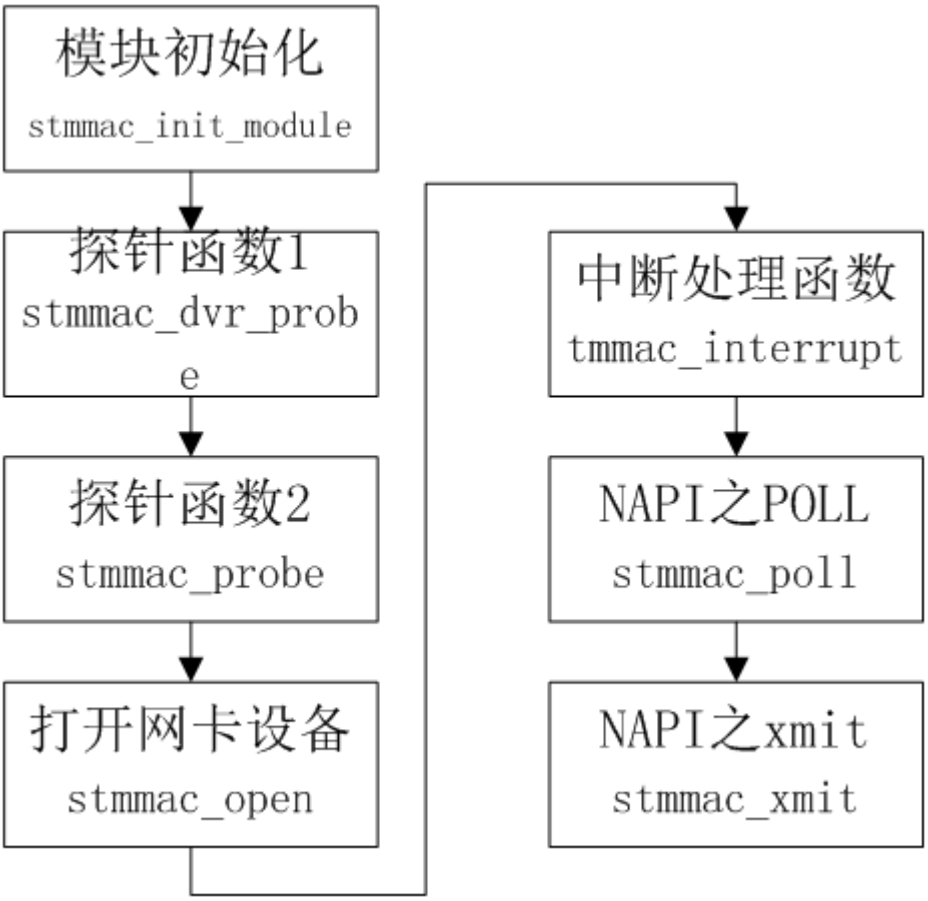
Hi35xx stmmac network card driver source code interpretation

1. Source files

drivers/net/stmmac/stmmac_main.c

The source code reading sequence is as follows:



Two, platform_device_register and platform_driver_register

platform_device_register: register device

platform_driver_register: register driver

The two are connected by name, and the names of all the two must be consistent.

platform_device_register must be completed before the driver is loaded, because the driver registration needs to match the registered device name in the kernel.

When platform_driver_register is registered, it will compare the currently registered drivername with the names in all registered devices, and only if the same one is found can the registration be

successful. After matching, the probe will be called to detect the device, and then the device will be bound to the driver.

Platform is a virtual address bus. Compared with pci and USB, it is mainly used to describe the on-chip resources on the SOC. Usage convention: If devices that do not belong to any bus, such as network cards, serial ports, etc., are hung on this virtual bus.

3. Platform-driven data structure

```
static struct platform_driver stmmac_driver = {
    .probe = stmmac_dvr_probe, (see Section 4)
   .remove = stmmac_dvr_remove,
   .driver = {
         .name = STMMAC_RESOURCE_NAME,
         .owner = THIS_MODULE,
         .pm = &stmmac_pm_ops,
         },
};


static struct platform_devicestmmac_platform_device[] = {
   {
      .name = STMMAC_RESOURCE_NAME,
      .id = 0,
      .dev = {
         .platform_data = & stmmac_ethernet_platform_data ,
         .dma_mask = (u64 *) ~0,
         .coherent_dma_mask = (u64) ~0,
```

```
        .release = stmmac_platform_device_release,
        },
    .num_resources = ARRAY_SIZE(stmmac_resources),
    .resource = stmmac_resources,
  }
};

static struct plat_stmmacenet_data stmmac_ethernet_platform_data = {
  .bus_id = 1,
  .pbl = DMA_BURST_LEN,
  .has_gmac = 1,
  .enh_desc = 1,
#ifdef TNK_HW_PLATFORM_FPGA
  .clk_csr = 0x2, /* For 24Mhz bus clock input */
#else
  .clk_csr = 0x4, /* For 155Mhz bus clock input */
#endif
};
```

4. Drive probe stmmac_dvr_probe

When the platform driver is registered, if it matches the platform device name, the initialization interface will be called.

1. Basic process of interface operation

The flow chart of this function is as follows:

(1) Obtain the base address of the system configuration register

ret = stmmac_syscfg_init(pdev);

(2) Obtain the base address of the MAC controller register

stmmac_base_ioaddr =ioremap_nocache(res->start, res->end - res->start + 1);

(3) Apply for network card equipment and private data

ndev =alloc_etherdev(sizeof(struct stmmac_priv));

The network card device and private data are closely together: network card device + private data structure, private data is obtained through netdev_priv. The network card device is a general data structure, and the private data is the data structure of each MAC controller

(4) MAC controller settings

ret =stmmac_mac_device_setup(ndev); (see section 2)

(5) The network device is actually registered

ret = stmmac_probe (ndev); (important, see Section 5)

(6) MDIO bus registration

ret = stmmac_mdio_register (ndev); (see section 3)

(7) Assign the network card device to the global variable

stmmac_device_list[priv->id] = ndev;, this global variable will be used in other interfaces.

Note: Steps 3~7, if there are multiple network cards, execute in a loop.

(8) Reset DMA, GMA and other registers

stmmac_reset();

(9) Determine whether to support checksum offload engine

priv->rx_coe =priv->hw->mac->rx_coe(priv->ioaddr); function initialization was performed in step 4 above

(10) Set the hardware DMA working mode

stmmac_dma_operation_mode(priv); (see Section 4)

(11) Initialize the RX/TX queue

priv->dma_tx_size = STMMAC_ALIGN(dma_txsize);

priv->dma_rx_size =STMMAC_ALIGN(dma_rxsize);

priv->dma_buf_sz = STMMAC_ALIGN(buf_sz);

init_dma_desc_rings(ndev); (see section 5)

(12) Hardware DMA initialization

priv->hw->dma->init(priv->dma_ioaddr,

priv->dma_channel,

priv->plat->pbl,

priv->dma_tx_phy,

priv->dma_rx_phy,0)

The most important opinion thing: priv->dma_tx_phy and priv->dma_rx_phy DMA handle applied by init_dma_desc_rings are written to receive and send description sub-list address registers

(13) Receive and send index initialization

priv->dirty_rx = priv->cur_rx = 0;

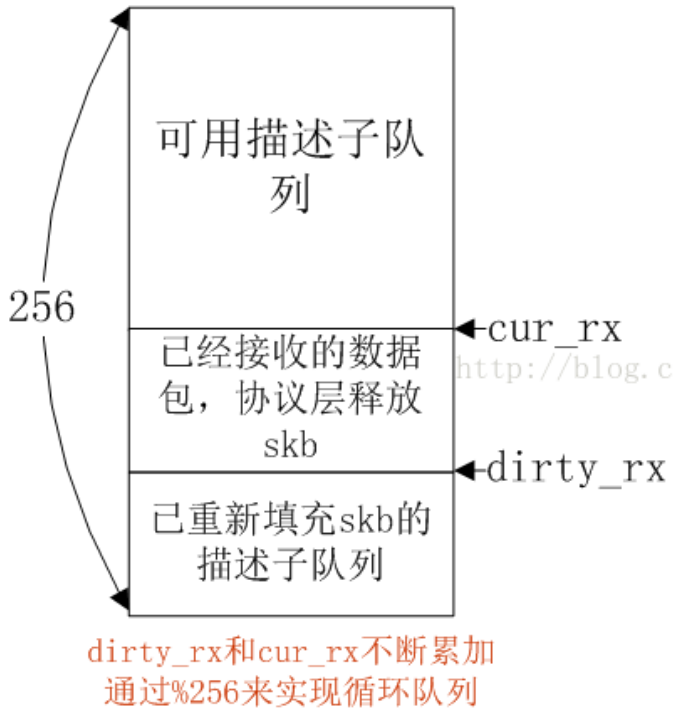
priv->dirty_tx = priv->cur_tx = 0;

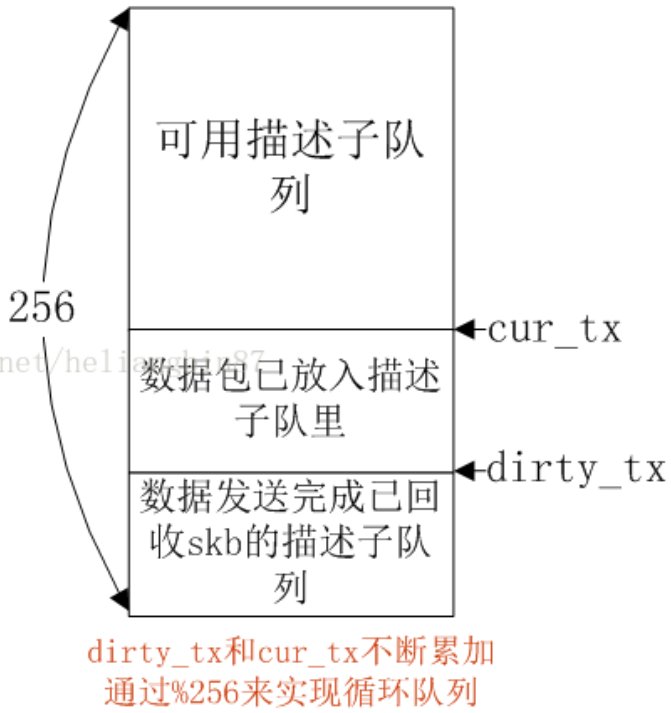
cur: The data frame index uploaded to the network protocol layer has been obtained, and the protocol layer will release skb

dirty: the index of the skb to the dma queue has been filled

The two values are continuously accumulated, and the dma queue is processed cyclically by taking the remainder %.

as the picture shows:





(14) Initialize the receive recovery queue

skb_queue_head_init(&priv->rx_recycle);

Note: Steps 9~11 will be executed cyclically if there are multiple network cards.

(15) Initialize TOE_NK

ret = tnk_init(stmmac_base_ioaddr,&pdev->dev); (see section 6)

(16) request interrupt

ret= request_irq(SYNOP_GMAC_IRQNUM, stmmac_interrupt , IRQF_SHARED,STMMAC_RESOURCE_NAME, pdev); (important, see Section VII)

(17) Enable all interrupts

writel(~0, stmmac_base_ioaddr +TNK_REG_INTR_EN);

2. MAC controller setting stmmac_mac_device_setup(ndev) ;

(1) Select the GMAC device and initialize it

device = dwmac1000_setup(priv->ioaddr);

A. Apply for a MAC device: mac =kzalloc();

B．GMAC operation: mac->mac = &dwmac1000_ops;

C．GMAC dma operation: mac->dma = &dwmac1000_dma_ops;

(2) Select the enhanced descriptor data structure

device->desc = &enh_desc_ops;

Note: Many of the above are basic operations on the MAC register.

3. MII bus registration stmmac_mdio_register

(1) Apply for the MII bus

stmmac_mii_bus = mdiobus_alloc();

(2) Initialize mdio working clock

tnkclk = mdio_clk_init();

(3) Initialize the MII bus structure

stmmac_mii_bus->name = "STMMAC MIIBus";

stmmac_mii_bus->read =&stmmac_mdio_read;

stmmac_mii_bus->write =&stmmac_mdio_write;

stmmac_mii_bus-> reset =&stmmac_mdio_reset;

……..

(4) Register MII bus mdiobus_register(stmmac_mii_bus);

A．Register bus device device_register(&bus->dev);

B．Reset bus bus->reset(bus);

C．Scan PHY devices on the bus, up to 32

phydev = mdiobus_scan(bus, i);

Mainly do three things: read PHY ID, create PHY device, and register PHY device.

Creating a PHY device interface get_phy_device will call phy_device_create to initialize the PHY device data structure, and then initialize the work queue phy_state_machine, which will call the

adjust_link interface.

(5) Assign MII bus to stmmac private data

priv->mii = stmmac_mii_bus;

4. Set the hardware DMA working mode stmmac_dma_operation_mode

priv->hw->dma->dma_mode(priv->dma_ioaddr,priv->dma_channel,

tc, SF_DMA_MODE);

Send: tc defaults to 64, does not support DMA store and forward. TxFIFO judges whether to forward according to the FIFO threshold

Receive: Support DMA store and forward. The frame is forwarded upwards only after the RxFIFO has received a complete frame.

5. Initialize the dma descriptor queue init_dma_desc_rings

(1) RX queue application

priv->rx_skbuff_dma = kmalloc(rxsize *sizeof(dma_addr_t), GFP_KERNEL);

Apply for the dma address queue corresponding to 256 skb

priv->rx_skbuff =kmalloc(sizeof(structsk_buff *) * rxsize, GFP_KERNEL);

Apply for 256 skb queues

priv->dma_rx =(struct dma_desc *)dma_alloc_coherent(priv->device,

rxsize *sizeof(structdma_desc),

&priv->dma_rx_phy,

GFP_KERNEL);

The DMA descriptor (receiving descriptor in the datasheet) establishes a consistent DMA mapping with a size of 256* sizeof(structdma_desc), and returns the virtual memory address of the buffer. priv->dma_rx_phy returns the correct DMA handle. I personally feel that it is the physical address corresponding to the virtual address, because this address needs to be written to the address register of the receiving description sub-link list. (This function is not very clear)

(2) TX queue application

priv->tx_skbuff = kmalloc(sizeof(struct sk_buff*) * txsize, GFP_KERNEL);

Apply for 256 skb pointers

priv->tx_page = kmalloc(sizeof(structpage *) * txsize, GFP_KERNEL);

Apply for 256 page pointers

priv->dma_tx = (struct dma_desc*)dma_alloc_coherent(priv->device,

txsize *sizeof(structdma_desc),

&priv->dma_tx_phy,

GFP_KERNEL);

ditto

(3) RX queue initialization and associated data requested in (1)

Apply for 256 skbs through a for loop, and each skb is 2KB in size.

A．skb = netdev_alloc_skb_ip_align(dev, bfsize);

Apply for skb data

B．priv->rx_skbuff[i]= skb;

put skb into receive queue

C．priv->rx_skbuff_dma[i]= dma_map_single(priv->device, skb->data, bfsize, DMA_FROM_DEVICE);

Then add the skb data to the skb dma address queue through streaming DMA mapping

D．(priv->dma_rx +i)->des2= priv->rx_skbuff_dma[i];

Then put the skb dma address into the dma descriptor

(4) The TX queue does not have an RX initialization operation similar to receiving

```
for (i = 0; i < txsize; i++) {
    priv->tx_skbuff[i] = NULL;
```

```
        priv->tx_page[i] = NULL;

        priv->dma_tx[i].des2 = 0;

}
```

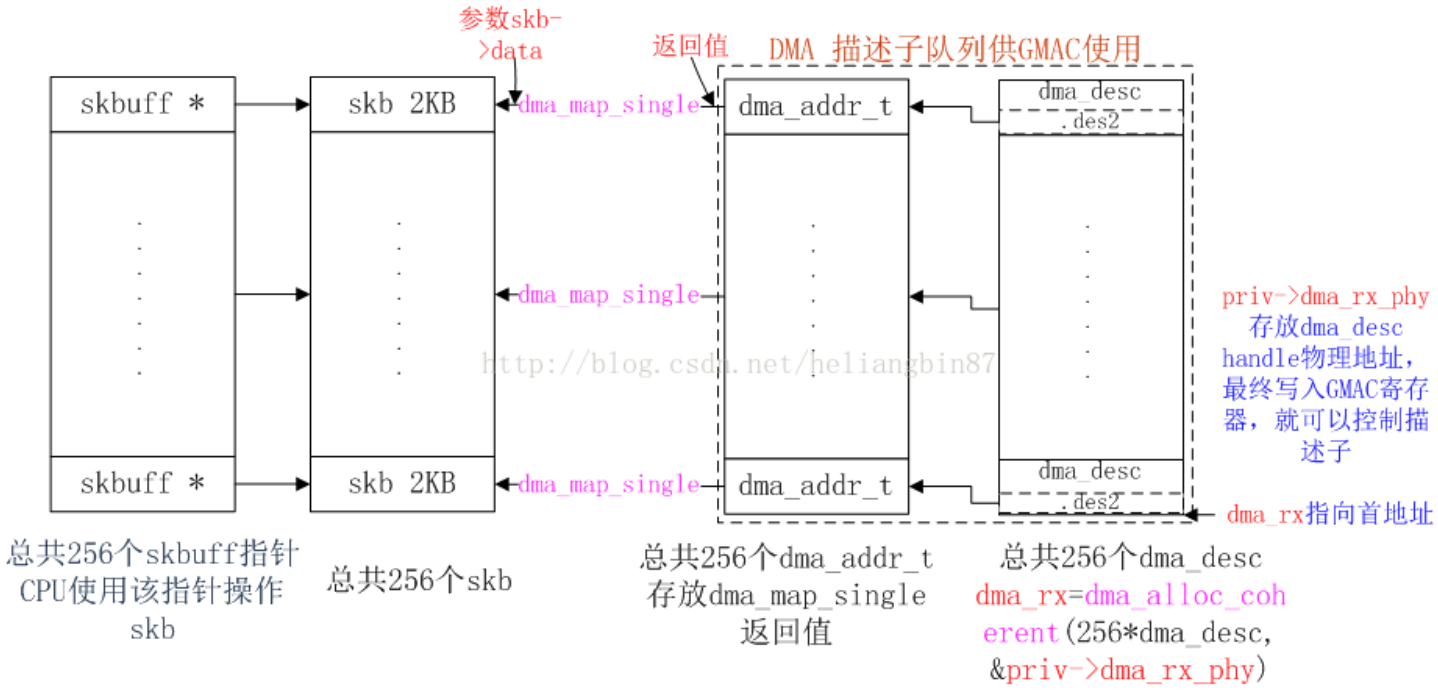
Because skb is applied at the network layer and passed to the driver.

(5) Initialize the RX/TX descriptor

```
priv->hw->desc->init_rx_desc(priv->dma_rx,rxsize,dis_ic);

priv->hw->desc->init_tx_desc(priv->dma_tx,txsize);
```

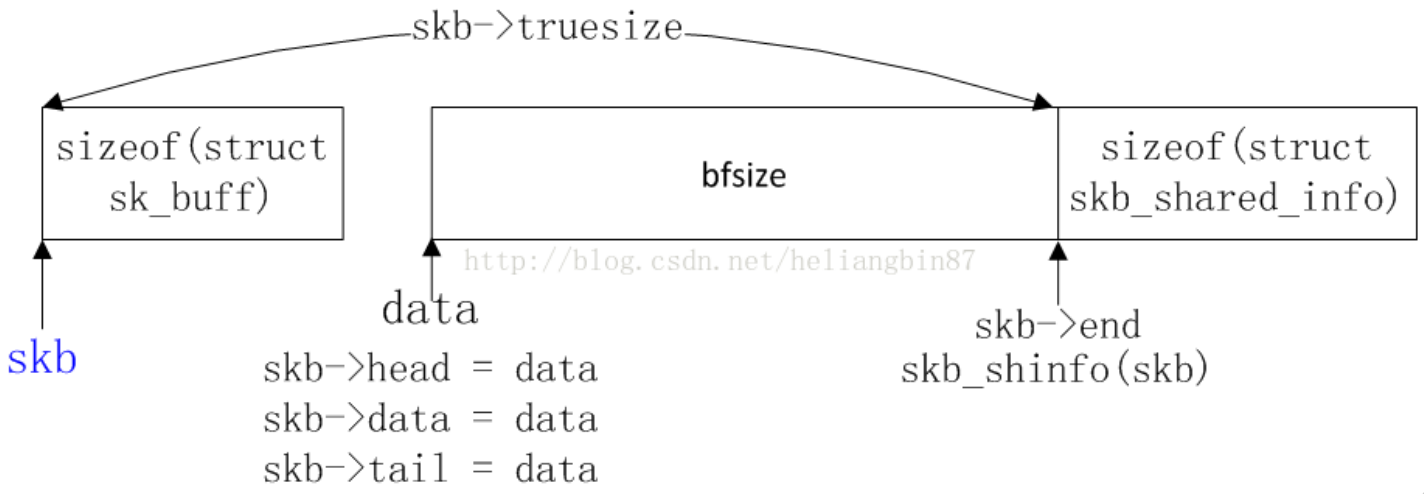
(6) dma description subqueue and skb mapping



The above is the establishment of the mapping process when receiving the DMA descriptor queue and skb initialization. For the sending DMA sending queue, the mapping will only be established when sending. If there are multiple data fragments in a data packet, the dma_map_page interface will be called for page mapping processing.

(7) skb application allocation space map

```
skb = netdev_alloc_skb_ip_align(dev,bfsize);
```



skb->len: the length of all data in the packet, skb->data_len separates and stores the length of the data segment. data_len = 0 when there is only one data segment.

6. TOE_NK initialization tnk_init ()

(1) Create a proc directory

ret = tnk_proc_init(hitoe ? max_connections: 0);

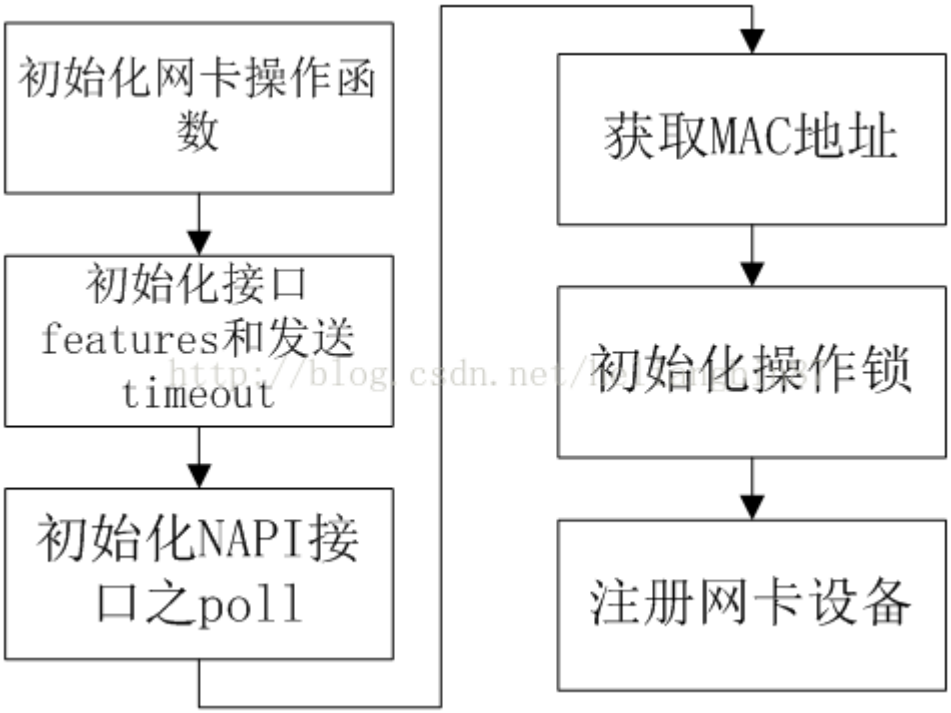
(2) If TOE is used, a series of initializations are performed on TOE

Since it is not used, it is not described here.

Five, really register the network device stmmac_probe

1. The basic operation process of the interface

The basic flowchart of this function:



(1) Initialize the basic information of the Ethernet network card device ether_setup();

dev->header_ops = ð_header_ops;

dev->hard_header_len = ETH_HLEN; (14)

dev->mtu = ETH_DATA_LEN; (1500)

dev->addr_len = ETH_ALEN; (6)

dev->tx_queue_len = 1000; /*Ethernet wants good queues */

dev->flags =IFF_BROADCAST|IFF_MULTICAST;

dev->priv_flags =IFF_TX_SKB_SHARING;

memset(dev->broadcast, 0xFF, ETH_ALEN);

(2) Initialize the network card operation function andethtool operation function

dev->netdev_ops =&stmmac_netdev_ops; (see section 2)

stmmac_set_ethtool_ops(dev); (see subsection 3)

dev->features |= NETIF_F_SG |NETIF_F_HIGHDMA | NETIF_F_IP_CSUM

| NETIF_F_IPV6_CSUM;

Set the interface characteristics of the network card, which will be used by the driver and protocol layer

dev->watchdog_timeo = msecs_to_jiffies(watchdog); Set the sending timeout, after the timeout will eventually call stmmac_tx_timeout for timeout processing

(3) Initialize the NAPI polling interface for receiving data packets

netif_napi_add(dev, &priv->napi, stmmac_poll , 64); (important, see Section 8)

Initialize napi, poll=stmmac_poll, weight=64, poll_list and other information

Weight: Describes the relative importance of the interface. When resources are tight, how much traffic can be tolerated on the interface.

(4) Obtain the MAC address from the register and judge the validity

priv->hw->mac->get_umac_addr((void __iomem *)dev->base_addr, dev->dev_addr, 0);

(5) Initialize the spin lock

spin_lock_init(&priv->lock);

tnk_lock_init(&priv->tlock);

(6) Register the network card device

ret = register_netdev(dev);

2. Basic operation functions of network card

static const struct net_device_opsstmmac_netdev_ops = {

.ndo_open = stmmac_open, (see Section 6)

```
    .ndo_start_xmit = stmmac_xmit, (see Section 9)
    .ndo_stop = stmmac_release,
    .ndo_change_mtu = stmmac_change_mtu,
    .ndo_set_multicast_list = stmmac_multicast_list,
    .ndo_tx_timeout = stmmac_tx_timeout,
    .ndo_do_ioctl = stmmac_ioctl,
    .ndo_set_config = stmmac_config,
#ifdef STMMAC_VLAN_TAG_USED
    .ndo_vlan_rx_register = stmmac_vlan_rx_register,
#endif
#ifdef CONFIG_NET_POLL_CONTROLLER
    .ndo_poll_controller = stmmac_poll_controller,
#endif
    .ndo_set_mac_address = eth_mac_addr,
};
```

3. ethtool operation function

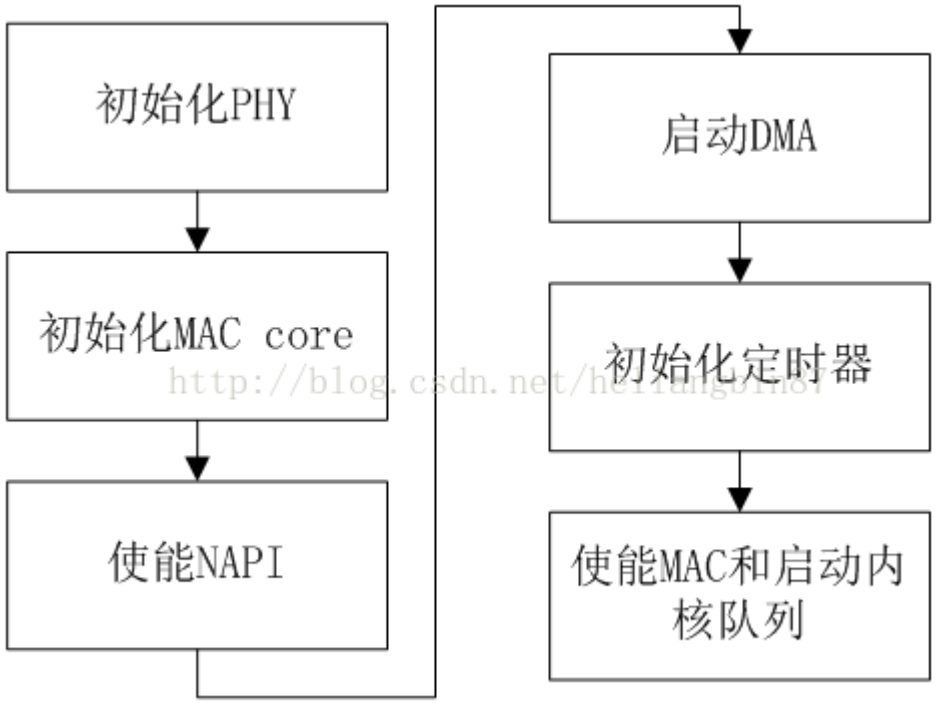
```
static struct ethtool_opsstmmac_ethtool_ops = {
    .begin = stmmac_check_if_running,
    .get_drvinfo = stmmac_ethtool_getdrvinfo,
    .get_settings = stmmac_ethtool_getsettings,
    .set_settings = stmmac_ethtool_setsettings,
    .get_msglevel = stmmac_ethtool_getmsglevel,
    .set_msglevel = stmmac_ethtool_setmsglevel,
    .get_regs = stmmac_ethtool_gregs,
```

```
    .get_regs_len = stmmac_ethtool_get_regs_len,
    .get_link = stmmac_ethtool_get_link,
    .get_rx_csum = stmmac_ethtool_get_rx_csum,
    .get_tx_csum = ethtool_op_get_tx_csum,
    .set_tx_csum = ethtool_op_set_tx_ipv6_csum,
    .get_sg = ethtool_op_get_sg,
    .set_sg = ethtool_op_set_sg,
    .get_pauseparam = stmmac_get_pauseparam,
    .set_pauseparam = stmmac_set_pauseparam,
    .get_ethtool_stats = stmmac_get_ethtool_stats,
    .get_strings = stmmac_get_strings,
    .get_wol = stmmac_get_wol,
    .set_wol = stmmac_set_wol,
    .get_sset_count = stmmac_get_sset_count,
    .get_tso = ethtool_op_get_tso,
    .set_tso = ethtool_op_set_tso,
};
```

6. Open the network card device stmmac_open

1. Basic process

Basic flowchart:



(1) MAC address validity

is_valid_ether_addr(dev->dev_addr)

(2) Modify the parameters passed to the driver

stmmac_verify_args();

(3) Initialize the PHY device

stmmac_init_phy(dev); (see section 2)

(4) Device MAC address to hardware

priv->hw->mac->set_umac_addr(priv->ioaddr,dev->dev_addr, 0);

(5) Initialize the MAC core

priv->hw->mac->core_init(priv->ioaddr);

(6) Enable NAPI and schedule

napi_enable(&priv->napi);

napi_schedule(&priv->napi)

(7) Start DMA transceiver (write register)

priv->hw->dma->start_tx(priv->dma_ioaddr,priv->dma_channel);

priv->hw->dma->start_rx(priv->dma_ioaddr,priv->dma_channel);

(8) Start the relevant timer

priv->poll_timer.function= stmmac_poll_func;

Scheduled wakeup of RxDMA/TxDMA in pending state

priv->check_timer.function= stmmac_check_func;

Read the register address to get the current receive descriptor address, if it is before cur_rx and dirty_cur, set the descriptor to belong to GMAC operation.

(9) Enable MAC TX/RX

stmmac_enable_mac(priv->ioaddr);

(10) Start queue

netif_start_queue(dev);

2. Initialize the PHY device stmmac_init_phy

(1) Notify the kernel that the carrier disappears

netif_carrier_off(dev);

(2) The Ethernet device is connected to the PHY device

phydev= phy_connect(dev, phy_id, & stmmac_adjust_link , 0, priv->phy_interface);

stmmac_adjust_link : According to the actual PHY situation, adjust the connection parameters: full-duplex / half-duplex, speed, MII/GMII

A．d =bus_find_device_by_name(&mdio_bus_type, NULL, bus_id);

Find the specified device in the MDIO bus PHY device list

B．device into a PHY device

phydev = to_phy_device(d);

C．rc = phy_connect_direct(dev, phydev, handler, flags, interface);

Connect the Ethernet device to the specified PHY device, and then start the PHY.

Seven, interrupt processing function stmmac_interrupt

There are 3 parallel processes in the interrupt: network card 1, network card 2, and TOE. The processing flow of network card and network card 2 is the same, and TOE is not used, and only network card 1 is analyzed.

When the first data packet is received or the data transmission is completed, an interrupt is generated and the interrupt processing function is entered.

1. Read the interrupt status register

Regardless of sending or receiving, when a DMA0 interrupt is generated, it enters the dma interrupt processing function

2. stmmac_dma_interrupt

status = priv->hw->dma->dma_interrupt(priv->dma_ioaddr,priv->dma_channel,

&priv->xstats);

Read the dma interrupt status, if it is received and sent correctly, perform NAPI scheduling _stmmac_schedule(priv).

It calls napi_schedule(&priv->napi) and disables interrupts (interrupts are enabled by stmmac_poll).

3. Simple description of napi_schedule

(1) Add priv->napi to softnet_data->napi. From the definition, each CPU has a softnet_data

(2) Start the network receiving soft interrupt NET_RX_SOFTIRQ

(3) Process the soft interrupt within a certain period of time in the CPU

(4) The soft interrupt processing function is net_rx_action, which will call n->poll for corresponding processing (the poll here is the stmmac_poll registered by the probe below , see the description in the following chapters)

(5) The soft interrupt processing function is registered during subsys_initcall (net_dev_init)

open_softirq(NET_TX_SOFTIRQ, net_tx_action);

```
open_softirq(NET_RX_SOFTIRQ, net_rx_action);
```

(6) Finally, all soft interrupts are uniformly processed in do_softirq.

Common soft interrupt types:

```
enum
{
  HI_SOFTIRQ=0,
  TIMER_SOFTIRQ,
  NET_TX_SOFTIRQ,
  NET_RX_SOFTIRQ,
  BLOCK_SOFTIRQ,
  BLOCK_IOPOLL_SOFTIRQ,
  TASKLET_SOFTIRQ,
  SCHED_SOFTIRQ,
  HRTIMER_SOFTIRQ,
  RCU_SOFTIRQ, /* Preferable RCU should always be the last softirq */
  NR_SOFTIRQS
};
```

4. The received data is not processed in the interrupt, but the NAPI interface is called to process it at an appropriate time.

8. stmmac_poll of NAPI interface

This interface does two things: one is to generate an interrupt when the data transmission is completed, enter this function for resource recovery; the other is to generate an interrupt when receiving data, and enter this function to receive and process data.

1. Basic operation process

(1) Record proc information when the interface starts

```
stmmac_poll_begin();
```

(2) Recycle the resources that have been sent (explained in the sending chapter)

stmmac_tx (priv); (see section 9, subsection 7)

(3) Receive data packets

stmmac_rx (priv, budget); (see subsection 2)

(4) If the received data packet does not exceed the capacity of the interface, enable the interrupt

```
if (work_done< budget) {
    stm_proc.polls_done++;
    napi_complete(napi);
    stmmac_enable_irq(priv);
  }
```

(5) End the interface call

```
stmmac_poll_end();
```

2. Data packet receiving and processing stmmac_rx

```
unsigned int entry = priv->cur_rx % rxsize;
struct dma_desc *p = priv->dma_rx +entry;
```

(1) priv->hw->desc->get_rx_owner(p)

Judging the ownership of the current descriptor: OWN bit in the descriptor data structure, 0: the current descriptor should be operated by the CPU, 1: the previous descriptor should be operated by the GMAC. For reception, set to 1 when initializing the DMA descriptor queue.

GMAC obtains the available receiving descriptor according to the register configuration, and then reads the Ethernet message received from the PHY from the RxFIFO. If the message meets the receiving condition, writes the message into the data buffer pointed to by the receiving descriptor and writes it back . Receive descriptor . This write-back will set the OWN bit to 0.

(2) Get the next frame descriptor

next_entry = (++priv->cur_rx) % rxsize;

p_next =priv->dma_rx + next_entry;

priv->cur_rx : the index that has been passed to the protocol layer

(3) Obtain the received frame status

status =(priv->hw->desc->rx_status(&priv->dev->stats, &priv->xstats,p));

If it is a discarded frame, nothing will be processed; otherwise, it will be uploaded to the upper layer network.

(4) Get the frame length

frame_len =priv->hw->desc->get_rx_frame_len(p);

(5) Get frame data

skb = priv->rx_skbuff[entry];

priv->rx_skbuff[entry] = NULL;

Note: skb will be released after the upper layer network is processed.

(6) Set skb data length and release streaming DMA mapping

skb_put(skb, frame_len);

dma_unmap_single(priv->device, priv->rx_skbuff_dma[entry],priv->dma_buf_sz, DMA_FROM_DEVICE);

(7) Obtain the protocol type of skb

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

skb->dev = priv->dev;

(8) Upload the skb to the upper layer network protocol through the NAPI interface for processing

napi_gro_receive(&priv->napi,skb); (see Section 9)

Note: The above is a while loop operation

(9) Refill the receive queue

stmmac_rx_refill(priv); (see subsection 3)

3. Receive queue filling function stmmac_rx_refill

From Section 2, the skb that received the data is stripped from the dma receiving queue, and after it is passed to the network layer for processing, the skb will be released. Therefore, it is necessary to re-apply for a new skb transfer to fill the dma receive queue.

(1) Determine whether refilling is required

Compare priv->cur_rx with priv->dirty_rx. If priv->cur_rx > priv->dirty_rx, that is, the number uploaded to the network protocol layer is greater than the number of filled dmas, the dma receiving queue will be refilled.

(2) Check the attribution of the descriptor operation

priv->hw->desc->get_rx_owner(p+ entry), only those belonging to the CPU will be filled.

(3) Apply for a new SKB

skb= __skb_dequeue(&priv->rx_recycle);

Get a valid skb from the recovery queue first. The recovery queue comes from the data frame skb that is requested by the protocol layer and sent. Recycling on the stmmac_tx interface does not release the skb, but puts it in the recycling queue to improve utilization efficiency.

skb= netdev_alloc_skb_ip_align(priv->dev,

If there is no recovery queue, re-apply for skb.

(4) Fill the receive DMA descriptor queue

priv->rx_skbuff[entry]= skb;

priv->rx_skbuff_dma[entry]= dma_map_single(priv->device, skb->data, bfsize, DMA_FROM_DEVICE);

(p+ entry)->des2 = priv->rx_skbuff_dma[entry];

(5) Set the descriptor attribution and hand it over to GMAC for operation

priv->hw->desc->set_rx_owner(p+ entry);

Note: The above is performed through a for loop until there is no need to fill.

(6) Activate RxDMA

STMMAC_SYNC_BARRIER();

Before processing the next instruction, first flush the DMA write buffers

priv->hw->dma->enable_dma_receive(priv->dma_ioaddr,priv->dma_channel);

Writing any value wakes up the pending RxDMA.

Nine, transfer skb data to the protocol stack napi_gro_receive (generic receive offload)

1. Understand the concept

TSO, UFO and GSO correspond to network transmission, and LRO and GRO correspond to network reception.

TSO (TCPSegmentation Offload) is a technology that uses network cards to fragment TCP packets and reduce CPU load. It is also called LSO (Largesegment offload), TSO-TCP, UFO-UDP. If the hardware supports the TSO function, the hardware-supported TCP checksum calculation and scatter/gather functions are also required.

GSO (GenericSegmentation Offload) is more general than TSO. Basic idea: delay data fragmentation as much as possible until before sending it to the network card driver. At this time, it will check whether the network card supports the fragmentation function (TSO, UFO). If it supports it, it can be sent directly to the network card. to the network card. In this way, a large data packet only needs to go through the protocol stack once, instead of being divided into several data packets to go separately, which improves efficiency.

LRO (Large Receive Offlaod) aggregates multiple received TCP data into a large data packet, and then passes it to the network protocol layer for processing, so as to reduce the processing overhead of the upper layer protocol stack and improve the system's ability to receive TCP data packets

The basic idea of GRO (Generic Receive Offload) is similar to LRO, but it is more general. Now the driver basically uses the GRO interface instead of LRO.

RSS (Receive Side Scaling) is a feature of a network card, commonly known as multi-queue. A network card with multiple RSS queues can divide different network flows into different queues, and then assign these queues to multiple CPU cores for processing, so as to distribute the load and make full use of the capabilities of multi-core processors.

2. GRO structure struct napi_gro_cb

The GRO function uses the private space char cb[48] in the skb structure to store some information used by gro.

The specific content is as follows:

struct napi_gro_cb {

/* Virtual address ofskb_shinfo(skb)->frags[0].page + offset. */

Points to the head of the data stored in skb_shinfo(skb)->frag[0].page. See skb_gro_reset_offset

```
  void *frag0;


   /*Length of frag0. The length of the data in the first page*/
  unsigned int frag0_len;
```

/* This indicates where we are processing relative to skb->data. It indicates the offset from skb->data to the data area to be processed by GRO*/

```
    For example, enter the ip layer for GRO processing. At this time, skb->data points to
the ip header, and the gro of the ip layer just needs to process the ip header. At this
time, the offset is 0. After entering the transport layer, GRO processing is performed. At
this time, skb->data still points to the ip header, and the tcp layer gro needs to process
the tcp header. At this time, the offset is the length of the ip header.
```

```
  intdata_offset;
```

/* This is non-zero if the packet may be of the same flow. Whether the packet hung on napi->gro_list matches the current packet*/

The gro_receive of each layer sets this flag bit. After receiving a message, use the message to match the message hanging on napi->gro_list. At the link layer, use the dev and mac headers for matching. If the same indicates that the two messages are sent by the same device, mark the same of the corresponding skb on the napi->gro_list as 1. Go to the network layer and proceed further When matching, it only needs to match at the network layer with the packet that has just been marked as 1 by the link layer on the napi->list, and it is not necessary to match with each packet. If the network layer does not match, this flag is cleared. At the transport layer, only the packets marked as 1 by the network layer can be configured. This is designed to reduce unnecessary matching operations

intsame_flow;

/* This is non-zero if the packet cannot be merged with the new skb. */

If this field is not 0, it means that the data message does not need to wait for the merge, and can be directly sent to the protocol stack for processing

int flush;

/*Number of segments aggregated. The number of times the segment has been aggregated*/

int count;

/*Free the skb? Should it be discarded*/

intfree;

};

3. The basic flow of the function

(1) skb_gro_reset_offset (see subsection 4)

Initialize NAPI_GRO_CB;

(2) __napi_gro_receive (see subsection 6)

Connection layer gro_receive, realize data packet merging or upload protocol stack

(3) napi_skb_finish (see section 5 , let's talk about the content first)

According to the return value of the second function, it is decided to merge, feed protocol stack, or free.

4. Initialize GRO cb: skb_gro_reset_offset(skb)

A network card that supports S/G IO has the following possibility: skb itself does not contain data (neither does the header), all data is stored in skb_share_info, and frags.page is not in higih_mem. In this case, if you want to merge, take out the packet header information, that is, frags[0] of skb_shared_info, and save the header information to frags0 of napi_gro_cb.

The specific source code is as follows:

```
NAPI_GRO_CB(skb)-> data_offset = 0;

NAPI_GRO_CB(skb)->frag0 = NULL;

NAPI_GRO_CB(skb)->frag0_len = 0;



If mac_header and skb->tail are equal and the address is not in high-end memory, it means
that the packet header is stored in skb_shinfo, so we need to get the corresponding packet
from frags

if(skb->mac_header == skb->tail && !PageHighMem(skb_shinfo(skb)->frags[0].page)){

You can see that frag0 saves the address of the first element of the corresponding skb
frags

NAPI_GRO_CB(skb)-> frag0 = page_address(skb_shinfo(skb)->frags[0].page)+

        skb_shinfo(skb)->frags[0].page_offset;
```

Then save the corresponding size

```
NAPI_GRO_CB(skb)-> frag0_len = skb_shinfo(skb)->frags[0].size;

   }
```

5. napi_skb_finish

Process the merged packet according to the return value of __napi_gro_receive.

```
switch(ret) {
    //Send the packet to the protocol stack
  case GRO_NORMAL:
    if (netif_receive_skb(skb)) (see subsection 8 )
      ret = GRO_DROP;
    break;
    //The message can be discarded or has been merged into gro, then the free message
  case GRO_DROP:
  case GRO_MERGED_FREE:
    kfree_skb(skb);
    break;
 //The data has been saved by gro, but it has not been merged, and the skb still needs to be reserved and cannot be released.
  case GRO_HELD:
  case GRO_MERGED:
    break;
  }
  return ret;
```

6. __napi_gro_receive receives the merge function

(1) Basic concepts

Each protocol defines its own GRO receive merge function and post-merge processing function, namely gro_receive and gro_complete. The GRO system will call the corresponding callback function according to the protocol. gro_receive merges skb into gro_list, and the return value: if it is empty, it means that it does not need to be sent to the protocol stack after merging, and if it is not empty, it needs to be sent to the protocol stack immediately. gro_complete is called when the gro merged data packet is sent to the protocol stack when the return value is not empty.

__napi_gro_receive and napi_gro_complete can be regarded as gro_receive and gro_complete of the link layer.

Each protocol layer gro_receive:

```
.gro_receive = tcp4_gro_receive,

.gro_receive = inet_gro_receive,

.gro_receive = ipv6_gro_receive,
```

Each protocol layer gro_complete:

```
.gro_complete = tcp4_gro_complete,

.gro_complete = inet_gro_complete,

.gro_complete = ipv6_gro_complete,
```

(2) Basic content of the function

```
A．for (p = napi->gro_list;p; p = p->next) {
    unsigned long diffs;


    diffs = (unsigned long)p->dev ^(unsigned long)skb->dev;
    diffs |= p->vlan_tci ^skb->vlan_tci;
    diffs |=compare_ether_header(skb_mac_header(p),
                skb_gro_mac_header(skb));
    NAPI_GRO_CB(p)->same_flow = !diffs;
    NAPI_GRO_CB(p)->flush = 0;
  }
```

Traverse the gro_list, find out whether the skb in the list has the same flow as the current skb, and then assign a value to same_flow. If this layer (link layer) is the same, then the ip layer and tcp layer will make the same flow judgment, otherwise no judgment will be made. The criteria for judging whether the streams of different layers are the same are different, and here are three judging conditions: the same device, the same VLAN, and the same MAC header.

B．_ returndev_gro_receive(napi, skb); (see section 7)

true receive merge processing

7 , dev_gro_receive real receive merge

It can be divided into two parts, one is the normal merge processing, and the other is the frags0 processing part. It should be noted that GRO does not support sliced IP packets. The grouping of IP slices will be done again at the kernel IP layer. It is meaningless to increase the complexity of GRO.

(1) Combine the same stream processing

A．First traverse the corresponding ptype (protocol class linked list), and after finding the matching protocol, call the corresponding callback function gro_receive; the link layer calls the gro_receive of the ip layer, that is, inet_gro_receive.

B．Enter the merge processing of the ip layer, mainly to judge whether the same_flow and flush are required. Only when the two packages are same_flow will the flush judgment

same_flow judgment: the protocol needs to be the same, the tos field needs to be the same, the source and destination addresses need to be the same; as long as one is different, it is set to 0.

Flush judgment: it is a sliced packet, the ttl is different, and the order of ids is wrong; as long as one meets the skb, the gro will be flushed to the protocol stack. Then enter the gro_receive of the TCP layer, namely tcp4_gro_receive.

C．Enter the merge processing of the TCP layer: similar to the IP layer, judge the same_flow and flush. Among them, there are many and complex judgments on flush. If flush is required, there is no need to perform merge processing, and the corresponding gro_list will be returned.

The real merge function: skb_gro_receive (not analyzed here).

(2) When the return value of gro_receive is not empty, the skb that is flushed out of gro is immediately fed into the protocol stack to process napi_gro_complete. If the same_flow of the current skb is non-zero, it means that the same flow has been found and merged and returned directly. If the same flow is not found, add skb to the gro_list.

(3) frags0 processing part

Move the frags stored in the skb_shinfo structure forward to skb. (why do you do that?)

8. netif_receive_skb _

Finally, the skb data is distributed to each protocol layer. I will explain when I have time.

10. Send data stmmac_xmit

This interface implements the Scatter/Gather I/O function, and the skb_shinfo macro is used to determine whether the data packet is composed of one data fragment or a large number of data fragments.

1. Obtain available sending descriptors

entry= priv->cur_tx % txsize;

desc= priv->dma_tx + entry;

first= desc; save the first data segment

2. Put the skb into the sending queue

priv->tx_skbuff[entry]= skb;

priv->tx_page[entry]= NULL;

3. Send a single or first packet

unsignedint nopaged_len = skb_headlen(skb);

desc->des2= dma_map_single(priv->device, skb->data, nopaged_len, DMA_TO_DEVICE);

priv->hw->desc->prepare_tx_desc(desc,1, nopaged_len, csum_insertion);

When there is only one data segment, skb->data will send all data; when there are multiple data segments, skb->data points to the first data segment, data length skb->len –skb->data_len, other data

Stored in the shared data structure frags array. (skb->len: the length of all data in the packet, skb->data_len separates and stores the length of the data segment)

4. Send the remaining data fragments

For multiple data fragments, data fragments must be sent, and page processing is adopted. Deal directly with page structures, not kernel virtual addresses.

```
for(i = 0; i < nfrags ; i++)

{

        skb_frag_t *frag =&skb_shinfo(skb)->frags[i];

        int len = frag->size;


        entry = (entry + 1) % txsize;

        desc = priv->dma_tx + entry;


        TX_DBG("\t[entry %d] segment len:%d\n", entry, len);

        desc->des2 = dma_map_page( priv->device, frag->page,

                    frag->page_offset,

                    len, DMA_TO_DEVICE);

        priv->tx_skbuff[entry] = NULL;

        priv->tx_page[entry] =frag->page;

        get_page(frag->page); (Need to check the principle)

       priv->hw->desc->prepare_tx_desc(desc, 0, len, csum_insertion);

       priv->hw->desc->set_tx_owner(desc);

}
```

5. Give the first data segment descriptor to GMAC, and record the current sending index

priv->hw->desc->set_tx_owner(first);

priv->cur_tx += count;

6. Activate RxDMA

STMMAC_SYNC_BARRIER();

Before processing the next instruction, first flush the DMA write buffers

priv->hw->dma->enable_dma_transmission(priv->dma_ioaddr,

priv->dma_channel);

Write any value to wake up the pending RxDMA

7. Send resource recovery interface stmmac_tx

When the data transmission is completed, an interrupt is generated, and the stmmac_poll function is called to enter the interface to perform the transmission resource recovery operation.

(1) Get the current transmit descriptor DMA index from the register

(2) Circularly judge dirty_tx and cur_tx

while(priv->dirty_tx != priv->cur_tx)

(3) Determine the DMA index and descriptor attribution

if (entry == hw_dma_index) // entry =dirty_tx % txsize

break;

if(priv->hw->desc->get_tx_owner(p))

break;

(4) skb joins the recycling receiving queue

if((skb_queue_len(&priv->rx_recycle) <priv->dma_rx_size)

&&skb_recycle_check(skb, priv->dma_buf_sz))

__skb_queue_head(&priv->rx_recycle,skb);

else

dev_kfree_skb(skb);

If the receive recovery queue does not exceed the total length of the receive queue (256), and skb can be recycled, add skb to the recovery queue. Others release skb.

(5) Release the page page of multiple data fragments

put_page (priv->tx_page[entry]);

priv->tx_page[entry]= NULL;

(6) Add dirty_tx

entry= (++priv->dirty_tx) % txsize;

11. Other Supplements

reference documents