
PHY Abstraction Layer
(Updated 2008-04-08)

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 necessary. 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/gianfar_mii.c and arch/ppc/syslib/mpc85xx_devices.c

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, flags, 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.

flags is a u32 which can optionally contain phy-specific flags.
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. 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, 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 you want to handle your own interrupts,
just set phydev->irq to PHY_IGNORE_INTERRUPT before you call phy_start.
Similarly, if you don't want to use interrupts, set phydev->irq to PHY_POLL.

When you want to disconnect from the network (even if just briefly), you call
phy_stop(phydev).

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.

```
int phy_clear_interrupt(struct phy_device *phydev);
int phy_config_interrupt(struct phy_device *phydev, u32 interrupts);
```

Clear the PHY's interrupt, and configure which ones are allowed,
respectively. Currently only supports all on, or all off.

```
int phy_enable_interrupts(struct phy_device *phydev);
int phy_disable_interrupts(struct phy_device *phydev);
```

Functions which enable/disable PHY interrupts, clearing them
before and after, respectively.

```
int phy_start_interrupts(struct phy_device *phydev);
int phy_stop_interrupts(struct phy_device *phydev);
```

Requests the IRQ for the PHY interrupts, then enables them for
start, or disables then frees them for stop.

```
struct phy_device * phy_attach(struct net_device *dev, const char *phy_id,
                u32 flags, 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. Passes in
any phy-specific flags as needed.

```
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.

```
void phy_sanitize_settings(struct phy_device *phydev)
```

Resolves differences between currently desired settings, and
supported settings for the given PHY device. Does not make
the changes in the hardware, though.

```
int phy_ethtool_sset(struct phy_device *phydev, struct ethtool_cmd *cmd);
int phy_ethtool_gset(struct phy_device *phydev, struct ethtool_cmd *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:

   config_init: configures PHY into a sane state after a reset.
     For instance, a Davicom PHY requires descrambling disabled.
   probe: Does any setup needed by the driver
   suspend/resume: power management
   config_aneg: Changes the speed/duplex/negotiation settings
   read_status: Reads the current speed/duplex/negotiation settings
   ack_interrupt: Clear a pending interrupt
   config_intr: Enable or disable interrupts
   remove: Does any driver take-down

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)

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.