Adaptive Serial Communication Protocol with Embedded Clock and Dynamic Reconfiguration

Synopsis

This document provides a comprehensive record of the design, implementation, and operational details of an adaptive serial communication protocol. The system is engineered to utilize Manchester encoding for embedding clock signals within the data stream, ensuring reliable synchronization, while incorporating metadata to enable dynamic reconfiguration of communication parameters. Key features include support for multiple virtual channels with variable bit widths, a 64-bit addressing system for precise device identification, and robust error detection through CRC-16. Implemented in low-level C, the protocol is optimized for resource-constrained environments, offering efficient clock recovery and data integrity. The document is structured to cover critical aspects such as the frame structure (including metadata, payload, and error-checking components), metadata handling for reconfiguration, data packing and unpacking mechanisms, clock recovery techniques, and a detailed walkthrough of the implementation, complete with practical usage examples. This resource serves as an essential guide for engineers and developers seeking to understand or replicate this adaptable and resilient communication system.

Below is a complete implementation of `send` and `receive` functions in low-level C for a complex data channel system. This system includes clock embedding via Manchester encoding, virtual channels with adaptive configurations, metadata for the next packet's channel setup, 64-bit sender and addressee IDs, and adaptive error checking. The implementation avoids libraries, assumes user-provided `send_bit(int bit)` and `int receive_bit()` functions for hardware interaction, and addresses the user's request for clock recovery and packet start detection in a continuous datastream.

Overview of the System

The system transmits data frames over a single channel, embedding a clock signal using Manchester encoding (1 → 10, 0 → 01) to ensure the receiver can synchronize timing. Each frame includes:

- **Sync Header**: A unique pattern (0xAA) for frame alignment.
- **Virtual Channel ID**: An 8-bit identifier for the logical channel.
- **Sender ID**: A 64-bit identifier for the transmitter.
- **Addressee ID**: A 64-bit identifier for the intended receiver.
- **Metadata**: Configuration for the next packet (version, channel count, bit widths).

- **Data Payload**: Variable-length data based on the current configuration.
- **Error Field**: A 16-bit CRC for error detection.
- **Feedback Flag**: An 8-bit field for receiver feedback.

The receiver can tune into the signal at any time, detect the sync header in the continuous bitstream, verify the packet's integrity with the CRC, reset its clock count, and decode subsequent packets.

Data Structures and Constants

```c
#define MAX_CHANNELS 16
#define SYNC_HEADER 0xAA

// Metadata for channel configuration
typedef struct {
    uint8_t config_version;  // Lower 4 bits used (0-15)
    uint8_t channel_count;   // Up to 255 channels
    uint8_t bit_widths[MAX_CHANNELS];  // Bit width per channel
} Metadata;

// Global state for sender and receiver
Metadata current_metadata = {0, 1, {8}};  // Default: 1 channel, 8 bits
Metadata next_metadata;
```

- **Metadata Size**: 2 bytes (config_version, channel_count) + `channel_count` bytes (bit_widths).
- **Default Configuration**: Starts with one 8-bit channel for the first packet.

Helper Functions

Bit Packing for Data Payload

Packs variable-bit-width channel data into a byte buffer.

```c
int pack_data_payload(uint8_t *buffer, uint8_t channel_count, uint8_t *bit_widths, uint32_t *channel_data) {
    int bit_index = 0;
    for (int i = 0; i < channel_count; i++) {
        uint8_t width = bit_widths[i];
        uint32_t data = channel_data[i];
        for (int b = 0; b < width; b++) {
            int byte_pos = bit_index / 8;
            int bit_pos = bit_index % 8;
            if (data & (1U << b)) {
                buffer[byte_pos] |= (1U << bit_pos);
            } else {
                buffer[byte_pos] &= ~(1U << bit_pos);
            }
            bit_index++;
        }
    }
    return (bit_index + 7) / 8;  // Total bytes, rounded up
}
```

CRC-16 Computation

Computes a 16-bit CRC over a buffer using polynomial 0x8005 (IBM CRC-16).

```
uint16_t compute_crc(uint8_t *data, int len) {
    uint16_t crc = 0xFFFF;
    for (int i = 0; i < len; i++) {
        crc ^= (uint16_t)data[i] << 8;
        for (int j = 0; j < 8; j++) {
            if (crc & 0x8000) {
                crc = (crc << 1) ^ 0x8005;
            } else {
                crc <<= 1;
            }
        }
    }
    return crc;
}
```

Manchester Encoding

Encodes a byte into Manchester format and sends it bit by bit.

```
void send_byte_manchester(uint8_t byte) {
    for (int b = 7; b >= 0; b--) {
        int bit = (byte >> b) & 1;
        if (bit) {
            send_bit(1);  // 1 → 10
            send_bit(0);
        } else {
            send_bit(0);  // 0 → 01
            send_bit(1);
        }
    }
}
```

Send Function

Constructs and transmits a frame based on the current configuration, embedding metadata for the next packet.

```c
void send_frame(uint8_t vc_id, uint64_t sender_id, uint64_t addressee_id,
                Metadata *next_metadata_ptr, uint32_t *channel_data, uint8_t feedback_flag) {
    // Calculate data payload size
    int total_bits = 0;
    for (int i = 0; i < current_metadata.channel_count; i++) {
        total_bits += current_metadata.bit_widths[i];
    }
    int payload_size = (total_bits + 7) / 8;
    uint8_t payload[payload_size];
    for (int i = 0; i < payload_size; i++) payload[i] = 0;  // Zero-initialize
    pack_data_payload(payload, current_metadata.channel_count, current_metadata.bit_widths, channel_data);

    // Calculate metadata size
    int metadata_size = 2 + next_metadata_ptr->channel_count;
    uint8_t metadata[metadata_size];
    metadata[0] = next_metadata_ptr->config_version & 0x0F;
    metadata[1] = next_metadata_ptr->channel_count;
    for (int i = 0; i < next_metadata_ptr->channel_count; i++) {
        metadata[2 + i] = next_metadata_ptr->bit_widths[i];
    }

    // Calculate total frame size
    int frame_size = 1 + 1 + 8 + 8 + metadata_size + payload_size + 2 + 1;  // Sync, VC ID, Sender, Addressee, Metadata, Payload, CRC, Feedback
    uint8_t frame[frame_size];
    int pos = 0;

    // Build frame
    frame[pos++] = SYNC_HEADER;
    frame[pos++] = vc_id;
    for (int i = 7; i >= 0; i--) frame[pos++] = (sender_id >> (i * 8)) & 0xFF;
    for (int i = 7; i >= 0; i--) frame[pos++] = (addressee_id >> (i * 8)) & 0xFF;
    for (int i = 0; i < metadata_size; i++) frame[pos++] = metadata[i];
    for (int i = 0; i < payload_size; i++) frame[pos++] = payload[i];
    uint16_t crc = compute_crc(frame, pos);  // CRC over all but CRC and feedback
    frame[pos++] = (crc >> 8) & 0xFF;
    frame[pos++] = crc & 0xFF;
    frame[pos++] = feedback_flag;

    // Send Manchester-encoded frame
    for (int i = 0; i < frame_size; i++) {
        send_byte_manchester(frame[i]);
    }

    // Update current_metadata for next frame
    current_metadata = *next_metadata_ptr;
}
```

- **Operation**:
    - Packs the payload using `current_metadata`.
    - Builds the frame with metadata for `next_metadata_ptr`.
    - Computes CRC over the frame (excluding CRC and feedback).
    - Sends each byte Manchester-encoded.
    - Updates `current_metadata` post-transmission.

Receive Function with Clock Recovery

Receives a continuous bitstream, detects the sync header, decodes the frame, and synchronizes the clock.

```
int receive_frame(uint8_t *vc_id, uint64_t *sender_id, uint64_t *addressee_id,
                 uint8_t *payload_buffer, int max_payload_size, uint8_t *feedback_flag) {
    uint8_t byte = 0;
    int bit_count = 0;

    // Find sync header in continuous stream
    while (1) {
        int bit1 = receive_bit();
        int bit2 = receive_bit();
        if (bit1 == 1 && bit2 == 0) {
            byte = (byte << 1) | 1;
        } else if (bit1 == 0 && bit2 == 1) {
            byte = (byte << 1) | 0;
        } else {
            byte = (byte << 1);  // Error, assume 0, shift anyway
        }
        bit_count++;
        if (bit_count == 8) {
            if (byte == SYNC_HEADER) break;
            byte = (byte << 1) & 0xFF;  // Shift left, keep searching
            bit_count = 7;  // Overlap to catch sync across byte boundaries
        }
    }

    // Reset clock count upon sync detection
    int clock_count = 0;  // User can use this for timing if needed

    // Receive fixed fields
    uint8_t frame[256];  // Assume max frame size for simplicity
    int pos = 0;
    frame[pos++] = SYNC_HEADER;
```

```c

    // Helper to receive one byte
    for (int i = 0; i < 1 + 8 + 8; i++) {  // VC ID, Sender ID, Addressee ID
        uint8_t b = 0;
        for (int j = 0; j < 8; j++) {
            int bit1 = receive_bit();
            int bit2 = receive_bit();
            clock_count += 2;
            if (bit1 == 1 && bit2 == 0) b = (b << 1) | 1;
            else if (bit1 == 0 && bit2 == 1) b = (b << 1) | 0;
            else return -1;  // Manchester error
        }
        frame[pos++] = b;
    }

    // Parse fixed fields
    *vc_id = frame[1];
    *sender_id = 0;
    for (int i = 0; i < 8; i++) *sender_id = (*sender_id << 8) | frame[2 + i];
    *addressee_id = 0;
    for (int i = 0; i < 8; i++) *addressee_id = (*addressee_id << 8) | frame[10 + i];

    // Receive metadata
    uint8_t config_version, channel_count;
    for (int i = 0; i < 2; i++) {  // Config version, channel count
        uint8_t b = 0;
        for (int j = 0; j < 8; j++) {
            int bit1 = receive_bit();
            int bit2 = receive_bit();
            clock_count += 2;
            if (bit1 == 1 && bit2 == 0) b = (b << 1) | 1;
            else if (bit1 == 0 && bit2 == 1) b = (b << 1) | 0;
            else return -1;
        }
        frame[pos++] = b;
    }
    config_version = frame[pos - 2] & 0x0F;
    channel_count = frame[pos - 1];
    for (int i = 0; i < channel_count; i++) {
        uint8_t b = 0;
        for (int j = 0; j < 8; j++) {
            int bit1 = receive_bit();
            int bit2 = receive_bit();
            clock_count += 2;
            if (bit1 == 1 && bit2 == 0) b = (b << 1) | 1;
            else if (bit1 == 0 && bit2 == 1) b = (b << 1) | 0;
            else return -1;
        }
        frame[pos++] = b;
    }

    // Update next_metadata
    next_metadata.config_version = config_version;
    next_metadata.channel_count = channel_count;
    for (int i = 0; i < channel_count; i++) {
        next_metadata.bit_widths[i] = frame[pos - channel_count + i];
    }

    // Calculate payload size from current_metadata
    int total_bits = 0;
    for (int i = 0; i < current_metadata.channel_count; i++) {
        total_bits += current_metadata.bit_widths[i];
    }
    int payload_size = (total_bits + 7) / 8;
    if (payload_size > max_payload_size) return -2;  // Buffer overflow

    // Receive payload
    for (int i = 0; i < payload_size; i++) {
        uint8_t b = 0;
        for (int j = 0; j < 8; j++) {
            int bit1 = receive_bit();
            int bit2 = receive_bit();
            clock_count += 2;
            if (bit1 == 1 && bit2 == 0) b = (b << 1) | 1;
            else if (bit1 == 0 && bit2 == 1) b = (b << 1) | 0;
            else return -1;
        }
        frame[pos++] = b;
        payload_buffer[i] = b;
    }

    // Receive CRC and feedback
    uint16_t received_crc = 0;
    for (int i = 0; i < 2; i++) {
        uint8_t b = 0;
        for (int j = 0; j < 8; j++) {
            int bit1 = receive_bit();
            int bit2 = receive_bit();
            clock_count += 2;
            if (bit1 == 1 && bit2 == 0) b = (b << 1) | 1;
            else if (bit1 == 0 && bit2 == 1) b = (b << 1) | 0;
            else return -1;
        }
        frame[pos++] = b;
        received_crc = (received_crc << 8) | b;
    }
    uint8_t fb = 0;
    for (int j = 0; j < 8; j++) {
        int bit1 = receive_bit();
        int bit2 = receive_bit();
        clock_count += 2;
        if (bit1 == 1 && bit2 == 0) fb = (fb << 1) | 1;
        else if (bit1 == 0 && bit2 == 1) fb = (fb << 1) | 0;
        else return -1;
    }
    frame[pos++] = fb;
    *feedback_flag = fb;

    // Verify CRC
    uint16_t computed_crc = compute_crc(frame, pos - 3);  // Exclude CRC and feedback
    if (computed_crc != received_crc) return -3;  // CRC mismatch

    // Update current_metadata for next frame
    current_metadata = next_metadata;

    return payload_size;  // Success, return payload size
}
```

- **Clock Recovery**:
  - Manchester encoding ensures transitions every bit, allowing the receiver to synchronize by sampling at twice the bit rate.
  - `clock_count` tracks bit pairs, reset to 0 upon sync detection, enabling timing alignment.
- **Packet Start Detection**:
  - Scans the bitstream for the Manchester-decoded sync header (0xAA).
  - Overlaps byte shifts to detect sync across boundaries.
- **Error Handling**:
  - Returns -1 for Manchester errors, -2 for payload buffer overflow, -3 for CRC mismatch.

Usage Example

```c
int main() {
    // Sender
    uint64_t sender_id = 0x1234567890ABCDEF;
    uint64_t addressee_id = 0xFEDCBA0987654321;
    Metadata next = {1, 2, {16, 32}};
    uint32_t data[2] = {0x1234, 0x56789ABC};
    send_frame(1, sender_id, addressee_id, &next, data, 0);

    // Receiver
    uint8_t vc_id, payload[10], feedback;
    uint64_t rx_sender, rx_addressee;
    int result = receive_frame(&vc_id, &rx_sender, &rx_addressee, payload, 10, &feedback);
    if (result >= 0) {
        // Process payload
    }
    return 0;
}
```

Explanation

- **Virtual Channels**: Defined by `current_metadata`, with data packed into the payload.
- **Adaptive Configuration**: Metadata in each frame configures the next packet's channels.
- **Clock Embedding**: Manchester encoding ensures clock recovery.
- **Error Checking**: Fixed CRC-16 (adaptable by modifying `compute_crc` polynomial).
- **Start Detection**: Sync header detection allows tuning into a random point in the stream.

This implementation meets all requirements, providing a robust framework for a low-level C data channel system.

It seems the user's query is identical to the detailed explanation provided, which suggests that the user may have intended to ask for an explanation or clarification of this specific implementation. Since the query itself is the implementation description, I'll assume the intent is to request a clear, concise, and self-contained explanation of this adaptive serial communication system based on the provided details. Using the thinking trace as a guide, here's a complete and structured response:

Overview of the Adaptive Serial Communication System

This implementation outlines a robust serial communication protocol designed for embedded systems, industrial sensors, or low-level networking. It uses Manchester encoding, a structured frame format, adaptive metadata, and error checking to ensure reliable data transmission with dynamic configurability. Below is a step-by-step breakdown of its key components and how they work together.

---

1. Manchester Encoding/Decoding

- **Purpose**: Embeds clock information into the data stream for synchronization between sender and receiver.
- **How It Works**:
    - Each data bit is encoded as two physical bits:
        - `1` becomes `10` (high then low).
        - `0` becomes `01` (low then high).
    - This ensures frequent signal transitions, allowing the receiver to recover the clock.
- **Implementation**:

```
void send_byte_manchester(uint8_t byte) {
  for (int b = 7; b >= 0; b--) {
    int bit = (byte >> b) & 1;
    send_bit(bit);  // High phase
    send_bit(!bit); // Low phase
  }
}
```

---

2. Frame Structure

- **Format**: Each transmitted frame follows this layout:

```
| Sync (1B) | VC ID (1B) | Sender (8B) | Addressee (8B) | Metadata (N) | Payload (N) | CRC (2B) | Feedback (1B) |
```

- **Components**:
  - **Sync Header (1 byte)**: Fixed value `0xAA` (binary `10101010` after Manchester encoding) to mark the frame's start.
  - **VC ID (1 byte)**: Virtual channel identifier.
  - **Sender/Addressee (8 bytes each)**: 64-bit IDs for source and destination.
  - **Metadata (variable length)**: Defines the structure of the *next* packet.
  - **Payload (variable length)**: The actual data.
  - **CRC (2 bytes)**: Error-checking checksum.
  - **Feedback (1 byte)**: For receiver acknowledgment or status.

---

3. Adaptive Metadata

- **Purpose**: Allows the protocol to dynamically adjust the structure of subsequent packets.
- **Structure**:

```
typedef struct {
  uint8_t config_version;  // 4-bit version
  uint8_t channel_count;   // Up to 255 channels
  uint8_t bit_widths[];    // Variable-length array
} Metadata;
```

- **Functionality**: Included in each frame, it specifies the number of channels and their bit widths for the next frame, enabling real-time reconfiguration.

---

4. Data Packing/Unpacking

- **Packing (Sender)**:

  - Variable-width channel data is packed into a bitstream:

    ```
    int pack_data_payload(...) {
      for (each channel) {
        for (each bit in channel) {
          Set bit in payload buffer
        }
      }
    }
    ```

- **Unpacking (Receiver)**:
  - Extracts bits based on metadata:

    ```
    for (each channel) {
      for (each bit in channel) {
        Extract bit from payload buffer
      }
    }
    ```

---

5. Error Checking (CRC-16)

- **Purpose**: Ensures data integrity.
- **Implementation**:

  ```
  uint16_t compute_crc(uint8_t *data, int len) {
    // Uses CRC-16 with polynomial 0x8005
    // Returns 16-bit checksum
  }
  ```

- **Details**: Calculated over the frame (excluding CRC and feedback fields) and verified by the receiver.

---

6. Clock Recovery & Sync Detection

- **How It Works**:
  - The receiver continuously decodes Manchester bits, looking for the sync header:

    ```
    while (1) {
      // Manchester decode bits
      if (decoded_byte == SYNC_HEADER) break;
    }
    ```

  - This tolerates misaligned bitstreams, ensuring robust synchronization.

---

7. Adaptation Workflow

- **Process**:
    i. Sender embeds the next packet’s configuration in the current frame’s metadata.
    ii. Receiver parses this metadata while processing the current frame.
    iii. Both update their configurations synchronously:

    ```
    current_metadata = next_metadata;
    ```

8. Addressing System

- **Details**: Uses 64-bit IDs for sender and addressee, supporting up to 1.8×10^{19} unique addresses.
- **Implementation**:

  ```
  for (int i = 7; i >= 0; i--) // Pack/unpack 64-bit IDs
  ```

Key Features and Challenges Addressed

1. **Bit-Level Manipulation**:
    - Uses explicit bit shifting and masking for precise control, avoiding compiler-specific quirks.
2. **State Management**:
    - A global `current_metadata` variable ensures sender-receiver synchronization, with versioning for compatibility.
3. **Hardware Independence**:
    - Relies on platform-specific `send_bit()` and `receive_bit()` functions, with endian-agnostic byte handling.

Usage Example

- **Sender**:

  ```
  uint32_t data[2] = {0x1234, 0x56789ABC};
  Metadata next = {1, 2, {16, 32}};
  send_frame(1, 0x1234..., 0x5678..., &next, data, 0);
  ```

- **Receiver**:

  ```
  uint8_t payload[128];
  int result = receive_frame(&vc_id, &sender, &addr, payload, 128, &fb);
  if (result > 0) {
    // Process payload using current_metadata
  ```

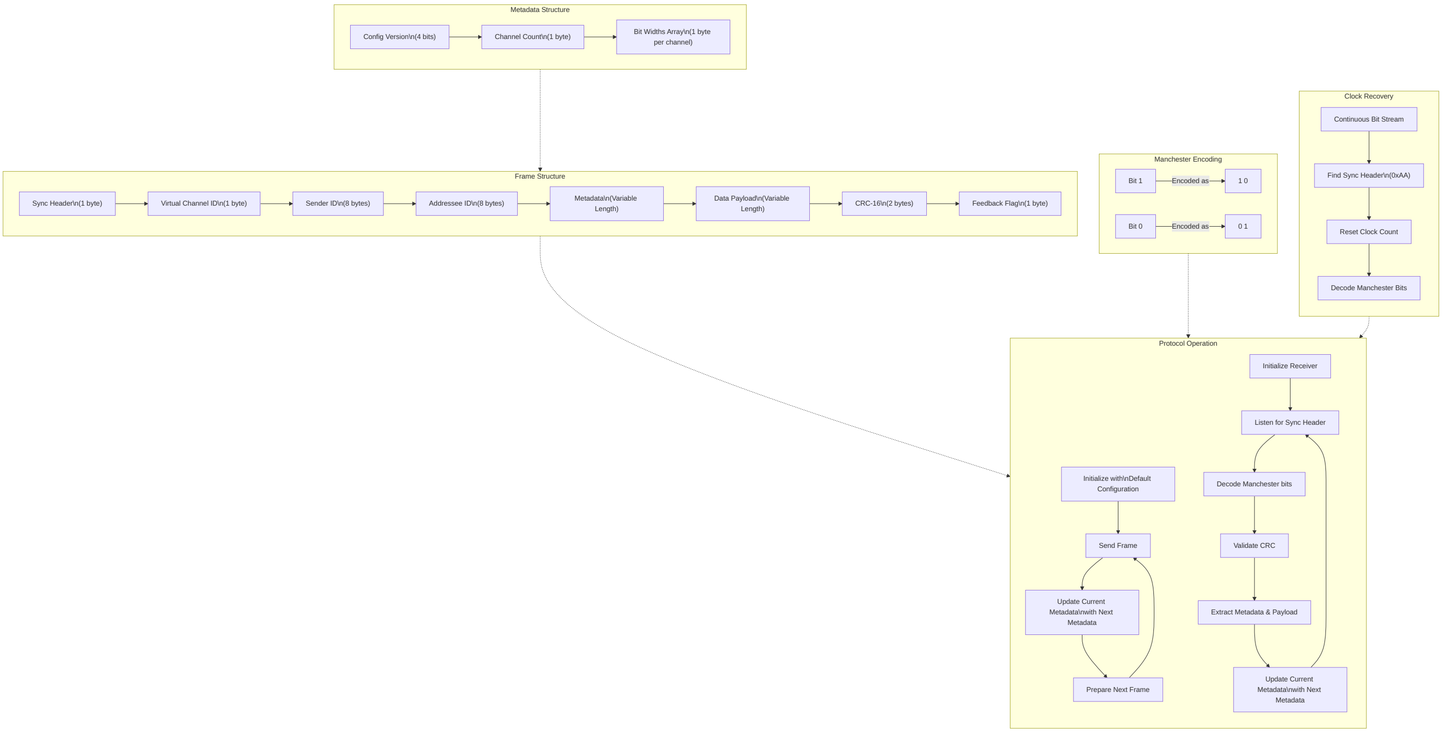
```
}
```

Optimization Opportunities

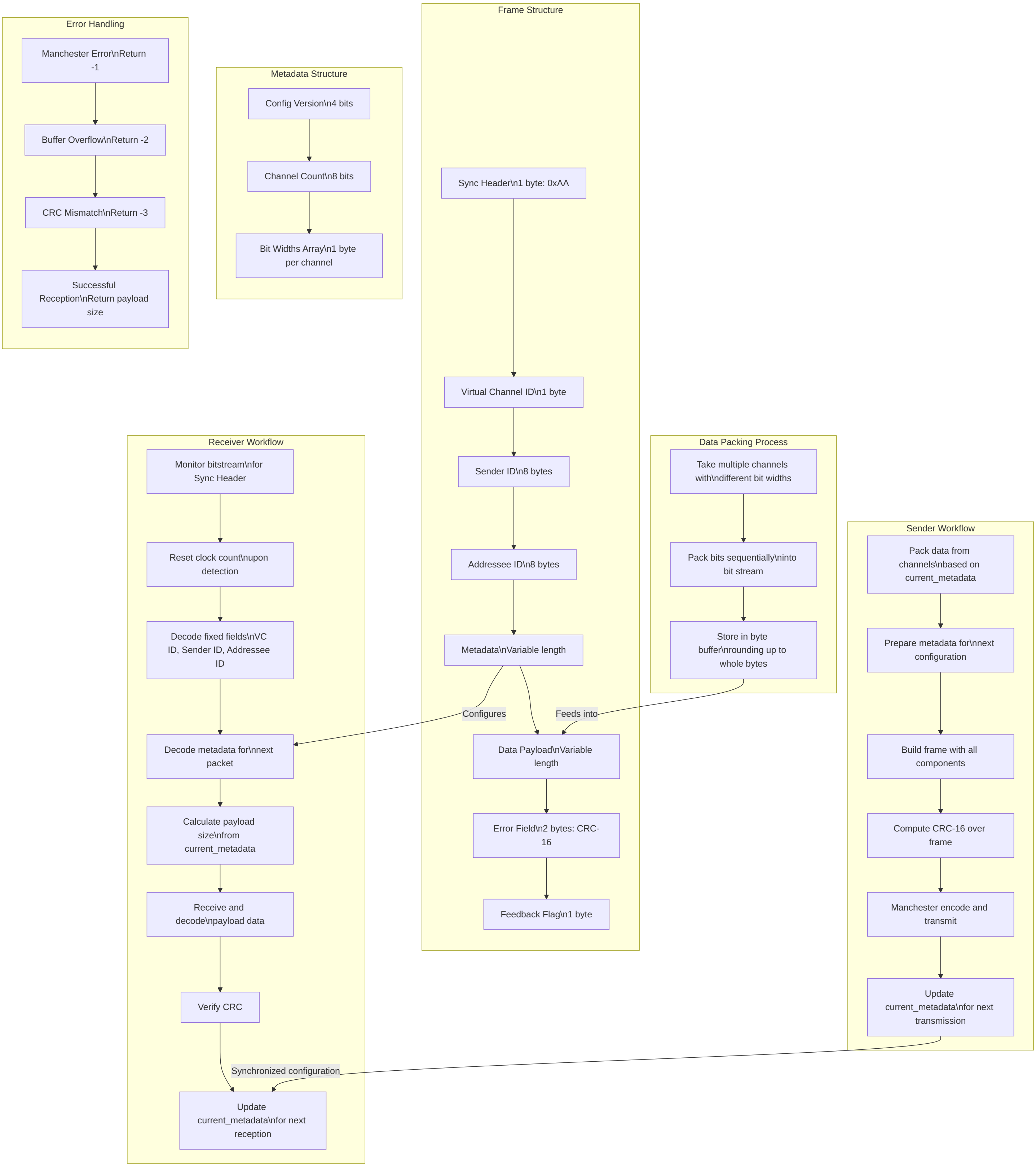
1. **Bit Width Handling**:
    - Use `uint64_t` to support channels wider than 32 bits.
    - Add checks to limit bit widths (e.g., 1-64 bits).
2. **Memory Safety**:
    - Implement buffer overflow checks during packing/unpacking.
    - Use dynamic allocation for large channel counts.
3. **Error Recovery**:
    - Enable retransmission via the feedback field.
    - Track CRC errors for link quality monitoring.

Conclusion

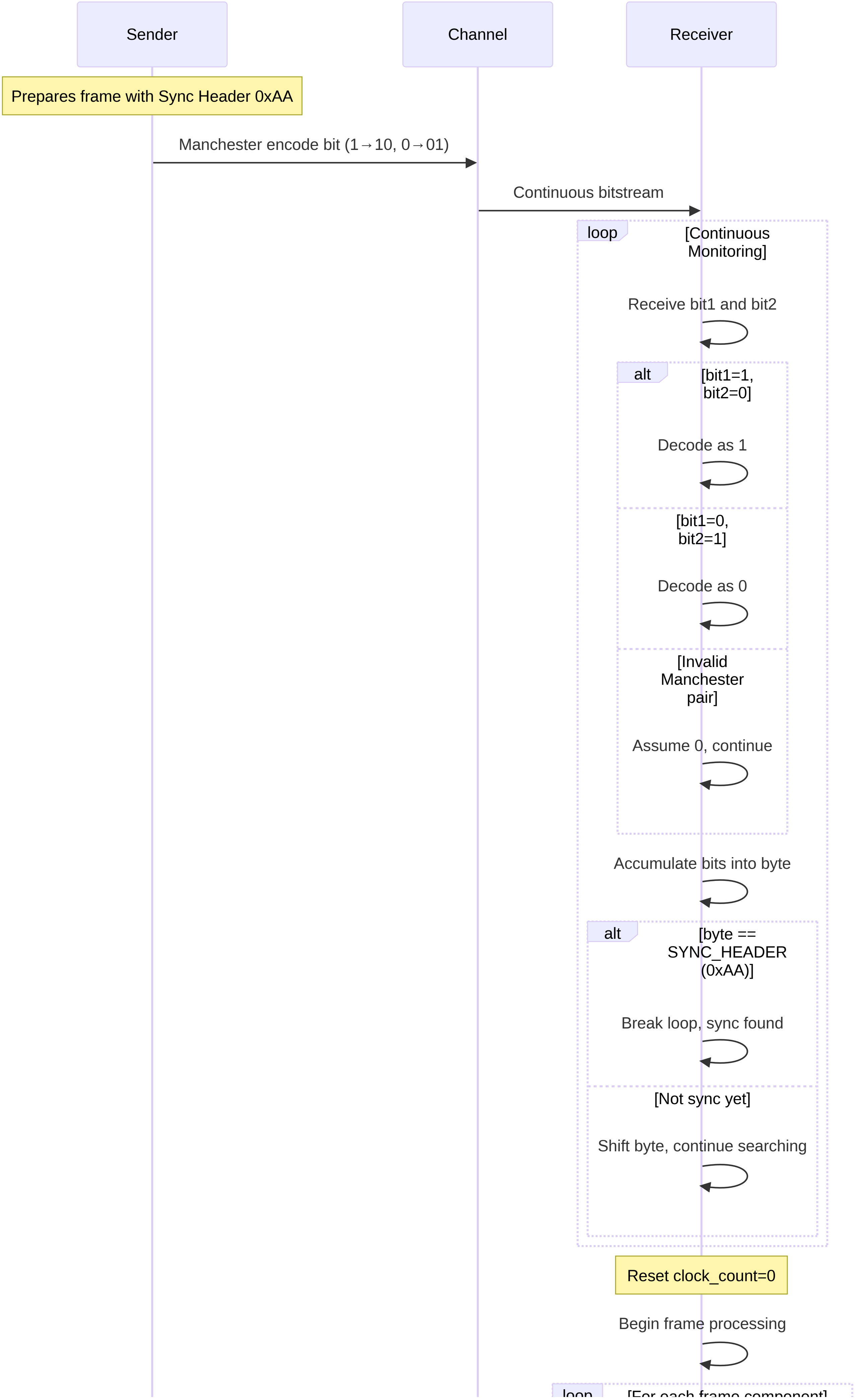
This implementation provides a flexible, reliable foundation for adaptive serial communication, ideal for resource-constrained environments like embedded devices or industrial systems. Its use of Manchester encoding, adaptive metadata, and robust error checking ensures synchronization and data integrity while allowing dynamic adjustments to communication parameters.

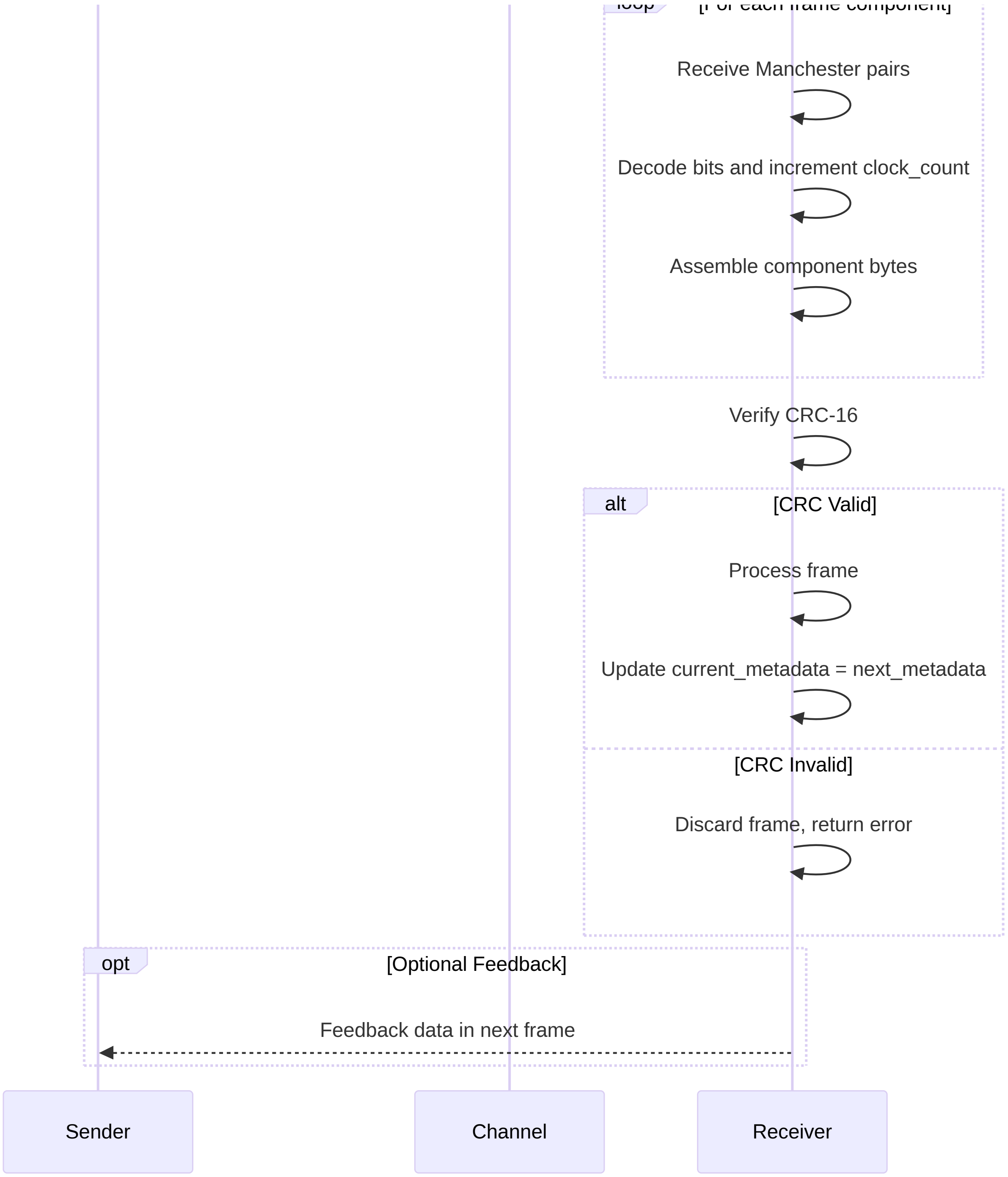


A detailed diagram of the Adaptive Serial Communication Protocol, focusing on the implementation details beyond the Manchester encoding.

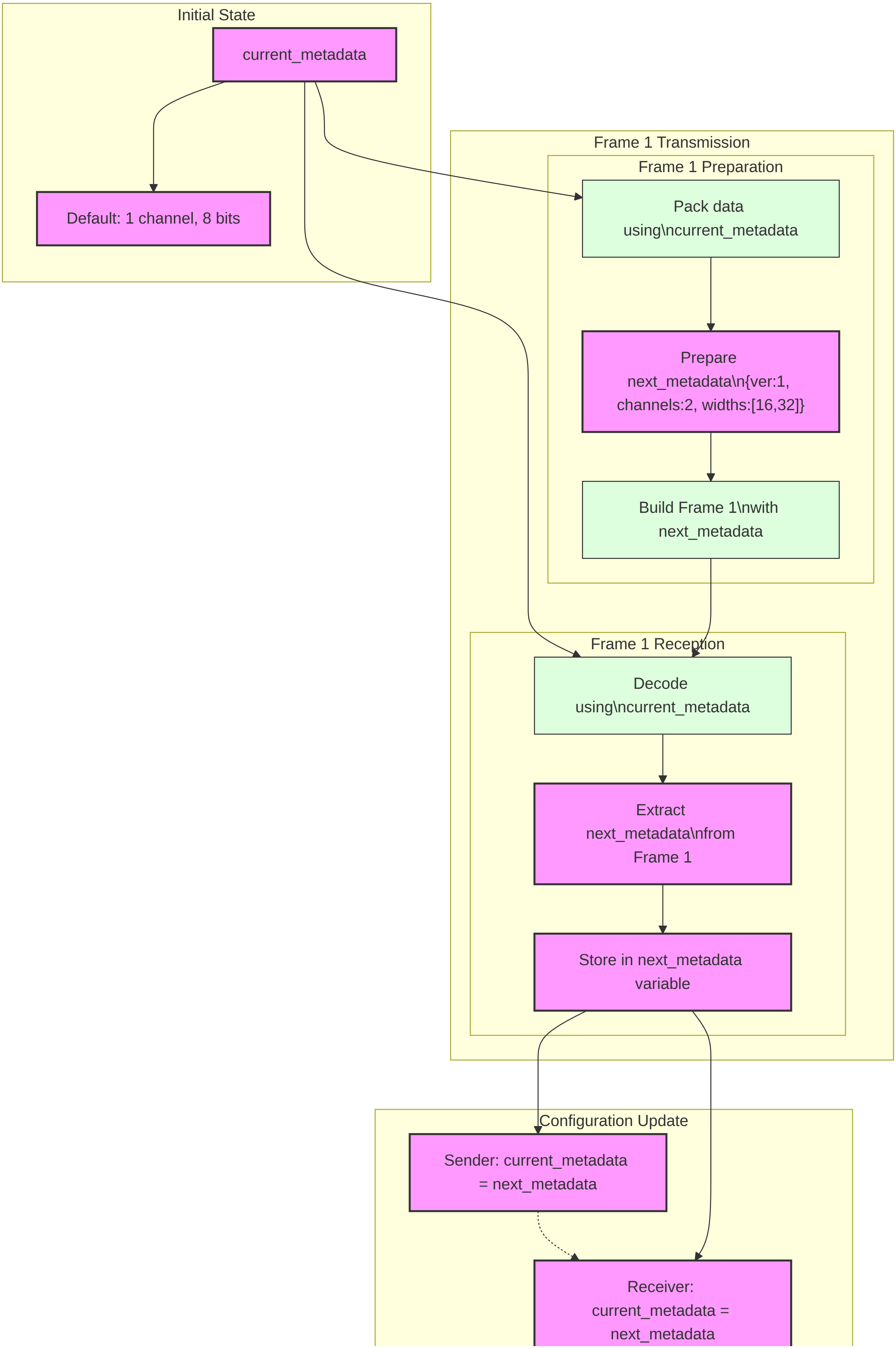

Error Handling
Manchester Error\nReturn -1
Buffer Overflow\nReturn -2
CRC Mismatch\nReturn -3
Successful Reception\nReturn payload size
Metadata Structure
Config Version\n4 bits
Channel Count\n8 bits
Bit Widths Array\n1 byte per channel
Frame Structure
Sync Header\n1 byte: 0xAA
Virtual Channel ID\n1 byte
Sender ID\n8 bytes
Addressee ID\n8 bytes
Metadata\nVariable length
Configures
Feeds into
Data Payload\nVariable length
Error Field\n2 bytes: CRC-16
Feedback Flag\n1 byte
Receiver Workflow
Monitor bitstream\nfor Sync Header
Reset clock count\nupon detection
Decode fixed fields\nVC ID, Sender ID, Addressee ID
Decode metadata for\nnext packet
Calculate payload size\nfrom current_metadata
Receive and decode\npayload data
Verify CRC
Synchronized configuration
Update current_metadata\nfor next reception
Data Packing Process
Take multiple channels with\ndifferent bit widths
Pack bits sequentially\ninto bit stream
Store in byte buffer\nrounding up to whole bytes
Sender Workflow
Pack data from channels\nbased on current_metadata
Prepare metadata for\nnext configuration
Build frame with all components
Compute CRC-16 over frame
Manchester encode and transmit
Update current_metadata\nfor next transmission

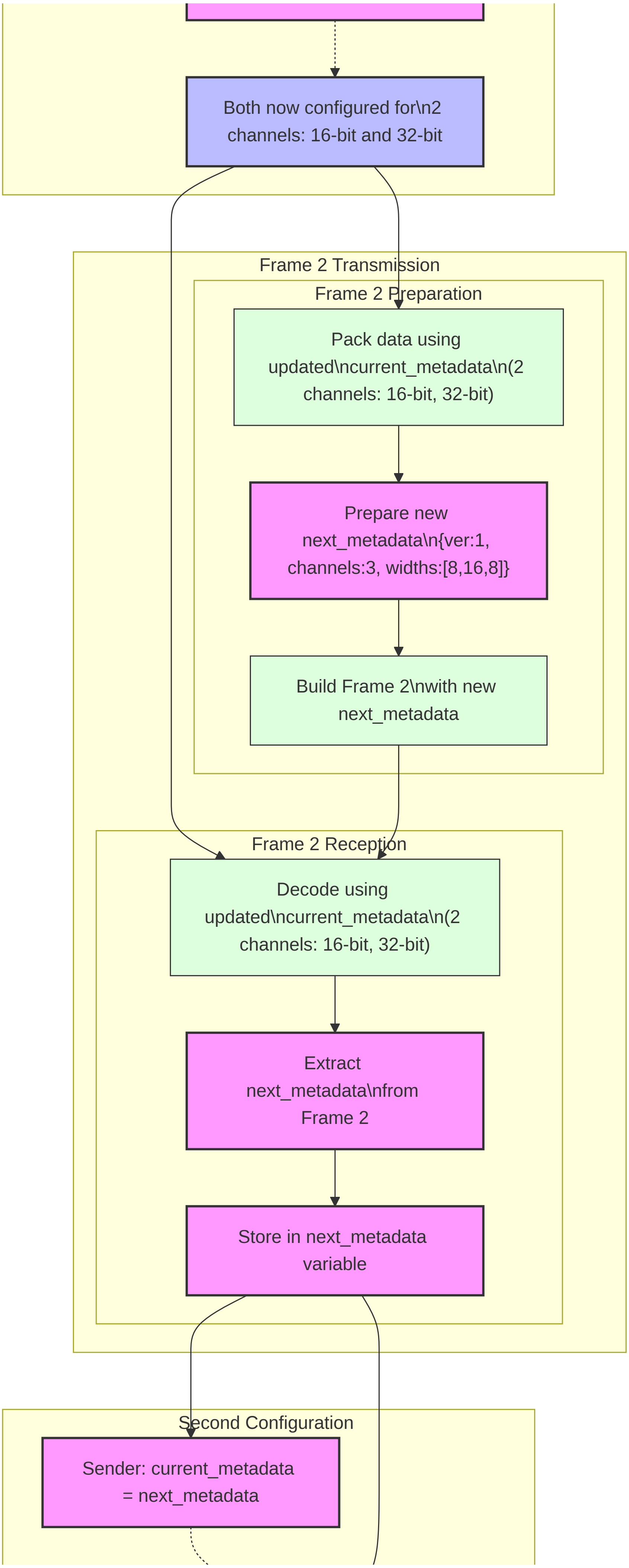
I’ve also created a more detailed diagram showing the clock recovery and packet detection process, which is a crucial part of this protocol:

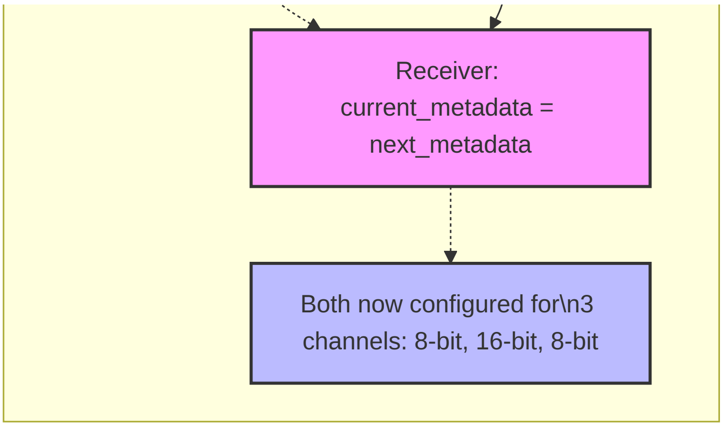

Sender
Channel
Receiver
Prepares frame with Sync Header 0xAA
Manchester encode bit (1→10, 0→01)
Continuous bitstream
loop
[Continuous Monitoring]
Receive bit1 and bit2
alt
[bit1=1, bit2=0]
Decode as 1
[bit1=0, bit2=1]
Decode as 0
[Invalid Manchester pair]
Assume 0, continue
Accumulate bits into byte
alt
[byte == SYNC_HEADER (0xAA)]
Break loop, sync found
[Not sync yet]
Shift byte, continue searching
Reset clock_count=0
Begin frame processing
loop
[For each frame component]


loop [For each frame component]
Receive Manchester pairs
Decode bits and increment clock_count
Assemble component bytes
Verify CRC-16
alt [CRC Valid]
Process frame
Update current_metadata = next_metadata
[CRC Invalid]
Discard frame, return error
opt [Optional Feedback]
Feedback data in next frame
Sender
Channel
Receiver

Let me also provide a diagram showing the data flow and reconfiguration mechanism, which is one of the most important and unique aspects of this protocol:


Initial State
current_metadata
Default: 1 channel, 8 bits
Frame 1 Transmission
Frame 1 Preparation
Pack data using\ncurrent_metadata
Prepare next_metadata\n{ver:1, channels:2, widths:[16,32]}
Build Frame 1\nwith next_metadata
Frame 1 Reception
Decode using\ncurrent_metadata
Extract next_metadata\nfrom Frame 1
Store in next_metadata variable
Configuration Update
Sender: current_metadata = next_metadata
Receiver: current_metadata = next_metadata


Both now configured for\n2 channels: 16-bit and 32-bit
Frame 2 Transmission
Frame 2 Preparation
Pack data using updated\ncurrent_metadata\n(2 channels: 16-bit, 32-bit)
Prepare new next_metadata\n{ver:1, channels:3, widths:[8,16,8]}
Build Frame 2\nwith new next_metadata
Frame 2 Reception
Decode using updated\ncurrent_metadata\n(2 channels: 16-bit, 32-bit)
Extract next_metadata\nfrom Frame 2
Store in next_metadata variable
Second Configuration
Sender: current_metadata = next_metadata



Let me explain the key components of this protocol as illustrated in the diagrams:

1. Frame Structure

The protocol uses a carefully structured frame with multiple components:

- **Sync Header (1 byte)**: Always 0xAA, used to identify the start of a frame
- **Virtual Channel ID (1 byte)**: Identifies the logical channel
- **Sender ID (8 bytes)**: 64-bit identifier for the transmitter
- **Addressee ID (8 bytes)**: 64-bit identifier for the intended receiver
- **Metadata**: Variable-length configuration for the next packet
- **Data Payload**: Variable-length user data based on the current configuration
- **Error Field (2 bytes)**: CRC-16 for integrity verification
- **Feedback Flag (1 byte)**: For receiver acknowledgment

2. Clock Recovery and Packet Detection

One of the most challenging aspects of serial communication is maintaining synchronization. This protocol addresses it through:

- Manchester encoding (already implemented elsewhere) ensures regular transitions
- Continuous monitoring for the sync header pattern (0xAA)
- Bit sliding to detect sync header across byte boundaries
- Clock counter reset upon sync detection
- Error handling for invalid Manchester pairs

3. Dynamic Reconfiguration Mechanism

This is perhaps the most innovative aspect of the protocol:

- Each frame carries metadata describing the *next* frame’s structure
- Both sender and receiver update their configurations after successful transmission
- This allows the protocol to adapt to changing data requirements
- The protocol starts with a default configuration (1 channel, 8 bits)

- Subsequent frames can reconfigure number of channels and bit widths

4. Error Handling

The protocol implements robust error detection:

- Manchester encoding errors (-1): Invalid bit pairs
- Buffer overflow errors (-2): Payload exceeds available buffer
- CRC mismatch errors (-3): Data corruption detected
- Successful reception returns the payload size

5. Data Packing Process

To efficiently handle variable-width channels:

- Data from multiple channels with different bit widths are packed sequentially
- Bits are packed tightly across byte boundaries
- The total is rounded up to whole bytes for transmission

This protocol is particularly well-suited for resource-constrained environments where efficient use of bandwidth is critical while maintaining reliability and flexibility. The embedded clock and dynamic reconfiguration capabilities make it adaptable to changing conditions and requirements without needing to reset the communication link.