

基于 TM4C1294 在 ucosii 系统下使用 Lwip 的学习文档

姓名： 胡纪甚

导师： 任家豪

部门： 观测系统部

职务： 嵌入式软件

时间： 2024. 10. 14

目 录

目录

目 录	1
引 言	2
第一章 TM4C1294 微控制器和相关外设	3
1.1 硬件解读	3
1.1.1 Eth 外设	3
1.1.2 MCU 硬件	4
1.2 开发手册重点	5
第二章 LWIP 协议栈	8
2.1 Lwip-core 架构	8
2.1.1 网络层-数据链路层交互接口	8
2.1.2 网络层检测	9
2.1.3 传输层-网络层交互接口	10
2.1.4 传输层(核心+关键)	11
第三章 lwIP 的移植	14
3.1 lwip 基本库的移植	14
3.1.1 BSPLwip.c 里的初始化	15
3.1.2 tcpip.c 里的初始化	17
第四章 网络通信示例	19
4.1 netconn 接口函数详解	19
4.1.1 创建连接	19
4.1.2 连接管理	20
1. 服务器的管理函数	20
2. 客户端的管理函数	20
4.1.3 数据收发	21
1. 数据接收	21
2. 数据发送	21
4.1.4 连接关闭	22
4.1.5 删除连接	22
4.2 客户端程序编写示例 (netconn)	22
4.3 服务器端程序编写 (netconn)	24
第五章 常见问题与解决方案	26
5.1 硬件抽象层 (HAL) 不兼容	26
5.2 无操作系统情况下交互异常	26
5.3 客户端编程上位机回应 TCP 第二次握手问题	27

引言

本学习文档旨在为 ZTT 嵌入式网络开发提供一个系统化的指导，同时也是我这段时间学习的总结。目的是在 TM4C1294NCPDT 微控制器上成功实现 uC/OS-II 操作系统与 lwIP 协议栈的结合。通过详细的步骤和示例，文档将涵盖从硬件连接到软件移植的各个方面，确保能够理解和掌握相关技术。

文章的大致结构如下, 如果只想实现 lwip 的移植和使用, 建议直接看 3, 4, 5 这三章, 前两章章主要是对基础知识的一些理解和学习:

1. TM4C1294 微控制器: 介绍这款 MCU 与网络通信相关的硬件连接和网络接口
2. lwIP 协议栈: 介绍 lwIP 的架构和配置, 帮助开发者理解其工作原理和如何进行有效配置。
3. lwIP 在 uC/OS-II 中的移植: 详细说明 lwIP 在 uC/OS-II 环境中的移植步骤。
4. 网络通信示例: 提供 TCP 和 UDP 通信的示例代码及其运行结果, 帮助开发者理解实际应用中的网络通信。
5. 常见问题与解决方案: 列出在开发过程中可能遇到的问题及其解决方案, 帮助开发者快速排查和解决问题。

第一章 TM4C1294 微控制器和相关外设

1.1 硬件解读

1.1.1 Eth 外设

Ethernet 是一种局域网技术，它允许电子设备通过电缆或无线信号相互连接。在 PCB 设计中，Ethernet 部分通常包括物理连接点（RJ45 接口）、数据传输所需的电子元件以及控制这些操作的微控制器。如图 1 所示。

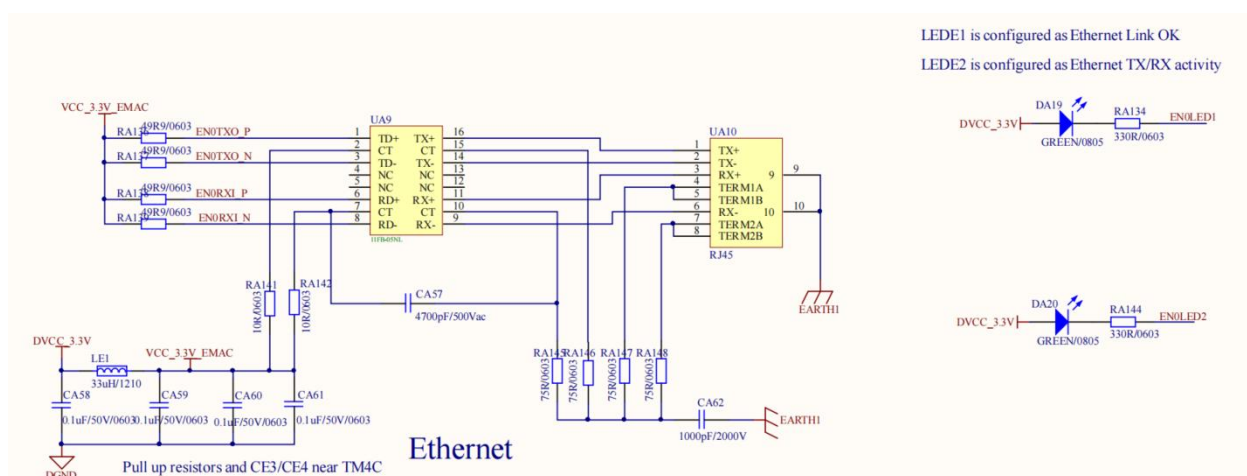


图 1 PCB 上的 Ethernet

Ethernet 部分涉及到的组件和概念如下：

1. **RJ45 接口：**物理连接器，用于通过双绞线（常见的以太网线）连接设备。有 8 个接触点，用于传输数据。
2. **VCC_3.3V_ETH：**这是 Ethernet 接口的电源线，提供 3.3 伏特的电压。
3. **CA57：**这是一个电容，通常用于滤除噪声或稳定电源线。
4. **LED1 和 LED2：**指示灯，用于显示 Ethernet 连接的状态，LEDE1 在连接成功时亮起，而 LEDE2 在数据传输时闪烁。如图 2 所示。
5. **TX+和 TX-：**发送数据的差分信号线。
6. **RX+和 RX-：**接收数据的差分信号线。
7. **TERM1A 和 TERM1B：**这些是终端电阻，用于匹配线路阻抗，减少信号反射。
8. **EARTH1：**接地符号，用于将电路的地线连接到共同的参考点。

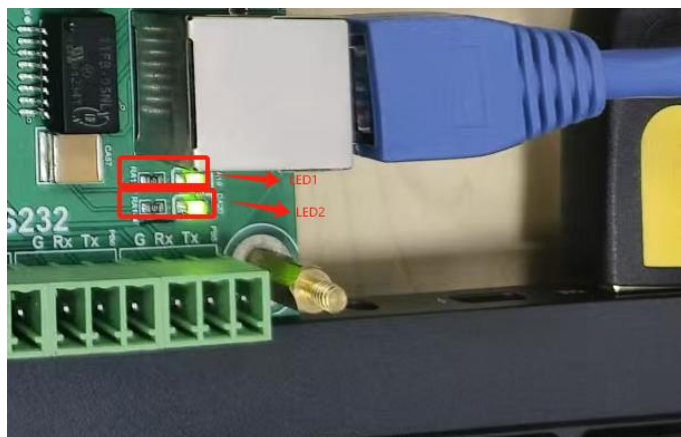


图 2 LED 指示灯

1.1.2 MCU 硬件

直接看看 mcu 关于 Eth 控制部分的布线逻辑，如图 3 所示：

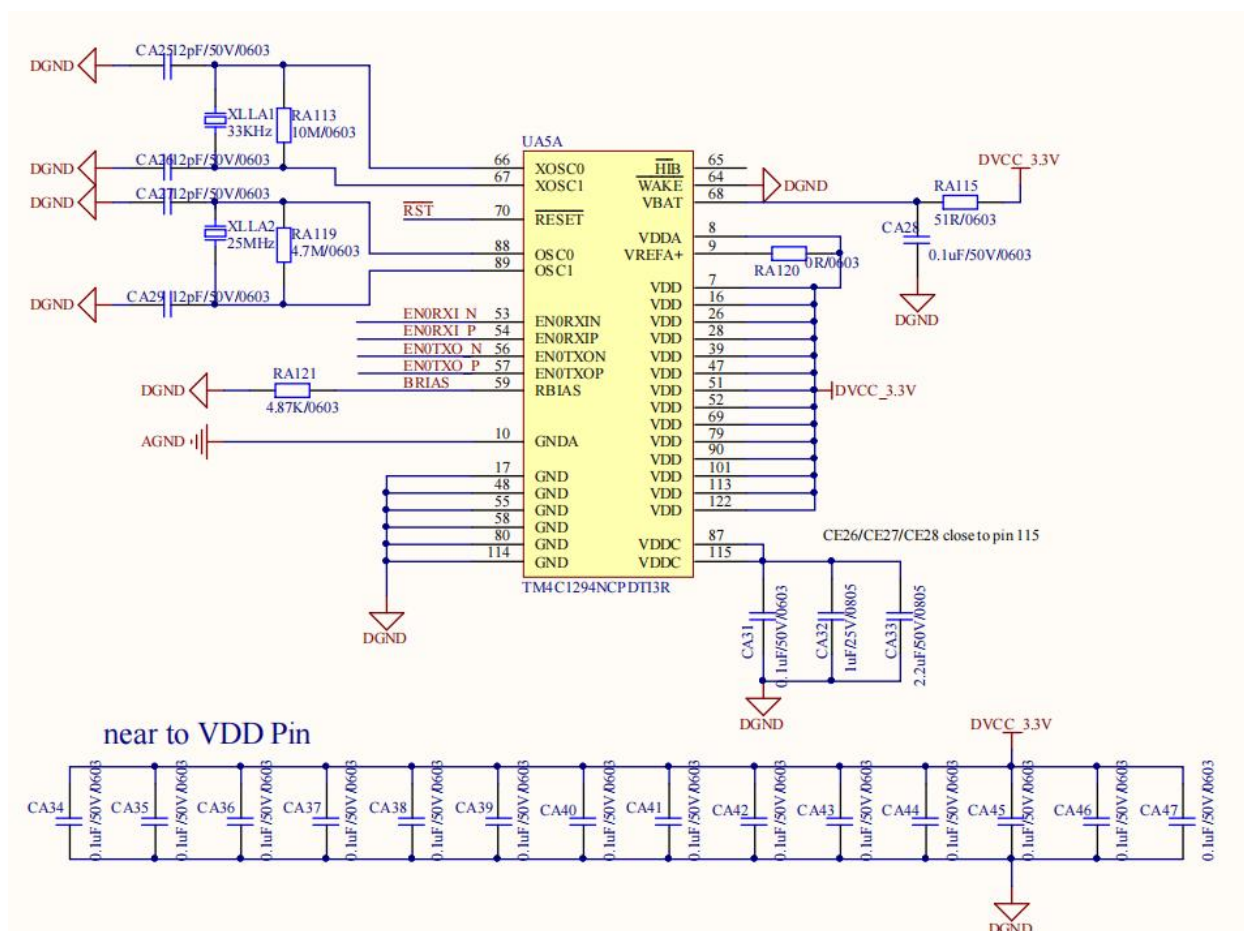


图 3 PCB 上 MCU 关于 ETH 的控制

Mcu 部分涉及到的组件和概念如下：

1. **TM4C1294NCPDTI3R**: 负责控制 Ethernet 通信部分的微控制器。

2. EN0TX0_P 和 EN0TX0_N: 这些是 Ethernet 接口的发送通道差分信号线, 用于发送数据。P (Positive) 代表正向信号线, N (Negative) 代表负向信号线。

3. EN0RXI_P 和 EN0RXI_N: 这些是 Ethernet 接口的接收通道差分信号线, 用于接收数据。

4. BRIAS: 偏置电阻, 用于调整信号线的水平。

5. RST: 复位线, 用于重置 Ethernet 控制器。

6. CA35 - CA47: 去耦电容, 用于滤除电源线上的高频噪声。

在实际应用中, 当网线插入 RJ45 接口时, 微控制器通过 TX+ 和 TX- 发送数据, 通过 RX+ 和 RX- 接收数据。LEDE1 和 LEDE2 提供视觉反馈, 指示连接和活动状态。微控制器通过内部的以太网控制器处理这些数据传输。

1.2 开发手册重点

先来一张图, 不然无法明白为什么要做这些事。如图 4 所示。

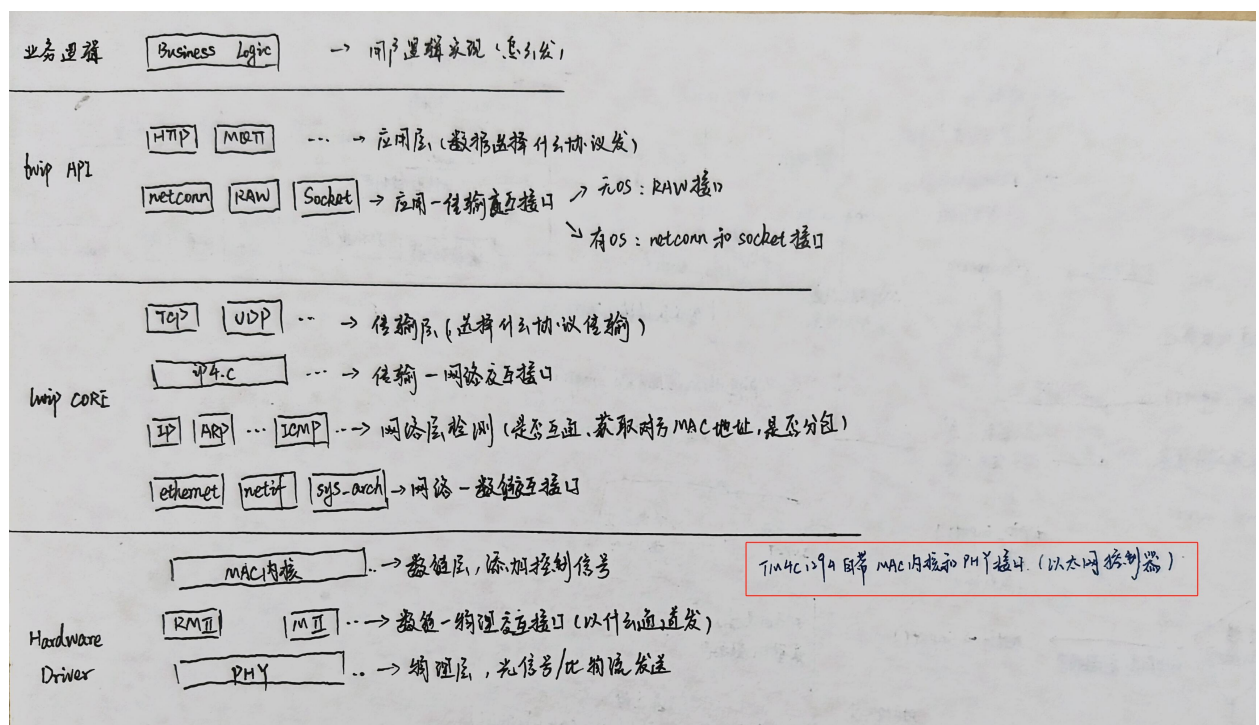


图 4 Lwip 层次结构 1

别的东西之后会讲到, 但目前来说对于 TM4C 来说最重要的一件事就是红框里的内容, 这个 MCU 自带了 MAC 内核和 PHY 接口。因此, 在手册里, 我们所要做的最重要的事情就是找到如何配置和初始化 MAC 内核和 PHY 接口。

来看看手册里怎么说的 (章节 20.5):

MAC 模块和寄存器在复位后被启用并上电。当复位完成后，应用程序应通过设置系统控制模块偏移地址 0x69C 的以太网控制器运行模式时钟门控寄存器（RCGCEMAC）中的 R0 位来启用对以太网 MAC 的时钟。当系统控制偏移地址 0xA9C 的预 MAC 寄存器（PREMAC）读值为 0x0000.0001 时，EMAC 寄存器已准备好被访问。

要使用默认配置启用以太网 PHY，步骤如下：

1. 为了在配置期间防止 PHY 在线上传输能量，将 EMACPC 寄存器中的 PHYHOLD 位设置为 1。

2. 通过写入 0x0000.0001 到系统控制偏移地址 0x630 的以太网 PHY 运行模式时钟门控寄存器（RCGCEPHY），启用 PHY 模块的时钟。当系统控制偏移地址 0xA30 的 PREPHY 寄存器中的 R0 位读为 1 时，继续初始化。

3. 通过在系统控制偏移地址 0x930 的 PCEPHY 寄存器中设置 P0 位，启用 PHY 的电源。当系统控制偏移地址 0xA30 的 PREPHY 寄存器中的 R0 位读为 1 时，PHY 寄存器已准备好编程。

讲了这么多其实也没啥用，因为 TM4C 已经帮我们封装好了，我们只要拿来用即可，关键部分代码如下：

```
1. void lwIPInit(uint32_t ui32SysClkHz, const uint8_t *pui8MAC, uint32_t ui32IPAddr,
2.               uint32_t ui32NetMask, uint32_t ui32GWAddr, uint32_t ui32IPMode)
3. {
4.     // 以太网外设启动
5.     SysCtlPeripheralEnable(SYSCTL_PERIPH_EMAC0);
6.     SysCtlPeripheralReset(SYSCTL_PERIPH_EMAC0);
7.
8.     // PHY 配置
9.     if ((EMAC_PHY_CONFIG & EMAC_PHY_TYPE_MASK) == EMAC_PHY_TYPE_INTERNAL)
10.    {
11.        if (SysCtlPeripheralPresent(SYSCTL_PERIPH_EPHY0))
12.        {
13.            SysCtlPeripheralEnable(SYSCTL_PERIPH_EPHY0);
14.            SysCtlPeripheralReset(SYSCTL_PERIPH_EPHY0);
15.        }
16.        else
17.        {
18.            while (1) {} // Internal PHY is not present on this part so hang here.
19.        }
20.    }
21.
22.    // 等待 MAC 复位完成
23.    while (!SysCtlPeripheralReady(SYSCTL_PERIPH_EMAC0)) {}
24.
```

```

25. // 配置 PHY
26. EMACPHYConfigSet(EMAC0_BASE, EMAC_PHY_CONFIG);
27.
28. // 初始化 MAC
29. EMACInit(EMAC0_BASE, ui32SysClkHz,
30.          EMAC_BCONFIG_MIXED_BURST | EMAC_BCONFIG_PRIORITY_FIXED,
31.          4, 4, 0);
32.
33. // 设置 MAC 配置选项
34. EMACConfigSet(EMAC0_BASE, (EMAC_CONFIG_FULL_DUPLEX |
35.                             EMAC_CONFIG_CHECKSUM_OFFLOAD |
36.                             EMAC_CONFIG_7BYTE_PREAMBLE |
37.                             EMAC_CONFIG_IF_GAP_96BITS |
38.                             EMAC_CONFIG_USE_MACADDR0 |
39.                             EMAC_CONFIG_SA_FROM_DESCRIPTOR |
40.                             EMAC_CONFIG_BO_LIMIT_1024),
41.               (EMAC_MODE_RX_STORE_FORWARD |
42.               EMAC_MODE_TX_STORE_FORWARD |
43.               EMAC_MODE_TX_THRESHOLD_64_BYTES |
44.               EMAC_MODE_RX_THRESHOLD_64_BYTES), 0);
45.
46. // 设置 MAC 地址
47. EMACAddrSet(EMAC0_BASE, 0, (uint8_t *)pui8MAC);
48. }

```

这段代码定义了一个名为 `lwIPInit` 的函数，主要目的就是刚刚说的那些，用于配置以太网控制器（EMAC）和物理层收发器（PHY）：

首先函数启用和复位以太网控制器（EMAC），确保其处于已知状态。通过调用 `SysCtlPeripheralEnable` 和 `SysCtlPeripheralReset`，函数为后续的配置做好准备。随后，代码检查是否使用内部 PHY，并确认该 PHY 是否存在。如果存在，函数会启用和复位 PHY；如果不存在，程序将进入无限循环，表示无法继续初始化。

在等待 EMAC 复位完成后，函数调用 `EMACPHYConfigSet` 来配置 PHY 的设置。以便与 MAC 层正常通信。接着，函数通过 `EMACInit` 初始化 MAC 层，设置系统时钟频率和其他配置参数，如混合突发模式和优先级固定。这些设置直接影响数据的发送和接收。

随后，函数调用 `EMACConfigSet` 来设置 MAC 的工作模式和配置选项，包括全双工模式、校验和卸载、前导码长度等。

最后函数通过 `EMACAddrSet` 设置设备的 MAC 地址，这是网络通信中设备的唯一标识符，确保设备能够在网络中被正确识别。

到了这一步，关于 TM4C 的底层数据链路层和物理层的配置便全部完成，我们将进入正式进入 Lwip。

第二章 LWIP 协议栈

Light weight IP, 意思是轻量化的 TCP/IP 协议, 是瑞典计算机科学院(SICS)的 Adam Dunkels 开发的一个小型开源的 TCP/IP 协议栈。LwIP 的设计初衷是: 用少量的资源消耗实现一个较为完整的 TCP/IP 协议栈, 其中“完整”主要指的是 TCP 协议的完整性, 实现的重点是在保持 TCP 协议主要功能的基础上减少对 RAM 的占用。此外 LwIP 既可以移植到操作系统上运行, 也可以在没有操作系统的情况下独立运行。

Lwip 的内容实在繁多, 无论是内存管理, 还是各种数据结构体都非常麻烦和复杂, 又不理解的结构体什么的直接找: [关于本项目 — \[野火\]LwIP 应用开发实战指南—基于野火 STM32 文档 \(embedfire.com\)](#)。这里只讲贴近项目工程和核心框架和我的学习理解。

2.1 Lwip-core 架构

再次回到刚刚的那张图, 不过这次我们将目光向上移动一点, 如图 5 所示, 最上面的一层是我们的主要编程工作, 放在第 4 章里讲:

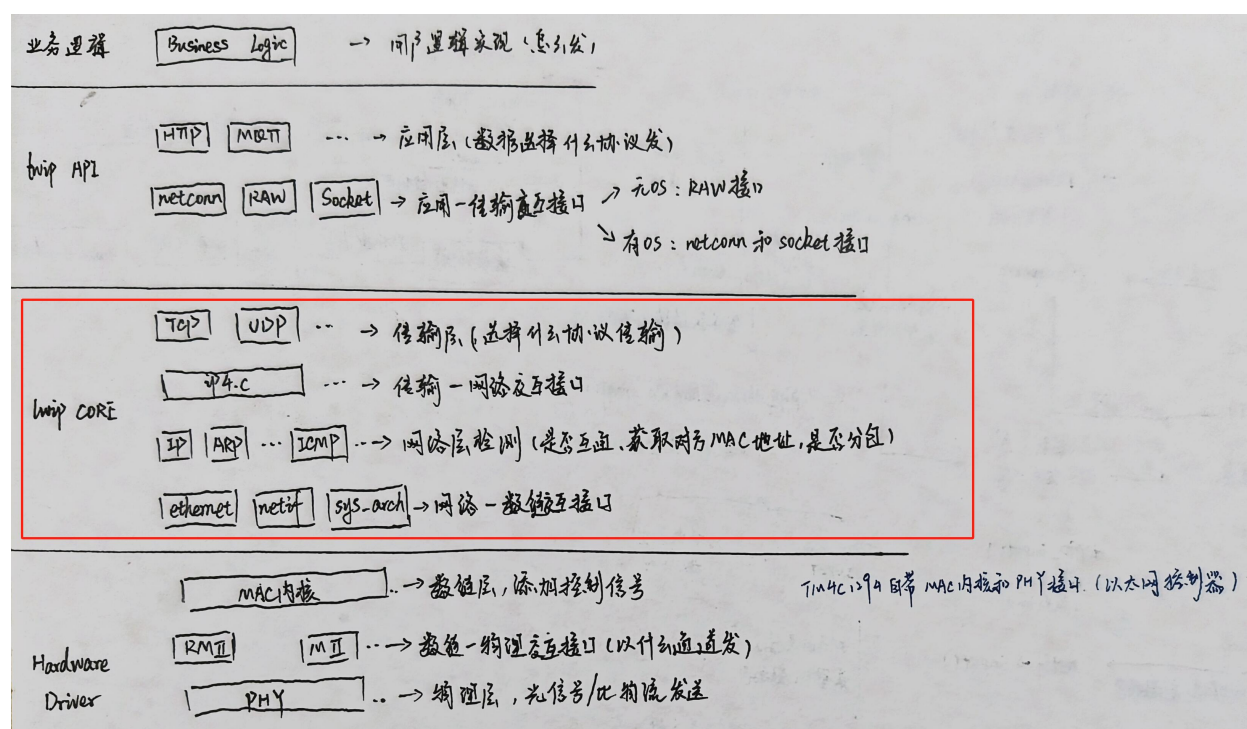


图 5 LWIP 层次结构 2

2.1.1 网络层-数据链路层交互接口

1. ethernet 组件

ethernet 组件负责以太网帧的封装与解封装。它处理以太网数据包的发送和接收，包括地址解析和以太网事件的管理。在 lwIP 中，相关的实现主要集中在 `ethernetif.c` 文件中。该文件包含了处理以太网帧的函数，`ethernetif_input()` 和 `ethernetif_output()`。`ethernetif_input()` 函数负责接收以太网帧并将其传递给 lwIP 的网络层，而 `ethernetif_output()` 函数则负责将网络层的数据封装成以太网帧并发送出去。通过这些函数，ethernet 组件实现了网络层与数据链路层之间的直接交互，确保数据能够在网络中正确传输。当然，这些都不是最底层的核心代码，真正处理输入输出的是 `low_level_input()` 和 `low_level_output()`。

2. netif 组件

netif 组件是 lwIP 中的网络接口抽象层，负责管理网络接口的状态和数据传输。它定义了网络接口的结构体和相关操作，允许 lwIP 支持多种网络接口类型。在 lwIP 中，netif 的实现主要在 `netif.c` 文件中。该文件包含了网络接口的初始化、配置和管理函数，例如 `netif_add` 和 `netif_set_default`。`netif_add` 函数用于添加新的网络接口，而 `netif_set_default` 则用于设置默认的网络接口。通过这些函数，netif 组件能够管理不同的网络接口，并为网络层提供统一的接口。

3. sys_arch 组件

sys_arch 组件提供了与操作系统相关的抽象接口，包括线程管理、同步机制和定时器支持。在 lwIP 中，sys_arch 的实现通常在 `arch/sys_arch.c` 文件中。该文件定义了 lwIP 在不同操作系统环境下的系统调用和同步机制，确保 lwIP 能够在多线程环境中安全运行。通过 sys_arch，lwIP 可以在不同的操作系统上实现相同的功能，提供了良好的可移植性。

2.1.2 网络层检测

在 lwIP 中，数据发送的过程涉及多个协议的协作。首先，当应用层数据需要发送时，数据会通过 TCP 或 UDP 封装成数据包。lwIP 将数据包进一步封装成 IP 数据包，并检查 ARP 表以获取目标设备的 MAC 地址。如果 ARP 表中没有目标 MAC 地址，lwIP 会发送 ARP 请求以获取该地址。此时，设备 A 发送的 ARP 请求会被网络中的所有设备接收，只有目标设备 B 会响应，提供其 MAC 地址。

1. ARP（地址解析协议）

ARP 的主要功能是将网络层的 IP 地址转换为数据链路层的 MAC 地址。当设备 A 想要与设备 B 通信，但不知道 B 的 MAC 地址时，A 会发送一个 ARP 请求。这个请求是一个广播消息，询问网络中“谁拥有这个 IP 地址？”所有连接到同一网络的设备都会接收到这个请求。设备 B 收到请求后，会识别出自己的 IP 地址，并发送一个 ARP 响应，包含其 MAC 地址。设备 A 接收到这个响应后，将 IP 地址与 MAC 地址进行映射，并将其缓存到 ARP 表中，以便后续通信时使用。如图 6 所示：

No.	Time	Source	Destination	Protocol	Length	Info
1	0.000000	WistronInfoC_4e:fc:d1	Broadcast	ARP	42	Who has 192.168.1.100? Tell 192.168.1.112
2	0.001925	ba:0b:00:00:00:00	WistronInfoC_4e:fc:d1	ARP	64	192.168.1.100 is at ba:0b:00:00:00:00

图 6 ARP 抓包

2. IP（互联网协议）

IP 协议负责在网络中传输数据包。它为数据包提供寻址和路由功能。当应用层的数据需要发送时，lwIP 会将其封装成 IP 数据包，添加源和目标 IP 地址。此时，lwIP 会检查目标 IP 地址，以决定数据包的发送路径。**如果目标设备在同一子网内，数据包将直接发送（之后有个错误就是基于这一点）**；如果不在同一子网，数据包将发送到默认网关。IP 协议还负责处理数据包的分片问题。如果数据包的大小超过了网络的最大传输单元（MTU），IP 协议会将其分片，以便在网络中传输。

3. ICMP（互联网控制消息协议）

ICMP 协议用于发送控制消息和错误报告，帮助管理和诊断网络。它在网络通信中扮演着重要角色。例如，当 IP 数据包无法到达目标时，路由器会发送 ICMP 错误消息（如目标不可达）回源设备。这种机制使得源设备能够及时了解网络状态并采取相应措施。此外，ICMP 还支持回显请求和应答功能，常用于网络连通性测试（如 ping 命令）。源设备发送 ICMP 回显请求，目标设备收到后会回复 ICMP 回显应答，从而确认网络的连通性。

2.1.3 传输层-网络层交互接口

ip.c 文件实现了 lwIP 中 IPv4 协议的核心功能，包括数据包的接收、转发和发送。它通过一系列函数处理 IP 数据包的生命周期，确保数据能够在网络中正确传输。文件中的调试信息和统计功能也为开发和调试提供了便利。整体上，该文件是 lwIP 协议栈中不可或缺的一部分，负责实现 IP 层的基本功能。

在函数实现部分：

ip_route() 函数。该函数用于查找适合给定目标 IP 地址的网络接口。它遍历所有网络接口，检查每个接口的状态和网络掩码，找到匹配的接口。如果没有找到合适的接口，则返回默认接口。

ip_canforward() 函数。该函数用于判断一个 IP 数据包是否可以被转发。它检查数据包的标志位、地址类型（如广播、组播）以及是否是实验性地址等，确保数据包符合转发条件。

ip_forward() 函数。该函数负责转发 IP 数据包。它首先检查数据包是否可以转发，然后找到合适的网络接口，减少 TTL（生存时间），并更新校验和。如果 TTL 减少到 0，则发送 ICMP 超时消息。最后，它将数据包发送到目标接口。

ip_input() 是处理接收到的 IP 数据包的主要函数。它执行以下步骤：检查 IP 版本是否为 IPv4，验证 IP 头部的长度和数据包的总长度，计算并验证 IP 校验和。根据目标地址确定数据包是否是发给本地接口的。如果不是，则尝试转发。如果数据包是分片的，调用重组函数。最后，将数据包传递给相应的上层协议（如 TCP、UDP、ICMP）。

ip_output_if() 函数用于通过指定的网络接口发送 IP 数据包。它构建 IP 头部，计算校验和，并处理 IP 选项。如果数据包的长度超过接口的 MTU（最大传输单元），则调用分片函数。

ip_output() 是一个简单的接口，用于查找合适的网络接口并调用 ip_output_if() 进行实际的数据包发送。

ip_debug_print() 函数用于调试，打印 IP 头部的详细信息，包括版本、头部长度的、服务类型、总长度、标识符、TTL、协议、校验和、源地址和目的地址等。

2.1.4 传输层(核心+关键)

因为暂时只实现了 TCP 通信，所以只讲讲 TCP 协议。

先看这张图 6，详细的展示了 tcp 三次握手和四次挥手的全过程：

The image shows a Wireshark packet capture window titled '正在捕获以太网'. The packet list table is as follows:

No.	Time	Source	Destination	Protocol	Length	Info
1	0.000000	WistronInfoC_4e:fc:d1	Broadcast	ARP	42	Who has 192.168.1.100? Tell 192.168.1.112
2	0.001925	ba:0b:00:00:00:00	WistronInfoC_4e:fc:d1	ARP	64	192.168.1.100 is at ba:0b:00:00:00:00
3	0.001938	192.168.1.112	192.168.1.100	TCP	66	51134 → 4567 [SYN] Seq=0 Win=64240 Len=0 MSS=1460 WS=256 SACK_PERM
4	0.007938	192.168.1.100	192.168.1.112	TCP	60	4567 → 51134 [SYN, ACK] Seq=0 Ack=0 Win=8760 Len=0 MSS=1460
5	0.008006	192.168.1.112	192.168.1.100	TCP	54	51134 → 4567 [ACK] Seq=1 Ack=0 Win=64240 Len=0
6	2.547553	192.168.1.112	192.168.1.100	TCP	61	51134 → 4567 [PSH, ACK] Seq=1 Ack=1 Win=64240 Len=7
7	2.551265	192.168.1.100	192.168.1.112	TCP	62	4567 → 51134 [PSH, ACK] Seq=1 Ack=8 Win=8753 Len=8
8	2.601527	192.168.1.112	192.168.1.100	TCP	54	51134 → 4567 [ACK] Seq=8 Ack=9 Win=64232 Len=0
9	4.405049	192.168.1.112	192.168.1.100	TCP	54	51134 → 4567 [FIN, ACK] Seq=8 Ack=9 Win=64232 Len=0
10	4.407054	192.168.1.100	192.168.1.112	TCP	60	4567 → 51134 [ACK] Seq=9 Ack=0 Win=8752 Len=0
11	4.412369	192.168.1.100	192.168.1.112	TCP	60	4567 → 51134 [FIN, ACK] Seq=0 Ack=0 Win=8752 Len=0
12	4.412407	192.168.1.112	192.168.1.100	TCP	54	51134 → 4567 [ACK] Seq=8 Ack=0 Win=64232 Len=0
13	7.304419	fe80::5f98:7d81:f5b0:ceeb	ff02::1:2	DHCPv6	150	Solicit XID: 0x1b8aed CID: 000100012e652f2484ba594efcd1

Red boxes highlight the three-way handshake (packets 3, 4, 5) and the four-way teardown (packets 9, 10, 11, 12). Red arrows point to these sections with labels '三次握手' and '四次挥手'.

图 6 抓包 TCP

TCP 协议根据连接时接收到报文的不同类型，采取相应动作也不同，还要处理各个状态的关系，如当收到握手报文时候、超时的时候、用户主动关闭的时候等都需要不一样的状态去采取不一样的处理。在 LwIP 中，为了实现 TCP 协议的稳定连接，采用数组的形式定义了 11 种连接时候的状态：

```
1.  const char * const tcp_state_str[] = {
2.     "CLOSED",
3.     "LISTEN",
4.     "SYN_SENT",
5.     "SYN_RCVD",
6.     "ESTABLISHED",
7.     "FIN_WAIT_1",
8.     "FIN_WAIT_2",
9.     "CLOSE_WAIT",
10.    "CLOSING",
11.    "LAST_ACK",
12.    "TIME_WAIT"
13. };
```

结合刚刚的内容，看看 tcp 的状态转移，图 7：

虚线：表示服务器的状态转移。

实线：表示客户端的状态转移。

图中所有“关闭”、“打开”都是应用程序主动处理。

图中所有的“超时”都是内核超时处理。

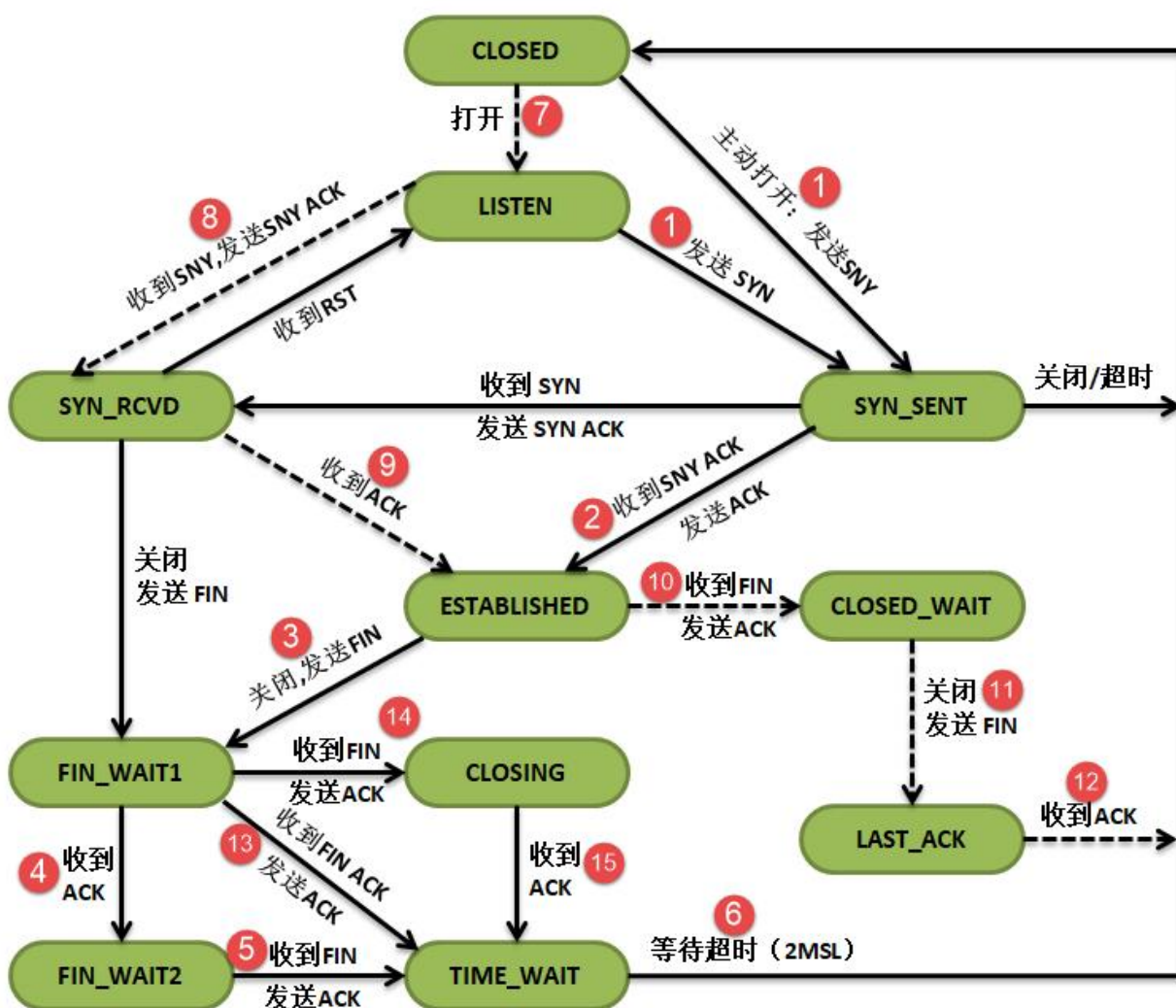


图 7 TCP 状态转移

最后来看一下 TCP 工作流程的示意，在脑海里有个框架：

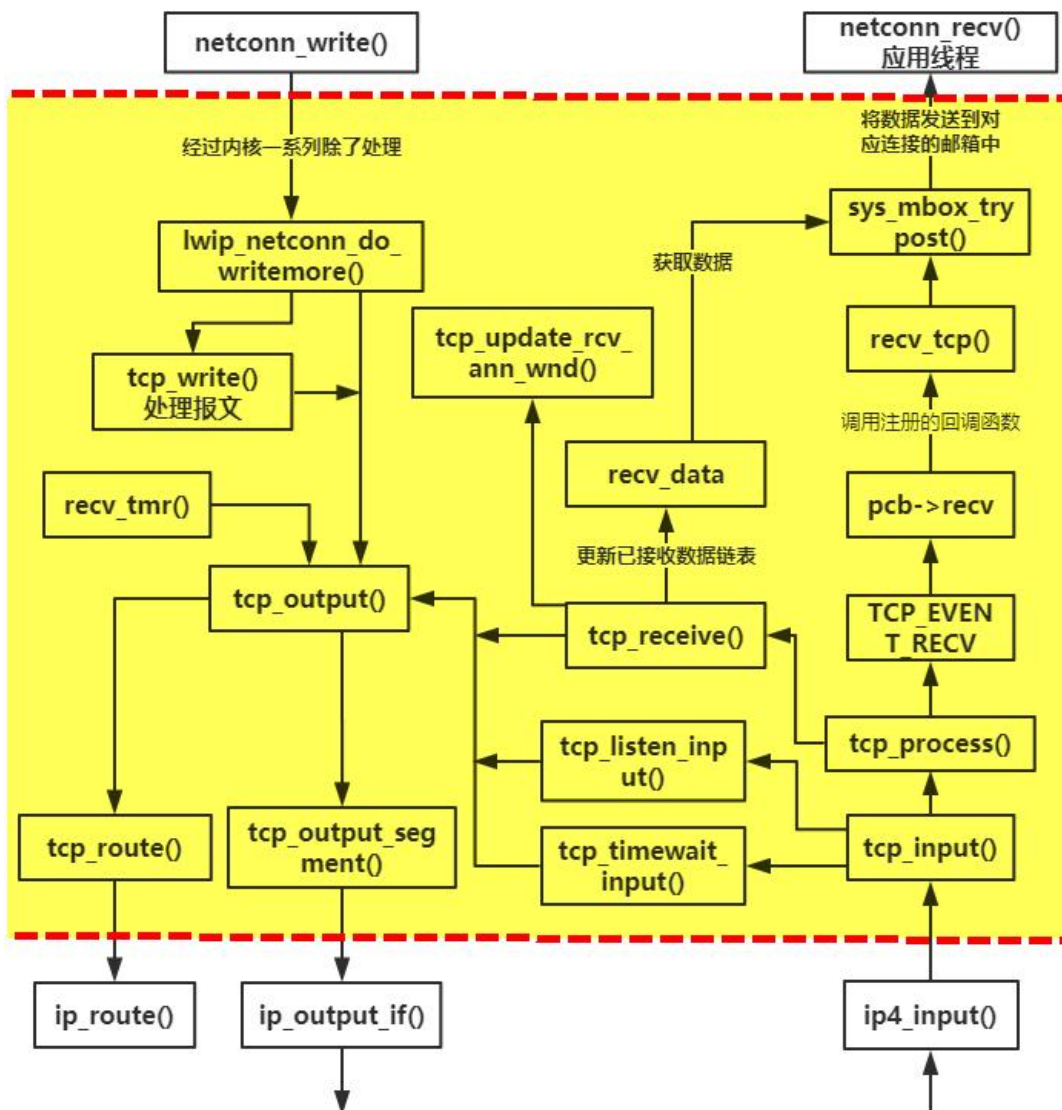


图 8 TCP 流程示意

第三章 lwIP 的移植

3.1 lwip 基本库的移植

先看这张图 9，展示了 lwip 初始化启动的全流程。

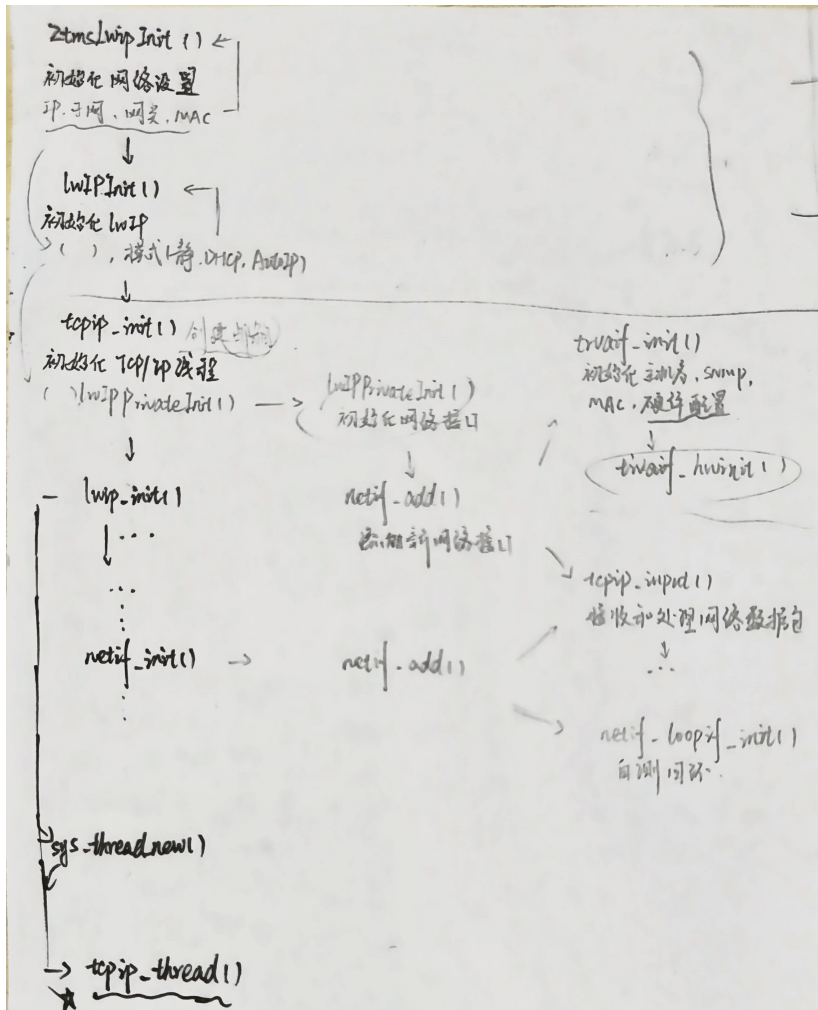


图 9 LWIP 初始化代码逻辑

3.1.1 app.c 里的初始化

1. `static NETWORK_IPADDR gs_localIpAddr = {192,168,1,100};`
2. `static NETWORK_IPADDR gs_lcaolMaskAddr = {255,255,255,0};`
3. `static NETWORK_IPADDR gs_localGwAddr = {192,168,1,1};`
4. `static uint64_t gs_mac = 2001 + 1001;`
- 5.
6. `static void AppTaskStart (void *p_arg)`

```

7.  {
8.    ...
9.    BSP_LwipInit(gs_localIpAddr, gs_lcaolMaskAddr, gs_localGwAddr, gs_mac);
10.   ...
11.  }

```

3.1.2 BSPLwip.c 里的初始化

```

1.  void BSP_LwipInit(NETWORK_IPADDR IpAddr, NETWORK_IPADDR MaskAddr, NETWORK_IPADDR GwAddr, uint64_t
    Mac)
2.  {
3.      uint8_t pui8MACArray[6];
4.      struct ip_addr ip_addr;
5.      struct ip_addr net_mask;
6.      struct ip_addr gw_addr;
7.      // MAC 地址分解
8.      pui8MACArray[0] = Mac & 0xff;
9.      pui8MACArray[1] = (Mac & 0xff00) >> 8;
10.     pui8MACArray[2] = (Mac & 0xff0000) >> 16;
11.     pui8MACArray[3] = (Mac & 0xff000000) >> 24;
12.     pui8MACArray[4] = (Mac & 0xff00000000) >> 32;
13.     pui8MACArray[5] = (Mac & 0xff0000000000) >> 40;
14.     // IP 地址组合
15.     ip_addr.addr = (((uint32_t)IpAddr.Addr0) << 24) |
16.                   (((uint32_t)IpAddr.Addr1) << 16) |
17.                   (((uint32_t)IpAddr.Addr2) << 8) |
18.                   ((uint32_t)IpAddr.Addr3);
19.     net_mask.addr = (((uint32_t)MaskAddr.Addr0) << 24) |
20.                    (((uint32_t)MaskAddr.Addr1) << 16) |
21.                    (((uint32_t)MaskAddr.Addr2) << 8) |
22.                    ((uint32_t)MaskAddr.Addr3);
23.     gw_addr.addr = (((uint32_t)GwAddr.Addr0) << 24) |
24.                   (((uint32_t)GwAddr.Addr1) << 16) |
25.                   (((uint32_t)GwAddr.Addr2) << 8) |
26.                   ((uint32_t)GwAddr.Addr3);
27.     // LwIP 初始化调用
28.     lwIPInit(BspGetSysClock(), pui8MACArray, ip_addr.addr,
29.              net_mask.addr, gw_addr.addr, IPADDR_USE_STATIC);
30. }

```

```

1.  void lwIPInit(uint32_t ui32SysClkKHz, const uint8_t *pui8MAC, uint32_t ui32IPAddr,
2.              uint32_t ui32NetMask, uint32_t ui32GWAddr, uint32_t ui32IPMode)
3.  {

```

```

4.      // 参数选择
5.      #if LWIP_DHCP && LWIP_AUTOIP
6.          ASSERT((ui32IPMode == IPADDR_USE_STATIC) ||
7.                (ui32IPMode == IPADDR_USE_DHCP) ||
8.                (ui32IPMode == IPADDR_USE_AUTOIP));
9.      #elif LWIP_DHCP
10.         ASSERT((ui32IPMode == IPADDR_USE_STATIC) ||
11.               (ui32IPMode == IPADDR_USE_DHCP));
12.      #elif LWIP_AUTOIP
13.         ASSERT((ui32IPMode == IPADDR_USE_STATIC) ||
14.               (ui32IPMode == IPADDR_USE_AUTOIP));
15.      #else
16.         ASSERT(ui32IPMode == IPADDR_USE_STATIC);
17.      #endif
18.
19.      // 以太网外设启动
20.      SysCtlPeripheralEnable(SYSCTL_PERIPH_EMAC0);
21.      SysCtlPeripheralReset(SYSCTL_PERIPH_EMAC0);
22.
23.      // PHY 配置
24.      if ((EMAC_PHY_CONFIG & EMAC_PHY_TYPE_MASK) == EMAC_PHY_TYPE_INTERNAL)
25.      {
26.          if (SysCtlPeripheralPresent(SYSCTL_PERIPH_EPHY0))
27.          {
28.              SysCtlPeripheralEnable(SYSCTL_PERIPH_EPHY0);
29.              SysCtlPeripheralReset(SYSCTL_PERIPH_EPHY0);
30.          }
31.          else
32.          {
33.              while (1) {} // Internal PHY is not present on this part so hang here.
34.          }
35.      }
36.
37.      // 等待MAC 复位完成
38.      while (!SysCtlPeripheralReady(SYSCTL_PERIPH_EMAC0)) {}
39.
40.      // 配置PHY
41.      EMACPHYConfigSet(EMAC0_BASE, EMAC_PHY_CONFIG);
42.
43.      // 初始化MAC
44.      EMACInit(EMAC0_BASE, ui32SysClkHz,
45.              EMAC_BCONFIG_MIXED_BURST | EMAC_BCONFIG_PRIORITY_FIXED,
46.              4, 4, 0);
47.

```

```

48. // 设置MAC 配置选项
49. EMACConfigSet(EMAC0_BASE, (EMAC_CONFIG_FULL_DUPLEX |
50.                          EMAC_CONFIG_CHECKSUM_OFFLOAD |
51.                          EMAC_CONFIG_7BYTE_PREAMBLE |
52.                          EMAC_CONFIG_IF_GAP_96BITS |
53.                          EMAC_CONFIG_USE_MACADDR0 |
54.                          EMAC_CONFIG_SA_FROM_DESCRIPTOR |
55.                          EMAC_CONFIG_BO_LIMIT_1024),
56.              (EMAC_MODE_RX_STORE_FORWARD |
57.              EMAC_MODE_TX_STORE_FORWARD |
58.              EMAC_MODE_TX_THRESHOLD_64_BYTES |
59.              EMAC_MODE_RX_THRESHOLD_64_BYTES), 0);
60.
61. // 设置MAC 地址
62. EMACAddrSet(EMAC0_BASE, 0, (uint8_t *)pui8MAC);
63.
64. // 将 IP 地址、子网掩码和网关地址保存到全局变量中，以便后续使用。
65. g_ui32IPMode = ui32IPMode;
66. g_ui32IPAddr = ui32IPAddr;
67. g_ui32NetMask = ui32NetMask;
68. g_ui32GWAddr = ui32GWAddr;
69.
70. // LwIP 初始化
71. #if NO_SYS
72.     lwIPPrivateInit(0);
73. #else
74.     tcpip_init(lwIPPrivateInit, 0);
75. #endif
76. }

```

到此为止，已经完成了 MAC，PHY 和初始化。随后进行的就是对 lwip 的初始化，内存空间的分配，和 tcpip 进程的创建。

3.1.3 tcpip.c 里的初始化

```

1. void
2. tcpip_init(tcpip_init_done_fn initfunc, void *arg)
3. {
4.     // 初始化 Lwip
5.     lwip_init();
6.
7.     // 设置回调函数参数
8.     tcpip_init_done = initfunc;
9.     tcpip_init_done_arg = arg;

```

```

10. // 创建邮箱
11. if(sys_mbox_new(&mbox, TCPIP_MBOX_SIZE) != ERR_OK) {
12.     LWIP_ASSERT("failed to create tcpip_thread mbox", 0);
13. }
14. // 创建互斥锁
15. #if LWIP_TCPIP_CORE_LOCKING
16. if(sys_mutex_new(&lock_tcpip_core) != ERR_OK) {
17.     LWIP_ASSERT("failed to create lock_tcpip_core", 0);
18. }
19. #endif /* LWIP_TCPIP_CORE_LOCKING */
20.
21. // 创建TCP/IP 线程
22. sys_thread_new(TCPIP_THREAD_NAME, tcpip_thread, NULL, TCPIP_THREAD_STACKSIZE, TCPIP_THREAD_PRIO)
    ;
23. }

```

lwip_init()里详尽的展示了所有可以初始化的部分，这些都需要使能信号的参与，而 sys_thread_new()则是创建了一个 tcp 进程，在有信息时进行处理，而没有信息时则进行保活探测。使能信号则是在 lwipopts.h 里进行选择。当我们完成这些后，ping 一下，如果通了，说明移植完成。

```

Microsoft Windows [版本 10.0.22631.4037]
(c) Microsoft Corporation. 保留所有权利。

C:\Users\hujsh>ping 192.168.1.100

正在 Ping 192.168.1.100 具有 32 字节的数据:
来自 192.168.1.100 的回复: 字节=32 时间=6ms TTL=255
来自 192.168.1.100 的回复: 字节=32 时间=3ms TTL=255
来自 192.168.1.100 的回复: 字节=32 时间=2ms TTL=255
来自 192.168.1.100 的回复: 字节=32 时间=3ms TTL=255

192.168.1.100 的 Ping 统计信息:
    数据包: 已发送 = 4, 已接收 = 4, 丢失 = 0 (0% 丢失),
    往返行程的估计时间(以毫秒为单位):
        最短 = 2ms, 最长 = 6ms, 平均 = 3ms

C:\Users\hujsh>

```

图 10 LWIP 移植成功

第四章 网络通信示例

4.1 netconn 接口函数详解

具体的实现细节就不介绍少了，主要讲讲用途和参数。对于 netconn 的函数编程，主要是按照以下的流程进行的：

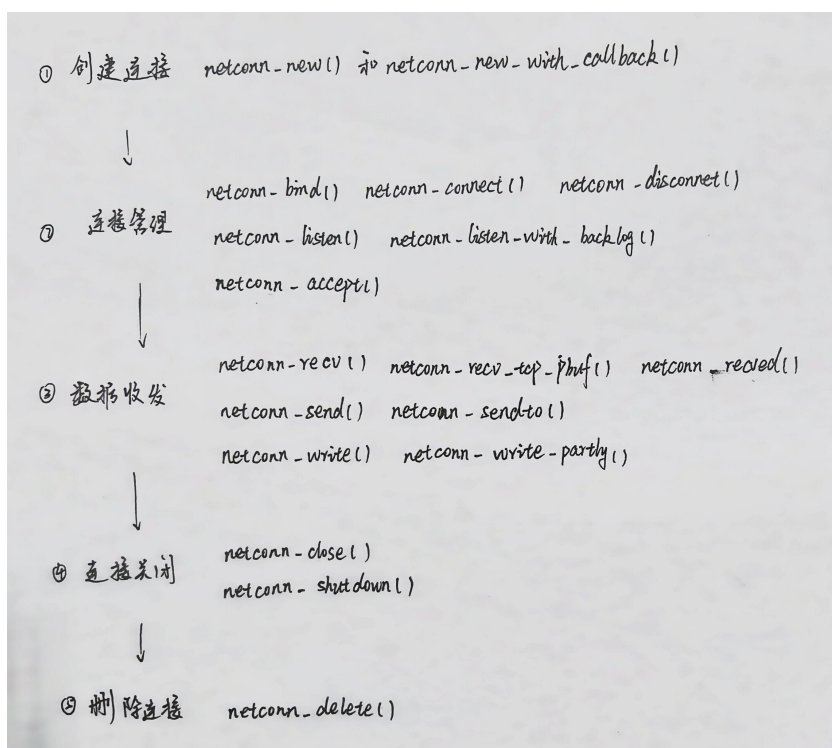


图 11 netconn 网络编程

4.1.1 创建连接

这两个函数的功能是为了“创建一个网络连接”，他们的主要区别是在于是否支持回调函数。定义如下：

```
1. /* Network connection functions: */
2. #define netconn_new(t) netconn_new_with_proto_and_callback(t, 0, NULL)
3. #define netconn_new_with_callback(t, c) netconn_new_with_proto_and_callback(t, 0, c)
```

这里来探讨两个问题，一个是在 lwip 中“创建网络连接”是一个怎么样的结构体；另一个是这里的回调函数有什么用。

在 lwIP 中，“创建网络连接”涉及到一个名为 **struct netconn** 的结构体。这个结构体是 lwIP 网络连接 API 的核心，封装了与网络连接相关的所有信息。struct netconn

包含多个关键字段，包括连接类型（TCP、UDP 等）、协议控制块（PCB）、接收邮箱、发送邮箱以及用于异步操作的信号量。这些字段使得 `netconn` 能够管理连接的状态、处理数据的接收与发送，并支持多线程环境下的并发访问。

而回调函数在 lwIP 中主要用于事件驱动编程。通过回调函数，用户可以在特定事件发生时执行自定义逻辑，例如连接建立、数据接收或连接关闭等。这种机制使得 lwIP 能够实现异步操作，提升了系统的响应能力和灵活性。当网络事件发生时，lwIP 会调用相应的回调函数，允许开发者在不阻塞主程序的情况下处理这些事件。这种设计不仅提高了资源利用率，还使得应用程序能够更好地适应实时性要求高的场景。

由此可知，所谓的创建一个网络连接，本质上就是新建一个 `netconn` 结构体，用于对这个网络结构进行信息管理。

4.1.2 连接管理

1. 服务器的管理函数

当我们使用下位机当服务器进行 TCP 通信的时候，是绕不开这几个函数的：

```
1.  err_t netconn_bind(struct netconn *conn, ip_addr_t *addr, u16_t port); 0, c)
2.  err_t netconn_listen_with_backlog(struct netconn *conn, u8_t backlog);
3.  #define netconn_listen(conn) netconn_listen_with_backlog(conn, TCP_DEFAULT_LISTEN_BACKLOG)
4.  err_t netconn_accept(struct netconn *conn, struct netconn **new_conn);
```

`netconn_bind()` 函数用于将一个网络连接绑定到指定的本地地址和端口。这是服务器端操作的第一步，允许服务器在特定的地址和端口上监听传入的连接请求。调用此函数后，连接将处于“绑定”状态，准备接受来自客户端的连接。成功绑定后，返回 `ERR_OK`，否则返回相应的错误代码。

`netconn_listen_with_backlog()` 函数将连接设置为监听状态，允许它接受传入的连接请求。`backlog` 参数指定在连接队列中可以等待的最大连接数。这对于服务器应用程序至关重要，因为它决定了在处理当前连接时可以接受多少新的连接请求。成功设置监听后，返回 `ERR_OK`。

`netconn_listen()` 是一个宏，简化了监听操作。它调用 `netconn_listen_with_backlog`，使用默认的监听队列长度（`TCP_DEFAULT_LISTEN_BACKLOG`）。这使得我们在不需要指定队列长度的情况下，快速将连接设置为监听状态。

`netconn_accept` 函数用于接受一个传入的连接请求。当服务器处于监听状态时，可以调用此函数来接受客户端的连接。成功接受后，返回一个新的 `netconn` 结构体，表示与客户端的连接。

2. 客户端的管理函数

同样，当我们使用下位机作为客户端进行 tcp 通信时，也是有两个关键函数：

```
1.  err_t netconn_connect(struct netconn *conn, ip_addr_t *addr, u16_t port);
2.  err_t netconn_disconnect(struct netconn *conn);
```

`netconn_connect` 函数用于客户端连接到指定的远程地址和端口。此函数会尝试建立与目标主机的 TCP 或 UDP 连接。成功连接后，连接状态将更新为已连接，允许数据的发送和接收。如果连接失败，函数将返回相应的错误代码，指示失败的原因。

`netconn_disconnect` 函数则用于断开与远程主机的连接。此操作通常在完成数据传输后进行，以释放资源并关闭连接。调用此函数后，连接将不再处于活动状态，后续的发送或接收操作将失败。成功断开连接后，返回 `ERR_OK`。

4.1.3 数据收发

1. 数据接收

```
1.  err_t  netconn_recv(struct netconn *conn, struct netbuf **new_buf);
2.  err_t  netconn_recv_tcp_pbuf(struct netconn *conn, struct pbuf **new_buf);
3.  void    netconn_recved(struct netconn *conn, u32_t length);
```

`netconn_recv(struct netconn *conn, struct netbuf **new_buf)`: 这个函数用于从指定的网络连接中接收数据。调用此函数时，lwIP 会检查连接的状态，并在有数据可用时将其读取到一个新的 `netbuf` 结构中。`netbuf` 是 lwIP 中用于封装数据的结构，支持多种协议。函数返回一个错误码，指示操作的成功与否。成功时，`new_buf` 将指向包含接收到数据的缓冲区。

`netconn_recv_tcp_pbuf(struct netconn *conn, struct pbuf **new_buf)`: 这个函数专门用于从 TCP 连接中接收数据。与 `netconn_recv` 类似，但它返回的是一个 `pbuf` 结构，适用于需要直接操作数据包的场景。`pbuf` 是 lwIP 中用于处理网络数据包的基本数据结构，支持链式缓冲区，便于处理大数据量的接收。

`netconn_recved(struct netconn *conn, u32_t length)`: 这个函数用于通知 lwIP 已经成功接收了指定长度的数据。它通常在应用层处理完接收到的数据后调用，以便 lwIP 可以更新内部状态，进行流量控制。这有助于 lwIP 管理接收缓冲区，确保不会因为数据未被处理而导致数据丢失。

2. 数据发送

```
1.  err_t  netconn_sendto(struct netconn *conn, struct netbuf *buf,
2.                      ip_addr_t *addr, u16_t port);
3.  err_t  netconn_send(struct netconn *conn, struct netbuf *buf);
4.  err_t  netconn_write_partly(struct netconn *conn, const void *dataptr, size_t size,
5.                             u8_t apiflags, size_t *bytes_written);
6.  #define netconn_write(conn, dataptr, size, apiflags) \
7.      netconn_write_partly(conn, dataptr, size, apiflags, NULL)
```

`netconn_sendto(struct netconn *conn, struct netbuf *buf, ip_addr_t *addr, u16_t port)`: 这个函数用于通过指定的地址和端口发送数据。它允许开发者在发送数据时指定目标地址，适用于无连接的协议（如 UDP）。函数会将 `netbuf` 中的数据发送到目标地址，并返回一个错误码，指示发送操作的结果。

`netconn_send(struct netconn *conn, struct netbuf *buf)`: 这个函数用于将数据发送到连接的对端。与 `netconn_sendto` 不同，它不需要指定目标地址，因为目标地址已经在连接建立时确定。函数会将 `netbuf` 中的数据发送到连接的对端，并返回操作结果。

`netconn_write_partly(struct netconn *conn, const void *dataptr, size_t size,`

u8_t apiflags, size_t *bytes_written): 这个函数允许开发者发送部分数据, 提供了更灵活的发送控制。dataptr 是指向要发送数据的指针, size 是数据的大小。apiflags 允许开发者指定发送时的行为 (如是否阻塞), 而 bytes_written 则用于返回实际发送的字节数。这个函数适用于需要**精细控制发送过程**的场景。

netconn_write(conn, dataptr, size, apiflags): 这是一个宏, 简化了 netconn_write_partly 的调用。它不关心已写入的字节数, 适合于简单的发送操作。开发者只需**提供连接**、数据指针、数据大小和 API 标志。

4.1.4 连接关闭

1. `err_t netconn_close(struct netconn *conn);`
2. `err_t netconn_shutdown(struct netconn *conn, u8_t shut_rx, u8_t shut_tx);`

netconn_close(struct netconn *conn): 在完成数据发送和接收后, 开发者通常会调用此函数关闭连接。它会释放与连接相关的资源, 确保不会有内存泄漏。

netconn_shutdown(struct netconn *conn, u8_t shut_rx, u8_t shut_tx): 这个函数允许开发者选择性地关闭连接的接收和发送功能。通过设置 shut_rx 和 shut_tx 参数, 开发者可以控制连接的行为, 适用于需要优雅关闭连接的场景。

4.1.5 删除连接

1. `err_t netconn_delete(struct netconn *conn);`

对于与创建连接, 这是结束一个网络流程的终点, 删除 conn, 结束一切。

4.2 客户端程序编写示例 (netconn)

```
1. void ClientTask(void *thread_param)
2. {
3.     struct netconn *conn;
4.     err_t err;
5.     ip_addr_t server_ip;
6.     const char *message = "I am client";
7.
8.     IP4_ADDR(&server_ip, 192, 168, 1, 112);
9.
10.    while (1) {
11.        // 创建一个新的 TCP 连接
12.        conn = netconn_new(NETCONN_TCP);
13.        while (conn == NULL) {
14.            conn = netconn_new(NETCONN_TCP);
15.            OSTimeDlyHMSM(0, 0, 1, 0); // 延迟 1 秒后重试
16.        }
17.        conn->recv_timeout = 200;
18.
19.        // 连接到服务器
20.        err = netconn_connect(conn, &server_ip, CLIENTPORT);
```

```

21.         if (err != ERR_OK) {
22.             netconn_delete(conn);
23.             OSTimeDlyHMSM(0, 0, 1, 0);
24.             continue;
25.         }
26.
27.         // 每秒向服务器发送数据
28.         while (1) {
29.             err = netconn_write(conn, message, strlen(message), NETCONN_NOCOPY);
30.             if (err != ERR_OK) {
31.                 break;
32.             }
33.             OSTimeDlyHMSM(0, 0, 1, 0); // 延迟 1 秒
34.         }
35.
36.         netconn_close(conn);
37.         netconn_delete(conn);
38.     }
39. }

```

这段代码定义了一个名为 ClientTask 的函数，主要用于实现一个 TCP 客户端的功能。函数的参数 thread_param 通常用于传递线程相关的参数，但在此代码中并未使用。

在函数开始时，首先声明了一些变量，包括一个指向 netconn 结构的指针 conn，用于表示 TCP 连接；一个 err_t 类型的变量 err，用于存储错误代码；一个 ip_addr_t 类型的变量 server_ip，用于存储服务器的 IP 地址；以及一个字符串 message，表示客户端要发送给服务器的消息。

接下来，通过 IP4_ADDR 宏将服务器的 IP 地址设置为 192.168.1.112。这一步确保客户端能够正确地连接到目标服务器。

进入一个无限循环后，客户端首先尝试创建一个新的 TCP 连接。使用 netconn_new(NETCONN_TCP) 函数创建连接，如果创建失败，代码会在内部循环中不断重试，直到成功为止。为了避免过于频繁的重试，使用 OSTimeDlyHMSM 函数延迟 1 秒后再进行下一次尝试。

一旦成功创建连接，设置接收超时为 200 毫秒。接着，客户端尝试连接到指定的服务器 IP 和端口 CLIENTPORT (4566)。如果连接失败，客户端会删除连接并再次延迟 1 秒后重试。

如果连接成功，客户端进入另一个无限循环，每秒向服务器发送一次消息。使用 netconn_write 函数将消息发送到服务器，如果发送失败，循环会中断，客户端将关闭连接并删除 conn 对象。

最后，客户端在发送完消息后会关闭连接并删除 conn，然后重新进入最外层的循环，准备再次尝试连接到服务器。这种设计使得客户端能够持续尝试与服务器建立连接，并定期发送数据，直到手动终止或发生错误。

4.3 服务器端程序编写（netconn）

```
1. void ServerCallbackTask(void *thread_param)
2. {
3.     struct netconn *conn, *newconn;
4.     struct netbuf *buf;
5.     void *data;
6.     u16_t len;
7.     err_t err;
8.     char timeStr[12];
9.
10.    // 创建
11.    conn = netconn_new(NETCONN_TCP);
12.    while (conn == NULL) {
13.        conn = netconn_new(NETCONN_TCP);
14.        OSTimeDlyHMSM(0, 0, 1, 0); // 延迟 1 秒后重试
15.    }
16.
17.    // 绑定
18.    err = netconn_bind(conn, IP_ADDR_ANY, SERVERPORT);
19.    if (err != ERR_OK) {
20.        netconn_delete(conn);
21.        return;
22.    }
23.
24.    // 监听
25.    err = netconn_listen(conn);
26.    if (err != ERR_OK) {
27.        netconn_delete(conn);
28.        return;
29.    }
30.
31.    // 主循环
32.    while (1) {
33.        err = netconn_accept(conn, &newconn);
34.        if (err != ERR_OK) {
35.            continue;
36.        }
37.
38.        // 接受和处理数据
39.        while ((err = netconn_recv(newconn, &buf)) == ERR_OK) {
40.            do {
41.                netbuf_data(buf, &data, &len);
42.                if (len == 7 && strncmp((char *)data, "GET RTC", 7) == 0)
```

```

43.     {
44.         sprintf(timeStr, "%02u:%02u:%02u", time.hour, time.minute, time.second);
45.         err = netconn_write(newconn, timeStr, strlen(timeStr), NETCONN_COPY);
46.     }
47.     else
48.     {
49.         err = netconn_write(newconn, data, len, NETCONN_COPY);
50.     }
51.     if (err != ERR_OK) {
52.         break;
53.     }
54.         } while (netbuf_next(buf) >= 0);
55.
56.         netbuf_delete(buf);
57.     }
58.
59.     netconn_close(newconn);
60.     netconn_delete(newconn);
61. }
62. }

```

首先，服务器通过调用 `netconn_new` 函数创建一个新的 TCP 网络连接。这个过程是一个循环，直到成功创建连接为止。如果连接创建失败，服务器会每隔一秒重试一次。这种设计确保了服务器在启动时能够稳定地建立连接，避免因瞬时错误而导致的启动失败。

一旦成功创建连接，服务器接着调用 `netconn_bind` 函数，将连接绑定到指定的 IP 地址和端口。这里使用 `IP_ADDR_ANY` 表示服务器可以接受来自任何 IP 地址的连接请求。如果绑定失败，服务器会删除连接并退出任务，确保不会留下无效的连接资源。

绑定成功后，服务器调用 `netconn_listen` 函数开始监听连接请求。这使得服务器能够接受来自客户端的连接。若监听失败，服务器同样会删除连接并退出，确保资源的有效管理。

服务器进入主循环，等待客户端的连接请求。通过调用 `netconn_accept`，服务器接受一个新的连接，并将其存储在 `newconn` 中。如果接受连接失败，服务器会继续循环，等待下一个连接请求。

在成功接受连接后，服务器进入一个内部循环，使用 `netconn_recv` 函数接收客户端发送的数据。接收到的数据被存储在 `buf` 中。服务器通过 `netbuf_data` 函数获取数据的指针和长度，并根据数据内容进行处理。如果接收到的数据是特定的请求（例如“GET RTC”），服务器会格式化当前时间并通过 `netconn_write` 函数将时间字符串发送回客户端。如果接收到的是其他数据，服务器则直接将原始数据发送回去。

在处理数据时，服务器使用一个 `do...while` 循环来处理 `buf` 中的所有数据片段。通过 `netbuf_next` 函数，服务器可以遍历所有的数据片段，确保完整地处理接收到的数据。每次发送数据后，服务器会检查发送操作的返回值，以确保数据成功发送。

一旦所有数据处理完成，服务器会调用 `netbuf_delete` 函数释放 `buf` 的内存。接着，服务器关闭与客户端的连接，调用 `netconn_close` 和 `netconn_delete` 函数释放与 `newconn` 相关的资源。这一过程确保了服务器在处理完每个连接后能够正确清理资源，为下一次连接做好准备。

第五章 常见问题与解决方案

5.1 硬件抽象层（HAL）不兼容

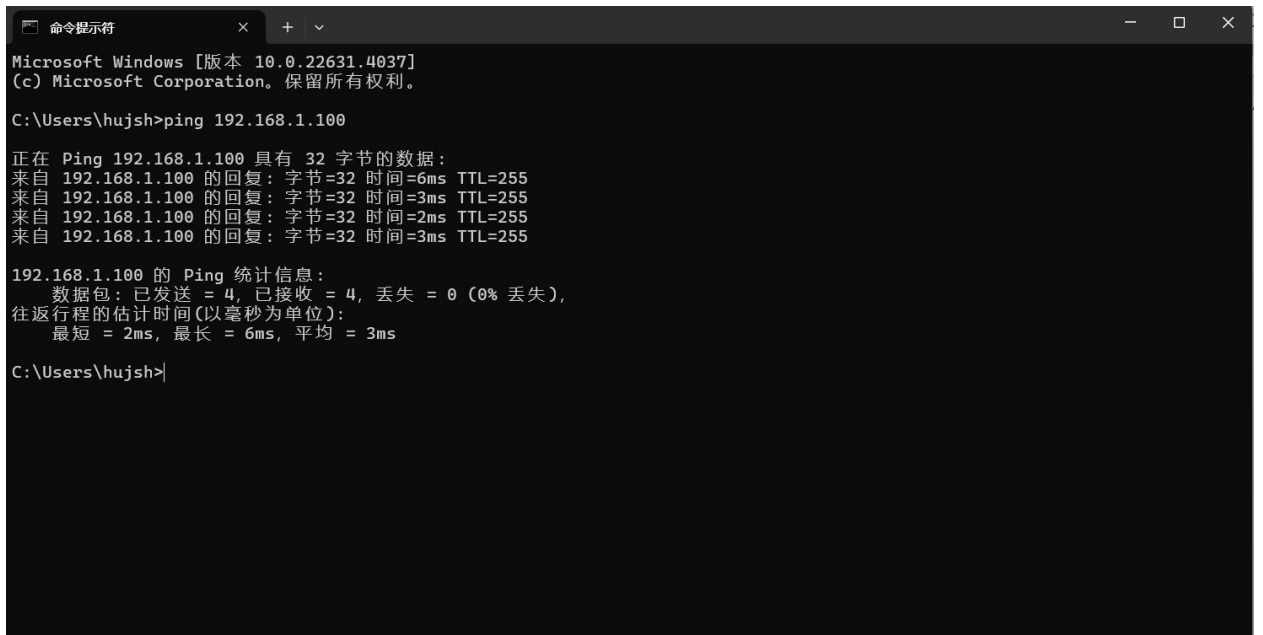
由于 lwIP 依赖于底层硬件的驱动程序来实现网络功能。如果不兼容，会导致网络接口无法正常工作。而能够收集到的 TM4C1294 的 mcu 与 LWIP1.4.1 版本的相关内容极其匮乏。导致只能使用 STM32 的相关内容和自带的 example 库进行摸索学习。

解决办法：深入观察学习底层逻辑，翻阅数据手册，理解前辈代码和相关的底层硬件配置。其中对 tivaif_hwinit() 函数的理解和配置格外重要。这是一个用于初始化 Tiva C 系列微控制器的以太网接口的函数。主要就是启动 GPIOK 外设，并配置 PK5 和 PK6 引脚作为以太网 LED 指示灯。设置网络接口的硬件地址长度和 MAC 地址，重置 PHY 并等待重置完成，配置时间戳功能，以便对接收到的所有数据包进行时间戳处理。在正确的位置放置这个函数的外界接口 tivaif_init() 就很关键。

5.2 无操作系统情况下交互异常

在一开始，学习的思路是从简单的无 os 开始，但真正开始操作时发现无 os 根本没有简单。lwip1.4.1 对无操作系统的 RAW 接口极其不友好。报文数据的处理、注册回调函数等都是需要自己去实现。还会出现异常 crash、内存申请不出来的各种情况。且裸机的中断和正常程序的执行流之间也有竞争关系。

解决办法：考虑到工程代码的实际需要和前辈的建议，果断放弃使用无 OS 情况下的编程，重新构建!NO_SYS 里的代码逻辑. 直到最后 ping 通静态 ip。



```
命令提示符
Microsoft Windows [版本 10.0.22631.4037]
(c) Microsoft Corporation. 保留所有权利。

C:\Users\hujsh>ping 192.168.1.100

正在 Ping 192.168.1.100 具有 32 字节的数据:
来自 192.168.1.100 的回复: 字节=32 时间=6ms TTL=255
来自 192.168.1.100 的回复: 字节=32 时间=3ms TTL=255
来自 192.168.1.100 的回复: 字节=32 时间=2ms TTL=255
来自 192.168.1.100 的回复: 字节=32 时间=3ms TTL=255

192.168.1.100 的 Ping 统计信息:
    数据包: 已发送 = 4, 已接收 = 4, 丢失 = 0 (0% 丢失),
    往返行程的估计时间(以毫秒为单位):
        最短 = 2ms, 最长 = 6ms, 平均 = 3ms

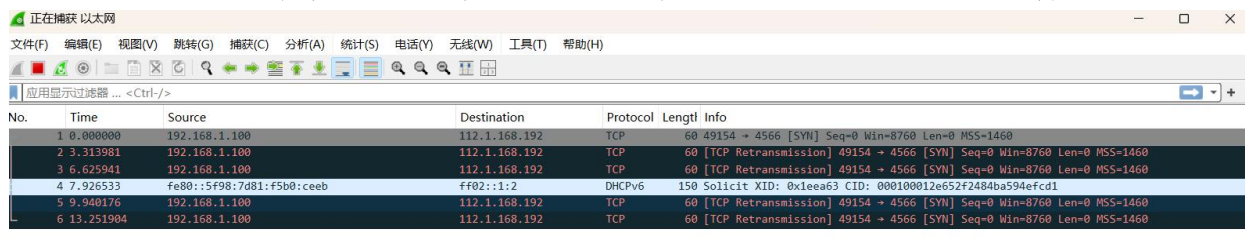
C:\Users\hujsh>
```

5.3 客户端编程上位机回应 TCP 第二次握手问题

这其实是个非常简单的问题，但是由于思路的错误花费了接近一天的时间。当写完客户端通信代码后，尝试与上位机联系。发现串口助手无法接受到正常的收发信息。

解决方案：

1. 抓包查看 tcp 握手情况，可以看到在下位机发送第一条信息后，上位机迟迟没有回应。当时就把我的思路带到了上位机有问题的死胡同里了。于是进行了以下操作。



No.	Time	Source	Destination	Protocol	Length	Info
1	0.000000	192.168.1.100	112.1.168.192	TCP	60	49154 → 4566 [SYN] Seq=0 Win=8760 Len=0 MSS=1460
2	3.313981	192.168.1.100	112.1.168.192	TCP	60	[TCP Retransmission] 49154 → 4566 [SYN] Seq=0 Win=8760 Len=0 MSS=1460
3	6.625941	192.168.1.100	112.1.168.192	TCP	60	[TCP Retransmission] 49154 → 4566 [SYN] Seq=0 Win=8760 Len=0 MSS=1460
4	7.926533	fe80::5f98:7d81:f5b0:ceeb	ff02::1:2	DHCPv6	150	Solicit XID: 0x1ee63 CID: 000100012e652f2484ba594efcd1
5	9.940176	192.168.1.100	112.1.168.192	TCP	60	[TCP Retransmission] 49154 → 4566 [SYN] Seq=0 Win=8760 Len=0 MSS=1460
6	13.251904	192.168.1.100	112.1.168.192	TCP	60	[TCP Retransmission] 49154 → 4566 [SYN] Seq=0 Win=8760 Len=0 MSS=1460

2. 打开串口助手权限和端口权限（连接失败）

3. 关闭防火墙（连接失败）

4. 修改网关信息（连接失败）

5. 重新配置上位机网络（连接失败）

最后，在隔天重写审视我写的代码后，发现了这么几行

```
1. static NETWORK_IPADDR ServerIpAddr = {192, 168, 1, 112};
2.     server_ip.addr = (((uint32_t)ServerIpAddr.Addr0) << 24) |
3.                         (((uint32_t)ServerIpAddr.Addr1) << 16) |
4.                         (((uint32_t)ServerIpAddr.Addr2) << 8) |
5.                         ((uint32_t)ServerIpAddr.Addr3);
```

我记得在偶然的情况下看到过 lwip 的内置 ip 地址编码如下

```
1. /** Set an IP address given by the four byte-parts.
2.     Little-endian version that prevents the use of htonl. */
3. #define IP4_ADDR(ipaddr, a,b,c,d) \
4.     (ipaddr)->addr = ((u32_t)((d) & 0xff) << 24) | \
5.                     ((u32_t)((c) & 0xff) << 16) | \
6.                     ((u32_t)((b) & 0xff) << 8) | \
7.                     (u32_t)((a) & 0xff)
```

终于破案了，本质上是大小端的表达方式的问题，导致了下位机其实一直在呼叫 112.1.168.192 这个 ip。