《嵌入式系统》
（第十三讲）

厦门大学信息学院软件工程系 曾文华

2024年12月10日

- 第1章：嵌入式系统概述
- 第2章：ARM处理器和指令集
- 第3章：嵌入式Linux操作系统
- 第4章：嵌入式软件编程技术
- 第5章：开发环境和调试技术
- 第6章：Boot Loader技术
- 第7章：ARM Linux内核
- 第8章：文件系统
- 第9章：设备驱动程序设计基础
- 第10章：字符设备和驱动程序设计
- 第11章：Android操作系统（增加）
- 第12章：块设备和驱动程序设计
- 第13章：网络设备驱动程序开发
- 第14章：嵌入式GUI及应用程序设计

第13章　网络设备驱动程序开发

- 13.1　以太网基础知识
- 13.2　嵌入式网络设备驱动开发概述
- 13.3　网络设备驱动基本数据结构
- 13.4　网络设备初始化
- 13.5　打开和关闭接口
- 13.6　数据接收与发送
- 13.7　查询状态与参数设置
- 13.8　AT91SAM9G45网卡驱动

- 网络设备是Linux三大设备之一（另外两类是字符设备，块设备）。

- 由于网络设备的特殊工作方式，网络驱动程序的开发与字符设备、块设备驱动的开发有很大的不同。

13.1 以太网基础知识

- 以太网（Ethernet）：最早由Xerox（施乐）公司创建，于1980年DEC、Intel和Xerox三家公司联合开发成为一个标准。

- 以太网是应用最为广泛的局域网，包括标准的以太网（10Mbit/s）、快速以太网（100Mbit/s）和10G（10Gbit/s）以太网，采用的是CSMA/CD访问控制法，它们都符合IEEE 802.3标准。

  - CSMA/CD（Carrier Sense Multiple Access/Collision Detection，带有冲突检测的载波侦听多路存取）是IEEE 802.3使用的一种媒体访问控制方法。

  - IEEE 802.3标准：规定了包括物理层的连线、电信号和介质访问层协议的内容。

- 1、早期的以太网
  - 1Base—5：使用双绞线电缆，最大网段长度为500m，传输速度为1Mbps。
- 2、10Mb/s以太网
  - 10Base—T：使用双绞线电缆，最大网段长度为100m。拓扑结构为星型；10Base—T组网主要硬件设备有：3类或5类非屏蔽双绞线、带有RJ-45插口的以太网卡、集线器、交换机、RJ-45插头等。
- 3、快速以太网
  - 100BASE—TX：是一种使用5类数据级无屏蔽双绞线或屏蔽双绞线的快速以太网技术。它使用两对双绞线，一对用于发送，一对用于接收数据。
- 4、千兆以太网
  - 1000Base—T：带宽1Gb/s，使用超5类双绞线或6类双绞线。
- 5、万兆以太网
  - 10GBASE-SR 和 10GBASE-SW：主要支持短波（850 nm）多模光纤（MMF），光纤距离为 2m 到 300 m。
- 6、4万兆以太网和10万兆以太网
  - 标准尚在起草中，尚未发布。

- **13.1.1 CSMA/CD协议**

  - **CSMA/CD（Carrier Sense Multiple Access/Collision Detection，带有冲突检测的载波侦听多路存取）是IEEE 802.3使用的一种媒体访问控制方法。从逻辑上可以划分为两大部分：数据链路层的媒体访问控制子层（MAC）和物理层。它严格对应于ISO开放系统互连模式的最低两层。LLC子层和MAC子层在一起完成OSI模式的数据链路层的功能。**

    - **CS：载波侦听，Carrier Sense**
    - **MA：多点接入（多路存取），Multiple Access**
    - **CD：冲突检测，Collision Detection**

- CSMA/CD的基本原理是：所有节点都共享网络传输信道，节点在发送数据之前，首先检测信道是否空闲，如果信道空闲则发送，否则就等待；在发送出信息后，再对冲突进行检测，当发现冲突时，则取消发送。

- CSMA/CD协议的侦听发送策略有三种：
  ① 非坚持CSMA
  ② 1-坚持CSMA
  ③ p-坚持CSMA

- CSMA/CD的控制规程主要包括以下四个处理内容：
  ① 侦听
  ② 数据发送
  ③ 冲突检测
  ④ 冲突处理

- **13.1.2 以太网帧结构**

  – 以太网帧结构（帧格式），即在以太网帧头、帧尾中用于实现以太网功能的域。在以太网的帧头和帧尾中有几个用于实现以太网功能的域，每个域也称为字段，有其特定的名称和目的。

  ① **Ethernet V2**：ARPA，由DEC，Intel和Xerox在1982年公布其标准。

  ② **Ethernet 802.3 RAW**：这是1983年Novell发布其划时代的Netware/86网络套件时采用的私有以太网帧格式。

  ③ **Ethernet 802.3/802.2 SNAP**：这是IEEE为保证在802.2 LLC上支持更多的上层协议同时更好的支持IP协议而发布的标准。

- **13.1.3 嵌入式系统中常用的网络协议**

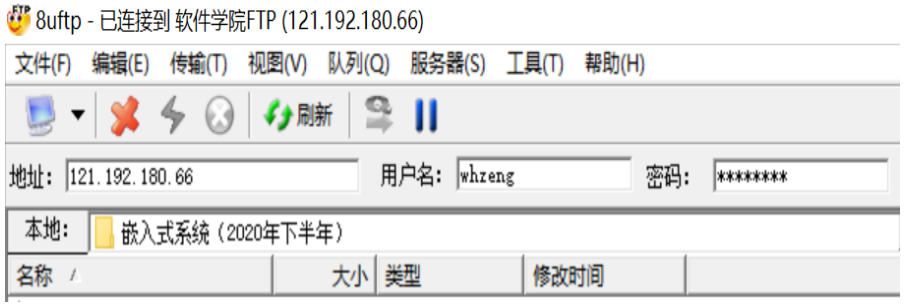
  - **ARP：地址解析协议，Address Resolution Protocol，是根据IP地址获取物理地址的一个TCP/IP协议。**
  - **RARP：反向地址转换协议，Reverse Address Resolution Protocol，允许局域网的物理机器从网关服务器的 ARP 表或者缓存上请求其 IP 地址。**
  - **IP：网际协议，Internet Protocol，网际协议也就是为计算机网络相互连接进行通信而设计的协议。**
  - **ICMP：Internet Control Message Protocol，互联网控制消息协议。它是TCP/IP协议族的一个子协议，用于在IP主机、路由器之间传递控制消息。**

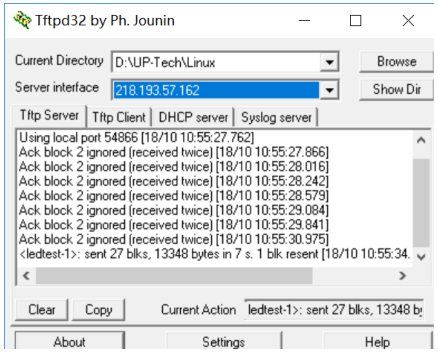
- **TCP：Transmission Control Protocol，传输控制协议。**是一种面向连接（连接导向）的、可靠的、基于字节流的运输层（Transport layer）通信协议，由IETF的RFC 793说明（specified）。

- **UDP：User Datagram Protocol，用户数据报协议，**是OSI（Open System Interconnection，开放式系统互联）参考模型中一种无连接的传输层协议，提供面向事务的简单不可靠信息传送服务，IETF RFC 768是UDP的正式规范。

- **FTP：File Transfer Protocol，文件传输协议。**是 TCP/IP 协议组中的协议之一。FTP协议包括两个组成部分，其一为FTP服务器，其二为FTP客户端。

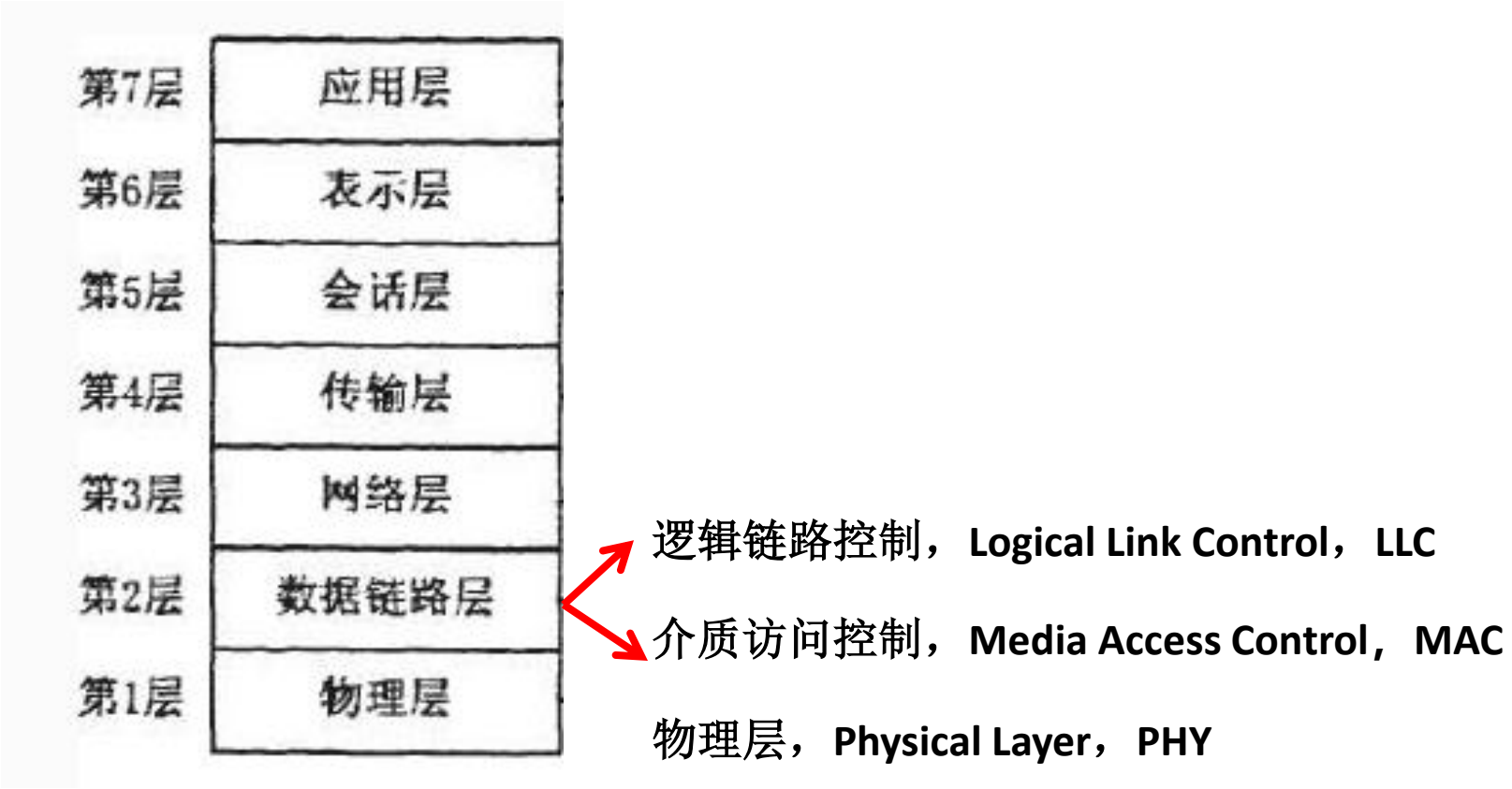
- **TFTP：Trivial File Transfer Protocol，简单文件传输协议。**是TCP/IP协议族中的一个用来在客户机与服务器之间进行简单文件传输的协议，提供不复杂、开销不大的文件传输服务，端口号为69。





13.2 嵌入式网络设备驱动开发概述

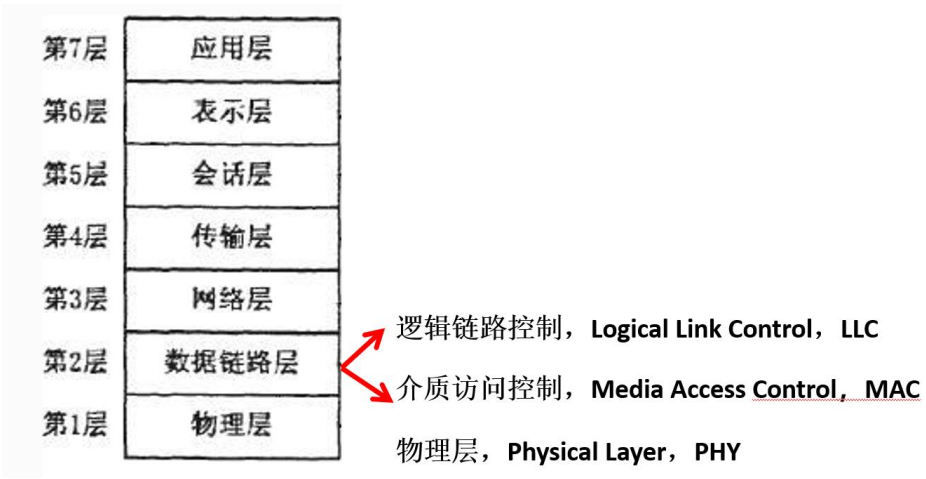
- 1、硬件描述
  - 以太网对应于ISO网络分层中的数据链路层和物理层



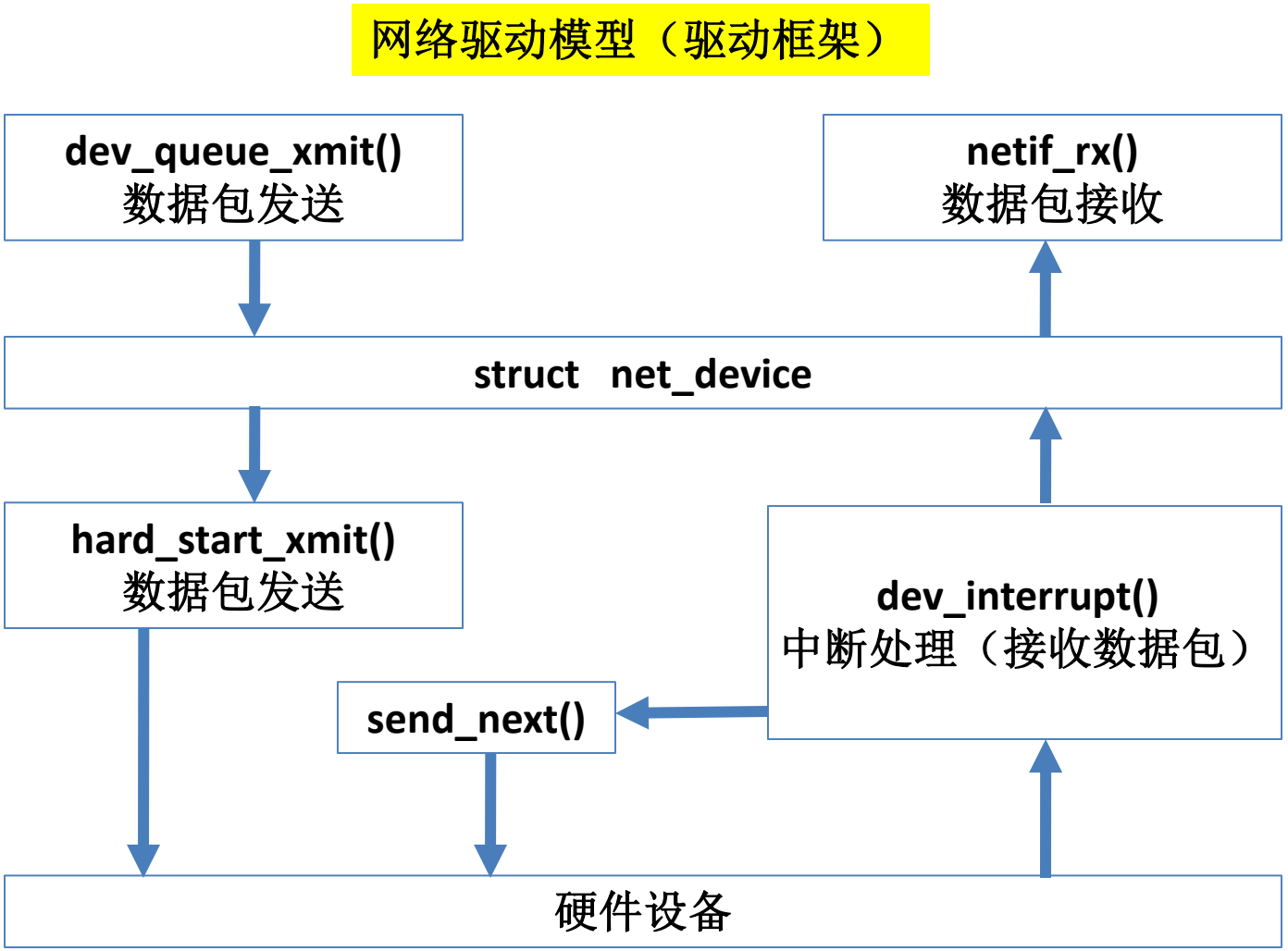
– 使用嵌入式以太网接口的两种方式：

① 嵌入式处理器集成MAC控制器，但是不集成物理层接收器（PHY），此时需要外接PHY芯片，如RTL8201BL、VT6103等芯片。

② 嵌入式处理器既不集成MAC控制器，又不集成物理层接收器（PHY），此时需要外接同时具有MAC控制器和PHY接收器的芯片，如DM9000、CS8900、SIS900等芯片。



- **2、驱动框架**
  - 在/dev目录下没有对应的设备文件。
  - 对网络设备的访问必须使用套接字（Socket），而非读写设备文件。



13.3 网络设备驱动基本数据结构

- **struct net_device**（网络设备结构体）：包含具体网络设备的各种信息和操作接口。

- **struct sk_buff**（套接字缓冲结构体）：是Linux网络子系统的核心，在Linux网络子系统各层协议中的数据传递实际上传递的是struct sk_buff结构，它为各层之间的数据交换单元提供了统一的定义。

```
struct net_device
{
        char                    name[IFNAMSIZ];
        unsigned long           rmem_end; /* shmem "recv" end     */
        unsigned long           rmem_start;         /* shmem "recv" start   */
        unsigned long           mem_end;  /* shared mem end       */
        unsigned long           mem_start; /* shared mem start    */
        unsigned long           base_addr;  /* device I/O address  */
        unsigned int            irq;                /* device IRQ number    */
        unsigned char           if_port;    /* Selectable AUI, TP,..*/
        unsigned char           dma;                /* DMA channel                  */
        unsigned long           state;
        struct net_device    *next;
        int                     (*init)(struct net_device *dev);
        struct net_device    *next_sched;
        int                     ifindex;
        int                     iflink;
```

struct net_device
网络设备结构体

```
struct net_device_stats* (*get_stats)(struct net_device *dev);
struct iw_statistics*    (*get_wireless_stats)(struct net_device *dev);
unsigned long            trans_start;          /* Time (in jiffies) of last Tx   */
unsigned long            last_rx;    /* Time of last Rx   */
unsigned short           flags;      /* interface flags (a la BSD)   */
unsigned short           gflags;
 unsigned short      priv_flags; /* Like 'flags' but invisible to userspace. */
 unsigned short      unused_alignment_fixer; /* Because we need priv_flags,
unsigned                 mtu;        /* interface MTU value                  */
unsigned short           type;       /* interface hardware type    */
unsigned short           hard_header_len;   /* hardware hdr length        */
void                     *priv;      /* pointer to private data      */
struct net_device        *master; /* Pointer to master device of a group,
unsigned char            broadcast[MAX_ADDR_LEN];              /* hw bcast add|
unsigned char            dev_addr[MAX_ADDR_LEN];/* hw address        */
unsigned char            addr_len;/* hardware address length   */
```

struct net_device
网络设备结构体

```
struct dev_mc_list *mc_list;    /* Multicast mac addresses          */
int                             mc_count;  /* Number of installed mcasts       */
int                             promiscuity;
int                             allmulti;
int                             watchdog_timeo;
struct timer_list    watchdog_timer;
void                            *atalk_ptr; /* AppleTalk link          */
void                            *ip_ptr;      /* IPv4 specific data     */
void              *dn_ptr;      /* DECnet specific data */
void              *ip6_ptr;     /* IPv6 specific data */
void                            *ec_ptr;      /* Econet specific data */
struct Qdisc                    *qdisc;
struct Qdisc                    *qdisc_sleeping;
struct Qdisc                    *qdisc_list;
struct Qdisc                    *qdisc_ingress;
unsigned long                   tx_queue_len;           /* Max frames per queue allowed */
spinlock_t                      xmit_lock;
int                             xmit_lock_owner;
spinlock_t                      queue_lock;
atomic_t                        refcnt;
int                             deadbeaf;
int                             features;
```

struct net_device
网络设备结构体

```
#define NETIF_F_SG              1       /* Scatter/gather IO. */
#define NETIF_F_IP_CSUM                 2       /* Can checksum only TCP/UDP over IPv4. */
#define NETIF_F_NO_CSUM                 4       /* Does not require checksum. F.e. loopack. */
#define NETIF_F_HW_CSUM                 8       /* Can checksum all the packets. */
#define NETIF_F_DYNALLOC        16      /* Self-dectructable device. */
#define NETIF_F_HIGHDMA                 32      /* Can DMA to high memory. */
#define NETIF_F_FRAGLIST        64      /* Scatter/gather IO. */
    void                (*uninit)(struct net_device *dev);
    void                (*destructor)(struct net_device *dev);
    int                 (*open)(struct net_device *dev);
    int                 (*stop)(struct net_device *dev);
    int                 (*hard_start_xmit) (struct sk_buff *skb,
                                            struct net_device *dev);
    int                 (*hard_header) (struct sk_buff *skb,
                                        struct net_device *dev,
                                        unsigned short type,
                                        void *daddr,
                                        void *saddr,
                                        unsigned len);
```

struct net_device
网络设备结构体

```
int                     (*rebuild_header)(struct sk_buff *skb);
#define HAVE_MULTICAST
    void                (*set_multicast_list)(struct net_device *dev);
#define HAVE_SET_MAC_ADDR
    int                 (*set_mac_address)(struct net_device *dev,
                                           void *addr);
#define HAVE_PRIVATE_IOCTL
    int                 (*do_ioctl)(struct net_device *dev,
                                    struct ifreq *ifr, int cmd);
#define HAVE_SET_CONFIG
    int                 (*set_config)(struct net_device *dev,
                                      struct ifmap *map);
#define HAVE_HEADER_CACHE
    int                 (*hard_header_cache)(struct neighbour *neigh,
                                             struct hh_cache *hh);
    void                (*header_cache_update)(struct hh_cache *hh,
                                               struct net_device *dev,
                                               unsigned char *  haddr);
```

struct net_device
网络设备结构体

```
#define HAVE_CHANGE_MTU
        int                             (*change_mtu)(struct net_device *dev, int new_mtu);
#define HAVE_TX_TIMEOUT
        void                            (*tx_timeout) (struct net_device *dev);

        int                             (*hard_header_parse)(struct sk_buff *skb,
                                                             unsigned char *haddr);
        int                             (*neigh_setup)(struct net_device *dev, struct neigh_parms *);
        int                             (*accept_fastpath)(struct net_device *, struct dst_entry*);
        struct module *owner;
        struct net_bridge_port          *br_port;
#ifdef CONFIG_NET_FASTROUTE
#define NETDEV_FASTROUTE_HMASK 0xF
        rwlock_t                        fastpath_lock;
        struct dst_entry        *fastpath[NETDEV_FASTROUTE_HMASK+1];
#endif
#ifdef CONFIG_NET_DIVERT
        struct divert_blk       *divert;
#endif /* CONFIG_NET_DIVERT */
};
```

struct net_device
网络设备结构体

```
struct sk_buff {
    struct sk_buff              *next;
    struct sk_buff              *prev;

    struct sock                 *sk;
    ktime_t                     tstamp;
    struct net_device   *dev;

    struct  dst_entry   *dst;
    struct   sec_path   *sp;

    char                        cb[48];

    unsigned int                len,
                                data_len;

    __u16                       mac_len,
                                hdr_len;

    union {
            __wsum              csum;
            struct {
                        __u16   csum_start;
                        __u16   csum_offset;
            };
    };
```

struct sk_buff
套接字缓冲结构体

```
    __u32           priority;
    __u8                    local_df:1,
                            cloned:1,
                            ip_summed:2,
                            nohdr:1,
                            nfctinfo:3;
    __u8                    pkt_type:3,
                            fclone:2,
                            ipvs_property:1,
                            nf_trace:1;
    __be16                  protocol;
    void                    (*destructor)(struct sk_buff *skb);
#if defined(CONFIG_NF_CONNTRACK) || defined(CONFIG_NF_CONNTRACK_MODULE)
    struct nf_conntrack     *nfct;
    struct sk_buff          *nfct_reasm;
#endif
```

struct sk_buff
套接字缓冲结构体

```
#ifdef CONFIG_BRIDGE_NETFILTER
    struct nf_bridge_info       *nf_bridge;
#endif
    int                         iif;
#ifdef CONFIG_NETDEVICES_MULTIQUEUE
    __u16                       queue_mapping;
#endif
#ifdef CONFIG_NET_SCHED
    __u16                       tc_index;
#ifdef CONFIG_NET_CLS_ACT
    __u16                       tc_verd;
#endif
#endif
```

struct sk_buff
套接字缓冲结构体

```
#ifdef CONFIG_NET_DMA
        dma_cookie_t            dma_cookie;
#endif
#ifdef CONFIG_NETWORK_SECMARK
        __u32                   secmark;
#endif
        __u32                   mark;
        sk_buff_data_t          transport_header;
        sk_buff_data_t          network_header;
        sk_buff_data_t          mac_header;
        sk_buff_data_t          tail;
        sk_buff_data_t          end;
        unsigned char           *head,      *data;
        unsigned int            truesize;
        atomic_t                users;
};
```

struct sk_buff
套接字缓冲结构体

- **13.3.1 struct net_device数据结构**

struct net_device
网络设备结构体

  - 1、全局信息

    - 网络设备的名称
      - **char name[IFNAMSIZ];**

    - 设备初始化函数指针
      - **int (*init)(struct net_device *dev);**

– 2、硬件信息

- 设备使用的共享内存的起始地址
  - **unsigned long mem_end;**
- 设备使用的共享内存的终止地址
  - **unsigned long mem_start;**
- 设备I/O基址
  - **unsigned long base_addr;**
- 设备中断号
  - **unsigned int irq;**
- 指定多端口设备使用哪个端口
  - **unsigned char if_port;**
- 保存分配给设备的DMA通道
  - **unsigned char dma;**
- 设备状态
  - **unsigned long state;**

– 3、接口信息

- 保存最大传输单元大小
  - **unsigned mtu;**
- 接口类型
  - **unsigned short type;**
- 硬件头长度
  - **unsigned short hard_header_len;**
- 存放设备硬件地址（**MAC**地址）
  - **unsigned char dev_addr[MAX_ADDR_LEN];**
- 存放广播地址
  - **unsigned char broadcast[MAX_ADDR_LEN];**
- 接口标志
  - **unsigned short flags;**

– 4、设备操作函数

- 打开接口
  - **int(*open)(struct net_device *dev);**
- 关闭接口
  - **int(*stop)(struct net_device *dev);**
- 启动数据包的发送
  - **int (*hard_start_xmit) (struct sk_buff *skb, struct net_device *dev);**
- 设置设备的组播列表
  - **void (*set_multicast_list)(struct net_device *dev);**
- 设置设备的MAC地址
  - **int(*set_mac_address)(struct net_device *dev, void *addr);**

- 执行接口特有的I/O控制命令
  - int(*do_ioctl)(struct net_device *dev, struct ifreq *ifr, int cmd);
- 改变接口的配置
  - int (*set_config)(struct net_device *dev,   struct ifmap *map);
- 在接口的MTU改变时，采取相应的设置
  - int (*change_mtu)(struct net_device *dev, int new_mtu);
- 如果数据包传送超时，则调用该函数解决问题并重新发送数据
  - void (*tx_timeout) (struct net_device *dev);
- 用于返回设备状态信息
  - struct net_device_stats* (*get_stats)(struct net_device *dev);
- 在禁止中断的情况下，要求驱动程序检测接口下的事件
  - viod (*poll_controller)(struct net_devices *dev);

– 5、工具成员

- 记录数据最后一次接收到数据包的时间戳
  - **unsigned long last_rx;**
- 记录数据开始发送时的时间戳
  - **unsigned long trans_start;**
- 在网络层驱动传输已超时，并调用**tx_timout**之前的最小时间
  - **intwatchdog_timeo;**

- **13.3.2 struct sk_buff数据结构**

struct sk_buff
套接字缓冲结构体

  - **1、网络协议头**
    - **sk_buff_data_t transport_header;**
    - **sk_buff_data_t network_header;**
    - **sk_buff_data_t mac_header;**
  - **2、缓冲区指针**
    - **sk_buff_data_t tail;**
    - **sk_buff_data_t end;**
    - **unsigned char *head, *data;**

– 3、操作函数

- **static inline struct sk_buff *alloc_skb(unsigned int size, gfp_t priority);**
- **static inline struct sk_buff *dev_alloc_skb(unsigned int length);**
- **void kfree_skb(struct sk_buff *skb);**
- **void kfree_skb(struct sk_buff *skb);**
- **void dev_kfree_skb(struct sk_buff *skb);**
- **void dev_kfree_skb_irq(struct sk_buff *skb);**
- **void dev_kfree_skb_any(struct sk_buff *skb);**
- **unsigned char *skb_put(struct sk_buff *skb, unsigned int len);**
- **unsigned char *__skb_put(struct sk_buff *skb, unsigned int len);**
- **unsigned char *skb_push(struct sk_buff *skb, unsigned int len);**
- **unsigned char *__skb_push(struct sk_buff *skb, unsigned int len);**
- **int skb_tailroom(const struct sk_buff *skb);**
- **unsigned int skb_headroom(const struct sk_buff *skb)**
- **void skb_reserve(struct sk_buff *skb, int len);**
- **char *skb_pull_rcsum(struct sk_buff *skb, unsigned int len);**

13.4 网络设备初始化

- 网络设备初始化由struct net_device数据结构的init函数指针指向的函数完成：
  - int (*init)(struct net_device *dev);

- 初始化工作包括以下几个方面的任务：
  ① 检测网络设备的硬件特征，检查物理设备是否存在；

  ② 如果检测到网络设备存在，则进行资源配置；

  ③ 对struct net_device成员变量进行赋值。

13.5 打开和关闭接口

- 打开接口的工作由struct net_device数据结构的open函数指针指向的函数完成：
  - int (*open)(struct net_device *dev);
- 该函数负责的工作包括请求系统资源，如申请I/O区域、DMA通道及中断等资源，并告知接口开始工作，调用netif_start_queue函数激活网络设备发送队列：
  - void netif_start_queue(struct net_device *dev);

  打开队列
- 关闭接口的工作由struct net_device数据结构的stop函数指针指向的函数完成：
  - int (*stop)(struct net_device *dev);
- 该函数需要调用netif_stop_queue函数停止数据包传送：
  - void netif_stop_queue(struct net_device *dev);

  停止队列

13.6 数据接收与发送

- 1、数据发送
  - 数据在实际发送的时候会调用struct net_device数据结构的hard_start_transmit函数指针指向的函数，该函数会将要发送的数据放入外发队列，并启动数据包发送。
- 2、并发控制
  - 发送函数可利用struct net_device数据结构的xmit_lock自旋锁来保护临界区资源。

- **3、传输超时**
  - 驱动程序需要处理超时带来的问题，调用struct net_device数据结构的tx_timeout函数，并调用netif_wake_queue函数重启设备发送队列。

- **4、数据接收**
  - 在Linux中有两种方式实现数据接收：
    - 第一种是中断方式：调用netif_rx函数实现数据接收；
    - 第二种是轮询方式：调用netif_receive_skb函数实现数据接收。

13.7 查询状态与参数设置

- 1、链路状态
  - 驱动程序需要掌握当前链路的状态，当链路状态改变时，驱动程序需要通知内核：
    - void netif_carrier_off(struct net_device *dev);
    - void netif_carrier_on(struct net_device *dev);
    - int netif_carrier_ok(const struct net_device *dev);

- 2、设备状态
  - 驱动程序的get_stats()函数用于向用户返回设备的状态和统计信息，这些信息保存在struct net_device_stats结构体中。

```
struct net_device_stats
{
    unsigned long   rx_packets;         /* total packets received       */
    unsigned long   tx_packets;         /* total packets transmitted    */
    unsigned long   rx_bytes;           /* total bytes received */
    unsigned long   tx_bytes;           /* total bytes transmitted      */
    unsigned long   rx_errors;          /* bad packets received                 */
    unsigned long   tx_errors;          /* packet transmit problems     */
    unsigned long   rx_dropped;                 /* no space in linux buffers    */
    unsigned long   tx_dropped;                 /* no space available in linux  */
    unsigned long   multicast;          /* multicast packets received   */
    unsigned long   collisions;
    unsigned long   rx_length_errors;
    unsigned long   rx_over_errors;             /* receiver ring buff overflow  */
    unsigned long   rx_crc_errors;              /* recved pkt with crc error    */
    unsigned long   rx_frame_errors;    /* recv'd frame alignment error */
    unsigned long   rx_fifo_errors;             /* recv'r fifo overrun          */
    unsigned long   rx_missed_errors;   /* receiver missed packet       */
    unsigned long   tx_aborted_errors;
    unsigned long   tx_carrier_errors;
    unsigned long   tx_fifo_errors;
    unsigned long   tx_heartbeat_errors;
    unsigned long   tx_window_errors;
    unsigned long   rx_compressed;
    unsigned long   tx_compressed;
};
```

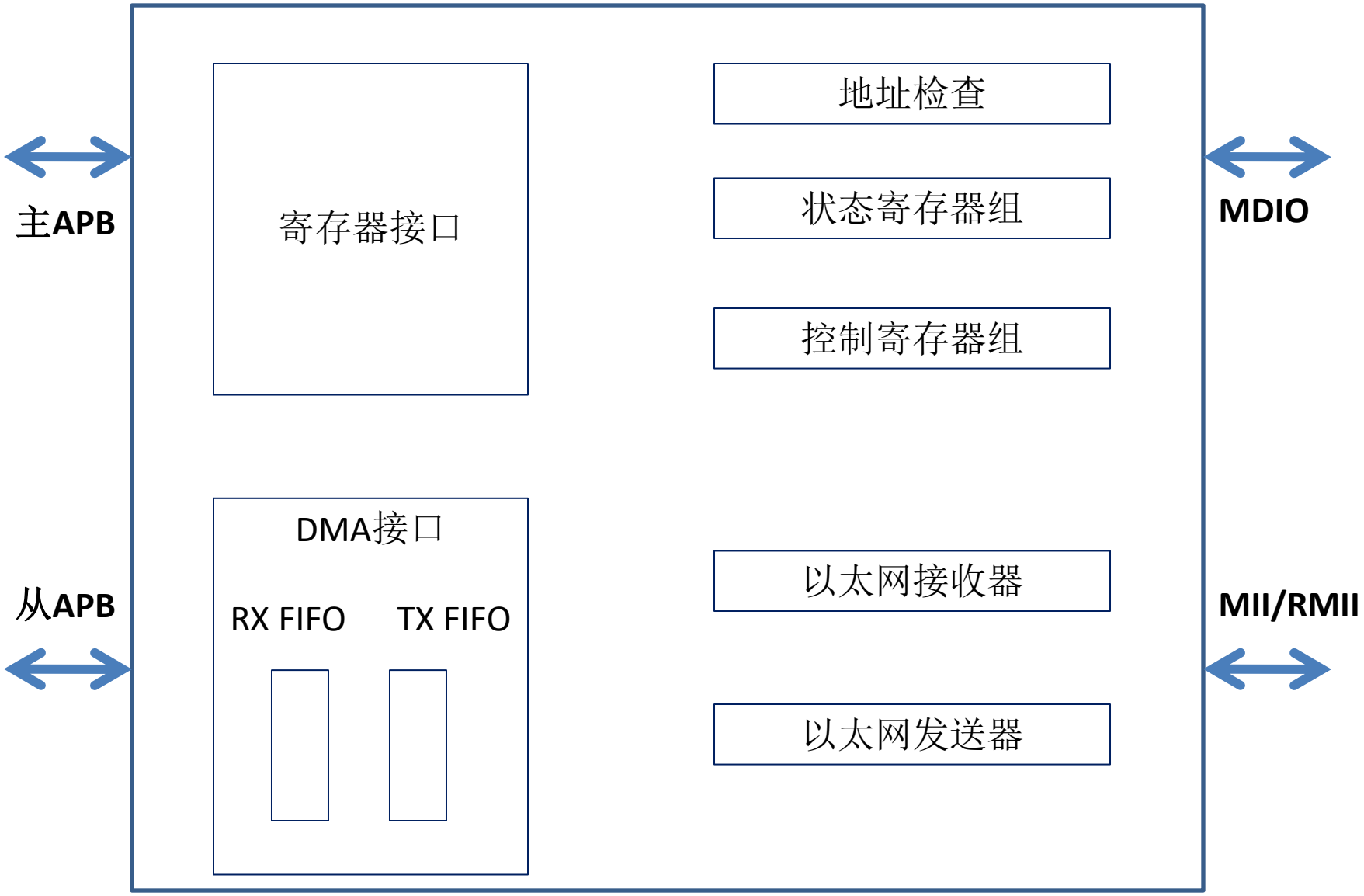
struct net_device_stats
网络设备状态结构体

- **3、设置MAC地址**
  - 调用**set_mac_address**函数，设置新的**MAC**地址。
- **4、接口参数设置**
  - 调用**set_config**函数，设置**I/O**地址、中断等信息。

13.8 AT91SAM9G45网卡驱动

- 13.8.1 EMAC模块简介
  - AT91SAM9G45：嵌入式微处理器
  - AT91SAM9G45的EMAC模块是一个完全兼容IEEE 802.3标准的10/100M的以太网控制器，它包含一个地址检查模块、统计和控制寄存器组、接收和发送模块，以及一个DMA接口。
- 13.8.2 模块图

AT91SAM9G45嵌入式微处理器的EMAC模块



• 13.8.3 功能描述

- 1、时钟
  - 系统总线时钟
  - 发送时钟
  - 接收时钟
- 2、内存接口
  - 帧数据在内存和EMAC之间的传输是通过DMA接口实现的。
- 3、接收模块
  - 接收模块检查以太网帧的前导帧、FCS、对齐和长度。
- 4、发送模块
  - 发送模块从DMA接口获取数据，填充前导帧、FCS等字段。

- **13.8.4 寄存器描述**
  - 1、网络控制寄存器（NCR）
  - 2、网络配置寄存器（NCFG）
  - 3、网络状态寄存器（NSR）
  - 4、发送状态寄存器（TSR）
  - 5、接收缓冲队列指针寄存器（RBQP）
  - 6、发送缓冲队列指针寄存器（TBQP）
  - 7、接收状态寄存器（RSR）
  - 8、中断状态寄存器（ISR）

- **13.8.5 AT91SAM9G45芯片EMAC控制器驱动分析**

  - **1、设备侦测**

    - 设备侦测主要完成各种资源的初始化工作：

      ① 获取I/O内存的地址，并进行I/O重定向；

      ② 获取设备中断号，注册中断处理程序；

      ③ 初始化net_device结构，并注册该结构；

      ④ 初始化时钟，并设置分频器。

```
static int __devinit macb_probe(struct platform_device *pdev)
{
        struct eth_platform_data *pdata;
        struct resource *regs;
        struct net_device *dev;
        struct macb *bp;
        struct phy_device *phydev;
        unsigned long pclk_hz;
        u32 config;
        int err = -ENXIO;
        DECLARE_MAC_BUF(mac);

        regs = platform_get_resource(pdev, IORESOURCE_MEM, 0);
        if (!regs) {
                dev_err(&pdev->dev, "no mmio resource defined\n");
                goto err_out;
        }
```

设备侦测
macb_probe

```
        err = -ENOMEM;
        dev = alloc_etherdev(sizeof(*bp));
        if (!dev) {
                dev_err(&pdev->dev, "etherdev alloc failed, aborting.\n");
                goto err_out;
        }

        SET_NETDEV_DEV(dev, &pdev->dev);

        dev->features |= 0;

        bp = netdev_priv(dev);
        bp->pdev = pdev;
        bp->dev = dev;

        spin_lock_init(&bp->lock);

#if defined(CONFIG_ARCH_AT91)
        bp->pclk = clk_get(&pdev->dev, "macb_clk");
        if (IS_ERR(bp->pclk)) {
                dev_err(&pdev->dev, "failed to get macb_clk\n");
                goto err_out_free_dev;
        }
        clk_enable(bp->pclk);
#else
```

设备侦测
macb_probe

```
    bp->pclk = clk_get(&pdev->dev, "pclk");
    if (IS_ERR(bp->pclk)) {
            dev_err(&pdev->dev, "failed to get pclk\n");
            goto err_out_free_dev;
    }
    bp->hclk = clk_get(&pdev->dev, "hclk");
    if (IS_ERR(bp->hclk)) {
            dev_err(&pdev->dev, "failed to get hclk\n");
            goto err_out_put_pclk;
    }

    clk_enable(bp->pclk);
    clk_enable(bp->hclk);
#endif

    bp->regs = ioremap(regs->start, regs->end - regs->start + 1);
    if (!bp->regs) {
            dev_err(&pdev->dev, "failed to map registers, aborting.\n");
            err = -ENOMEM;
            goto err_out_disable_clocks;
    }
```

设备侦测
macb_probe

```
dev->irq = platform_get_irq(pdev, 0);
err = request_irq(dev->irq, macb_interrupt, IRQF_SAMPLE_RANDOM,
                  dev->name, dev);
if (err) {
        printk(KERN_ERR
            "%s: Unable to request IRQ %d (error %d)\n",
            dev->name, dev->irq, err);
        goto err_out_iounmap;
}

dev->open = macb_open;
dev->stop = macb_close;
dev->hard_start_xmit = macb_start_xmit;
dev->get_stats = macb_get_stats;
dev->set_multicast_list = macb_set_rx_mode;
dev->do_ioctl = macb_ioctl;
netif_napi_add(dev, &bp->napi, macb_poll, 64);
dev->ethtool_ops = &macb_ethtool_ops;

dev->base_addr = regs->start;
```

设备侦测
macb_probe

```
        pclk_hz = clk_get_rate(bp->pclk);
        if (pclk_hz <= 20000000)
                config = MACB_BF(CLK, MACB_CLK_DIV8);
        else if (pclk_hz <= 40000000)
                config = MACB_BF(CLK, MACB_CLK_DIV16);
        else if (pclk_hz <= 80000000)
                config = MACB_BF(CLK, MACB_CLK_DIV32);
        else
                config = MACB_BF(CLK, MACB_CLK_DIV64);
        macb_writel(bp, NCFGR, config);

        macb_get_hwaddr(bp);
        pdata = pdev->dev.platform_data;

        if (pdata && pdata->is_rmii)
#if defined(CONFIG_ARCH_AT91)
                macb_writel(bp, USRIO, (MACB_BIT(RMII) | MACB_BIT(CLKEN)) );
#else
                macb_writel(bp, USRIO, 0);
#endif
        else
#if defined(CONFIG_ARCH_AT91)
                macb_writel(bp, USRIO, MACB_BIT(CLKEN));
#else
                macb_writel(bp, USRIO, MACB_BIT(MII));
#endif

        bp->tx_pending = DEF_TX_RING_PENDING;

        err = register_netdev(dev);
```

设备侦测
macb_probe

```
if (err) {
        dev_err(&pdev->dev, "Cannot register net device, aborting.\n");
        goto err_out_free_irq;
}

if (macb_mii_init(bp) != 0) {
        goto err_out_unregister_netdev;
}

platform_set_drvdata(pdev, dev);

printk(KERN_INFO "%s: Atmel MACB at 0x%08lx irq %d "
     "(%s)\n",
     dev->name, dev->base_addr, dev->irq,
     print_mac(mac, dev->dev_addr));

phydev = bp->phy_dev;
printk(KERN_INFO "%s: attached PHY driver [%s] "
        "(mii_bus:phy_addr=%s, irq=%d)\n",
        dev->name, phydev->drv->name, phydev->dev.bus_id, phydev->irq);

return 0;
```

设备侦测
macb_probe

```
err_out_unregister_netdev:
        unregister_netdev(dev);
err_out_free_irq:
        free_irq(dev->irq, dev);
err_out_iounmap:
        iounmap(bp->regs);
err_out_disable_clocks:
#ifndef CONFIG_ARCH_AT91
        clk_disable(bp->hclk);
        clk_put(bp->hclk);
#endif
        clk_disable(bp->pclk);
#ifndef CONFIG_ARCH_AT91
err_out_put_pclk:
#endif
        clk_put(bp->pclk);
err_out_free_dev:
        free_netdev(dev);
err_out:
        platform_set_drvdata(pdev, NULL);
        return err;
}
```

设备侦测
macb_probe

– 2、设备打开与关闭

- 设备打开时主要完成以下任务：

① 分配DMA缓冲区，初始化缓冲区；

② 初始化硬件；

③ 打开PHY（物理层）；

④ 通知上层开启传输。

```
static int macb_open(struct net_device *dev)
{
    struct macb *bp = netdev_priv(dev);
    int err;

    dev_dbg(&bp->pdev->dev, "open\n");

    if (!bp->phy_dev)
        return -EAGAIN;

    if (!is_valid_ether_addr(dev->dev_addr))
        return -EADDRNOTAVAIL;

    err = macb_alloc_consistent(bp);
    if (err) {
        printk(KERN_ERR
            "%s: Unable to allocate DMA memory (error %d)\n",
            dev->name, err);
        return err;
    }
```

设备打开
macb_open

```
    napi_enable(&bp->napi);

    macb_init_rings(bp);
    macb_init_hw(bp);

    phy_start(bp->phy_dev);

    netif_start_queue(dev);

    return 0;
}
```

设备打开
macb_open

```
static int macb_close(struct net_device *dev)
{
    struct macb *bp = netdev_priv(dev);
    unsigned long flags;

    netif_stop_queue(dev);
    napi_disable(&bp->napi);

    if (bp->phy_dev)
        phy_stop(bp->phy_dev);

    spin_lock_irqsave(&bp->lock, flags);
    macb_reset_hw(bp);
    netif_carrier_off(dev);
    spin_unlock_irqrestore(&bp->lock, flags);

    macb_free_consistent(bp);

    return 0;
}
```

设备关闭
macb_close

- 3、数据的接收
- 4、数据的发送
- 5、中断处理

```
static int macb_rx_frame(struct macb *bp, unsigned int first_frag,
                         unsigned int last_frag)
{
    unsigned int len;
    unsigned int frag;
    unsigned int offset = 0;
    struct sk_buff *skb;

    len = MACB_BFEXT(RX_FRMLEN, bp->rx_ring[last_frag].ctrl);

    dev_dbg(&bp->pdev->dev, "macb_rx_frame frags %u - %u (len %u)\n",
            first_frag, last_frag, len);

    skb = dev_alloc_skb(len + RX_OFFSET);
    if (!skb) {
        bp->stats.rx_dropped++;
        for (frag = first_frag; ; frag = NEXT_RX(frag)) {
            bp->rx_ring[frag].addr &= ~MACB_BIT(RX_USED);
            if (frag == last_frag)
                break;
        }
        wmb();
        return 1;
    }
```

数据接收
macb_rx_frame

```
        skb_reserve(skb, RX_OFFSET);
        skb->ip_summed = CHECKSUM_NONE;
        skb_put(skb, len);

        for (frag = first_frag; ; frag = NEXT_RX(frag)) {
                unsigned int frag_len = RX_BUFFER_SIZE;

                if (offset + frag_len > len) {
                        BUG_ON(frag != last_frag);
                        frag_len = len - offset;
                }
                skb_copy_to_linear_data_offset(skb, offset,
                                                (bp->rx_buffers +
                                                 (RX_BUFFER_SIZE * frag)),
                                                frag_len);
                offset += RX_BUFFER_SIZE;
                bp->rx_ring[frag].addr &= ~MACB_BIT(RX_USED);
                wmb();

                if (frag == last_frag)
                        break;
        }

        skb->protocol = eth_type_trans(skb, bp->dev);

        bp->stats.rx_packets++;
        bp->stats.rx_bytes += len;
        bp->dev->last_rx = jiffies;
        dev_dbg(&bp->pdev->dev, "received skb of length %u, csum: %08x\n",
                skb->len, skb->csum);
        netif_receive_skb(skb);

        return 0;
}
```

数据接收
macb_rx_frame

```
static void macb_tx(struct macb *bp)
{
        unsigned int tail;
        unsigned int head;
        u32 status;

        status = macb_readl(bp, TSR);
        macb_writel(bp, TSR, status);

        dev_dbg(&bp->pdev->dev, "macb_tx status = %02lx\n",
                (unsigned long)status);

        if (status & MACB_BIT(UND)) {
                int i;
                printk(KERN_ERR "%s: TX underrun, resetting buffers\n",
                                bp->dev->name);

                head = bp->tx_head;

                /*Mark all the buffer as used to avoid sending a lost buffer*/
                for (i = 0; i < TX_RING_SIZE; i++)
                                bp->tx_ring[i].ctrl = MACB_BIT(TX_USED);

                /* free transmit buffer in upper layer*/
```

数据发送
macb_tx

```
        for (tail = bp->tx_tail; tail != head; tail = NEXT_TX(tail)) {
                struct ring_info *rp = &bp->tx_skb[tail];
                struct sk_buff *skb = rp->skb;

                BUG_ON(skb == NULL);

                rmb();

                dma_unmap_single(&bp->pdev->dev, rp->mapping, skb->len,
                                                                DMA_TO_DEVICE);
                rp->skb = NULL;
                dev_kfree_skb_irq(skb);
        }

        bp->tx_head = bp->tx_tail = 0;
}

if (!(status & MACB_BIT(COMP)))
        /*
         * This may happen when a buffer becomes complete
         * between reading the ISR and scanning the
         * descriptors.  Nothing to worry about.
         */
        return;
```

数据发送
macb_tx

```
        head = bp->tx_head;
        for (tail = bp->tx_tail; tail != head; tail = NEXT_TX(tail)) {
                struct ring_info *rp = &bp->tx_skb[tail];
                struct sk_buff *skb = rp->skb;
                u32 bufstat;

                BUG_ON(skb == NULL);

                rmb();
                bufstat = bp->tx_ring[tail].ctrl;

                if (!(bufstat & MACB_BIT(TX_USED)))
                        break;

                dev_dbg(&bp->pdev->dev, "skb %u (data %p) TX complete\n",
                        tail, skb->data);
                dma_unmap_single(&bp->pdev->dev, rp->mapping, skb->len,
                                 DMA_TO_DEVICE);
                bp->stats.tx_packets++;
                bp->stats.tx_bytes += skb->len;
                rp->skb = NULL;
                dev_kfree_skb_irq(skb);
        }

        bp->tx_tail = tail;
        if (netif_queue_stopped(bp->dev) &&
           TX_BUFFS_AVAIL(bp) > MACB_TX_WAKEUP_THRESH)
                netif_wake_queue(bp->dev);
}
```

数据发送
macb_tx

```
static irqreturn_t macb_interrupt(int irq, void *dev_id)
{
    struct net_device *dev = dev_id;
    struct macb *bp = netdev_priv(dev);
    u32 status;

    status = macb_readl(bp, ISR);

    if (unlikely(!status))
        return IRQ_NONE;

    spin_lock(&bp->lock);

    while (status) {
        /* close possible race with dev_close */
        if (unlikely(!netif_running(dev))) {
            macb_writel(bp, IDR, ~0UL);
            break;
        }
```

中断处理
macb_interrupt

中断处理
macb_interrupt

```
if (status & MACB_RX_INT_FLAGS) {
        if (netif_rx_schedule_prep(dev, &bp->napi)) {
                /*
                 * There's no point taking any more interrupts
                 * until we have processed the buffers
                 */
                macb_writel(bp, IDR, MACB_RX_INT_FLAGS);
                dev_dbg(&bp->pdev->dev,
                        "scheduling RX softirq\n");
                __netif_rx_schedule(dev, &bp->napi);
        }
}

if (status & (MACB_BIT(TCOMP) | MACB_BIT(ISR_TUND)))
        macb_tx(bp);

/*
 * Link change detection isn't possible with RMII, so we'll
 * add that if/when we get our hands on a full-blown MII PHY.
 */
```

```
        if (status & MACB_BIT(HRESP)) {
                /*
                 * TODO: Reset the hardware, and maybe move the printk
                 * to a lower-priority context as well (work queue?)
                 */
                printk(KERN_ERR "%s: DMA bus error: HRESP not OK\n",
                       dev->name);
        }

        status = macb_readl(bp, ISR);
    }

    spin_unlock(&bp->lock);

    return IRQ_HANDLED;
}
```

中断处理
macb_interrupt

小结

- 本章简单介绍了以太网的基础知识，网络驱动的基本模型，网络驱动中的两个基本数据结构**struct net_device**（网络设备结构体）和**struct sk_buff**（套接字缓冲结构体），数据包的发送，数据包的接收。

- 分析**AT91SAM9G45**网卡的驱动代码。

- 网络设备驱动涉及中断处理、并发控制、高级**I/O**操作等驱动开发难点。

进一步探索

- 网络设备驱动实现时经常会用到哪两个数据结构？它们各自在驱动中的作用是什么？

- 网络设备驱动程序中数据的接收是如何实现的？

Thanks