# BMW IDCEvo :: Ethernet driver - sxgmac/stmmac

## Fixed Link for switch

The Ethernet node requires the specification on how the MAC controller is connected to the external interface. For instances where a Non-MDIO managed device is used, we need to use a fixed-link.

```
&gmac0
{ ....
fixed-link {
                speed = <1000>;
                full-duplex;
        };
.....
}
```

## Code Flow

Collect the data about the PHY through the platform driver

Configure the necessary in the Ethernet driver

#### stmmac_probe_config_dt

This sets the the phylink_node as the ethernet node.

```
/* PHYLINK automatically parses the phy-handle property */
        plat->phylink_node = np;
```

#### stmmac_dvr_probe

Needs to setup the phy connection

Here we populate the phylink struct

plcfg_link_an_mode is configured for fixed-phy.

```
stmmac_dvr_probe(*)
{..
/* Setup the phylink */
ret = stmmac_phy_setup(priv);
.....}

/*Configure the phylink struct */
static int stmmac_phy_setup(struct stmmac_priv *priv)
{
        struct stmmac_mdio_bus_data *mdio_bus_data = priv->plat->mdio_bus_data;
        struct fwnode_handle *fwnode = of_fwnode_handle(priv->plat->phylink_node);
        int mode = priv->plat->phy_interface;  /* Refers to phy-mode(rgmii-id)
        struct phylink *phylink;

        priv->phylink_config.dev = &priv->dev->dev;
        priv->phylink_config.type = PHYLINK_NETDEV; /*Ethernet PHY */
        priv->phylink_config.pcs_poll = true;
        if (priv->plat->mdio_bus_data)
                priv->phylink_config.ovr_an_inband =
                        mdio_bus_data->xpcs_an_inband;
```

```
        if (!fwnode)
                fwnode = dev_fwnode(priv->device);

        phylink = phylink_create(&priv->phylink_config, fwnode,
                                 mode, &stmmac_phylink_mac_ops);
        if (IS_ERR(phylink))
                return PTR_ERR(phylink);

        if (priv->hw->xpcs)
                phylink_set_pcs(phylink, &priv->hw->xpcs->pcs);

        priv->phylink = phylink;
        return 0;
}



** Here setup a fixed phy connection
 * phylink_create() - create a phylink instance
 * @config: a pointer to the target &struct phylink_config
 * @fwnode: a pointer to a &struct fwnode_handle describing the network
 *      interface
 * @iface: the desired link mode defined by &typedef phy_interface_t
 * @mac_ops: a pointer to a &struct phylink_mac_ops for the MAC.
 *
 * Create a new phylink instance, and parse the link parameters found in @np.
 * This will parse in-band modes, fixed-link or SFP configuration.
 *
 * Note: the rtnl lock must not be held when calling this function.
 *
 * Returns a pointer to a &struct phylink, or an error-pointer value. Users
 * must use IS_ERR() to check for errors from this function.
 */
struct phylink *phylink_create(struct phylink_config *config,
                               struct fwnode_handle *fwnode,
                               phy_interface_t iface,
                               const struct phylink_mac_ops *mac_ops)
{
        struct phylink *pl;
        int ret;

        pl = kzalloc(sizeof(*pl), GFP_KERNEL);
        if (!pl)
                return ERR_PTR(-ENOMEM);

        mutex_init(&pl->state_mutex);
        INIT_WORK(&pl->resolve, phylink_resolve);

        pl->config = config;
        if (config->type == PHYLINK_NETDEV) {
                pl->netdev = to_net_dev(config->dev);
        } else if (config->type == PHYLINK_DEV) {
                pl->dev = config->dev;
        } else {
                kfree(pl);
                return ERR_PTR(-EINVAL);
        }

        pl->phy_state.interface = iface;
        pl->link_interface = iface;
        if (iface == PHY_INTERFACE_MODE_MOCA)
                pl->link_port = PORT_BNC;
        else
                pl->link_port = PORT_MII;
        pl->link_config.interface = iface;
        pl->link_config.pause = MLO_PAUSE_AN;
        pl->link_config.speed = SPEED_UNKNOWN;
        pl->link_config.duplex = DUPLEX_UNKNOWN;
        pl->link_config.an_enabled = true;
        pl->mac_ops = mac_ops;
        __set_bit(PHYLINK_DISABLE_STOPPED, &pl->phylink_disable_state);
```

```
        timer_setup(&pl->link_poll, phylink_fixed_poll, 0);

        bitmap_fill(pl->supported, __ETHTOOL_LINK_MODE_MASK_NBITS);
        linkmode_copy(pl->link_config.advertising, pl->supported);
        phylink_validate(pl, pl->supported, &pl->link_config);

                /* Checks what type of ohy connection is present, sets pl->cfg_link_an_mode = MLO_AN_FIXED; for
fixed phy */
        ret = phylink_parse_mode(pl, fwnode);
        if (ret < 0) {
                kfree(pl);
                return ERR_PTR(ret);
        }

                /* Configure the phylink struct for fixed-phy */
        if (pl->cfg_link_an_mode == MLO_AN_FIXED) {
                ret = phylink_parse_fixedlink(pl, fwnode);
                if (ret < 0) {
                        kfree(pl);
                        return ERR_PTR(ret);
                }
        }

        pl->cur_link_an_mode = pl->cfg_link_an_mode;

        ret = phylink_register_sfp(pl, fwnode);
        if (ret < 0) {
                kfree(pl);
                return ERR_PTR(ret);
        }

        return pl;
}


/* This will read the property from  device tree and configure the phylink accordingly */
static int phylink_parse_fixedlink(struct phylink *pl,
                                   struct fwnode_handle *fwnode)
{
        struct fwnode_handle *fixed_node;
        const struct phy_setting *s;
        struct gpio_desc *desc;
        u32 speed;
        int ret;

        fixed_node = fwnode_get_named_child_node(fwnode, "fixed-link");
        if (fixed_node) {
                ret = fwnode_property_read_u32(fixed_node, "speed", &speed);

                pl->link_config.speed = speed;
                pl->link_config.duplex = DUPLEX_HALF;

                if (fwnode_property_read_bool(fixed_node, "full-duplex"))
                        pl->link_config.duplex = DUPLEX_FULL;

                /* We treat the "pause" and "asym-pause" terminology as
                 * defining the link partner's ability.
                 */
                if (fwnode_property_read_bool(fixed_node, "pause"))
                        __set_bit(ETHTOOL_LINK_MODE_Pause_BIT,
                                  pl->link_config.lp_advertising);
                if (fwnode_property_read_bool(fixed_node, "asym-pause"))
                        __set_bit(ETHTOOL_LINK_MODE_Asym_Pause_BIT,
                                  pl->link_config.lp_advertising);

                if (ret == 0) {
                        desc = fwnode_gpiod_get_index(fixed_node, "link", 0,
                                                      GPIOD_IN, "?");

                        if (!IS_ERR(desc))
                                pl->link_gpio = desc;
```

```
                        else if (desc == ERR_PTR(-EPROBE_DEFER))
                                ret = -EPROBE_DEFER;
                }
                fwnode_handle_put(fixed_node);

                if (ret)
                        return ret;
        } else {
                u32 prop[5];

                ret = fwnode_property_read_u32_array(fwnode, "fixed-link",
                                                     NULL, 0);
                if (ret != ARRAY_SIZE(prop)) {
                        phylink_err(pl, "broken fixed-link?\n");
                        return -EINVAL;
                }

                ret = fwnode_property_read_u32_array(fwnode, "fixed-link",
                                                     prop, ARRAY_SIZE(prop));
                if (!ret) {
                        pl->link_config.duplex = prop[1] ?
                                                 DUPLEX_FULL : DUPLEX_HALF;
                        pl->link_config.speed = prop[2];
                        if (prop[3])
                                __set_bit(ETHTOOL_LINK_MODE_Pause_BIT,
                                          pl->link_config.lp_advertising);
                        if (prop[4])
                                __set_bit(ETHTOOL_LINK_MODE_Asym_Pause_BIT,
                                          pl->link_config.lp_advertising);
                }
        }

        if (pl->link_config.speed > SPEED_1000 &&
            pl->link_config.duplex != DUPLEX_FULL)
                phylink_warn(pl, "fixed link specifies half duplex for %dMbps link?\n",
                             pl->link_config.speed);

        bitmap_fill(pl->supported, __ETHTOOL_LINK_MODE_MASK_NBITS);
        linkmode_copy(pl->link_config.advertising, pl->supported);
        phylink_validate(pl, pl->supported, &pl->link_config);

        s = phy_lookup_setting(pl->link_config.speed, pl->link_config.duplex,
                               pl->supported, true);
        linkmode_zero(pl->supported);
        phylink_set(pl->supported, MII);
        phylink_set(pl->supported, Pause);
        phylink_set(pl->supported, Asym_Pause);
        phylink_set(pl->supported, Autoneg);
        if (s) {
                __set_bit(s->bit, pl->supported);
                __set_bit(s->bit, pl->link_config.lp_advertising);
        } else {
                phylink_warn(pl, "fixed link %s duplex %dMbps not recognised\n",
                             pl->link_config.duplex == DUPLEX_FULL ? "full" : "half",
                             pl->link_config.speed);
        }

        linkmode_and(pl->link_config.advertising, pl->link_config.advertising,
                     pl->supported);

        pl->link_config.link = 1;
        pl->link_config.an_complete = 1;

        return 0;
}
```

**stmmac_open**

This will connect the ethernet driver to the phylink instance

In stmmac_open we will initialize the phy connection

If an real PHY device was present "phy_attach_direct" will configure the connection and load the appropriate phydriver

In the case of fixed-phy (phylinkcfg_link_an_mode = fixed-phy ) , we just return true to phylink_connect. (The PHY driver is not loaded)

Start the phylink instance (phylink_start(privphylink);

```
static int stmmac_open(struct net_device *dev)
{...
if (priv->hw->pcs != STMMAC_PCS_TBI &&
            priv->hw->pcs != STMMAC_PCS_RTBI &&
            (!priv->hw->xpcs ||
             xpcs_get_an_mode(priv->hw->xpcs, mode) != DW_AN_C73)) {
                pr_alert("Harman Origin");
                                /* This will return 0 as we have configured fixed-phy */
                ret = stmmac_init_phy(dev);  /* This will initialize the PHY connection */
                if (ret) {
                        netdev_err(priv->dev,
                                   "%s: Cannot attach to PHY (error: %d)\n",
                                   __func__, ret);
                        goto init_phy_error;
                }
          ....
          /* Start the phy connection */
          phylink_start(priv->phylink);
         }
....}

/* Check if valid PHY connection is present */
static int stmmac_init_phy(struct net_device *dev)
{
        struct stmmac_priv *priv = netdev_priv(dev);
        struct device_node *node;
        int ret;

        node = priv->plat->phylink_node;

        if (node)
        {       pr_alert("Harman test 1");
                ret = phylink_of_phy_connect(priv->phylink, node, 0);  /* this function will call
phylink_fwnode_phy_connect */
        }

/* If the connection is a fixed-phy, then we return 0 */
int phylink_fwnode_phy_connect(struct phylink *pl,
                               struct fwnode_handle *fwnode,
                               u32 flags)
{
        struct fwnode_handle *phy_fwnode;
        struct phy_device *phy_dev;
        int ret;

        /* Fixed links and 802.3z are handled without needing a PHY */
        if (pl->:q!== MLO_AN_FIXED ||                           /*  MLO_AN_FIXED checks for fixed-phy */
            (pl->cfg_link_an_mode == MLO_AN_INBAND &&
             phy_interface_mode_is_8023z(pl->link_interface)))
        {
        /* Return the phy connect status as true, this will allow for fixed-phy connections */
                return 0;
        }
```

Receive Process

```
************Register the interrupt ***************************

stmmac_open  -->         ret = stmmac_request_irq(dev); --> stmmac_request_irq_multi(struct net_device *dev)

- Device has one common interrupt

  ret = request_irq(dev->irq, stmmac_mac_interrupt, 0, int_name, dev);



- Rx AND Tx interrupts based on DMA given in eth dts( DMA 0, 12, 13, 14, 15). Rx is shown below

     for (i = 0; i < rx_channels_count ; i++) {
 22                 struct stmmac_rx_queue *rx_q = &priv->rx_queue[i];
         /* basically, during probe an array is created with mapping to DMA channels. queue numbers are in
ascending order .*/
 21                 int hw_chan_idx = stmmac_map_queue_to_dma(priv, i);
 20
                /* rx_irq[[ is derived from dts during probe. this is the irq GIC number and +32 = IRQ number*/
 19                 if (priv->rx_irq[hw_chan_idx] == 0)
 18                         continue;
 17
 16                 int_name = priv->int_name_rx_irq[i];
 15                 sprintf(int_name, "%s:%s-%d(%s-%d)", dev->name, "rxdma",
 14                         hw_chan_idx, "rxq", i);
 13                 ret = request_irq(priv->rx_irq[hw_chan_idx],
 12                                 stmmac_rxdma_interrupt, 0, int_name, rx_q);
 11                 if (unlikely(ret < 0)) {
 10                         netdev_err(priv->dev,
  9                                    "%s: ERROR: allocating the RxDMA IRQ %d (error: %d)\n",
  8                                    __func__, priv->rx_irq[hw_chan_idx], ret);
  7                         goto rxdma_x_irq_error;
  6                 }
  5                 rxq_success_cnt++;
  4         }


cat /proc/interrupts to view the created interrupts

107:          0          0          0          0     GICv3 531 Level     eth0:mac
112:        114          0          0          0     GICv3 546 Level     eth0:rxdma-0(rxq-0)
124:          0          0          0          0     GICv3 558 Level     eth0:rxdma-12(rxq-1)
125:          0          0          0          0     GICv3 559 Level     eth0:rxdma-13(rxq-2)
126:          0          0          0          0     GICv3 560 Level     eth0:rxdma-14(rxq-3)
127:          0          0          0          0     GICv3 561 Level     eth0:rxdma-15(rxq-4)
128:          0          0          0          0     GICv3 564 Level     eth0:txdma-0(txq-0)
140:          0          0          0          0     GICv3 576 Level     eth0:txdma-12(txq-1)
141:          0          0          0          0     GICv3 577 Level     eth0:txdma-13(txq-2)
142:          0          0          0          0     GICv3 578 Level     eth0:txdma-14(txq-3)
143:          2          0          0          0     GICv3 579 Level     eth0:txdma-15(txq-4)


************ Schedule the Napi***************************

The Rx interrupt handler is " static irqreturn_t stmmac_rxdma_interrupt(int irq, void *data) "

The data will be the pointer to the struct stmmac_rx_queue, which has the private data.

  5 static irqreturn_t stmmac_rxdma_interrupt(int irq, void *data)
  4 {
  3         struct stmmac_rx_queue *rx_q = (struct stmmac_rx_queue *)data;
  2         struct stmmac_priv *priv = rx_q->priv_data;
  1
  0         stmmac_napi_check(priv, rx_q->queue_index, DMA_DIR_RX);
  1
```

```
 2         return IRQ_HANDLED;
 3 }
```

This fucntion  needs to confirm the interrupt and scheduel NAPI if needed

```
  static int stmmac_napi_check(struct stmmac_priv *priv, u32 chan, u32 dir)
 41 {
 40         int status;
 39         struct stmmac_rx_queue *rx_q = &priv->rx_queue[chan];
 38         struct stmmac_tx_queue *tx_q = &priv->tx_queue[chan];
 37         struct stmmac_channel *ch = &priv->channel[chan];
 36         struct napi_struct *rx_napi;
 35         struct napi_struct *tx_napi;
 34         unsigned long flags;
         /* Get the dma channel from queue number */
 33         int hw_chan_idx = stmmac_map_queue_to_dma(priv, chan);
 32
 31         rx_napi = rx_q->xsk_pool ? &ch->rxtx_napi : &ch->rx_napi;
 30         tx_napi = tx_q->xsk_pool ? &ch->rxtx_napi : &ch->tx_napi;
 29
 28         status = stmmac_dma_interrupt_status(priv, priv->ioaddr, &priv->xstats,
 27                                              hw_chan_idx, dir);
 26
            /* Disable dma irq and Enable NAPI */
 25         if ((status & handle_rx) && (chan < priv->plat->rx_queues_to_use)) {
 24                 if (napi_schedule_prep(rx_napi)) {
 23                         spin_lock_irqsave(&ch->lock, flags);
 22                         stmmac_disable_dma_irq(priv, priv->ioaddr, hw_chan_idx,
 21                                                1, 0);
 20                         spin_unlock_irqrestore(&ch->lock, flags);
 19                         __napi_schedule(rx_napi);
 18                 }
 17         }
 16
 15         if ((status & handle_tx) && (chan < priv->plat->tx_queues_to_use)) {
 14                 if (napi_schedule_prep(tx_napi)) {
 13                         spin_lock_irqsave(&ch->lock, flags);
 12                         stmmac_disable_dma_irq(priv, priv->ioaddr, hw_chan_idx,
 11                                                0, 1);
 10                         spin_unlock_irqrestore(&ch->lock, flags);
  9                         __napi_schedule(tx_napi);
  8                 }
  7         }
  6
  5         return status;
  4 }
```