PHY Abstraction Layer

Purpose

Most network devices consist of set of registers which provide an interface to a MAC layer, which communicates with the physical connection through a PHY. The PHY concerns itself with negotiating link parameters with the link partner on the other side of the network connection (typically, an ethernet cable), and provides a register interface to allow drivers to determine what settings were chosen, and to configure what settings are allowed.

While these devices are distinct from the network devices, and conform to a standard layout for the registers, it has been common practice to integrate the PHY management code with the network driver. This has resulted in large amounts of redundant code. Also, on embedded systems with multiple (and sometimes quite different) ethernet controllers connected to the same management bus, it is difficult to ensure safe use of the bus.

Since the PHYs are devices, and the management busses through which they are accessed are, in fact, busses, the PHY Abstraction Layer treats them as such. In doing so, it has these goals:

1. Increase code-reuse
2. Increase overall code-maintainability
3. Speed development time for new network drivers, and for new systems

Basically, this layer is meant to provide an interface to PHY devices which allows network driver writers to write as little code as possible, while still providing a full feature set.

The MDIO bus

Most network devices are connected to a PHY by means of a management bus. Different devices use different busses (though some share common interfaces). In order to take advantage of the PAL, each bus interface needs to be registered as a distinct device.

1. read and write functions must be implemented. Their prototypes are:

   ```
   int write(struct mii_bus *bus, int mii_id, int regnum, u16 value);
   int read(struct mii_bus *bus, int mii_id, int regnum);
   ```

   mii_id is the address on the bus for the PHY, and regnum is the register number. These functions are guaranteed not to be called from interrupt time, so it is safe for them to block, waiting for an interrupt to signal the operation is complete

2. A reset function is optional. This is used to return the bus to an initialized state.
3. A probe function is needed. This function should set up anything the bus driver needs, setup the mii_bus structure, and register with the PAL using mdiobus_register. Similarly, there's a remove function to undo all of that (use mdiobus_unregister).
4. Like any driver, the device_driver structure must be configured, and init exit functions are used to register the driver.
5. The bus must also be declared somewhere as a device, and registered.

As an example for how one driver implemented an mdio bus driver, see drivers/net/ethernet/freescale/fsl_pq_mdio.c and an associated DTS file for one of the users. (e.g. "git grep fsl,.*-mdio arch/powerpc/boot/dts/")

(RG)MII/electrical interface considerations

The Reduced Gigabit Medium Independent Interface (RGMII) is a 12-pin electrical signal interface using a synchronous 125Mhz clock signal and several data lines. Due to this design decision, a 1.5ns to 2ns delay must be added between the clock line (RXC or TXC) and the data lines to let the PHY (clock sink) have a large enough setup and hold time to sample the data lines correctly. The PHY library offers different types of PHY_INTERFACE_MODE_RGMII* values to let the PHY driver and optionally the MAC driver, implement the required delay. The values of phy_interface_t must be understood from the perspective of the PHY device itself, leading to the following:

- PHY_INTERFACE_MODE_RGMII: the PHY is not responsible for inserting any internal delay by itself, it assumes that either the Ethernet MAC (if capable) or the PCB traces insert the correct 1.5-2ns delay
- PHY_INTERFACE_MODE_RGMII_TXID: the PHY should insert an internal delay for the transmit data lines (TXD[3:0]) processed by the PHY device
- PHY_INTERFACE_MODE_RGMII_RXID: the PHY should insert an internal delay for the receive data lines (RXD[3:0]) processed by the PHY device
- PHY_INTERFACE_MODE_RGMII_ID: the PHY should insert internal delays for both transmit AND receive data lines from/to the PHY device

Whenever possible, use the PHY side RGMII delay for these reasons:

- PHY devices may offer sub-nanosecond granularity in how they allow a receiver/transmitter side delay (e.g: 0.5, 1.0, 1.5ns) to

be specified. Such precision may be required to account for differences in PCB trace lengths
- PHY devices are typically qualified for a large range of applications (industrial, medical, automotive...), and they provide a constant and reliable delay across temperature/pressure/voltage ranges
- PHY device drivers in PHYLIB being reusable by nature, being able to configure correctly a specified delay enables more designs with similar delay requirements to be operate correctly

For cases where the PHY is not capable of providing this delay, but the Ethernet MAC driver is capable of doing so, the correct phy_interface_t value should be PHY_INTERFACE_MODE_RGMII, and the Ethernet MAC driver should be configured correctly in order to provide the required transmit and/or receive side delay from the perspective of the PHY device. Conversely, if the Ethernet MAC driver looks at the phy_interface_t value, for any other mode but PHY_INTERFACE_MODE_RGMII, it should make sure that the MAC-level delays are disabled.

In case neither the Ethernet MAC, nor the PHY are capable of providing the required delays, as defined per the RGMII standard, several options may be available:

- Some SoCs may offer a pin pad/mux/controller capable of configuring a given set of pins'strength, delays, and voltage; and it may be a suitable option to insert the expected 2ns RGMII delay.
- Modifying the PCB design to include a fixed delay (e.g: using a specifically designed serpentine), which may not require software configuration at all.

Common problems with RGMII delay mismatch

When there is a RGMII delay mismatch between the Ethernet MAC and the PHY, this will most likely result in the clock and data line signals to be unstable when the PHY or MAC take a snapshot of these signals to translate them into logical 1 or 0 states and reconstruct the data being transmitted/received. Typical symptoms include:

- Transmission/reception partially works, and there is frequent or occasional packet loss observed
- Ethernet MAC may report some or all packets ingressing with a FCS/CRC error, or just discard them all
- Switching to lower speeds such as 10/100Mbits/sec makes the problem go away (since there is enough setup/hold time in that case)

Connecting to a PHY

Sometime during startup, the network driver needs to establish a connection between the PHY device, and the network device. At this time, the PHY's bus and drivers need to all have been loaded, so it is ready for the connection. At this point, there are several ways to connect to the PHY:

1. The PAL handles everything, and only calls the network driver when the link state changes, so it can react.
2. The PAL handles everything except interrupts (usually because the controller has the interrupt registers).
3. The PAL handles everything, but checks in with the driver every second, allowing the network driver to react first to any changes before the PAL does.
4. The PAL serves only as a library of functions, with the network device manually calling functions to update status, and configure the PHY

Letting the PHY Abstraction Layer do Everything

If you choose option 1 (The hope is that every driver can, but to still be useful to drivers that can't), connecting to the PHY is simple:

First, you need a function to react to changes in the link state. This function follows this protocol:

```
static void adjust_link(struct net_device *dev);
```

Next, you need to know the device name of the PHY connected to this device. The name will look something like, "0:00", where the first number is the bus id, and the second is the PHY's address on that bus. Typically, the bus is responsible for making its ID unique.

Now, to connect, just call this function:

```
phydev = phy_connect(dev, phy_name, &adjust_link, interface);
```

phydev is a pointer to the phy_device structure which represents the PHY. If phy_connect is successful, it will return the pointer. dev, here, is the pointer to your net_device. Once done, this function will have started the PHY's software state machine, and registered for the PHY's interrupt, if it has one. The phydev structure will be populated with information about the current state, though the PHY will not yet be truly operational at this point.

PHY-specific flags should be set in phydev->dev_flags prior to the call to phy_connect() such that the underlying PHY driver can check for flags and perform specific operations based on them. This is useful if the system has put hardware restrictions on the PHY/controller, of which the PHY needs to be aware.

interface is a u32 which specifies the connection type used between the controller and the PHY. Examples are GMII, MII, RGMII, and SGMII. See "PHY interface mode" below. For a full list, see include/linux/phy.h

Now just make sure that phydev->supported and phydev->advertising have any values pruned from them which don't make sense for your controller (a 10/100 controller may be connected to a gigabit capable PHY, so you would need to mask off

SUPPORTED_1000baseT*). See include/linux/ethtool.h for definitions for these bitfields. Note that you should not SET any bits, except the SUPPORTED_Pause and SUPPORTED_AsymPause bits (see below), or the PHY may get put into an unsupported state.

Lastly, once the controller is ready to handle network traffic, you call phy_start(phydev). This tells the PAL that you are ready, and configures the PHY to connect to the network. If the MAC interrupt of your network driver also handles PHY status changes, just set phydev->irq to PHY_MAC_INTERRUPT before you call phy_start and use phy_mac_interrupt() from the network driver. If you don't want to use interrupts, set phydev->irq to PHY_POLL. phy_start() enables the PHY interrupts (if applicable) and starts the phylib state machine.

When you want to disconnect from the network (even if just briefly), you call phy_stop(phydev). This function also stops the phylib state machine and disables PHY interrupts.

PHY interface modes

The PHY interface mode supplied in the phy_connect() family of functions defines the initial operating mode of the PHY interface. This is not guaranteed to remain constant; there are PHYs which dynamically change their interface mode without software interaction depending on the negotiation results.

Some of the interface modes are described below:

`PHY_INTERFACE_MODE_SMII`

> This is serial MII, clocked at 125MHz, supporting 100M and 10M speeds. Some details can be found in https://opencores.org/ocsvn/smii/smii/trunk/doc/SMII.pdf

`PHY_INTERFACE_MODE_1000BASEX`

> This defines the 1000BASE-X single-lane serdes link as defined by the 802.3 standard section 36. The link operates at a fixed bit rate of 1.25Gbaud using a 10B/8B encoding scheme, resulting in an underlying data rate of 1Gbps. Embedded in the data stream is a 16-bit control word which is used to negotiate the duplex and pause modes with the remote end. This does not include "up-clocked" variants such as 2.5Gbps speeds (see below.)

`PHY_INTERFACE_MODE_2500BASEX`

> This defines a variant of 1000BASE-X which is clocked 2.5 times as fast as the 802.3 standard, giving a fixed bit rate of 3.125Gbaud.

`PHY_INTERFACE_MODE_SGMII`

> This is used for Cisco SGMII, which is a modification of 1000BASE-X as defined by the 802.3 standard. The SGMII link consists of a single serdes lane running at a fixed bit rate of 1.25Gbaud with 10B/8B encoding. The underlying data rate is 1Gbps, with the slower speeds of 100Mbps and 10Mbps being achieved through replication of each data symbol. The 802.3 control word is re-purposed to send the negotiated speed and duplex information from to the MAC, and for the MAC to acknowledge receipt. This does not include "up-clocked" variants such as 2.5Gbps speeds.
>
> Note: mismatched SGMII vs 1000BASE-X configuration on a link can successfully pass data in some circumstances, but the 16-bit control word will not be correctly interpreted, which may cause mismatches in duplex, pause or other settings. This is dependent on the MAC and/or PHY behaviour.

`PHY_INTERFACE_MODE_5GBASER`

> This is the IEEE 802.3 Clause 129 defined 5GBASE-R protocol. It is identical to the 10GBASE-R protocol defined in Clause 49, with the exception that it operates at half the frequency. Please refer to the IEEE standard for the definition.

`PHY_INTERFACE_MODE_10GBASER`

> This is the IEEE 802.3 Clause 49 defined 10GBASE-R protocol used with various different mediums. Please refer to the IEEE standard for a definition of this.
>
> Note: 10GBASE-R is just one protocol that can be used with XFI and SFI. XFI and SFI permit multiple protocols over a single SERDES lane, and also defines the electrical characteristics of the signals with a host compliance board plugged into the host XFP/SFP connector. Therefore, XFI and SFI are not PHY interface types in their own right.

`PHY_INTERFACE_MODE_10GKR`

> This is the IEEE 802.3 Clause 49 defined 10GBASE-R with Clause 73 autonegotiation. Please refer to the IEEE standard for further information.
>
> Note: due to legacy usage, some 10GBASE-R usage incorrectly makes use of this definition.

`PHY_INTERFACE_MODE_25GBASER`

> This is the IEEE 802.3 PCS Clause 107 defined 25GBASE-R protocol. The PCS is identical to 10GBASE-R, i.e. 64B/66B encoded running 2.5 as fast, giving a fixed bit rate of 25.78125 Gbaud. Please refer to the IEEE standard for further information.

`PHY_INTERFACE_MODE_100BASEX`

This defines IEEE 802.3 Clause 24. The link operates at a fixed data rate of 125Mpbs using a 4B/5B encoding scheme, resulting in an underlying data rate of 100Mpbs.

Pause frames / flow control

The PHY does not participate directly in flow control/pause frames except by making sure that the SUPPORTED_Pause and SUPPORTED_AsymPause bits are set in MII_ADVERTISE to indicate towards the link partner that the Ethernet MAC controller supports such a thing. Since flow control/pause frames generation involves the Ethernet MAC driver, it is recommended that this driver takes care of properly indicating advertisement and support for such features by setting the SUPPORTED_Pause and SUPPORTED_AsymPause bits accordingly. This can be done either before or after phy_connect() and/or as a result of implementing the ethtool::set_pauseparam feature.

Keeping Close Tabs on the PAL

It is possible that the PAL's built-in state machine needs a little help to keep your network device and the PHY properly in sync. If so, you can register a helper function when connecting to the PHY, which will be called every second before the state machine reacts to any changes. To do this, you need to manually call phy_attach() and phy_prepare_link(), and then call phy_start_machine() with the second argument set to point to your special handler.

Currently there are no examples of how to use this functionality, and testing on it has been limited because the author does not have any drivers which use it (they all use option 1). So Caveat Emptor.

Doing it all yourself

There's a remote chance that the PAL's built-in state machine cannot track the complex interactions between the PHY and your network device. If this is so, you can simply call phy_attach(), and not call phy_start_machine or phy_prepare_link(). This will mean that phydev->state is entirely yours to handle (phy_start and phy_stop toggle between some of the states, so you might need to avoid them).

An effort has been made to make sure that useful functionality can be accessed without the state-machine running, and most of these functions are descended from functions which did not interact with a complex state-machine. However, again, no effort has been made so far to test running without the state machine, so tryer beware.

Here is a brief rundown of the functions:

```
int phy_read(struct phy_device *phydev, u16 regnum);
int phy_write(struct phy_device *phydev, u16 regnum, u16 val);
```

Simple read/write primitives. They invoke the bus's read/write function pointers.

```
void phy_print_status(struct phy_device *phydev);
```

A convenience function to print out the PHY status neatly.

```
void phy_request_interrupt(struct phy_device *phydev);
```

Requests the IRQ for the PHY interrupts.

```
struct phy_device * phy_attach(struct net_device *dev, const char *phy_id,
                               phy_interface_t interface);
```

Attaches a network device to a particular PHY, binding the PHY to a generic driver if none was found during bus initialization.

```
int phy_start_aneg(struct phy_device *phydev);
```

Using variables inside the phydev structure, either configures advertising and resets autonegotiation, or disables autonegotiation, and configures forced settings.

```
static inline int phy_read_status(struct phy_device *phydev);
```

Fills the phydev structure with up-to-date information about the current settings in the PHY.

```
int phy_ethtool_ksettings_set(struct phy_device *phydev,
                              const struct ethtool_link_ksettings *cmd);
```

Ethtool convenience functions.

```
int phy_mii_ioctl(struct phy_device *phydev,
                  struct mii_ioctl_data *mii_data, int cmd);
```

The MII ioctl. Note that this function will completely screw up the state machine if you write registers like BMCR, BMSR, ADVERTISE, etc. Best to use this only to write registers which are not standard, and don't set off a renegotiation.

PHY Device Drivers

With the PHY Abstraction Layer, adding support for new PHYs is quite easy. In some cases, no work is required at all! However, many PHYs require a little hand-holding to get up-and-running.

Generic PHY driver

If the desired PHY doesn't have any errata, quirks, or special features you want to support, then it may be best to not add support, and let the PHY Abstraction Layer's Generic PHY Driver do all of the work.

Writing a PHY driver

If you do need to write a PHY driver, the first thing to do is make sure it can be matched with an appropriate PHY device. This is done during bus initialization by reading the device's UID (stored in registers 2 and 3), then comparing it to each driver's phy_id field by ANDing it with each driver's phy_id_mask field. Also, it needs a name. Here's an example:

```
static struct phy_driver dm9161_driver = {
      .phy_id         = 0x0181b880,
      .name           = "Davicom DM9161E",
      .phy_id_mask    = 0x0ffffff0,
      ...
}
```

Next, you need to specify what features (speed, duplex, autoneg, etc) your PHY device and driver support. Most PHYs support PHY_BASIC_FEATURES, but you can look in include/mii.h for other features.

Each driver consists of a number of function pointers, documented in include/linux/phy.h under the phy_driver structure.

Of these, only config_aneg and read_status are required to be assigned by the driver code. The rest are optional. Also, it is preferred to use the generic phy driver's versions of these two functions if at all possible: genphy_read_status and genphy_config_aneg. If this is not possible, it is likely that you only need to perform some actions before and after invoking these functions, and so your functions will wrap the generic ones.

Feel free to look at the Marvell, Cicada, and Davicom drivers in drivers/net/phy/ for examples (the lxt and qsemi drivers have not been tested as of this writing).

The PHY's MMD register accesses are handled by the PAL framework by default, but can be overridden by a specific PHY driver if required. This could be the case if a PHY was released for manufacturing before the MMD PHY register definitions were standardized by the IEEE. Most modern PHYs will be able to use the generic PAL framework for accessing the PHY's MMD registers. An example of such usage is for Energy Efficient Ethernet support, implemented in the PAL. This support uses the PAL to access MMD registers for EEE query and configuration if the PHY supports the IEEE standard access mechanisms, or can use the PHY's specific access interfaces if overridden by the specific PHY driver. See the Micrel driver in drivers/net/phy/ for an example of how this can be implemented.

Board Fixups

Sometimes the specific interaction between the platform and the PHY requires special handling. For instance, to change where the PHY's clock input is, or to add a delay to account for latency issues in the data path. In order to support such contingencies, the PHY Layer allows platform code to register fixups to be run when the PHY is brought up (or subsequently reset).

When the PHY Layer brings up a PHY it checks to see if there are any fixups registered for it, matching based on UID (contained in the PHY device's phy_id field) and the bus identifier (contained in phydev->dev.bus_id). Both must match, however two constants, PHY_ANY_ID and PHY_ANY_UID, are provided as wildcards for the bus ID and UID, respectively.

When a match is found, the PHY layer will invoke the run function associated with the fixup. This function is passed a pointer to the phy_device of interest. It should therefore only operate on that PHY.

The platform code can either register the fixup using phy_register_fixup():

```
int phy_register_fixup(const char *phy_id,
        u32 phy_uid, u32 phy_uid_mask,
        int (*run)(struct phy_device *));
```

Or using one of the two stubs, phy_register_fixup_for_uid() and phy_register_fixup_for_id():

```
int phy_register_fixup_for_uid(u32 phy_uid, u32 phy_uid_mask,
               int (*run)(struct phy_device *));
int phy_register_fixup_for_id(const char *phy_id,
               int (*run)(struct phy_device *));
```

The stubs set one of the two matching criteria, and set the other one to match anything.

When phy_register_fixup() or *_for_uid()/*_for_id() is called at module load time, the module needs to unregister the fixup and free allocated memory when it's unloaded.

Call one of following function before unloading module:

```
int phy_unregister_fixup(const char *phy_id, u32 phy_uid, u32 phy_uid_mask);
int phy_unregister_fixup_for_uid(u32 phy_uid, u32 phy_uid_mask);
```

```
int phy_register_fixup_for_id(const char *phy_id);
```

Standards

IEEE Standard 802.3: CSMA/CD Access Method and Physical Layer Specifications, Section Two: http://standards.ieee.org/getieee802/download/802.3-2008_section2.pdf

RGMII v1.3: http://web.archive.org/web/20160303212629/http://www.hp.com/rnd/pdfs/RGMIIv1_3.pdf

RGMII v2.0: http://web.archive.org/web/20160303171328/http://www.hp.com/rnd/pdfs/RGMIIv2_0_final_hp.pdf