STMAC Network Driver Understanding Self Note

Contents

[1. Platform or Chip specific Nw Driver’s probe function: - 1](#_Toc163061766)

[2. DWC or mainline driver specific Nw Driver’s probe function: - 2](#_Toc163061767)

[3. Allocating Tx and Rx Queues: - 2](#_Toc163061768)

# DWC Platform or Chip specific Nw Driver’s probe function: -

File: - drivers/net/ethernet/stmicro/stmmac/dwmac-sxgmac.c

static int dwmac\_sxgmac\_probe(struct platform\_device \*pdev)

{

//….

sxgmac = devm\_kzalloc(&pdev->dev, sizeof(\*sxgmac), GFP\_KERNEL);

sxgmac->base = devm\_platform\_ioremap\_resource\_byname(pdev, "sxgmac");

//….

ret = stmmac\_dvr\_probe(&pdev->dev, plat\_dat, &stmmac\_res);

//….

}

static const struct of\_device\_id dwmac\_sxgmac\_match[] = {

{ .compatible = "samsung,sxgmac"},

{ }

};

MODULE\_DEVICE\_TABLE(of, dwmac\_sxgmac\_match);

static struct platform\_driver dwmac\_sxgmac\_driver = {

.probe = dwmac\_sxgmac\_probe,

.remove = dwmac\_sxgmac\_remove,

.driver = {

.name = "sxgmac",

.pm = &dwmac\_sxgmac\_pm\_ops,

.of\_match\_table = of\_match\_ptr(dwmac\_sxgmac\_match),

},

};

module\_platform\_driver(dwmac\_sxgmac\_driver);

# Mainline driver specific Nw Driver’s probe function: -

File: - drivers/net/ethernet/stmicro/stmmac/stmmac\_main.c

int stmmac\_dvr\_probe(struct device \*device, struct plat\_stmmacenet\_data \*plat\_dat, struct stmmac\_resources \*res)

{

struct net\_device \*ndev = NULL;

/\*Allocate Driver's Ring Buffer\*/

ndev = devm\_alloc\_etherdev\_mqs(device, sizeof(struct stmmac\_priv), MTL\_MAX\_TX\_QUEUES, MTL\_MAX\_RX\_QUEUES);

SET\_NETDEV\_DEV(ndev, device);

priv = netdev\_priv(ndev);

priv->device = device;

priv->dev = ndev;

//...

stmmac\_set\_ethtool\_ops(ndev);

//...

priv->plat = plat\_dat;

priv->ioaddr = res->addr;

priv->dev->base\_addr = (unsigned long)res->addr;

//....

dev\_set\_drvdata(device, priv->dev);

//....

ret = stmmac\_hw\_init(priv);

//….

ndev->netdev\_ops = &stmmac\_netdev\_ops;

/\* Setup channels NAPI \*/

stmmac\_napi\_add(ndev);

ret = stmmac\_mdio\_register(ndev);

//...

ret = register\_netdev(ndev);

//...

}

# net\_device\_ops: -

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

.ndo\_open = stmmac\_open,

.ndo\_start\_xmit = stmmac\_xmit,

.ndo\_stop = stmmac\_release,

.ndo\_change\_mtu = stmmac\_change\_mtu,

.ndo\_fix\_features = stmmac\_fix\_features,

.ndo\_set\_features = stmmac\_set\_features,

.ndo\_set\_rx\_mode = stmmac\_set\_rx\_mode,

.ndo\_tx\_timeout = stmmac\_tx\_timeout,

.ndo\_eth\_ioctl = stmmac\_ioctl,

.ndo\_siocdevprivate = stmmac\_ioctl\_priv,

.ndo\_setup\_tc = stmmac\_setup\_tc,

.ndo\_select\_queue = stmmac\_select\_queue,

#ifdef CONFIG\_NET\_POLL\_CONTROLLER

.ndo\_poll\_controller = stmmac\_poll\_controller,

#endif

.ndo\_set\_mac\_address = stmmac\_set\_mac\_address,

.ndo\_vlan\_rx\_add\_vid = stmmac\_vlan\_rx\_add\_vid,

.ndo\_vlan\_rx\_kill\_vid = stmmac\_vlan\_rx\_kill\_vid,

.ndo\_bpf = stmmac\_bpf,

.ndo\_xdp\_xmit = stmmac\_xdp\_xmit,

.ndo\_xsk\_wakeup = stmmac\_xsk\_wakeup,

};

static void stmmac\_set\_rx\_mode(struct net\_device \*dev)

{

struct stmmac\_priv \*priv = netdev\_priv(dev);

stmmac\_set\_filter(priv, priv->hw, dev);

}

# Allocating Tx and Rx Queues: -

Function call trace: -

------------------------------

stmmac\_dvr\_probe -> devm\_alloc\_etherdev\_mqs -> alloc\_etherdev\_mqs -> alloc\_netdev\_mqs -> netif\_alloc\_netdev\_queues

struct net\_device \*alloc\_netdev\_mqs(int sizeof\_priv, const char \*name,

unsigned char name\_assign\_type,

void (\*setup)(struct net\_device \*),

unsigned int txqs, unsigned int rxqs)

{

//....

dev->num\_tx\_queues = txqs;

dev->real\_num\_tx\_queues = txqs;

if (netif\_alloc\_netdev\_queues(dev))

goto free\_all;

dev->num\_rx\_queues = rxqs;

dev->real\_num\_rx\_queues = rxqs;

if (netif\_alloc\_rx\_queues(dev))

goto free\_all;

//...

}

**Tx Queue Allocation: -**

static int netif\_alloc\_netdev\_queues(struct net\_device \*dev)

{

unsigned int count = dev->num\_tx\_queues;

struct netdev\_queue \*tx;

size\_t sz = count \* sizeof(\*tx);

if (count < 1 || count > 0xffff)

return -EINVAL;

tx = kvzalloc(sz, GFP\_KERNEL\_ACCOUNT | \_\_GFP\_RETRY\_MAYFAIL);

if (!tx)

return -ENOMEM;

dev->\_tx = tx;

netdev\_for\_each\_tx\_queue(dev, netdev\_init\_one\_queue, NULL);

spin\_lock\_init(&dev->tx\_global\_lock);

return 0;

}

Tx Queue structure: -

struct [**netdev\_queue**](https://elixir.bootlin.com/linux/v6.10.6/C/ident/netdev_queue) {

*/\**

*\* read-mostly part*

*\*/*

struct [**net\_device**](https://elixir.bootlin.com/linux/v6.10.6/C/ident/net_device) \*dev;

[**netdevice\_tracker**](https://elixir.bootlin.com/linux/v6.10.6/C/ident/netdevice_tracker) [**dev\_tracker**](https://elixir.bootlin.com/linux/v6.10.6/C/ident/dev_tracker);

struct [**Qdisc**](https://elixir.bootlin.com/linux/v6.10.6/C/ident/Qdisc) [**\_\_rcu**](https://elixir.bootlin.com/linux/v6.10.6/C/ident/__rcu) \*[**qdisc**](https://elixir.bootlin.com/linux/v6.10.6/C/ident/qdisc);

struct [**Qdisc**](https://elixir.bootlin.com/linux/v6.10.6/C/ident/Qdisc) [**\_\_rcu**](https://elixir.bootlin.com/linux/v6.10.6/C/ident/__rcu) \*[**qdisc\_sleeping**](https://elixir.bootlin.com/linux/v6.10.6/C/ident/qdisc_sleeping);

#ifdef [**CONFIG\_SYSFS**](https://elixir.bootlin.com/linux/v6.10.6/K/ident/CONFIG_SYSFS)

struct [**kobject**](https://elixir.bootlin.com/linux/v6.10.6/C/ident/kobject)

//….

}

Rx Queue allocation: -

static int netif\_alloc\_rx\_queues(struct net\_device \*dev)

{

unsigned int i, count = dev->num\_rx\_queues;

struct netdev\_rx\_queue \*rx;

size\_t sz = count \* sizeof(\*rx);

int err = 0;

BUG\_ON(count < 1);

rx = kvzalloc(sz, GFP\_KERNEL\_ACCOUNT | \_\_GFP\_RETRY\_MAYFAIL);

if (!rx)

return -ENOMEM;

dev->\_rx = rx;

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

rx[i].dev = dev;

/\* XDP RX-queue setup \*/

err = xdp\_rxq\_info\_reg(&rx[i].xdp\_rxq, dev, i, 0);

if (err < 0)

goto err\_rxq\_info;

}

return 0;

}

Rx Queue structure: -

/\* This structure contains an instance of an RX queue. \*/

struct netdev\_rx\_queue {

struct xdp\_rxq\_info xdp\_rxq;

#ifdef CONFIG\_RPS

struct rps\_map \_\_rcu \*rps\_map;

struct rps\_dev\_flow\_table \_\_rcu \*rps\_flow\_table;

#endif

struct kobject kobj;

struct net\_device \*dev;

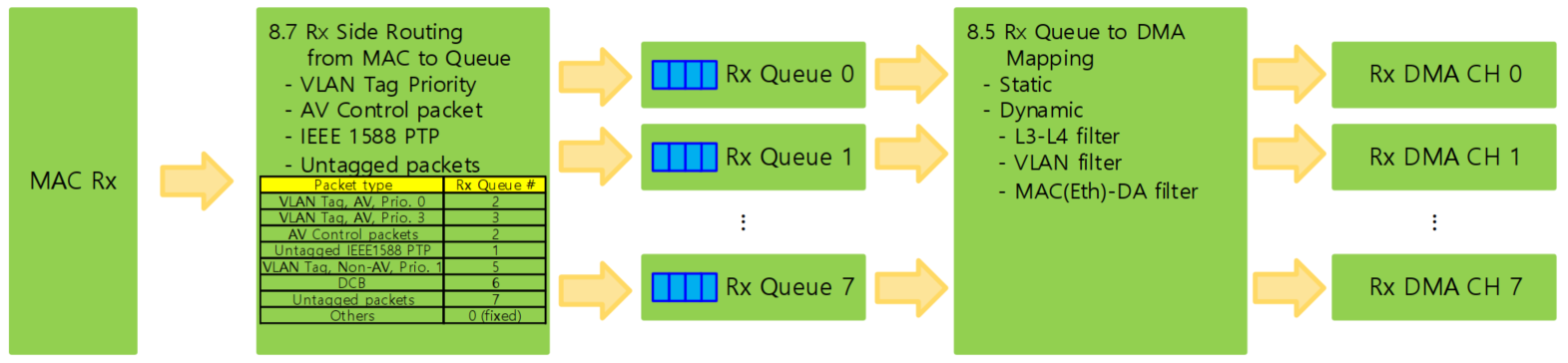
#ifdef CONFIG\_XDP\_SOCKETS

struct xsk\_buff\_pool \*pool;

#endif

} \_\_\_\_cacheline\_aligned\_in\_smp;

# Rx Packet flow concept: -



In Rx side, there are 2 functions to control Rx Packet Flow: -

1. Routing from MAC to Queue
2. Mapping between Rx Queues and Rx DMA CHs
3. **Routing from MAC to Queue: -**

We can map Rx Queues with a packet type.

For example, in figure above,

* We can route received packets, which have VLAN TAG with AV type and PCP (Priority Code Point) 0x000, to Rx Queue 2.
* We can route received untagged IEEE 1588 PTP packets to Rx Queue 1.
* If we do not set up the class for received packets, they go to Rx Queue 0 by default.

1. **Mapping between Rx Queues and Rx DMA CHs**

* We can map a specific Rx Queue with a specific Rx DMA CH.
* Each Rx Queue can select one of method for mapping to Rx DMA CH.

1. Static Mapping
2. Dynamic Mapping
3. Static Mapping: -

We can route all packets from a specific Queue to specific Rx DMA CH.

For example, Rx Queue 0's all received packets can go to Rx DMA CH 0. Or we can send all Rx Queue 1's packets to Rx DMA CH 2.

1. Dynamic Mapping: -

For Dynamic Mapping, Synopsys IP supports RxPacketFilter.

If we select Dynamic Mapping in an Rx Queue, the Rx Queue's packets will be classified by RxPacketFilter configuration.

A diagram of a computer network

Description automatically generated