PTP Packet Generator for Testbench

Based on the provided PTP implementation code, I'll create a PTP packet generator that can be used in a testbench to verify the PTP protocol handling. I'll focus on generating a Sync message, which is one of the fundamental PTP message types.

PTP Sync Packet Generator

c

Copy

Download

```c
#include <stdint.h>

#include <string.h>

#include <arith.h>  // From your codebase

#include <msg.h>    // From your codebase


#define PTPD_HEADER_LENGTH 34

#define PTPD_SYNC_LENGTH 44


// Function to generate a PTP Sync message

void generate_ptp_sync_packet(uint8_t *packet_buffer, ptp_clock_t *clock,
time_interval_t *sync_time) {

   msg_header_t header;

   timestamp_t origin_timestamp;


   // Initialize header (using functions from msg.c)

   msg_pack_header(clock, packet_buffer);


   // Set message type to SYNC (0x0)

   packet_buffer[0] = (packet_buffer[0] & 0xF0) | SYNC;


   // Set message length (44 bytes for Sync)
```

```
    uint16_t length = flip16(PTPD_SYNC_LENGTH);
    memcpy(packet_buffer + 2, &length, sizeof(uint16_t));

    // Set sequence ID (using clock's sequence counter)
    uint16_t seq_id = flip16(clock->sent_sync_sequence_id);
    memcpy(packet_buffer + 30, &seq_id, sizeof(uint16_t));

    // Set control field to SYNC (0x0)
    packet_buffer[32] = CTRL_SYNC;

    // Set logSyncInterval
    packet_buffer[33] = clock->port_ds.log_sync_interval;

    // Zero out correction field (8 bytes)
    memset(packet_buffer + 8, 0, 8);

    // Convert internal time to timestamp format
    ptp_time_from_internal(sync_time, &origin_timestamp);

    // Pack the origin timestamp (34-43 bytes)
    uint16_t seconds_msb = flip16(origin_timestamp.seconds_field.msb);
    uint32_t seconds_lsb = flip32(origin_timestamp.seconds_field.lsb);
    uint32_t nanoseconds = flip32(origin_timestamp.nanoseconds_field);

    memcpy(packet_buffer + 34, &seconds_msb, sizeof(uint16_t));
    memcpy(packet_buffer + 36, &seconds_lsb, sizeof(uint32_t));
    memcpy(packet_buffer + 40, &nanoseconds, sizeof(uint32_t));
}
```

```c
}

// Testbench function to verify PTP Sync handling
void test_ptp_sync_handling(ptp_clock_t *clock) {
    uint8_t sync_packet[PTPD_SYNC_LENGTH];
    time_interval_t sync_time;

    // Initialize test values
    sync_time.seconds = 1234567890;
    sync_time.nanoseconds = 500000000;

    // Generate the Sync packet
    generate_ptp_sync_packet(sync_packet, clock, &sync_time);

    // Simulate receiving the packet (using functions from protocol.c)
    time_interval_t recv_time = {1234567890, 500000100}; // Slightly after sync time
    bool isFromSelf = false;

    // This would normally be called from handle() in protocol.c
    on_sync(clock, &recv_time, isFromSelf);

    // Verify the clock state was updated appropriately
    if (clock->waiting_for_followup) {
        printf("Test passed: Clock properly processed Sync message\n");
    } else {
        printf("Test failed: Clock didn't process Sync message correctly\n");
    }
}
```

PTP Follow-Up Packet Generator

c

Copy

Download

```c
void generate_ptp_followup_packet(uint8_t *packet_buffer, ptp_clock_t *clock, time_interval_t *precise_time) {
    msg_header_t header;
    timestamp_t precise_origin_timestamp;

    // Initialize header (using functions from msg.c)
    msg_pack_header(clock, packet_buffer);

    // Set message type to FOLLOW_UP (0x8)
    packet_buffer[0] = (packet_buffer[0] & 0xF0) | FOLLOW_UP;

    // Set message length (44 bytes for FollowUp)
    uint16_t length = flip16(PTPD_FOLLOW_UP_LENGTH);
    memcpy(packet_buffer + 2, &length, sizeof(uint16_t));

    // Set sequence ID (should match the Sync message)
    uint16_t seq_id = flip16(clock->sent_sync_sequence_id - 1);
    memcpy(packet_buffer + 30, &seq_id, sizeof(uint16_t));

    // Set control field to FOLLOW_UP (0x2)
    packet_buffer[32] = CTRL_FOLLOW_UP;

    // Set logSyncInterval
    packet_buffer[33] = clock->port_ds.log_sync_interval;
```

```
    // Convert internal time to timestamp format
    ptp_time_from_internal(precise_time, &precise_origin_timestamp);

    // Pack the precise origin timestamp (34-43 bytes)
    uint16_t seconds_msb = flip16(precise_origin_timestamp.seconds_field.msb);
    uint32_t seconds_lsb = flip32(precise_origin_timestamp.seconds_field.lsb);
    uint32_t nanoseconds = flip32(precise_origin_timestamp.nanoseconds_field);

    memcpy(packet_buffer + 34, &seconds_msb, sizeof(uint16_t));
    memcpy(packet_buffer + 36, &seconds_lsb, sizeof(uint32_t));
    memcpy(packet_buffer + 40, &nanoseconds, sizeof(uint32_t));
}
```

Testbench Main Function

c

Copy

Download

```
int main() {
    ptp_clock_t test_clock;
    ptpd_opts opts;
    foreign_master_record_t
foreign_records[PTPD_DEFAULT_MAX_FOREIGN_RECORDS];

    // Initialize test clock (similar to ptp_startup in startup.c)
    memset(&test_clock, 0, sizeof(ptp_clock_t));
    test_clock.opts = &opts;
    test_clock.foreign_master_ds.records = foreign_records;

    // Set some default options
```

```
    opts.slave_only = false;
    opts.clock_quality.clock_class = PTPD_DEFAULT_CLOCK_CLASS;
    opts.servo.ap = 16;
    opts.servo.ai = 16;
    opts.delay_mechanism = E2E;

    // Initialize the clock (similar to bcm_init_data in bmc.c)
    // This would normally be done by ptp_startup()
    test_clock.port_ds.port_state = PTP_UNCALIBRATED;
    test_clock.port_ds.delay_mechanism = E2E;
    test_clock.port_ds.log_sync_interval = 0; // 1 second sync interval

    // Run the Sync message test
    test_ptp_sync_handling(&test_clock);

    return 0;
}
```

Key Points About the Testbench:

1. **Packet Structure**: The generator creates properly formatted PTP packets according to the IEEE 1588 standard, matching the implementation in msg.c.
2. **Endian Handling**: Uses the flip16() and flip32() functions (assumed to be in your codebase) to handle network byte order.
3. **State Verification**: The testbench verifies that the clock state changes appropriately after receiving a Sync message (should set waiting_for_followup).
4. **Integration**: The packet generation functions are designed to work with your existing PTP implementation, particularly the on_sync() handler in protocol.c.
5. **Extensibility**: The same pattern can be used to generate other PTP message types (Announce, Delay_Req, etc.)

To fully test your PTP implementation, you would want to create similar generators and test functions for all message types and verify all the state transitions and timing calculations.

how does the packet structure look like based of the following code and how many bytes

Based on the provided code (particularly msg.c), I'll explain the PTP packet structure for the different message types and their byte sizes.

General PTP Packet Structure (Header)

All PTP messages share a common 34-byte header:

Offset	**Field**	**Size (bytes)**	**Description**
0	transportSpecific + messageType	1	Nibble for transport (0x1), nibble for message type
1	versionPTP	1	PTP version (0x02 for PTPv2)
2	messageLength	2	Total message length
4	domainNumber	1	PTP domain number
5	reserved	1	
6	flagField	2	Various flags
8	correctionField	8	Time correction (scaled nanoseconds)
16	reserved	4	
20	sourcePortIdentity (clockIdentity)	8	Clock ID of sender

Offset	Field	Size (bytes)	Description
28	sourcePortIdentity (portNumber)	2	Port number of sender
30	sequenceId	2	Message sequence number
32	controlField	1	Message control type
33	logMessageInterval	1	Log2 of message period

Total header size: 34 bytes

Message-Specific Structures

1. Sync Message (Type 0x0)

Total size: 44 bytes (34 header + 10 body)

Body structure (after header):

Offset	Field	Size	Description
34	originTimestamp	10	When sync was sent

2. Follow-Up Message (Type 0x8)

Total size: 44 bytes (34 header + 10 body)

Body structure:

Offset	Field	Size	Description
34	preciseOriginTimestamp	10	Precise sync send time

3. Delay_Req Message (Type 0x1)

Total size: 44 bytes (34 header + 10 body)

Body structure:

Offset	Field	Size	Description
34	originTimestamp	10	When delay_req was sent

4. Delay_Resp Message (Type 0x9)

Total size: 54 bytes (34 header + 20 body)

Body structure:

Offset	Field	Size	Description
34	receiveTimestamp	10	When delay_req was received
44	requestingPortIdentity (clockIdentity)	8	Clock ID of requester
52	requestingPortIdentity (portNumber)	2	Port number of requester

5. Pdelay_Req Message (Type 0x2)

Total size: 54 bytes (34 header + 20 body)

Body structure:

Offset	Field	Size	Description
34	originTimestamp	10	When pdelay_req was sent
44	reserved	10	

6. Pdelay_Resp Message (Type 0x3)

Total size: 54 bytes (34 header + 20 body)

Body structure:

Offset	Field	Size	Description
34	requestReceiptTimestamp	10	When pdelay_req was received
44	requestingPortIdentity (clockIdentity)	8	Clock ID of requester
52	requestingPortIdentity (portNumber)	2	Port number of requester

7. Pdelay_Resp_Follow_Up Message (Type 0xA)

Total size: 54 bytes (34 header + 20 body)

Body structure:

Offset	Field	Size	Description
34	responseOriginTimestamp	10	When pdelay_resp was sent
44	requestingPortIdentity (clockIdentity)	8	Clock ID of requester
52	requestingPortIdentity (portNumber)	2	Port number of requester

8. Announce Message (Type 0xB)

Total size: 64 bytes (34 header + 30 body)

Body structure:

Offset	Field	Size	Description
34	originTimestamp	10	
44	currentUtcOffset	2	
46	reserved	1	

Offset	Field	Size	Description
47	grandmasterPriority1	1	
48	grandmasterClockQuality (clockClass)	1	
49	grandmasterClockQuality (clockAccuracy)	1	
50	grandmasterClockQuality (offsetScaledLogVariance)	2	
52	grandmasterPriority2	1	
53	grandmasterIdentity	8	
61	stepsRemoved	2	
63	timeSource	1	

Timestamp Format (10 bytes)

Used in multiple message types:

c

Copy

Download

```c
struct timestamp_t {
  int16_t seconds_field_msb;  // High 16 bits of seconds
  uint32_t seconds_field_lsb; // Low 32 bits of seconds
  uint32_t nanoseconds;      // Nanoseconds
};
```

Key Observations from the Code:

1. The message lengths are defined as constants in the code:

c

Copy

Download

```
#define PTPD_HEADER_LENGTH 34

#define PTPD_SYNC_LENGTH 44

#define PTPD_FOLLOW_UP_LENGTH 44

#define PTPD_DELAY_REQ_LENGTH 44

#define PTPD_DELAY_RESP_LENGTH 54

#define PTPD_PDELAY_REQ_LENGTH 54

#define PTPD_PDELAY_RESP_LENGTH 54

#define PTPD_PDELAY_RESP_FOLLOW_UP_LENGTH 54

#define PTPD_ANNOUNCE_LENGTH 64
```

2. The code uses network byte order (big-endian) for multi-byte fields, with flip16() and flip32() functions for conversion.
3. The correction field is a 64-bit fixed-point number (16.48 format) for high-precision time adjustments.

This structure matches the IEEE 1588-2008 (PTPv2) standard implementation in your codebase. The packet generators I provided earlier create properly formatted packets according to these specifications.

how should the header look like for each message type

Here's a detailed breakdown of how the PTP header should look for each message type, based on the code implementation:

Common Header Structure (34 bytes)

All PTPv2 messages share this header format, with variations in specific fields:

Byte Offset	**Field Name**	**Size**	**Description**	**Value by Message Type**
0	transportSpecific + messageType	1	High nibble: transport, Low nibble: message type	Varies by message

Byte Offset	Field Name	Size	Description	Value by Message Type
1	versionPTP	1	PTP version (0x02)	Always 0x02
2-3	messageLength	2	Total message length	Varies (44, 54, 64)
4	domainNumber	1	PTP domain	From clock config
5	reserved	1	Must be 0	0x00
6-7	flagField	2	Message flags	Varies
8-15	correctionField	8	Time correction	Usually 0 except FollowUp
16-19	reserved	4	Must be 0	0x00000000
20-27	sourcePortIdentity (clockIdentity)	8	Sender's clock ID	From clock config
28-29	sourcePortIdentity (portNumber)	2	Sender's port number	From clock config
30-31	sequenceId	2	Message sequence number	Incrementing counter
32	controlField	1	Message control type	Varies by message
33	logMessageInterval	1	Log2 of message period	From clock config

Message-Specific Header Configurations

1. Sync (0x0)

text

Copy

Download

[0] = 0x10 (transportSpecific=1, messageType=0)

[32] = 0x00 (controlField=SYNC)

Flags: Two-step flag may be set (bit 0 of flagField[0])

2. Follow_Up (0x8)

text

Copy

Download

[0] = 0x18 (transportSpecific=1, messageType=8)

[32] = 0x02 (controlField=FOLLOW_UP)

CorrectionField: Contains precise correction value

3. Delay_Req (0x1)

text

Copy

Download

[0] = 0x11 (transportSpecific=1, messageType=1)

[32] = 0x01 (controlField=DELAY_REQ)

Flags: Typically no special flags

4. Delay_Resp (0x9)

text

Copy

Download

[0] = 0x19 (transportSpecific=1, messageType=9)

[32] = 0x03 (controlField=DELAY_RESP)

5. Pdelay_Req (0x2)

text

Copy

Download

[0] = 0x12 (transportSpecific=1, messageType=2)

[32] = 0x05 (controlField=PDELAY_REQ)

6. Pdelay_Resp (0x3)

text

Copy

Download

[0] = 0x13 (transportSpecific=1, messageType=3)

[32] = 0x06 (controlField=PDELAY_RESP)

7. Pdelay_Resp_Follow_Up (0xA)

text

Copy

Download

[0] = 0x1A (transportSpecific=1, messageType=10)

[32] = 0x07 (controlField=PDELAY_RESP_FOLLOW_UP)

8. Announce (0xB)

text

Copy

Download

[0] = 0x1B (transportSpecific=1, messageType=11)

[32] = 0x04 (controlField=MANAGEMENT)

Flags: UTC valid, timescale, etc. may be set

Flag Field Details (bytes 6-7)

Common flag bits:

- Bit 0 (byte 6): TWO_STEP (for Sync messages)
- Bit 1 (byte 6): UNICAST

- Bit 8 (byte 7): ALTERNATE_MASTER
- Bit 9 (byte 7): PTP_TIMESCALE
- Bit 10 (byte 7): TIME_TRACEABLE
- Bit 11 (byte 7): FREQUENCY_TRACEABLE

Control Field Values (byte 32)

From the code:

c

Copy

Download

```c
#define CTRL_SYNC 0x00

#define CTRL_DELAY_REQ 0x01

#define CTRL_FOLLOW_UP 0x02

#define CTRL_DELAY_RESP 0x03

#define CTRL_MANAGEMENT 0x04

#define CTRL_OTHER 0x05

#define CTRL_PDELAY_REQ 0x06

#define CTRL_PDELAY_RESP 0x07
```

Example Headers

1. **Sync Header** (Two-step clock):

text

Copy

Download

```text
00: 0x10  // Transport+Type

01: 0x02  // Version

02: 0x00  // Length MSB (44)

03: 0x2C  // Length LSB

04: 0x00  // Domain

05: 0x00  // Reserved
```

06: 0x01 // Flags[0] (TWO_STEP)

07: 0x00 // Flags[1]

08-15: 00 00 00 00 00 00 00 00 // Correction

16-19: 00 00 00 00 // Reserved

20-27: 00 11 22 33 44 55 66 77 // Clock ID

28-29: 00 01   // Port Number

30-31: 00 01   // Sequence ID

32: 0x00   // Control (SYNC)

33: 0x00   // Log interval (1 sec)

2. **Announce Header**:

text

Copy

Download

00: 0x1B // Transport+Type

01: 0x02 // Version

02: 0x00 // Length MSB (64)

03: 0x40 // Length LSB

04: 0x00 // Domain

05: 0x00 // Reserved

06: 0x04 // Flags[0] (PTP_TIMESCALE)

07: 0x00 // Flags[1]

08-15: 00 00 00 00 00 00 00 00 // Correction

16-19: 00 00 00 00 // Reserved

20-27: 00 11 22 33 44 55 66 77 // Clock ID

28-29: 00 01   // Port Number

30-31: 00 01   // Sequence ID

32: 0x04   // Control (MANAGEMENT)

33: 0x00   // Log interval (1 sec)

The code shows these headers are constructed in msg_pack_header() and then modified per message type in the respective packing functions (like msg_pack_sync(), msg_pack_announce(), etc.). The transportSpecific field is hardcoded to 0x1 in the implementation (see msg_pack_header() where it sets nibble_t transport = 0x80 which becomes 0x1 after shift).

how is transport + type decided

In the PTP protocol implementation shown in your code, the **transportSpecific** and **messageType** fields (which combine to form the first byte of the PTP header) are determined as follows:

---

1. Structure of Byte 0

The first byte contains two 4-bit fields:

- **Bits 7-4 (High nibble)**: transportSpecific
- **Bits 3-0 (Low nibble)**: messageType

text

Copy

Download

Byte 0: [transportSpecific (4 bits) | messageType (4 bits)]

---

2. How These Fields Are Set in Your Code

A. TransportSpecific (High nibble)

- **Default value**: 0x1 (PTP over UDP/IPv4 or IPv6, as per IEEE 1588-2008 Annex D)
- **Set in** msg_pack_header() (from msg.c):

c

Copy

Download

nibble_t transport = 0x80; // Binary: 1000 0000

(uint8_t)(buf + 0) = transport; // Sets high nibble to 0x1 (after shift)

  - The value 0x80 is shifted right by 4 bits (due to nibble packing), resulting in 0x1.

B. MessageType (Low nibble)

- Defined by the message type (e.g., SYNC, DELAY_REQ, FOLLOW_UP, etc.)
- **Set in each message packing function** (e.g., msg_pack_sync(), msg_pack_announce()):

c

Copy

Download

```
*(char*)(buf + 0) = *(char*)(buf + 0) & 0xF0;  // Clear low nibble (keep transport)
*(char*)(buf + 0) = *(char*)(buf + 0) | SYNC;  // Set message type (e.g., SYNC=0x0)
```

---

3. Message Type Values (From msg.c)

The message types are defined as enum values (likely in ptpd.h):

Message Type	Value (Hex)	Description
SYNC	0x0	Synchronization message
DELAY_REQ	0x1	Delay Request
PDELAY_REQ	0x2	Peer Delay Request
PDELAY_RESP	0x3	Peer Delay Response
FOLLOW_UP	0x8	Follow-Up to Sync
DELAY_RESP	0x9	Delay Response
PDELAY_RESP_FOLLOW_UP	0xA	Peer Delay Follow-Up
ANNOUNCE	0xB	Announcement (for BMC)
SIGNALING	0xC	Signaling message

Message Type	Value (Hex)	Description
MANAGEMENT	0xD	Management message

---

4. Final Byte 0 Values for Each Message

Combining transportSpecific=0x1 with each messageType:

Message Type	Byte 0 (Hex)	Binary Breakdown
SYNC	0x10	0001 0000
DELAY_REQ	0x11	0001 0001
PDELAY_REQ	0x12	0001 0010
PDELAY_RESP	0x13	0001 0011
FOLLOW_UP	0x18	0001 1000
DELAY_RESP	0x19	0001 1001
PDELAY_RESP_FOLLOW_UP	0x1A	0001 1010
ANNOUNCE	0x1B	0001 1011
SIGNALING	0x1C	0001 1100
MANAGEMENT	0x1D	0001 1101

---

5. Why transportSpecific=0x1?

- Defined in **IEEE 1588-2008 Annex D**:
  - 0x0: Reserved

- 0x1: Default (UDP/IPv4, UDP/IPv6)
- 0x2: IEEE 802.3/Ethernet
- 0x3: DeviceNet
- 0x4: ControlNet
- 0x5: PROFINET

Your code hardcodes it to 0x1 because it assumes UDP/IP transport.

---

6. Summary

- **Byte 0 =** (transportSpecific << 4) | messageType
- **Default** transportSpecific **=** 0x1 **(UDP/IP)**
- messageType **depends on the PTP message (e.g.,** 0x0 **for SYNC,** 0x8 **for FOLLOW_UP)**
- **Example: SYNC message has Byte 0 =** 0x10 **(**0001 0000**)**

This matches the PTPv2 standard and ensures proper interpretation by PTP-aware devices.

so if sync message how will it look like

Here's how a **PTP Sync message (Message Type 0x0)** would look in your implementation, based on the code:

Sync Message Structure (44 bytes total)

Byte Range	Field	Value (Example)	Description
0	transportSpecific + messageType	0x10	0x1 (UDP/IP) + 0x0 (Sync)
1	versionPTP	0x02	PTPv2
2-3	messageLength	0x002C	44 bytes (flipped for

Byte Range	Field	Value (Example)	Description
			network byte order)
4	domainNumber	0x00	Default domain 0
5	Reserved	0x00	Must be zero
6-7	flagField	0x0001	TWO_STEP flag set (if applicable)
8-15	correctionField	0x0000000000000000	Zero for Sync (updated in FollowUp)
16-19	Reserved	0x00000000	Must be zero
20-27	sourcePortIdentity (clockIdentity)	00:11:22:33:44:55:66:77	Sender's clock ID (EUI-64)
28-29	sourcePortIdentity (portNumber)	0x0001	Port 1 (flipped for network order)
30-31	sequenceId	0x0001	Incrementing sequence number
32	controlField	0x00	CTRL_SYNC

Byte Range	Field	Value (Example)	Description
33	logMessageInterval	0x00	Log2 of sync interval (0 = 1 sec)
34-43	originTimestamp	[10 bytes]	When Sync was sent (see below)

OriginTimestamp (10 bytes)

This is packed as:

c

Copy

Download

```c
struct {
  int16_t  seconds_msb;  // High 16 bits of seconds (network byte order)
  uint32_t seconds_lsb;  // Low 32 bits of seconds (network byte order)
  uint32_t nanoseconds;  // Nanoseconds (network byte order)
};
```

Example:
If the Sync was sent at 1234567890.500000000 seconds:

text

Copy

Download

```text
34-35: 0x0000      // seconds_msb (high 16 bits of 1234567890 = 0x499602D2 → msb=0x4996 → 0x0000 in 16-bit)

36-39: 0x499602D2  // seconds_lsb (low 32 bits)

40-43: 0x1DCD6500  // nanoseconds (500,000,000 = 0x1DCD6500)
```

Full Sync Message Example (Hex Dump)

text

Copy

Download

```
Offset: 00 01 02 03 04 05 06 07 08 09 0A 0B 0C 0D 0E 0F
0000:  10 02 00 2C 00 00 00 01 00 00 00 00 00 00 00 00  ...,............
0010:  00 11 22 33 44 55 66 77 00 01 00 01 00 00 00 00  .."3DUfw........
0020:  00 00 49 96 02 D2 1D CD 65 00                    ..I.....e.
```

Breakdown:

- 10 → transportSpecific=0x1 + messageType=0x0 (Sync)
- 02 → PTPv2
- 00 2C → 44 bytes total length
- 00 → Domain 0
- 00 01 → TWO_STEP flag set
- 00...00 → Correction field (zero for Sync)
- 00:11:22:33:44:55:66:77 → Clock ID
- 00 01 → Port 1
- 00 01 → Sequence ID 1
- 00 → CTRL_SYNC
- 00 → Log interval = 0 (1 second)
- 00 00 49 96 02 D2 1D CD 65 00 → Timestamp (1234567890.500000000)

Key Notes from the Code:

1. **Header Initialization** (msg_pack_header() in msg.c):

c

Copy

Download

(uint8_t)(buf + 0) = 0x80; // Sets transportSpecific=0x1 after shift

(uint8_t)(buf + 1) = ptpClock->port_ds.versionNumber; // 0x02

(uint8_t)(buf + 4) = ptpClock->default_ds.domain_number; // Domain

2. **Sync-Specific Fields** (msg_pack_sync() in msg.c):

c

Copy

Download

(char)(buf + 0) = *(char*)(buf + 0) | SYNC; // Sets messageType=0x0

(int16_t)(buf + 30) = flip16(ptpClock->sent_sync_sequence_id); // Sequence ID

(uint8_t)(buf + 32) = CTRL_SYNC; // 0x00

3. **Timestamp Handling**:
    - The originTimestamp is converted from internal time format using ptp_time_from_internal() (from arith.c).

When Would TWO_STEP **Flag Be Set?**

- If the clock is a **two-step clock**, the flag is set in msg_pack_header():

c

Copy

Download

if (ptpClock->default_ds.two_step_flag) {

  (uint8_t)(buf + 6) = FLAG0_TWO_STEP; // Sets bit 0 of flagField[0]

}

- This indicates that the precise timestamp will be sent later in a FollowUp message.

This matches the IEEE 1588-2008 standard and your code's implementation. Let me know if you'd like examples for other message types!