

# 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

```
1  #define MAX_CHANNELS 16
2  #define SYNC_HEADER 0xAA
3
4  // Metadata for channel configuration
5  typedef struct {
6      uint8_t config_version; // Lower 4 bits used (0-15)
7      uint8_t channel_count;  // Up to 255 channels
8      uint8_t bit_widths[MAX_CHANNELS]; // Bit width per channel
9  } Metadata;
10
11 // Global state for sender and receiver
12 Metadata current_metadata = {0, 1, {8}}; // Default: 1 channel, 8 bits
13 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.

```
1  int pack_data_payload(uint8_t *buffer, uint8_t channel_count, uint8_t
   *bit_widths, uint32_t *channel_data) {
2      int bit_index = 0;
3      for (int i = 0; i < channel_count; i++) {
4          uint8_t width = bit_widths[i];
5          uint32_t data = channel_data[i];
6          for (int b = 0; b < width; b++) {
7              int byte_pos = bit_index / 8;
8              int bit_pos = bit_index % 8;
9              if (data & (1U << b)) {
10                 buffer[byte_pos] |= (1U << bit_pos);
11             } else {
12                 buffer[byte_pos] &= ~(1U << bit_pos);
```

```

13         }
14         bit_index++;
15     }
16 }
17 return (bit_index + 7) / 8; // Total bytes, rounded up
18 }

```

## CRC-16 Computation

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

```

1  uint16_t compute_crc(uint8_t *data, int len) {
2      uint16_t crc = 0xFFFF;
3      for (int i = 0; i < len; i++) {
4          crc ^= (uint16_t)data[i] << 8;
5          for (int j = 0; j < 8; j++) {
6              if (crc & 0x8000) {
7                  crc = (crc << 1) ^ 0x8005;
8              } else {
9                  crc <<= 1;
10             }
11         }
12     }
13     return crc;
14 }

```

## Manchester Encoding

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

```

1  void send_byte_manchester(uint8_t byte) {
2      for (int b = 7; b >= 0; b--) {
3          int bit = (byte >> b) & 1;
4          if (bit) {
5              send_bit(1); // 1 → 10
6              send_bit(0);
7          } else {
8              send_bit(0); // 0 → 01
9              send_bit(1);
10         }
11     }
12 }

```

---

## Send Function

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

```
1 void send_frame(uint8_t vc_id, uint64_t sender_id, uint64_t addressee_id,
2               Metadata *next_metadata_ptr, uint32_t *channel_data, uint8_t
feedback_flag) {
3     // Calculate data payload size
4     int total_bits = 0;
5     for (int i = 0; i < current_metadata.channel_count; i++) {
6         total_bits += current_metadata.bit_widths[i];
7     }
8     int payload_size = (total_bits + 7) / 8;
9     uint8_t payload[payload_size];
10    for (int i = 0; i < payload_size; i++) payload[i] = 0; // Zero-initialize
11    pack_data_payload(payload, current_metadata.channel_count,
current_metadata.bit_widths, channel_data);
12
13    // Calculate metadata size
14    int metadata_size = 2 + next_metadata_ptr->channel_count;
15    uint8_t metadata[metadata_size];
16    metadata[0] = next_metadata_ptr->config_version & 0x0F;
17    metadata[1] = next_metadata_ptr->channel_count;
18    for (int i = 0; i < next_metadata_ptr->channel_count; i++) {
19        metadata[2 + i] = next_metadata_ptr->bit_widths[i];
20    }
21
22    // Calculate total frame size
23    int frame_size = 1 + 1 + 8 + 8 + metadata_size + payload_size + 2 + 1; //
Sync, VC ID, Sender, Addressee, Metadata, Payload, CRC, Feedback
24    uint8_t frame[frame_size];
25    int pos = 0;
26
27    // Build frame
28    frame[pos++] = SYNC_HEADER;
29    frame[pos++] = vc_id;
30    for (int i = 7; i >= 0; i--) frame[pos++] = (sender_id >> (i * 8)) & 0xFF;
31    for (int i = 7; i >= 0; i--) frame[pos++] = (addressee_id >> (i * 8)) &
0xFF;
32    for (int i = 0; i < metadata_size; i++) frame[pos++] = metadata[i];
33    for (int i = 0; i < payload_size; i++) frame[pos++] = payload[i];
34    uint16_t crc = compute_crc(frame, pos); // CRC over all but CRC and
feedback
35    frame[pos++] = (crc >> 8) & 0xFF;
36    frame[pos++] = crc & 0xFF;
37    frame[pos++] = feedback_flag;
38
39    // Send Manchester-encoded frame
40    for (int i = 0; i < frame_size; i++) {
41        send_byte_manchester(frame[i]);
42    }
43
44    // Update current_metadata for next frame
```

```

45     current_metadata = *next_metadata_ptr;
46 }

```

- **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.

```

1  int receive_frame(uint8_t *vc_id, uint64_t *sender_id, uint64_t *addressee_id,
2                    uint8_t *payload_buffer, int max_payload_size, uint8_t
   *feedback_flag) {
3      uint8_t byte = 0;
4      int bit_count = 0;
5
6      // Find sync header in continuous stream
7      while (1) {
8          int bit1 = receive_bit();
9          int bit2 = receive_bit();
10         if (bit1 == 1 && bit2 == 0) {
11             byte = (byte << 1) | 1;
12         } else if (bit1 == 0 && bit2 == 1) {
13             byte = (byte << 1) | 0;
14         } else {
15             byte = (byte << 1); // Error, assume 0, shift anyway
16         }
17         bit_count++;
18         if (bit_count == 8) {
19             if (byte == SYNC_HEADER) break;
20             byte = (byte << 1) & 0xFF; // Shift left, keep searching
21             bit_count = 7; // Overlap to catch sync across byte boundaries
22         }
23     }
24
25     // Reset clock count upon sync detection
26     int clock_count = 0; // User can use this for timing if needed
27
28     // Receive fixed fields
29     uint8_t frame[256]; // Assume max frame size for simplicity
30     int pos = 0;
31     frame[pos++] = SYNC_HEADER;

```

```

32
33 // Helper to receive one byte
34 for (int i = 0; i < 1 + 8 + 8; i++) { // VC ID, Sender ID, Addressee ID
35     uint8_t b = 0;
36     for (int j = 0; j < 8; j++) {
37         int bit1 = receive_bit();
38         int bit2 = receive_bit();
39         clock_count += 2;
40         if (bit1 == 1 && bit2 == 0) b = (b << 1) | 1;
41         else if (bit1 == 0 && bit2 == 1) b = (b << 1) | 0;
42         else return -1; // Manchester error
43     }
44     frame[pos++] = b;
45 }
46
47 // Parse fixed fields
48 *vc_id = frame[1];
49 *sender_id = 0;
50 for (int i = 0; i < 8; i++) *sender_id = (*sender_id << 8) | frame[2 + i];
51 *addressee_id = 0;
52 for (int i = 0; i < 8; i++) *addressee_id = (*addressee_id << 8) |
frame[10 + i];
53
54 // Receive metadata
55 uint8_t config_version, channel_count;
56 for (int i = 0; i < 2; i++) { // Config version, channel count
57     uint8_t b = 0;
58     for (int j = 0; j < 8; j++) {
59         int bit1 = receive_bit();
60         int bit2 = receive_bit();
61         clock_count += 2;
62         if (bit1 == 1 && bit2 == 0) b = (b << 1) | 1;
63         else if (bit1 == 0 && bit2 == 1) b = (b << 1) | 0;
64         else return -1;
65     }
66     frame[pos++] = b;
67 }
68 config_version = frame[pos - 2] & 0x0F;
69 channel_count = frame[pos - 1];
70 for (int i = 0; i < channel_count; i++) {
71     uint8_t b = 0;
72     for (int j = 0; j < 8; j++) {
73         int bit1 = receive_bit();
74         int bit2 = receive_bit();
75         clock_count += 2;
76         if (bit1 == 1 && bit2 == 0) b = (b << 1) | 1;
77         else if (bit1 == 0 && bit2 == 1) b = (b << 1) | 0;
78         else return -1;
79     }
80     frame[pos++] = b;
81 }
82
83 // Update next_metadata

```

```

84     next_metadata.config_version = config_version;
85     next_metadata.channel_count = channel_count;
86     for (int i = 0; i < channel_count; i++) {
87         next_metadata.bit_widths[i] = frame[pos - channel_count + i];
88     }
89
90     // Calculate payload size from current_metadata
91     int total_bits = 0;
92     for (int i = 0; i < current_metadata.channel_count; i++) {
93         total_bits += current_metadata.bit_widths[i];
94     }
95     int payload_size = (total_bits + 7) / 8;
96     if (payload_size > max_payload_size) return -2; // Buffer overflow
97
98     // Receive payload
99     for (int i = 0; i < payload_size; i++) {
100         uint8_t b = 0;
101         for (int j = 0; j < 8; j++) {
102             int bit1 = receive_bit();
103             int bit2 = receive_bit();
104             clock_count += 2;
105             if (bit1 == 1 && bit2 == 0) b = (b << 1) | 1;
106             else if (bit1 == 0 && bit2 == 1) b = (b << 1) | 0;
107             else return -1;
108         }
109         frame[pos++] = b;
110         payload_buffer[i] = b;
111     }
112
113     // Receive CRC and feedback
114     uint16_t received_crc = 0;
115     for (int i = 0; i < 2; i++) {
116         uint8_t b = 0;
117         for (int j = 0; j < 8; j++) {
118             int bit1 = receive_bit();
119             int bit2 = receive_bit();
120             clock_count += 2;
121             if (bit1 == 1 && bit2 == 0) b = (b << 1) | 1;
122             else if (bit1 == 0 && bit2 == 1) b = (b << 1) | 0;
123             else return -1;
124         }
125         frame[pos++] = b;
126         received_crc = (received_crc << 8) | b;
127     }
128     uint8_t fb = 0;
129     for (int j = 0; j < 8; j++) {
130         int bit1 = receive_bit();
131         int bit2 = receive_bit();
132         clock_count += 2;
133         if (bit1 == 1 && bit2 == 0) fb = (fb << 1) | 1;
134         else if (bit1 == 0 && bit2 == 1) fb = (fb << 1) | 0;
135         else return -1;
136     }

```

```

137     frame[pos++] = fb;
138     *feedback_flag = fb;
139
140     // Verify CRC
141     uint16_t computed_crc = compute_crc(frame, pos - 3); // Exclude CRC and
feedback
142     if (computed_crc != received_crc) return -3; // CRC mismatch
143
144     // Update current_metadata for next frame
145     current_metadata = next_metadata;
146
147     return payload_size; // Success, return payload size
148 }

```

- **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

```

1  int main() {
2      // Sender
3      uint64_t sender_id = 0x1234567890ABCDEF;
4      uint64_t addressee_id = 0xFEDCBA0987654321;
5      Metadata next = {1, 2, {16, 32}};
6      uint32_t data[2] = {0x1234, 0x56789ABC};
7      send_frame(1, sender_id, addressee_id, &next, data, 0);
8
9      // Receiver
10     uint8_t vc_id, payload[10], feedback;
11     uint64_t rx_sender, rx_addressee;
12     int result = receive_frame(&vc_id, &rx_sender, &rx_addressee, payload, 10,
&feedback);
13     if (result >= 0) {
14         // Process payload
15     }
16     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:**

```
1 | void send_byte_manchester(uint8_t byte) {
```

```
2   for (int b = 7; b >= 0; b--) {
3       int bit = (byte >> b) & 1;
4       send_bit(bit); // High phase
5       send_bit(!bit); // Low phase
6   }
7 }
```

---

## 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:**

```
1 typedef struct {
2     uint8_t config_version; // 4-bit version
3     uint8_t channel_count;  // Up to 255 channels
4     uint8_t bit_widths[];   // Variable-length array
5 } 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:

```
1  int pack_data_payload(...) {
2      for (each channel) {
3          for (each bit in channel) {
4              Set bit in payload buffer
5          }
6      }
7  }
```

- **Unpacking (Receiver):**

- Extracts bits based on metadata:

```
1  for (each channel) {
2      for (each bit in channel) {
3          Extract bit from payload buffer
4      }
5  }
```

---

## 5. Error Checking (CRC-16)

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

```
1  uint16_t compute_crc(uint8_t *data, int len) {
2      // Uses CRC-16 with polynomial 0x8005
3      // Returns 16-bit checksum
4  }
```

- **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:

```
1  while (1) {
2      // Manchester decode bits
3      if (decoded_byte == SYNC_HEADER) break;
4  }
```

- 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  $1.8 \times 10^{19}$  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:**

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

- **Receiver:**

```
1 | uint8_t payload[128];
2 | int result = receive_frame(&vc_id, &sender, &addr, payload, 128, &fb);
3 | if (result > 0) {
4 |     // 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.