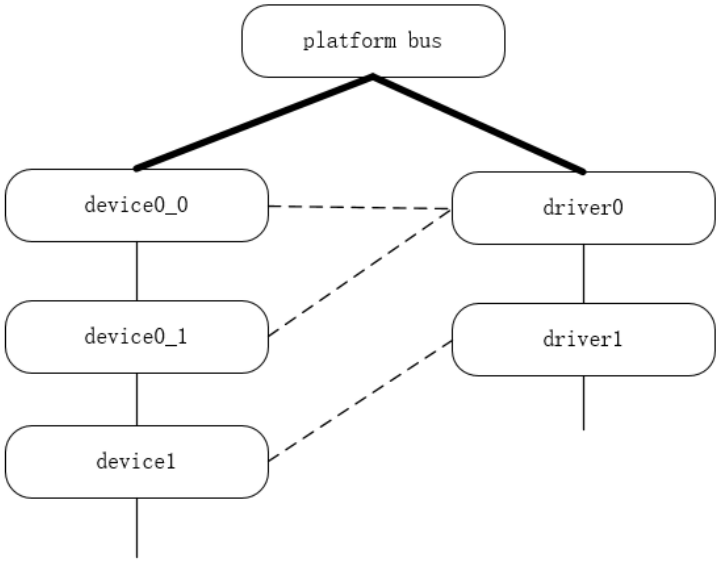
ONE_Tech

Next Mile is My Destination 博客新家: sketch2sky.com , 欢迎交流

Linux Platform驱动模型(一) _设备信息

我在Linux字符设备驱动框架一文中简单介绍了Linux字符设备编程模型，在那个模型中，只要应用程序**open()**了相应的设备文件，就可以使用ioctl通过驱动程序来控制我们的硬件，这种模型直观，但是从软件设计的角度看，却是一种十分糟糕的方式，它有一个致命的问题，就是设备信息和驱动代码冗余在一起，一旦硬件信息发生改变甚至设备已经不在了，就必须要修改驱动源码，非常的麻烦，为了解决这种驱动代码和设备信息耦合的问题，Linux提出了**platform bus(平台总线)**的概念，即使用虚拟总线将**设备信息和驱动程序进行分离**，设备树的提出就是进一步深化这种思想，将设备信息进行更好的整理。平台总线会维护两条链表，分别管理设备和驱动，当一个设备被注册到总线上的时候，总线会根据其名字搜索对应的驱动，如果找到就将设备信息导入驱动程序并执行驱动；当一个驱动被注册到平台总线的时候，总线也会搜索设备。总

之，平台总线负责将设备信息和驱动代码匹配，这样就可以做到驱动和设备信息的分离。



在设备树出现之前，设备信息只能使用C语言的方式进行编写，在3.0之后，设备信息就开始同时支持两种编写方式——设备树、C语言，如果用设备树，**手动**将设备信息写到设备树中之后，内核就可以**自动**从设备树中提取相应的设备信息并将其封装成相应的platform_device对象，i2c_device对象并注册到相应的总线中，如果使用**设备树**，我们就不需要对设备信息再进行编码。如果使用C语言，显然，我们需要将使用内核提供的结构将设备信息进行**手动**封装，这种封装又分为两种形式，一种是使用**平台文件(静态)**，将整个板子的所有设备都写在一个文件中并编译进内核。另一种是使用**模块(动态)**，将我们需要的设备信息编译成模块在insmod进内核。对于ARM平台，使用设备树封装设备信息是将来的趋势，但是由于历史原因，当下的内核中这三种方式并存。封装好后再创建相应的xxx_device实例最后注册到总线中。针对平台总线的设备信息，我在Linux设备树语法详解一文中已经讨论了设备树的写法，所以，本文主要讨论4个问题：

1. 如何使用C语言封装设备信息?
2. 设备树的设备信息和C语言的设备信息如何转换?
3. 如何将C语言设备信息封装到platform_device结构中?
4. 如何将封装好的platform_device结构注册到平台总线中?

何为资源?

所谓的设备信息，主要分为两种：硬件信息、软件信息，硬件信息主要包括xxx控制器在xxx地址上，xxx设备占用了xxx中断号，即**地址资源**，**中断资源**等。内核提供了struct resource来对这些资源进行封装。软件信息的种类就比较多样，比如网卡设备中的MAC地址等等，这些信息需要我们以私有数据的形式封装的设备对象(内核使用面向对象的思想编写，一个设备的设备信息是一个对象，即一个结构体实例，一个设备的驱动方法也是一个对象)中，这部分信息就需要我们自定义结构进行封装。

struct resource那点事

这个结构用来描述一个地址资源或中断资源，除了这个结构，内核还提供了一些宏来帮助我们快速的创建一个resource对象。

```
//include/linux/ioport.h
 18 struct resource {
 19         resource_size_t start;
 20         resource_size_t end;
 21         const char *name;
 22         unsigned long flags;
 23         unsigned long desc;
 24         struct resource *parent, *sibling, *child;
 25 };
```

> struct resource
> --19--> **start**表示资源开始的位置，如果是IO地址资源，就是起始物理地址，如果是中断资源，就是中断号;
> --20--> **end**表示资源结束的位置，如果是IO地址地址，就是映射的最后一个**物理地址**，如果是中断资源，就不用填;
> --21--> **name**就是这个资源的名字。
> --22--> **flags**表示资源类型，提取函数在寻找资源的时候会对比自己传入的参数和这个成员，理论上只要和可以随便写，但是合格的工程师应该使用内核提供的宏，这些宏也在**"ioport.h"**中进行了定义，比如**IORESOURCE_MEM**表示这个资源是地址资源，**IORESOURCE_IRQ**表示这个资源是中断资源...。

```
//include/linux/ioport.h
 33 #define IORESOURCE_BITS         0x000000ff      /* Bus-specific bits */
 35 #define IORESOURCE_TYPE_BITS    0x00001f00      /* Resource type */
 36 #define IORESOURCE_IO           0x00000100      /* PCI/ISA I/O ports */
 37 #define IORESOURCE_MEM          0x00000200
 38 #define IORESOURCE_REG          0x00000300      /* Register offsets */
 39 #define IORESOURCE_IRQ          0x00000400
 40 #define IORESOURCE_DMA          0x00000800
 41 #define IORESOURCE_BUS          0x00001000
 ...
147 #define DEFINE_RES_IO(_start, _size)
152 #define DEFINE_RES_MEM(_start, _size)
157 #define DEFINE_RES_IRQ(_irq)
162 #define DEFINE_RES_DMA(_dma)
```

有了这几个属性，就可以完整的描述一个资源，但如果每个资源都需要单独管理而不是组成某种数据结构，显然是一种非常愚蠢的做法，所以内核的resource结构还提供了三个指针：**parent,sibling,child(24)**，分别用来表示资源的父资源，兄弟资源，子资源，这样内核就可以使用树结构来高效的管理大量的系统资源，linux内核有两种树结构：iomem_resource,ioport_resource，进行板级开发的时候，通常将主板上的ROM资源放入iomem_resource树的一个节点，而将系统固有的I/O资源挂到ioport_resource树上。

下面是一个小例子，分别用两种写法表示了地址资源和中断资源，强烈推荐使用**DEFINE_RES_XXX**的版本。

```
//IO地址资源，自己填充resource结构体+flags宏
struct resource res= {
        .start = 0x10000000,
        .end     = 0x20000000-1,
        .flags = IORESOURCE_MEM
};
```

```
//IO地址资源，使用内核提供的定义宏
struct resource res = DEFINE_RES_MEM(0x20000000, 1024);
```

```
//中断资源，自己填充resource结构体+flags宏
struct resource res = {
                .start  = 10,
                .flags  = IORESOURCE_IRQ,
};
```

```
//中断资源，使用内核提供的定义宏
struct resource res = DEFINE_RES_IRQ(11);
```

下面是一个资源数组的实例，多个资源的时候就写成数组，这里我同时使用了上面两种写法。

```
struct resource res[] = {
        [0] = {
                .start  = 0x10000000,
                .end    = 0x20000000-1,
                .flags  = IORESOURCE_MEM
        },
        [1] = DEFINE_RES_MEM(0x20000000, 1024),
        [2] = {
                .start  = 10,   //中断号
                .flags  = IORESOURCE_IRQ|IRQF_TRIGGER_RISING
//include/linux/interrupt.h
        },
        [3] = DEFINE_RES_IRQ(11),
};
```

resource VS dts

至此，我们已经讨论了使用设备树和resource结构两种方式写设备信息，显然，这两种方式最终是殊途同归的，这里我们简单的讨论一个二者之间的转换问题。下图是我在Linux设备树语法详解一文中用到的dm9000网卡的节点

```
81          srom-cs1@5000000{
82                  compatible = "simple-bus";
83                  #address-cells = <1>;
84                  #size-cells = <1>;
85                  reg = <0x05000000 0x01000000>;
86                  ranges;
87                  ethernet@5000000 {
88                          compatible = "davicom,dm9000";
89                          reg = <0x05000000 0x2 0x05000004 0x2>;
90                          interrupt-parent = <&gpx0>;
91                          interrupts = <6 4>;
92                          local-mac-address = [00 00 de ad be ef];
93                          davicom,no-eeprom;
94                  };
```

将它的地址资源写成resource就是这个样子，清晰起见，这里也是两种写法：

```
struct resource res[] = {
    [0] = {
        .start = 0x05000000,
        .end = 0x05000000+0x2-1,
        .flags = IORESOURCE_MEM,
    },
    [1] = DEFINE_RES_MEM(0x05000004,2),
};
```

platform_device对象

这个对象就是我们最终要注册到平台总线上的设备信息对象，对设备信息进行编码，其实就是创建一个platform_device对象，可以看出，platform_device和其他设备一样，都是device的子类

```
//include/linux/platform_device.h
 22 struct platform_device {
 23         const char      *name;
 24         int             id;
 25         bool            id_auto;
 26         struct device   dev;
 27         u32             num_resources;
```

```
28          struct resource *resource;
29
30          const struct platform_device_id *id_entry;
31
32          /* MFD cell pointer */
33          struct mfd_cell *mfd_cell;
34
35          /* arch specific additions */
36          struct pdev_archdata    archdata;
37  };
```

在这个对象中，我们主要关心以下几个成员

> **struct platform_device**
> --23-->name就是设备的名字，注意， 模块名(lsmod)!=设备名(/proc/devices)!=设备文件名(/dev)，这个名字就是驱动方法和设备信息匹配的桥梁
> --24-->表示这个platform_device对象表征了几个设备，当多个设备有共用资源的时候(MFD)，里面填充相应的设备数量，如果只是一个，填-1
> --26-->父类对象(include/linux/device.h +722)，我们通常关心里面的**platform_data**和**release**,前者是用来存储私有设备信息的，后者是供当这个设备的最后引用被删除时被内核回调，注意和rmmod没关系。
> --27-->资源的数量，即resource数组中元素的个数，我们用**ARRAY_SIZE()**宏来确定数组的大小(include/linux/kernel.h +54 #define ARRAY_SIZE(arr) (sizeof(arr) / sizeof((arr)[0]))
> --28-->资源指针，如果是多个资源就是struct resource[]数组名，

下面是一个实例。

```
static struct platform_device demoobj = {
        //2. init obj
        .name    = "demo0",
        .id      = -1,
        .dev     = {
                .platform_data = &priv,
```

```
                .release = dev_release,
        },
        .num_resources  = ARRAY_SIZE(res),
        .resource       = res,
};
```

设备对象的注册与注销

准备好了platform_device对象，接下来就可以将其注册进内核，显然内核已经为我们准备好了相关的函数

```
/**
 *注册：把指定设备添加到内核中平台总线的设备列表，等待匹配，匹配成功则回调驱动中probe;
 */
int platform_device_register(struct platform_device *);
/**
 *注销：把指定设备从设备列表中删除，如果驱动已匹配则回调驱动方法和设备信息中的release;
 */
void platform_device_unregister(struct platform_device *);
```

通常，我们会将platform_device_register写在模块加载的函数中，将platform_device_unregister写在模块卸载函数中。我们可以模仿内核的宏写一个注册注销的快捷方式

```
#define module_platform_device(xxx)                     \
static int __init xxx#_init(void)                       \
{                                                       \
        return platform_device_register(&xxx);          \
}                                                       \
static void __exit xxx#_exit(void)                      \
{                                                       \
        platform_device_unregister(&xxx);               \
}                                                       \
module_init(xxx#_init);                                 \
module_exit(xxx#_exit);
```

彩蛋

Linux中几乎所有的"设备"都是"device"的子类，无论是平台设备还是i2c设备还是网络设备，但唯独字符设备不是，从"Linux字符设备驱动框架"一文中我们可以看出cdev并不是继承自device，从"Linux设备管理（二）_从cdev_add说起"一文中我们可以看出注册一个cdev对象到内核其实只是将它放到cdev_map中，直到"Linux设备管理（四）_从sysfs回到ktype"一文中对device_create的分析才知道此时才创建device结构并将kobj挂接到相应的链表，，所以，基于历史原因，当下cdev更合适的一种理解是一种接口(使用mknod时可以当作设备)，而不是而一个具体的设备，和platform_device,i2c_device有着本质的区别

分类: Linux驱动开发

标签: Linux Platform, 设备信息

« 上一篇： Linux设备文件三大结构：inode,file,file_operations
» 下一篇： Linux Platform驱动模型(二)_驱动方法

posted @ 2017-02-05 10:40 Abnor 阅读(9385) 评论(0) 编辑 收藏 举报