Platform Devices and Drivers

- There are devices connected to CPU through different bus interfaces (**ex: I2C, SPI, PCI, and USB etc.**)
- These buses can be divided in two types
- **Discoverable**
    - The devices which are connected to PCI or USB interfaces are auto discoverable devices.
    - It supports hot plugging of devices
    - Devices connected to PCI/USB bus can tell the system what kind of devices it is and where its resources are. So, the kernel can enumerate the available device and driver to initialized device using probe method.
- **Non-Discoverable**
    - In system, there are devices which are not connected to any standard bus and because of this these devices are not hot-pluggable and non-discoverable. All non-discoverable devices are called platform devices.
    - So, for non-discoverable devices, platform device/driver comes in picture.
    - For example: I2C, SPI buses.

Platform bus

- Linux has given a common name to all these non-discoverable buses and that name is platform bus.
- From Linux point of view, all these devices which are non-discoverable are connected to the CPU through platform bus. But from hardware point of view, all these devices are connected to the CPU through different buses (**I2C/SPI).**
- Platform bus is not physically existed. It is just a Linux's virtual bus.

Platform Driver

- A driver who is responsible for handling platform devices are called platform drivers.

Register platform device and driver

Platform driver registration

- To register your platform driver to the kernel, use **platform_driver_register(drv)** macro defined in **include/linux/platform_device.h.**

Platform driver structure

- A platform driver is represented by a structure called **struct platform_driver**.

```
struct platform_driver {
        int (*probe)(struct platform_device *);
        int (*remove)(struct platform_device *);
        void (*shutdown)(struct platform_device *);
        int (*suspend)(struct platform_device *, pm_message_t state);
        int (*resume)(struct platform_device *);
        struct device_driver driver;
        const struct platform_device_id *id_table;
        bool prevent_deferred_probe;
        /*
         * For most device drivers, no need to care about this flag as long as
         * all DMAs are handled through the kernel DMA API. For some special
         * ones, for example VFIO drivers, they know how to manage the DMA
         * themselves and set this flag so that the IOMMU layer will allow them
         * to setup and manage their own I/O address space.
         */
        bool driver_managed_dma;
};
```

- **Note: Some member elements are mandatory to initialize.**

Platform device register

- To register your platform device, use **platform_device_register(struct platform_device* pdev).**
- Platform device is represented by structure called **struct platform_device**

```
struct platform_device {
        const char      *name;
        int             id;
        bool            id_auto;
        struct device   dev;
        u64             platform_dma_mask;
        struct device_dma_parameters dma_parms;
        u32             num_resources;
        struct resource *resource;

        const struct platform_device_id *id_entry;
        /*
         * Driver name to force a match.  Do not set directly, because core
         * frees it.  Use driver_set_override() to set or clear it.
         */
        const char *driver_override;

        /* MFD cell pointer */
        struct mfd_cell *mfd_cell;

        /* arch specific additions */
        struct pdev_archdata    archdata;
};
```

Probe function

- The kernel calls the probe() function of the driver when it discovers the corresponding platform device.
- Probe function is responsible for:
    - Initialization
    - Registering interrupt handlers.
    - Mapping I/O memory

- Registering device to the kernel framework.

- It returns 0 on success or error code on failure.

Remove function

- Remove function of the driver will get called by the kernel when the corresponding platform device is no longer used by the kernel.
- It is responsible for:
  - Unregistering the device from the kernel framework
  - Freeing allocated memory
  - Shutdown/De-initialize the device

Platform devices and Platform drivers linking

- There are three mechanisms to bind the actual devices to the driver.
  - id_table argument
  - name field of the driver structure
  - device tree
- If id_table is present, the platform bus code will scan through it every time it has to find a driver for a new platform device. If the device's name matches the name in an id_table entry, then the device will be given to the driver for management.
- Most platform drivers do not provide id_table Instead they provide a name of the driver in driver field.
- Using device tree:
  - platform device needs to be specified in the device tree for the platform bus driver to match it against this driver when it is loaded.
  - The compatible field entry in the device tree is matched against the one mentioned in the driver file.

platform_get_resource

- Syntax:
  - **struct resource* platform_get_resource (struct platform_device* dev, unsigned int type, unsigned int num);**
- get a resource for a device
- **Arguments:**
  - ***struct* platform_device *dev**: Platform device
  - ***type:*** *resource type*
  - ***num:*** which resource of that type is desired, with zero indicating the first one
- **Return:** pointer to the resource or NULL on failure.

platform_get_irq

- Syntax:
  - int platform_get_irq (struct platform_device* dev, unsigned int num);
- Gets an IRQ for a platform device and prints an error message if finding the IRQ fails.
- **Arguments:**
  - **struct platform_device* dev:** platform device
  - **unsigned int num:** IRQ number index

- **Return:** non-zero IRQ number on success, negative error number on failure.

Implementation

- Create **platform.h** file and define its driver name, start address, end address and IRQ number.

```
#ifndef PLATFORM_H
#define PLATFORM_H

#define DRIVER_NAME "platform_driver"
#define START_ADD 0x200000
#define END_ADD   0x2FFFFF

#define DEV_IRQ_NUM   6
#endif
```

- Create a **platform_dev.c** file and register your platform device.
- Create a **platform_driver.c** file and register your platform driver with probe and remove method
- Create a **Makefile**

```
obj-m += platform_driver.o platform_dev.o

all:
    make -C /lib/modules/$(shell uname -r)/build M=$(shell pwd) modules
clean:
    make -C /lib/modules/$(shell uname -r)/build M=$(shell pwd) clean
```

- Compile the driver using make command
- Install the driver with **insmod** command (sudo insmod platform_dev.ko , sudo insmod platform_driver.ko)
- See the output of **dmesg**

```
[ 1525.003571]
                Memory Area1
[ 1525.003579] Start:200000,  End:2fffff Size:1048576
[ 1525.003581]
                IRQ Number of Device :6
```

- Remove the driver with **rmmod** command. (sudo rmmod platform_dev.ko, sudo rmmod platform_driver.ko)

Code

- https://github.com/Jainam103/Platform_Driver.git