CO321 Embedded Systems – 2025

PROJECT – MORSE COMMUNICATOR

GROUP NO : 12

(E/19/ 477) KOHOMBANARACHCHI N.D (MISS)

(E/19/ 484) PERAMUNAGE V.P. (MR)

TABLE OF CONTENT

TABLE OF FIGURES

INTRODUCTION

In a world where conventional communication infrastructure has been rendered unusable by global breakdown, there is a need for low-tech, resilient communication. That is the problem being addressed by this project, the Morse Communicator, in building a two-way Morse code communications terminal from basic embedded hardware components.

The system accommodates two basic modes of operation: Encoding Mode, where typed characters are translated into Morse code and emitted through an LED and buzzer, and Decoding Mode, where Morse code inputs are entered through a push button and translated back to readable text shown on an LCD screen. The project is implemented on an Arduino Uno in C with real-time timing needs and utilizing hardware elements such as LEDs, piezo buzzer, push button, and LCD display.

What this project shows is how fundamental concepts of embedded systems such as real-time processing of signals, serial communication, and user interaction via peripheral devices can be implemented to create a communications device that can save lives under adverse conditions.

HARDWARE

2.1 Hardware Components:

- **Arduino Uno:** This is the central microcontroller board that runs the entire program. It processes inputs from the serial port and the push button, controls the timing, and manages all outputs like the LED, buzzer, and LCD screen.

Figure 1.1 Arduino Uno Board

- **LCD Display:** A standard liquid-crystal display is used to show the current mode (Encoding/Decoding) and to display the final decoded message from the Morse code input.

Figure 1.2 LCD Display

- **LED:** A light-emitting diode provides visual feedback. In both encoding and decoding modes, it lights up to represent the dots and dashes being transmitted or received.

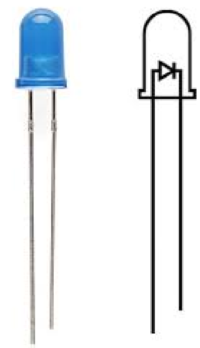

Figure 1.3 LEDs

- **Piezo Buzzer:** This component produces an audible tone. It works in sync with the LED to provide audio feedback for the Morse code signals, making it easier to interpret the timing of dots and dashes.

Figure 1.4 Piezo Buzzer

- **Push Button:** This is the primary input device for the decoding mode. The user taps out Morse code by pressing and holding the button for different durations to create dots and dashes.

Figure 1.5 Push Button

- **Resistors:** Used to limit current to components like the LED, protecting them from damage, and to act as pull-up or pull-down resistors for the push button to ensure stable input signals.

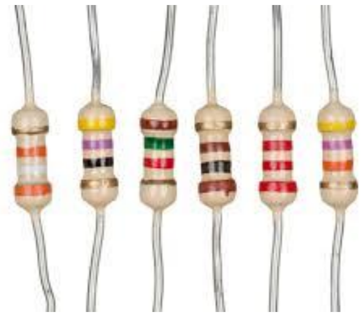

Figure 1.6 Resistors

- **Breadboard and Jumper Cables:** These are used to create a temporary, solderless circuit, connecting all the components to each other and to the Arduino's input/output pins.

Figure 1.7 Breadboard and Jump Wires

- **USB Connector:** Provides the connection for serial communication between a computer and the Arduino Uno, which is essential for the encoding mode, and powers the board.

Figure 1.8 USB Connector

2.2 Hardware Implementation Details

The hardware setup connects the various components to the Arduino Uno's digital and analog pins. The connections are defined in both the main project file and the LCD library header (lcd.h).

Outputs:

- The **LED** is connected to digital pin `PD7`.
- The **Piezo Buzzer** is connected to digital pin `PD6`.
- The project uses a standard HD44780-compatible **LCD** in 4-bit mode to save I/O pins. The specific pin mappings are defined in lcd.h as follows:
    - RS (Register Select): Connects to Arduino pin PC0 (Analog A0).
    - RW (Read/Write): Connects to Arduino pin PC1 (Analog A1).
        - *Note*: The lcd_waitbusy() function in lcd.c was modified to use a fixed 2ms delay instead of reading the busy flag. This means the RW pin can simply be grounded, though the library defines a connection.
    - E (Enable): Connects to Arduino pin PC2 (Analog A2).
    - D4 (Data Pin 4): Connects to Arduino pin PB1 (Digital 9).
    - D5 (Data Pin 5): Connects to Arduino pin PB2 (Digital 10).
    - D6 (Data Pin 6): Connects to Arduino pin PB3 (Digital 11).
    - D7 (Data Pin 7): Connects to Arduino pin PB4 (Digital 12).

Inputs:

- The **Push Button** is connected to digital pin PD2. The code configures this pin with an internal pull-up resistor (PORTD |= (1 << BUTTON_PIN);), which means the pin is normally HIGH. Pressing the button pulls the pin to LOW, and this change is detected as a press.

Communication:

- Serial communication for the encoding mode is handled via the Arduino's built-in USB port at a baud rate of 9600. This allows the Arduino to receive text from a computer to be encoded into Morse code. Also, we used HyperTerminal Software to send the Sentences to the microcontroller.

2.3 Circuit Diagram

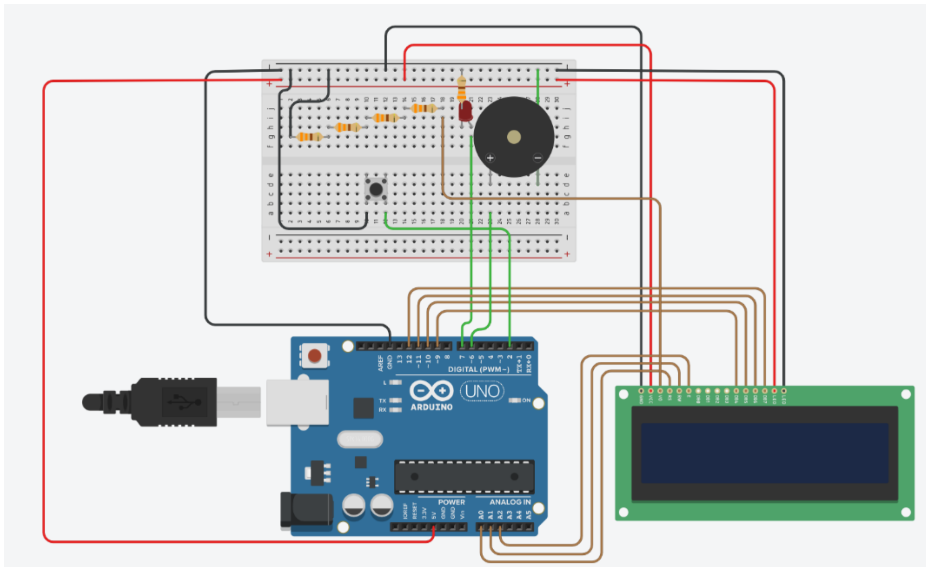

Figure 2.1 Circuit Diagram - Tinker Cad Model

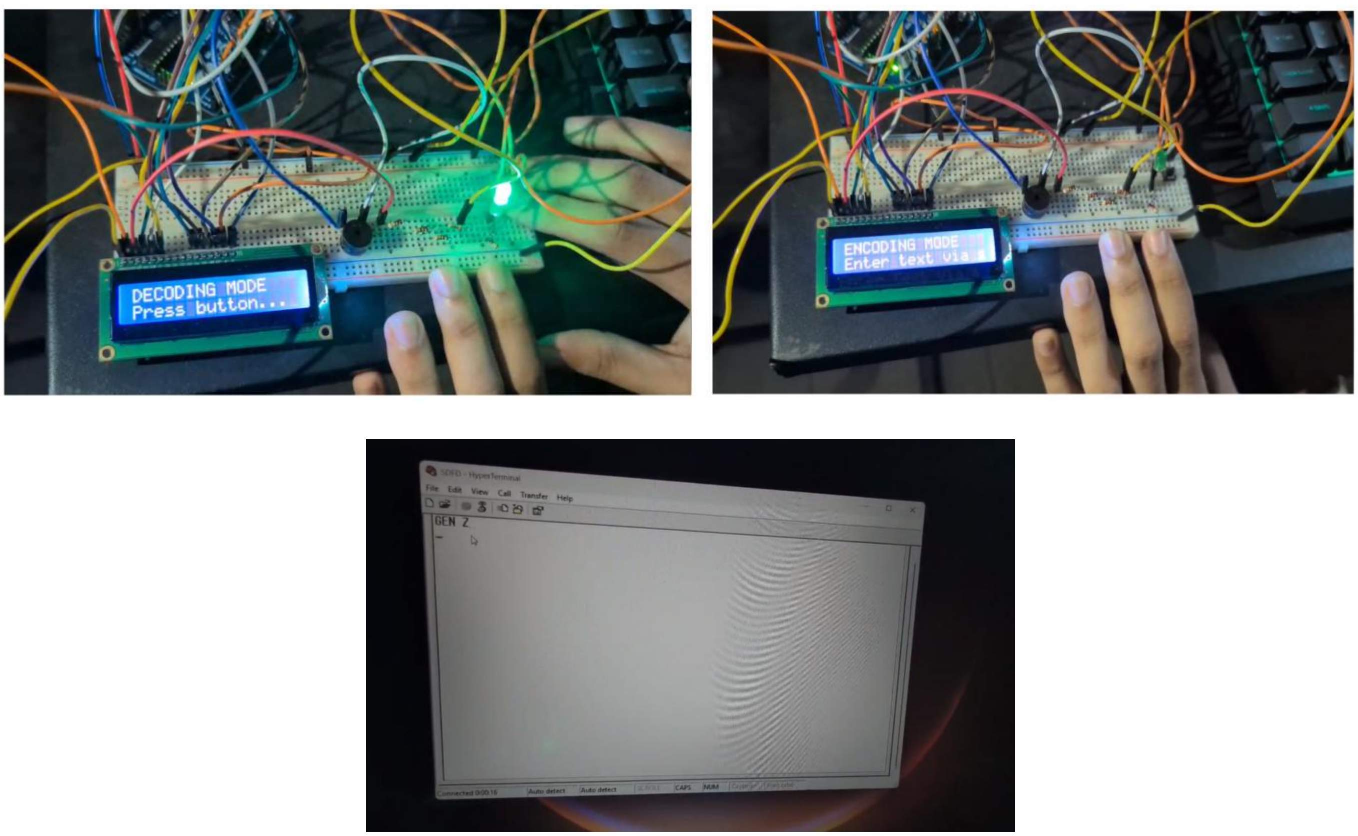

Figure 2.1 Circuit Diagram – Real Time

CODE EXPLANATION

3.1 Code

```
#include <avr/io.h>
#include <avr/interrupt.h>
#include <util/atomic.h>
#include <string.h>
#include <ctype.h>
#include "lcd.h"

// pin definitions
#define LED_PIN        PD7
#define BUZZER_PIN     PD6
#define BUTTON_PIN     PD2

//USART configuration
#define F_CPU 16000000UL
#define BAUD 9600
#define UBRR_VAL ((F_CPU / (16UL * BAUD)) - 1)

//Morse Code Timing Constants (in milliseconds)
#define DOT_PERIOD      200
#define DASH_PERIOD     (3 * DOT_PERIOD)
#define SYMBOL_GAP      DOT_PERIOD
#define CHAR_GAP        (3 * DOT_PERIOD)
#define WORD_GAP        (7 * DOT_PERIOD)


// Decoding Timing Thresholds (in milliseconds)
#define DOT_THRESHOLD   300
#define LETTER_GAP      (3 * DOT_PERIOD)
#define SPACE_THRESHOLD 1000


// Encoding and Decoding Modes
typedef enum {
    MODE_ENCODE,
    MODE_DECODE
} ProgramMode;
```

```c
volatile ProgramMode current_mode = MODE_ENCODE;

// Global Tick Counter
volatile uint32_t system_ticks = 0;

//Morse Code Table
typedef struct {
    char letter;
    const char* code;
} MorseEntry;

const MorseEntry morse_dict[] = {
    {'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', "--.."}, {'0', "-----"}, {'1', ".----"}, {'2', "..---"}, {'3', "...--"},
    {'4', "....-"},{'5', "....."}, {'6', "-...."}, {'7', "--..."}, {'8', "---.."},
    {'9', "----."}
};
const uint8_t MORSE_DICT_SIZE = sizeof(morse_dict) / sizeof(MorseEntry);


// Timer and Delay Functions
void timer0_init(void) {
    TCCR0A = (1 << WGM01);              // CTC mode
    TCCR0B = (1 << CS01) | (1 << CS00); // Prescaler = 64
    OCR0A = 249;                        // Compare match every 1ms for 16MHz clock
    TIMSK0 |= (1 << OCIE0A);            // Enable compare interrupt
}

// Timer0 Compare Match Interrupt Service Routine
// This ISR increments the system tick counter every millisecond.
ISR(TIMER0_COMPA_vect) {
    system_ticks++;
}

// Atomic access to system_ticks
// This function returns the current tick count in a thread-safe manner.
uint32_t get_ticks(void) {
```

```c
    uint32_t ticks;
    ATOMIC_BLOCK(ATOMIC_RESTORESTATE) {
        ticks = system_ticks;
    }
    return ticks;
}

// Delay function that uses busy-waiting to create a delay in milliseconds.
void delay_time(uint32_t ms) {
    uint32_t start_time = get_ticks();
    while ((get_ticks() - start_time) < ms) {
        // Busy wait
    }
}

//Hardware Initialization
void hardware_init(void) {
    // LED and Buzzer pins (PD7, PD6) as output
    DDRD |= (1 << LED_PIN) | (1 << BUZZER_PIN);

    // MODIFIED: Button pin (PD2) as input with pull-up resistor
    DDRD &= ~(1 << BUTTON_PIN);
    PORTD |= (1 << BUTTON_PIN);

    // USART Initialization
    UBRR0H = (UBRR_VAL >> 8);
    UBRR0L = UBRR_VAL;
    UCSR0B = (1 << RXEN0); // Receiver enable
    UCSR0C = (1 << UCSZ01) | (1 << UCSZ00); // 8-bit data

    // LCD Initialization using the provided library function
    // This function also configures the required LCD pins as outputs.
    lcd_init(LCD_DISP_ON);

    // Timer Initialization
    timer0_init();

    // Enable Global Interrupts
    sei();
}
```

```
unsigned char USART_receive(void) {
    while (!(UCSR0A & (1 << RXC0))); // Wait for data to be received
    return UDR0;                     // Get and return received data from the
buffer
}

// LED/Buzzer Control
void led_buzzer_on(void) {
    PORTD |= (1 << LED_PIN) | (1 << BUZZER_PIN);
}

void led_buzzer_off(void) {
    PORTD &= ~((1 << LED_PIN) | (1 << BUZZER_PIN));
}

//Get the Morse Code From Dictionary
// This function retrieves the Morse code for a given character from the
morse_dict array.
const char* get_morse_code(char c) {
    for (uint8_t i = 0; i < MORSE_DICT_SIZE; i++) {
        if (morse_dict[i].letter == c) {
            return morse_dict[i].code;
        }
    }
    return NULL;
}


// Play a Morse code symbol (dot or dash)
void sound_morse_symbol(const char* symbol) {
    for (int i = 0; symbol[i] != '\0'; i++) {
        led_buzzer_on();
        if (symbol[i] == '.') {
            delay_time(DOT_PERIOD);
        } else {
            delay_time(DASH_PERIOD);
        }
        led_buzzer_off();
        delay_time(SYMBOL_GAP);
    }
}
```

```
// Encoding Mode Functions
void run_encoding_mode(void) {
    lcd_clrscr();                              // Clear the LCD screen
    lcd_puts("ENCODING MODE");            // Display message
    lcd_gotoxy(0, 1);
    lcd_puts("Enter text via serial...");

    char received_char;
    char text_buffer[256];
    int index = 0;

    while (current_mode == MODE_ENCODE) {
        received_char = USART_receive(); // Receive character from USART

        if (received_char == '\n' || received_char == '\r') {
            if (index > 0) {
                text_buffer[index] = '\0';          // Add null terminator to the
string

                lcd_clrscr();
                lcd_puts("Transmitting...");
                lcd_gotoxy(0, 1);                       // Set cursor to second
line
                lcd_puts(text_buffer);

                for (int j = 0; text_buffer[j] != '\0'; j++) {
                    char c = toupper(text_buffer[j]);             // Convert to
uppercase
                    if (c == ' ') {
                        delay_time(WORD_GAP - CHAR_GAP);        // dealay for word
gap
                    } else {
                        const char* morse_symbol = get_morse_code(c);
                        if (morse_symbol) {
                            sound_morse_symbol(morse_symbol);        // sound the
morse code in buzzer and LED
                            delay_time(CHAR_GAP - SYMBOL_GAP);
                        }
                    }
```

```
                }
                current_mode = MODE_DECODE;
            }
            index = 0;
        } else if (index < sizeof(text_buffer) - 1) {
            text_buffer[index++] = received_char; // Store the character in the
text buffer array
        }
    }
}

//Decoding Morse Function
char decode_morse(const char* code) {
    for (uint8_t i = 0; i < MORSE_DICT_SIZE; i++) {
        if (strcmp(code, morse_dict[i].code) == 0) {
            return morse_dict[i].letter;
        }
    }
    return '?';
}

// Decoding Mode Function
void run_decoding_mode(void) {
    lcd_clrscr();
    lcd_puts("DECODING MODE");
    lcd_gotoxy(0, 1);
    lcd_puts("Press button...");

    // Morse Buffers
    char morse_char_buffer[10] = "";
    char message_buffer[64] = "";
    uint8_t msg_idx = 0;

    //Timing & State Variables
    uint32_t press_start_time = 0;
    uint32_t release_time = get_ticks();

    //Debounce Variables
    const uint8_t DEBOUNCE_DELAY = 50;
    uint8_t stable_button_state = 0;
    uint8_t last_raw_reading = !(PIND & (1 << BUTTON_PIN));
```

```
    uint32_t last_debounce_time = 0;

    while (current_mode == MODE_DECODE) {
        // Read Button State with Debouncing
        // Read the button state and apply debouncing logic.
        uint8_t raw_reading = !(PIND & (1 << BUTTON_PIN));
        if (raw_reading != last_raw_reading) {
            last_debounce_time = get_ticks();
        }

        if ((get_ticks() - last_debounce_time) > DEBOUNCE_DELAY) {
            if (raw_reading != stable_button_state) {
                stable_button_state = raw_reading;

                if (stable_button_state == 1) { // Button was PRESSED
                    press_start_time = get_ticks();
                    led_buzzer_on();
                } else { // Button was RELEASED
                    uint32_t duration = get_ticks() - press_start_time;
                    led_buzzer_off();

                    // Handle three press durations: dot, dash, and space.
                    if (duration >= SPACE_THRESHOLD) {
                        // A "space press" finalizes the previous character and
adds a space.
                        // Decode any pending character.
                        if (morse_char_buffer[0] != '\0') {
                            char decoded_char = decode_morse(morse_char_buffer);
                            if (msg_idx < sizeof(message_buffer) - 1) {
                                message_buffer[msg_idx++] = decoded_char;
                            }
                            morse_char_buffer[0] = '\0';
                        }
                        // Add the space.
                        if (msg_idx < sizeof(message_buffer) - 1) {
                            message_buffer[msg_idx++] = ' ';
                            message_buffer[msg_idx] = '\0';
                        }
                        lcd_gotoxy(0, 1); lcd_puts("                    ");
lcd_gotoxy(0, 1); lcd_puts(message_buffer); // Update LCD with current message
```

```
                    } else if (duration >= DOT_THRESHOLD) {
                        // Dash identified
                        if (strlen(morse_char_buffer) < sizeof(morse_char_buffer)
- 1) {
                            strcat(morse_char_buffer, "-");
                        }
                    } else {
                        // Dot identified
                        if (strlen(morse_char_buffer) < sizeof(morse_char_buffer)
- 1) {
                            strcat(morse_char_buffer, ".");
                        }
                    }
                    release_time = get_ticks();
                }
            }
        }
        last_raw_reading = raw_reading;

        // Process Gaps for LETTER separation
        if (stable_button_state == 0) {
            uint32_t gap = get_ticks() - release_time;

            if (morse_char_buffer[0] != '\0' && gap >= LETTER_GAP) {
                // Check for "AR" end-of-message sequence
                if (strcmp(morse_char_buffer, ".-.-.") == 0) {
                    lcd_clrscr();
                    lcd_puts("Msg Complete");
                    lcd_gotoxy(0, 1);
                    lcd_puts(message_buffer);
                    delay_time(3000);
                    current_mode = MODE_ENCODE;
                    return;
                }

                // Decode the character
                char decoded_char = decode_morse(morse_char_buffer);
                if (msg_idx < sizeof(message_buffer) - 1) {
                    message_buffer[msg_idx++] = decoded_char;
                    message_buffer[msg_idx] = '\0';
                }
```

```
                morse_char_buffer[0] = '\0';
                lcd_gotoxy(0, 1); lcd_puts("                     "); lcd_gotoxy(0,
1); lcd_puts(message_buffer);
            }
        }
    }
}


int main(void) {
    hardware_init();

    lcd_clrscr();
    while (1) {
        if (current_mode == MODE_ENCODE) {
            run_encoding_mode();
        } else {
            run_decoding_mode();
        }
    }

    return 0;
}
```

3.2 Explanation

1. Headers, Definitions, and Global Variables

The program starts by setting up the fundamental parts and establishing the variables and constants that make up the Morse Communicator system. It contains a number of crucial libraries, including <avr/io.h> for accessing AVR input/output registers, <avr/interrupt.h> for configuring interrupts (which is essential for responsiveness and timing), <util/atomic.h> for managing atomic code blocks that prevent interrupt interference, <string.h> for string manipulation functions, and <ctype.h> for character conversion utilities like toupper(). A special "lcd.h" library is also included to support LCD functions like string display and initialization. Hardware pins and configuration values can be meaningfully named using the #define directives; for instance, PD7, PD6, and PD2 are represented by LED_PIN, BUZZER_PIN, and BUTTON_PIN, respectively. Serial communication parameters are configured with `F_CPU`, `BAUD`, and `UBRR_VAL` to initialize the system for communication at baud rate 9600 at a clock speed of 16 MHz. Timing constants such as `DOT_PERIOD` (200 ms) and others as calculated from it (e.g., dash, symbol gap, and word gap duration) establish transmission and Morse code decoding intervals, while `DOT_THRESHOLD`, `LETTER_GAP`, and `SPACE_THRESHOLD` help discriminate among different input timing in decoding mode. The program defines a global `enum` type `ProgramMode` to denote the two modes of the system: encoding and decoding. A global volatile variable `current_mode` stores the system status, and the `volatile` keyword ensures real-time accuracy in the event of interrupts or mode changes. A volatile variable, `system_ticks`, serves as an internal clock, and its value is incremented by a timer interrupt. Finally, the `MorseEntry` struct and the `morse_dict[]` array are a lookup table mapping every one of the characters supported (A–Z, 0–9) to its corresponding Morse code representation in order to encode and decode efficiently.

2. Timer and Timing Functions

These arrays and structures are crucial to fulfill the condition of employing non-blocking `delay()` functions in the project by implementing a timer-based approach based on AVR's Timer0.

timer0_init(void): The `timer0_init(void)` function initializes Timer0 to count in CTC (Clear Timer on Compare Match) mode by setting `TCCR0A = (1 << WGM01)` so that the timer automatically clears when it reaches a set number. The prescaler is configured through `TCCR0B = (1 << CS01) | (1 << CS00)`, dividing the 16 MHz system clock by 64, resulting in the timer frequency of 250 kHz. The compare match register `OCR0A` is set to 249 such that the timer counts from 0 to 249, which will be 250 clock cycles or precisely 1 millisecond.

ISR(TIMER0_COMPA_vect): The line `TIMSK0 |= (1 << OCIE0A)` enables the compare match interrupt so that an interrupt will be raised every 1 ms. The associated Interrupt Service Routine `ISR(TIMER0_COMPA_vect)` is run automatically on each interrupt and increments the global `system_ticks` counter. This incrementing tick counter is a useful non-blocking system timer.

get_ticks(void): To avoid unsafe reading of the `system_ticks` value, the `get_ticks(void)` function uses an `ATOMIC_BLOCK`, which blocks interrupts for a while during the read to prevent partial or inconsistent reads of this multi-byte value.

delay_time(uint32_t ms): Finally, the `delay_time(uint32_t ms)` function gives a non-blocking delay by reading the current tick value and waiting until the specified time has elapsed. Unlike traditional `delay()`

functions, this technique allows the system to remain responsive as interrupts and other tasks can continue while waiting.

3. Encoding Mode

Initialization: The `run_encoding_mode` function controls the conversion of serial text input into Morse code signals via light and sound, addressing the project's encoding goals. It begins by clearing the LCD display and showing "ENCODING MODE" to the user to indicate the present system operating mode.

Receive Loop: In the main loop `while (current_mode == MODE_ENCODE)`, the system continues waiting for character input through the USB serial port with `received_char = USART_receive()`. The input character is stored sequentially in the `text_buffer` array for later processing if it is not newline or carriage return .

Transmission: As the newline character is encountered, the message is ready to be sent. The whole message stored in `text_buffer` is printed onto the second line of the LCD. The program now iterates through each character in the buffer, grabbing each and making it uppercase using `toupper()` so that it matches the keys in the `morse_dict` lookup table. For each valid character, the corresponding Morse code sequence is retrieved using `get_morse_code(c)` and `sound_morse_symbol(morse_symbol)` is invoked to generate the Morse pattern using the LED and piezo buzzer. Each dot or dash is printed with suitable timing (`DOT_PERIOD` or `DASH_PERIOD`) and succeeded with a `SYMBOL_GAP`. After each character, an inter-character gap (`CHAR_GAP - SYMBOL_GAP`) is inserted, and if the character is a space, a longer `WORD_GAP` is inserted to maintain correct Morse code format.

Mode Switch: Upon encoding the entire message and sending it, the mode is switched by setting `current_mode = MODE_DECODE`, which starts the main program loop's switch to decoding mode.

4. Decoding Mode

The decoding process is the most involved feature of the program, as it operates on live user input from a push button with precise timing and noise reduction.

State Variables: Several significant variables are used for controlling state: `morse_char_buffer` holds the sequence of dots and dashes that comprise one character in temporary buffer form, `message_buffer` holds the full decoded message, and `press_start_time` and `release_time` track the timing of when the button is pressed.

Debouncing: To receive accurate readings, a debouncing mechanism is used. The system monitors the raw button state (`raw_reading`) for changes and, upon observing a change, it resets a timer (`last_debounce_time`). Only after the signal has remained stable for some time period (`DEBOUNCE_DELAY`, 50 ms) is the new state accepted, preventing false triggering caused by electrical noise.

Press and Release Logic:

- **Button Pressed:** Whenever the steady press is felt, the current time is saved in `press_start_time`, and both LED and buzzer are activated to provide immediate feedback to the user.
- **Button Released:** When the button is released, the time for which the button was pressed is calculated as the difference between `press_start_time` and current tick value. If the length of the pressing is greater than `DOT_THRESHOLD`, it's interpreted as a dash (`-`); otherwise, a dot (`.`).The corresponding symbol is appended to morse_char_buffer using strcat().The time of release is recorded in release_time.
- **Gap Processing:** When the button is not being pressed (`stable_button_state == 0`), the code retrieves the duration since last release. If the gap is larger than `LETTER_GAP` and there is a valid sequence in `morse_char_buffer`, the system discovers that a letter has been completed. It then calls `decode_morse(morse_char_buffer)` to decode the Morse string into its corresponding alphanumeric character and this is added to `message_buffer`. The `morse_char_buffer` is cleared for the next input.
- **End of Message:** The system also searches for the end-of-message signal in the form of the Morse prosign "AR" (`.-.-.`). If the sequence is discovered, the program displays the last decoded message on the LCD with a delay of 3 seconds and restores the mode to `MODE_ENCODE` so that the program can proceed to the encoding stage.

5. The Main Function

The `main()` function is a function that acts as the main controller, and it begins by calling `hardware_init()` to initialize all peripherals needed. It then enters into an infinite loop (`while(1)`) that constantly checks the `current_mode` variable. Depending on its value, it calls either `run_encoding_mode()` or `run_decoding_mode()`, each of which is responsible for its own running and changes the mode upon completion. This design is an extremely simple but effective state machine that repeatedly alternates between encoding and decoding and produces perfect bidirectional Morse communication.

TIMELINE

Activity - Date	21–23 June	24–27 June	28–30 June	01–04 July	05–09 July	10–14 July	15–19 July	20–23 July	24–26 July	27–28 July	29-Jul	30-Jul
Project understanding, role assignment, planning												
Technical research: Morse code, timers, interrupts												
Hardware setup: Arduino, LCD, button, buzzer, LED												
Implement non-blocking Timer0 using ISR												
Serial encoding logic & USART input handling												
Morse code lookup table, buzzer + LED signal output												
Decoding input: button logic, debouncing algorithm												
Decoding logic: dot/dash detection, Morse parsing												
Mode switching logic between encoding/decoding												
Final testing, hardware cleanup, bug fixes												
Prepare demonstration video and report writing												
Final review and submission												

Figure 3.1 TimeLine

CONCLUSION

Using an AVR microcontroller and simple hardware parts, the Morse Communicator project effectively illustrates a working, low-level communication system based on Morse code. One of the main technological criteria is met by the system's accurate timing management without the need for delay functions thanks to its efficient use of non-blocking Timer0 interrupts. Real-time feedback was given via an LCD, buzzer, and LED, and the dual-mode operation—encoding via serial input and decoding by manual button presses—was executed flawlessly. A complete bidirectional communication cycle is completed by the automatic handling of the mode change. This project stresses practical design, circuit integration, and user interaction in addition to highlighting fundamental embedded system principles like timers, interrupts, I/O control, and serial communication. All things considered, the system fits the stated learning objectives quite nicely.