Tina Linux Ethernet
开发指南

版本号: 1.0
发布日期: 2021.11.10

版本历史

版本号	日期	制/修订人	内容描述
1.0	2021.11.10	AWA1381	1. 建立初始版本

目　录

插　图

1 前言

1.1 文档简介

介绍 Allwinner 平台上以太网驱动移植，介绍 Tina 以太网框架和使用方法，以及记录常见问题的分析过程。

1.2 目标读者

适用 Tina 平台的广大客户和对 Tina 以太网感兴趣的同事。

1.3 适用范围

Allwinner 软件平台 Tina v3.0 版本及以上。

Allwinner 硬件平台 R 系列 (R6，R11，R16，R18，R30，R40，R328，R331，R329，R818，T113...)。

Allwinner 硬件平台 MR 系列 (MR133，MR813...)。

Allwinner 硬件平台 H 系列 (H133...)。

1.4 相关术语介绍

术语	解释说明
SUNXI	Allwinner 一系列 SOC 硬件平台
MAC	Media Access Control，媒体访问控制协议
GMAC	千兆以太网控制器
PHY	物理收发器
MII	Media Independent Interface，媒体独立接口，是 MAC 与 PHY 之间的接口
RMII	简化媒体独立接口
RGMII	简化千兆媒体独立接口
RTL8363NB_VB	瑞昱的一种双网口芯片 (目前仅适配 T113/R818 平台)

2 模块介绍

2.1 功能介绍

以太网是一种局域网通信技术，遵循 IEEE802.3 协议规范，包括 10M、100M、1000M 和 10G 等多种速率的以太网。简单讲就是有线网络。

2.1.1 主要分类

第一类是经典以太网，是以太网的原始形式，运行速度从 3~10 Mbps 不等；

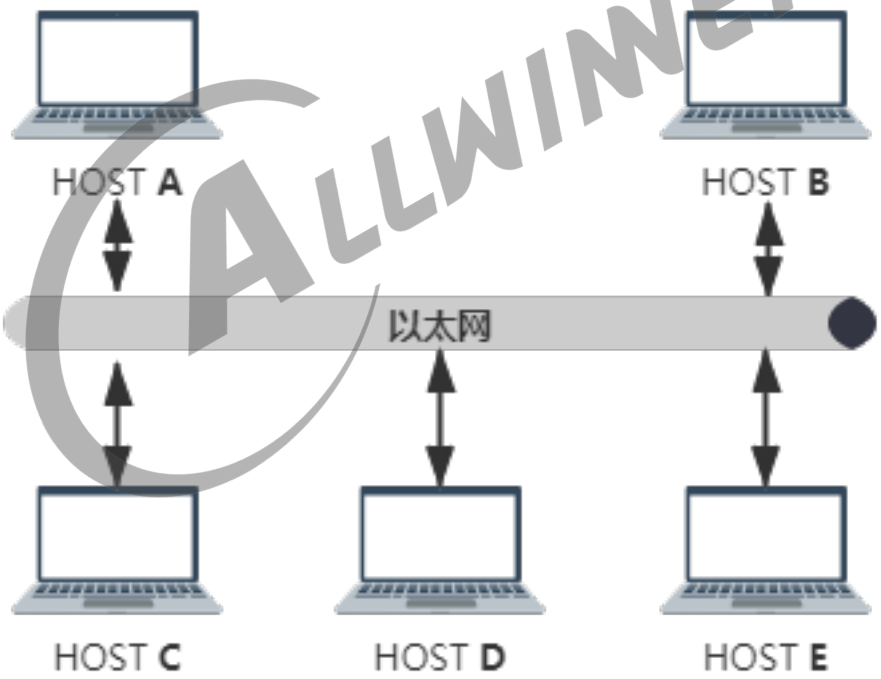

图 2-1: 经典以太网结构

第二类是交换式以太网，使用了交换机设备，可运行在 100、1000 和 10000Mbps；

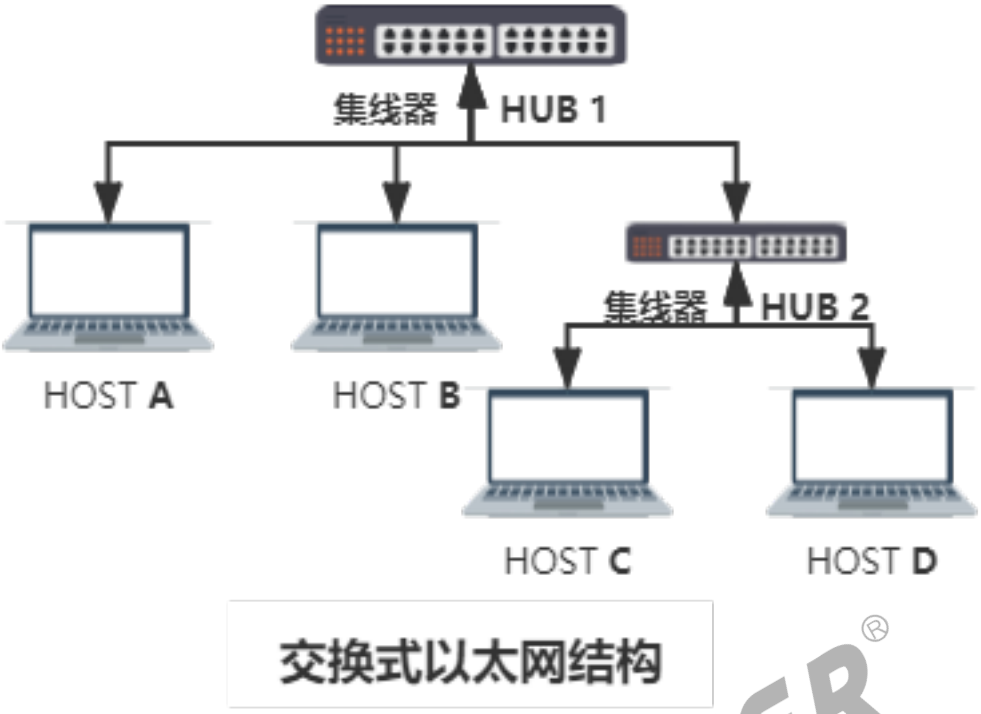

图 2-2: 交换式以太网

2.1.2 主要特点

1.IEEE 组织的 IEEE 802.3 标准制定了以太网的技术标准。[802.11]

2. 在 OSI 七层协议中工作在数据链路层和物理层。

3. 采用载波多路访问和冲突检测（CSMA/CD ）机制。[CSMA/CA]

4. 多种速率：10/100/1000Mbps，10Gbps。

5. 拓扑结构丰富。

6. 信息的发送采用多种编码方式。

2.2 硬件框架

以太网涉及到的硬件并不只是我们常说的网线连接网口，有丰富的器件组成。

2.2.1 组成器件

以 T113 介绍

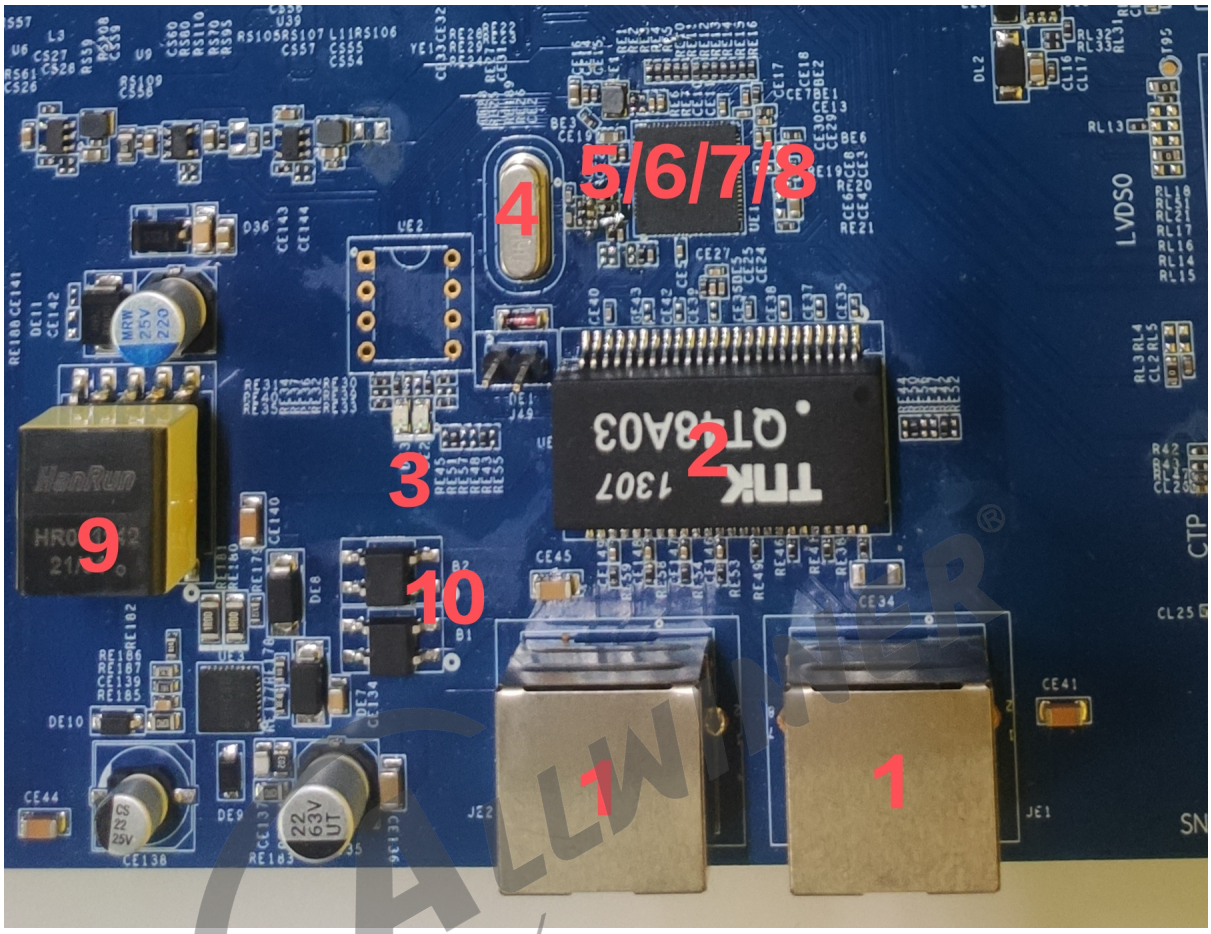

图 2-3: T113 以太网硬件组成

1.RJ45 连接器.

2. 网络变压器 (讯康).

3. 状态指示灯.

4. 晶体.

5/6/7/8.PHY 收发器、MAC 控制器、MII 接口，SMI 接口.

9.PoE 电源变压器 (汉仁电子).

10.ESD 保护芯片.

2.2.2 拓扑结构

以太网的拓扑架构有多种方式，可以使用主控芯片内置的 phy，也可以使用外挂单 phy，还可以是 mac 和 phy 集成的。

全集成：

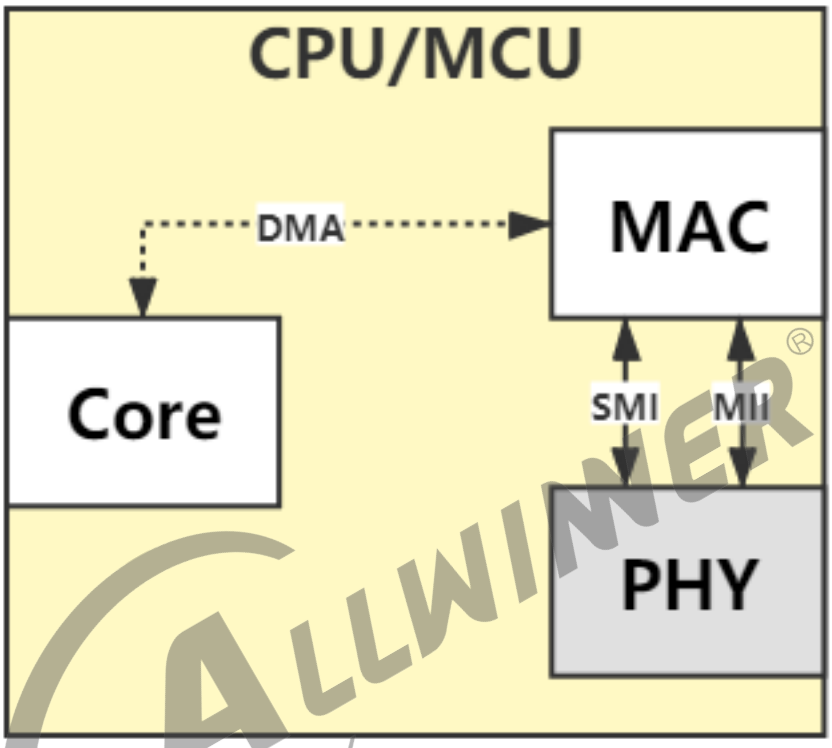

图 2-4: 全集成于 soc

SOC 和 ETHNET 结构：

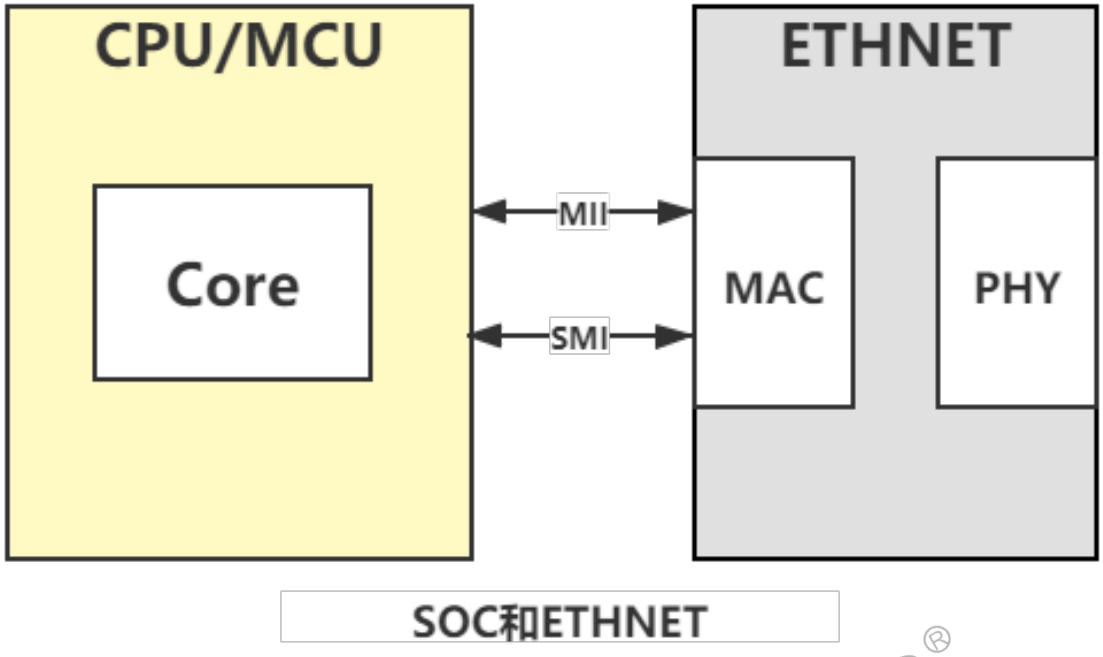

图 2-5: soc 和 ethnet

单 PHY 外挂：

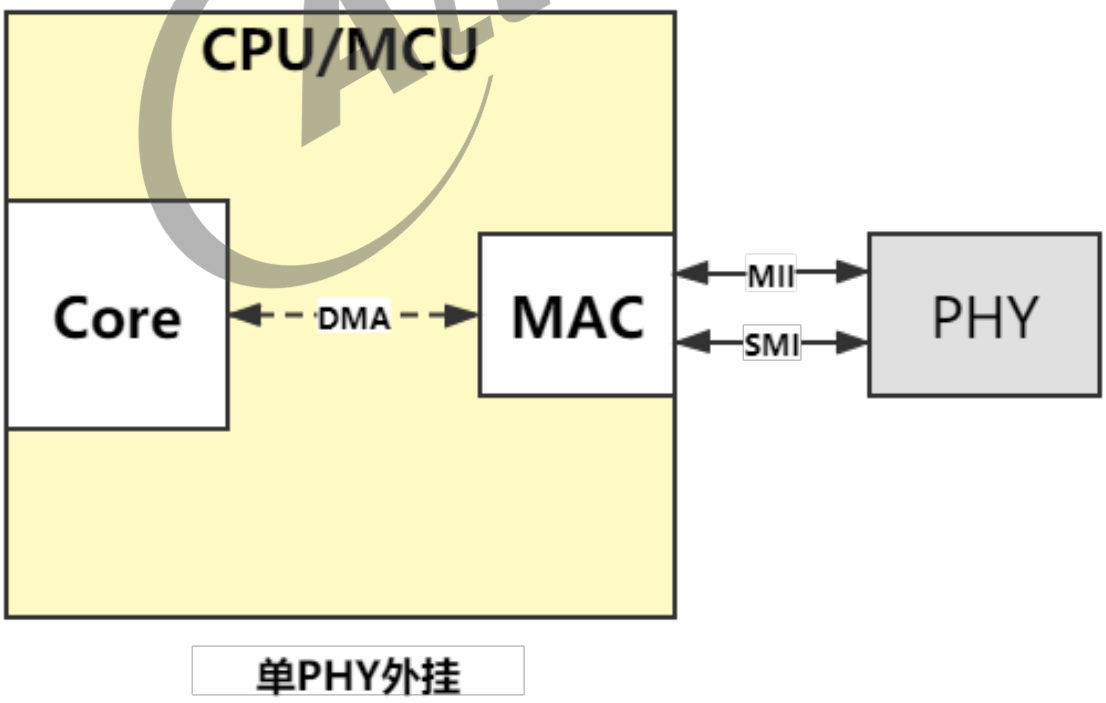

图 2-6: 单 phy 外挂

SOC 和 SWITCH:

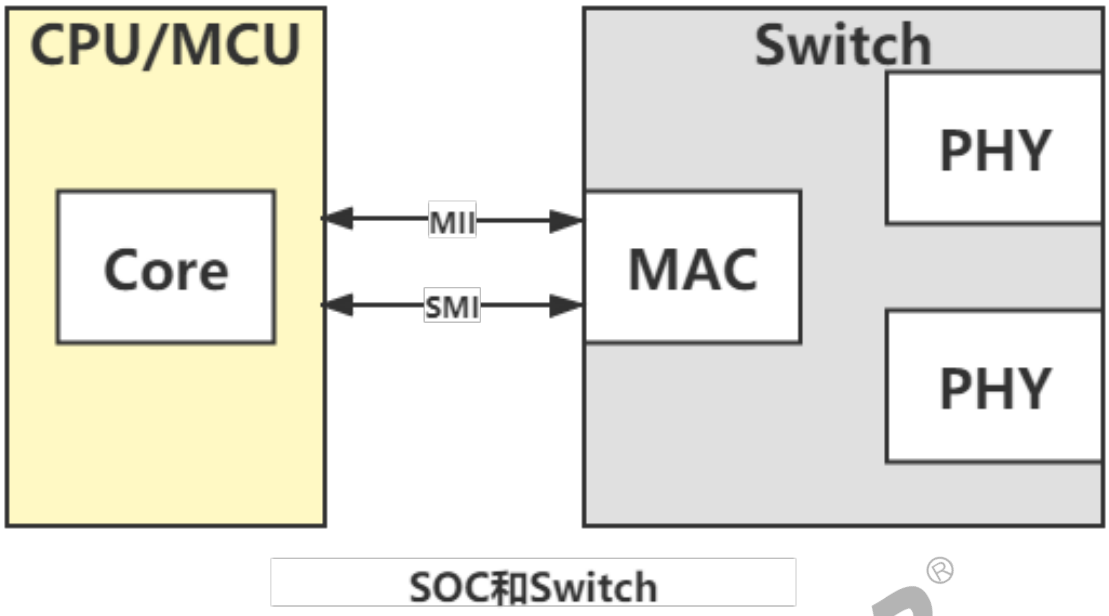

图 2-7: soc 和 switch

SOC 的 mac 和 switch:

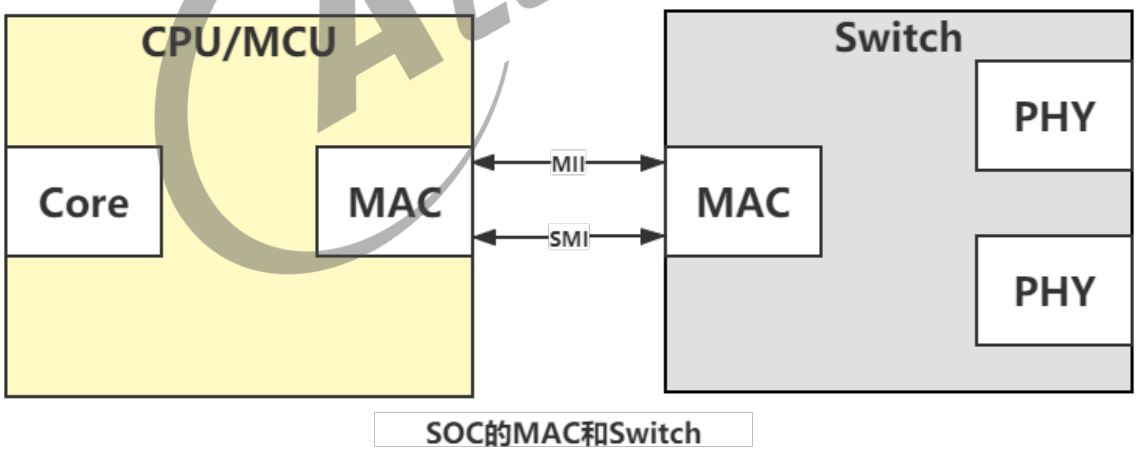

图 2-8: soc 集成 mac 和 switch

2.3 软件框架

2.3.1 协议框架

以太网在 TCP/IP 协议族中的位置如下图所示：

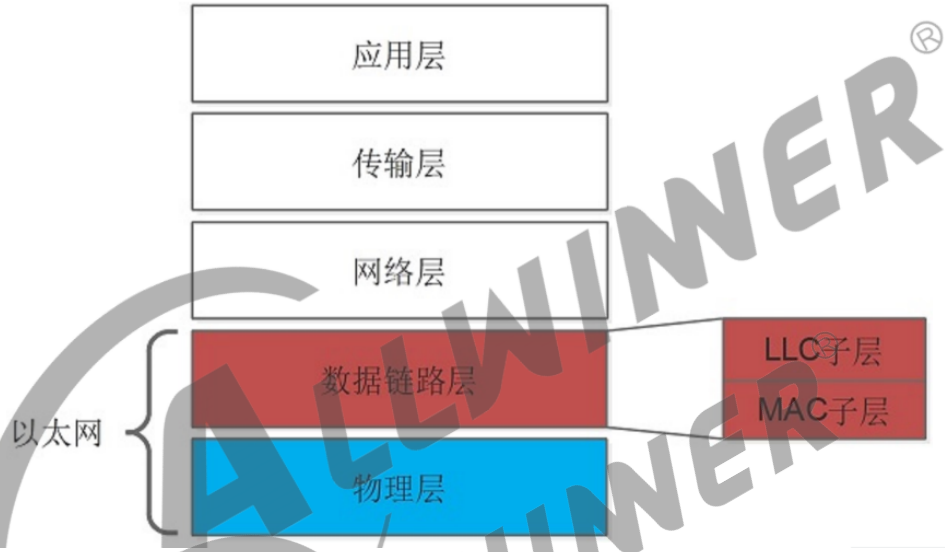

图 2-9: 以太网在 TCPIP 协议族中的位置

以太网与 TCP/IP 协议族的物理层（L1）和数据链路层（L2）相关，其中数据链路层包括逻辑链路控制（LLC）和媒体访问控制（MAC）子层。

2.3.2 驱动框架

Linux 内核中网络设备框架如下图所示:

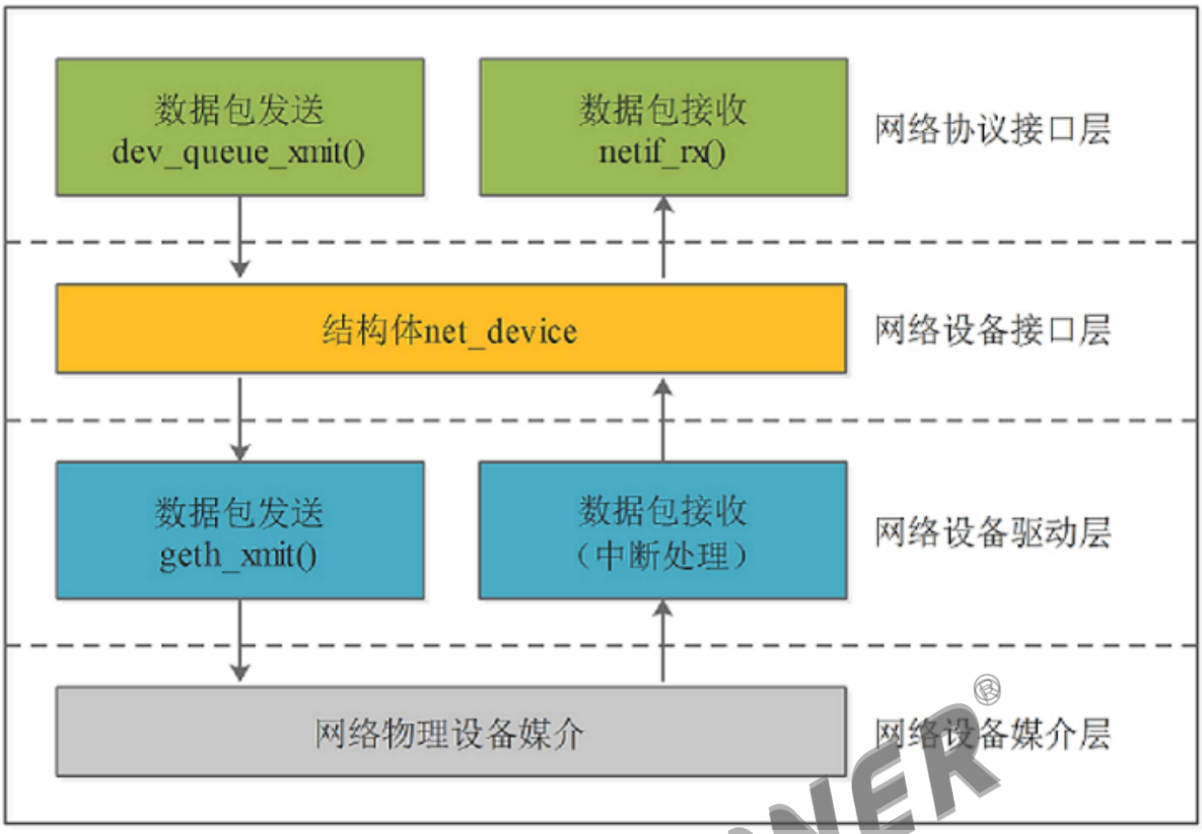

图 2-10：以太网网络设备框架

从上至下分为 4 层：

（1）网络协议接口层：向网络协议层提供统一的数据包收发接口，通过 dev_queue_xmit() 发送数据，并通过 netif_rx() 接收数据；

（2）网络设备接口层：向协议接口层提供统一的用于描述网络设备属性和操作的结构体 net_device，该结构体是设备驱动层中各函数的容器；

（3）网络设备驱动层：实现 net_device 中定义的操作函数指针（通常不是全部），驱动硬件完成相应动作；

（4）网络设备媒介层：完成数据包发送和接收的物理实体，包括网络适配器和具体的传输媒介。

3 模块配置

3.1 内核配置

内核目录下执行 make kernel_menuconfig

3.1.1 协议栈配置

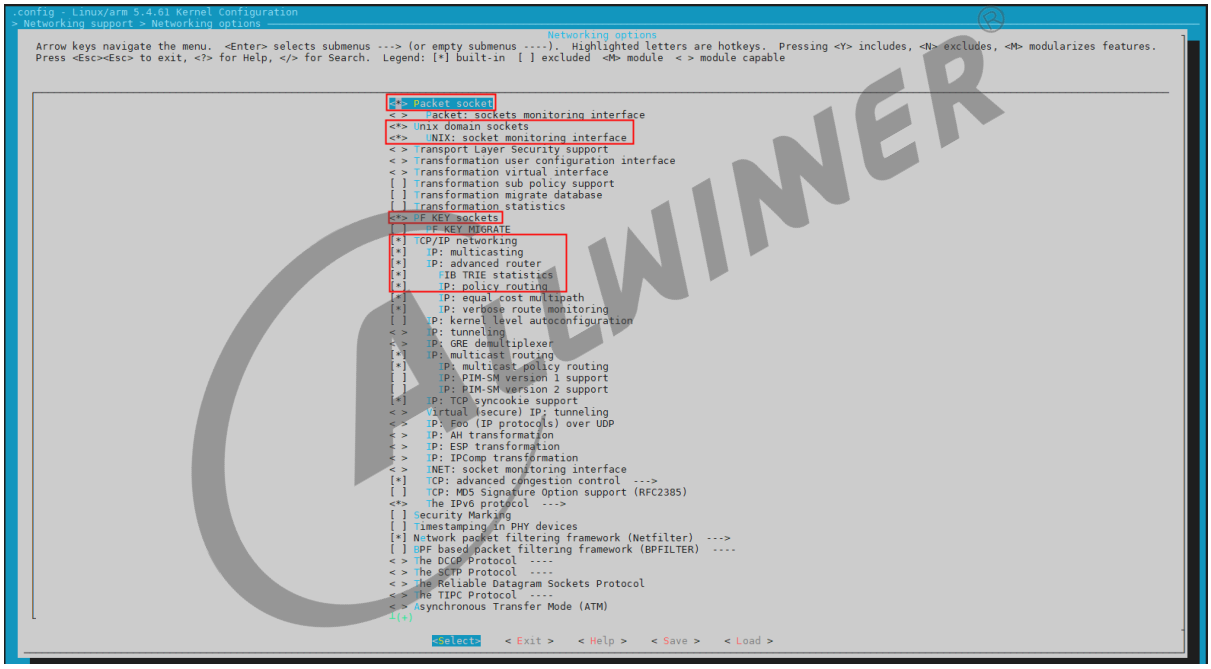

图 3-1: gmac_ 协议栈配置

3.1.2 驱动配置

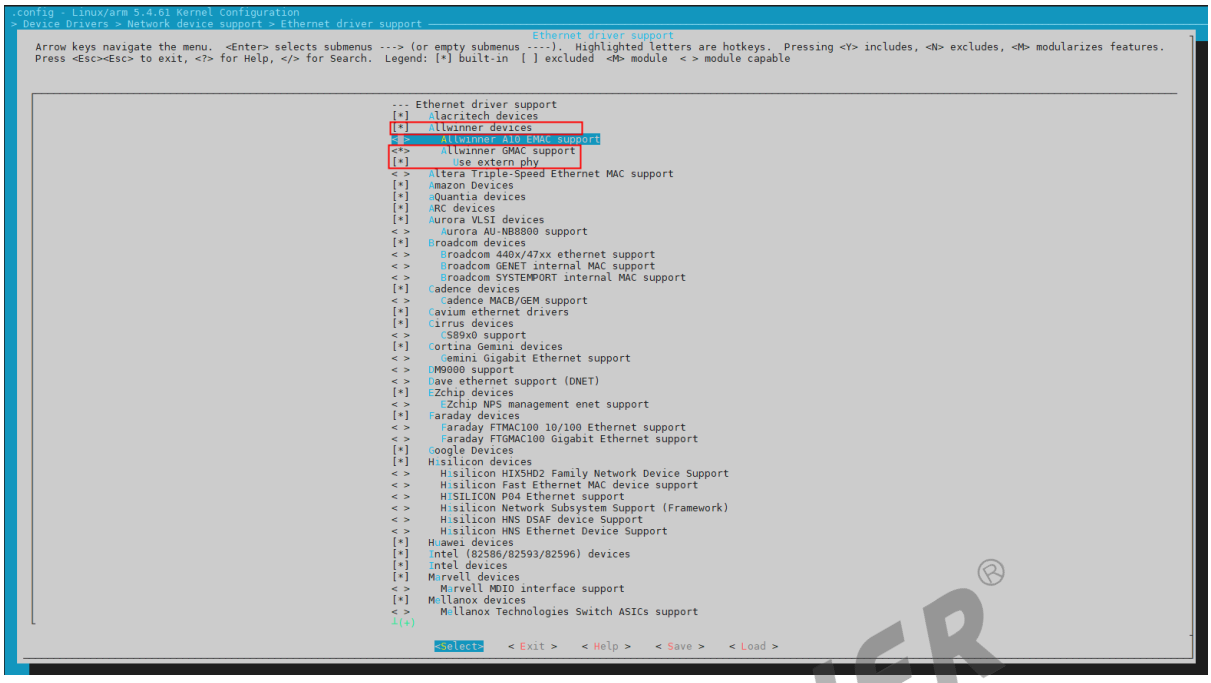

图 3-2: gmac_allwinner_ 驱动配置

注：这里只介绍使用外部 PHY 的配置。

若使用双网口需要配置原厂的驱动，如：RTL8363NB_VB

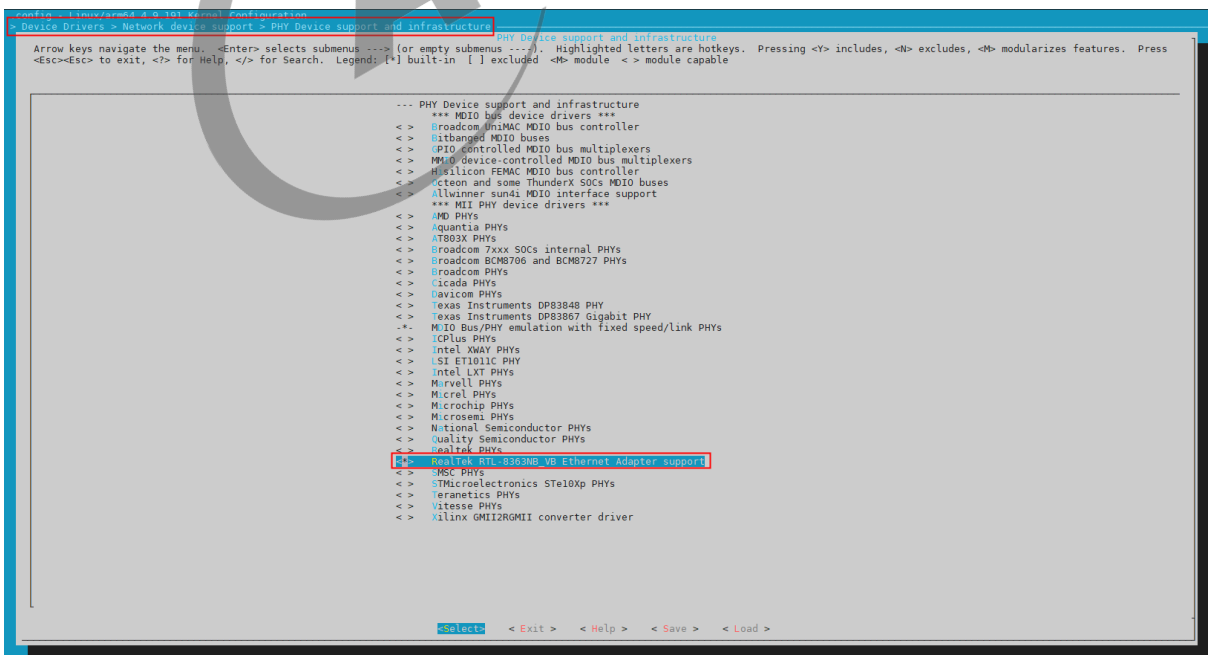

图 3-3: RTL8363NB_VB 配置

3.2 设备树配置

```
429         gmac0_pins_a: gmac@0 {
430                 pins = "PE0", "PE1", "PE2", "PE3",
431                        "PE4", "PE5", "PE6",
432                        "PE10","PE11",
433                        "PE12", "PE13", "PE14", "PE15";
434                 function = "gmac0";
435                 drive-strength = <10>;
436         };

438         gmac0_pins_b: gmac@1 {
439                 pins = "PE0", "PE1", "PE2", "PE3",
440                        "PE4", "PE5", "PE6",
441                        "PE10", "PE11",
442                        "PE12", "PE13", "PE14", "PE15";
443                 function = "gpio_in";
444         };

809 &gmac0 {
810         phy-mode = "rgmii";
811         use_ephy25m = <1>;
812         pinctrl-0 = <&gmac0_pins_a>;
813         pinctrl-1 = <&gmac0_pins_b>;
814         pinctrl-names = "default", "sleep";
815         tx-delay = <7>;
816         rx-delay = <31>;
817         /*phy-rst = <&pio PA 14 GPIO_ACTIVE_HIGH>;*/
818         status = "okay";
819 };
```

属性	说明
phy-mode	GMAC 与 PHY 之间的物理接口，如 MII、RMII、RGMII 等；
use_ephy25m	PHY 使用的 25M 时钟来源，1，代表 PHY 使用 SOC 内部 EPHY_25M 时钟；0 或者不配置该参数，需外挂 25M 晶振为 PHY 提供时钟；
pinctrl-0	设备 active 状态下的 GPIO 配置；
pinctrl-1	设备 suspend 状态下的 GPIO 配置；
tx-delay	tx 时钟延迟，tx-delay 取值 0-7，一档约 536ps（皮秒）；
rx-delay	rx 时钟延迟，rx-delay 取值 0-31，一档约 186ps（皮秒）；
phy-rst	PHY 复位脚，有时候硬件设计没有通过 gpio，而是强行上拉；
status	是否使能该设备节点；

4 源码分析

4.1 源码结构

GMAC 驱动的源代码位于内核 drivers/net/ethernet/allwinner 目录下:

```
.
├── Kconfig
├── Makefile
├── sunxi-gmac.c      // Sunxi平台GMAC驱动核心代码
├── sunxi-gmac.h      // Sunxi平台GMAC驱动头文件，里面定义了一些宏、数据结构及内部接口
├── sunxi-gmac-ops.c   // Sunxi平台GMAC驱动各个内部接口具体实现
```

4.2 流程分析

4.2.1 初始化

drivers/net/ethernet/allwinner /sunxi-gmac.c

2078 static int geth_probe(struct platform_device *pdev)

```
2073 /**
2074  * geth_probe
2075  * @pdev: platform device pointer
2076  * Description: the driver is initialized through platform_device.
2077  */
2078 static int geth_probe(struct platform_device *pdev)
2079 {
...
2084         pr_info("sunxi gmac driver's version: %s\n", SUNXI_GMAC_VERSION);
...
2091         ndev = alloc_etherdev(sizeof(struct geth_priv));  //1.初始化net_device结构体
...
2098         priv = netdev_priv(ndev);  //2.初始化geth_priv结构体，存放私有数据
2099         platform_set_drvdata(pdev, ndev); //3.存储用户在probe()中主动申请的内存区域的指针以防
    止丢失
2100
2101         /* Must set private data to pdev, before call it */
2102         ret = geth_hw_init(pdev);  //4."gmac-power0: NULL"
...
     #ifdef CONFIG_RTL8363NB_VB
            rtl8363nb_vb_init();  //5.这里会穿插对双网口的初始化操作。
     #endif
2108         /* setup the netdevice, fill the field of netdevice */
2109         ether_setup(ndev);    //6.
```

```
2110         ndev->netdev_ops = &geth_netdev_ops; //7.
2111         netdev_set_default_ethtool_ops(ndev, &geth_ethtool_ops);
2112         ndev->base_addr = (unsigned long)priv->base;
2113
2114         priv->ndev = ndev;
2115         priv->dev = &pdev->dev;
...
2132         /* The last val is mdc clock ratio */
2133         sunxi_geth_register((void *)ndev->base_addr, HW_VERSION, 0x03);  //8.
2134
2135         ret = register_netdev(ndev);  //9.
...
2142         /* Before open the device, the mac address should be set */
2143         geth_check_addr(ndev, mac_str);   //10.获取mac地址“eth0: Use random mac address
    ”
...          /*文件节点的创建*/  //11.
2148         device_create_file(&pdev->dev, &dev_attr_gphy_test);
2149         device_create_file(&pdev->dev, &dev_attr_mii_reg);
2150         device_create_file(&pdev->dev, &dev_attr_loopback_test);
2151         device_create_file(&pdev->dev, &dev_attr_extra_tx_stats);
2152         device_create_file(&pdev->dev, &dev_attr_extra_rx_stats);
2153
2154         device_enable_async_suspend(&pdev->dev);
```

1. 初始化 net_device 结构体

ndev = alloc_etherdev(sizeof(struct geth_priv));

linux 标准函数

```
#define alloc_etherdev(sizeof_priv) alloc_etherdev_mq(sizeof_priv, 1)
#define alloc_etherdev_mq(sizeof_priv, count) alloc_etherdev_mqs(sizeof_priv, count, count)
sizeof_priv: synopGMACPciNetworkAdapter结构体大小。
因为net_device可以由驱动程序扩展私有空间，此参数表示扩展的私有空间大小。是网络设备驱动程序私有数据块的大小，
    在alloc_netdev_mqs函数中，将和net_device数据结构一起分配，但是sizeof_priv也可以设置为0，不需要私
    有数据块，或者自己分配私有数据块内存。如果和net_device数据结构一起分配驱动程序的私有数据块，则其私有数
    据块的内存地址通过函数net_dec_priv获取。

count: 发送队列的个数

count: 接收队列的个数
```

2. 初始化 geth_priv 结构体，存放私有数据

```
include/linux/netdevice.h
#define NETDEV_ALIGN        32
#define NETDEV_ALIGN_CONST  (NETDEV_ALIGN - 1)
static inline void *netdev_priv(struct net_device *dev)
{
    return (char *)dev + ((sizeof(struct net_device)+ NETDEV_ALIGN_CONST)
                & ~NETDEV_ALIGN_CONST);
}
给sizeof(struct net_device)加上一定的值，确保加过后的值的大小为32的倍数。即此句的大小是32的倍数。
这样，当sizeof(struct net_device)的值小于32时，那这句的值32；当sizeof(struct net_device)的值大于32
    而小于64时，那这句的值64
```

3. 存储用户在 probe() 中主动申请的内存区域的指针以防止丢失

ndev是我们在probe函数中定义的局部变量，如果我想在其他地方使用它怎么办呢？ 这就需要把它保存起来。内核提供了这个方法，使用函数platform_set_drvdata()可以将ndev保存成平台总线设备的私有数据。
函数static inline void platform_set_drvdata(struct platform_device *pdev, void *data)
目的存储用户在probe()中主动申请的内存区域的指针以防止丢失
static inline void *platform_get_drvdata(const struct platform_device *pdev)则是将其取出。
函数platform_set_drvdata()和platform_get_drvdata(),定义于..\include\linux\platform_device.h

4.ret = geth_hw_init(pdev);

```
1873 /* config hardware resource */
1874 static int geth_hw_init(struct platform_device *pdev)
1875 {
1893         /* config memery resource */
1894         res = platform_get_resource(pdev, IORESOURCE_MEM, 0);
...
1920         /* config IRQ */
1921         ndev->irq = platform_get_irq_byname(pdev, "gmacirq");
...
1928         ret = request_irq(ndev->irq, geth_interrupt, IRQF_SHARED, dev_name(&pdev->dev)
    , ndev);
...
1934         /* get gmac rst handle */
1935         priv->reset = devm_reset_control_get(&pdev->dev, NULL);
...
1941         /* config clock */
1942         priv->geth_clk = of_clk_get_by_name(np, "gmac");
...
1949         if (INT_PHY == priv->phy_ext) {
1950                 priv->ephy_clk = of_clk_get_by_name(np, "ephy");
1951                 if (unlikely(IS_ERR_OR_NULL(priv->ephy_clk))) {
1952                         pr_err("Get ephy clock failed!\n");
1953                         ret = -EINVAL;
1954                         goto clk_err;
1955                 }
1956         } else {
1957                 if (!of_property_read_u32(np, "use-ephy25m", &(priv->use_ephy_clk))
1958                                 && priv->use_ephy_clk) {
1959                         priv->ephy_clk = of_clk_get_by_name(np, "ephy");
1960                         if (unlikely(IS_ERR_OR_NULL(priv->ephy_clk))) {
1961                                 pr_err("Get ephy clk failed!\n");
1962                                 ret = -EINVAL;
1963                                 goto clk_err;
1964                         }
1965                 }
1966         }
1967
1968         /* config power regulator */
...
1986         /* config other parameters */
1987         priv->phy_interface = of_get_phy_mode(np);
1988         if (priv->phy_interface != PHY_INTERFACE_MODE_MII &&
1989             priv->phy_interface != PHY_INTERFACE_MODE_RGMII &&
1990             priv->phy_interface != PHY_INTERFACE_MODE_RMII) {
1991                 pr_err("Not support phy type!\n");
1992                 priv->phy_interface = PHY_INTERFACE_MODE_MII;
1993         }
1994
1995         if (!of_property_read_u32(np, "tx-delay", &value))
1996                 priv->tx_delay = value;
```

```
1997
1998        if (!of_property_read_u32(np, "rx-delay", &value))
1999                priv->rx_delay = value;
2000
2001        /* config pinctrl */
2002        if (EXT_PHY == priv->phy_ext) {
2003                priv->phyrst = of_get_named_gpio_flags(np, "phy-rst", 0, (enum
    of_gpio_flags *)&cfg);
2004                priv->rst_active_low = (cfg.data == OF_GPIO_ACTIVE_LOW) ? 1 : 0;
2005
2006                if (gpio_is_valid(priv->phyrst)) {
2007                        if (gpio_request(priv->phyrst, "phy-rst") < 0) {
2008                                pr_err("gmac gpio request fail!\n");
2009                                ret = -EINVAL;
2010                                goto pin_err;
2011                        }
2012                }
2013
2014                priv->pinctrl = devm_pinctrl_get_select_default(&pdev->dev);
2015                if (IS_ERR_OR_NULL(priv->pinctrl)) {
2016                        pr_err("gmac pinctrl error!\n");
2017                        priv->pinctrl = NULL;
2018                        ret = -EINVAL;
2019                        goto pin_err;
2020                }
2021        }
...
```

注：这里主要即使关于时钟，gpio，rst 等的配置处理，获取 dts 的配置。

5. 对 RTL8363NB_VB 的初始化

```
 591 #ifdef CONFIG_RTL8363NB_VB
 592 /* rtl8363nb_vb switch init. */
 593 static int rtl8363nb_vb_init(void)
 594 {
 595         pr_info("%s->%d rtk8363nb_vb init=====\n", __func__, __LINE__);
 596
 597         if (rtk_switch_init() != RT_ERR_OK) {       //调用原厂的驱动的初始化函数
 598                 pr_info("rtk switch init failed!\n");
 599                 return -1;
 600         }
 601         mode = MODE_EXT_RGMII;                      //设置rtl8363nb_vb的模式
 602         mac_cfg.forcemode = MAC_FORCE;
 603         mac_cfg.speed = SPD_1000M;
 604         mac_cfg.duplex = FULL_DUPLEX;
 605         mac_cfg.link = PORT_LINKUP;
 606         mac_cfg.nway = DISABLED;
 607         mac_cfg.txpause = ENABLED;
 608         mac_cfg.rxpause = ENABLED;
 609
 610         if (rtk_port_macForceLinkExt_set(EXT_PORT0, mode, &mac_cfg) != RT_ERR_OK) {//设
    置端口
 611                 pr_info("macForceLinkExt set failed!\n");
 612                 return -1;
 613         }
 614
 615         rtk_port_rgmiiDelayExt_set(EXT_PORT0, 1, 0);  //设置delay参数
 616         rtk_port_phyEnableAll_set(ENABLED);  //enable
```

```
617         return 0;
618 }
```

6.ether_setup(ndev)

linux-5.4/net/ethernet/eth.c +377

```
371 /**
372  * ether_setup - setup Ethernet network device
373  * @dev: network device
374  *
375  * Fill in the fields of the device structure with Ethernet-generic values.
376  */
377 void ether_setup(struct net_device *dev)
378 {
379         dev->header_ops         = &eth_header_ops;
380         dev->type               = ARPHRD_ETHER;
381         dev->hard_header_len    = ETH_HLEN;
382         dev->min_header_len     = ETH_HLEN;
383         dev->mtu                = ETH_DATA_LEN;
384         dev->min_mtu            = ETH_MIN_MTU;
385         dev->max_mtu            = ETH_DATA_LEN;
386         dev->addr_len           = ETH_ALEN;
387         dev->tx_queue_len       = DEFAULT_TX_QUEUE_LEN;
388         dev->flags              = IFF_BROADCAST|IFF_MULTICAST;
389         dev->priv_flags         |= IFF_TX_SKB_SHARING;
390
391         eth_broadcast_addr(dev->broadcast);
392
393 }
394 EXPORT_SYMBOL(ether_setup);
网卡驱动为每一个新的接口在一个全局的网络设备列表里插入一个数据结构.每一个接口由一个结构 net_device 项来描写
    叙述，它在 <linux/netdevice.h> 里定义。该结构必须动态分配。

进行这样的分配的内核函数是 alloc_netdev，它有下列原型：
struct net_device *alloc_netdev(int sizeof_priv, const char *name, void (*setup)(struct
    net_device *));
sizeof_priv 是驱动的的"私有数据"区的大小；name 是这个接口的名子；这个名子能够有一个 printf 风格的 %d 在
    里面. 内核用下一个可用的接口号来替换这个 %d。setup 是一个初始化函数的指针，被调用来设置 net_device
    结构的剩余部分。
网络子系统为各种接口提供了一些帮助函数，包裹着 alloc_netdev。最通用的是 alloc_etherdev，定义在 <linux/
    etherdevice.h>，还有其它网络设备接口，如alloc_fcdev ( 定义在 <linux/fcdevice.h> ) 为 fiber-
    channel 设备，alloc_fddidev (<linux/fddidevice.h>) 为 FDDI 设备，或者 aloc_trdev (<linux/
    trdevice.h>) 为令牌环设备。
alloc_etherdev函数原型为:
struct net_device *alloc_etherdev(int sizeof_priv);
当中sizeof_priv 是驱动的的"私有数据"区的大小；这个函数分配一个网络设备使用 eth%d 作为参数 name. 它提供了
    自己的初始化函数 ( ether_setup )来设置几个 net_device 字段，使用对以太网设备合适的值。 因此，没有
    驱动提供的初始化函数给 alloc_etherdev;
```

7.ndev->netdev_ops = &geth_netdev_ops

```
实际上ether_setup（）函数也是不过负责了一些以太网范围中的缺省值而已。一般alloc_netdev函数提供的setup函数
    也会调用该函数来设置这些缺省值。
事实上无论使用alloc_netdev函数还是alloc_etherdev函数都不只设置这些缺省值就够了。
既然是编写网络设备驱动，就要完毕网卡的基本功能：收发数据包，统计网卡数据等，所以一般还要设置该结构体里面的这些
    功能函数指针，而这些函数正是网络设备驱动须要实现的基本功能。所以假设使用函数alloc_netdev分配空间，那么
    setup函数一般实现为：
```

```
{
        ether_setup(dev);

    #if (LINUX_VERSION_CODE >= KERNEL_VERSION(2,6,29))
        dev->netdev_ops = &xxx_netdev_ops;
    #else
        dev->open               = xxx_open;
        dev->stop               = xxx_stop;
        dev->hard_start_xmit    = xxx_tx;
        dev->get_stats          = xxx_stats;
    //    dev->change_mtu         = xxx_change_mtu;
        ...
    #endif
}
假设使用alloc_etherdev函数分配，那么在该函数之后还须要作例如以下设置：
#if (LINUX_VERSION_CODE >= KERNEL_VERSION(2,6,29))
        dev->netdev_ops = &xxx_netdev_ops;
    #else
        dev->open               = xxx_open;
        dev->stop               = xxx_stop;
        dev->hard_start_xmit    = xxx_tx;
        dev->get_stats          = xxx_stats;
    //    dev->change_mtu         = xxx_change_mtu;
        ...
    #endif

事实上二者的区别就是alloc_etherdev函数默认自己主动调用ether_setup函数而且自己主动分配一个网络设备使用
    eth%d 作为参数 name，而函数alloc_netdev须要自己传入参数来设置这些项而已。
net_device 结构初始化之后，传递这个结构给 register_netdev函数完毕注册。
```

所以这里通过句柄自己实现了一些回调：ndev->netdev_ops = &geth_netdev_ops;

```
1768 static const struct net_device_ops geth_netdev_ops = {
1769         .ndo_init = NULL,
1770         .ndo_open = geth_open,
1771         .ndo_start_xmit = geth_xmit,
1772         .ndo_stop = geth_stop,
1773         .ndo_change_mtu = geth_change_mtu,
1774         .ndo_fix_features = geth_fix_features,
1775         .ndo_set_rx_mode = geth_set_rx_mode,
1776         .ndo_tx_timeout = geth_tx_timeout,
1777         .ndo_do_ioctl = geth_ioctl,
1778         .ndo_set_config = geth_config,
1779 #if IS_ENABLED(CONFIG_NET_POLL_CONTROLLER)
1780         .ndo_poll_controller = geth_poll_controller,
1781 #endif
1782         .ndo_set_mac_address = geth_set_mac_address,
1783         .ndo_set_features = geth_set_features,
1784 };
```

8.sunxi_geth_register((void *)ndev->base_addr, HW_VERSION, 0x03);

```
669 int sunxi_geth_register(void *iobase, int version, unsigned int div)
670 {
671         hwdev.ver = version;
672         hwdev.iobase = iobase;
673         hwdev.mdc_div = div;
674
```

```
675         return 0;
676 }
```
这个函数相对简单，就是对硬件的一些缺省值赋值。

9.ret = register_netdev(ndev);

设备注册，这一步后当然也就会调用前面的那些回调，比如，6 中的 ndev->netdev_ops = &geth_netdev_ops;-》static int geth_open(struct net_device *ndev);

10.geth_check_addr(ndev, mac_str);

获取 mac 地址

```
1096 static void geth_check_addr(struct net_device *ndev, unsigned char *mac)
1097 {
1098         int i;
1099         char *p = mac;
1100
1101         if (!is_valid_ether_addr(ndev->dev_addr)) {
1102                 for (i = 0; i < ETH_ALEN; i++, p++)
1103                         ndev->dev_addr[i] = simple_strtoul(p, &p, 16);
1104
1105                 if (!is_valid_ether_addr(ndev->dev_addr))
1106                         geth_chip_hwaddr(ndev->dev_addr);
1107
1108                 if (!is_valid_ether_addr(ndev->dev_addr)) {
1109                         random_ether_addr(ndev->dev_addr);
1110                         pr_warn("%s: Use random mac address\n", ndev->name);
1111                 }
1112         }
1113 }
```
两种方式，优先从chipid获取，后再使用随机生成。

11. 常见一些文件节点，调试

```
root@TinaLinux:/sys/class/net# ls
br-lan  eth0    lo      sit0    wlan0
root@TinaLinux:/sys/class/net# cd eth0/
root@TinaLinux:/sys/devices/platform/soc@3000000/4500000.eth/net/eth0# ls
addr_assign_type      flags                 phys_switch_id
addr_len              gro_flush_timeout     power
address               ifalias               proto_down
broadcast             ifindex               queues
brport                iflink                speed
carrier               link_mode             statistics
carrier_changes       master                subsystem
carrier_down_count    mtu                   tx_queue_len
carrier_up_count      name_assign_type      type
dev_id                netdev_group          uevent
dev_port              operstate             upper_br-lan
device                phydev                waiting_for_supplier
dormant               phys_port_id
duplex                phys_port_name
```

总结一下：以太网的使用，首先需要理清楚当前自己硬件的框架，是单 phy 还是 mac 和 phy 集成的，甚至是类似双网口的多 mac 和 phy 集成。然后做好 soc 和 phy 乃至 switch 的软件初始

化，最后调整一下 delay 参数即可。

5 调试手段

5.1 常用命令

1. 打开/关闭网络设备

```
打开网络设备：ifconfig eth0 up
关闭网络设备：ifconfig eth0 down
```

2. 查看网络设备信息

```
查看网口状态：ifconfig eth0
查看收发包统计：cat /proc/net/dev
查看当前速率：cat /sys/class/net/eth0/speed
```

3. 配置网络设备

```
配置静态IP地址：ifconfig eth0 192.168.1.100
配置MAC地址：ifconfig eth0 hw ether 00:11:22:aa:bb:cc
动态获取IP地址：udhcpc -i eth0
PHY强制模式：ethtool -s eth0 speed 100 duplex full autoneg on（设置100Mbps速率、全双工、开启自协
    商）
```

4. 连通性测试

```
测试设备连通性：ping 192.168.1.100
```

5. 吞吐测试

```
TCP吞吐测试：
Server端：iperf -s -i 1
Client端：iperf -c 192.168.1.100 -i 1 -t 60 -P 4
UDP吞吐测试：
Server端：iperf -s -u -i 1
Client端：iperf -c 192.168.1.100 -u -b 100M -i 1 -t 60 -P 4
```

5.2 调试方法

5.2.1 软件排查方法

(1) 检查 phy mode 配置是否正确，如 rgmii、rmii 等;
(2) 检查 clk 配置是否正确，如 gmac clk、ephy_25m clk;
(3) 检查 GPIO 配置是否正确，如 IO 复用功能、驱动能力等;
(4) 检查 phy reset 配置是否正确;
(5) 通过 cat /proc/net/dev 命令查看 eth0 收发包统计情况;

5.2.2 硬件排查方法

(1) 检查 phy 供电（vcc-ephy）是否正常;
(2) 检查 phy 时钟波形是否正常;

6 常见问题

6.1 1.ifconfig 命令无 eth0 节点

```
问题现象：
执行 ifconfig eth0 无相关 log 信息
问题分析：
以太网模块配置未打开或存在 GPIO 冲突
排查步骤：
（1）抓取内核启动 log，检查 gmac 驱动 probe 是否成功；
（2）如果无 gmac 相关打印，请确认以太网基本配置是否打开；
（3）如果 gmac 驱动 probe 失败，请结合 log 定位具体原因，常见原因是GPIO 冲突导致；
```

6.2 2.ifconfig eth0 up 失败

```
问题现象：
执行 ifconfig eth0 up，出现 “Initialize hardware error” 或 “No phy found” 异常 log
如：
root@TinaLinux:/# ifconfig eth0 up
[84181.469933] libphy: 4500000.eth: probed
[84181.474230] sunxi-gmac 4500000.eth eth0: No PHY found!
ifconfig: SIOCSIFFLAGS: Invalid argument
问题分析：
常见原因是供给 phy 使用的 25M 时钟异常
排查步骤：
（1）检查软件 phy_mode 配置与板级情况一致；
（2）检查 phy 供电是否正常；
（3）若步骤 1 和步骤 2 正常，需重点检查 phy 使用的 25M 时钟（ephy25M 或外部晶振）是否正常；
```

6.3 3. 网络不通或网络丢包严重

```
问题现象：
ping 不通对端设备、无法动态获取 ip 地址或有丢包现象
问题分析：
一般原因是 tx/rx 通路不通
排查步骤：
（1）检查 ifconfig eth0 up 是否正常；
（2）检查 eth0 能否动态获取 ip 地址；
（3）若步骤 1 正常，但步骤 2 异常，需首先确认 tx/rx 哪条通路不通；
（4）若无法动态获取 ip 地址，可配置静态 IP，和对端设备互相 ping；
（5）检查对端设备能否收到数据包，若能收到，则说明 tx 通路正常，否则 tx 通路异常；
（6）检查本地设备能否收到数据包，若能收到，则说明 rx 通路正常，否则 rx 通路异常；
```

（7）若 tx 通路异常，可调整 tx-delay 参数或对照原理图检查 tx 通路是否异常，如漏焊关键器件；
（8）若 rx 通路异常，可调整 rx-delay 参数或对照原理图检查 rx 通路是否异常，如漏焊关键器件；
（9）若经过上述排查步骤问题仍未解决，需检查 phy 供电与 GPIO 耐压是否匹配；

6.4 4. 吞吐率异常

问题现象：
千兆网络吞吐率偏低，如小于 300Mbps
排查步骤：
（1）检查内核有无开启 CONFIG_SLUB_DEBUG_ON 宏，若有，则关闭此宏后再进行测试；
（2）如问题仍没有解决，请检查网络是否有丢包、错包现象。

著作权声明

商标声明

免责声明