Simple understanding of Linux platform_driver

原创 yanhe156 Posted at 2018-01-15 11:51:48 1445 Favorite 6 copyright

Category column: zedboard linux

zedboard linux The column includes this content 0 subscribe 23 articles subscribe column

Simple understanding of Linux platform_driver

- GitHub project
- This project is the AXIDMA driver of the Zynq platform, which may be more complicated than the general driver.
- This article is mainly to learn how to write its platform driver, and does not look at the part that uses DMA Engine

drive entry

- When using the platform mode (device driver mode introduced after linux 2.6),
  when using the platform mode to develop the driver, the code structure includes two parts:
  - platform_device (hardware structure, first address, offset address, resource type is memory or interrupt number, etc.)
  - platform_driver (operation)

Where platform_device can be defined in the device tree.

```
static int __init axidma_init(void)
{
    return platform_driver_register(&axidma_driver);
}
static void __exit axidma_exit(void)
{
    return platform_driver_unregister(&axidma_driver);
}
module_init(axidma_init);
module_exit(axidma_exit);
```

- **What is the platform driver** ?
  A real Linux device and driver are usually attached to a bus, which is naturally not a problem for devices that are attached to PCI, USB, I2C, SPI, etc., but in embedded systems, integrated in SoC systems Independent peripheral controllers, peripherals attached to SoC memory space, etc. are not attached to such buses. Based on this background, Linux invented a virtual bus called **platform bus** , the corresponding device is called **platform device** , and the driver is called **platform driver** .
  Note platform device is not a concept alongside character devices, block devices and network devices, but a way to implement them. What is implemented in this way is called a platform device.

- The process of developing platform devices
  - Define the initialization platform bus
  - Define the platform device
  - register platform device
  - Define the relevant platform driver
  - Register related platform driver
  - Operate related equipment
- `platform_driver_register()` The function of the function: register plarform driver

  The definition of this function in the kernel:

```
int platform_driver_register(struct platform_driver *drv)
{
       drv->driver.bus = &platform_bus_type;//它将要注册到的总线
            /*设置成platform_bus_type这个很重要，因为driver和device是通过bus联系在一起的，
            具体在本例中是通过 platform_bus_type中注册的回调例程和属性来是实现的，
            driver与device的匹配就是通过 platform_bus_type注册的回调例程platform_match ()来完成的。
            */
       if (drv->probe)
              drv-> driver.probe = platform_drv_probe;
       if (drv->remove)
              drv->driver.remove = platform_drv_remove;
       if (drv->shutdown)
              drv->driver.shutdown = platform_drv_shutdown;
       return driver_register(&drv->driver);//注册驱动
}
```

It can be seen that when the platform driver is registered, the `driver_register()` function is finally called.
- `platform_driver_register()` The parameter
parameter is `struct platform_driver *drv` ,
here is to `&axidma_driver` view its definition:

```
static struct platform_driver axidma_driver = {
    .driver = {
        .name = MODULE_NAME,
        .owner = THIS_MODULE,
        .of_match_table = axidma_compatible_of_ids,
    },
    .probe = axidma_probe,
    .remove = axidma_remove,
};
```

Device tree replaces platform device

- The platform device (including id, address, etc.) is not directly defined in the driver, but obtained from the device tree.
For zynq, you can use xilinx sdk to generate a device tree through official documents and current hardware circuits.
- How to match device and driver?
When directly defining platform device, use the variable .name in the structure to match with driver. Looking at the kernel source code, it can be found that when obtained through the device tree, it is matched through the attribute **"compatible" .**
So when defining platform_driver, it is written like this:

```
static const struct of_device_id axidma_compatible_of_ids[] = {
    { .compatible = "xlnx,axidma-chrdev" },
    {}
};

static struct platform_driver axidma_driver = {
    .driver = {
        .name = MODULE_NAME,
        .owner = THIS_MODULE,
        .of_match_table = axidma_compatible_of_ids,
    },
    .probe = axidma_probe,
    .remove = axidma_remove,
};
```

If it is not obtained from the device tree, the member .of_match_table is not added when filling the .driver.
At the same time, looking at the source code, it is found that there are many ways to match the driver and device, and these different ways also have different priorities. If the high-priority match is successful, then return 1, that is, the match is successful. It seems that

through "compatible" matching Highest priority.

- **`.prob` and `.remove` the corresponding function usage**

The platform device `.prob` function executes roughly four steps:

1. Apply for the device number
2. The **physical device (if it is a character device, it is cdev) is initialized and registered, and the filling is added `file_operation`**
3. Read the hardware resources from pdev
4. For the hardware Resource initialization, `ioremap()`, `request_irq()` and other operations

The function of the platform device `.remove` implements the cancellation of various resources allocated.

- **Note `platform_driver_register(), platform_driver_unregister()` that only the registration and deregistration of the platform driver is implemented.**
    - So why use platform driver instead of directly writing drivers for character devices, block devices, and network devices?
        - Because the bus-device-driver mode is very useful, so if there is no peripheral device connected to an actual bus to use this mode, then make a virtual bus, that is, the platform bus. At the same time, the process of driver development is also unified.
          (The peripherals integrated in the SoC uniformly address the control registers, such as zynq development in vivado, the added IP is uniformly addressed, that is, there is an Address Editor interface after adding, and the address can be seen)
            - So why is this pattern useful?
              For example, an SPI bus (usually connected to multiple devices), different hardware will be connected to different devices, such as lcd, sound card, NAND, etc. For example, when using NAND, it is normal to directly write a driver for SPI to operate NAND, but when the NAND device is updated, the corresponding driver also needs to be updated, which will affect the SPI part. But if you use the bus-device-driver mode, the SPI bus itself is a module that does not need any modification. It does not know what kind of device it is connected to. Until the register, the .prob function is called.
        - Related to Bootloader? I don't understand
          . Provide platform_device and platform_driver data structures, embed traditional device and driver data structures, and add resource members to facilitate integration with OpenFirmware, a new type of bootloader and kernel that dynamically transfers device resources.

refer to

- http://blog.csdn.net/zqixiao_09/article/details/50888795
- http://blog.csdn.net/zqixiao_09/article/details/50889458
- http://blog.csdn.net/zqixiao_09/article/details/50865480
- https://www.zhihu.com/question/33414159?sort=created
- http://blog.csdn.net/faxiang1230/article/details/46414227