

MeshCore Companion Protocol (BLE API)

Companion Protocol

- **Last Updated:** 2026-03-08
- **Protocol Version:** Companion Firmware v1.12.0+

“ NOTE: This document is still in development. Some information may be inaccurate.

This document provides a comprehensive guide for communicating with MeshCore devices over Bluetooth Low Energy (BLE).

It is platform-agnostic and can be used for Android, iOS, Python, JavaScript, or any other platform that supports BLE.

Official Libraries

Please see the following repos for existing MeshCore Companion Protocol libraries.

- JavaScript: <https://github.com/meshcore-dev/meshcore.js>
- Python: https://github.com/meshcore-dev/meshcore_py

Important Security Note

All secrets, hashes, and cryptographic values shown in this guide are example values only.

- All hex values, public keys and hashes are for demonstration purposes only
- Never use example secrets in production
- Always generate new cryptographically secure random secrets

- Please implement proper security practices in your implementation
- This guide is for protocol documentation only

Table of Contents

1. [BLE Connection](#)
2. [Packet Structure](#)
3. [Commands](#)
4. [Channel Management](#)
5. [Message Handling](#)
6. [Response Parsing](#)
7. [Example Implementation Flow](#)
8. [Best Practices](#)
9. [Troubleshooting](#)

BLE Connection

Service and Characteristics

MeshCore Companion devices expose a BLE service with the following UUIDs:

- **Service UUID:** 6E400001-B5A3-F393-E0A9-E50E24DCCA9E
- **RX Characteristic** (App → Firmware): 6E400002-B5A3-F393-E0A9-E50E24DCCA9E
- **TX Characteristic** (Firmware → App): 6E400003-B5A3-F393-E0A9-E50E24DCCA9E

Connection Steps

1. Scan for Devices

- Scan for BLE devices advertising the MeshCore Service UUID
- Optionally filter by device name (typically contains "MeshCore" prefix)
- Note the device MAC address for reconnection

1. Connect to GATT

- Connect to the device using the discovered MAC address
- Wait for connection to be established

1. Discover Services and Characteristics

- Discover the service with UUID `6E400001-B5A3-F393-E0A9-E50E24DCCA9E`
- Discover the RX characteristic `6E400002-B5A3-F393-E0A9-E50E24DCCA9E`
- Your app writes to this, the firmware reads from this
- Discover the TX characteristic `6E400003-B5A3-F393-E0A9-E50E24DCCA9E`
- The firmware writes to this, your app reads from this

1. Enable Notifications

- Subscribe to notifications on the TX characteristic to receive data from the firmware

1. Send Initial Commands

- Send `CMD_APP_START` to identify your app to firmware and get radio settings
- Send `CMD_DEVICE_QUERY` to fetch device info and negotiate supported protocol versions
- Send `CMD_SET_DEVICE_TIME` to set the firmware clock
- Send `CMD_GET_CONTACTS` to fetch all contacts
- Send `CMD_GET_CHANNEL` multiple times to fetch all channel slots
- Send `CMD_SYNC_NEXT_MESSAGE` to fetch the next message stored in firmware
- Setup listeners for push codes, such as `PUSH_CODE_MSG_WAITING` or `PUSH_CODE_ADVERT`
- See [Commands](#) section for information on other commands

Note: MeshCore devices may disconnect after periods of inactivity. Implement auto-reconnect logic with exponential backoff.

BLE Write Type

When writing commands to the RX characteristic, specify the write type:

- **Write with Response** (default): Waits for acknowledgment from device
- **Write without Response:** Faster but no acknowledgment

Platform-specific:

- **Android:** Use `BluetoothGattCharacteristic.WRITE_TYPE_DEFAULT` or `WRITE_TYPE_NO_RESPONSE`
- **iOS:** Use `CBCharacteristicWriteType.withResponse` or `.withoutResponse`
- **Python (bleak):** Use `write_gatt_char()` with `response=True` or `False`

Recommendation: Use write with response for reliability.

MTU (Maximum Transmission Unit)

The default BLE MTU is 23 bytes (20 bytes payload). For larger commands like `SET_CHANNEL` (50 bytes), you may need to:

1. **Request Larger MTU:** Request MTU of 512 bytes if supported

- Android: `gatt.requestMtu(512)`
- iOS: `peripheral.maximumWriteValueLength(for:)`
- Python (bleak): MTU is negotiated automatically

Command Sequencing

Critical: Commands must be sent in the correct sequence:

1. After Connection:

- Wait for BLE connection to be established
- Wait for services/characteristics to be discovered
- Wait for notifications to be enabled
- Now you can safely send commands to the firmware

1. Command-Response Matching:

- Send one command at a time
- Wait for a response before sending another command
- Use a timeout (typically 5 seconds)
- Match response to command by type (e.g: `CMD_GET_CHANNEL` → `RESP_CODE_CHANNEL_INFO`)

Command Queue Management

For reliable operation, implement a command queue.

Queue Structure:

- Maintain a queue of pending commands
- Track which command is currently waiting for a response
- Only send next command after receiving response or timeout

Error Handling:

- On timeout, clear current command, process next in queue
- On error, log error, clear current command, process next

Packet Structure

The MeshCore protocol uses a binary format with the following structure:

- **Commands:** Sent from app to firmware via RX characteristic
- **Responses:** Received from firmware via TX characteristic notifications

- **All multi-byte integers:** Little-endian byte order (except CayenneLPP which is Big-endian)
- **All strings:** UTF-8 encoding

Most packets follow this format:

```
[Packet Type (1 byte)] [Data (variable length)]
```

The first byte indicates the packet type (see [Response Parsing](#)).

Commands

1. App Start

Purpose: Initialize communication with the device. Must be sent first after connection.

Command Format:

```
Byte 0: 0x01  
Bytes 1-7: Reserved (currently ignored by firmware)  
Bytes 8+: Application name (UTF-8, optional)
```

Example (hex):

```
01 00 00 00 00 00 00 00 6d 63 63 6c 69
```

Response: `PACKET_SELF_INFO` (0x05)

2. Device Query

Purpose: Query device information.

Command Format:

```
Byte 0: 0x16  
Byte 1: 0x03
```

Example (hex):

16 03

Response: `PACKET_DEVICE_INFO` (0x0D) with device information

3. Get Channel Info

Purpose: Retrieve information about a specific channel.

Command Format:

Byte 0: 0x1F

Byte 1: Channel Index (0-7)

Example (get channel 1):

1F 01

Response: `PACKET_CHANNEL_INFO` (0x12) with channel details

4. Set Channel

Purpose: Create or update a channel on the device.

Command Format:

Byte 0: 0x20

Byte 1: Channel Index (0-7)

Bytes 2-33: Channel Name (32 bytes, UTF-8, null-padded)

Bytes 34-49: Secret (16 bytes)

Total Length: 50 bytes

Channel Index:

- Index 0: Reserved for public channels (no secret)
- Indices 1-7: Available for private channels

Channel Name:

- UTF-8 encoded
- Maximum 32 bytes

- Padded with null bytes (0x00) if shorter

Secret Field (16 bytes):

- For **private channels**: 16-byte secret
- For **public channels**: All zeros (0x00)

Example (create channel "YourChannelName" at index 1 with secret):

```
20 01 53 4D 53 00 00 ... (name padded to 32 bytes)
[16 bytes of secret]
```

Note: The 32-byte secret variant is unsupported and returns `PACKET_ERROR`.

Response: `PACKET_OK` (0x00) on success, `PACKET_ERROR` (0x01) on failure

5. Send Channel Message

Purpose: Send a text message to a channel.

Command Format:

```
Byte 0: 0x03
Byte 1: 0x00
Byte 2: Channel Index (0-7)
Bytes 3-6: Timestamp (32-bit little-endian Unix timestamp, seconds)
Bytes 7+: Message Text (UTF-8, variable length)
```

Timestamp: Unix timestamp in seconds (32-bit unsigned integer, little-endian)

Example (send "Hello" to channel 1 at timestamp 1234567890):

```
03 00 01 D2 02 96 49 48 65 6C 6C 6F
```

Response: `PACKET_MSG_SENT` (0x06) on success

6. Send Channel Data Datagram

Purpose: Send binary datagram data to a channel.

Command Format:

Byte 0: 0x3E
Byte 1: Channel Index (0-7)
Byte 2: Path Length (0xFF = flood, otherwise actual path length)
Bytes 3 .. 2+path_len: Path (omitted when path_len == 0xFF)
Next 2 bytes (little-endian): Data Type (`data\_type`, uint16)
Remaining bytes: Binary payload (variable length)

Data Type / Transport Mapping:

- 0x0000 is invalid for this command.
- 0xFFFF (DATA_TYPE_DEV) is the developer namespace for experimenting and developing apps.
- Other non-zero values can be used as assigned application/community namespaces.

Note: Applications that need a timestamp should encode it inside the binary payload.

Limits:

- Maximum payload length is 163 bytes.
- Larger payloads are rejected with PACKET_ERROR.

Response: PACKET_OK (0x00) on success

6. Get Message

Purpose: Request the next queued message from the device.

Command Format:

Byte 0: 0x0A

Example (hex):

0A

Response:

- PACKET_CHANNEL_MSG_RECV (0x08) or PACKET_CHANNEL_MSG_RECV_V3 (0x11) for channel messages
- PACKET_CONTACT_MSG_RECV (0x07) or PACKET_CONTACT_MSG_RECV_V3 (0x10) for contact messages
- PACKET_NO_MORE_MSGS (0x0A) if no messages available

Note: Poll this command periodically to retrieve queued messages. The device may also send `PACKET_MESSAGES_WAITING` (0x83) as a notification when messages are available.

7. Get Battery and Storage

Purpose: Query device battery voltage and storage usage.

Command Format:

Byte 0: 0x14

Example (hex):

14

Response: `PACKET_BATTERY` (0x0C) with battery millivolts and storage information

Channel Management

Channel Types

1. Public Channel

- Uses a publicly known 16-byte key: `8b3387e9c5cdea6ac9e5edbaa115cd72`
- Anyone can join this channel, messages should be considered public
- Used as the default public group chat

1. Hashtag Channels

- Uses a secret key derived from the channel name
- It is the first 16 bytes of `sha256("#test")`
- For example hashtag channel `#test` has the key: `9cd8fcf22a47333b591d96a2b848b73f`
- Used as a topic based public group chat, separate from the default public channel

1. Private Channels

- Uses a randomly generated 16-byte secret key
- Messages should be considered private between those that know the secret
- Users should keep the key secret, and only share with those you want to communicate with

- Used as a secure private group chat

Channel Lifecycle

1. **Set Channel:**

- Fetch all channel slots, and find one with empty name and all-zero secret
- Generate or provide a 16-byte secret
- Send `CMD_SET_CHANNEL` with name and a 16-byte secret

1. **Get Channel:**

- Send `CMD_GET_CHANNEL` with channel index
- Parse `RESP_CODE_CHANNEL_INFO` response

1. **Delete Channel:**

- Send `CMD_SET_CHANNEL` with empty name and all-zero secret
- Or overwrite with a new channel

Message Handling

Receiving Messages

Messages are received via the TX characteristic (notifications). The device sends:

1. **Channel Messages:**

- `PACKET_CHANNEL_MSG_RECV` (0x08) - Standard format
- `PACKET_CHANNEL_MSG_RECV_V3` (0x11) - Version 3 with SNR

1. **Contact Messages:**

- `PACKET_CONTACT_MSG_RECV` (0x07) - Standard format
- `PACKET_CONTACT_MSG_RECV_V3` (0x10) - Version 3 with SNR

1. **Notifications:**

- `PACKET_MESSAGES_WAITING` (0x83) - Indicates messages are queued

Contact Message Format

Standard Format (`PACKET_CONTACT_MSG_RECV`, 0x07):

Byte 0: 0x07 (packet type)
Bytes 1-6: Public Key Prefix (6 bytes, hex)
Byte 7: Path Length
Byte 8: Text Type
Bytes 9-12: Timestamp (32-bit little-endian)
Bytes 13-16: Signature (4 bytes, only if txt_type == 2)
Bytes 17+: Message Text (UTF-8)

V3 Format (`PACKET_CONTACT_MSG_RECV_V3`, 0x10):

Byte 0: 0x10 (packet type)
Byte 1: SNR (signed byte, multiplied by 4)
Bytes 2-3: Reserved
Bytes 4-9: Public Key Prefix (6 bytes, hex)
Byte 10: Path Length
Byte 11: Text Type
Bytes 12-15: Timestamp (32-bit little-endian)
Bytes 16-19: Signature (4 bytes, only if txt_type == 2)
Bytes 20+: Message Text (UTF-8)

Parsing Pseudocode:

```
def parse_contact_message(data):  
    packet_type = data[0]  
    offset = 1  
  
    # Check for V3 format  
    if packet_type == 0x10: # V3  
        snr_byte = data[offset]  
        snr = ((snr_byte if snr_byte < 128 else snr_byte - 256) / 4.0)  
        offset += 3 # Skip SNR + reserved  
  
        pubkey_prefix = data[offset:offset+6].hex()  
        offset += 6  
  
        path_len = data[offset]  
        txt_type = data[offset + 1]  
        offset += 2  
  
        timestamp = int.from_bytes(data[offset:offset+4], 'little')
```

```

offset += 4

# If txt_type == 2, skip 4-byte signature
if txt_type == 2:
    offset += 4

message = data[offset:].decode('utf-8')

return {
    'pubkey_prefix': pubkey_prefix,
    'path_len': path_len,
    'txt_type': txt_type,
    'timestamp': timestamp,
    'message': message,
    'snr': snr if packet_type == 0x10 else None
}

```

Channel Message Format

Standard Format (`PACKET_CHANNEL_MSG_RECV`, 0x08):

Byte 0: 0x08 (packet type)
 Byte 1: Channel Index (0-7)
 Byte 2: Path Length
 Byte 3: Text Type
 Bytes 4-7: Timestamp (32-bit little-endian)
 Bytes 8+: Message Text (UTF-8)

V3 Format (`PACKET_CHANNEL_MSG_RECV_V3`, 0x11):

Byte 0: 0x11 (packet type)
 Byte 1: SNR (signed byte, multiplied by 4)
 Bytes 2-3: Reserved
 Byte 4: Channel Index (0-7)
 Byte 5: Path Length
 Byte 6: Text Type
 Bytes 7-10: Timestamp (32-bit little-endian)
 Bytes 11+: Message Text (UTF-8)

Parsing Pseudocode:

```
def parse_channel_message(data):
    packet_type = data[0]
    offset = 1

    # Check for V3 format
    if packet_type == 0x11: # V3
        snr_byte = data[offset]
        snr = ((snr_byte if snr_byte < 128 else snr_byte - 256) / 4.0)
        offset += 3 # Skip SNR + reserved

    channel_idx = data[offset]
    path_len = data[offset + 1]
    txt_type = data[offset + 2]
    timestamp = int.from_bytes(data[offset+3:offset+7], 'little')
    message = data[offset+7:].decode('utf-8')

    return {
        'channel_idx': channel_idx,
        'timestamp': timestamp,
        'message': message,
        'snr': snr if packet_type == 0x11 else None
    }
```

Sending Messages

Use the `SEND_CHANNEL_MESSAGE` command (see [Commands](#)).

Important:

- Outbound text message length is bounded by frame size: a DM text field is up to ~160 bytes (channel messages: 160 minus advert-name length minus 2 per the companion protocol). Treat ~130–150 characters as a safe practical limit and split longer messages.
- Long messages should be split into chunks
- Include a chunk indicator (e.g., "[1/3] message text")

Response Parsing

Packet Types

Value	Name	Description
0x00	PACKET_OK	Command succeeded
0x01	PACKET_ERROR	Command failed
0x02	PACKET_CONTACT_START	Start of contact list
0x03	PACKET_CONTACT	Contact information
0x04	PACKET_CONTACT_END	End of contact list
0x05	PACKET_SELF_INFO	Device self-information
0x06	PACKET_MSG_SENT	Message sent confirmation
0x07	PACKET_CONTACT_MSG_RECV	Contact message (standard)
0x08	PACKET_CHANNEL_MSG_RECV	Channel message (standard)
0x09	PACKET_CURRENT_TIME	Current time response
0x0A	PACKET_NO_MORE_MSGS	No more messages available
0x0C	PACKET_BATTERY	Battery level
0x0D	PACKET_DEVICE_INFO	Device information
0x10	PACKET_CONTACT_MSG_RECV_V3	Contact message (V3 with SNR)
0x11	PACKET_CHANNEL_MSG_RECV_V3	Channel message (V3 with SNR)
0x12	PACKET_CHANNEL_INFO	Channel information
0x80	PACKET_ADVERTISEMENT	Advertisement packet
0x82	PACKET_ACK	Acknowledgment
0x83	PACKET_MESSAGES_WAITING	Messages waiting notification
0x88	PACKET_LOG_DATA	RF log data (can be ignored)

Parsing Responses

PACKET_OK (0x00):

Byte 0: 0x00
Bytes 1-4: Optional value (32-bit little-endian integer)

PACKET_ERROR (0x01):

Byte 0: 0x01
Byte 1: Error code (optional)

PACKET_CHANNEL_INFO (0x12):

```
Byte 0: 0x12
Byte 1: Channel Index
Bytes 2-33: Channel Name (32 bytes, null-terminated)
Bytes 34-49: Secret (16 bytes)
```

Note: The device returns the 16-byte channel secret in this response.

PACKET_DEVICE_INFO (0x0D):

```
Byte 0: 0x0D
Byte 1: Firmware Version (uint8)
Bytes 2+: Variable length based on firmware version

For firmware version >= 3:
Byte 2: Max Contacts Raw (uint8, actual = value * 2)
Byte 3: Max Channels (uint8)
Bytes 4-7: BLE PIN (32-bit little-endian)
Bytes 8-19: Firmware Build (12 bytes, UTF-8, null-padded)
Bytes 20-59: Model (40 bytes, UTF-8, null-padded)
Bytes 60-79: Version (20 bytes, UTF-8, null-padded)
Byte 80: Client repeat enabled/preferred (firmware v9+)
Byte 81: Path hash mode (firmware v10+)
```

Parsing Pseudocode:

```
def parse_device_info(data):
    if len(data) < 2:
        return None

    fw_ver = data[1]
    info = {'fw_ver': fw_ver}

    if fw_ver >= 3 and len(data) >= 80:
        info['max_contacts'] = data[2] * 2
        info['max_channels'] = data[3]
        info['ble_pin'] = int.from_bytes(data[4:8], 'little')
        info['fw_build'] = data[8:20].decode('utf-8').rstrip('\x00').strip()
        info['model'] = data[20:60].decode('utf-8').rstrip('\x00').strip()
        info['ver'] = data[60:80].decode('utf-8').rstrip('\x00').strip()
```

```
return info
```

PACKET_BATTERY (0x0C):

Byte 0: 0x0C

Bytes 1-2: Battery Voltage (16-bit little-endian, millivolts)

Bytes 3-6: Used Storage (32-bit little-endian, KB)

Bytes 7-10: Total Storage (32-bit little-endian, KB)

Parsing Pseudocode:

```
def parse_battery(data):  
    if len(data) < 3:  
        return None  
  
    mv = int.from_bytes(data[1:3], 'little')  
    info = {'battery_mv': mv}  
  
    if len(data) >= 11:  
        info['used_kb'] = int.from_bytes(data[3:7], 'little')  
        info['total_kb'] = int.from_bytes(data[7:11], 'little')  
  
    return info
```

PACKET_SELF_INFO (0x05):

Byte 0: 0x05

Byte 1: Advertisement Type

Byte 2: TX Power

Byte 3: Max TX Power

Bytes 4-35: Public Key (32 bytes, hex)

Bytes 36-39: Advertisement Latitude (32-bit little-endian, divided by 1e6)

Bytes 40-43: Advertisement Longitude (32-bit little-endian, divided by 1e6)

Byte 44: Multi ACKs

Byte 45: Advertisement Location Policy

Byte 46: Telemetry Mode (bitfield)

Byte 47: Manual Add Contacts (bool)

Bytes 48-51: Radio Frequency (32-bit little-endian, divided by 1000.0)

Bytes 52-55: Radio Bandwidth (32-bit little-endian, divided by 1000.0)

Byte 56: Radio Spreading Factor

Byte 57: Radio Coding Rate

Bytes 58+: Device Name (UTF-8, variable length, no null terminator required)

Parsing Pseudocode:

```
def parse_self_info(data):
    if len(data) < 36:
        return None

    offset = 1
    info = {
        'adv_type': data[offset],
        'tx_power': data[offset + 1],
        'max_tx_power': data[offset + 2],
        'public_key': data[offset + 3:offset + 35].hex()
    }
    offset += 35

    lat = int.from_bytes(data[offset:offset+4], 'little') / 1e6
    lon = int.from_bytes(data[offset+4:offset+8], 'little') / 1e6
    info['adv_lat'] = lat
    info['adv_lon'] = lon
    offset += 8

    info['multi_acks'] = data[offset]
    info['adv_loc_policy'] = data[offset + 1]
    telemetry_mode = data[offset + 2]
    info['telemetry_mode_env'] = (telemetry_mode >> 4) & 0b11
    info['telemetry_mode_loc'] = (telemetry_mode >> 2) & 0b11
    info['telemetry_mode_base'] = telemetry_mode & 0b11
    info['manual_add_contacts'] = data[offset + 3] > 0
    offset += 4

    freq = int.from_bytes(data[offset:offset+4], 'little') / 1000.0
    bw = int.from_bytes(data[offset+4:offset+8], 'little') / 1000.0
    info['radio_freq'] = freq
    info['radio_bw'] = bw
    info['radio_sf'] = data[offset + 8]
    info['radio_cr'] = data[offset + 9]
    offset += 10
```

```
if offset < len(data):
    name_bytes = data[offset:]
    info['name'] = name_bytes.decode('utf-8').rstrip('\x00').strip()

return info
```

PACKET_MSG_SENT (0x06):

Byte 0: 0x06

Byte 1: Route Flag (0 = direct, 1 = flood)

Bytes 2-5: Tag / Expected ACK (4 bytes, little-endian)

Bytes 6-9: Suggested Timeout (32-bit little-endian, milliseconds)

PACKET_ACK (0x82):

Byte 0: 0x82

Bytes 1-4: ACK Code (4 bytes)

Bytes 5-8: Round-trip time (uint32, milliseconds)

Error Codes

PACKET_ERROR (0x01) may include an error code in byte 1:

Error Code	Description
0 / absent	No specific error code provided (the byte is optional; treat as a generic error)
0x01	ERR_CODE_UNSUPPORTED_CMD — unknown or unsupported command byte / sub-command
0x02	ERR_CODE_NOT_FOUND — target not found (channel, contact, message, etc.)
0x03	ERR_CODE_TABLE_FULL — internal queue or table is full, retry later
0x04	ERR_CODE_BAD_STATE — operation not valid in current device state (e.g. iterator already running)
0x05	ERR_CODE_FILE_IO_ERROR — filesystem or storage I/O failure
0x06	ERR_CODE_ILLEGAL_ARG — invalid argument (bad length, out-of-range value, reserved field, etc.)

Note: Error codes may vary by firmware version. Always check byte 1 of `PACKET_ERROR` response.

Frame Handling

BLE implementations enqueue and deliver one protocol frame per BLE write/notification at the firmware layer.

- Apps should treat each characteristic write/notification as exactly one companion protocol frame
- Apps should still validate frame lengths before parsing
- Future transports or firmware revisions may differ, so avoid assuming fixed payload sizes for variable-length responses

Response Handling

1. Command-Response Pattern:

- Send command via RX characteristic
- Wait for response via TX characteristic (notification)
- Match response to command using sequence numbers or command type
- Handle timeout (typically 5 seconds)
- Use command queue to prevent concurrent commands

1. Asynchronous Messages:

- Device may send messages at any time via TX characteristic
- Handle `PACKET_MESSAGES_WAITING` (0x83) by polling `GET_MESSAGE` command
- Parse incoming messages and route to appropriate handlers
- Validate frame length before decoding

1. Response Matching:

- Match responses to commands by expected packet type:
- `APP_START` → `PACKET_SELF_INFO`
- `DEVICE_QUERY` → `PACKET_DEVICE_INFO`
- `GET_CHANNEL` → `PACKET_CHANNEL_INFO`
- `SET_CHANNEL` → `PACKET_OK` or `PACKET_ERROR`
- `SEND_CHANNEL_MESSAGE` → `PACKET_MSG_SENT`
- `GET_MESSAGE` → `PACKET_CHANNEL_MSG_RECV`, `PACKET_CONTACT_MSG_RECV`, or `PACKET_NO_MORE_MSGS`
- `GET_BATTERY` → `PACKET_BATTERY`

1. Timeout Handling:

- Default timeout: 5 seconds per command
- On timeout: Log error, clear current command, proceed to next in queue
- Some commands may take longer (e.g., `SET_CHANNEL` may need 1-2 seconds)
- Consider longer timeout for channel operations

1. Error Recovery:

- On `PACKET_ERROR`: Log error code, clear current command

- On connection loss: Clear command queue, attempt reconnection
- On invalid response: Log warning, clear current command, proceed

Example Implementation Flow

Initialization

```
# 1. Scan for MeshCore device
device = scan_for_device("MeshCore")

# 2. Connect to BLE GATT
gatt = connect_to_device(device)

# 3. Discover services and characteristics
service = discover_service(gatt, "6E400001-B5A3-F393-E0A9-E50E24DCCA9E")
rx_char = discover_characteristic(service, "6E400002-B5A3-F393-E0A9-E50E24DCCA9E")
tx_char = discover_characteristic(service, "6E400003-B5A3-F393-E0A9-E50E24DCCA9E")

# 4. Enable notifications on TX characteristic
enable_notifications(tx_char, on_notification_received)

# 5. Send AppStart command
send_command(rx_char, build_app_start())
wait_for_response(PACKET_SELF_INFO)
```

Creating a Private Channel

```
# 1. Generate 16-byte secret
secret_16_bytes = generate_secret(16) # Use CSPRNG
secret_hex = secret_16_bytes.hex()

# 2. Build SET_CHANNEL command
channel_name = "YourChannelName"
channel_index = 1 # Use 1-7 for private channels
command = build_set_channel(channel_index, channel_name, secret_16_bytes)

# 3. Send command
```

```
send_command(rx_char, command)

response = wait_for_response(PACKET_OK)


# 4. Store secret locally
store_channel_secret(channel_index, secret_hex)
```

Sending a Message

```
# 1. Build channel message command
channel_index = 1
message = "Hello, MeshCore!"
timestamp = int(time.time())
command = build_channel_message(channel_index, message, timestamp)


# 2. Send command
send_command(rx_char, command)
response = wait_for_response(PACKET_MSG_SENT)
```

Receiving Messages

```
def on_notification_received(data):
    packet_type = data[0]

    if packet_type == PACKET_CHANNEL_MSG_RECV or packet_type == PACKET_CHANNEL_MSG_RECV_V3:
        message = parse_channel_message(data)
        handle_channel_message(message)
    elif packet_type == PACKET_MESSAGES_WAITING:
        # Poll for messages
        send_command(rx_char, build_get_message())
```

Best Practices

1. Connection Management:

- Implement auto-reconnect with exponential backoff
- Handle disconnections gracefully
- Store last connected device address for quick reconnection

1. Secret Management:

- Always use cryptographically secure random number generators
- Store secrets securely (encrypted storage)
- Never log or transmit secrets in plain text

1. **Message Handling:**

- Send `CMD_SYNC_NEXT_MESSAGE` when `PUSH_CODE_MSG_WAITING` is received
- Implement message deduplication to avoid display the same message twice

1. **Channel Management:**

- Fetch all channel slots even if you encounter an empty slot
- Ideally save new channels into the first empty slot

1. **Error Handling:**

- Implement timeouts for all commands (typically 5 seconds)
- Handle `RESP_CODE_ERR` responses appropriately

Troubleshooting

Connection Issues

- **Device not found:** Ensure device is powered on and advertising
- **Connection timeout:** Check Bluetooth permissions and device proximity
- **GATT errors:** Ensure proper service/characteristic discovery

Command Issues

- **No response:** Verify notifications are enabled, check connection state
- **Error responses:** Verify command format and check error code
- **Timeout:** Increase timeout value or try again

Message Issues

- **Messages not received:** Poll `GET_MESSAGE` command periodically
- **Duplicate messages:** Implement message deduplication using timestamp/content as a unique id
- **Message truncation:** Send long messages as separate shorter messages