**Hi35xx stmmac网卡驱动源码解读**

——何良斌

# 一、源文件

drivers/net/stmmac/stmmac\_main.c

源码阅读顺序如下图：



# 二、platform\_device\_register与platform\_driver\_register

platform\_device\_register：注册设备

platform\_driver\_register：注册驱动

两者通过name联系起来，所有两者的name必须要保持一致。

platform\_device\_register必须在驱动加载前完成，原因驱动注册时需要匹配内核中已经注册的设备名。

platform\_driver\_register注册时，会将当前注册的driver name和已注册的所有device中name进行比较，只有找到相同了的才能注册成功。匹配后就会调用probe进行检测设备，然后将该设备绑定到驱动上。

Platform是一个虚拟的地址总线，相比pci，USB，它主要用于描述SOC上的片上资源。使用约定：如果不属于任何总线的设备，例如网卡，串口等设备，都挂在这个虚拟总线上。

# 三、平台驱动数据结构

static struct platform\_driver stmmac\_driver = {

.probe = stmmac\_dvr\_probe,（见第四节）

.remove = stmmac\_dvr\_remove,

.driver = {

.name = STMMAC\_RESOURCE\_NAME,

.owner = THIS\_MODULE,

.pm = &stmmac\_pm\_ops,

},

};

static struct platform\_device stmmac\_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

};

# 四、驱动探针stmmac\_dvr\_probe

当平台驱动在注册时，如果和平台设备名字匹配了，就会调用该初始化接口。

## 1、接口操作基本流程

该函数流程图如下：



（1）获取系统配置寄存器基地址

ret = stmmac\_syscfg\_init(pdev);

（2）获取MAC控制器寄存器基地址

stmmac\_base\_ioaddr = ioremap\_nocache(res->start, res->end - res->start + 1);

（3）申请网卡设备和私有数据

ndev = alloc\_etherdev(sizeof(struct stmmac\_priv));

网卡设备和私有数据紧紧的挨在一起：网卡设备+私用数据结构，通过netdev\_priv获取私有数据。网卡设备是通用数据结构，私有数据则为各个MAC控制器的数据结构

（4）MAC控制器设置

ret = stmmac\_mac\_device\_setup(ndev);（见第2小节）

（5）网络设备真正注册

ret = stmmac\_probe(ndev);（重要，见第五节）

（6）MDIO总线注册

ret = stmmac\_mdio\_register(ndev);（见第3小节）

（7）将网卡设备赋给全局变量

stmmac\_device\_list[priv->id] = ndev;，在其他接口中会使用这个全局变量。

注：3~7步骤，如果是多网卡则循环执行。

（8）复位DMA，GMA等寄存器

stmmac\_reset();

（9）判断是否支持checksum offload engine

priv->rx\_coe = priv->hw->mac->rx\_coe(priv->ioaddr);在上面的第4步进行了函数初始化

（10）设置硬件DMA工作模式

stmmac\_dma\_operation\_mode(priv);（见第4小节）

（11）初始化RX/TX队列

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);（见第5小节）

（12）硬件DMA初始化

priv->hw->dma->init(priv->dma\_ioaddr,

priv->dma\_channel,

priv->plat->pbl,

priv->dma\_tx\_phy,

priv->dma\_rx\_phy, 0)

最重要的意见事情：由init\_dma\_desc\_rings 申请的priv->dma\_tx\_phy和priv->dma\_rx\_phy DMA handle写入接收和发送描述子链表地址寄存器

（13）接收发送index初始化

priv->dirty\_rx = priv->cur\_rx = 0;

priv->dirty\_tx = priv->cur\_tx = 0;

cur：已获取上传给网络协议层的数据帧index，协议层会释放skb

dirty：已经填充skb到dma队列的index

两个数值不断累加，通过取余%的方式循环处理dma队列。

如图所示：



（14）初始化接收回收队列

skb\_queue\_head\_init(&priv->rx\_recycle);

注意：9~11步骤，如果多网卡会循环执行。

（15）初始化TOE\_NK

ret = tnk\_init(stmmac\_base\_ioaddr, &pdev->dev);（见第6小节）

（16）请求中断

ret = request\_irq(SYNOP\_GMAC\_IRQNUM, stmmac\_interrupt, IRQF\_SHARED, STMMAC\_RESOURCE\_NAME, pdev);（重要，见第七节）

（17）使能所有中断

writel(~0, stmmac\_base\_ioaddr + TNK\_REG\_INTR\_EN);

## 2、MAC控制器设置 stmmac\_mac\_device\_setup(ndev);

（1）选择GMAC设备并初始化

device = dwmac1000\_setup(priv->ioaddr);

A．申请一个MAC设备：mac = kzalloc()；

B．GMAC操作: mac->mac = &dwmac1000\_ops;

C．GMAC dma操作: mac->dma = &dwmac1000\_dma\_ops;

（2）选择增强型描述子数据结构

device->desc = &enh\_desc\_ops；

注：以上很多都是对MAC寄存器的基本操作。

## 3、MII bus注册stmmac\_mdio\_register

（1）申请MII总线

stmmac\_mii\_bus = mdiobus\_alloc();

（2）初始化mdio工作时钟

tnkclk = mdio\_clk\_init();

（3）初始化MII总线结构体

stmmac\_mii\_bus->name = "STMMAC MII Bus";

stmmac\_mii\_bus->read = &stmmac\_mdio\_read;

stmmac\_mii\_bus->write = &stmmac\_mdio\_write;

stmmac\_mii\_bus->reset = &stmmac\_mdio\_reset;

……..

（4）注册MII总线mdiobus\_register(stmmac\_mii\_bus);

A．注册总线设备device\_register(&bus->dev);

B．复位总线bus->reset(bus);

C．扫描总线上的PHY设备，最大支持32个

phydev = mdiobus\_scan(bus, i);

主要做三件事情：读取PHY ID，创建PHY device，然后注册PHY设备。

创建PHY设备接口get\_phy\_device中会调用phy\_device\_create会对PHY设备数据结构初始化，然后初始化工作队列phy\_state\_machine，该接口中会调用adjust\_link接口。

（5）将MII总线赋给stmmac私有数据

priv->mii = stmmac\_mii\_bus;

## 4、设置硬件DMA工作模式stmmac\_dma\_operation\_mode

priv->hw->dma->dma\_mode(priv->dma\_ioaddr, priv->dma\_channel,

tc, SF\_DMA\_MODE);

发送：tc默认为64，不支持DMA存储转发。TxFIFO根据FIFO阀值来判断是否转发

接收：支持DMA存储转发。当RxFIFO接收到一个完整帧后才将该帧向上转发。

## 5、初始化dma描述子队列init\_dma\_desc\_rings

（1）RX队列申请

priv->rx\_skbuff\_dma = kmalloc(rxsize \* sizeof(dma\_addr\_t), GFP\_KERNEL);

申请256个skb对应的dma地址队列

priv->rx\_skbuff =kmalloc(sizeof(struct sk\_buff \*) \* rxsize, GFP\_KERNEL);

申请256个skb队列

priv->dma\_rx =(struct dma\_desc \*) dma\_alloc\_coherent(priv->device,

rxsize \*sizeof(struct dma\_desc),

&priv->dma\_rx\_phy,

GFP\_KERNEL);

DMA描述符（在datasheet为接收描述子）建立一致性DMA映射，大小为256\* sizeof(struct dma\_desc)，返回缓冲区虚拟内存地址。priv->dma\_rx\_phy返回正确的DMA handle，个人感觉是虚拟地址对应的物理地址，因为这个地址需要写入接收描述子链表地址寄存器。（该函数不是很明白）

（2）TX队列申请

priv->tx\_skbuff = kmalloc(sizeof(struct sk\_buff \*) \* txsize, GFP\_KERNEL);

申请256个skb指针

priv->tx\_page = kmalloc(sizeof(struct page \*) \* txsize, GFP\_KERNEL);

申请256个page指针

priv->dma\_tx = (struct dma\_desc \*)dma\_alloc\_coherent(priv->device,

txsize \*sizeof(struct dma\_desc),

&priv->dma\_tx\_phy,

GFP\_KERNEL);

同上

（3）RX 队列初始化以及关联（1）中申请的数据

通过for循环申请256个skb，每个skb大小为2KB。

A．skb = netdev\_alloc\_skb\_ip\_align(dev, bfsize);

申请skb数据

B．priv->rx\_skbuff[i] = skb;

将skb放入接收队列

C． priv->rx\_skbuff\_dma[i] = dma\_map\_single(priv->device, skb->data, bfsize, DMA\_FROM\_DEVICE);

然后将skb数据通过流式DMA映射加入skb dma地址队列

D．(priv->dma\_rx +i)->des2 = priv->rx\_skbuff\_dma[i];

再将skb dma地址放入dma描述符

（4）TX队列没有类似接收的RX初始化操作

for (i = 0; i < txsize; i++) {

priv->tx\_skbuff[i] = NULL;

priv->tx\_page[i] = NULL;

priv->dma\_tx[i].des2 = 0;

}

因为skb是在网络层被申请，传递到驱动中来。

（5）初始化RX/TX描述符

priv->hw->desc->init\_rx\_desc(priv->dma\_rx, rxsize, dis\_ic);

priv->hw->desc->init\_tx\_desc(priv->dma\_tx, txsize);

（6） dma描述子队列和skb映射图



上面是接收DMA描述符队列和skb初始化时建立映射过程。对于发送DMA发送队列只有在发送时才会建立映射，如果一个数据包存在多个数据片段，还会调用dma\_map\_page接口进行页映射处理。

（7）skb申请分配空间图

skb = netdev\_alloc\_skb\_ip\_align(dev, bfsize);



skb->len:数据包中全部数据的长度，skb->data\_len分隔存储数据片段长度。当只有一个数据片段时data\_len = 0。

## 6、TOE\_NK初始化tnk\_init()

（1）建立proc目录

ret = tnk\_proc\_init(hitoe ? max\_connections : 0);

（2）如果使用TOE，则对TOE进行进行一系列初始化

由于没有使用，所以此处不讲述。

# 五、真正注册网络设备stmmac\_probe

## 1、接口基本操作流程

该函数基本流程图：



（1）初始化网卡设备以太网基本信息ether\_setup();

dev->header\_ops = &eth\_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）初始化网卡操作函数和ethtool操作函数

dev->netdev\_ops = &stmmac\_netdev\_ops;（见第2小节）

stmmac\_set\_ethtool\_ops(dev);（见第3小节）

dev->features |= NETIF\_F\_SG | NETIF\_F\_HIGHDMA | NETIF\_F\_IP\_CSUM

| NETIF\_F\_IPV6\_CSUM;

设置网卡接口特性，这些特性驱动和协议层将会用到

dev->watchdog\_timeo = msecs\_to\_jiffies(watchdog); 设置发送超时，超时后最终会调用stmmac\_tx\_timeout这个进行超时处理

（3）初始化NAPI轮询接收数据包接口

netif\_napi\_add(dev, &priv->napi, stmmac\_poll, 64); （重要，见第八节）

初始化napi，poll=stmmac\_poll，weight=64，poll\_list等信息

Weight: 描述接口的相对重要性，当资源紧张时，在接口上能承受多大的流量。

（4）从寄存器中获取MAC地址，并判断有效性

priv->hw->mac->get\_umac\_addr((void \_\_iomem \*)dev->base\_addr, dev->dev\_addr, 0);

（5）初始化自旋锁

spin\_lock\_init(&priv->lock);

tnk\_lock\_init(&priv->tlock);

（6）注册网卡设备

ret = register\_netdev(dev);

## 2、网卡基本操作函数

static const struct net\_device\_ops stmmac\_netdev\_ops = {

.ndo\_open = stmmac\_open,（见第六节）

.ndo\_start\_xmit = stmmac\_xmit,（见第九节）

.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操作函数

static struct ethtool\_ops stmmac\_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,

};

# 六、打开网卡设备stmmac\_open

## 1、基本流程

基本流程图：



（1）MAC地址有效性

is\_valid\_ether\_addr(dev->dev\_addr)

（2）修正传递给驱动参数

stmmac\_verify\_args();

（3）初始化PHY设备

stmmac\_init\_phy(dev);（见第2小节）

（4）设备MAC地址到硬件

priv->hw->mac->set\_umac\_addr(priv->ioaddr, dev->dev\_addr, 0);

（5）初始化MAC core

priv->hw->mac->core\_init(priv->ioaddr);

（6）使能NAPI并schedule

napi\_enable(&priv->napi);

napi\_schedule(&priv->napi)

（7）启动DMA收发（写寄存器）

priv->hw->dma->start\_tx(priv->dma\_ioaddr, priv->dma\_channel);

priv->hw->dma->start\_rx(priv->dma\_ioaddr, priv->dma\_channel);

（8）启动相关定时器

priv->poll\_timer.function = stmmac\_poll\_func;

定时唤醒处于挂起状态的RxDMA/TxDMA

priv->check\_timer.function = stmmac\_check\_func;

读取寄存器地址获取当前接收描述符地址，如果在cur\_rx和dirty\_cur之前，则设置描述子归属为GMAC操作。

（9）使能MAC TX/RX

stmmac\_enable\_mac(priv->ioaddr);

（10）启动队列

netif\_start\_queue(dev);

## 2、初始化PHY设备stmmac\_init\_phy

（1）通知内核载波消失

netif\_carrier\_off(dev);

（2）以太网设备连接到PHY设备

phydev = phy\_connect(dev, phy\_id, &stmmac\_adjust\_link, 0, priv->phy\_interface);

stmmac\_adjust\_link：根绝实际PHY情况，调整连接参数：全双工/半双功，速度，MII/GMII

A．d = bus\_find\_device\_by\_name(&mdio\_bus\_type, NULL, bus\_id);

在MDIO bus总线PHY device列表中找到指定的设备

B．设备转化为PHY设备

phydev = to\_phy\_device(d);

C．rc = phy\_connect\_direct(dev, phydev, handler, flags, interface);

连接以太网device到指定的PHY 设备，然后启动PHY。

# 七、中断处理函数stmmac\_interrupt

中断里3个并行流程：网卡1、网卡2、TOE，网卡和网卡2处理流程一样，且未使用TOE，只分析网卡1。

接收到第一个数据包或者数据发送完成时，产生中断，进入该中断处理函数。

## 1、读取中断状态寄存器

不管发送或者接收，产生DMA0中断时，则进入dma中断处理函数

## 2、stmmac\_dma\_interrupt

status = priv->hw->dma->dma\_interrupt(priv->dma\_ioaddr, priv->dma\_channel,

&priv->xstats);

读取dma中断状态，如果是正确的接收和发送，则进行NAPI调度\_stmmac\_schedule(priv)。

里面调用napi\_schedule(&priv->napi)，同时禁止中断（中断由stmmac\_poll里面开启）。

## 3、napi\_schedule简单说明

（1）将priv->napi加入到softnet\_data->napi，从定义看每个CPU有一个softnet\_data

（2）启动网络接收软中断NET\_RX\_SOFTIRQ

（3）在CPU一定时间内处理该软中断

（4）软中断处理函数为net\_rx\_action，里面则会调用n->poll进行相应处理（该处的poll为下面probe是注册的stmmac\_poll，见后面章节说明）

（5）软中断处理函数在subsys\_initcall(net\_dev\_init)时注册

open\_softirq(NET\_TX\_SOFTIRQ, net\_tx\_action);

open\_softirq(NET\_RX\_SOFTIRQ, net\_rx\_action);

（6）最后在do\_softirq中统一处理所有的软中断。

常见软中断类型：

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、中断里没对接收到的数据进行处理，而是调用NAPI接口在适当的时间进行处理。

# 八、NAPI接口之stmmac\_poll

该接口完成两个事情：一个是当数据发送完成时产生中断，进入该函数进行资源回收；另外一个是收到数据产生中断，进入该函数进行数据接收和处理。

## 1、基本操作流程

（1）接口开始时记录proc信息

stmmac\_poll\_begin();

（2）回收已经发送完成的资源（放在发送章节讲解）

stmmac\_tx(priv);（见第九节7小节）

（3）接收数据包

stmmac\_rx(priv, budget);（见第2小节）

（4）如果接收到的数据包没有超过接口的容量，则开启中断

if (work\_done < budget) {

stm\_proc.polls\_done++;

napi\_complete(napi);

stmmac\_enable\_irq(priv);

}

（5）结束接口调用

stmmac\_poll\_end();

## 2、数据包接收处理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)

判断当前描述子的归属：描述子数据结构中OWN位，0:当前描述子应该由CPU操作，1:前描述子应该由GMAC操作。对于接收，初始化dma描述子队列时设置为1。

GMAC根据寄存器配置，获取可用接收描述子，然后从RxFIFO中读取从PHY接收的Ethernet报文，如果报文符合接收条件，将该报文写入接收描述子指向的数据缓冲区，并回写接收描述子。这个回写就会将OWN位设置为0。

（2）获取下一帧描述符

next\_entry = (++priv->cur\_rx) % rxsize;

p\_next = priv->dma\_rx + next\_entry;

priv->cur\_rx：已经传递给协议层的index

（3）获取收到帧状态

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

如果是丢弃帧，则什么都不处理；否则上传到上层网络。

（4）获取帧长度

frame\_len = priv->hw->desc->get\_rx\_frame\_len(p);

（5）获取帧数据

skb = priv->rx\_skbuff[entry];

priv->rx\_skbuff[entry] = NULL;

注意：skb将有上层网络处理完后进行释放。

（6）设置skb数据长度和解除流式DMA映射

skb\_put(skb, frame\_len);

dma\_unmap\_single(priv->device, priv->rx\_skbuff\_dma[entry], priv->dma\_buf\_sz, DMA\_FROM\_DEVICE);

（7）获取skb的协议类型

skb->protocol = eth\_type\_trans(skb, priv->dev);

skb->dev = priv->dev;

（8）将skb通过NAPI接口上传上层网络协议处理

napi\_gro\_receive(&priv->napi, skb);（见第九节）

注：以上是一个while循环操作

（9）重新填充接收队列

stmmac\_rx\_refill(priv);（见第3小节）

## 3、接收队列填充函数stmmac\_rx\_refill

由第2小节，接收到数据的skb从dma接收队列剥离，传递给网络层处理完成后，会将skb给释放。所以，需要重新申请新的skb传递填充dma接收队列。

（1）判断是否需要重新填充

将priv->cur\_rx和priv->dirty\_rx进行对比，如果priv->cur\_rx > priv->dirty\_rx即上传给网络协议层的数量大于已经填充dma的数量时，才会重新填充dma接收队列。

（2）检查描述子操作归属

priv->hw->desc->get\_rx\_owner(p + entry)，只有属于CPU的，才进行填充操作。

（3）申请新的SKB

skb = \_\_skb\_dequeue(&priv->rx\_recycle);

先从回收队列中获取有效的skb。该回收队列来自由协议层申请且发送完成的数据帧skb。在stmmac\_tx接口进行回收，它没有释放skb，而是放在了回收队列中，提高利用效率。

skb = netdev\_alloc\_skb\_ip\_align(priv->dev,

若回收队列没有，则重新申请skb。

（4）填充接收DMA描述符队列

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）设置描述子归属，交给GMAC操作

priv->hw->desc->set\_rx\_owner(p + entry);

注：以上通过for循环进行，直到没有需要填充为止。

（6）激活RxDMA

STMMAC\_SYNC\_BARRIER();

处理下一条指令前，先flush一下DMA write buffers

priv->hw->dma->enable\_dma\_receive(priv->dma\_ioaddr, priv->dma\_channel);

写任何值唤醒处于挂起的RxDMA。

# 九、传递skb数据给协议栈napi\_gro\_receive（generic receive offload）

## 1、概念了解

TSO、UFO和GSO是对应网络发送，LRO、GRO是对应网络接收。

TSO（TCP Segmentation Offload）是一种利用网卡对TCP数据包分片，减轻CPU负荷的一种技术，也叫LSO（Large segment offload），TSO—TCP，UFO—UDP。如果硬件支持TSO功能，也需要硬件支持的TCP校验计算和分散/聚集功能。

GSO（Generic Segmentation Offload）比TSO更通用。基本思想：尽可能的推迟数据分片直至发送到网卡驱动之前，此时会检查网卡是否支持分片功能（TSO、UFO），如果支持直接发送到网卡，如果不支持就进行分片后再发往网卡。这样大数据包只需走一次协议栈，而不是被分割成几个数据包分别走，提高效率。

LRO（Large Receive Offlaod） 通过接收到的多个TCP数据聚合成一个大的数据包，然后传递给网络协议层处理，以减少上层协议栈处理开销，提高系统接收TCP数据包能力

GRO（Generic Receive Offload）基本思想跟LRO类似，但是更加通用，现在驱动基本使用GRO接口，而不是LRO。

RSS（Receive Side Scaling）是一项网卡的特性，俗称多队列。具备多个RSS队列的网卡，可以将不同的网络流分成不同的队列，再分别将这些队列分配到多个CPU核心进行处理，从而将负荷分散，充分利用多核处理器的能力。

## 2、GRO结构体struct napi\_gro\_cb

GRO功能使用skb结构体内私有空间char cb[48]来存放gro所用到的一些信息。

具体内容如下：

struct napi\_gro\_cb {

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

指向存在skb\_shinfo(skb)->frag[0].page页的数据的头部 。见skb\_gro\_reset\_offset

void \*frag0;

/\* Length of frag0. 第一页中数据的长度 \*/

unsigned int frag0\_len;

/\* This indicates where we are processing relative to skb->data. 表明skb->data到GRO需要处理的数据区的偏移量 \*/

比如进入ip层进行GRO处理，这时skb->data指向ip 头， 而ip层的gro 正好要处理ip头，这时偏移量就为0。进入传输层后进行GRO处理，这时skb->data还指向ip头， 而tcp层gro要处理tcp头，这时偏移量就是ip头部长度

int data\_offset;

/\* This is non-zero if the packet may be of the same flow. 挂在napi->gro\_list上的报文是否跟现在的报文进行匹配\*/

每层的gro\_receive都设置该标记位。 接收到一个报文后，使用该报文和挂在napi->gro\_list 上 的报文进行匹配。 在链路层，使用dev 和 mac头进行匹配，如果一样表示两个报文是通一个 设备发过来的，就标记napi->gro\_list上对应的skb的same为1. 到网络层，再进一步进行匹配时，只需跟napi->list上刚被链路层标记 same为1的报文进行网络层的匹配即可，不需再跟每个报文进行匹配。 如果网络层不匹配，就清除该标记。 到传输层，也是只配置被网络层标记same为1 的报文即可。这样设计为的是减少没必要的匹配操作

int same\_flow;

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

如果该字段不为0，表示该数据报文没必要再等待合并，可以直接送进协议栈进行处理了

int flush;

/\* Number of segments aggregated. 该报文被合并过的次数\*/

int count;

/\* Free the skb? 是否该被丢弃\*/

int free;

};

## 3、函数基本流程

（1）skb\_gro\_reset\_offset（见第4小节）

初始化NAPI\_GRO\_CB;

（2）\_\_napi\_gro\_receive（见第6小节）

连路层gro\_receive，实现数据包合并还是上传协议栈

（3）napi\_skb\_finish（见第5小节，内容少先讲）

根据第二个函数的返回值决定合并，feed协议栈，还是free。

## 4、初始化GRO cb: skb\_gro\_reset\_offset(skb)

支持S/G IO的网卡存在如下一种可能：skb本身不包含数据（头也没有），所有的数据都保存在skb\_share\_info中，且frags.page不在higih\_mem。此种情况如果要合并，将包头信息取出来即skb\_shared\_info的frags[0]，将头信息保存到napi\_gro\_cb 的frags0中。

具体源码如下：

NAPI\_GRO\_CB(skb)->data\_offset = 0;

NAPI\_GRO\_CB(skb)->frag0 = NULL;

NAPI\_GRO\_CB(skb)->frag0\_len = 0;

如果mac\_header和skb->tail相等并且地址不在高端内存，则说明包头保存在skb\_shinfo中，所以我们需要从frags中取得对应的数据包

if (skb->mac\_header == skb->tail && !PageHighMem(skb\_shinfo(skb)->frags[0].page)) {

可以看到frag0保存的就是对应的skb的frags的第一个元素的地址

NAPI\_GRO\_CB(skb)->frag0 = page\_address(skb\_shinfo(skb)->frags[0].page) +

skb\_shinfo(skb)->frags[0].page\_offset;

然后保存对应的大小

NAPI\_GRO\_CB(skb)->frag0\_len = skb\_shinfo(skb)->frags[0].size;

}

## 5、napi\_skb\_finish

根据\_\_napi\_gro\_receive的返回值处理合并后的数据包。

switch (ret) {

//将数据包送进协议栈

case GRO\_NORMAL:

if (netif\_receive\_skb(skb))（见第8小节）

ret = GRO\_DROP;

break;

//报文可以丢弃或者已经合并进gro，则free报文

case GRO\_DROP:

case GRO\_MERGED\_FREE:

kfree\_skb(skb);

break;

//数据已经被gro保存起来，但是并没有合并的，skb还需要保留不能释放。

case GRO\_HELD:

case GRO\_MERGED:

break;

}

return ret;

## 6、\_\_napi\_gro\_receive接收合并函数

（1）基本概念

每个协议中定义自己的GRO接收合并函数和合并后处理函数，即gro\_receive和gro\_complete。GRO系统会根据协议来调用对应回调函数。gro\_receive将 skb合并到gro\_list中，返回值：空则表示合并后无需送入协议栈，非空则需要立即送入协议栈。gro\_complete则是当返回值非空时将gro合并数据包送到协议栈时被调用。

\_\_napi\_gro\_receive和napi\_gro\_complete可以被看做是链路层的gro\_receive和gro\_complete。

各个协议层gro\_receive:

.gro\_receive = tcp4\_gro\_receive,

.gro\_receive = inet\_gro\_receive,

.gro\_receive = ipv6\_gro\_receive,

各个协议层gro\_complete:

.gro\_complete = tcp4\_gro\_complete,

.gro\_complete = inet\_gro\_complete,

.gro\_complete = ipv6\_gro\_complete,

（2）函数基本内容

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;

}

遍历gro\_list，查找列表中skb是否有和当前skb相同的流，然后给same\_flow赋值。如果此层（链路层）相同，则ip层、tcp层再进行相同流判断，否则不会进行判断。不同层流是否相同的判断条件不一样，此处判断3个判断条件：同一设备、同一VLAN、同一MAC头。

B． return dev\_gro\_receive(napi, skb);（见第7节）

真正的接收合并处理

7、dev\_gro\_receive真正的接收合并

可以分为两部分看待，一个是正常的合并处理，另外一个就是frags0处理部分。需要注意GRO不支持切片的IP包，IP切片的组包在内核IP层会做一遍，GRO再做没意义增加复杂度。

（1）合并相同流处理

A．先遍历对应的ptype（协议的类链表），找到匹配协议后，调用对应的回调函数gro\_receive；链路层则调用ip层的gro\_receive即inet\_gro\_receive。

B．进入ip层合并处理，主要判断是否same\_flow和是否需要flush。只有两个包是same\_flow的情况下才会flush判断

same\_flow判断：协议需要相同，tos域需要相同，源和目的地址需要相同；只要一个不同就是设置为0。

flush判断：是切片包，ttl不一样，id顺序不对；只要一个满足skb就会flush出gro到协议栈。然后进入TCP层的gro\_receive即tcp4\_gro\_receive。

C．进入TCP层合并处理：类似IP层，对same\_flow和flush判断。其中对flush的判断比较多和复杂，如果需要flush就不需要进行合并处理，返回相应的gro\_list。

真正的合并函数：skb\_gro\_receive（此处暂不分析）。

（2）当gro\_receive返回值非空时，对flush出gro的skb进行立即feed进协议栈处理napi\_gro\_complete。如果当前skb的same\_flow非0表明找到同流且已合并直接返回，如果未找到同流，则添加skb到gro\_list中。

（3）frags0处理部分

将skb\_shinfo结构体存放的frags，往前移动到skb。（为啥这么做？）

8、netif\_receive\_skb

最终将skb数据分发给各个协议层。有空再讲解。

# 十、发送数据stmmac\_xmit

该接口实现了Scatter/Gather I/O功能，通过skb\_shinfo宏来判断数据包是一个数据片段组成，还是由大量数据片段组成。

## 1、获取可用发送描述子

entry = priv->cur\_tx % txsize;

desc = priv->dma\_tx + entry;

first = desc; 保存第一个数据片段

## 2、将skb放到发送队列

priv->tx\_skbuff[entry] = skb;

priv->tx\_page[entry] = NULL;

## 3、发送单个或第一个数据包

unsigned int 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);

当只有一个数据片段时，skb->data将发送所有数据；当有多个数据片段时，skb->data则指向第一个数据片段，数据长度skb->len – skb->data\_len，其他数据存放在共享数据结构frags数组中。（skb->len:数据包中全部数据的长度，skb->data\_len分隔存储数据片段长度）

## 4、发送剩余数据片段

对于多个数据片段时，还要进行数据片段的发送，采用页处理。直接处理页结构，而不是内核虚拟地址。

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);（需要查一下原理）

priv->hw->desc->prepare\_tx\_desc(desc, 0, len, csum\_insertion);

priv->hw->desc->set\_tx\_owner(desc);

}

## 5、将第一个数据片段描述子交给GMAC，记录当前发送index

priv->hw->desc->set\_tx\_owner(first);

priv->cur\_tx += count;

## 6、激活RxDMA

STMMAC\_SYNC\_BARRIER();

处理下一条指令前，先flush一下DMA write buffers

priv->hw->dma->enable\_dma\_transmission(priv->dma\_ioaddr,

priv->dma\_channel);

写任何值唤醒处于挂起的RxDMA

## 7、发送资源回收接口stmmac\_tx

当数据发送完成时产生中断，调用stmmac\_poll函数进入该接口进行发送资源回收操作。

（1）从寄存器获取当前发送描述符DMA index

（2）循环判断dirty\_tx 和 cur\_tx

while (priv->dirty\_tx != priv->cur\_tx)

（3）判断DMA index 和 描述符归属

if (entry == hw\_dma\_index) // entry = dirty\_tx % txsize

break;

if (priv->hw->desc->get\_tx\_owner(p))

break;

（4）skb加入回收接收队列

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);

接收回收队列没有超出接收队列总长度（256），且skb可以进行回收的情况下，将skb加入回收队里。其他就释放skb。

（5）释放多数据数据片段的page页

put\_page(priv->tx\_page[entry]);

priv->tx\_page[entry] = NULL;

（6）增加dirty\_tx

entry = (++priv->dirty\_tx) % txsize;

# 十一、其他补充

参考文档

