A short press is recognized if the overflow count remains 0, while a long press is detected when the overflow count exceeds 1. The overflow count is stored in a single byte (8 bits), supporting up to 255 counts, enabling detection of press durations up to $255\times t_{overflow}=267s$ under this configuration. To provide intuitive feedback for distinguishing dots and dashes, the overflow count is displayed on PORTD, which is connected to 8 LEDs representing its 8 bits. The LEDs update as the overflow count increments, creating a visible "jump" in the LED pattern. Users can hold the button until the LED pattern changes on PORTD to confirm a long press.

To store the input pattern, we allocate 5 bytes of RAM for the Pattern buffer and 1 byte for Length. To store Morse code inputs, dots are encoded as 0x01 and dashes as 0x02 in the Pattern buffer. Since PIC18 does not support direct access to individual bytes within the buffer, we utilize File Select Registers (FSR) for indirect addressing. PIC18 features FSR0 and INDF0 (Indirect File Register 0), where FSR0 holds the memory address of the desired byte, and INDF0 allows read and write operations at the address specified by FSR0. During each input, Length value is incremented by 1 and FSR0 is updated to point to the starting address of the Pattern buffer plus the current Length value. We then write 0x01 or 0x02 to INDF0 to write new input bytes sequentially into the buffer. The full logic of software implementation of the timing process is shown in Fig.3(a).

A delay is necessary to debounce button inputs and ensure reliable signal detection, as mechanical contact bounce can cause unintended signal fluctuations [14]. Without debouncing, these fluctuations may be misinterpreted as multiple presses. Testing indicates that a delay of 10-30ms is optimal for effective debouncing. The software implementation of the delay is shown in Fig. 3(b). We designed a nested loop structure with both the outer and inner loops counting down from the maximum 8-bit value (0xFF, 255), resulting in a total of 255^2=65,025 instruction cycles. The duration of one instruction cycle (T_{cycle}) is determined by the crystal oscillation frequency (f_{osc}) of the EasyPIC Pro V7 board, which is 16MHz [15]:

$$T_{cycle} = \frac{4}{f_{osc}} = \frac{4}{16{,}000{,}000} = 0.25\mu s, \qquad (2)$$

The numerator 4 exists because each instruction generally requires four phases to complete (fetch, decode, execute, and write-back). The total delay (T_{delay}) is therefore given by:

$$T_{delay} = T_{cycle} \cdot 65025 \approx 16.3ms, \qquad (3)$$

which falls in the range of the optimal delay times.

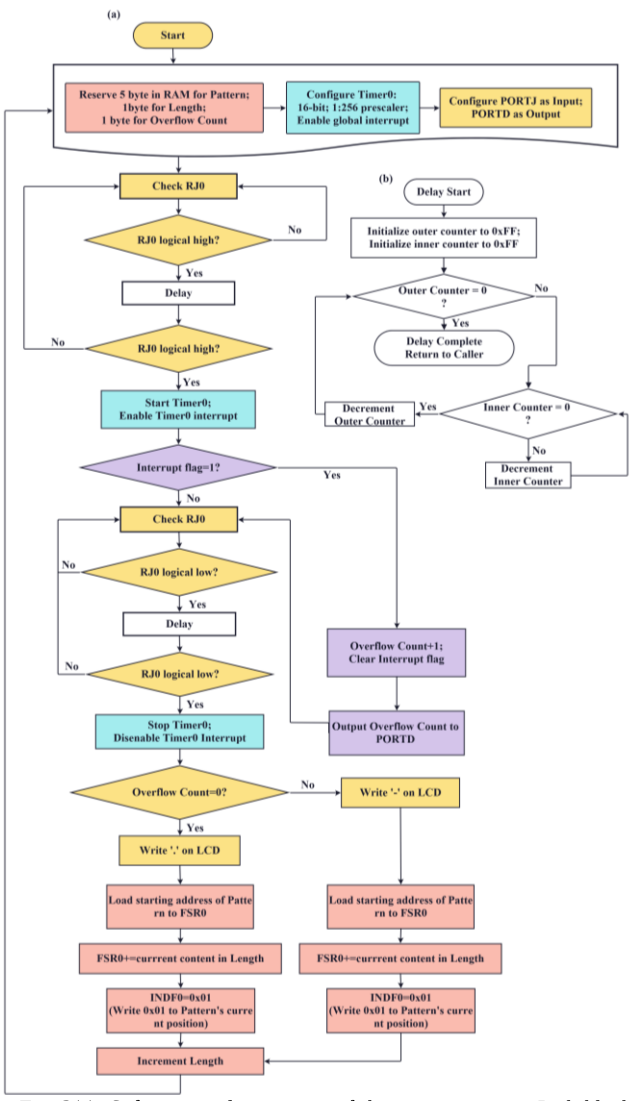
(a)
Start
Reserve 5 byte in RAM for Pattern; 1byte for Length; 1 byte for Overflow Count
Configure Timer0: 16-bit; 1:256 prescaler; Enable global interrupt
Configure PORTJ as Input; PORTD as Output
Check RJ0
RJ0 logical high?
No
Yes
Delay
RJ0 logical high?
No
Yes
Start Timer0; Enable Timer0 interrupt
Interrupt flag=1?
Yes
No
Check RJ0
RJ0 logical low?
No
Yes
Delay
RJ0 logical low?
No
Yes
Stop Timer0; Disenable Timer0 Interrupt
Overflow Count+1; Clear Interrupt flag
Output Overflow Count to PORTD
Overflow Count=0?
No
Yes
Write '-' on LCD
Write '.' on LCD
Load starting address of Pattern to FSR0
FSR0+=currrent content in Length
INDF0=0x01 (Write 0x01 to Pattern's current position)
Increment Length
(b)
Delay Start
Initialize outer counter to 0xFF; Initialize inner counter to 0xFF
Outer Counter = 0 ?
No
Yes
Delay Complete Return to Caller
Decrement Outer Counter
Yes
Inner Counter = 0 ?
No
Decrement Inner Counter

Fig. 3(a): Software implementation of the timing process. Pink blocks represent operations involving Pattern/Length buffer. Cyan blocks indicate Timer0-related processes, purple blocks show the ISR, and yellow blocks represent other regular processes. Fig. 3(b): The debounce delay process.

C. Decode

A lookup table is stored in Flash memory to define Morse code patterns and lengths for 26 letters (A–Z) and 10 numbers (0–9). Flash memory is used for its larger capacity and non-volatile storage. Each character requires 1 byte for length, up to 5 bytes for the pattern (0x01 or 0x02), and 1 byte for the ASCII value. Accessing the table uses the TBLPTR register, which consists of three parts: TBLPTRU (upper high byte), TBLPTRH (high byte), and TBLPTRL (low byte). These collectively address the 24-bit program memory space. The table pointer is set to the required address, and the TBLRD instruction transfers data from Flash memory at the specified location into the TABLAT register for processing. When incrementing TBLPTR, carry propagation across the three registers must be handled, as shown in Fig. 5(b).

During decoding, the lookup table enables efficient comparisons by first matching the length of the input sequence and then verifying the Morse code pattern byte by byte. Any mismatch skips the current symbol and jumps to the next entry, reducing computational overhead. A variable bit_count determines how many bytes to skip. It is initialized to the symbol's length (stored as the first byte of the current entry). If the lengths differ, the system skips bit_count+2 bytes to the next entry. If the lengths match, the system increments TBLPTR to compare patterns byte by byte. For each matching byte, bit_count is decremented. If a pattern mismatch occurs, the system skips bit_count+1 bytes to the next entry. Fig. 4 illustrates the skipped bytes in the table structure. An end-of-table marker, 0x06, is placed at the end of the table. This value is unique as lengths cannot exceed 0x05, patterns only contain 0x01 and 0x02, and ASCII values do not include 0x06. If 0x06 is detected, it indicates the table has been fully searched with no match, rendering the input invalid. In such cases, a '?' is displayed on the LCD. If a match is found, the decoded symbol is displayed. The full software implementation of decoding is shown in Fig.5.

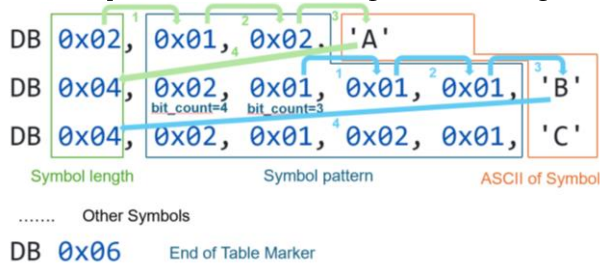

Fig. 4: Snapshot of the lookup table. Assuming an input pattern "C". Green arrows indicate a mismatch when comparing the first byte in the "A" entry (0x02). Here, bit_count is set to 0x02, and 4 bytes are skipped to the first byte in the "B" entry (bit_count+2). Blue arrows show a mismatch at the 4th byte in "B", with bit_count at 3. 4 bytes are skipped to the first byte in "C" (bit_count+1). After each skip, we check for the end-of-table marker (0x06).

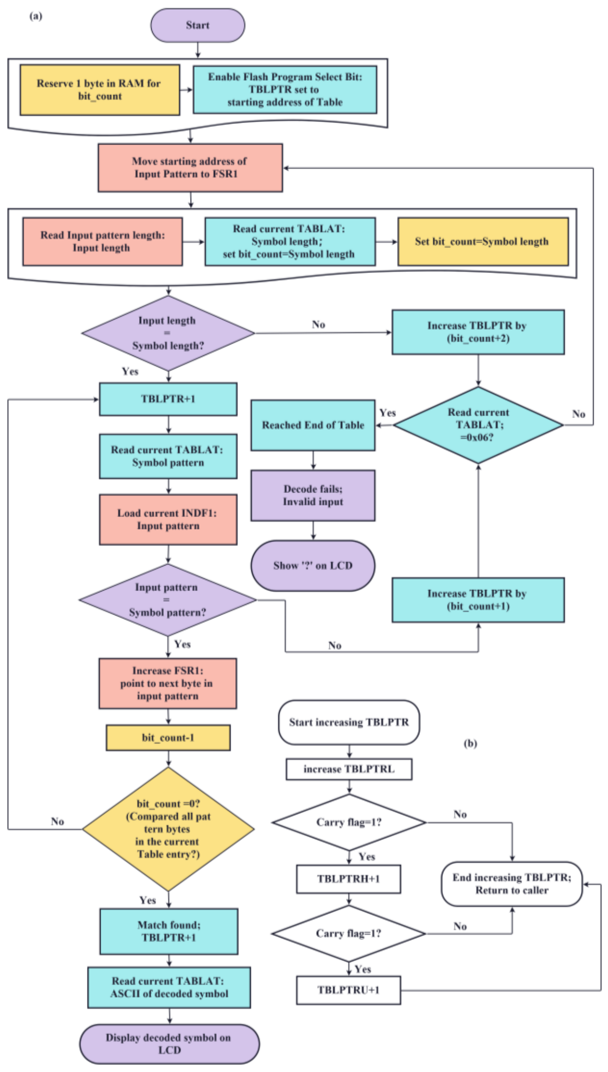
(a)
Start
Reserve 1 byte in RAM for bit_count
Enable Flash Program Select Bit: TBLPTR set to starting address of Table
Move starting address of Input Pattern to FSR1
Read Input pattern length: Input length
Read current TABLAT: Symbol length; set bit_count=Symbol length
Set bit_count=Symbol length
Input length = Symbol length?
No
Yes
Increase TBLPTR by (bit_count+2)
TBLPTR+1
Reached End of Table
Yes
Read current TABLAT; =0x06?
No
Read current TABLAT: Symbol pattern
Decode fails; Invalid input
Load current INDF1: Input pattern
Show '?' on LCD
Increase TBLPTR by (bit_count+1)
Input pattern = Symbol pattern?
Yes
No
Increase FSR1: point to next byte in input pattern
bit_count-1
bit_count =0? (Compared all pattern bytes in the current Table entry?)
No
Yes
Match found; TBLPTR+1
Read current TABLAT: ASCII of decoded symbol
Display decoded symbol on LCD
(b)
Start increasing TBLPTR
increase TBLPTRL
Carry flag=1?
No
Yes
TBLPTRH+1
End increasing TBLPTR; Return to caller
Carry flag=1?
No
Yes
TBLPTRU+1

Fig. 5(a): Software design of decoding. Pink blocks show operations involving Pattern/Length buffer, cyan blocks indicate table-related processes, yellow blocks show bit_count operations, and purple blocks represent other processes. Fig. 5(b): Incrementing TBLPTR with carry handling.

D. Keypad

The system utilizes a 4x4 matrix keypad connected to PORTE of the PIC18, as shown in Fig. 2(c). Pressing key 'D' initiates the decoding process, while key 'C' triggers a routine to clear the LCD and reset the input Pattern buffer, managed via the FSR2 register, as shown in Fig. 6(b).

When 'D' is pressed, the corresponding row is RE2 and the column is RE7. To detect the column, the row pins (RE0–RE3) are configured as outputs with their LATE register cleared to 0. The LATE register directly controls the output levels of PORT E pins when configured as outputs, ensuring all row pins are driven low. The column pins (RE4–RE7) are configured as inputs with internal pull-ups enabled, which pull them high when no signal is present. Pressing 'D' connects RE2 to RE7, causing RE7 to detect a low signal, resulting in a PORTE value of 01110000. For row detection, the column pins are configured as outputs and driven low, while the row pins are configured as inputs. RE2 detects a low signal, resulting in a PORTE value of 00001011. The inclusive OR of these values produces 01111011, identifying 'D'. Similarly, key 'C' produces 01110111. The complete software design for key detection is shown in Fig. 6(a).

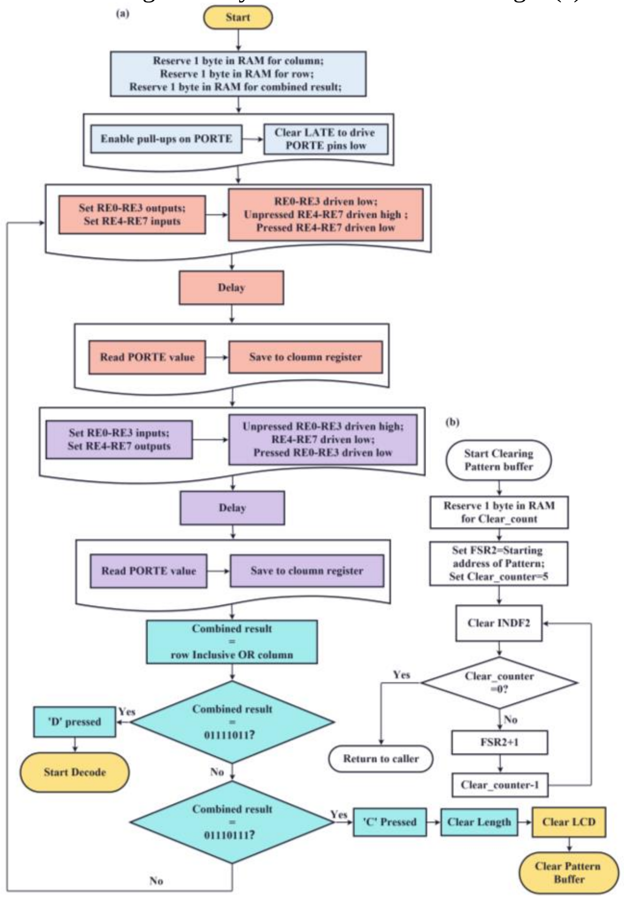
(a)
Start
Reserve 1 byte in RAM for column;
Reserve 1 byte in RAM for row;
Reserve 1 byte in RAM for combined result;
Enable pull-ups on PORTE
Clear LATE to drive PORTE pins low
Set RE0-RE3 outputs;
Set RE4-RE7 inputs
RE0-RE3 driven low;
Unpressed RE4-RE7 driven high ;
Pressed RE4-RE7 driven low
Delay
Read PORTE value
Save to cloumn register
Set RE0-RE3 inputs;
Set RE4-RE7 outputs
Unpressed RE0-RE3 driven high;
RE4-RE7 driven low;
Pressed RE0-RE3 driven low
Delay
Read PORTE value
Save to cloumn register
Combined result = row Inclusive OR column
Combined result = 01111011?
Yes
'D' pressed
Start Decode
No
Combined result = 01110111?
Yes
'C' Pressed
Clear Length
Clear LCD
Clear Pattern Buffer
No
(b)
Start Clearing Pattern buffer
Reserve 1 byte in RAM for Clear_count
Set FSR2=Starting address of Pattern;
Set Clear_counter=5
Clear INDF2
Clear_counter =0?
Yes
Return to caller
No
FSR2+1
Clear_counter-1

Fig. 6(a): Software design of the keypad. Pastel blue blocks represent setup processes, pink blocks and purple blocks indicate column and row detection processes, respectively, cyan blocks identify the pressed key, and yellow blocks link to external routines defined elsewhere. Fig. 6(b): Process for clearing the Pattern buffer.

E. LCD (Liquid Crystal Display)

A 2x16 character LCD is connected to PORTB of the PIC18 microcontroller in 4-bit data mode, as shown in Fig. 2(a). Data and commands are transferred via RB0–RB3 in two nibbles (high nibble followed by low nibble). The RS (Register Select) line specifies the input type: RS=0 for commands and RS=1 for data, while the E (Enable) line, connected to RB5, is pulsed high to latch inputs [16]. The VEE pin, connected to a potentiometer, adjusts display contrast for optimal readability [17].

The LCD uses two key registers: the Instruction Register (IR) and the Data Register (DR) [18]. The IR processes commands to control the LCD's behavior. For instance, pressing 'C' on the keypad sends the 0x01 command to the IR to clear the screen and reset the cursor to row 1, column 1. The DR holds data to be displayed. The LCD interprets any 8-bit value sent to the DR as an ASCII code and writes it to the Display Data RAM (DDRAM) at the current cursor address [19]. Each DDRAM location corresponds to a specific screen position, and the character represented by the ASCII code stored in DDRAM is displayed on the screen at the current DDRAM address. The Entry Mode Set command configures the DDRAM address to automatically increment after each write, simplifying sequential updates.

The LCD_Send_Byte_D function writes both Morse code patterns (dots and dashes) and decoded characters to the LCD. It transmits an 8-bit value to the DR by splitting it into high and low nibbles and sending them sequentially via RB0–RB3. The RS line is set high (RS=1) for data writes, and the E line is pulsed to latch each nibble. This implementation is based on code provided by Dr. Mark Neil [20]. With this configuration, input dots and dashes are displayed in real-time on the first row of the LCD as they are entered. When the 'D' key on the keypad is pressed, the decoded symbol replaces the pattern on the screen.

F. High Level Software Design

Combining all software implementation mentioned, a top-down modular diagram showing the high-level software design of this system is shown in Fig.7.

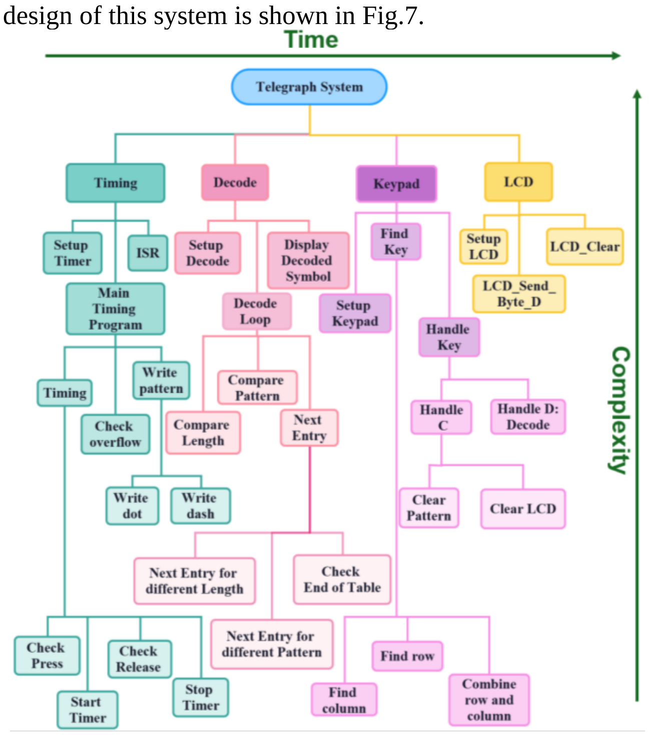

Fig. 7. High-level software design of the system. The diagram illustrates a top-down flow where complexity decreases as the granularity of operations increases. The vertical axis represents complexity, while the horizontal flow generally corresponds to the time sequence of module execution.

IV. Results And Performance

A. System Stability and Functionality

1) Decoding Accuracy

The system achieved 100% decoding accuracy in recognizing and interpreting Morse code patterns. Testing involved five participants inputting 50 predefined sequences, covering all letters (A-Z), numbers (0-9), and incorrect patterns that displayed a '?' on the LCD. Inputs were verified against a lookup table, and the 'C' key successfully cleared incorrect entries for re-input. All decoded outputs matched expected results, confirming the system's reliability for real-time decoding. Further tests verified LCD accuracy by comparing displayed characters with corresponding MPLAB register values, achieving 100% accuracy across 50 trials. The 'C' key's ability to clear the Pattern buffer and Length bytes was also confirmed with a 100% success rate. These results demonstrate the system's robustness and suitability for real-time user feedback applications.

2) *Input Accuracy*

To assess the accuracy of user input, tests were conducted to determine whether the Morse code patterns entered by participants matched their intended inputs.

a) Standard Conditions

A total of 50 tests were conducted under controlled conditions, with firm button presses ensuring reliable contact. Successive presses were spaced ~2 seconds apart to simulate realistic input rhythms. Short and long presses were held for ~0.5s and ~1.5s, respectively, adhering to Morse code timing standards. The system achieved 100% accuracy, confirming its precision in realistic input scenarios.

b) Extreme Conditions

(1) Light and Slow Presses

This is the scenario where users apply minimal pressure on the button and press it slowly (>1s). This behavior can lead to contact bouncing, where the mechanical contacts of the button momentarily open and close multiple times as the button is pressed or released. In 50 tests, 7 cases resulted in multiple dots appearing on the LCD during a single button press, yielding an accuracy of 86%.

(2) Quick Presses and Very Short Intervals

In the Quick Press scenario, where participants released the button as quickly as possible, 50 tests were conducted, with 5 instances of no characters appearing on the LCD. This was likely due to the press duration being shorter than the debounce delay time, causing the input to be ignored, resulting in 90% accuracy. In the Very Short Intervals scenario, where users pressed the button almost immediately after releasing it, 50 tests yielded 2 instances of no character display, likely because the debounce delay time had not elapsed. This scenario achieved an accuracy of 96%.

(3) Long Presses

For press durations under 267s, the system maintained 100% accuracy. Beyond this, accuracy dropped to 0% due to the overflow of Overflow_count register, which aligns with design constraints and is irrelevant in practical scenarios.

(4) Buffer Overflow Handling

When entering patterns exceeding the 5-byte Pattern buffer, the LCD correctly displayed characters, and decoding output was '?' as expected. Inspection of MPLAB registers revealed that excess inputs were stored in adjacent RAM regions, which fortuitously did not affect critical RAM operations. Despite this, it presents a potential risk in extended use cases.

For input patterns exceeding 16 characters, the LCD failed to display characters on the second row. This limitation occurred because the system did not implement functionality to update DDRAM for handling longer patterns.

3) *Timing Accuracy*

In our design, the PORTD LED indicator is configured to toggle with each timer overflow, corresponding to an interval of 1.05 seconds. To verify the actual timing, we conducted experiments by recording the LED indicator using a 30 frame per second (fps) camera during a 12-second continuous button press. The recordings were analyzed in Adobe Premiere Pro (PR) to measure the exact frame intervals between LED toggles. The relationship between frames and seconds is given by the formula:

$$t = \frac{number\ of\ frames}{fps}. \tag{4}$$

We repeated the experiment 5 times, results were averaged and plotted in Fig.8, with uncertainties calculated as:

$$uncertainty = \frac{max - min}{2}. \tag{5}$$

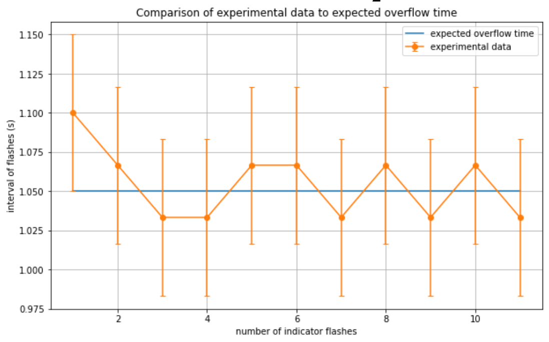

Fig. 8. Experimental Results. The orange data points represent the averaged values of the same flash order across five experiments, while the blue line indicates the expected 1.05s overflow time.

The ~0.05s deviation in the first interval is attributed to debounce delay before the system initiates overflow counting. While slightly exceeding the calculated value in Equation (3), this deviation is linked to the larger error margin of the first data point. From the second flash onward, intervals stabilize at the expected 1.05s. However, the variance in flashing intervals does not affect accuracy. Observations in MPLAB registers confirm that PORTD updates within one instruction cycle (~0.25 µs) after an overflow event. Given that human reaction times exceed 200 ms (10^6 of T_{cycle}), releasing the button after the PORTD change guarantees accurate recognition of a dash. Therefore, as long as users rely on the indicator's state to determine short or long presses, rather than timing manually, the system will reliably register inputs.

Theoretically, releasing the button before the indicator flashes could result in a dot being misinterpreted as a dash if the 0.16 ms debounce delay causes the timing to exceed 1.05 seconds. In 30 edge-case tests, all attempts successfully registered as dots, indicating negligible practical impact. To ensure reliability, users are advised to press for 300–1000 ms for dots and rely on the indicator for dashes, ensuring durations exceed 1.15 seconds in worst-case scenarios.

B. *Real-Time Interaction*

To evaluate real-time interaction, we measured the delay from user input to visual feedback on the LCD. Specifically, the time from releasing the button to the pattern appearing, and from pressing the keypad to the decoded pattern being displayed, were recorded. The delay primarily results from the LCD program's design, which incorporates deliberate timing to ensure stable operation. Across five experiments, the average delay for both cases was 12 ± 2 frames, equivalent to 0.4 ± 0.07 seconds. This delay is comparable to human reaction time, and while it may be slightly perceptible, it remains well within tolerable limits for seamless interaction.

C. *Resource Utilization*

The resource utilization of PIC18 demonstrates highly efficient use of both flash and RAM. The program memory usage was 1,184 bytes, corresponding to only 1% utilization, leaving 99% available for future enhancements. Similarly, the data memory usage was 27 bytes, utilizing just 1% of the total available 3,862 bytes.

V. Updates, Modifications and Improvements

A. *Input Accuracy*

Based on input accuracy tests, the system achieves 100% accuracy under realistic conditions but faces limitations under extreme scenarios due to reliance on fixed debounce delays. Longer delays effectively filter out bouncing in Light and Slow Pressing but risk ignoring rapid inputs in Quick Presses or Very Short Intervals. Conversely, shorter delays improve responsiveness but increase susceptibility to false signals caused by bouncing. This trade-off highlights the limitations of relying solely on fixed delays.

Schmitt trigger addresses this challenge by converting noisy, bouncing inputs into clean digital transitions [23]. Its hysteresis mechanism defines distinct high and low voltage thresholds, switching states only when the input crosses these thresholds. By connecting the push button and RJ0 to input and output of the Schmitt trigger respectively, the signal is stabilized, eliminating bouncing effects at the hardware level.

B. *Overflow Handling*

The handling of long press durations and buffer overflows requires further improvement. For instance, the Overflow_count register could be expanded to more than one byte, allowing carry-over for prolonged presses. Although such scenarios are rare in practical use, this modification ensures robustness against unintended overflow events.

Similarly, the Pattern buffer could include safeguards to prevent data from spilling into adjacent RAM regions. One effective method is to implement bounds checking in the code, where every write to the buffer is validated against its defined size before proceeding. Alternatively, a circular buffer mechanism could be adopted, ensuring that new data overwrites the oldest data instead of exceeding the allocated memory. These measures mitigate potential risks, such as corruption of critical RAM regions, which, while not problematic in the current design, could pose issues in extended or more complex use cases.

C. *LCD Display*

At present, characters appear sequentially on the first line of the LCD, with no functionality to display text on the second line. Additionally, the decoded pattern overwrites the dot-dash input on the LCD, making the display less intuitive for users. To address these issues, the LCD's DDRAM can be programmed to control character positions more flexibly. For example, decoded patterns could be displayed after the input pattern on the same line or moved to the second line. This enhancement would preserve the input pattern for reference while showing the decoded result in a more user-friendly manner. By utilizing the full capabilities of the DDRAM, the system can offer a clearer and more intuitive display, improving the overall user experience.

D. *Push Button's Hardware Setup*

The pushbutton is currently mounted on a breadboard, connected via DuPont wires. This configuration lacks mobility and stability, as users must hold the setup steady during operation. Additionally, the DuPont wires' inductance and stray capacitance from the breadboard contribute to signal instability [24]. To address these issues, the system can be improved by replacing the current breadboard setup with a soldered connection. The need for DuPont wires would be eliminated, ensures stable and secure connections, enhancing the reliability of signal transmission.

In the future, the system can evolve into a fully independent device by integrating all components onto a single PCB with a built-in microcontroller that directly stores the program. A dedicated power source, such as a battery module, would enable standalone operation, while a custom enclosure would enhance portability and usability. Additionally, programming ports can be retained for future updates, ensuring the system remains adaptable and scalable for extended applications.

VI. Conclusion

This project successfully implements a real-time telegraph system using the PIC18F87K22 microcontroller. Users input Morse sequences via a pushbutton, with dots and dashes distinguished by press duration, and view both the input and decoded results on an LCD. Testing demonstrated 100% decoding accuracy under realistic conditions, showcasing efficient resource utilization and seamless user interaction supported by visual indicators and precise timing mechanisms. While timing variations were observed due to debounce delays, the system remains reliable as long as users follow the indicator's state for short and long presses. For the most conservative operation, dots should be pressed for 300–1000 ms, while dashes should exceed 1.15 seconds.

However, the project also identified areas for improvement, particularly under extreme conditions. Proposed modifications include integrating a Schmitt trigger for signal stability, expanding the Overflow_count register, and implementing bounds checking or circular buffers for safer memory management. Enhancements such as programming the LCD's DDRAM for better usability and transitioning to soldered or PCB-based connections would further improve the system's reliability and user experience.

Product Specification

Product Name: A Microprocessor-Based Telegraph System.

System Capabilities: This project implements a real-time telegraph system, allowing users to input Morse sequences via a pushbutton and interact with a 4x4 keypad for decoding and clearing commands. The system distinguishes dots and dashes based on press duration and displays both the input pattern and decoded results on an LCD.

Processor Peripherals: internal timer (Timer0), interrupts, and GPIO of the PIC18F87K22 microcontroller.

External Devices: a 16x2 LCD (on the EasyPIC PRO v7 Board), a pushbutton (RJSGME1-Y-12V-D) connected to a 10MΩ pull-down resistor, a 4x4 keypad, and a Global Protoboard PB-503 for prototyping additional connections.

Code Size: 1184 bytes (1% of flash memory)

Memory Usage: 27 bytes (1% of total RAM)

Operating Voltage: 5V, with the PIC18F87K22 and pushbutton directly powered by this voltage.

Speed: Handles input processing and updates the display within 0.4 seconds.

Accuracy: 100% accuracy in trial tests under realistic conditions and over 90% accuracy under extreme conditions.

REFERENCES

[1] Department of the Army, International Morse Code (Instructions), Washington, DC, USA: U.S. Government Printing Office, 1968, pp. 6–7.

[2] CIA, "The Art of Radio-Telegraphy: Proficiency Requirements and Operational Challenges," Central Intelligence Agency, Document No. CIA-RDP79-01578A000200080030-8. [Online]. Available: https://www.cia.gov/readingroom/docs/CIA-RDP79-01578A000200080030-8.pdf. [Accessed: Dec. 30, 2024].

[3] Microchip Technology Inc., "PIC18F87K22 Family Data Sheet," Microchip Technology, pp. 1, 2010. [Online]. Available: https://ww1.microchip.com/downloads/en/DeviceDoc/39960d.pdf. [Accessed: Dec. 30, 2024].

[4] ALPS Alpine Co., Ltd., "Datasheet for Pushbutton Switch RJSGME1-Y-12V-D," ALPS Alpine, pp. 1–3, 2020. [Online]. Available: https://www.farnell.com/datasheets/3466305.pdf. [Accessed: Dec. 30, 2024].

[5] Microchip Technology Inc., "PIC18F87K22 Family Data Sheet," Microchip Technology, p. 36, 2010. [Online]. Available: https://ww1.microchip.com/downloads/en/DeviceDoc/39960d.pdf. [Accessed: Dec. 30, 2024]

[6] S. K. Sahoo, S. K. Sahu, and S. K. Patra, "Enhancing Noise Immunity of Schmitt Triggers through Dual PMOS Feedback," IEEE Transactions on Circuits and Systems II: Express Briefs, vol. 70, no. 1, pp. 1-5, Jan. 2023.

[7] Microchip Technology Inc., "PIC18F87K22 Family Data Sheet," Microchip Technology, p. 497, 2010. [Online]. Available: https://ww1.microchip.com/downloads/en/DeviceDoc/39960d.pdf. [Accessed: Dec. 30, 2024]

[8] Microchip Technology Inc., "PIC18F87K22 Family Data Sheet," Microchip Technology, pp. 1, 2010. [Online]. Available: https://ww1.microchip.com/downloads/en/DeviceDoc/39960d.pdf. [Accessed: Dec. 30, 2024].

[9] Morse Code World, "Morse Code Timing," [Online]. Available: https://morsecode.world/international/timing.html. [Accessed: Dec. 30, 2024].

[10] "Reaction Time Test," Human Benchmark. [Online]. Available: https://humanbenchmark.com/tests/reactiontime. [Accessed: Dec. 30, 2024].

[11] Microchip Technology Inc., "PIC18F87K22 Family Data Sheet," Microchip Technology, p. 3, 2010. [Online]. Available: https://ww1.microchip.com/downloads/en/DeviceDoc/39960d.pdf. [Accessed: Dec. 30, 2024]

[12] Microchip Technology Inc., "PIC18F87K22 Family Data Sheet," Microchip Technology, p. 163, 2010. [Online]. Available: https://ww1.microchip.com/downloads/en/DeviceDoc/39960d.pdf. [Accessed: Dec. 30, 2024]

[13] Microchip Technology Inc., "PIC18F87K22 Family Data Sheet," Microchip Technology, p. 195, 2010. [Online]. Available: https://ww1.microchip.com/downloads/en/DeviceDoc/39960d.pdf. [Accessed: Dec. 30, 2024]

[14] T. K. Hareendran, "Switch Debouncing & Hardware Secrets," ElectroSchematics.com, Aug. 22, 2024. [Online]. Available: https://www.electroschematics.com/switch-debouncing/. [Accessed: Dec. 30, 2024].

[15] Microchip Technology Inc., "PIC18F87K22 Family Data Sheet," Microchip Technology, p. 194, 2010. [Online]. Available: https://ww1.microchip.com/downloads/en/DeviceDoc/39960d.pdf. [Accessed: Dec. 30, 2024]

[16] Microchip Technology Inc., "PIC18F87K22 Family Data Sheet," Microchip Technology, p. 195, 2010. [Online]. Available: https://ww1.microchip.com/downloads/en/DeviceDoc/39960d.pdf. [Accessed: Dec. 30, 2024]

[17] Microchip Technology Inc., "PIC18F87K22 Family Data Sheet," Microchip Technology, p. 195, 2010. [Online]. Available: https://ww1.microchip.com/downloads/en/DeviceDoc/39960d.pdf. [Accessed: Dec. 30, 2024]

[18] Hitachi, "HD44780U (LCD-II) Dot Matrix Liquid Crystal Display Controller/Driver," pp. 1–59, Sep. 1999. [Online]. Available: https://www.sparkfun.com/datasheets/LCD/HD44780.pdf. [Accessed: Dec. 30, 2024].

[19] Stanford University, "HD44780 LCD Starter Guide," [Online]. Available: https://www-leland.stanford.edu/class/ee281/handouts/lcd_tutorial.pdf. [Accessed: Dec. 30, 2024].

[20] M. Neil, "Microprocessors Lab Code Repository," Imperial College London, GitHub repository, 2024. [Online]. Available: https://github.com/ImperialCollegeLondon/MicroprocessorsLab. [Accessed: Dec. 30, 2024].

APPENDIX

The source code of this project can be found at:
https://github.com/ZZZiyao/Year3-Microprocessor-Project