**Your setup already includes:**

* main() function with platform init, lwIP init.
* ptp\_clock, ptpd\_opts, and foreign master records.
* ptpd\_opts\_init() that sets basic options.
* Network configuration with netif\_add, netif\_set\_default, etc.
* PTPD stack boot-up via ptp\_startup().
* Event loop with ptpd\_periodic\_handler().

**🔧 What needs to be curated next**

**1. ✅ platform\_init():**

You already use init\_platform(), which is good. Ensure this also includes:

* Xil\_ICacheEnable(), Xil\_DCacheEnable() if not already handled.
* UART for xil\_printf.

No changes needed here **if you're using the SDK-provided platform functions.**

**2. ✅ lwIP Network:**

Your call:

c

CopyEdit

netif\_add(&server\_netif, &ipaddr, &netmask, &gw, NULL, xemac\_add, tcpip\_input);

is **correct for raw lwIP with Xilinx EMAC**.

✅ You can continue using static IPs unless DHCP is needed.

**3. 🔁 Replace or extend ptpd\_periodic\_handler():**

This function must internally:

* Call protocol() to handle incoming messages.
* Maintain PTP state machine transitions.
* Respond to SYNC, FOLLOW\_UP, DELAY\_REQ/RESP, etc.

You may eventually break this down and directly call lower-level functions (protocol(), issue\_sync(), etc.) if needed, but using ptpd\_periodic\_handler() is a good abstraction at this stage.

**4. ✅ UDP Receive Handlers:**

You're calling:

c

CopyEdit

ptpd\_net\_init(&ptp\_clock);

Which internally:

* Creates two UDP PCBs (event and general).
* Registers callbacks (e.g., ptpd\_recv\_event\_callback).

You **must ensure** ptpd\_recv\_event\_callback() and ptpd\_recv\_general\_callback() are implemented and properly dispatch messages to protocol().

If these are present in ptpd.c or protocol.c, you're good.

**5. ✅ Message queue:**

You're creating:

c

CopyEdit

ptp\_alert\_queue = sys\_mbox\_new();

which is used by the PTPD stack for event handling. Keep it.

**6. 🧩 Header Integration:**

In your ptpd.h, make sure you're including:

c

CopyEdit

#include "lwip/udp.h"

#include "lwip/sys.h"

#include "lwip/ip\_addr.h"

#include "ptpd\_dep.h"

#include "datatypes.h"

#include "constants.h"

This ensures platform definitions, UDP types, etc., are accessible.

**✅ Final PTP Main Flow**

If you want a **clean summary for the main.c integration**, it's already almost perfect. Just ensure:

**➕ Add this near top if not present:**

c

CopyEdit

#define LWIP\_RAW 1

#define NO\_SYS 1

#define LWIP\_TIMEVAL\_PRIVATE 0

If using SDK GUI, add these as compiler flags instead:

bash

CopyEdit

-DLWIP\_RAW=1 -DNO\_SYS=1 -DLWIP\_TIMEVAL\_PRIVATE=0

**🧠 Summary Checklist**

| **Task** | **Done?** | **Notes** |
| --- | --- | --- |
| Hardware setup in Vivado + SDK | ✅ | Block design already done |
| Platform code in main.c | ✅ | Already integrated |
| lwIP raw stack init | ✅ | via lwip\_init, netif\_add |
| UDP + PTPD stack integration | ✅ | ptpd\_net\_init() handles it |
| Message callbacks | ⚠️ | Ensure ptpd\_recv\_event\_callback is properly wired |
| Event loop with periodic handler | ✅ | Already calling ptpd\_periodic\_handler() |
| Timestamp handling (future) | 🔜 | Needed in protocol.c and hardware path |
| Time setting functions | ✅ | Check sys\_set\_clocktime() and ptpd\_adj\_frequency() in servo.c |

Would you like me to now give you:

1. A full cleaned main.c with comments?
2. A minimal list of source files you must copy into your src/?
3. The SDK build settings for the Xilinx project?

Let me know how you'd like to proceed.

2/2

Ask ChatGPT

**You said:**

main.c

**ChatGPT said:**

Here is a **clean, ready-to-use main.c** for your **bare-metal PTPv2 implementation on KCU105**, using the Xilinx SDK with the lwIP raw API and ptpd stack:

**✅ main.c — Bare-Metal PTP Implementation for KCU105 with lwIP**

c

CopyEdit

#include <stdio.h>

#include <string.h>

#include <lwip/init.h>

#include <lwip/netif.h>

#include <lwip/timers.h>

#include <netif/xemacif.h>

#include "xil\_printf.h"

#include "platform.h"

#include "xparameters.h"

#include <lwip/apps/ptpd.h>

// Global PTP variables

ptp\_clock\_t ptp\_clock;

ptpd\_opts ptp\_opts;

foreign\_master\_record\_t foreign\_records[PTPD\_DEFAULT\_MAX\_FOREIGN\_RECORDS];

sys\_mbox\_t ptp\_alert\_queue;

// Network interface

struct netif server\_netif;

// PTP runtime option initialization

void ptpd\_opts\_init()

{

xil\_printf("Initializing PTP options...\n");

memset(&ptp\_opts, 0, sizeof(ptpd\_opts));

ptp\_opts.slave\_only = 0; // Set 1 for slave-only mode

ptp\_opts.sync\_interval = 1; // 2^1 = 2s Sync interval

ptp\_opts.announce\_interval = 1; // 2^1 = 2s Announce interval

ptp\_opts.clock\_quality.clock\_class = 248; // Default

ptp\_opts.clock\_quality.clock\_accuracy = 0xFE; // Unknown accuracy

ptp\_opts.clock\_quality.offset\_scaled\_log\_variance = 0xFFFF;

ptp\_opts.priority1 = 128;

ptp\_opts.priority2 = 128;

if (ptp\_startup(&ptp\_clock, &ptp\_opts, foreign\_records) != 0)

{

xil\_printf("PTP startup failed!\n");

}

}

int main()

{

xil\_printf("\nStarting PTP on KCU105...\n");

// Initialize hardware platform

init\_platform();

// Initialize lwIP stack

lwip\_init();

// Configure static IP address

ip\_addr\_t ipaddr, netmask, gw;

IP4\_ADDR(&ipaddr, 192, 168, 1, 10);

IP4\_ADDR(&netmask, 255, 255, 255, 0);

IP4\_ADDR(&gw, 192, 168, 1, 1);

// Add network interface using Xilinx EMAC driver

if (!netif\_add(&server\_netif, &ipaddr, &netmask, &gw, NULL, xemac\_add, tcpip\_input)) {

xil\_printf("Failed to add network interface!\n");

return -1;

}

netif\_set\_default(&server\_netif);

netif\_set\_up(&server\_netif);

xil\_printf("Network initialized: %s\n", ip4addr\_ntoa(&ipaddr));

// Create message queue used by PTPD

ptp\_alert\_queue = sys\_mbox\_new();

// Initialize PTP options and startup

ptpd\_opts\_init();

// Initialize UDP ports and callback handlers (event/general)

ptpd\_net\_init(&ptp\_clock);

xil\_printf("PTP initialized. Entering main loop...\n");

// Main polling loop

while (1)

{

sys\_check\_timeouts(); // Check for lwIP timer events

ptpd\_periodic\_handler(); // PTP stack logic

sys\_msleep(10); // Sleep/yield to avoid tight loop

}

cleanup\_platform();

return 0;

}

**🔧 Build Flags Required (set in SDK project settings):**

* -DLWIP\_RAW=1
* -DNO\_SYS=1
* -DLWIP\_TIMEVAL\_PRIVATE=0
* Optimization flags: -O2 -g -Wall

**📦 Files You Need in src/ Directory:**

* main.c (the one above)
* ptpd.c, protocol.c, servo.c, bmc.c, arith.c, startup.c
* Header files: ptpd.h, ptpd\_dep.h, constants.h, datatypes.h, ptpd\_exports.h
* Any helper: sys.c, sys\_arch.c if you're using lwIP raw mode

**✅ Next Steps:**

Would you like a minimal working .h set or do you want help compiling in SDK? Or testing the first SYNC packet on wire?

Let me know — I can walk you through it step-by-step.

**You said:**

sure

**ChatGPT said:**

Great! Now that your main.c is ready and you're working on a **bare-metal PTP implementation for KCU105 using lwIP**, here's how we’ll proceed to **bring everything together on the board**.

**✅ Step-by-Step Guide to Integrate Software + Hardware in Xilinx SDK**

**🧰 Prerequisites**

* Vivado Block Design with:
  + MicroBlaze
  + AXI Ethernet + TEMAC
  + DDR4
  + AXI Timer (optional for timestamping)
* Exported .xsa from Vivado
* Vitis SDK or Xilinx SDK workspace
* Template used: **lwIP RAW Mode + NO\_SYS + TEMAC**

**🧩 1. Project Structure in SDK**

Inside your SDK src/ folder, you should have:

swift

CopyEdit

src/

├── main.c ✅ (You already have it)

├── ptpd.c ✅

├── protocol.c ✅

├── servo.c ✅

├── bmc.c ✅

├── arith.c ✅

├── startup.c ✅

├── sys.c ✅ if using raw lwIP

├── sys\_arch.c ✅ if needed by lwIP

├── ptpd.h ✅

├── ptpd\_dep.h ✅

├── ptpd\_exports.h ✅

├── constants.h ✅

├── datatypes.h ✅

└── (any lwIP/lwIP-PTPd headers required)

**⚙️ 2. Set Compilation Flags**

In SDK:

* Go to **Project Settings → C/C++ Build → Settings → Compiler → Symbols**  
  Add these **preprocessor definitions**:

c

CopyEdit

-DLWIP\_RAW=1

-DLWIP\_TIMEVAL\_PRIVATE=0

-DNO\_SYS=1

Also, in optimization flags:

bash

CopyEdit

-O2 -g -Wall

**🌐 3. lwIP Configuration (in platform\_config.h or lwipopts.h)**

Ensure:

c

CopyEdit

#define LWIP\_RAW 1

#define NO\_SYS 1

#define LWIP\_DNS 0

#define LWIP\_SOCKET 0

#define LWIP\_NETCONN 0

#define SYS\_LIGHTWEIGHT\_PROT 0

#define LWIP\_TIMERS 1

#define LWIP\_STATS 0

If you don’t already have a lwipopts.h, I can generate one tailored to PTP.

**📡 4. Hardware Link with Ethernet**

From Vivado → you must:

* Export hardware including bitstream (.xsa)
* Confirm the **Ethernet interface** (e.g., xemacps or xemac) is properly connected to PHY (e.g., SGMII or RGMII)
* Validate **MDIO is accessible** and auto-negotiation succeeds

**🧪 5. Run + Validate**

Once everything is built:

* Program FPGA using Program FPGA in SDK
* Launch UART terminal (e.g., TeraTerm @ 115200 baud)
* You should see:

nginx

CopyEdit

Starting PTP on KCU105...

Initializing PTP options...

Network initialized: 192.168.1.10

PTP initialized. Entering main loop...

Then you can:

* Use **Wireshark** on the same subnet to capture PTP packets (UDP port 319/320)
* Validate SYNC, FOLLOW\_UP, DELAY\_REQ, and DELAY\_RESP

**⏱️ Next Steps:**

Would you like help now with:

1. ✅ **Generating lwipopts.h**
2. ✅ **Creating the .xsa properly from Vivado**
3. ✅ **Generating a full bitstream + SDK workspace setup guide**
4. ✅ **Adding timestamp capture logic for TEMAC (IEEE 1588 support)**

Let me know which direction you’d like to go next.

**You said:**

minimal working .h

**ChatGPT said:**

Here's a **minimal working ptpd.h** header file tailored for your **bare-metal PTP implementation using lwIP on the KCU105**:

**✅ ptpd.h — Minimal Required Version**

c

CopyEdit

#ifndef PTPD\_H

#define PTPD\_H

#include <stdint.h>

#include <stdbool.h>

#include <string.h>

#include "lwip/opt.h"

#include "lwip/udp.h"

#include "lwip/timeouts.h"

#include "lwip/sys.h"

// ---- Constants ---- //

#define PTPD\_DEFAULT\_MAX\_FOREIGN\_RECORDS 5

#define PTPD\_DEFAULT\_PARENTS\_STATS 1

#define PTPD\_DEFAULT\_CALIBRATED\_OFFSET\_NS 100

#define PTPD\_DEFAULT\_UNCALIBRATED\_OFFSET\_NS 1000

#define PTPD\_MAX\_ADJ\_OFFSET\_NS 100000000

#define ADJ\_FREQ\_MAX 512000

// ---- Enums ---- //

typedef enum {

PTP\_INITIALIZING = 1,

PTP\_FAULTY,

PTP\_DISABLED,

PTP\_LISTENING,

PTP\_PRE\_MASTER,

PTP\_MASTER,

PTP\_PASSIVE,

PTP\_UNCALIBRATED,

PTP\_SLAVE

} ptp\_state\_t;

typedef enum {

E2E,

P2P

} delay\_mechanism\_t;

typedef enum {

SYNC\_INTERVAL\_TIMEOUT,

ANNOUNCE\_RECEIPT\_TIMEOUT,

SYNCHRONIZATION\_FAULT,

MASTER\_CLOCK\_SELECTED

} ptp\_event\_t;

// ---- Structs ---- //

typedef struct {

int32\_t seconds;

int32\_t nanoseconds;

} time\_interval\_t;

typedef struct {

int n;

int s;

int s\_prev;

int32\_t y\_sum;

int32\_t y\_prev;

} Filter;

typedef struct {

uint8\_t clock\_class;

uint8\_t clock\_accuracy;

uint16\_t offset\_scaled\_log\_variance;

} clock\_quality\_t;

typedef struct {

bool parent\_stats;

int32\_t observed\_parent\_clock\_phase\_change\_rate;

int32\_t observed\_parent\_offset\_scaled\_log\_variance;

} parent\_dataset\_t;

typedef struct {

ptp\_state\_t port\_state;

delay\_mechanism\_t delay\_mechanism;

int log\_sync\_interval;

int priority1;

int priority2;

clock\_quality\_t clock\_quality;

parent\_dataset\_t parent\_ds;

time\_interval\_t peer\_mean\_path\_delay;

} port\_ds\_t;

typedef struct {

time\_interval\_t offset\_from\_master;

time\_interval\_t mean\_path\_delay;

} current\_ds\_t;

typedef struct {

bool no\_adjust;

bool no\_reset\_clock;

int ap;

int ai;

int s\_delay;

int s\_offset;

} servo\_config\_t;

typedef struct {

int slave\_only;

int sync\_interval;

int announce\_interval;

clock\_quality\_t clock\_quality;

int priority1;

int priority2;

} ptpd\_opts;

typedef struct {

struct udp\_pcb\* event\_pcb;

struct udp\_pcb\* general\_pcb;

} ptp\_net\_path\_t;

typedef struct {

time\_interval\_t time\_ms, time\_sm;

time\_interval\_t pdelay\_t1, pdelay\_t2, pdelay\_t3, pdelay\_t4;

current\_ds\_t current\_ds;

port\_ds\_t port\_ds;

parent\_dataset\_t parent\_ds;

ptp\_net\_path\_t net\_path;

servo\_config\_t servo;

int observed\_drift;

bool waiting\_for\_followup;

bool waiting\_for\_pdelay\_resp\_followup;

Filter owd\_filt;

Filter ofm\_filt;

#if PTPD\_DEFAULT\_PARENTS\_STATS

Filter slv\_filt;

int offset\_history[2];

#endif

uint8\_t events[16]; // Bitfield

} ptp\_clock\_t;

typedef struct {

uint8\_t dummy; // Add fields as needed

} foreign\_master\_record\_t;

// ---- Function Prototypes ---- //

int ptp\_startup(ptp\_clock\_t\* clock, ptpd\_opts\* opts, foreign\_master\_record\_t\* foreign);

void ptpd\_periodic\_handler(void);

void ptpd\_net\_init(ptp\_clock\_t\* clock);

void ptpd\_adj\_frequency(int32\_t adj);

void ptpd\_empty\_event\_queue(ptp\_net\_path\_t\* net\_path);

// Time math

void ptp\_sub\_time(time\_interval\_t\* result, const time\_interval\_t\* a, const time\_interval\_t\* b);

void ptp\_time\_add(time\_interval\_t\* result, const time\_interval\_t\* a, const time\_interval\_t\* b);

void ptp\_time\_halve(time\_interval\_t\* t);

int ptp\_floor\_log2(int n);

// Clock time interface (bare-metal wrappers)

void sys\_get\_clocktime(time\_interval\_t\* t);

void sys\_set\_clocktime(const time\_interval\_t\* t);

// Debug logging

#define DBG(...) xil\_printf(\_\_VA\_ARGS\_\_)

#define DBGV(...) // Uncomment for verbose debug

// Event flags

#define setFlag(e, flag) ((e)[flag] = 1)

#define clearFlag(e, flag) ((e)[flag] = 0)

#define checkFlag(e, flag) ((e)[flag])

#endif // PTPD\_H

**🧩 What’s Included**

* **PTP Clock definitions**
* **Options, filter, and dataset structs**
* **Function prototypes used across ptpd.c, servo.c, and protocol.c**
* Support for lwIP RAW API + **bare-metal** operation

Would you like me to now generate the accompanying ptpd\_dep.h (dependency definitions) or lwipopts.h as the next step?

**You said:**

main.h

**ChatGPT said:**

Here’s a **minimal main.h** for your **bare-metal PTP project on the KCU105** using lwIP and ptpd.c. This header will cleanly declare all required functions and external symbols used by main.c.

**✅ main.h**

c

CopyEdit

#ifndef MAIN\_H

#define MAIN\_H

#include <lwip/netif.h>

#include <lwip/sys.h>

#include "ptpd.h"

// ---- Extern declarations ---- //

extern ptp\_clock\_t ptp\_clock;

extern ptpd\_opts ptp\_opts;

extern foreign\_master\_record\_t foreign\_records[PTPD\_DEFAULT\_MAX\_FOREIGN\_RECORDS];

extern sys\_mbox\_t ptp\_alert\_queue;

extern struct netif server\_netif;

// ---- Function declarations ---- //

void ptpd\_opts\_init(void);

#endif // MAIN\_H

**🧩 Summary**

* Declares all **globals** defined in main.c
* Provides a forward declaration for ptpd\_opts\_init()
* Clean and modular for inclusion in other files like servo.c, protocol.c, etc.

Would you like a platform\_config.h (to manage IP, masks, etc.) as a next step?

**You said:**

: ✅ Generating lwipopts.h ✅ Creating the .xsa properly from Vivado ✅ Generating a full bitstream + SDK workspace setup guide ✅ Adding timestamp capture logic for TEMAC (IEEE 1588 support)

**ChatGPT said:**

Perfect — here’s how to **systematically complete each of those four tasks** for your **PTP-enabled KCU105 bare-metal project with lwIP and TEMAC**:

**✅ 1. Generating lwipopts.h (for raw API and PTP)**

Place this in your src/ or bsp/ directory.

c

CopyEdit

#ifndef \_\_LWIPOPTS\_H\_\_

#define \_\_LWIPOPTS\_H\_\_

// --------- SYSTEM OPTIONS ----------

#define NO\_SYS 1

#define LWIP\_TIMERS 1

#define LWIP\_TIMEVAL\_PRIVATE 0

// --------- MEMORY OPTIONS ----------

#define MEM\_ALIGNMENT 4

#define MEM\_SIZE (16 \* 1024)

#define MEMP\_NUM\_UDP\_PCB 8

#define MEMP\_NUM\_SYS\_TIMEOUT 10

// --------- PBUF OPTIONS ------------

#define PBUF\_POOL\_SIZE 32

#define PBUF\_POOL\_BUFSIZE 512

// --------- UDP SUPPORT -------------

#define LWIP\_UDP 1

#define LWIP\_RAW 1

// --------- NETWORK INTERFACE -------

#define LWIP\_NETIF\_STATUS\_CALLBACK 1

#define LWIP\_NETIF\_LINK\_CALLBACK 1

// --------- CHECKSUM OPTIONS --------

#define CHECKSUM\_BY\_HARDWARE 1

#define CHECKSUM\_GEN\_IP 0

#define CHECKSUM\_GEN\_UDP 0

#define CHECKSUM\_CHECK\_UDP 0

// --------- Debug (Optional) --------

// #define LWIP\_DEBUG 1

// #define UDP\_DEBUG LWIP\_DBG\_ON

#endif /\* \_\_LWIPOPTS\_H\_\_ \*/

**✅ 2. Generating the .xsa File from Vivado (for SDK/Vitis)**

**In Vivado:**

1. **Design Hardware:**
   * Include ZYNQ/PS, AXI Ethernet Subsystem (TEMAC), MIG (for DDR), and MicroBlaze if using soft-core.
   * Connect **TEMAC** to **AXI DMA**, with a clock wizard generating 125 MHz.
2. **Validate & Synthesize:**
   * Click **"Run Synthesis"**, then **"Run Implementation"**.
3. **Generate Bitstream:**
   * Tools → **Generate Bitstream**
   * Wait until bitstream is complete.
4. **Export Hardware with Bitstream:**
   * File → Export → **Export Hardware**
   * ✅ Check **Include Bitstream**
   * Save the .xsa to a known path.

**✅ 3. Create SDK Workspace or Vitis Project**

**In Vitis:**

1. **Launch Vitis IDE.**
2. **Create New Application Project**:
   * Select your .xsa
   * Name it ptpd\_lwip\_kcu105
3. **Choose Platform:**
   * Select standalone, MicroBlaze or Zynq based on your design.
4. **Select Application Template**:
   * Choose **Empty Application**
5. **Import Your Source Code**:
   * main.c, ptpd.c, protocol.c, servo.c, etc.
6. **Add lwipopts.h and enable lwIP support**:
   * BSP Settings → lwip213 → Enable raw\_api + udp + ipv4

**✅ 4. Add Timestamp Capture Logic for TEMAC (IEEE 1588)**

**Option A: Hardware timestamping (recommended for accuracy)**

1. **Enable 1588 in TEMAC Core**:
   * In IP config for TEMAC, enable **"1588 Timestamp Support"**
   * Connect timestamp ports:
     + tx\_timestamp\_out
     + rx\_timestamp\_in
2. **Route timestamps to logic**:
   * Use AXI stream registers or custom logic to latch rx/tx\_timestamp data.
   * Use 64-bit capture (e.g., seconds + nanoseconds).
3. **Interface with Software**:
   * Modify ptpd\_recv\_event\_callback() and issue\_sync() to read/write timestamp registers via MMIO or AXI BRAM.

**Option B: Software timestamping (less accurate)**

* Timestamp at ISR:

c

CopyEdit

sys\_now(); // Not high-precision, ~1ms

* Use xemacif\_input() hooks to inject timestamps during packet capture.

**🧱 Bonus: Create platform.c and platform.h**

To abstract board-specific calls:

c

CopyEdit

void ptpd\_adj\_frequency(int32\_t adj);

void sys\_set\_clocktime(time\_interval\_t\* t);

void sys\_get\_clocktime(time\_interval\_t\* t);

Would you like me to auto-generate platform.c to support this project next?

**You said:**

#if \_\_MICROBLAZE\_\_ #include "arch/cc.h" #include "platform.h" #include "platform\_config.h" #include "xil\_cache.h" #include "xparameters.h" #include "xintc.h" #include "xil\_exception.h" #include "lwip/tcp.h" #ifdef STDOUT\_IS\_16550 #include "xuartns550\_l.h" #endif #include "lwip/tcp.h" #if LWIP\_DHCP==1 volatile int dhcp\_timoutcntr = 24; void dhcp\_fine\_tmr(); void dhcp\_coarse\_tmr(); #endif volatile int TcpFastTmrFlag = 0; volatile int TcpSlowTmrFlag = 0; volatile u64\_t tickcntr = 0; void timer\_callback() { /\* we need to call tcp\_fasttmr & tcp\_slowtmr at intervals specified \* by lwIP. \* It is not important that the timing is absoluetly accurate. \*/ static int odd = 1; #if LWIP\_DHCP==1 static int dhcp\_timer = 0; #endif tickcntr++; if(tickcntr % 25 == 0){ TcpFastTmrFlag = 1; odd = !odd; if (odd) { TcpSlowTmrFlag = 1; #if LWIP\_DHCP==1 dhcp\_timer++; dhcp\_timoutcntr--; dhcp\_fine\_tmr(); if (dhcp\_timer >= 120) { dhcp\_coarse\_tmr(); dhcp\_timer = 0; } #endif } } } static XIntc intc; void platform\_setup\_interrupts() { XIntc \*intcp; intcp = &intc; XIntc\_Initialize(intcp, XPAR\_INTC\_0\_DEVICE\_ID); XIntc\_Start(intcp, XIN\_REAL\_MODE); /\* Start the interrupt controller \*/ XIntc\_MasterEnable(XPAR\_INTC\_0\_BASEADDR); #ifdef \_\_MICROBLAZE\_\_ microblaze\_register\_handler((XInterruptHandler)XIntc\_InterruptHandler, intcp); #endif platform\_setup\_timer(); #ifdef XPAR\_ETHERNET\_MAC\_IP2INTC\_IRPT\_MASK /\* Enable timer and EMAC interrupts in the interrupt controller \*/ XIntc\_EnableIntr(XPAR\_INTC\_0\_BASEADDR, #ifdef \_\_MICROBLAZE\_\_ PLATFORM\_TIMER\_INTERRUPT\_MASK | #endif XPAR\_ETHERNET\_MAC\_IP2INTC\_IRPT\_MASK); #endif #ifdef XPAR\_INTC\_0\_LLTEMAC\_0\_VEC\_ID #ifdef \_\_MICROBLAZE\_\_ XIntc\_Enable(intcp, PLATFORM\_TIMER\_INTERRUPT\_INTR); #endif XIntc\_Enable(intcp, XPAR\_INTC\_0\_LLTEMAC\_0\_VEC\_ID); #endif #ifdef XPAR\_INTC\_0\_AXIETHERNET\_0\_VEC\_ID XIntc\_Enable(intcp, PLATFORM\_TIMER\_INTERRUPT\_INTR); XIntc\_Enable(intcp, XPAR\_INTC\_0\_AXIETHERNET\_0\_VEC\_ID); #endif #ifdef XPAR\_INTC\_0\_EMACLITE\_0\_VEC\_ID #ifdef \_\_MICROBLAZE\_\_ XIntc\_Enable(intcp, PLATFORM\_TIMER\_INTERRUPT\_INTR); #endif XIntc\_Enable(intcp, XPAR\_INTC\_0\_EMACLITE\_0\_VEC\_ID); #endif } void enable\_caches() { #ifdef \_\_MICROBLAZE\_\_ #ifdef XPAR\_MICROBLAZE\_USE\_ICACHE Xil\_ICacheEnable(); #endif #ifdef XPAR\_MICROBLAZE\_USE\_DCACHE Xil\_DCacheEnable(); #endif #endif } void disable\_caches() { Xil\_DCacheDisable(); Xil\_ICacheDisable(); } void init\_platform() { enable\_caches(); #ifdef STDOUT\_IS\_16550 XUartNs550\_SetBaud(STDOUT\_BASEADDR, XPAR\_XUARTNS550\_CLOCK\_HZ, 9600); XUartNs550\_SetLineControlReg(STDOUT\_BASEADDR, XUN\_LCR\_8\_DATA\_BITS); #endif platform\_setup\_interrupts(); } void cleanup\_platform() { disable\_caches(); } u64\_t get\_time\_ms() { return tickcntr \* 10; } #endif

**ChatGPT said:**

Thanks for sharing the platform.c source. It’s a **custom board support implementation for MicroBlaze** with a timer and interrupt setup — perfect for a bare-metal PTP build.

To **tailor this platform.c for your PTP project**, I’ll now:

**✅ 1. Add PTP-Specific Platform Functions**

These are required by the servo.c and PTP engine to:

* Read the system clock (sys\_get\_clocktime)
* Set the clock (sys\_set\_clocktime)
* Adjust clock frequency (ptpd\_adj\_frequency)

We'll **extend** your existing platform.c with those definitions.

**✅ 2. Minimal PTP-Compatible Additions**

Below your existing code, add:

c

CopyEdit

#include "ptpd.h"

#include "xil\_printf.h"

/\* Software emulated clock in nanoseconds (for demo) \*/

static time\_interval\_t sys\_time;

/\* Simulate reading the system clock \*/

void sys\_get\_clocktime(time\_interval\_t\* t)

{

\*t = sys\_time;

}

/\* Simulate setting the system clock \*/

void sys\_set\_clocktime(const time\_interval\_t\* t)

{

sys\_time = \*t;

xil\_printf("PTP Clock Set: %d sec, %d ns\n", t->seconds, t->nanoseconds);

}

/\* Simulate adjusting clock frequency (emulated) \*/

void ptpd\_adj\_frequency(int32\_t adj)

{

// For real implementation, use timer reload or clock generator APIs

xil\_printf("PTP Adjust Frequency: %d (ignored in software sim)\n", adj);

}

🔧 In real hardware timestamping systems (like TEMAC with 1588 support), ptpd\_adj\_frequency() would interface with:

* a hardware timer (via AXI timer PWM)
* or a PLL clock generator to fine-tune oscillator frequency

If you're planning to support this, we can connect it to an AXI Timer or PLL IP block.

**✅ 3. platform.h – Minimal Header**

c

CopyEdit

#ifndef PLATFORM\_H

#define PLATFORM\_H

#include <stdint.h>

#include "ptpd\_datatypes.h"

// Clock control for PTP

void sys\_get\_clocktime(time\_interval\_t\* t);

void sys\_set\_clocktime(const time\_interval\_t\* t);

void ptpd\_adj\_frequency(int32\_t adj);

// Existing

void init\_platform();

void cleanup\_platform();

#endif

**✅ Final Notes**

* Your existing timer (tickcntr) runs every 10ms; it’s useful for simulation but not suitable for sub-microsecond PTP accuracy.
* For **accurate PTP** (e.g. nanosecond-level), consider using:
  + **TEMAC hardware timestamps** (preferred)
  + Or **AXI Timer + Capture mode** to drive sys\_get\_clocktime()

Would you like a version of sys\_get\_clocktime() that uses a high-resolution AXI Timer instead of emulated tickcntr?

**You said:**

show the project structure everytime u give an answer #include "ptpd/ptpd.h" #include <string.h> #include <stdio.h> #include <stdbool.h> // These are linked externally extern sys\_mbox\_t ptp\_alert\_queue; extern ptpd\_opts opts; extern ptp\_clock\_t ptp\_clock; void ptpd\_alert(void) { if (chMBPostI(ptp\_alert\_queue, (msg\_t)((void\*)NULL)) == MSG\_TIMEOUT) DBG("ptp: failed to post alert\n"); } void ptpd\_queue\_init(ptp\_buf\_queue\_t\* queue) { queue->head = queue->tail = 0; sys\_mutex\_new(&queue->mutex); } bool ptpd\_queue\_put(ptp\_buf\_queue\_t\* queue, struct pbuf\* p) { sys\_mutex\_lock(&queue->mutex); int next = (queue->head + 1) & PTPD\_PBUF\_QUEUE\_MASK; if (next == queue->tail) { sys\_mutex\_unlock(&queue->mutex); return false; } queue->head = next; queue->pbuf[queue->head] = p; sys\_mutex\_unlock(&queue->mutex); return true; } void\* ptpd\_queue\_get(ptp\_buf\_queue\_t\* queue) { void\* result = NULL; sys\_mutex\_lock(&queue->mutex); if (queue->tail != queue->head) { queue->tail = (queue->tail + 1) & PTPD\_PBUF\_QUEUE\_MASK; result = queue->pbuf[queue->tail]; } sys\_mutex\_unlock(&queue->mutex); return result; } bool ptpd\_is\_queue\_empty(ptp\_buf\_queue\_t\* queue) { bool result; sys\_mutex\_lock(&queue->mutex); result = (queue->tail == queue->head); sys\_mutex\_unlock(&queue->mutex); return result; } void ptpd\_empty\_queue(ptp\_buf\_queue\_t\* queue) { sys\_mutex\_lock(&queue->mutex); while (queue->tail != queue->head) { queue->tail = (queue->tail + 1) & PTPD\_PBUF\_QUEUE\_MASK; pbuf\_free(queue->pbuf[queue->tail]); } sys\_mutex\_unlock(&queue->mutex); } void ptpd\_empty\_event\_queue(net\_path\_t\* net\_path) { ptpd\_empty\_queue(&net\_path->event\_q); } void ptpd\_empty\_general\_queue(net\_path\_t\* net\_path) { ptpd\_empty\_queue(&net\_path->general\_q); } ssize\_t ptpd\_recv\_event(net\_path\_t\* net\_path, octet\_t\* buf, time\_interval\_t\* time) { return ptpd\_net\_recv(buf, time, &net\_path->event\_q); } ssize\_t ptpd\_recv\_general(net\_path\_t\* net\_path, octet\_t\* buf, time\_interval\_t\* time) { return ptpd\_net\_recv(buf, time, &net\_path->general\_q); } ssize\_t ptpd\_send\_event(net\_path\_t\* net\_path, const octet\_t\* buf, int16\_t length, time\_interval\_t\* time) { return ptpd\_net\_send(buf, length, time, &net\_path->addr\_multicast, net\_path->event\_pcb); } ssize\_t ptpd\_send\_general(net\_path\_t\* net\_path, const octet\_t\* buf, int16\_t length) { return ptpd\_net\_send(buf, length, NULL, &net\_path->addr\_multicast, net\_path->general\_pcb); } ssize\_t ptpd\_peer\_send\_general(net\_path\_t\* net\_path, const octet\_t\* buf, int16\_t length) { return ptpd\_net\_send(buf, length, NULL, &net\_path->addr\_peer\_multicast, net\_path->general\_pcb); } ssize\_t ptpd\_peer\_send\_event(net\_path\_t\* net\_path, const octet\_t\* buf, int16\_t length, time\_interval\_t\* time) { return ptpd\_net\_send(buf, length, time, &net\_path->addr\_peer\_multicast, net\_path->event\_pcb); } void ptpd\_update\_time(const time\_interval\_t\* time) { sys\_set\_clocktime(time); } bool ptpd\_adj\_frequency(int32\_t adj) { DBG("ptpd\_adj\_frequency %ld\n", adj); return false; } // Networking callbacks void ptpd\_recv\_event\_callback(void\* arg, struct udp\_pcb\* pcb, struct pbuf\* p, const ip\_addr\_t\* addr, u16\_t port) { (void) pcb; (void) addr; (void) port; net\_path\_t\* net\_path = (net\_path\_t\*)arg; if (!ptpd\_queue\_put(&net\_path->event\_q, p)) { pbuf\_free(p); ERROR("ptpd\_recv\_event\_callback: queue full\n"); return; } ptpd\_alert(); } void ptpd\_recv\_general\_callback(void\* arg, struct udp\_pcb\* pcb, struct pbuf\* p, const ip\_addr\_t\* addr, u16\_t port) { (void) pcb; (void) addr; (void) port; net\_path\_t\* net\_path = (net\_path\_t\*)arg; if (!ptpd\_queue\_put(&net\_path->general\_q, p)) { pbuf\_free(p); ERROR("ptpd\_recv\_general\_callback: queue full\n"); return; } ptpd\_alert(); } bool ptpd\_net\_init(net\_path\_t\* net\_path, ptp\_clock\_t\* clock) { struct in\_addr addr\_net; ip\_addr\_t addr\_interface; err\_t ret\_bind; char addr\_str[NET\_ADDRESS\_LENGTH]; DBG("ptpd\_net\_init\n"); ptpd\_queue\_init(&net\_path->event\_q); ptpd\_queue\_init(&net\_path->general\_q); addr\_interface.addr = ptpd\_find\_iface(clock->opts->iface\_name, clock->port\_uuid\_field, net\_path); if (!(addr\_interface.addr)) { DBG("ptpd: ptpd\_net\_init: Failed to find interface address\n"); goto fail01; } net\_path->event\_pcb = udp\_new(); if (NULL == net\_path->event\_pcb) { DBG("ptpd: ptpd\_net\_init: Failed to open Event UDP PCB\n"); goto fail02; } net\_path->general\_pcb = udp\_new(); if (NULL == net\_path->general\_pcb) { ERROR("ptpd: ptpd\_net\_init: Failed to open General UDP PCB\n"); goto fail03; } net\_path->addr\_unicast = 0; memcpy(addr\_str, DEFAULT\_PTP\_DOMAIN\_ADDRESS, NET\_ADDRESS\_LENGTH); if (!inet\_aton(addr\_str, &addr\_net)) { DBG("ptpd: ptpd\_net\_init: failed to encode multi-cast address: %s\n", addr\_str); goto fail04; } net\_path->addr\_multicast = addr\_net.s\_addr; DBG("join default group with %d\r\n", igmp\_joingroup(&addr\_interface, (ip\_addr\_t\*)&addr\_net)); memcpy(addr\_str, PEER\_PTP\_DOMAIN\_ADDRESS, NET\_ADDRESS\_LENGTH); if (!inet\_aton(addr\_str, &addr\_net)) { DBG("ptpd: ptpd\_net\_init: failed to encode peer multi-cast address: %s\n", addr\_str); goto fail04; } net\_path->addr\_peer\_multicast = addr\_net.s\_addr; DBG("join peer group with %d\r\n", igmp\_joingroup(&addr\_interface, (ip\_addr\_t\*)&addr\_net)); udp\_recv(net\_path->event\_pcb, ptpd\_recv\_event\_callback, net\_path); ret\_bind = udp\_bind(net\_path->event\_pcb, IP\_ADDR\_ANY, PTP\_EVENT\_PORT); if (ret\_bind != ERR\_OK) DBG("failed to bind event port | %d\r\n", ret\_bind); udp\_recv(net\_path->general\_pcb, ptpd\_recv\_general\_callback, net\_path); ret\_bind = udp\_bind(net\_path->general\_pcb, IP\_ADDR\_ANY, PTP\_GENERAL\_PORT); if (ret\_bind != ERR\_OK) DBG("failed to bind general port | %d\r\n", ret\_bind); return true; fail04: udp\_remove(net\_path->general\_pcb); fail03: udp\_remove(net\_path->event\_pcb); fail02: fail01: return false; } bool ptpd\_shutdown(net\_path\_t\* net\_path) { ip\_addr\_t addr\_multicast; DBG("ptpd\_shutdown\n"); addr\_multicast.addr = net\_path->addr\_multicast; igmp\_leavegroup(IP\_ADDR\_ANY, &addr\_multicast); if (net\_path->event\_pcb) { udp\_disconnect(net\_path->event\_pcb); udp\_remove(net\_path->event\_pcb); net\_path->event\_pcb = NULL; } if (net\_path->general\_pcb) { udp\_disconnect(net\_path->general\_pcb); udp\_remove(net\_path->general\_pcb); net\_path->general\_pcb = NULL; } net\_path->addr\_multicast = 0; net\_path->addr\_unicast = 0; return true; } int32\_t ptpd\_net\_select(net\_path\_t\* net\_path, const time\_interval\_t\* timeout) { (void)timeout; if (!ptpd\_is\_queue\_empty(&net\_path->event\_q) || !ptpd\_is\_queue\_empty(&net\_path->general\_q)) return 1; return 0; } ssize\_t ptpd\_net\_recv(octet\_t\* buf, time\_interval\_t\* time, ptp\_buf\_queue\_t\* msg\_queue) { int i, j; u16\_t length; struct pbuf\* p; struct pbuf\* pcopy; p = (struct pbuf\*)ptpd\_queue\_get(msg\_queue); if (p == NULL) { return 0; } if (p->tot\_len > PACKET\_SIZE) { ERROR("ptpd\_net\_recv: received truncated message\n"); pbuf\_free(p); return 0; } if (p->tot\_len == 0) { ERROR("ptpd\_net\_recv: received empty packet\n"); pbuf\_free(p); return 0; } if (time != NULL) { #if LWIP\_PTP time->seconds = p->time\_sec; time->nanoseconds = p->time\_nsec; #else sys\_get\_clocktime(time); #endif } length = p->tot\_len; pcopy = p; j = 0; for (i = 0; i < length; i++) { buf[i] = ((u8\_t\*)pcopy->payload)[j++]; if (j == pcopy->len) { pcopy = pcopy->next; j = 0; } } pbuf\_free(p); return length; } ssize\_t ptpd\_net\_send(const octet\_t\* buf, int16\_t length, time\_interval\_t\* time, const int32\_t\* addr, struct udp\_pcb\* pcb) { err\_t result; struct pbuf\* p; p = pbuf\_alloc(PBUF\_TRANSPORT, length, PBUF\_RAM); if (p == NULL) { ERROR("ptpd\_net\_send: Failed to allocate Tx Buffer\n"); goto fail01; } result = pbuf\_take(p, buf, length); if (result != ERR\_OK) { ERROR("ptpd\_net\_send: Failed to copy data to Pbuf (%d)\n", result); goto fail02; } result = udp\_sendto(pcb, p, (ip\_addr\_t\*)addr, pcb->local\_port); if (result != ERR\_OK) { ERROR("ptpd\_net\_send: Failed to send data (%d)\n", result); goto fail02; } if (time != NULL) { #if LWIP\_PTP time->seconds = p->time\_sec; time->nanoseconds = p->time\_nsec; #else sys\_get\_clocktime(time); #endif DBGV("ptpd\_net\_send: %d sec %d nsec\n", time->seconds, time->nanoseconds); } else { DBGV("ptpd\_net\_send\n"); } fail02: pbuf\_free(p); fail01: return length; } can this is ptpd.c file

**ChatGPT said:**

Perfect — this is the complete ptpd.c from your PTP daemon. Let’s walk through how to integrate this into your **bare-metal MicroBlaze + lwIP-based PTP project**, with **project structure** and a few **tailored integration notes**.

**✅ Project Structure for PTPD Bare-Metal Integration**

css

CopyEdit

<ptpd\_project\_root>/

├── include/

│ ├── platform.h ← Platform-specific declarations

│ ├── ptpd/

│ │ ├── ptpd.h ← Main PTP header

│ │ ├── ptpd\_dep.h ← Time + platform interface header

│ │ ├── ptpd\_options.h ← Runtime configuration

│ │ ├── ptpd\_constants.h ← Message types, states, etc.

│ │ ├── ptpd\_datatypes.h ← Core structures

│ │ └── <...>

├── src/

│ ├── main.c ← Entry point with lwIP init

│ ├── platform.c ← Timer, clock, and frequency adjustment

│ ├── ptpd.c ← Message send/recv, queueing, net init

│ ├── protocol.c ← PTP state machine logic (attach later)

│ ├── bmc.c ← Best master clock selection logic

│ ├── servo.c ← Offset + delay calculation and frequency adjust

│ └── <other .c files>

├── lwipopts.h ← lwIP config

├── system.mss / system.xsa ← SDK hardware description

├── Makefile ← Optional if building without XSDK

└── README.md

**✅ Your Current ptpd.c Summary**

This file handles:

| **Functionality** | **Description** |
| --- | --- |
| ptpd\_net\_init() | Initializes UDP PCBs, binds PTP ports, joins IGMP multicast groups |
| ptpd\_net\_send() / ptpd\_net\_recv() | Sends and receives PTP event/general packets |
| Queue API (ptpd\_queue\_\*) | Used to manage lwIP packets in thread-safe circular queues |
| ptpd\_recv\_event\_callback / general | lwIP callback handlers to queue up packets |
| ptpd\_alert() | Notifies the PTP FSM that data is available |
| ptpd\_adj\_frequency() | (Stubbed) Function to adjust clock frequency |
| Time stamping logic | Uses p->time\_sec and p->time\_nsec if LWIP\_PTP is enabled |

**🔧 What to Do Next**

To make this work in **your setup** (MicroBlaze + KCU105 + lwIP bare-metal), follow these steps:

**✅ 1. Enable Hardware Timestamping (Optional but Ideal)**

* If your system uses **Xilinx TEMAC with PTP support**, ensure LWIP\_PTP is defined in lwipopts.h.
* The p->time\_sec and p->time\_nsec are filled in by the Ethernet MAC if configured for 1588 timestamping.

In lwipopts.h:

c

CopyEdit

#define LWIP\_PTP 1

**✅ 2. Implement ptpd\_adj\_frequency() in platform.c**

Currently in ptpd.c:

c

CopyEdit

bool ptpd\_adj\_frequency(int32\_t adj) {

DBG("ptpd\_adj\_frequency %ld\n", adj);

return false;

}

Update platform.c with:

c

CopyEdit

bool ptpd\_adj\_frequency(int32\_t adj) {

xil\_printf("Adjusting clock frequency by: %ld ppb\n", adj);

// Add logic to control a hardware timer/PLL here

return true;

}

This will allow servo.c to call it when drift is detected.

**✅ 3. Clock Interface**

Make sure your platform.c provides:

c

CopyEdit

void sys\_get\_clocktime(time\_interval\_t\* t);

void sys\_set\_clocktime(const time\_interval\_t\* t);

These are used in both ptpd.c and servo.c for clock corrections.

**✅ 4. Start the PTP Stack**

In your main.c, ensure you:

c

CopyEdit

// Configure and initialize everything

ptpd\_opts\_init();

ptpd\_net\_init(&ptp\_clock.net\_path, &ptp\_clock);

ptp\_startup(&ptp\_clock, &ptp\_opts, foreign\_records);

// In main loop

while (1) {

sys\_check\_timeouts();

ptpd\_periodic\_handler(); // <-- runs PTP state machine

sys\_msleep(10);

}

**🔁 Coming Up Next**

Now that you’ve connected ptpd.c, next up:

* ✅ Confirm protocol.c and bmc.c are fully integrated
* ✅ Ensure your servo.c and platform.c collaborate via ptpd\_adj\_frequency
* ✅ Optionally create a Makefile or use SDK for build

Would you like me to go ahead and organize the remaining source files (protocol.c, bmc.c, ptpd.h) into this structure and tailor them accordingly?

Let me know your next step.

**You said:**

yup

**ChatGPT said:**

Updated saved memory

Great — let's organize your full PTP project for bare-metal MicroBlaze on KCU105, using lwIP and the ptpd stack.

**✅ Project Structure for PTP on Bare-Metal KCU105**

graphql

CopyEdit

PTP\_KCU105\_Project/

│

├── src/

│ ├── main.c # Main entrypoint (initializes lwIP + PTPD)

│ ├── main.h # Header for main declarations

│ ├── platform.c # Cache, timer, and interrupt setup

│ ├── platform.h

│ ├── protocol.c # State machine and message handling logic

│ ├── bmc.c # Best master clock selection logic

│ ├── ptpd.c # Event loop, queues, networking glue

│ ├── ptpd.h # PTP core definitions and types

│ └── lwipopts.h # lwIP configuration for PTP

│

├── include/

│ └── ptpd/ # Optional: move ptpd headers here

│ └── ptpd.h

│

├── bsp/ # Auto-generated from Vivado .xsa

│ └── system.mss

│

├── hw/ # Vivado design with TEMAC, DDR, MIG, etc.

│ └── PTP\_KCU105.xsa

│

├── bitstream/ # Full bitstream from Vivado

│ └── system\_wrapper.bit

│

└── README.md

**🧩 Integration Summary**

* ✅ main.c sets up lwIP + calls ptp\_startup() with your ptp\_clock\_t and ptpd\_opts
* ✅ ptpd.c handles lwIP callbacks, queues, and UDP PCBs (you already pasted this and it’s ready)
* ✅ platform.c/h handles MicroBlaze timer + cache setup (you’ve already pasted)
* 🟨 protocol.c and bmc.c still need to be ported completely — I will now give you the **full protocol.c** next, tailored to your project