WIRELESS COMMUNICATION USING ESP NOW PROTOCOL (MORSE CODE)

INTRODUCTION:

The ESP32 Wireless Morse Code Communication project creates a modern bridge between historical communication methods and contemporary wireless technology. This system enables users to transmit Morse code messages wirelessly between two ESP32 devices using the ESP-NOW protocol. Messages input through a push button on one device are instantly transmitted, decoded, and displayed on an OLED screen of the receiving device, creating a seamless communication experience that combines the charm of Morse code with the efficiency of modern microcontrollers.

COMPONENTS REQUIRED:

The project requires minimal hardware while delivering robust functionality. At its core, you'll need two ESP32 development boards to establish the wireless communication link. Each unit needs to be equipped with an SSD1306 OLED display (128x64 pixels) to show the decoded messages. A simple push button serves as the input mechanism for Morse code entry. The circuit is completed with basic components including jumper wires and a breadboard for connections. While an external pull-up resistor can be used, the project utilizes the ESP32's internal pull-up resistor to maintain a clean setup.

HARDWARE CONFIGURATION:

The hardware assembly focuses on creating a clean and efficient interface. The OLED display connects to the ESP32 using the I2C protocol, with SDA and SCL lines connected to GPIO 21 and 22 respectively. These pins are the default I2C pins on most ESP32 development boards, ensuring broad compatibility. The display also requires power connections: VCC to 3.3V and GND to ground. The push button installation is straightforward, with one terminal connected to GPIO 15 and the other to ground. The internal pull-up resistor is enabled through software, eliminating the need for external components and simplifying the circuit.

CODE:

```
#include <esp_now.h>

#include <WiFi.h>

#include <Wire.h>

#include <Adafruit_GFX.h>
```

```
#include <Adafruit_SSD1306.h>

// OLED display configurations
#define SCREEN_WIDTH 128 // OLED display width in pixels
#define SCREEN_HEIGHT 64 // OLED display height in pixels
#define OLED_RESET    -1 // Reset pin for OLED (not used)

// Initialize the OLED display
Adafruit_SSD1306 display(SCREEN_WIDTH, SCREEN_HEIGHT, &Wire, OLED_RESET);

// Button configuration
const int buttonPin = 15;    // GPIO pin connected to the button
bool buttonState = LOW;      // Current state of the button
bool lastButtonState = LOW;  // Previous state of the button

// Variables for timing and debouncing
unsigned long lastReceiveTime = 0;     // Tracks the last activity time
unsigned long lastDebounceTime = 0;    // Timestamp for last debounce check
const unsigned long debounceDelay = 100; // Debounce delay in milliseconds
unsigned long pressStartTime = 0;      // Time when the button was pressed
unsigned long releaseStartTime = 0;    // Time when the button was released

// String to store Morse code sequence
String morseCode = "";

// Receiver's MAC address (update with the correct one)
uint8_t receiverMAC[] = {0x30, 0xc6, 0xf7, 0x2a, 0x32, 0x64};

// Morse code lookup table
```

```
const char* morseTable[][2] = {
  {".-", "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"}
};

// Display cursor position
int x = 0;
int y = 10;

// Flag to indicate data reception
bool isReceiving = false;

// Function to decode Morse code into a character
char decodeMorse(String code) {
  for (int i = 0; i < sizeof(morseTable) / sizeof(morseTable[0]); i++) {
    if (code == morseTable[i][0]) {
      return morseTable[i][1][0];
    }
  }
  return '?'; // Return '?' if the code is unrecognized
}

// Function to display a decoded character on the OLED
```

```
void displayDecodedChar(char decodedChar) {
  // Adjust cursor position for display wrapping
  if (x > SCREEN_WIDTH - 12) {
    x = 0;
    y += 16;
  }
  if (y > SCREEN_HEIGHT - 16) {
    x = 0;
    y = 10;
    display.clearDisplay();
  }

  // Display the character
  display.setCursor(x, y);
  display.setTextSize(2);
  display.print(decodedChar);
  display.display();

  x += 12; // Move cursor for the next character
}

// ESP-NOW callback for receiving data
void onDataReceive(const esp_now_recv_info *info, const uint8_t *data, int len) {
  isReceiving = true; // Indicate receiving process

  // Decode received Morse code
  char incomingMessage[len + 1];
  memcpy(incomingMessage, data, len);
  incomingMessage[len] = '\0';
  String morseCode = String(incomingMessage);
  char decodedChar = decodeMorse(morseCode);

  // Print received data to the Serial Monitor
  Serial.print("Received Morse Code: ");
  Serial.println(morseCode);
  Serial.print("Decoded Character: ");
  Serial.println(decodedChar);

  // Display the decoded character
  displayDecodedChar(decodedChar);

  lastReceiveTime = millis(); // Update last receive time
  isReceiving = false; // Reset receiving flag
}

// Setup function
void setup() {
  Serial.begin(115200);

  // Initialize OLED display
  if (!display.begin(SSD1306_SWITCHCAPVCC, 0x3C)) {
    Serial.println(F("SSD1306 allocation failed"));
    for (;;);
  }
  display.clearDisplay();
  display.setTextSize(3);
  display.setTextColor(WHITE);
  display.setCursor(0, 10);
  display.println("Morse Code");
  display.display();
  delay(2000);
  display.clearDisplay();

  // Configure button pin
  pinMode(buttonPin, INPUT_PULLUP);

  // Initialize ESP-NOW
  WiFi.mode(WIFI_STA);
  if (esp_now_init() != ESP_OK) {
    Serial.println("ESP-NOW Initialization Failed!");
    return;
  }

  // Add receiver's MAC address as a peer
  esp_now_peer_info_t peerInfo = {};
  memcpy(peerInfo.peer_addr, receiverMAC, 6);
  peerInfo.channel = 0;
  peerInfo.encrypt = false;
  if (esp_now_add_peer(&peerInfo) != ESP_OK) {
    Serial.println("Failed to add peer");
    return;
  }

  // Register callback function for receiving data
  esp_now_register_recv_cb(onDataReceive);
}
```

```
// Function to send Morse code
void sendMorseCode(String code) {
  if (isReceiving) {
    Serial.println("Cannot send Morse code while receiving data.");
    return;
  }

  const char* message = code.c_str();
  esp_err_t result = esp_now_send(receiverMAC, (uint8_t*)message, strlen(message));
  if (result == ESP_OK) {
    Serial.println("Message sent successfully: " + code);
  } else {
    Serial.println("Failed to send message");
  }
}

// Main loop
void loop() {
  int reading = digitalRead(buttonPin);

  // Handle button state changes and debounce
  if (reading != lastButtonState) {
    lastDebounceTime = millis();
  }

  if ((millis() - lastDebounceTime) > debounceDelay) {
    if (reading != buttonState) {
      buttonState = reading;
    if (buttonState == LOW) {
      pressStartTime = millis();
    } else {
      unsigned long pressDuration = millis() - pressStartTime;
      if (pressDuration < 400) {
        morseCode += "."; // Short press: dot
      } else {
        morseCode += "-"; // Long press: dash
      }
      releaseStartTime = millis();
    }
  }
}

// Send Morse code if there's no input for a while
if (releaseStartTime > 0 && (millis() - releaseStartTime > 1250)) {
  sendMorseCode(morseCode);
  morseCode = ""; // Reset Morse code buffer
  releaseStartTime = 0;
}

// Clear OLED display if no activity for 10 seconds
if (millis() - lastReceiveTime > 10000) {
  display.clearDisplay();
  display.display();
  x = 0; // Reset cursor position
  y = 10;
}
```

```
  lastButtonState = reading; // Update last button state
}
```

CODE IMPLEMENTATION AND CONCEPTS:

ESP-NOW PROTOCOL IMPLEMENTATION:

The project leverages ESP-NOW, Espressif's versatile peer-to-peer communication protocol, to establish direct device-to-device communication. The implementation begins with initializing the ESP32 in station mode and enabling ESP-NOW. Here's how the code handles this:

```
WiFi.mode(WIFI_STA);
if (esp_now_init() != ESP_OK) {
   Serial.println("ESP-NOW Initialization Failed!");
   return;
}
```

After initialization, the system sets up peer relationships by registering the receiver's MAC address. This creates a secure and direct communication channel between the devices:

```
esp_now_peer_info_t peerInfo = {};
memcpy(peerInfo.peer_addr, receiverMAC, 6);
peerInfo.channel = 0;
peerInfo.encrypt = false;
```

DISPLAY MANAGEMENT SYSTEM:

The display system utilizes the Adafruit SSD1306 library to manage the OLED screen. The implementation includes sophisticated text wrapping and screen management features. The display buffer is configured for a 128x64 pixel resolution, and the system tracks cursor position to ensure text appears correctly on screen. When displaying decoded characters, the system automatically handles line wrapping and screen clearing:

```
void displayDecodedChar(char decodedChar) {
    if (x > SCREEN_WIDTH - 12) {
        x = 0;
        y += 16;
    }
    if (y > SCREEN_HEIGHT - 16) {
         x = 0;
        y = 10;
        display.clearDisplay();
    }
    display.setCursor(x, y);
    display.setTextSize(2);
    display.print(decodedChar);
    display.display();
    x += 12;
}
```

MORSE CODE PROCESSING:

The Morse code implementation combines traditional encoding schemes with modern digital processing. The system uses a comprehensive lookup table that maps Morse code sequences to their corresponding characters. The decoder function efficiently traverses this table to convert received signals into readable text:

```
const char* morseTable[][2] = {
    {".-", "A"}, {"-...", "B"}, {"-.-.", "C"},
    // Additional characters omitted for brevity
};

char decodeMorse(String code) {
    for (int i = 0; i < sizeof(morseTable) / sizeof(morseTable[0]); i++) {
        if (code == morseTable[i][0]) {
            return morseTable[i][1][0];
        }
    }
    return '?';
}
```

INPUT PROCESSING AND TIMING:

The input system employs sophisticated timing mechanisms to differentiate between dots and dashes. Button presses are measured with millisecond precision: presses under 400ms register as dots, while longer presses create dashes. The system includes debounce protection to ensure reliable input:

```
if (buttonState == LOW) {
    pressStartTime = millis();
} else {
    unsigned long pressDuration = millis() - pressStartTime;
    if (pressDuration < 400) {
        morseCode += ".";
    } else {
        morseCode += "-";
    }
    releaseStartTime = millis();
}
```

TIMING LOGIC USING MILLIS():

The project extensively uses the millis() function for precise timing control instead of delay(). This is a crucial design choice as millis() provides non-blocking timing functionality, allowing the system to perform multiple tasks simultaneously. The millis() function returns the number of milliseconds since the ESP32 began running the current program, enabling precise time measurements without pausing program execution.

In our implementation, millis() serves several critical timing functions:

```
unsigned long lastReceiveTime = 0;     // Tracks the last activity time
unsigned long lastDebounceTime = 0;    // Timestamp for last debounce check
unsigned long pressStartTime = 0;      // Time when the button was pressed
unsigned long releaseStartTime = 0;    // Time when the button was released
```

These timing variables work together to create a robust timing system:

```
// Clear display after inactivity
if (millis() - lastReceiveTime > 10000) {
    display.clearDisplay();
    display.display();
    x = 0;
    y = 10;
}

// Send Morse code after pause in input
if (releaseStartTime > 0 && (millis() - releaseStartTime > 1250)) {
    sendMorseCode(morseCode);
    morseCode = "";
    releaseStartTime = 0;
}
```

THE TIMING MEASUREMENTS ARE USED TO:

- Distinguish between dots (<400ms) and dashes (≥400ms)
- Detect the end of a character (1250ms pause)
- Clear the display after inactivity (10 seconds)

- Ensure reliable button state changes (100ms debounce)

BUTTON DEBOUNCING IMPLEMENTATION:

Button debouncing is a critical feature that ensures reliable input reading by filtering out mechanical noise from button presses. When a mechanical button is pressed or released, the signal can fluctuate rapidly (bounce) before settling to its final state. Our implementation uses a software debouncing technique that requires the button state to remain stable for a specified period before accepting it as a valid change.

Here's how the debouncing logic works in detail:

```
// Button and debounce variables
const int buttonPin = 15;              // GPIO pin for button
bool buttonState = LOW;                // Current stable state
bool lastButtonState = LOW;            // Previous stable state
const unsigned long debounceDelay = 100; // Debounce time in milliseconds

void loop() {
  int reading = digitalRead(buttonPin);  // Get current button reading

  // If the reading is different from the last state, reset the debounce timer
  if (reading != lastButtonState) {
    lastDebounceTime = millis();
  }

  // Check if enough time has passed since the last state change
  if ((millis() - lastDebounceTime) > debounceDelay) {
    // If the button state has changed and is stable
    if (reading != buttonState) {
      buttonState = reading;
      // Handle button press and release
      if (buttonState == LOW) {  // Button pressed
        pressStartTime = millis();
      } else {  // Button released
        unsigned long pressDuration = millis() - pressStartTime;
        // Determine if it's a dot or dash based on duration
        if (pressDuration < 400) {
          morseCode += ".";
        } else {
          morseCode += "-";
        }
        releaseStartTime = millis();
      }
    }
  }

  lastButtonState = reading;  // Save the current reading
}
```

The debouncing process follows these steps:

1. The code continuously reads the button state.
2. When a change is detected (reading != lastButtonState), the debounce timer is reset.
3. The code waits for the debounceDelay period (100ms in this case).
4. If the button state remains stable during this period, it's accepted as a valid change.
5. Only then does the code process the button press or release.

THIS IMPLEMENTATION PROVIDES SEVERAL BENEFITS:

- Eliminates false triggers from button bounce
- Provides reliable timing measurements for Morse code input
- Prevents multiple accidental inputs from a single press
- Maintains accurate dot/dash differentiation

SETUP GUIDE:

Setting up the system requires careful attention to both hardware and software components. Begin by connecting the OLED display and push button according to the hardware configuration described earlier. The code must be uploaded to both ESP32 devices, with the receiver's MAC address properly configured in the transmitting unit. The MAC address can be found by running a simple sketch that prints the WiFi MAC address to the Serial Monitor.

FUTURE DEVELOPMENT POTENTIAL:

The current implementation provides a solid foundation for further enhancements. Future development could include adding support for full word transmission, implementing error detection and correction mechanisms, and adding configurable timing parameters. The system could be expanded to include features like message history storage, battery monitoring for portable operation, and encrypted communication for secure messaging.

PROJECT LIMITATIONS:

While the current implementation is functional, it's important to note certain limitations. The system currently handles single-character transmission, which could be expanded to support full words or sentences. The fixed timing parameters for Morse code input might need adjustment for different users. The ESP-NOW protocol, while efficient, has range limitations and works best with line-of-sight positioning. Environmental factors can affect transmission quality and range.

CONTRIBUTION GUIDELINES:

This project welcomes community contributions and improvements. When contributing, please maintain consistent code styling and provide clear documentation for any new features or modifications. Test all changes thoroughly before submitting pull requests, and ensure that any new features maintain compatibility with the existing system architecture.