UDP-PCB: Intro and Purpose

24 February 2026 14:45

In lwIP, udp_pcb is the **software control structure** that represents one UDP endpoint.

Think of it as:

A kernel-level object that describes one UDP socket-like instance.

In our UDP server logic, we have used it in:

```
struct udp_pcb *udp_pcb_global;
udp_pcb_global = udp_new();
udp_bind(udp_pcb_global, IP_ADDR_ANY, UDP_PORT);
udp_recv(udp_pcb_global, udp_echo_recv, NULL);
```

That single PCB represents:

- "UDP port 8888"
- listening on all local IPs
- waiting for incoming datagrams

Conceptual Mapping To our UDP server Project

```
main.c
  |
 → udp_new()
 → udp_bind()
 → udp_recv()

lwIP runtime
  |
 → waits for Ethernet interrupt
 → parses IP/UDP
 → finds pcb->local_port match
 → calls pcb->recv()

udpServer.c
  |
 → udp_echo_recv()
 → udp_sendto()
```

What happens internally when packet arrives?

Flow:

1. Ethernet interrupt
2. lwIP receives frame
3. IP layer parses
4. UDP layer extracts:
   - destination port
5. Searches PCB list
6. Finds matching PCB
7. Executes:

UDP PCB: code…

24 February 2026 15:28

Core UDP PCB definition

```
struct udp_pcb {
/** Common members of all PCB types */
  IP_PCB;

/* Protocol specific PCB members */

  struct udp_pcb *next;

  u8_t flags;
  /** ports are in host byte order */
  u16_t local_port, remote_port;

#if LWIP_MULTICAST_TX_OPTIONS
#if LWIP_IPV4
  /** outgoing network interface for multicast packets, by IPv4 address (if not 'any') */
  ip4_addr_t mcast_ip4;
#endif /* LWIP_IPV4 */
  /** outgoing network interface for multicast packets, by interface index (if nonzero) */
  u8_t mcast_ifindex;
  /** TTL for outgoing multicast packets */
  u8_t mcast_ttl;
#endif /* LWIP_MULTICAST_TX_OPTIONS */

#if LWIP_UDPLITE
  /** used for UDP_LITE only */
  u16_t chksum_len_rx, chksum_len_tx;
#endif /* LWIP_UDPLITE */

  /** receive callback function */
  udp_recv_fn recv;
  /** user-supplied argument for the recv callback */
  void *recv_arg;
};
```

UDP-PCB: IP_PCB

24 February 2026 15:35

IP_PCB — The Common IP Layer Part

IP_PCB;
This macro expands into shared IP-level fields like:

- local_ip
- remote_ip
- netif_idx
- ttl
- tos

In our UDP server:

Used indirectly by:

udp_bind(pcb, IP_ADDR_ANY, 8888)
This sets:

- pcb->local_ip = 0.0.0.0
- pcb->local_port = 8888

And when a packet is received:
lwIP matches:

Incoming packet's destination port
↓
Search udp_pcbs list
↓
Find matching pcb->local_port
So IP_PCB provides the IP identity layer

ttl (Time To Live)
Maximum number of router hops before packet is discarded.

Maximum number of router hops before packet is discarded.
Default usually:

ttl = 255

In our UDP server project:

- Packets are local LAN.
- No routers involved.
- TTL almost irrelevant.

Only matters if:

- Sending across routed networks
- Doing multicast
- Doing diagnostic network testing

In your current use: not modified, default used.

The local IP address this UDP endpoint is bound to.

it can be

udp_bind(pcb, IP_ADDR_ANY, 8888);

Any IP address assigned to our existing system with multiple Ethernet ports

udp_bind(pcb, &my_ip, 8888);

Specific address assigned to our existing system with single/ multiple Ethernet ports

Then only packets sent specifically to local_ip would match.

The default destination IP if the PCB is "connected".

udp_sendto(pcb, p, addr, port);

remote_ip would only matter if you used:

udp_connect(pcb, &ip, port);

udp_send(pcb, p);

Implicatly taken

Which network interface this PCB is tied to.
Important only if:

- Multiple Ethernet interfaces exist
- Multiple netifs are registered in lwIP

If we use only one Ethernet Interface (PS GEM)
netif_idx = 0

Dig more on netif_idx in netif section

tos (Type of Service)
Now called DSCP in modern IP.
What it represents:
Packet priority / QoS marking
Examples:

- Low latency
- High throughput
- Background traffic

In our project:

- Not modified.
- Default = 0
- No QoS differentiation.

Would matter only if:

- You were doing real-time streaming
- Working in managed enterprise networks

UDP-PCB: struct udp_pcb *next

24 February 2026 17:03

This links all UDP PCBs into a global linked list.
Internally lwIP maintains:

```
udp_pcbs -> pcb1 -> pcb2 -> pcb3
```

When a packet arrives:

1. lwIP scans this linked list
2. Compares ports
3. Finds matching PCB
4. Calls its callback

You do not touch this directly, but your PCB is inserted here automatically when you call:

```
udp_new();
udp_bind();
```

Why are UDP PCBs in a linked list?
For pbufs it makes sense (data chaining), but why PCBs?

Suppose your Zynq firmware later has:

- Port 8888 → Debug server
- Port 5000 → Sensor streaming
- Port 69 → TFTP
- Port 123 → NTP client

Now you need:

pcb1 → port 8888
pcb2 → port 5000
pcb3 → port 69
pcb4 → port 123

Each one has:

- Different local_port
- Different callback function
- Different behavior

Now when a packet arrives, lwIP must decide:

Which application does this packet belong to?

- lwIP is designed for embedded systems
- Memory must be dynamic
- Number of PCBs not fixed at compile time
- PCBs created/destroyed at runtime

Linked list advantages:

No fixed max count
No wasted slots
Simple insert/remove
Low RAM overhead

Remember lwIP runs on tiny microcontrollers too.

udp_bind

24 February 2026 15:58

```
#define IP4
_ADDR_ANY
(&ip_addr_any)
```

```
const ip_addr_t
ip_addr_any =
IPADDR4
_INIT(IPADDR_ANY);
```

```
#define IPADDR_ANY
((u32_t)0x00000000UL)
```

```
udp_bind(struct udp_pcb *pcb, const ip_addr_t *ipaddr, u16_t port)
{
  struct udp_pcb *ipcb;
  u8_t rebind;
#if LWIP_IPV6 && LWIP_IPV6_SCOPES
  ip_addr_t zoned_ipaddr;
#endif /* LWIP_IPV6 && LWIP_IPV6_SCOPES */

  LWIP_ASSERT_CORE_LOCKED();

#if LWIP_IPV4
  /* Don't propagate NULL pointer (IPv4 ANY) to subsequent functions */
  if (ipaddr == NULL) {
    ipaddr = IP4_ADDR_ANY;
  }
#else /* LWIP_IPV4 */
  LWIP_ERROR("udp_bind: invalid ipaddr", ipaddr != NULL, return ERR_ARG);
#endif /* LWIP_IPV4 */

  LWIP_ERROR("udp_bind: invalid pcb", pcb != NULL, return ERR_ARG);

  LWIP_DEBUGF(UDP_DEBUG | LWIP_DBG_TRACE, ("udp_bind(ipaddr = "));
  ip_addr_debug_print(UDP_DEBUG | LWIP_DBG_TRACE, ipaddr);
  LWIP_DEBUGF(UDP_DEBUG | LWIP_DBG_TRACE, (", port = %"U16_F")\n",
port));

  rebind = 0;
  /* Check for double bind and rebind of the same pcb */
  for (ipcb = udp_pcbs; ipcb != NULL; ipcb = ipcb->next) {
    /* is this UDP PCB already on active list? */
    if (pcb == ipcb) {
      rebind = 1;
      break;
    }
  }
```

```
err = udp_bind(udp_pcb_global, IP_ADDR_ANY, UDP_PORT);
  if (err != ERR_OK) {
      xil_printf("Bind failed: %d\r\n", err);
      return -2;
  }

  udp_recv(udp_pcb_global, udp_recv_callback, NULL);

  // Target client (static IPv4)
  IP4_ADDR(&remote_ip, 192, 168, 1, 111);

  xil_printf("Listening on %d. Target: 192.168.1.111:%d\r\n",
           UDP_PORT, UDP_PORT);
  return 0;
}
```

UDP-PCB: flags

24 February 2026 17:36

u8_t flags is just a compact bitfield storing boolean configuration states that modify how the UDP PCB behaves — mainly for connected mode, checksum handling, and broadcast permission.

Stores configuration flags such as:

- connected state
- checksum behavior
- broadcast options

Unless you explicitly call:

```
udp_connect()
```

which sets a flag marking the PCB as “connected”.

Why u8_t Is Enough

u8_t = 8 bits.
Each bit can represent one independent boolean option.
Example:

Bit 0 → connected
Bit 1 → no-checksum
Bit 2 → broadcast allowed
Bit 3 → reuse
...

So one byte can store up to **8 independent flags**.
This is called a **bitmask**.

How It Actually Works Internally

Example conceptually:

flags = 00000101
Means:

- Bit 0 = 1 (connected)
- Bit 2 = 1 (broadcast enabled)
- Others = 0

lwIP defines macros like:

```
#define UDP_FLAGS_CONNECTED   0x01
#define UDP_FLAGS_NOCHKSUM    0x02
#define UDP_FLAGS_BROADCAST   0x04
```

So setting a flag is:

```
pcb->flags |= UDP_FLAGS_CONNECTED;
```

Checking:

```
if (pcb->flags & UDP_FLAGS_CONNECTED)
```

Example

Connected State

Set when you call:

```
udp_connect(pcb, &ip, port);
```

What it does:

- Stores remote_ip
- Stores remote_port
- Sets CONNECTED flag

Effect:
You can now use:

```
udp_send(pcb, p);
```

instead of udp_sendto().
In your UDP server:
X Not used
You are unconnected.

Because UDP is inherently connectionless.
So "connected" is just:
A software convenience mode.
It does NOT create:

- Handshake
- Session
- Reliability
- State in network

It only changes how lwIP behaves internally.

UDP-PCB: udp_recv_fn recv

24 February 2026 18:17

A user-defined function that lwIP calls automatically when a UDP packet arrives for that PCB.

This stores the callback function pointer.
When you do:

call back fn

```
udp_recv(pcb, udp_recv_fn, NULL);
```

lwIP internally sets:

```
pcb->recv = udp_echo_recv;
pcb->recv_arg = NULL;
```

```
typedef void (*udp_recv_fn)(void *arg, struct udp_pcb *pcb, struct pbuf *p,
    const ip_addr_t *addr, u16_t port);
```

```
void    udp_recv       (struct udp_pcb *pcb, udp_recv_fn recv,
                       void *recv_arg);
```

UDP-PCB: void *recv_arg

24 February 2026 19:16

A user-defined context pointer that lwIP stores and later passes back to your receive callback.

It allows you to attach **your own data structure** to a UDP PCB.
lwIP does not interpret it.
It just stores and forwards it.

Where It Is Set

When you call:

```
udp_recv(pcb, udp_echo_recv, my_pointer);
```

Internally lwIP does:

```
pcb->recv = udp_echo_recv;
pcb->recv_arg = my_pointer;
```

Later when a packet arrives:

```
pcb->recv(pcb->recv_arg, pcb, p, addr, port);
```

So whatever you passed becomes arg inside callback.

Practical Example (In Your Zynq Context)

Suppose you want to count packets per PCB.
Define a structure:

```
struct udp_context {
   u32_t packet_count;
};
```

Create and initialize:

```
static struct udp_context server_ctx;
server_ctx.packet_count = 0;

udp_recv(pcb, udp_echo_recv, &server_ctx);
```

Now in callback:

```
void udp_echo_recv(void *arg,
                   struct udp_pcb *pcb,
                   struct pbuf *p,
                   const ip_addr_t *addr,
                   u16_t port)
{
   struct udp_context *ctx = (struct udp_context *)arg;

   ctx->packet_count++;

   xil_printf("Packets received: %lu\n", ctx->packet_count);

   udp_sendto(pcb, p, addr, port);
   pbuf_free(p);
}
```

Now you have persistent state across packets.

UDP functions: udp_sendto, udp_new & udp_bind

25 February 2026 19:02

udp_sendto

Syntax

```
err_t udp_sendto(struct udp_pcb *pcb,
                 struct pbuf *p,
                 const ip_addr_t *dst_ip,
                 u16_t dst_port);
```

Purpose

Sends UDP packet to specified IP and port.

Operates On

- Walks pbuf chain
- Builds UDP/IP headers
- Driver creates TX descriptors
- GEM DMA transmits from RAM to wire

udp_new

Syntax

```
struct udp_pcb *udp_new(void);
```

Purpose

Creates a new UDP protocol control block (PCB).

Operates On

- Allocates udp_pcb structure from lwIP memory
- Stores socket-like state

udp_bind

Syntax

```
err_t udp_bind(struct udp_pcb *pcb,
               const ip_addr_t *ipaddr,
               u16_t port);
```

Purpose

Binds UDP PCB to local IP and port.

Operates On

- Registers port in lwIP internal table
- No hardware action

UDP functions: udp_recv

25 February 2026 19:05

```
void udp_recv(struct udp_pcb *pcb,
            udp_recv_fn recv,
            void *recv_arg);
```

Purpose

Registers receive callback function.

Operates On

- Stores function pointer inside PCB
- No hardware interaction