PHY subsystem

Author: Kishon Vijay Abraham I <kishon@ti.com>

This document explains the Generic PHY Framework along with the APIs provided, and how-to-use.

Introduction

PHY is the abbreviation for physical layer. It is used to connect a device to the physical medium e.g., the USB controller has a PHY to provide functions such as serialization, de-serialization, encoding, decoding and is responsible for obtaining the required data transmission rate. Note that some USB controllers have PHY functionality embedded into it and others use an external PHY. Other peripherals that use PHY include Wireless LAN, Ethernet, SATA etc.

The intention of creating this framework is to bring the PHY drivers spread all over the Linux kernel to drivers/phy to increase code re-use and for better code maintainability.

This framework will be of use only to devices that use external PHY (PHY functionality is not embedded within the controller).

Registering/Unregistering the PHY provider

PHY provider refers to an entity that implements one or more PHY instances. For the simple case where the PHY provider implements only a single instance of the PHY, the framework provides its own implementation of of_xlate in of_phy_simple_xlate. If the PHY provider implements multiple instances, it should provide its own implementation of of_xlate. of_xlate is used only for dt boot case.

```
#define of_phy_provider_register(dev, xlate)    \
        __of_phy_provider_register((dev), NULL, THIS_MODULE, (xlate))

#define devm_of_phy_provider_register(dev, xlate)       \
        __devm_of_phy_provider_register((dev), NULL, THIS_MODULE,
                                        (xlate))
```

of_phy_provider_register and devm_of_phy_provider_register macros can be used to register the phy_provider and it takes device and of_xlate as arguments. For the dt boot case, all PHY providers should use one of the above 2 macros to register the PHY provider.

Often the device tree nodes associated with a PHY provider will contain a set of children that each represent a single PHY. Some bindings may nest the child nodes within extra levels for context and extensibility, in which case the low level of_phy_provider_register_full() and devm_of_phy_provider_register_full() macros can be used to override the node containing the children.

```
#define of_phy_provider_register_full(dev, children, xlate) \
        __of_phy_provider_register(dev, children, THIS_MODULE, xlate)

#define devm_of_phy_provider_register_full(dev, children, xlate) \
        __devm_of_phy_provider_register_full(dev, children,
                                             THIS_MODULE, xlate)

void devm_of_phy_provider_unregister(struct device *dev,
        struct phy_provider *phy_provider);
void of_phy_provider_unregister(struct phy_provider *phy_provider);
```

devm_of_phy_provider_unregister and of_phy_provider_unregister can be used to unregister the PHY.

Creating the PHY

The PHY driver should create the PHY in order for other peripheral controllers to make use of it. The PHY framework provides 2 APIs to create the PHY.

```
struct phy *phy_create(struct device *dev, struct device_node *node,
                       const struct phy_ops *ops);
struct phy *devm_phy_create(struct device *dev,
                            struct device_node *node,
                            const struct phy_ops *ops);
```

The PHY drivers can use one of the above 2 APIs to create the PHY by passing the device pointer and phy ops. phy_ops is a set of function pointers for performing PHY operations such as init, exit, power_on and power_off.

Inorder to dereference the private data (in phy_ops), the phy provider driver can use phy_set_drvdata() after creating the PHY and use phy_get_drvdata() in phy_ops to get back the private data.

4. Getting a reference to the PHY

Before the controller can make use of the PHY, it has to get a reference to it. This framework provides the following APIs to get a reference to the PHY.

```
struct phy *phy_get(struct device *dev, const char *string);
struct phy *phy_optional_get(struct device *dev, const char *string);
struct phy *devm_phy_get(struct device *dev, const char *string);
struct phy *devm_phy_optional_get(struct device *dev,
                                  const char *string);
struct phy *devm_of_phy_get_by_index(struct device *dev,
                                     struct device_node *np,
                                     int index);
```

phy_get, phy_optional_get, devm_phy_get and devm_phy_optional_get can be used to get the PHY. In the case of dt boot, the string arguments should contain the phy name as given in the dt data and in the case of non-dt boot, it should contain the label of the PHY. The two devm_phy_get associates the device with the PHY using devres on successful PHY get. On driver detach, release function is invoked on the devres data and devres data is freed. phy_optional_get and devm_phy_optional_get should be used when the phy is optional. These two functions will never return -ENODEV, but instead returns NULL when the phy cannot be found.Some generic drivers, such as ehci, may use multiple phys and for such drivers referencing phy(s) by name(s) does not make sense. In this case, devm_of_phy_get_by_index can be used to get a phy reference based on the index.

It should be noted that NULL is a valid phy reference. All phy consumer calls on the NULL phy become NOPs. That is the release calls, the phy_init() and phy_exit() calls, and phy_power_on() and phy_power_off() calls are all NOP when applied to a NULL phy. The NULL phy is useful in devices for handling optional phy devices.

Releasing a reference to the PHY

When the controller no longer needs the PHY, it has to release the reference to the PHY it has obtained using the APIs mentioned in the above section. The PHY framework provides 2 APIs to release a reference to the PHY.

```
void phy_put(struct phy *phy);
void devm_phy_put(struct device *dev, struct phy *phy);
```

Both these APIs are used to release a reference to the PHY and devm_phy_put destroys the devres associated with this PHY.

Destroying the PHY

When the driver that created the PHY is unloaded, it should destroy the PHY it created using one of the following 2 APIs:

```
void phy_destroy(struct phy *phy);
void devm_phy_destroy(struct device *dev, struct phy *phy);
```

Both these APIs destroy the PHY and devm_phy_destroy destroys the devres associated with this PHY.

PM Runtime

This subsystem is pm runtime enabled. So while creating the PHY, pm_runtime_enable of the phy device created by this subsystem is called and while destroying the PHY, pm_runtime_disable is called. Note that the phy device created by this subsystem will be a child of the device that calls phy_create (PHY provider device).

So pm_runtime_get_sync of the phy_device created by this subsystem will invoke pm_runtime_get_sync of PHY provider device because of parent-child relationship. It should also be noted that phy_power_on and phy_power_off performs phy_pm_runtime_get_sync and phy_pm_runtime_put respectively. There are exported APIs like phy_pm_runtime_get, phy_pm_runtime_get_sync, phy_pm_runtime_put, phy_pm_runtime_put_sync, phy_pm_runtime_allow and phy_pm_runtime_forbid for performing PM operations.

PHY Mappings

In order to get reference to a PHY without help from DeviceTree, the framework offers lookups which can be compared to clkdev that allow clk structures to be bound to devices. A lookup can be made during runtime when a handle to the struct phy already exists.

The framework offers the following API for registering and unregistering the lookups:

```
int phy_create_lookup(struct phy *phy, const char *con_id,
                      const char *dev_id);
void phy_remove_lookup(struct phy *phy, const char *con_id,
                       const char *dev_id);
```

DeviceTree Binding

The documentation for PHY dt binding can be found @ Documentation/devicetree/bindings/phy/phy-bindings.txt