MicroChip:

Howto PHY driver

For release 2023.06
September 27, 2023

1 PHY drivers

A PHY driver is not paired with a given switch port directly. Instead the switch application reads IEEE802.3 clause 22 register 2 and 3 over the MDIO bus, if the port configuration has stated that a PHY is present on the switch port in question.

This 2*16bit number is matched against the set of PHY drivers until one that match is found.

The driver that is matched, is here after used to provide when accessing the PHY on the port in question.

1.1 Init process.

This section is an overview of how the PHYs and their drivers are mapped.

The PHYs are initialized by calling `meba_reset( ,MEBA_PHY_INITIALIZE)`, which is a function provided in `meba.c`. The code in `meba.c` looks like this

```
static mesa_rc xxx_reset(meba_inst_t inst,
                         meba_reset_point_t reset)
{
  ...
  switch (reset) {
     ...
     case MEBA_PHY_INITIALIZE:
       inst->phy_devices = (mepa_device_t **)&board->phy_devices;
       inst->phy_device_cnt = board->port_cnt;
       meba_phy_driver_init(inst);
       break;
     ...
  }
  ...
}
```

The `meba_phy_driver_init()` is in `meba/src/meba_generic.c` and looks like this:

```
void meba_phy_driver_init(meba_inst_t inst)
{
  ...
  for (port_no = 0; port_no < inst->phy_device_cnt; port_no++) {
    ...
    inst->api.meba_port_entry_get(inst, port_no, &entry);
    meba_port_cap_t port_cap = entry.cap;

    if ( port_cap & (MEBA_PORT_CAP_COPPER     |
                     MEBA_PORT_CAP_DUAL_COPPER|
                     MEBA_PORT_CAP_VTSS_10G_PHY) ) {
      inst->phy_device_ctx[port_no].port_no=port_no;
      ...
      inst->phy_devices[port_no] = mepa_create(&inst->mepa_callout,
                                               &inst->phy_device_ctx[port_no],
```

```
                                                    &board_conf);
    }
  }
}
```

The point here is, that the MEBA capability flag are retrieved, and if they indicate that this is a PHY port, then structure `phy_device_ctx` is set up, and the PHY driver is mapped with `mepa_create()` which is in `mepa/common/src/phy.c`:

```
struct mepa_device* mepa_create(const mepa_callout_t    MEPA_SHARED_PTR *callout,
                                struct mepa_callout_ctx MEPA_SHARED_PTR *callout_ctx,
                                struct mepa_board_conf  *conf)
{
  ...
  phy_id = mepa_phy_id_get(callout, callout_ctx);

  for (int i = 0; i < PHY_FAMILIES; i++) {
    ...
    for (uint32_t j = 0; j < MEPA_phy_lib[i].count; j++) {
      mepa_driver_t *driver = &MEPA_phy_lib[i].phy_drv[j];

      if ((driver->id & driver->mask) == (phy_id & driver->mask)) {
        dev = driver->mepa_driver_probe(driver, callout, callout_ctx, conf);
        if (dev) return dev;
      }
    }
  }
  return NULL; // I.e. No driver found for this PHY
}
```

Each PHY driver is hooked up into the `MEPA_phy_lib` array, and `mepa_create()` will run through this list and find a match. If a match is not found, then `NULL` is returned, and that is an error, since a driver must exist for the PHY in question.

If the `phy_id` is acceptable, then the `mepa_driver_probe()` is called. This function may return `NULL` if it decide that it can not support the PHY. That means the search will continue. If it does not return `NULL` then this will be the driver for the PHY and the search will stop.

1.2 Providing PHY drivers

The organization of the PHY driver code is under `vtss_api/mepa/`. Under this folder you can se folders called `aqr`, `intel`, `ksz9031`, `microchip` and `vtss`. The orgainzation of the drivers under these folders vary, but when looking into a specific source file, then pattern is the same.

A good example on how a driver is implemented is `microchip/lan8814/src/lan8814.c`.

For release 2023.06 there is a patch[1] that can be applied, which will add the folder `mepa/example/` where the code for registering the driver is made, but none of the other

[1] `https://github.com/microchip-ung/phy-driver-example`

driver functions are. Then you can go and implement the subset that is necessary. It is suggested to run this patch since it will edit a number of files around the system in order for the drive to be hooked up.

In `vtss_api/meba/src/meba_generic.c` the function `meba_phy_driver_init()` will hook up the init functions to the PHY drivers. These init functions are typically called something like `mepa_xxx_driver_init()`, and they are defined in `vtss_api/mepa/src`.

```
mepa_drivers_t mepa_xxx_driver_init()
{
   static const int nr_xxx_phy = 1;
   static mepa_driver_t xxx_drivers[] = {
     { // 1st driver
       .id = 0x8200,
       .mask = 0x0000FF00,
       .mepa_driver_delete = phy_xxx_delete,
       // ... and a number of functions more.
     }
   };

   mepa_drivers_t result;
   result.phy_drv = xxx_drivers;
   result.count = nr_xxx_phy;

   return result;
}
```

The array `xxx_drivers` above can have any number of elements, which must be reflected by `nr_xxx_phy`. The `id` and `mask` are used to figure out, if this driver can be used with a specific PHY. If e.g. the PHY return `0x12348256` in register 2 and 3 as mentioned then `mask` will say, that the `id` and the `0x12348256` from the PHY shall match in the bit positions as given by the `mask`, which it will in this example. But `0x12347156` would not.

If a match is found, then the functions provided will be used in conjunction which this PHY.

1.3 Driver functions

All the functions provided can be seen in `vtss_api/mepa/include/microchip/ethernet/phy/api/phy.h`.

In `mepa/common/src/phy.c` it can be seen that the PHY driver functions are called via constructions like this:

```
mepa_rc mepa_poll(struct mepa_device *dev,
                  mepa_status_t *status)
{
    if (!dev || !dev->drv->mepa_driver_poll) {
        return MESA_RC_NOT_IMPLEMENTED;
    }

    return dev->drv->mepa_driver_poll(dev, status);
```

```
}
```

This means that calling a not implemented function of a PHY driver is okay. You will just get `MESA_RC_NOT_IMPLEMENTED` in the return code.