网络协议栈驱动移植笔记

RT-THREAD 文档中心

WWW.RT-THREAD.ORG

Friday 28th September, 2018

目录

!!! abstract “摘要” 本应用笔记描述了如何在 RT-Thread 中，如何根据具体的硬件配置网络驱动，并灵活运用调试手段解决问题。

1 本文的目的和结构

1.1 本文的目的和背景

在 RT-Thread 所支持的 BSP 中，大部分都有支持以太网驱动。但具体到用户的硬件中，可能会和默认的代码有所差异。本文选择相对以太网驱动比较完善 stm32f40x，介绍了驱动的主要实现方式，以及针对不同硬件的修改方法。

1.2 本文的结构

本文首先介绍了常见的以太网芯片种类和一些常用的名词解释，然后详细描述了 RT-Thread 以太网驱动架构，驱动接口和驱动的移植，并给出了在正点原子 STM32F4 探索者开发板上移植的代码示例，最后介绍了驱动的调试方法。

2 常见的以太网芯片种类

以太网芯片有很多种，大致可以分成 3 种：

- 以太网芯片只有 PHY（物理接口收发器），需要单片机带 MAC（以太网媒体接入控制器），通过 MII 或者 RMII 接口和单片机通讯。例如 LAN8720。
- 以太网芯片带 MAC 和 PHY，通过 SPI 接口和单片机通讯。例如 ENC28J60。
- 以太网芯片带 MAC 和 PHY，通过 SPI 接口和单片机通讯，同时内置硬件协议栈，适合低速单片机。例如 W5500。

3 常见名词解释

MAC：媒体介入控制层，属于 OSI 模型中数据链路层下层子层。

PHY：PHY 指物理层，OSI 的最底层。一般指与外部信号接口的芯片。

MII：MII (Media Independent Interface)，介质无关接口，也被称为媒体独立接口，它是 IEEE-802.3 定义的以太网行业标准，支持 10Mbit/s 和 100Mbit/s 数据传输模式。

RMII：RMII (Reduced Media Independent Interface) ，简化媒体独立接口，是 IEEE 802.3u 标准中除 MII 接口之外的另一种实现，支持 10Mbit/s 和 100Mbit/s 数据传输模式。相比 MII，精简了引脚数量。

lwIP：lwIP 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。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。

pbuf：lwIP 中用来管理数据包的结构体。

4 驱动架构图

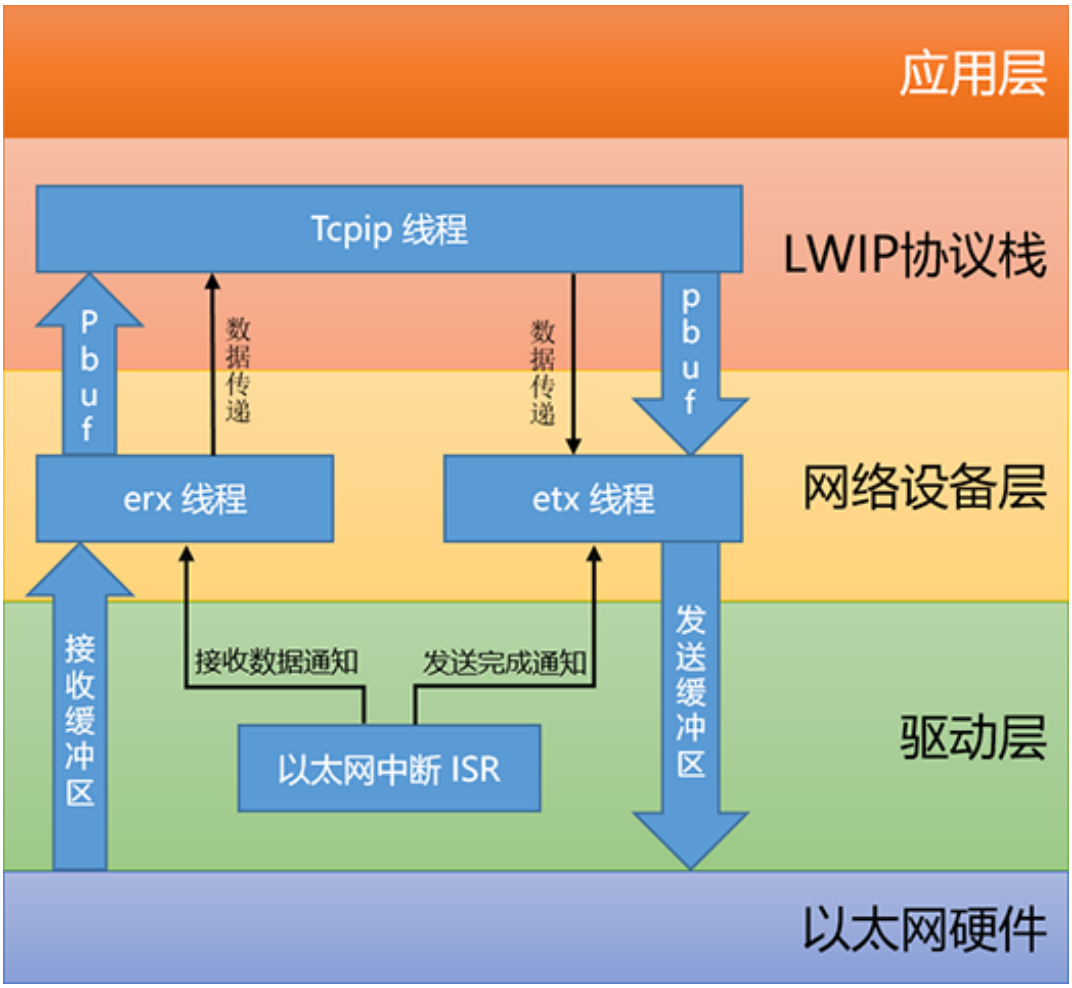

图 1: *lwip_block*

RT-Thread 的 lwIP 移植在原版的基础上，添加了网络设备层以替换原来的驱动层。和原来的驱动层不同的是，对于以太网数据的收发采用了独立的双线程结构，erx 线程和 etx 线程在正常情况下，两者的优先级设置成相同，用户可以根据自身实际要求进行微调以侧重接收或发送。

4.1 数据接收流程

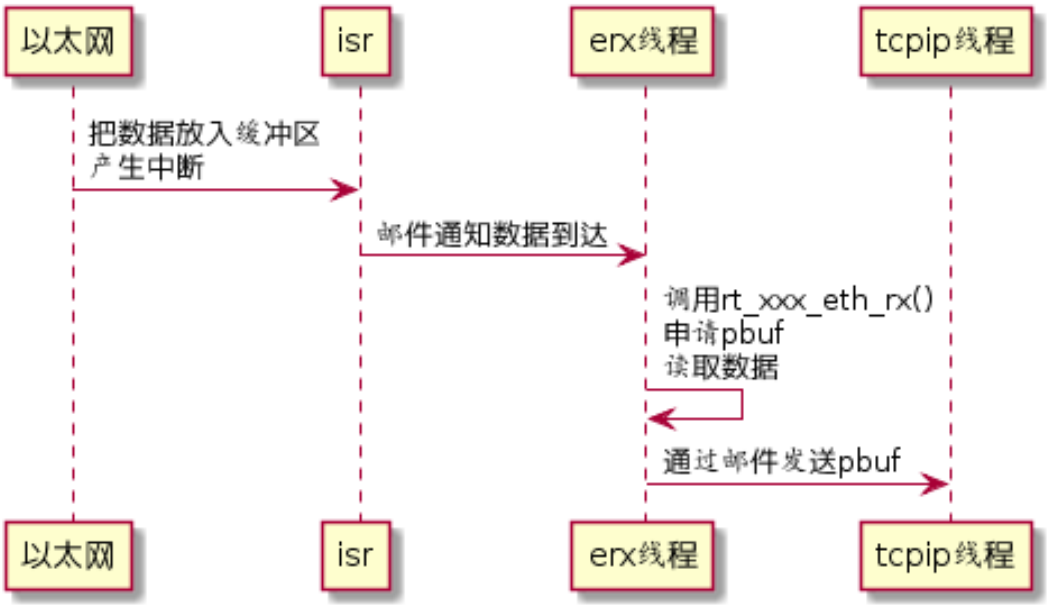

图 2: *rt_xxx_eth_rx*

当以太网硬件设备收到网络报文产生中断时，接收到的数据会被存放到接收缓冲区，然后以太网中断程序会发送邮件来唤醒 erx 线程，erx 线程会按照接收到的数据长度来申请 pbuf，并将数据放入 pbuf 的 payload 中，然后将 pbuf 通过邮件发送给去处理。

4.2 数据发送流程

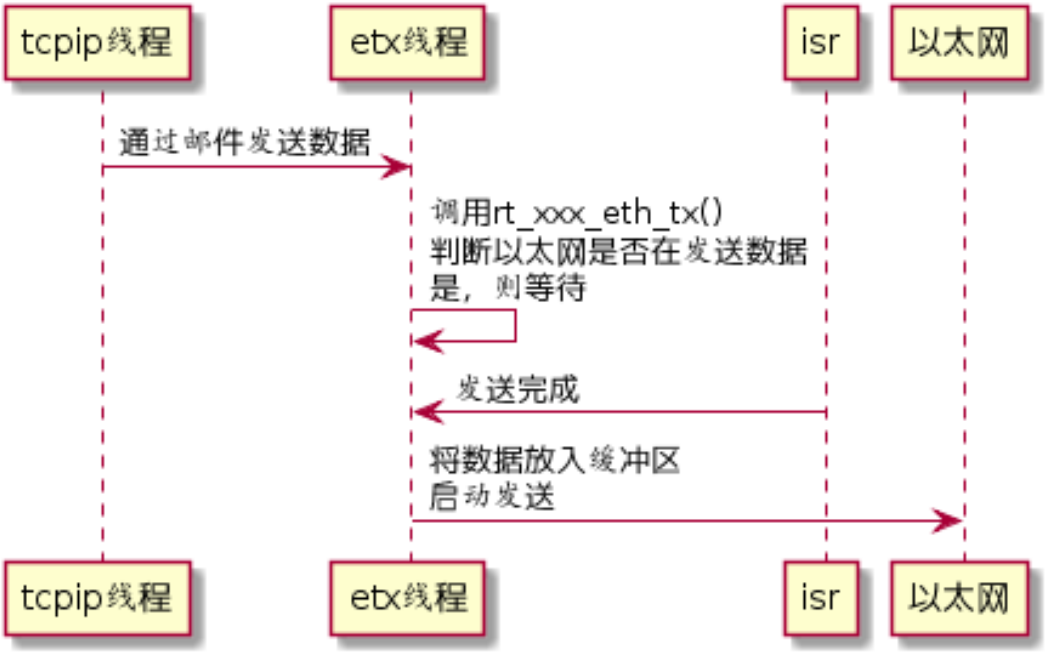

图 3: *rt_xxx_erh_tx*

当有数据需要发送时，tcpip 线程会发送一个邮件来唤醒 etx 线程，etx 线程会先判断现在是否正在发送数据，如果没在发送数据，etx 线程会将要发送的数据放入发送缓冲区，然后启动以太网发送来发送数据。如果正在发送数据，etx 线程会将自己挂起，直到获取到发送等待信号量后再发送数据。

> 注意：在一定条件下也可以把 RT-Thread 中加入的 etx/erx 任务移除掉，当移除掉 RX_THREAD 时，需要由其他线程或中断把接收到的 pbuf 数据包提交给 lwIP 主任务，这里不做详细介绍。

5 网络设备介绍

RT-Thread 网络设备继承了标准设备，是由 eth_device 结构体定义的，这里贴出 eth_device 结构体代码提供参考

```
struct eth_device
{
    /* 标准设备 */
    struct rt_device parent;

    /* lwIP 网络接口 */
```

```
    struct netif *netif;
    /* 发送应答信号量 */
    struct rt_semaphore tx_ack;

    /* 网络状态标志 */
    rt_uint16_t flags;
    rt_uint8_t  link_changed;
    rt_uint8_t  link_status;

    /* 数据包收发接口 */
    struct pbuf* (*eth_rx)(rt_device_t dev);
    rt_err_t (*eth_tx)(rt_device_t dev, struct pbuf* p);
};
```

本文以太网驱动比较完善 stm32f40x 为例进行讲解，除 ST 固件库以外，需要实现以下驱动：

5.1 标准设备接口

标准设备接口需要提供给 eth_device 结构体中的 parent 元素

```
static rt_err_t rt_stm32_eth_init(rt_device_t dev);
static rt_err_t rt_stm32_eth_open(rt_device_t dev, rt_uint16_t oflag);
static rt_err_t rt_stm32_eth_close(rt_device_t dev);
static rt_size_t rt_stm32_eth_read(rt_device_t dev, rt_off_t pos, void*
    buffer, rt_size_t size);
static rt_size_t rt_stm32_eth_write (rt_device_t dev, rt_off_t pos, const
     void* buffer, rt_size_t size);
static rt_err_t rt_stm32_eth_control(rt_device_t dev, int cmd, void *args
    );
```

rt_stm32_eth_init 用于初始化 DMA 和 MAC 控制器。

rt_stm32_eth_open 用于上层应用打开网络设备，目前未使用到，直接返回 RT_EOK。

rt_stm32_eth_close 用于上层应用关闭网络设备，目前未使用到，直接返回 RT_EOK。

rt_stm32_eth_read 用于上层应用向底层设备进行直接读写的情况，对于网络设备，每个报文都有固定的格式，所以这个接口目前并未使用，直接返回 0 值。

rt_stm32_eth_write 用于上层应用向底层设备进行直接读写的情况，对于网络设备，每个报文都有固定的格式，所以这个接口目前并未使用，直接返回 0 值。

rt_stm32_eth_control 用于控制以太网接口设备，目前用于获取以太网接口的 mac 地址。如果需要，也可以通过增加控制字的方式来扩展其他控制功能。

5.2 数据包收发接口

对应了 eth_device 结构体中的 eth_rx 及 eth_tx 元素，实现数据包收发功能，如下所示：

```
rt_err_t rt_stm32_eth_tx( rt_device_t dev, struct pbuf* p);
struct pbuf *rt_stm32_eth_rx(rt_device_t dev);
```

rt_stm32_eth_tx 函数被 etx 线程调用，实现了数据发送的功能，这里贴出 stm32f40x 部分代码片段提供参考

```
rt_err_t rt_stm32_eth_tx( rt_device_t dev, struct pbuf* p)
{
    ......

    /* 判断以太网是否正在发送数据 */
    while ((DMATxDescToSet->Status & ETH_DMATxDesc_OWN) != (uint32_t)
        RESET)
    {
        rt_err_t result;
        rt_uint32_t level;

        /* 进入发送等待状态 */
        level = rt_hw_interrupt_disable();
        tx_is_waiting = RT_TRUE;
        rt_hw_interrupt_enable(level);

        /* 等待获取发送完成信号量，该信号量会在中断中释放 */
        result = rt_sem_take(&tx_wait, RT_WAITING_FOREVER);
        if (result == RT_EOK) break;
        if (result == -RT_ERROR) return -RT_ERROR;
    }

    /* 将pbuf中数据放入发送缓冲区 */
    offset = 0;
    for (q = p; q != NULL; q = q->next)
    {
        uint8_t *to;

        to = (uint8_t*)((DMATxDescToSet->Buffer1Addr) + offset);
        memcpy(to, q->payload, q->len);
        offset += q->len;
    }
```

```
    /* 启动发送 */
    ......

}
```

rt_stm32_eth_rx 函数被 erx 线程调用，实现了接收数据的功能，这里贴出 stm32f40x 部分代码片段提供参考

```
struct pbuf *rt_stm32_eth_rx(rt_device_t dev)
{
    ......

    /* 得到帧长度 */
    framelength = ((DMARxDescToGet->Status & ETH_DMARxDesc_FL) >>
        ETH_DMARXDESC_FRAME_LENGTHSHIFT) - 4;

    /* 申请pbuf */
    p = pbuf_alloc(PBUF_LINK, framelength, PBUF_RAM);
    if (p != RT_NULL)
    {
        struct pbuf* q;

        /* 将接收到的数据拷贝到pbuf中 */
         for (q = p; q != RT_NULL; q= q->next)
         {
             memcpy(q->payload, (uint8_t *)((DMARxDescToGet->Buffer1Addr)
                 + offset), q->len);
             offset += q->len;
         }
    }

    ......
    /* 返回pbuf指针 */
    return p;
}
```

5.3 驱动初始化入口

```
void rt_hw_stm32_eth_init(void);
```

rt_hw_stm32_eth_init 用于注册以太网设备，以太网硬件，配置 MAC 地址等。

6 移植准备

- 下载RT-Thread 源码
- 下载 env 工具
- 打开 env，进入 `rt-thread/bsp/stm32f40x` 目录
- 在 env 命令行中输入 `set RTT_CC=keil` ，设置工具链类型为 keil
- 在 env 命令行中输入 `menuconfig`，进入配置界面，使用 menuconfig 工具配置工程。
  - 修改控制台输出为自己板子引出的串口号
  - 启用 lwIP

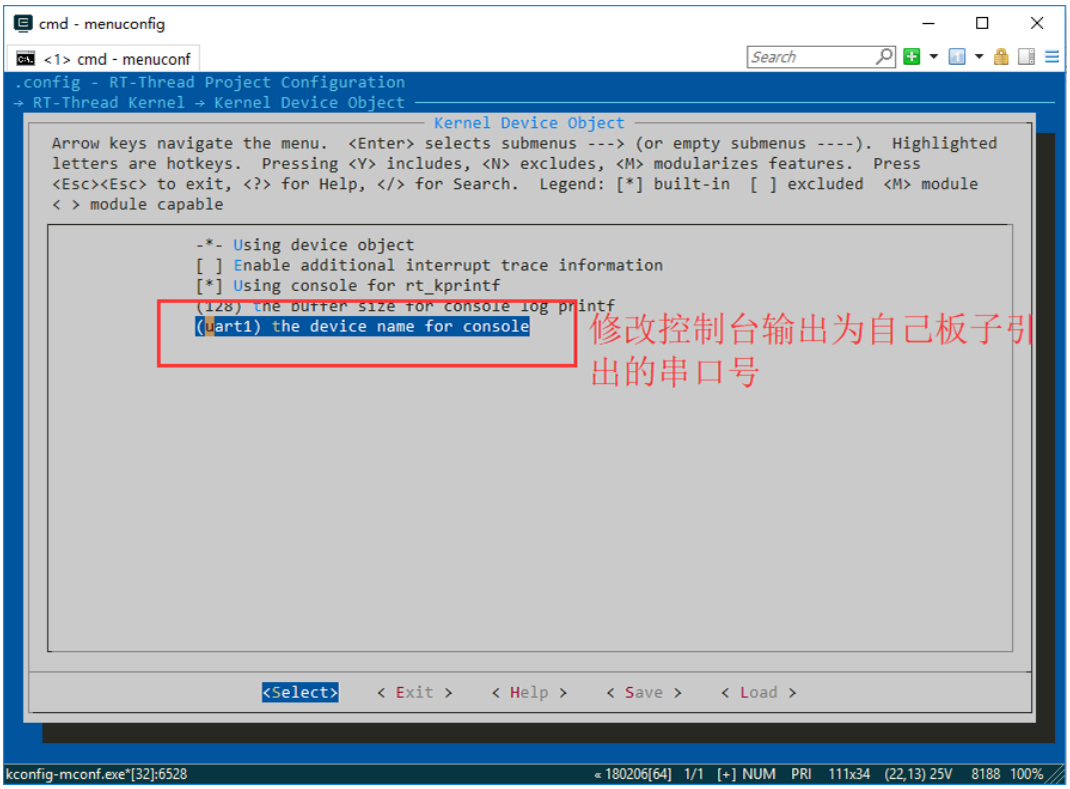

图 4: *ENV_uart*

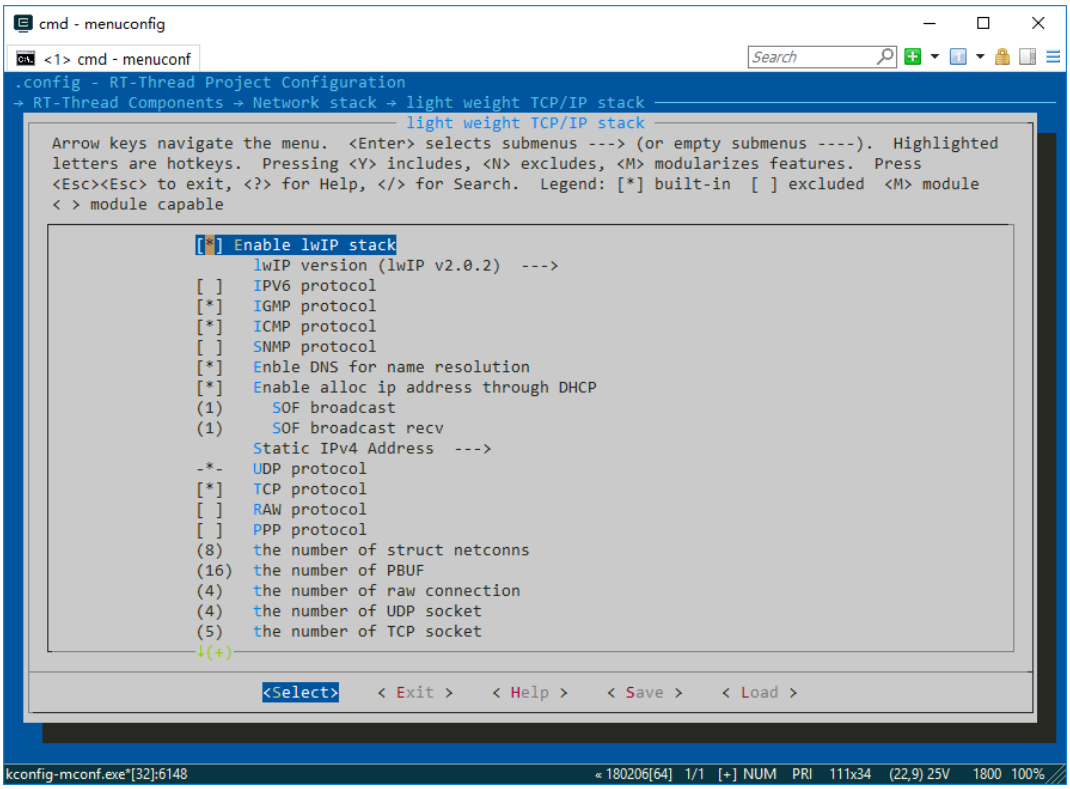

图 5: *ENV_lwIP*

- 输入命令 scons –target=mdk5 -s 生成 mdk5 工程。
- 打开工程，打开 stm32f4xx_eth.c 文件。

7 驱动移植

STM32F407 芯片自带以太网模块，该模块包括带专用 DMA 控制器的 MAC 802.3（介质访问控制）控制器，支持介质独立接口 (MII) 和简化介质独立接口 (RMII)，并自带了一个用于外部 PHY 通信的 SMI 接口，通过一组配置寄存器，用户可以为 MAC 控制器和 DMA 控制器选择所需模式和功能。

因为同系列 stm32 的 MAC 控制器初始化基本一样，所以同系列的驱动 MAC 部分的代码不需要做修改，要修改的只是 PHY 的部分，这里以 LAN8720 为例讲解。

7.1 数据包打印

在调试驱动时，建议先打开 stm32f4xx_eth.c 中的日志功能。

```
/* debug 设置 */
#define ETH_DEBUG
#define ETH_RX_DUMP
```

```
#define ETH_TX_DUMP
```

7.2 更新 PHY 复位引脚

查看原理图，RESET 引脚为 PD3，修改复位管脚为 PD3。

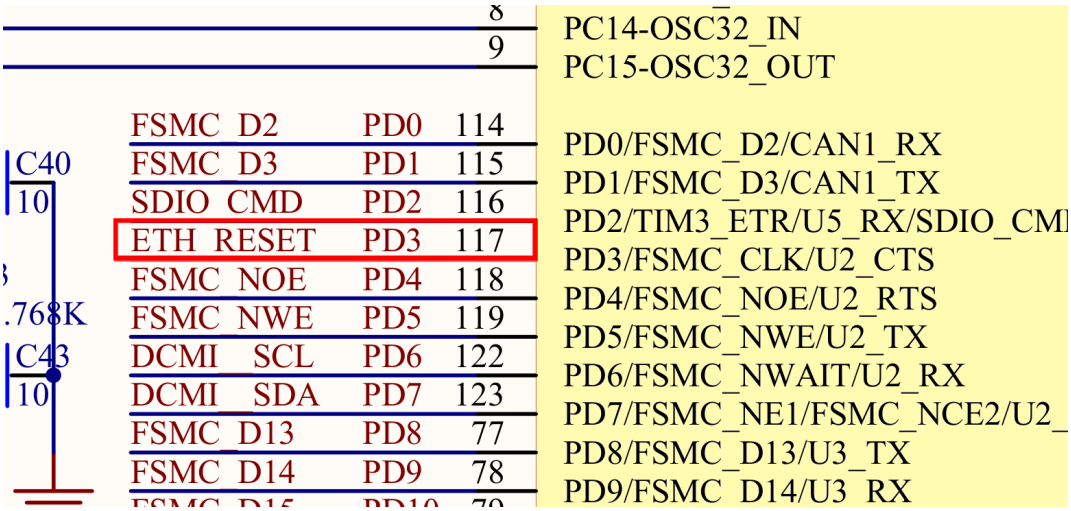

图 6: *reset_pin*

```
void rt_hw_stm32_eth_init(void)
{
    {
        GPIO_InitTypeDef GPIO_InitStructure;

        GPIO_InitStructure.GPIO_Mode  = GPIO_Mode_OUT;
        GPIO_InitStructure.GPIO_Speed = GPIO_Speed_2MHz;
        GPIO_InitStructure.GPIO_OType = GPIO_OType_PP;
        GPIO_InitStructure.GPIO_PuPd  = GPIO_PuPd_NOPULL;

        RCC_AHB1PeriphClockCmd(RCC_AHB1Periph_GPIOD, ENABLE);

        GPIO_InitStructure.GPIO_Pin = GPIO_Pin_3;
        GPIO_Init(GPIOD, &GPIO_InitStructure);

        GPIO_ResetBits(GPIOD, GPIO_Pin_3);
        rt_thread_delay(2);
        GPIO_SetBits(GPIOD, GPIO_Pin_3);
        rt_thread_delay(2);
    }
}
```

在rt_hw_stm32_eth_init函数中，使用 GPIO 对 PHY 芯片进行了复位。如果不修改为正确的引脚，除了 PHY 可能没有被正确复位外，还可能造成引脚冲突而损坏板子。

7.3 确认 MII/RMII 模式

根据原理图，确认外接的 PHY 是使用 MII 还是 RMII 模式。

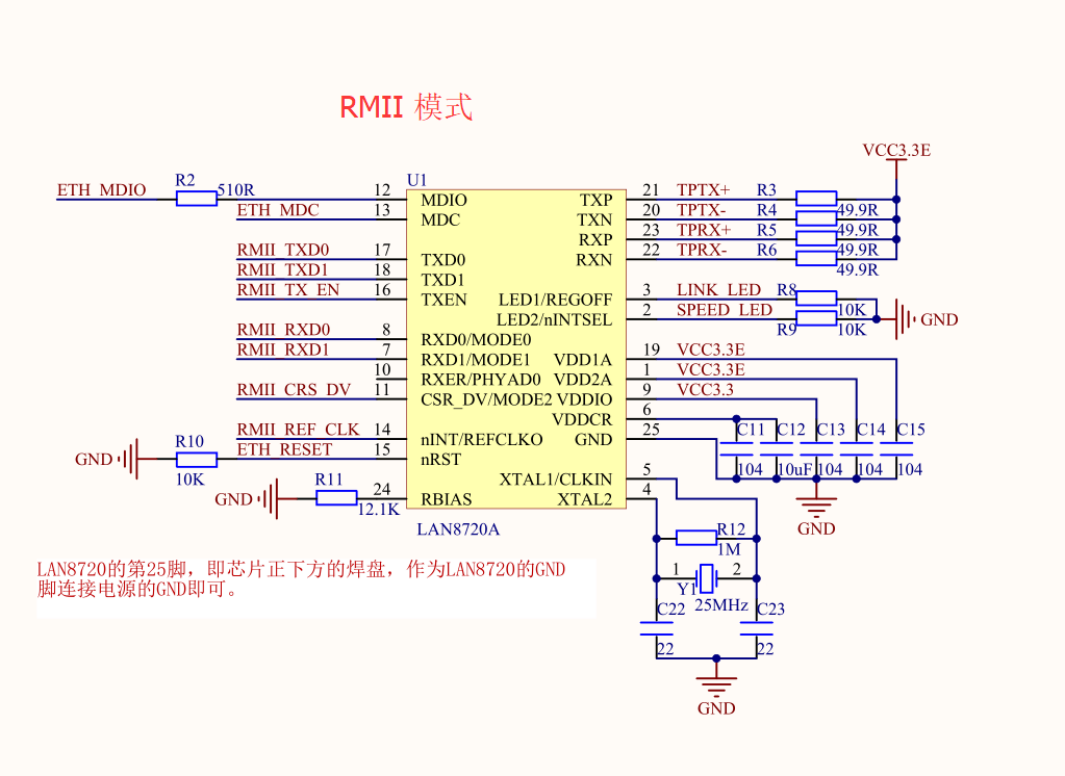

图 7: *LAN8720*

修改 stm32f4xx_eth.c 中的模式设置

```
#define RMII_MODE                       /* MII_MODE or RMII_MODE */
```

驱动默认使用 IO 比较简洁的 RMII 模式，当使用 MII 模式时需要修改为

```
#define MII_MODE                        /* MII_MODE or RMII_MODE */
```

7.4 确认引脚映射

不少单片机为了布线方便，都有引脚映射功能，需要根据原理图来确定和 PHY 相连的引脚。

STM32F4 的以太网，可以通过 IO 映射选择不同的 IO，默认是按组选择的，正点原子探索者用的是 GPIOG, 不需要修改代码。

```
#define RMII_TX_GPIO_GROUP          2       /* 1:GPIOB or 2:GPIOG */
```

实际上 F4 的每个 IO 都可以独立设置，如果 TX 所有的 IO 不在同一个 GPIO PORT 上面，如 TX1 在 GPIOB 上面，而 TX0 在 GPIOG 上面。这种情况则需要大篇幅修改代码，确保每个 IO 都映射到对应的端口。

除了数据 IO 外，部分引脚多的芯片，MDIO 接口也可以映射到不同的端口，这部分配置也需要一一核对。

7.5 引脚初始化

下面列出正点原子探索者用于连接外部 PHY 的引脚, 用户需要按照自己的原理图对相应的管脚进行初始化。

```
    /*
     ETH_MDIO -------------> PA2
     ETH_MDC --------------> PC1

     ETH_RMII_REF_CLK ----> PA1

     ETH_RMII_CRS_DV -----> PA7
     ETH_RMII_RXD0 -------> PC4
     ETH_RMII_RXD1 -------> PC5

     ETH_RMII_TX_EN ------> PG11
     ETH_RMII_TXD0 -------> PG13
     ETH_RMII_TXD1 -------> PG14

     ETH_RMII_TX_EN ------> PB11
     ETH_RMII_TXD0 -------> PB12
     ETH_RMII_TXD1 -------> PB13
     */

    GPIO_PinAFConfig(GPIOA, GPIO_PinSource1, GPIO_AF_ETH); /*
        RMII_REF_CLK */
    GPIO_PinAFConfig(GPIOA, GPIO_PinSource7, GPIO_AF_ETH); /* RMII_CRS_DV
         */

    GPIO_InitStructure.GPIO_Pin = GPIO_Pin_1 | GPIO_Pin_7;
    GPIO_Init(GPIOA, &GPIO_InitStructure);
```

```
    GPIO_PinAFConfig(GPIOC, GPIO_PinSource4, GPIO_AF_ETH); /* RMII_RXD0
        */
    GPIO_PinAFConfig(GPIOC, GPIO_PinSource5, GPIO_AF_ETH); /* RMII_RXD1
        */

    GPIO_InitStructure.GPIO_Pin = GPIO_Pin_4 | GPIO_Pin_5;
    GPIO_Init(GPIOC, &GPIO_InitStructure);

    RCC_AHB1PeriphClockCmd(RCC_AHB1Periph_GPIOG, ENABLE);

    GPIO_PinAFConfig(GPIOG, GPIO_PinSource11, GPIO_AF_ETH); /* RMII_TX_EN
         */
    GPIO_PinAFConfig(GPIOG, GPIO_PinSource13, GPIO_AF_ETH); /* RMII_TXD0
        */
    GPIO_PinAFConfig(GPIOG, GPIO_PinSource14, GPIO_AF_ETH); /* RMII_TXD1
        */

    GPIO_InitStructure.GPIO_Pin =  GPIO_Pin_11 | GPIO_Pin_13 |
        GPIO_Pin_14;
    GPIO_Init(GPIOG, &GPIO_InitStructure);
```

7.6 更新 PHY 管理程序

PHY 是 IEEE802.3 中定义的一个标准模块。PHY 寄存器的地址空间为 5 位，因此寄存器范围是 0 到 31 ，最多有 32 个寄存器。IEEE802.3 定义了地址为 0-15 这 16 个基础寄存器的功能，因此只需要修改少量寄存器的定义即可完成移植。

驱动默认使用 LAN8720，如果使用的是其它型号的 PHY。应该修改自动协商、状态和速率指示寄存器相关的宏定义，使程序能正确设置自动协商并读取 PHY 的连接状态和速率。

下面列出需要在 stm32f4xx_eth.h 中配置 LAN8720 常用的几个寄存器和控制状态位的定义

```
#define PHY_BCR                          0          /* 基础控制寄存器 */

/* 基础控制寄存器的控制位定义 */
#define PHY_Reset                       ((uint16_t)0x8000)      /* PHY 软
    件复位 */
#define PHY_Loopback                    ((uint16_t)0x4000)
#define PHY_FULLDUPLEX_100M             ((uint16_t)0x2100)
#define PHY_HALFDUPLEX_100M             ((uint16_t)0x2000)
#define PHY_FULLDUPLEX_10M              ((uint16_t)0x0100)
#define PHY_HALFDUPLEX_10M              ((uint16_t)0x0000)
```

```
#define PHY_AutoNegotiation             ((uint16_t)0x1000)      /* 使能自
    动协商 */
#define PHY_Restart_AutoNegotiation     ((uint16_t)0x0200)      /* 重新启
    动自动协商 */
#define PHY_Powerdown                   ((uint16_t)0x0800)
#define PHY_Isolate                     ((uint16_t)0x0400)

#define PHY_BSR                          1          /* 基础状态寄存器*/

/* 基础状态寄存器的状态位定义 */
#define PHY_AutoNego_Complete           ((uint16_t)0x0020)      /* 自动协
    商完成 */
#define PHY_Linked_Status               ((uint16_t)0x0004)      /* 连接状
    态 */
#define PHY_Jabber_detection            ((uint16_t)0x0002)      /* jabber
     检测指示位 */
```

RT-Thread 的驱动实现中，做了 PHY 地址搜索的功能，可以正确搜索出 PHY 的地址，所以不必定义 PHY 地址

这里贴出 stm32f4xx_eth.c 文件中 `phy_monitor_thread_entry`函数中的地址搜索函数供用户参考

```
    /* 通过能否读取 PHY 芯片 ID 号来判断地址*/
    {
        rt_uint32_t i;
        rt_uint16_t temp;

        for(i=0; i<=0x1F; i++)
        {
            temp = ETH_ReadPHYRegister(i, 0x02);

            if( temp != 0xFFFF )
            {
                phy_addr = i;
                break;
            }
        }
    }

    /* 未搜索到PHY地址 */
    if(phy_addr == 0xFF)
    {
        STM32_ETH_PRINTF("phy not probe!\r\n");
        return;
```

```
}
```

如果提示phy not probe!则应该检查 IO 配置和硬件。

小提示：现在主流的 PHY 一般在复位后都默认工作在自动协商模式下，且现在的交换机和网线，一般都可以支持支持 100M 全双工。所以在未正确适配 PHY 以前，也可以临时修改 PHY 的工作模式为 100M 全双工供测试用（修改基础控制寄存器中相应的控制位）。

7.7 中断函数

接收：中断函数在接收到数据时会发邮件来通知 “erx” 线程来读取数据。

发送：在发送完毕后中断函数会释放信号量来告诉发送程序可以继续发送了。

这里只贴出发送和接收相关的代码

```
status = ETH->DMASR;
ier = ETH->DMAIER;

if(status & ETH_DMA_IT_NIS)
{
    rt_uint32_t nis_clear = ETH_DMA_IT_NIS;

    /* 发送中断 */
    if((status & ier) & ETH_DMA_IT_T)
    {
        STM32_ETH_PRINTF("ETH_DMA_IT_T\r\n");

       /* 发送已经完成，释放发送等待信号量 */
        if (tx_is_waiting == RT_TRUE)
        {
            tx_is_waiting = RT_FALSE;
            rt_sem_release(&tx_wait);
        }

        nis_clear |= ETH_DMA_IT_T;
    }

    /* 接收中断 */
    if((status & ier) & ETH_DMA_IT_R) /* packet reception */
    {
        STM32_ETH_PRINTF("ETH_DMA_IT_R\r\n");
        /* 发送邮件通知erx线程来接收数据 */
        eth_device_ready(&(stm32_eth_device.parent));
```

```
            nis_clear |= ETH_DMA_IT_R;
        }

        /* 清除中断标志位 */
        ETH_DMAClearITPendingBit(nis_clear);
    }
```

7.8 ETH 设备初始化

RT-Thread 实时操作系统提供了一套设备管理框架，应用程序通过 RT-Thread 的设备操作接口实现通用的设备驱动。我们这里对 ETH 设备，实现 Network Interface 类型的设备驱动，然后注册到 RT-Thread。

stm32f4xx_eth.c 中的 rt_hw_stm32_eth_init 是 ETH 设备初始化入口，负责 stm32_eth_device 结构体的初始化，并将其注册到 RT-Thread。

```
    /* 设置工作速度和模式 */
    stm32_eth_device.ETH_Speed = ETH_Speed_100M;
    stm32_eth_device.ETH_Mode  = ETH_Mode_FullDuplex;

    /* 利用STM32全球唯一ID设置MAC地址 */
    stm32_eth_device.dev_addr[0] = 0x00;
    stm32_eth_device.dev_addr[1] = 0x80;
    stm32_eth_device.dev_addr[2] = 0xE1;
    stm32_eth_device.dev_addr[3] = *(rt_uint8_t*)(0x1FFF7A10+4);
    stm32_eth_device.dev_addr[4] = *(rt_uint8_t*)(0x1FFF7A10+2);
    stm32_eth_device.dev_addr[5] = *(rt_uint8_t*)(0x1FFF7A10+0);

    /* 设置标准驱动接口 */
    stm32_eth_device.parent.parent.init       = rt_stm32_eth_init;
    stm32_eth_device.parent.parent.open       = rt_stm32_eth_open;
    stm32_eth_device.parent.parent.close      = rt_stm32_eth_close;
    stm32_eth_device.parent.parent.read       = rt_stm32_eth_read;
    stm32_eth_device.parent.parent.write      = rt_stm32_eth_write;
    stm32_eth_device.parent.parent.control    = rt_stm32_eth_control;
    stm32_eth_device.parent.parent.user_data  = RT_NULL;

    /* 设置网络驱动接收和发送接口 */
    stm32_eth_device.parent.eth_rx     = rt_stm32_eth_rx;
    stm32_eth_device.parent.eth_tx     = rt_stm32_eth_tx;

    /* 初始化发送信号量 */
    rt_sem_init(&tx_wait, "tx_wait", 0, RT_IPC_FLAG_FIFO);
```

```
    /* 注册网络设备 */
    eth_device_init(&(stm32_eth_device.parent), "e0");
```

7.8.1. 设置标准驱动接口

7.8.1.1. rt_stm32_eth_init rt_stm32_eth_init是初始化以太网外设的，应按照实际需求初始化

这里只贴出 MAC 配置和 DMA 配置的代码

```
    /*------------------------   MAC
        -----------------------------------*/
    ETH_InitStructure.ETH_AutoNegotiation = ETH_AutoNegotiation_Enable;
    ETH_InitStructure.ETH_Speed = stm32_eth->ETH_Speed;
    ETH_InitStructure.ETH_Mode  = stm32_eth->ETH_Mode;
    ETH_InitStructure.ETH_LoopbackMode = ETH_LoopbackMode_Disable;
    ETH_InitStructure.ETH_RetryTransmission =
        ETH_RetryTransmission_Disable;
    ETH_InitStructure.ETH_AutomaticPadCRCStrip =
        ETH_AutomaticPadCRCStrip_Disable;
    ETH_InitStructure.ETH_ReceiveAll = ETH_ReceiveAll_Disable;
    ETH_InitStructure.ETH_BroadcastFramesReception =
        ETH_BroadcastFramesReception_Enable;
    ETH_InitStructure.ETH_PromiscuousMode = ETH_PromiscuousMode_Disable;
    ETH_InitStructure.ETH_MulticastFramesFilter =
        ETH_MulticastFramesFilter_HashTable;
    ETH_InitStructure.ETH_HashTableHigh = stm32_eth->ETH_HashTableHigh;
    ETH_InitStructure.ETH_HashTableLow  = stm32_eth->ETH_HashTableLow;
    ETH_InitStructure.ETH_UnicastFramesFilter =
        ETH_UnicastFramesFilter_Perfect;
#ifdef CHECKSUM_BY_HARDWARE
    ETH_InitStructure.ETH_ChecksumOffload = ETH_ChecksumOffload_Enable;
#endif

    /*------------------------   DMA
        -----------------------------------*/
    ETH_InitStructure.ETH_DropTCPIPChecksumErrorFrame =
        ETH_DropTCPIPChecksumErrorFrame_Enable;
    ETH_InitStructure.ETH_ReceiveStoreForward =
        ETH_ReceiveStoreForward_Enable;
    ETH_InitStructure.ETH_TransmitStoreForward =
        ETH_TransmitStoreForward_Enable;
```

```
    ETH_InitStructure.ETH_ForwardErrorFrames =
        ETH_ForwardErrorFrames_Disable;
    ETH_InitStructure.ETH_ForwardUndersizedGoodFrames =
        ETH_ForwardUndersizedGoodFrames_Disable;
    ETH_InitStructure.ETH_SecondFrameOperate =
        ETH_SecondFrameOperate_Enable;
    ETH_InitStructure.ETH_AddressAlignedBeats =
        ETH_AddressAlignedBeats_Enable;
    ETH_InitStructure.ETH_FixedBurst = ETH_FixedBurst_Enable;
    ETH_InitStructure.ETH_RxDMABurstLength = ETH_RxDMABurstLength_32Beat;
    ETH_InitStructure.ETH_TxDMABurstLength = ETH_TxDMABurstLength_32Beat;
    ETH_InitStructure.ETH_DMAArbitration =
        ETH_DMAArbitration_RoundRobin_RxTx_2_1;

    /* 配置Ethernet外设 */
    ETH_Init(&ETH_InitStructure);

    /* DMA中断设置 */
    ETH_DMAITConfig(ETH_DMA_IT_NIS | ETH_DMA_IT_R | ETH_DMA_IT_T, ENABLE)
        ;

    /* 初始化描述符列表 */
    ETH_DMATxDescChainInit(DMATxDscrTab, &Tx_Buff[0][0], ETH_TXBUFNB);
    ETH_DMARxDescChainInit(DMARxDscrTab, &Rx_Buff[0][0], ETH_RXBUFNB);

    /* DMA地址设置 */
    ETH_MACAddressConfig(ETH_MAC_Address0, (u8*)&stm32_eth_device.
        dev_addr[0]);
```

7.8.1.2. rt_stm32_eth_control

rt_stm32_eth_control函数需要实现获取 MAC 地址的功能

```
static rt_err_t rt_stm32_eth_control(rt_device_t dev, int cmd, void *args
    )
{
    switch(cmd)
    {
    case NIOCTL_GADDR:
        /* 获取MAC地址 */
        if(args) rt_memcpy(args, stm32_eth_device.dev_addr, 6);
        else return -RT_ERROR;
        break;

    default :
```

```
        break;
    }

    return RT_EOK;
}
```

别的设置标准驱动接口可以不实现，先写个空函数即可。

7.8.2. 数据包收发接口

7.8.2.1. rt_stm32_eth_rx `rt_stm32_eth_rx`会去读取接收缓冲区中的数据，并放入 pbuf（lwIP 中利用结构体 pbuf 来管理数据包）中，并返回 pbuf 指针。

“erx” 接收线程会阻塞在获取`eth_rx_thread_mb`邮箱上，当它接收到邮件时，会调用`rt_stm32_eth_rx`去接收数据。

7.8.2.2. rt_stm32_eth_tx `rt_stm32_eth_tx`会将要发送的数据放入发送缓冲区，等待 DMA 来发送数据。

“etx” 发送线程会阻塞在获取`eth_tx_thread_mb`邮箱上，当它接收到邮件时，会调用`rt_stm32_eth_tx`来发送数据。

8 EMAC 驱动调试

8.1 实验环境搭建

工程默认启用了 DHCP 功能，需要有 DHCP 服务器来分配 IP 地址，常见的连接拓展如图：

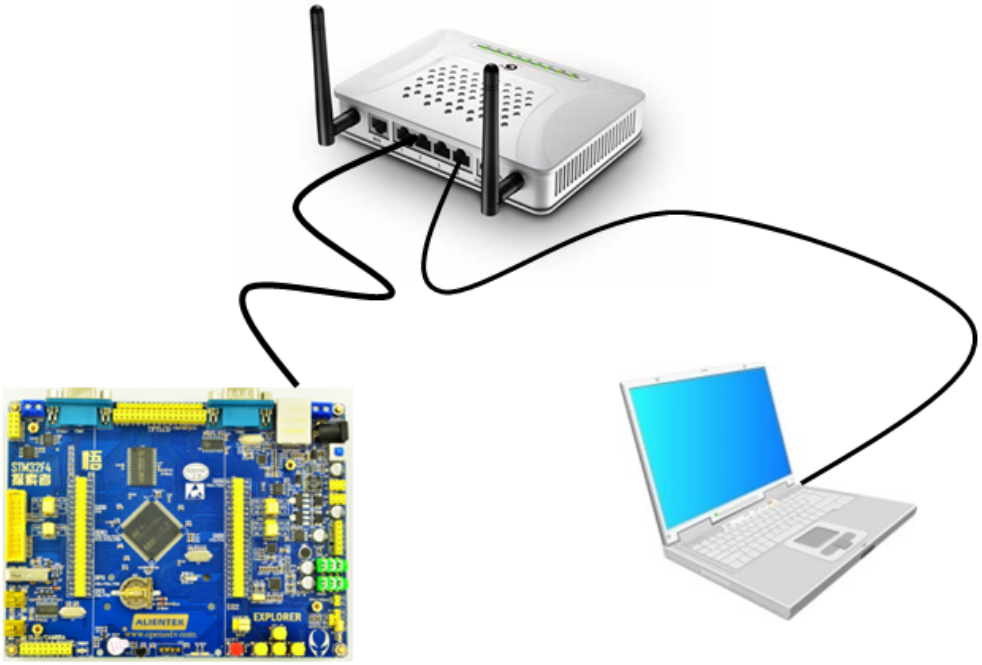

图 8: *eth_RJ45*

如果没有方便的实际环境，也可以先通过 ENV 配置固定 IP，然后用网线直接连接到调试用的电脑。

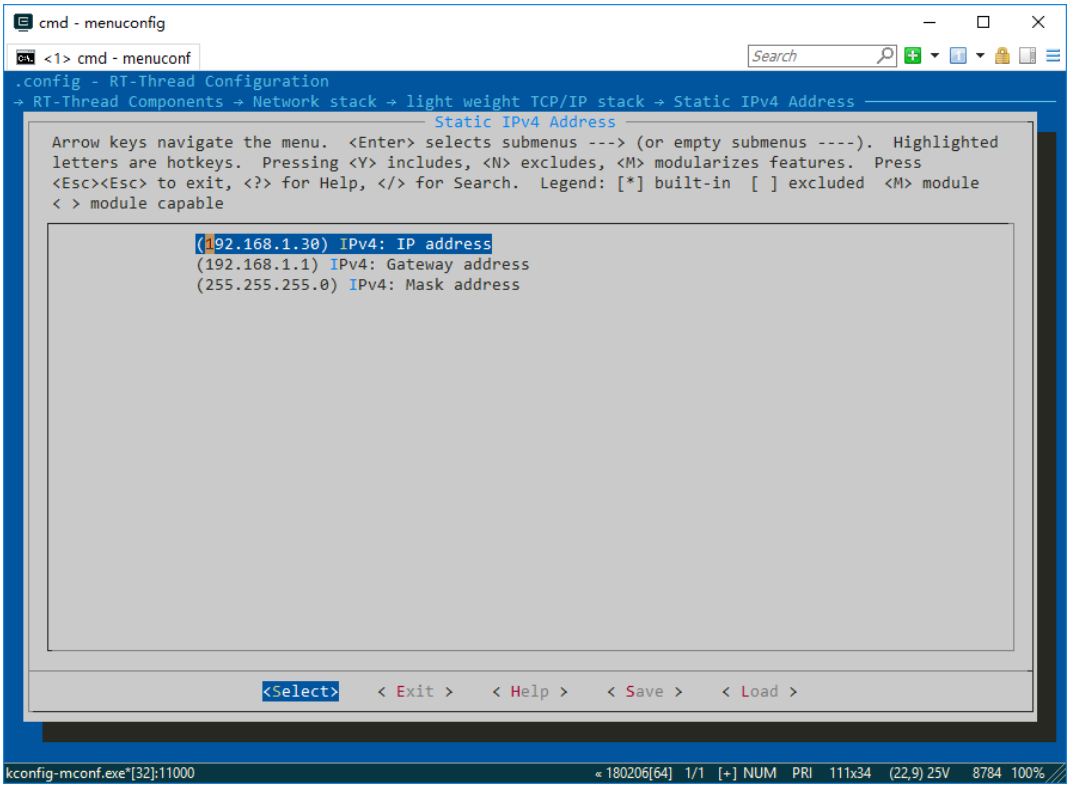

图 9: *ipadress*

电脑和开发板需要设置同网段的 IP 地址。

8.2 确认 PHY 连接状态

当支持网络的固件在开发板上面运行起来后，应该首先检查 RJ45 指示灯的状态。

正常应该是有灯常亮，且有数据收发时会出现闪烁。

如果发现灯没有亮，请先确认开发板硬件是否完好；然后检查供电是否充足，网线是否接好。

然后程序上配合硬件确认是否有正确复位 PHY，直到正常闪烁为止。

8.3 确认 IP 地址

在 shell 命令行执行`ifconfig`命令即可查看网卡的 IP 地址。

如果启用了 DHCP 功能，且此时显示已经获取到了 IP 地址，说明驱动的收发功能都已经正常。

如果是静态 IP 或是没有获取到 IP，请留意网卡 FLAG 中的 UP 和 LINK_UP 标志。如果显示有 LINK_DOWN，请确认 PHY 的管理程序有正确识别到 PHY 的速率和双工状态。并正确通过`eth_device_linkchange`通知到 lwIP。

如果启用了 DHCP 功能，且已经显示 LINK_UP。但没能正确获取到 IP。说明开发板与 DHCP 服务器通信不畅，需要进一步调试。

```
msh />ifconfig
network interface: e0 (Default)
MTU: 1500
MAC: 00 04 9f 05 44 e5
FLAGS: UP LINK_UP ETHARP
ip address: 192.168.12.127
gw address: 192.168.10.1
net mask   : 255.255.0.0
dns server #0: 192.168.10.1
dns server #1: 223.5.5.5
msh />
```

图 10: *ifconfig*

可以通过拔插网线观察 LINK 状态来判断是否正确读取了 PHY 寄存器的值

8.4 打印数据包

通过打开驱动中的 ETH_RX_DUMP 和 ETH_TX_DUMP 功能。可以把收发的数据包打印出来。

```
tx_dump, len:350
ff ff ff ff ff ff 00 80   e1 0c 26 27 08 00 45 00
01 50 00 05 00 00 ff 11   ba 98 00 00 00 00 ff ff
ff ff 00 44 00 43 01 3c   1d ca 01 01 06 00 56 47
a0 b0 00 00 00 00 00 00   00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 80   e1 0c 26 27 00 00 00 00
00 00 00 00 00 00 00 00   00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00   00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00   00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00   00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00   00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00   00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00   00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00   00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00   00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00   00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00   00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00   00 00 00 00 00 00 00 00
00 00 00 00 00 00 63 82   53 63 35 01 01 39 02 05
dc 37 04 01 03 1c 06 ff   00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00   00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00   00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00   00 00 00 00 00 00
dump done!
```

图 **11:** *dump*

当 RJ45 指示灯正常闪烁时，说明有数据包在收发。因为网络中经常会有广播数据包，如果此时有数据包进来，会有 RX_DUMP 的打印。如果一直没有打印，则重点检查两点：

- MII/RMII 的 RX 线路有问题，包括硬件问题或 IO 映射错误。
- EMAC 和 PHY 的速率和双工模式不配置，如 EMAC 工作在 10M，而 PHY 连接为 100M。

需要确认有正确获取 PHY 的速率和双工模式，同时可以与网线另外一端对比。如电脑上在显示 10M，而板子上面没有更新 PHY 管理程序，默认为 100M。

如果有必要，也可以打印 PHY 的 LOOPBACK 功能。以确认 MII/RMII 总线是完好的。

8.5 ping 测试

可以通过电脑 ping 板子或者板子 ping 电脑（需要开启netutils组件包中的 ping 功能）来测试驱动是否移植成功。

```
C:\Users\yhd>ping 192.168.12.127

正在 Ping 192.168.12.127 具有 32 字节的数据:
来自 192.168.12.127 的回复: 字节=32 时间<1ms TTL=255
来自 192.168.12.127 的回复: 字节=32 时间<1ms TTL=255
来自 192.168.12.127 的回复: 字节=32 时间<1ms TTL=255
来自 192.168.12.127 的回复: 字节=32 时间<1ms TTL=255

192.168.12.127 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 = 0ms，最长 = 0ms，平均 = 0ms
```

图 **12:** *ping1*

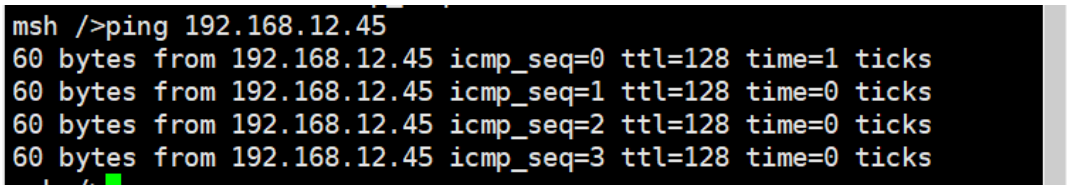

图 13: *ping2*

ping 之前要注意一下事项：

- 部分 PHY 芯片需要按照手册进行额外的初始化
- 板子和电脑要在一个网段内
- 如果电脑同时连着网线和 wifi，会出现 ping 不通板子的现象
- 查看防火墙是否禁用了 ping 功能
- 有些企业网络会禁止 ping 功能，建议更换网络环境
- MAC 地址不规范，板子无法 ping 通外部网络
- 如果 erx 栈大小设置的太小，会造成 erx 线程栈溢出

8.6 wireshark 抓包

如果板子有 RX DUMP，但依然无法通信时，可以在电脑上面使用 wireshark 抓包。

电脑 ping 开发板，开发板收到目标地址为自己的 IP 地址的请求包 (request)，然后，板子会给电脑做出回应，发送一个目标地址为电脑 IP 地址的回应包 (reply)。

根据是否有回应 (reply)，可以分两种情况检查

- 板子是否收到了请求包 (request)。
- 板子是否有发送回应 (reply)。

如果有回应，但是 ping 不成功，可以检查数据包内容是否符合规范。

8.6.1. 按 MAC 地址过滤

当板子有发出广播包，在电脑上面可以收到。（如 DHCP Discoverer）

在使用 wireshark 抓包过程中，主要是灵活使用各种过滤器，过滤出我们所关心的数据包。

当开发板还没有拿到 IP 地址时，可以使用开发板的 MAC 地址作为过滤器条件。

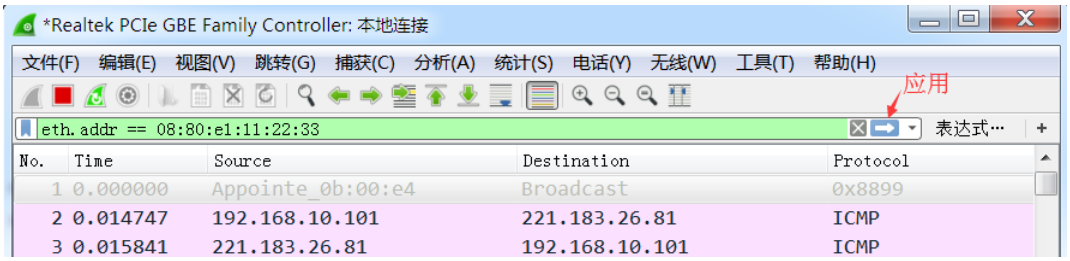

图 14: *wireshark_mac*

当开发板成功 link_up 时，会主动发出 DHCP 请求包，源地址就是开发板自己的 MAC 地址。

8.6.2. 按 IP 地址过滤

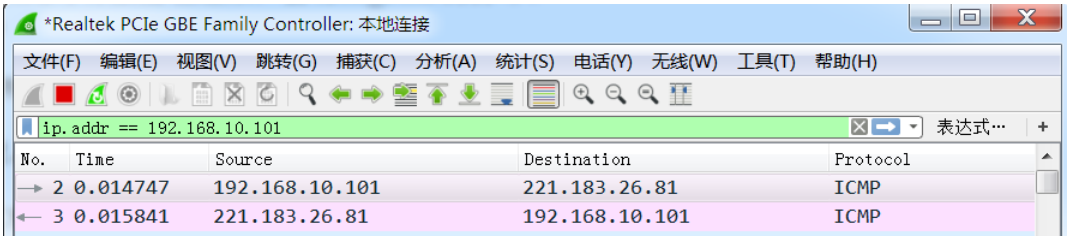

图 15: *wireshark_ip*

或配置开发板为静态 IP，然后 PC 上面执行 ping 命令，ping 开发板的 IP 地址。

这里把开发板的 IP 地址作为过滤条件，正常情况下，PC 会先发出请求包 (request)。