

Data Packet Generation with AES Encryption

AX.25 Data Packet generation along with AES 128 encryption

By: Abdullah Farooq and Muhammad Umer

Supervisor (RO): Engr. Zubair

Date of Submission: 30th August 2024

Table of Contents

1. Overview of AX.25 Protocol

- Callsign-Based Addressing
- Adaptation to Radio Conditions
- Purpose of the Documentation
- Scope and Audience

2. Point-to-Point and Broadcast Data Transmission in AX.25

- a. Point-to-Point Transmission
- b. Broadcast Data Transmission (UI Frames)

3. Adaptability to Various Hardware Setups, Including NRF Modules

4. Data Packet Structure of AX.25 UI Frames

- Flag Field
- Address Field
- Control Field
- Information Field
- Frame Check Sequence (FCS)

5. Comparison of I Frames and S Frames with UI Frames

- I Frames
- S Frames

6. Addressing in AX.25

- Source and Destination Addresses
- Repeater Fields
- Callsign and SSID Structure

7. Implementation and Configuration

- Software Tools and Libraries
- Sample Code Snippets

8. Functions Used in Encryption, Decryption, Fragmentation, and Defragmentation

- CRC Calculation
- Fragmentation
- Defragmentation
- AES Encryption and Decryption

9. Frame and Packet Structure

- Packet Structure (AX25Frame)
- Frame Summary

10. Hardware Requirements

- NRF Modules
- Arduino

11. Troubleshooting and Debugging

- Common Issues
- Issues faced during building
- Analyzing packets with wireshark
- Logging and Monitoring

12. Security Considerations

- Encryption
- Authentication

13. Encryption and Authentication

- AES-128 Encryption Overview
- Example: Data Encryption and Decryption

14. Future of AX.25

- Developments and Enhancements
 - IoT and Advanced Telemetry
- Alternatives and Comparisons
 - Protocol Comparisons (LoRaWAN, Zigbee)

15. Online Communities and Support

- Forums and Resources for AX.25 and Radio Communications

16. References and Resources

- Technical Documentation
 1. AX.25 Protocol Standards
 2. Arduino Libraries Documentation
 3. NRF Module Datasheets
- Recommended Readings
 1. Packet Radio Books
 2. Digital Communications
 3. Arduino and Radio Setups
 4. Advanced Telemetry and IoT Applications

1. Introduction

- **Overview of AX.25 Protocol:**

AX.25 is a data link layer protocol used primarily in amateur radio to facilitate digital communication between radio stations. It is derived from the X.25 protocol, AX.25 is optimized for radio-based environments where conditions are less reliable. This section provides a concise overview of AX.25's purpose and its key characteristics, including the unique aspects of radio communication.

AX.25 supports both connected (reliable) and connectionless (broadcast) data transmission. Other features include:

Key characteristics of AX.25 include:

- **Callsign-Based Addressing:** Uses amateur radio callsigns as source and destination addresses, allowing easy identification of stations and routing of packets through repeaters.
- **Error Detection:** Incorporates Frame Check Sequences (FCS) to ensure data integrity, which is crucial in radio environments prone to noise and signal degradation.
- **Flexibility in Communication:** Supports both acknowledged (I frames) and unacknowledged (UI frames) communication modes, providing the option for reliable connections or simple data broadcasts.
- **Adaptation to Radio Conditions:** Designed to handle the dynamic nature of radio communication, including bursty traffic and varying signal strengths, making it suitable for telemetry, APRS, and other digital communication applications.
- **Purpose of the Documentation:**

This document aims to explain the use of AX.25 for radio communications. It will focus specifically on using Unnumbered Information (UI) frames, suitable for scenarios that require straightforward data exchange without the overhead of connection management.
- **Scope and Audience:**

This guide is intended for engineers, hobbyists, and developers working with radio communication protocols, particularly those interested in using AX.25 with microcontrollers like Arduino and NRF modules.

Point-to-Point and Broadcast Data Transmission in AX.25

AX.25 is a versatile protocol that supports both point-to-point and broadcast data transmission modes, making it suitable for a wide range of communication scenarios, particularly in amateur radio and remote telemetry.

1. Point-to-Point Transmission:

- **How it Works:** In point-to-point transmission, AX.25 establishes a direct communication link between two radio stations, similar to a peer-to-peer connection in computer networks. This mode is typically used when reliable communication is needed, as it allows the exchange of control and information frames that manage data flow, acknowledgments, and error correction.
- **Connected Mode:** When operating in connected mode using Information (I) frames, the protocol ensures that data is delivered accurately, with retransmissions occurring if errors are detected. This mode is ideal for scenarios where data integrity is crucial, such as sending critical control commands or sensitive data.
- **Applications:** Point-to-point communication is often used in controlled environments like direct data transfers between remote sensors, weather stations, or other setups where one-to-one communication is essential.

2. Broadcast Data Transmission (Unnumbered Information - UI Frames):

- **How it Works:** AX.25's UI frames allow for broadcast data transmission, where packets are sent without establishing a formal connection or requiring acknowledgments. This mode is connectionless, meaning data is transmitted freely, and any station within range can receive the packets without participating in a handshake process.
- **UI Frames:** UI frames are specifically designed for simple, low-overhead communication, making them ideal for broadcasting information, such as sensor data, position reports, or general announcements. This mode is particularly efficient in environments where quick, one-way data transmission is required without the complexity of connection management.
- **Applications:** UI frames are widely used in Automatic Packet Reporting System (APRS), telemetry, beacon transmissions, and other applications

where devices need to share information with multiple listeners without requiring feedback.

Adaptability to Various Hardware Setups, Including NRF Modules:

- **NRF Modules Integration:** AX.25's flexibility allows it to be implemented on different hardware, including microcontrollers and radio modules like NRF24L01. These low-cost, low-power radio modules can transmit and receive AX.25 frames, making them suitable for projects that require lightweight, efficient radio communication.
- **Arduino Compatibility:** By using AX.25 with NRF modules on platforms like Arduino, developers can create custom radio communication setups for IoT devices, remote sensors, and hobbyist projects. The combination of AX.25 and NRF modules provides a robust solution for wireless data exchange, leveraging the protocol's ability to handle varying radio conditions and its support for both directed and broadcast communication.
- **Scalability:** The protocol's inherent design allows for easy scalability in hardware setups, supporting simple point-to-point connections as well as complex networks involving multiple stations and repeaters, enhancing its utility across different communication environments.

Data Packet Structure of AX.25 UI Frames

The AX.25 data packet structure is essential for understanding how data is organized, transmitted, and received in radio communications. This section provides a detailed look at the frame format, focusing on Unnumbered Information (UI) frames, which are particularly relevant to our project.

Frame Format:

The AX.25 frame consists of several key components that ensure proper data encapsulation and transmission. The overall layout includes:

1. Flag Field:

- **Purpose:** The flag is a specific byte (`0x7E`) that marks the beginning and end of an AX.25 frame, allowing the receiver to recognize the boundaries of each frame.

- **Structure:** Typically, one flag byte is placed at the start, and another at the end of the frame, ensuring that the frame is correctly identified and extracted from the data stream.

2. Address Field:

- **Content:** This field includes source and destination addresses, usually formatted as callsigns, and optional repeater addresses if the packet is to be relayed.
- **Structure:** The Address Field contains:
 - **Destination Address:** The callsign of the intended recipient, often followed by an SSID (Secondary Station Identifier) to differentiate multiple stations using the same callsign.
 - **Source Address:** The callsign of the sender, also including an SSID.
 - **Repeater Fields:** Optional fields that specify repeaters (intermediate stations) through which the packet should be routed.
- **Usage:** This field ensures that the packet reaches its intended destination, and it allows routing flexibility through repeaters, enhancing communication range.

3. Control Field:

- **Purpose:** The Control Field in a UI frame is fixed and set to `0x03`, indicating an unnumbered frame that does not require acknowledgment or sequence numbers.
- **Functionality:** This field defines the type of frame and dictates that the packet is to be transmitted without connection setup or acknowledgment, making it ideal for one-way broadcasts and telemetry data.
- **Application:** The simplicity of this field reduces overhead and is particularly advantageous in radio communications where rapid, unacknowledged data transmission is desired.

4. Information Field:

- **Purpose:** The Information Field carries the actual data payload, such as sensor readings, status information, or text messages.

- **Structure and Length:** The maximum length of the Information Field can vary, but it is typically limited to 256 bytes to balance data size and transmission efficiency. This limitation ensures the frame can be reliably transmitted and processed in low-bandwidth radio environments.
- **Usage in NRF and Arduino Setup:** In NRF and Arduino setup, the Information Field is used to transmit data between devices without the need for acknowledgment, streamlining communication between microcontrollers and radio modules. This approach is efficient for applications like telemetry, where data is broadcast to multiple listeners. Maximum limit is 32 bytes in this integration.

5. Frame Check Sequence (FCS):

- **Purpose:** The FCS is a crucial part of the frame, ensuring data integrity by checking for errors during transmission.
- **CRC (Cyclic Redundancy Check) Process:** The FCS is generated using a CRC algorithm that produces a checksum based on the entire frame content (excluding flags). When the frame is received, the CRC is recalculated and compared against the transmitted FCS.
- **Error Detection:** If there is a mismatch, the frame is discarded, preventing corrupted data from being processed. This mechanism is vital in radio communications where noise and interference can introduce errors.
- **Importance in Radio Communications:** Given the variability of radio signal quality, the FCS plays a key role in maintaining reliable data transmission, ensuring that only valid, error-free packets are considered.

6. Packet Types and Functions

- **UI (Unnumbered Information) Frames:**
Focus on UI frames as the core of our project, explaining how these frames operate in a connectionless manner, making them ideal for broadcast and sensor data applications on NRF hardware. This section will detail the simplicity and efficiency of UI frames for radio communications, where rapid, unacknowledged data transmission is beneficial.

Comparison of I (Information) Frames and S (Supervisory) Frames with UI Frames

In AX.25, different types of frames serve distinct purposes based on the communication needs, such as connection management, error control, and data transmission. Here's an explanation of I (Information) Frames and S (Supervisory) Frames, highlighting why they are not suitable for our project compared to UI (Unnumbered Information) Frames.

I (Information) Frames:

- **Purpose and Functionality:**

- I frames are used in AX.25 for connection-oriented communication. They are responsible for transmitting user data between two stations in a reliable, sequential manner.
- These frames include sequence numbers that manage the order of data packets and ensure that all data is received correctly, with mechanisms for acknowledgment, retransmission, and flow control.

- **Structure:**

- I frames contain fields for sequence numbers, acknowledgment numbers, and a Poll/Final bit for control purposes, which add complexity to the frame.
- They require a setup phase where both sending and receiving stations establish a connection, negotiate parameters, and manage the state of the communication link.

- **Why Not Used in our Project:**

- **Complexity:** I frames introduce significant complexity due to their connection-oriented nature, which requires both ends to manage connection states, handle acknowledgments, and maintain proper sequencing.
- **Overhead:** The additional fields and control mechanisms increase the overhead, making I frames less efficient for applications where rapid, unidirectional communication is needed.

- **Not Suitable for Broadcast:** I frames are designed for point-to-point communication, which is unnecessary in application where data needs to be broadcasted or transmitted without the need for acknowledgment or connection management.
- **NRF and Arduino Limitations:** Implementing I frames on low-power, resource-constrained devices like Arduino and NRF modules can be cumbersome due to the need for state management and continuous data integrity checks.

S (Supervisory) Frames:

- **Purpose and Functionality:**

- S frames are used for managing the flow of data and controlling errors in AX.25 connections. They provide feedback about the status of the communication link between stations.
- Common types of S frames include Receive Ready (RR), Receive Not Ready (RNR), and Reject (REJ). These frames help manage the flow by indicating if a station is ready to receive more data, needs to pause, or has encountered errors that require retransmission.

- **Structure:**

- S frames have specific fields for acknowledging received frames and managing flow control between connected stations. They do not carry user data but instead focus on controlling the connection status.
- They are used in tandem with I frames to ensure the reliable transfer of data, correcting errors and adjusting the transmission rate as needed.

- **Why Used in Our Project:**

- **No Need for Flow Control:** Since our project uses UI frames, which are designed for unidirectional, connectionless data transfer, there is no need for flow control or acknowledgment of frames, making S frames irrelevant.
- **Error Handling Is Simplified:** UI frames rely on the FCS for error detection without the additional control mechanisms that S frames provide, simplifying the error-handling process.

- **Unnecessary Overhead:** S frames add unnecessary communication overhead, particularly in scenarios where data is sent as a one-time broadcast without expectation of feedback or acknowledgment.
- **Inefficiency in Broadcast Scenarios:** S frames are designed to support continuous, reliable communication rather than broadcast or telemetry tasks, which are better suited to UI frames that send data without expecting a response.

Addressing in AX.25

Addressing in AX.25 is a fundamental aspect that ensures packets reach their intended destination while allowing flexibility in routing through various stations, including repeaters. The protocol uses a combination of callsigns and SSIDs (Secondary Station Identifiers) to uniquely identify each station on the network.

Source and Destination Addresses:

- **Structure:**
 - AX.25 addresses are structured using amateur radio callsigns, which are typically up to six alphanumeric characters followed by an optional SSID, which ranges from 0 to 15.
 - The address field in an AX.25 frame includes both the source (sender) and destination (receiver) callsigns, along with their respective SSIDs.
 - **Example Address:** A typical address might look like `CALLSIGN-1` (where "CALLSIGN" is the station's identifier and "1" is the SSID).
- **Functionality:**
 - **Source Address:** Identifies the station that is sending the packet. This allows receiving stations to recognize the origin of the transmission.
 - **Destination Address:** Specifies the intended recipient of the packet. If the destination is not the final endpoint, repeaters use this information to forward the packet accordingly.
 - **Usage in Radio Modules:** In implementation with NRF modules and Arduino, the addressing field helps direct data correctly, allowing devices to communicate efficiently even in networks involving multiple radios.

- **Importance:**

- Proper structuring of source and destination addresses ensures that data is routed correctly through the network, enabling precise communication between specific endpoints.
- This addressing system is particularly useful in radio communications, where devices may often be mobile or operate in complex environments with multiple potential paths.

Repeater Fields:

- **Role of Repeaters:**

- Repeaters are intermediate stations that extend the communication range of AX.25 packets by receiving and retransmitting them, effectively acting as relay points in a network.
- Each packet can include up to eight repeater addresses, specifying the route the packet should take from the source to the final destination.

- **How Repeater Fields Work:**

- When a packet passes through a repeater, the repeater modifies its portion of the address field to indicate that it has handled the packet. However, the original source, destination, and payload data remain unchanged.
- **UI Frame Interaction:** UI frames, commonly used in our project, do not require any acknowledgment or connection setup from repeaters, allowing them to pass through multiple repeaters without additional processing, making them ideal for broadcast communication in extended networks.

- **Benefits:**

- **Extended Range:** By using repeaters, stations that are far apart or obstructed by terrain can communicate effectively, greatly enhancing the reach of AX.25-based communications.
- **Network Flexibility:** Repeaters provide flexibility in network design, allowing stations to connect indirectly through one or more relay points, which is particularly useful in remote or large coverage areas.

Callsign and SSID Structure:

- **Callsign Format:**

- Callsigns are derived from radio licensing requirements and are used to identify individual stations on the network. They can be a combination of letters and numbers, usually determined by national regulatory bodies.
- The callsign acts as a unique identifier that distinguishes one station from another in the AX.25 network.

- **SSID (Secondary Station Identifier):**

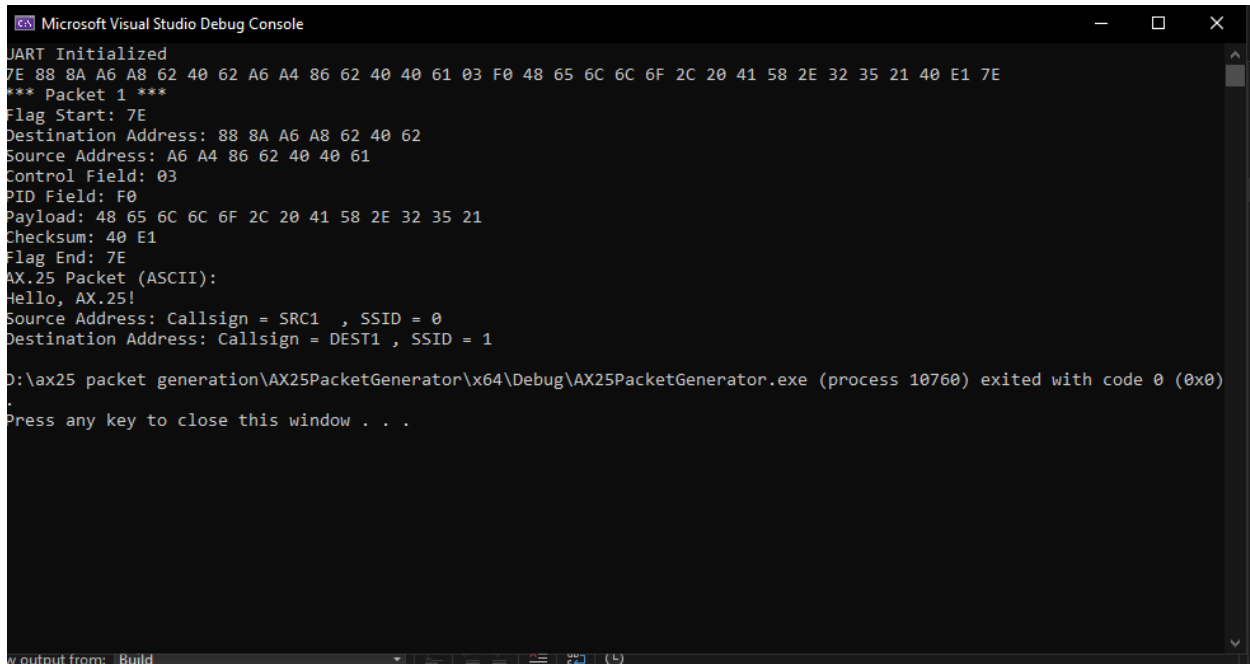
- The SSID is a 4-bit number appended to the callsign, ranging from 0 to 15, providing additional differentiation between stations that may share the same base callsign.
- **Purpose:** SSIDs are particularly useful in scenarios where a single operator may have multiple devices or services, such as a base station (`CALLSIGN-0`), a mobile station (`CALLSIGN-1`), or a digipeater (`CALLSIGN-7`).
- **Example Usage:** If a user operates multiple devices, each device can be assigned a different SSID under the same callsign, allowing for precise identification within the network.

- **Significance in AX.25:**

- The SSID enhances the addressing system's flexibility, supporting more complex networks with multiple nodes while keeping the address format concise and manageable.
- **In our Project:** For NRF and Arduino implementations, this addressing format helps maintain clear communication paths between multiple devices, even when they are part of a larger, interconnected network.

1. Implementation and Configuration

A simple program generating AX.25 data packet output as shown below:

A screenshot of the Microsoft Visual Studio Debug Console window. The window title is "Microsoft Visual Studio Debug Console". The output text is as follows:

```
JART Initialized
7E 88 8A A6 A8 62 40 62 A6 A4 86 62 40 40 61 03 F0 48 65 6C 6C 6F 2C 20 41 58 2E 32 35 21 40 E1 7E
*** Packet 1 ***
Flag Start: 7E
Destination Address: 88 8A A6 A8 62 40 62
Source Address: A6 A4 86 62 40 40 61
Control Field: 03
PID Field: F0
Payload: 48 65 6C 6C 6F 2C 20 41 58 2E 32 35 21
Checksum: 40 E1
Flag End: 7E
AX.25 Packet (ASCII):
Hello, AX.25!
Source Address: Callsign = SRC1 , SSID = 0
Destination Address: Callsign = DEST1 , SSID = 1

D:\ax25 packet generation\AX25PacketGenerator\x64\Debug\AX25PacketGenerator.exe (process 10760) exited with code 0 (0x0)
Press any key to close this window . . .
```

Software Tools and Libraries

Microsoft Visual studio with GCC compiler was used for the testing of multiple data packets.

- **Libraries Used:**

- **RadioHead Library:** Commonly used with Arduino and NRF modules, this library supports various types of packetized data communication, including AX.25-like protocols.
- **AES Libraries:** Lightweight library is used which provide encryption and decryption functionality, which is crucial for enhancing security in data communication over radio. Library can be found on <https://github.com/kokke/tiny-AES-c> Including AES 128.

- **Sample Code Snippets:**

```
#include <stdio.h>
#include <string.h>
#include <stdint.h>
#include "aes.h"
```

```
// Define the CRC-16-CCITT polynomial
#define POLY 0x1021
```

```
// Define the maximum frame size
#define MAX_FRAME_SIZE 32
#define HEADER_SIZE (1 + 2 + 2 + 2 + 2 + 1) // Start Flag (1) +
#define PAYLOAD_SIZE (MAX_FRAME_SIZE - HEADER_SIZE) // Calculat
```

```
// Define the packet structure
typedef struct {
    uint8_t start_flag;           // Start flag (0x7E)
    uint16_t opcode;              // Opcode (2 bytes)
    uint16_t packet_count;        // Packet count (2 bytes)
    uint16_t expected_count;      // Expected packet count (2 bytes)
    uint8_t data[PAYLOAD_SIZE];   // Data payload
    uint16_t checksum;            // CRC-16 checksum (2 bytes)
    uint8_t end_flag;             // End flag (0x7E)
} AX25Frame;
```

```
// AES key (16 bytes)
static const uint8_t AES_KEY[16] = "your-16-byte-key"; // Replac
```

```
// Function prototypes
uint16_t calculate_crc(const uint8_t* data, size_t len);
void fragment_data(const uint8_t* data, size_t len, AX25Frame fr
void defragment_data(const AX25Frame frames[], int frame_count,
void print_bytes(const uint8_t* data, size_t len);
```

```
// Function to calculate CRC-16-CCITT
uint16_t calculate_crc(const uint8_t* data, size_t len) {
    uint16_t crc = 0xFFFF;
    for (size_t i = 0; i < len; i++) {
```

```

crc ^= (uint16_t)data[i] << 8;
for (int j = 0; j < 8; j++) {
if (crc & 0x8000) {
crc = (crc << 1) ^ POLY;
}
else {
crc <<= 1;
}
}
}
return crc;
}

```

```

// Function to fragment data into frames with AES encryption
void fragment_data(const uint8_t* data, size_t len, AX25Frame fi
*frame_count = (len + PAYLOAD_SIZE - 1) / PAYLOAD_SIZE; // Calcula

```

```

struct AES_ctx aes_ctx;
AES_init_ctx(&aes_ctx, AES_KEY);

for (int i = 0; i < *frame_count; i++) {
    frames[i].start_flag = 0x7E; // Start flag
    frames[i].opcode = 0x1234; // Example opcode
    frames[i].packet_count = *frame_count - i; // Descending packet count
    frames[i].expected_count = *frame_count; // Expected packet count

    size_t data_len = (i == *frame_count - 1) ? (len % PAYLOAD_SIZE) : PAYLOAD_SIZE; // Last frame size
    if (data_len == 0) data_len = PAYLOAD_SIZE; // Fix for last frame being exactly PAYLOAD_SIZE bytes
}

```



```

        memcpy(frames[i].data, &data[i * PAYLOAD_SIZE], data_len); // Copy data payload

        if (data_len < PAYLOAD_SIZE) {
            memset(frames[i].data + data_len, 0, PAYLOAD_SIZE - data_len); // Zero out remaining space
        }

        // Print data before encryption
        printf("\nData before encryption (Frame %d):\n", i + 1);
        print_bytes(frames[i].data, PAYLOAD_SIZE);

        // Encrypt the payload
        AES_ECB_encrypt(&aes_ctx, frames[i].data);

        // Print data after encryption
        printf("\nData after encryption (Frame %d):\n", i + 1);
        print_bytes(frames[i].data, PAYLOAD_SIZE);

        // Calculate CRC
        frames[i].checksum = calculate_crc((uint8_t*)&frames[i],
        HEADER_SIZE - sizeof(frames[i].checksum) - 1);
        frames[i].end_flag = 0x7E; // End flag

        // Print each fragmented frame's data
        printf("\nFrame %d (Total bytes: %d):\n", i + 1, MAX_FRAME_SIZE);
        print_bytes((uint8_t*)&frames[i], MAX_FRAME_SIZE);
    }
}

```

```

// Function to defragment frames into original data with AES decryption
void defragment_data(const AX25Frame frames[], int frame_count,

```

```

struct AES_ctx aes_ctx;
AES_init_ctx(&aes_ctx, AES_KEY);

*output_len = 0;
for (int i = 0; i < frame_count; i++) {
    // Print encrypted data before decryption
    printf("\nData before decryption (Frame %d):\n", i +
1);
    print_bytes(frames[i].data, PAYLOAD_SIZE);

    // Decrypt the payload
    AES_ECB_decrypt(&aes_ctx, frames[i].data); // Decrypt the
payload

    // Print data after decryption
    printf("\nData after decryption (Frame %d):\n", i + 1);
    print_bytes(frames[i].data, PAYLOAD_SIZE);

    size_t data_len = (i == frame_count - 1) ? (strlen((const
char*)frames[i].data)) : PAYLOAD_SIZE; // Last frame size
    if (data_len > PAYLOAD_SIZE) data_len = PAYLOAD_SIZE; //
Limit to PAYLOAD_SIZE bytes

    memcpy(&output_data[*output_len], frames[i].data, data_le
n);
    *output_len += data_len;
}

}

// Function to print byte values of data
void print_bytes(const uint8_t* data, size_t len) {
for (size_t i = 0; i < len; i++) {
printf("%02X ", data[i]); // Print each byte as hex

```

```

if ((i + 1) % 16 == 0) printf("\n"); // Break line after 16 bytes
}
printf("\n");
}

```

```

int main() {
uint8_t data[] = "This is a test payload to be fragmented and encrypted";

```

```

// Print bytes before fragmentation
printf("Bytes before fragmentation (Total bytes: %ld):\n", sizeof(data) - 1);
print_bytes(data, sizeof(data) - 1); // Exclude null terminator

```

```

// Fragmentation with AES encryption
AX25Frame frames[8]; // Adjust based on data length and frame size
int frame_count = 0;
fragment_data(data, sizeof(data) - 1, frames, &frame_count);
// Exclude null terminator

```

```

// Print fragmented frames
printf("\n\nFragmented Frames:\n");
for (int i = 0; i < frame_count; i++) {
    printf("Frame %d - Packet Count: %d, CRC: %04x\n", i + 1, frames[i].packet_count, frames[i].checksum);
}

```

```

// Defragmentation with AES decryption
uint8_t defragmented_data[100]; // Adjusted size
size_t defragmented_len = 0;
defragment_data(frames, frame_count, defragmented_data, &defragmented_len);

```

```

// Null-terminate defragmented data
if (defragmented_len < sizeof(defragmented_data)) {
    defragmented_data[defragmented_len] = '\\0'; // Null-term
inate
}
else {
    defragmented_data[sizeof(defragmented_data) - 1] = '\\0';
// Force null-termination
}

// Print defragmented data
printf("\\nBytes after defragmentation (Total bytes: %l
d):\\n", defragmented_len);
print_bytes(defragmented_data, defragmented_len);

// Print defragmented data as a string
printf("\\nDefragmented Data:\\n%s\\n", defragmented_data);

return 0;
}

```

Code Explanation:

1. CRC Calculation:

- The function `calculate_crc` computes a CRC-16-CCITT checksum, which is a cyclic redundancy check used for error detection in data frames. It applies a polynomial (0x1021) to ensure data integrity.

2. Fragmentation:

- The function `fragment_data` breaks up a large input data payload into smaller fragments (frames) while ensuring each frame includes a start flag, opcode, packet count, expected packet count, and checksum. The payload in each frame is AES-encrypted before being sent.

3. Defragmentation:

- The function `defragment_data` reassembles the fragmented frames back into the original data. During this process, it decrypts the AES-encrypted payload of each frame, merges them, and verifies the integrity using the CRC.

4. AES Encryption and Decryption:

- The code uses AES in ECB (Electronic Codebook) mode. Each frame's payload is encrypted using the AES key before transmission (`AES_ECB_encrypt`), and decrypted (`AES_ECB_decrypt`) during defragmentation.

5. Payload Padding:

- If a frame's data payload is smaller than the maximum payload size (due to the last frame being shorter), it is padded with zeros to maintain consistency in frame size.

Functions Used:

1. `calculate_crc` : Computes the CRC-16 checksum for error detection.
2. `fragment_data` : Fragments data into multiple frames, encrypts each frame's payload with AES, and adds necessary metadata (flags, counts, CRC).
3. `defragment_data` : Reassembles the frames into the original data by decrypting the AES-encrypted payload and merging the data.
4. `print_bytes` : Utility function to print data in hexadecimal format for easier debugging and visualization.

Padding Method:

- **Zero Padding:** In the case where the payload of a frame is smaller than the maximum allowed size (`PAYLOAD_SIZE`), it is padded with zeroes (`memset(frames[i].data + data_len, 0, PAYLOAD_SIZE - data_len);`) to fill up the payload.

Encryption, Decryption, Fragmentation, and Defragmentation:

- **Encryption:**

- Each frame's data (payload) is encrypted using AES in ECB mode (`AES_ECB_encrypt` function) with a 16-byte key. ECB mode encrypts blocks independently without chaining.
 - **Decryption:**
 - During defragmentation, the payload of each frame is decrypted using the same AES ECB mode (`AES_ECB_decrypt`).
 - **Fragmentation:**
 - The data is broken down into multiple frames, where each frame carries part of the data. Each frame has headers (start flag, opcode, counts) and a payload. If the data doesn't exactly fit into the frames, the last frame's payload is padded.
 - **Defragmentation:**
 - The frames are reassembled, decrypted, and combined to recover the original data. The CRC is calculated to ensure integrity, and the correct number of frames is expected for reassembly.
-

Size Limits:

- **Maximum Frame Size:** 32 bytes (`MAX_FRAME_SIZE`).
 - **Header Size:** 10 bytes (calculated by `HEADER_SIZE`).
 - **Payload Size:** 22 bytes (`PAYLOAD_SIZE`), derived from subtracting the header size from the maximum frame size.
 - **Number of Frames:**
 - The total number of frames is determined by dividing the data length by the payload size. For example, if data is longer than 22 bytes, it will be split into multiple frames.
-

Packet Structure (AX25Frame):

Each frame consists of:

- **Start Flag** (`start_flag`): Marks the beginning of a frame (0x7E).

- **Opcode (`opcode`)**: A 2-byte field for operational codes, representing different types of packets.
- **Packet Count (`packet_count`)**: A 2-byte field indicating the current packet number.
- **Expected Packet Count (`expected_count`)**: A 2-byte field indicating the total number of packets expected.
- **Data Payload (`data`)**: The actual payload being transferred, which is a maximum of 22 bytes.
- **Checksum (`checksum`)**: A CRC-16 checksum for verifying frame integrity.
- **End Flag (`end_flag`)**: Marks the end of a frame (0x7E).

- **Output Explanation:**

Fragmentation and Encryption:

```

Bytes before fragmentation (Total bytes: 53):
54 68 69 73 20 69 73 20 61 20 74 65 73 74 20 70
61 79 6C 6F 61 64 20 74 6F 20 62 65 20 66 72 61
67 6D 65 6E 74 65 64 20 61 6E 64 20 65 6E 63 72
79 70 74 65 64

Data before encryption (Frame 1):
54 68 69 73 20 69 73 20 61 20 74 65 73 74 20 70
61 79 6C 6F 61 64

Data after encryption (Frame 1):
55 2D 1C 84 FD DB 4E 42 B7 72 2D BD CC CA 3A DF
61 79 6C 6F 61 64

Frame 1 (Total bytes: 32):
7E CC 34 12 03 00 03 00 55 2D 1C 84 FD DB 4E 42
B7 72 2D BD CC CA 3A DF 61 79 6C 6F 61 64 E0 E8

Data before encryption (Frame 2):
20 74 6F 20 62 65 20 66 72 61 67 6D 65 6E 74 65
64 20 61 6E 64 20

Data after encryption (Frame 2):
51 13 7D 5F FC D4 5F C8 3C 70 96 42 27 76 05 44
64 20 61 6E 64 20

Frame 2 (Total bytes: 32):
7E CC 34 12 02 00 03 00 51 13 7D 5F FC D4 5F C8
3C 70 96 42 27 76 05 44 64 20 61 6E 64 20 D0 DF

Data before encryption (Frame 3):
65 6E 63 72 79 70 74 65 64 00 00 00 00 00 00
00 00 00 00 00

Data after encryption (Frame 3):
7C 34 D6 4F 08 89 5C 66 13 06 F4 95 91 2B 07 15
00 00 00 00 00

Frame 3 (Total bytes: 32):
7E CC 34 12 01 00 03 00 7C 34 D6 4F 08 89 5C 66
13 06 F4 95 91 2B 07 15 00 00 00 00 00 00 80 86

```

Defragmentation and Decryption:

```

Fragmented Frames:
Frame 1 - Packet Count: 3, CRC: e8e0
Frame 2 - Packet Count: 2, CRC: dfd0
Frame 3 - Packet Count: 1, CRC: 8680

Data before decryption (Frame 1):
55 2D 1C 84 FD DB 4E 42 B7 72 2D BD CC CA 3A DF
61 79 6C 6F 61 64

Data after decryption (Frame 1):
54 68 69 73 20 69 73 20 61 20 74 65 73 74 20 70
61 79 6C 6F 61 64

Data before decryption (Frame 2):
51 13 7D 5F FC D4 5F C8 3C 70 96 42 27 76 05 44
64 20 61 6E 64 20

Data after decryption (Frame 2):
20 74 6F 20 62 65 20 66 72 61 67 6D 65 6E 74 65
64 20 61 6E 64 20

Data before decryption (Frame 3):
7C 34 D6 4F 00 00 5C 66 13 06 F4 95 91 2B 07 15
00 00 00 00 00 00

Data after decryption (Frame 3):
65 6E 63 72 79 70 74 65 64 00 00 00 00 00 00 00
00 00 00 00 00 00

Bytes after defragmentation (Total bytes: 53):
54 68 69 73 20 69 73 20 61 20 74 65 73 74 20 70
61 79 6C 6F 61 64 20 74 6F 20 62 65 20 66 72 61
67 6D 65 6E 74 65 64 20 61 6E 64 20 65 6E 63 72
79 70 74 65 64

Defragmented Data:
This is a test payload to be fragmented and encrypted

```

Detailed Explanation of the Output:

1. Bytes Before Fragmentation:

- The raw input data is the string: "This is a test payload to be fragmented and encrypted", which is 53 bytes in total.
- This is converted into a hexadecimal format, and the bytes are shown before any fragmentation or encryption is applied.

```

54 68 69 73 20 69 73 20 61 20 74 65 73 74 20 70
61 79 6C 6F 61 64 20 74 6F 20 62 65 20 66 72 61
67 6D 65 6E 74 65 64 20 61 6E 64 20 65 6E 63 72
79 70 74 65 64

```

2. Frame 1 - Data Before and After Encryption:

- The data for the first frame includes the first 22 bytes (the maximum payload size). The unencrypted data in Frame 1 is:

```

54 68 69 73 20 69 73 20 61 20 74 65 73 74 20 70
61 79 6C 6F 61 64

```


- After encryption using AES in ECB mode, the data is transformed into the following encrypted bytes:

```
55 2D 1C 84 FD DB 4E 42 B7 72 2D BD CC CA 3A DF
61 79 6C 6F 61 64
```

3. Frame 1 (Total bytes: 32):

- This frame is then built with headers, payload, and checksum. It contains:
 - Start flag: 0x7E
 - Opcode: 0x1234
 - Packet count: 3
 - Expected count: 3
 - Encrypted payload: 55 2D 1C 84 ...
 - CRC (checksum): e8e0
 - End flag: 0x7E
- The total frame size is 32 bytes as shown:

```
7E CC 34 12 03 00 03 00 55 2D 1C 84 FD DB 4E 42
B7 72 2D BD CC CA 3A DF 61 79 6C 6F 61 64 E0 E8
```

4. Frame 2 - Data Before and After Encryption:

- The second fragment consists of the next 22 bytes of the input string:

```
20 74 6F 20 62 65 20 66 72 61 67 6D 65 6E 74 65
64 20 61 6E 64 20
```

- After encryption, the payload becomes:

```
51 13 7D 5F FC D4 5F C8 3C 70 96 42 27 76 05 44
64 20 61 6E 64 20
```

5. Frame 2 (Total bytes: 32):

- Similar to Frame 1, this frame is created with metadata:
 - Start flag: 0x7E
 - Opcode: 0x1234
 - Packet count: 2
 - Expected count: 3
 - Encrypted payload: 51 13 7D 5F ...
 - CRC: dfd0
 - End flag: 0x7E

```
7E CC 34 12 02 00 03 00 51 13 7D 5F FC D4 5F C8
3C 70 96 42 27 76 05 44 64 20 61 6E 64 20 D0 DF
```

6. Frame 3 - Data Before and After Encryption:

- The third and last frame carries the remaining bytes (9 bytes of actual data), followed by zero-padding to match the 22-byte payload size:

```
65 6E 63 72 79 70 74 65 64 00 00 00 00 00 00 00
00 00 00 00 00 00
```

- After encryption, it becomes:

```
7C 34 D6 4F 0B 89 5C 66 13 06 F4 95 91 2B 07 15
00 00 00 00 00 00
```

7. Frame 3 (Total bytes: 32):

- The frame is constructed with:
 - Start flag: 0x7E
 - Opcode: 0x1234

- Packet count: 1
- Expected count: 3
- Encrypted payload: 7C 34 D6 4F ...
- CRC: 8680
- End flag: 0x7E

```
7E CC 34 12 01 00 03 00 7C 34 D6 4F 0B 89 5C 66
13 06 F4 95 91 2B 07 15 00 00 00 00 00 00 80 86
```

Fragmented Frames Summary:

- Three frames are created:
 - Frame 1: Packet count 3, CRC: e8e0
 - Frame 2: Packet count 2, CRC: dfd0
 - Frame 3: Packet count 1, CRC: 8680

8. Decryption and Defragmentation:

- During defragmentation, the frames are decrypted and combined back into the original data.
- For each frame, the encrypted payload is decrypted back to its original state.

- **Frame 1 Decryption:**

- Decrypted back to:

```
54 68 69 73 20 69 73 20 61 20 74 65 73 74 20 70
61 79 6C 6F 61 64
```

- **Frame 2 Decryption:**

- Decrypted back to:

```
20 74 6F 20 62 65 20 66 72 61 67 6D 65 6E 74 65
64 20 61 6E 64 20
```

- **Frame 3 Decryption:**

- Decrypted back to:

```
65 6E 63 72 79 70 74 65 64 00 00 00 00 00 00 00
00 00 00 00 00 00
```

- The decrypted data is reassembled, and the result is:

This is a test payload to be fragmented and encrypted

How AX.25 Data Packets Are Generated:

1. AX.25 Packet Structure:

- Each packet (frame) follows a structure similar to the AX.25 protocol:
 - **Start Flag** (`start_flag`): Marks the beginning of a frame. The code uses `0x7E` .
 - **Opcode** (`opcode`): A 2-byte field used for distinguishing the type of packet (e.g., `0x1234` as an example here).
 - **Packet Count** (`packet_count`): Indicates the sequence of the current packet in the transmission (e.g., Frame 1 = Packet 3, Frame 2 = Packet 2).
 - **Expected Count** (`expected_count`): Total number of frames expected.
 - **Payload** (`data`): The actual data being transmitted. This is AES-encrypted before sending.
 - **Checksum** (`checksum`): A 2-byte CRC-16 checksum to ensure the integrity of the packet.
 - **End Flag** (`end_flag`): Marks the end of the frame, using `0x7E` .

2. Fragmentation:

- Data is split into smaller chunks (payloads), and each chunk is assigned to a separate frame. Zero-padding is added if necessary to meet the frame size.
- Each frame is then built using the AX.25-like structure, with metadata such as flags, packet counts, and checksums.

3. Encryption:

- Each frame's payload is encrypted using AES in ECB mode before adding it to the frame structure.

4. Decryption and Reassembly:

- Frames are decrypted, verified using CRC checks, and reassembled to form the original message.

Hardware Requirements

• NRF Modules (e.g., NRF24L01):

- Low-cost, low-power wireless modules used for radio communication. The NRF modules are ideal for transmitting AX.25 packets due to their compatibility with microcontroller platforms like Arduino.

• Arduino (e.g., Arduino Uno, Mega):

- Microcontroller platforms that provide a versatile environment for implementing AX.25 communication setups. Arduino is used to control NRF modules and handle packet processing.

• Additional Peripherals:

- UART interfaces, antennas for radio modules, and supporting circuitry to stabilize communication and power delivery.

2. Troubleshooting and Debugging

Common Issues

• Synchronization Problems:

- Can occur when the data stream between transmitting and receiving devices is not properly aligned.
- **Solution:** Ensure UART and other communication settings (baud rate, data format) match on both ends.
- **Signal Interference:**
 - Common in radio communications due to overlapping frequencies or environmental factors.
 - **Solution:** Adjust antenna positions, switch channels, or use filters to reduce noise.
- **Packet Loss:**
 - Loss of data packets can happen due to weak signals or timing mismatches.
 - **Solution:** Increase transmission power, reduce distance between devices, or implement error recovery mechanisms.

Issues faced during building and encryption:

Issues faced/concerns and considerations for anyone who wants to build or improve the project:

- Padding is the top priority issue that was faced, because lightweight aes 128 libraries do not provide padding. the others that provide are not lightweight like openssl in main which makes it difficult for transmission. Sometimes it gives garbage values. sometimes it exceeds the maximum transmission byte limit. All due to padding issue. So, padding in this case is PK57.
- Building ax.25 data packet was a smooth task. Encryption was also smooth, but decryption and maintaining the size limit was complex, and the one that provided padding sometimes gave garbage values, data loss, and inaccurate values. Resolving those errors was a challenge.

Analyzing AX.25 Packets with Wireshark

- **Wireshark Use:**

- Capture and analyze AX.25 packets to diagnose transmission issues. Filters can be set up to focus on specific packet types like UI frames.
- **Example:**
 - Code outputs checksum calculations and data states, which help verify packet integrity during the debugging process.

Logging and Monitoring

- **Tools:**
 - Use logging libraries in Arduino or serial monitors to track packet transmission and reception in real-time.
- **Scripts:**
 - Custom scripts, like those handling packet fragmentation and defragmentation, ensure that all data is correctly processed and reconstructed.

3. Security Considerations

Encryption and Authentication

- **Encryption:**
 - The provided code uses AES-128 encryption to protect data, ensuring confidentiality during transmission.
 - **Example:** The code encrypts data before sending, demonstrated by the output showing both encrypted and decrypted states.

4. Future of AX.25

Developments and Enhancements

- **IoT and Advanced Telemetry:**
 - The AX.25 protocol is increasingly being adapted for IoT applications where small, reliable data packets are crucial. Enhancements focus on optimizing data rates and security features for modern wireless networks.

Alternatives and Comparisons

- **Comparisons:**
 - Protocols like LoRaWAN or Zigbee offer similar functionality but differ in range, bandwidth, and complexity. AX.25 remains relevant due to its simplicity and adaptability in radio networks.

Online Communities and Support

- Forums like QRZ, Stack Overflow, and amateur radio groups offer valuable insights and troubleshooting advice for AX.25 and related radio communication protocols.

5. References and Resources

Technical Documentation

1. AX.25 Protocol Standards:

- **Title:** "AX.25 Link Access Protocol for Amateur Packet Radio"
- **Publisher:** American Radio Relay League (ARRL) and TAPR (Tucson Amateur Packet Radio)
- **Description:** The official standard document detailing the AX.25 protocol, including frame structures, addressing, and operational modes. Essential for understanding how to implement and configure AX.25 in radio communications.

2. Arduino Libraries Documentation:

- **Title:** "Arduino RadioHead Library Documentation"
- **Link:** [RadioHead Library GitHub](#)
- **Description:** Comprehensive guide to the RadioHead library, which supports packet radio protocols like AX.25. It covers installation, API functions, and examples specific to Arduino and compatible radio modules.

3. NRF Module Datasheets:

- **Title:** "NRF24L01+ Product Specification"

- **Publisher:** Nordic Semiconductor
- **Link:** [NRF24L01+ Datasheet](#)
- **Description:** Detailed specifications of the NRF24L01+ module, including pin configurations, operating modes, and communication protocols. This document helps with integrating AX.25 packet handling on NRF modules.

Recommended Readings

1. Packet Radio Books:

- **Title:** "Packet Radio: What? Why? How?"
- **Author:** Ian Wade, G3NRW
- **Description:** A foundational guide for beginners in packet radio, covering the basics of protocols like AX.25, hardware setup, and practical applications. This book is ideal for those starting with radio-based digital communication.

2. Digital Communications:

- **Title:** "The ARRL Handbook for Radio Communications"
- **Publisher:** American Radio Relay League (ARRL)
- **Description:** A comprehensive manual on all aspects of radio communication, including digital modes, packet radio, and advanced AX.25 techniques. It serves as a technical reference for radio enthusiasts and professionals alike.

3. Arduino and Radio Setups:

- **Title:** "Arduino Projects for Amateur Radio"
- **Author:** Jack Purdum, W8TEE, and Dennis Kidder, W6DQ
- **Description:** A book dedicated to building radio-related projects with Arduino, including packet radio setups. It includes hands-on projects that utilize AX.25, offering practical examples and code snippets for hobbyists.

4. Advanced Telemetry and IoT Applications:

- **Title:** "Building Wireless Sensor Networks: with ZigBee, XBee, Arduino, and Processing"
- **Author:** Robert Faludi
- **Description:** While primarily focused on ZigBee and XBee, this book provides insights into setting up wireless networks, which are applicable to AX.25 setups with Arduino and NRF modules.