http://blog.chinaunix.net/uid/20937170/cid-143383-abstract-3.html

总线设备驱动模型总结 2011-12-19 14:43:47

分类： LINUX

我的环境：

主机开发环境：Fedora14

开发板： TQ2440

编译器： arm-linux-gcc-4.3.2

总线设备驱动模型其实现主要是基于 Kobject 和 sysfs 等机制，对于驱动模型程序开发主要是理解三个元素：总线、设备、驱动的关系。三者之间因为一定的联系性实现对设备的控制。

首先是总线，总线是三者联系起来的基础，通过一种总线类型，将设备和驱动联系起来。总线类型中的 match 函数用来匹配设备和驱动。当匹配操作晚餐之后就会控制驱动程序中的 probe 函数。

总线设备驱动模型的设计主要包括三个元素的注册，将三个元素加载到内核中，然后通过内核的内部机制将三者联系起来。

首先，总线类型的注册，包括属性文件的添加，总线也是一种设备，也需要将总线设备注册。

其次，完成设备的注册和添加以及对设备添加设备属性文件，同时填充最基本的函数操作。

最后，完成驱动的注册和天极以及对设备驱动添加属性文件，同时填充最基本的函数操作。

1、总线

总线类型是通过结构体 bus_type 表示的。其源码如下所示：

```
1.  struct bus_type {
2.          /*总线名*/
3.          const char          *name;
4.          /*总线、设备、驱动属性*/
5.          struct bus_attribute    *bus_attrs;
6.          struct device_attribute     *dev_attrs;
7.          struct driver_attribute     *drv_attrs;
8.
9.          /*总线支持的函数操作*/
10.         /*匹配函数，主要用来识别相应的设备和驱动，是两者直接形成关联
```

```
11.             用来判断指定的驱动程序能否处理指定的设备
12.         */
13.         int (*match)(struct device *dev, struct device_driver *drv);
14.         /*在进行热插拔事件之前，为设备配置环境变量操作函数*/
15.         int (*uevent)(struct device *dev, struct kobj_uevent_env *env);
16.         int (*probe)(struct device *dev);
17.         int (*remove)(struct device *dev);
18.         void (*shutdown)(struct device *dev);
19.
20.         int (*suspend)(struct device *dev, pm_message_t state);
21.         int (*suspend_late)(struct device *dev, pm_message_t state);
22.         int (*resume_early)(struct device *dev);
23.         int (*resume)(struct device *dev);
24.
25.         struct dev_pm_ops *pm;
26.
27.         struct bus_type_private *p;
28. };
```

其中的 int (*match)(struct device * dev, struct device_driver * drv)是必须实现的函数，因为这个函数主要是实现设备和驱动之间的匹配管理。其中匹配的具体逻辑关系需要驱动设计着设定。

int (*uevent)(struct device *dev, char **envp, int num_envp,char *buffer, int buffer_size)则在热插拔事件之前，允许总线为设备添加环境变量。

通常创建一种总线类型的过程中只要完成总线类型结构体的填充，然后完成相应的注册、属性文件创建即可实现总线类型的添加。并不需要对 bus_type 类型中的所有变量进行赋值，只要将其中的 name,bus_attribute,match 实现即可。

最后不要忘了总线也是设备，需要将总线设备添加到内核中（注册函数）。

关于总线类型的属性设置，实质上就是完成一个结构体的操作。

如下源码所示：

```
1. struct bus_attribute {
2.         /*属性结构体*/
3.         struct attribute    attr;
4.         /*属性读操作函数，即显示函数*/
5.         ssize_t (*show)(struct bus_type *bus, char *buf);
```

```
        /*属性写操作函数，也就是存储到结构体中*/
         ssize_t (*store)(struct bus_type *bus, const char *buf, size_t
count);
};
  /*可以通过宏命令定义一个总线结构体，但是需要自己实现属性读写操作函数，如果没有，
可设置为 NULL*/
  /*总线属性定义宏*/
  #define BUS_ATTR(_name, _mode, _show, _store) \
       struct bus_attribute bus_attr_##_name = __ATTR(_name, _mode, _show,
_store)


  /*__ATTR 的宏实现如下所示：*/
  #define __ATTR(_name,_mode,_show,_store) { \
.attr = {.name = __stringify(_name), .mode = _mode }, \
.show = _show, \
.store = _store, \
  }
```

由于通常情况下需要查找总线的版本信息，可以将版本信息添加到属性的读属性操作函数中，这样就能显示具体的版本信息。

在宏定义中##是指链接符的作用，相当于将两个部分链接起来，比如 **bus_attr_##name = bus_attr_name**。这是在宏定义中比较常用的定义方式之一。

总线类型的注册和总线类型属性文件创建，以及总线设备的注册主要是依据下面几个函数来实现：

```
/*总线类型注册函数，由于可能会出错，因此必须对返回值进行检查*/
int __must_check bus_register(struct bus_type *bus);
/*总线类型释放函数*/
void bus_unregister(struct bus_type *bus);
    /*总线文件属性创建函数，将相关的文件属性添加给总线类型，同时也必须检查返回值是否正确*/
int __must_check bus_create_file(struct bus_type *,struct bus_attribute *);
/*总线类型文件属性删除函数，将两者之间的关联性切断*/
void bus_remove_file(struct bus_type *, struct bus_attribute *);
```

最后需要将总线设备添加到系统中，主要采用设备注册函数；

设备注册函数：

```
int __must_check device_register(struct device *dev);
```

设备释放函数：

```
void device_unregister(struct device *dev);
```

2、设备

设备的实现主要是依靠 struct device 函数实现的，设备的实现主要是对结构体的填充。实现相应的函数即可。

```
struct device {
        /*父设备，通常就是总线设备，这也是为什么需要将总线作为设备添加的原因*/
        struct device        *parent;

        struct device_private    *p;

        struct kobject kobj;
       /*init_name 是新添加的，替代了原来的 bus_id,但是 init_name 不能直接被读写操作*/
        const char        *init_name; /* initial name of the device */
        struct device_type    *type;

        struct semaphore    sem;    /* semaphore to synchronize calls to
                                     * its driver.
                                     */
       /*总线类型，主要是关联总线类型，这是前面添加的总线类型，通过相同的总线类型关联设备和驱动*/
        struct bus_type    *bus;        /* type of bus device is on */
        struct device_driver *driver;    /* which driver has allocated
this device */
        void        *driver_data;    /* data private to the driver */
        void        *platform_data;    /* Platform specific data, device
                                       core doesn't touch it */
        struct dev_pm_info    power;

#ifdef CONFIG_NUMA
        int        numa_node;    /* NUMA node this device is close to */
#endif
        u64        *dma_mask;    /* dma mask (if dma'able device) */
        u64        coherent_dma_mask; /* Like dma_mask, but for
                                         alloc_coherent mappings as
```

```
                                                not all hardware supports
                                                64 bit addresses for consistent
                                                allocations such descriptors. */

        struct device_dma_parameters *dma_parms;

        struct list_head    dma_pools;    /* dma pools (if dma'ble) */

        struct dma_coherent_mem    *dma_mem; /* internal for coherent mem
                                                override */
        /* arch specific additions */
        struct dev_archdata    archdata;

        dev_t            devt;    /* dev_t, creates the sysfs "dev" */

        spinlock_t        devres_lock;
        struct list_head    devres_head;

        struct klist_node    knode_class;
        struct class        *class;
        struct attribute_group    **groups;    /* optional groups */

        /*必须实现的 release 函数*/
        void    (*release)(struct device *dev);
};

/*由于 init_name 不能直接读写，只能通过*dev_name 来读写设备名*/
static inline const char *dev_name(const struct device *dev)
{
        return kobject_name(&dev->kobj);
}
/*实现对设备名的设置*/
int dev_set_name(struct device *dev, const char *name, ...)
                            __attribute__((format(printf, 2, 3)));

/*设备文件属性结构体，必须注意的改变点*/
```

```
struct device_attribute {
        /*属性值*/
        struct attribute    attr;
       /*设备属性读函数，必须注意是三个参数，不再是两个参数*/
        ssize_t (*show)(struct device *dev, struct
device_attribute *attr,char *buf);
       /*设备属性写操作，必须注意是四个参数，不是三个参数*/
        ssize_t (*store)(struct device *dev, struct
device_attribute *attr,const char *buf, size_t count);
};
```

```
    /*设备属性宏定义，主要用来实现设备文件属性*/
#define DEVICE_ATTR(_name, _mode, _show, _store) \
struct device_attribute dev_attr_##_name = __ATTR(_name, _mode, _show, _store)
    /*创建设备文件属性函数，必须检查返回值*/
int __must_check device_create_file(struct device *device,struct
device_attribute *entry);
/*删除设备文件属性函数*/
void device_remove_file(struct device *dev,struct device_attribute *attr);
/*设备注册函数，必须检查返回值*/
int __must_check device_register(struct device *dev);
/*设备释放函数*/
void device_unregister(struct device *dev);
```

需要注意的是linux-2.6.30内核以前，没有init_name元素，而是元素bus_id，这个主要是实现设备名的填充，但是linux-2.6.30内核之后的在struct device中init_name代替了bus_id,但是需要注意的是init_name不能直接被读写，当需要读写设备名时只能采用特定的函数实现:dev_name()，set_dev_name()。当直接读写init_name会导致内核错误，出现Unable to handle kernel NULL pointer dereference at virtual address 00000000的错误。

最后注意device_attribute中的show,store函数不同于其他类型（总线、驱动）的函数，device_attribute中的show和store函数中的参数数量多了一个。

3、驱动

驱动管理一定的设备，其中的关系主要是内核的内部机制实现的，但是实现的具体逻辑需要在bus_type中的match函数中具体设计。通常是一定的设备名和

驱动名匹配，当然也可以有其他的逻辑，具体的只需要设计好 bus_type 中的 match 函数。

驱动是由驱动结构体实现的。具体如下所示：

```
/*驱动结构体*/
struct device_driver {
        /*驱动名，通常用来匹配设备*/
        const char        *name;
        /*关联的总线类型，总线、设备、驱动关联的总线类型*/
        struct bus_type        *bus;

        struct module        *owner;
        const char        *mod_name;    /* used for built-in modules */

       /*驱动中最应该实现的操作函数主要包括 probe 和 remove 函数*/
        /*当匹配完成以后的，入口函数*/
        int (*probe) (struct device *dev);
       /*驱动卸载时操作的相关函数，退出函数*/
        int (*remove) (struct device *dev);

        void (*shutdown) (struct device *dev);
        int (*suspend) (struct device *dev, pm_message_t state);
        int (*resume) (struct device *dev);
        struct attribute_group **groups;

        struct dev_pm_ops *pm;

        struct driver_private *p;
};

/*驱动注册函数，返回值必须检测*/
int __must_check driver_register(struct device_driver *drv);
/*驱动释放函数*/
void driver_unregister(struct device_driver *drv);

/*驱动属性结构体*/
```

```
struct driver_attribute {
        /*属性值*/
        struct attribute attr;
       /*属性读操作函数*/
        ssize_t (*show)(struct device_driver *driver, char *buf);
       /*属性写操作函数*/
        ssize_t (*store)(struct device_driver *driver, const char *buf,
                        size_t count);
};

/*驱动属性定义宏命令*/
#define DRIVER_ATTR(_name, _mode, _show, _store)    \
struct driver_attribute driver_attr_##_name =       \
        __ATTR(_name, _mode, _show, _store)
```

```
/*驱动属性文件创建函数，返回值必须检测*/
int __must_check driver_create_file(struct device_driver *driver,struct
driver_attribute *attr);
/*驱动属性文件移除函数*/
void driver_remove_file(struct device_driver *driver,struct driver_attribute *attr);
```

驱动结构体的定义不需要完成所有元素的赋值，只需要完成主要的几个变量的赋值即可，其中主要的元素包含 name,bus,以及 probe 和 remove 函数的实现。

其中的 probe 函数是当总线中的 match 完成匹配操作以后，进入驱动的入口函数，因此必须实现。remove 我认为就是对应的退出函数，因此也有必要实现。

驱动的注册,释放也有相关的函数来操作，主要是 driver_register()和 driver_unregister()。

总结：

1、在总线驱动模型中我认为最主要的是搞清楚三个不同的结构体，分别是总线、驱动、设备。了解三个元素对应的属性结构体以及相应的属性操作函数的差异性。

2、不同驱动设计的关键主要是完成不同结构体的填充过程，但是并不需要对结构体中所有的对象进行赋值，只需要完成重要的几个元素的值。

3、总线是一种类型，同时也是一种设备，在总线的相关处理中需要首先添加总线类型，然后添加总线设备，这是需要注意的。由于总线类型关联驱动和设备，因此需要导出总线类型变量。由于总线设备是设备的父设备，因此也需要将总线设备变量导出。同样在驱动和设备中也要导出相关的结构体变量，便于总线中的 match 函数实现驱动和设备的匹配操作。

4、XXX_attr 结构体基本相同，都是一个属性结构体和函数 show()、stroe()。但是不同的 XXX 可能会导致 show、stroe 函数的参数发生变化。这需要对照源码。

5、struct device 中的 init_name 是一个特殊的量，不能直接读写操作，只能采用函数 device_name()和 set_device_name 来设置设备名。

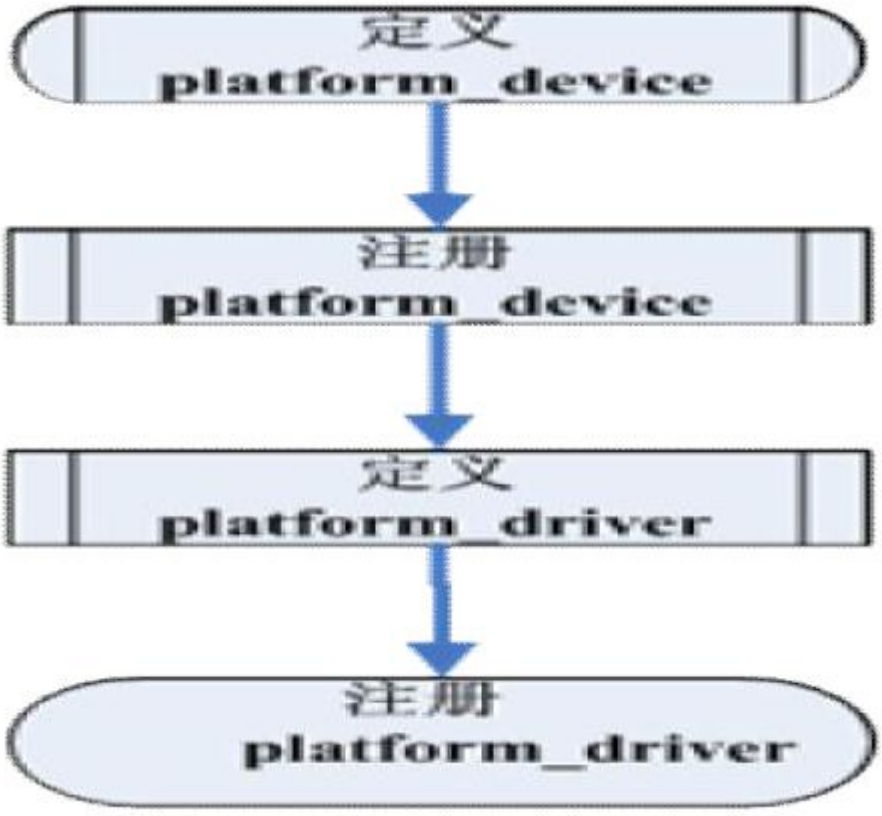
6、xxx_register()之类的函数，需要对返回值进行检查。因为很有可能不成功。

platform 平台总结 2011-12-20 21:35:50

分类： LINUX

总线设备驱动模型主要包含总线、设备、驱动三个部分，总线可以是一条真实存在的总线，例如 USB、I2C 等典型的设备。但是对于一些设备（内部的设备）可能没有现成的总线。Linux 2.6 内核中引入了总线设备驱动模型。总线设备驱动模型与之前的三类驱动（字符、块设备、网络设备）没有必然的联系。设备只是搭载到了总线中。在 linux 内核中假设存在一条虚拟总线，称之为 platform 总线。

platform 总线相比与常规的总线模型其优势主要是 platform 总线是由内核实现的，而不用自己定义总线类型，总线设备来加载总线。platform 总线是内核已经实现好的。只需要添加相应的 platform device 和 platform driver。具体的实现过程主要包括如下的过程：



整体而言只需要完成两个步骤，也就是设备的实现和驱动的实现，每一个实现都包括相关结构体的定义和注册。

platform_device 注册

需要注意的是 platform_device 实质上是经过处理过的设备，在 platform_device 结构体中存在一个设备结构体，与之前的设备存在差别的是引入了设备资源。这些设备资源就能实现对设备寄存器，中断等资源的访问。平台设备的基本结构体如下：

```
struct platform_device {
    /*设备名*/
      const char       * name;
      /*设备 ID 号*/
      int               id;
    /*结构体包含一个具体的 device 结构体*/
      struct device       dev;
      /*资源的数量*/
      u32               num_resources;
      /*资源结构体，用来保存硬件的资源*/
      struct resource       * resource;
      /*平台设备的 ID*/
      struct platform_device_id       *id_entry;
};
```

其中 struct device 和 struct resource 是重要的结构体。struct device 在总线设备驱动模型中已经提到了。这次讨论一下 struct resource。

```
struct resource {
      /*资源的起始值，如果是地址，那么是物理地址，不是虚拟地址*/
      resource_size_t start;
      /*资源的结束值，如果是地址，那么是物理地址，不是虚拟地址*/
      resource_size_t end;
      /*资源名*/
      const char *name;
      /*资源的标示，用来识别不同的资源*/
      unsigned long flags;
      /*资源指针，可以构成链表*/
      struct resource *parent, *sibling, *child;
};
```

platform_device 的注册很简单，只需要在设备的初始化函数中首先定义相应的设备，通常采用函数 platform_device *platform_device_alloc(const char

*name, int id)动态申请，通常 name 就是需要申请的设备名，而 id 为-1。然后采用

int platform_device_add(struct platform_device *pdev)

或者

int platform_device_register(struct platform_device *pdev)

注册定义好的设备即可。

同样在退出函数中释放注册好的设备即可，可以采用函数：

void platform_device_unregister(struct platform_device *pdev)。

然后一个平台设备就完成了，不需要像自己实现模型时定义相关的文件属性等。

设备资源可以通过相关函数得到：

struct resource *platform_get_resource(struct platform_device *dev,
unsigned int type,
unsigned int num)

中断资源也可以通过：

int platform_get_irq(struct platform_device *dev, unsigned int num)

资源的使用主要是驱动实现过程中需要使用到的，但是后期的使用一般需要在驱动的 probe 函数中实现申请中断或者 IO 内存才能使用，而不能直接使用。特别是资源中的地址通常是物理地址，需要通过申请 IO 内存和映射完成物理到虚拟地址的转换，便于进程的访问。

platform_driver 注册

平台驱动结构体 platform_driver 实现如下：

```
struct platform_driver {
        /*平台驱动需要实现的相关函数操作，
        其中的前 4 个函数与最后一个函数与 device_driver 中的函数是相同的
      本质是实现对 device_driver 中相关函数的赋值。
      */
        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 (*suspend_late)(struct platform_device *, pm_message_t
state);
        int (*resume_early)(struct platform_device *);
        int (*resume)(struct platform_device *);
```

```
        /*内嵌了一个设备驱动结构体*/
        struct device_driver driver;
        /*平台设备 ID,这与 platform_device 中的 struct platform_device_id
*id_entry 是相同的
            主要是完成总线的匹配操作，platform 总线的匹配操作第一匹配要素
就是该元素。而不再是简单的 name 选项。
     */
        struct platform_device_id *id_table;
};
```

通常驱动的入口函数:

```
int (*probe)(struct platform_device *);
```

当总线完成了设备的 match 操作以后就会进入驱动中该函数的运行。

总线函数的匹配操作如下：

```
static int platform_match(struct device *dev, struct device_driver
*drv)
{
        /*得到平台设备的指针*/
        struct platform_device *pdev = to_platform_device(dev);
        /*得到平台驱动指针*/
        struct platform_driver *pdrv = to_platform_driver(drv);
        /* match against the id table first */
        /*从定义上分析，id_table 是首先匹配的对象，然后才是 name 的匹配，
当 ID 匹配完成时就说明匹配好了*/
        if (pdrv->id_table)
                return platform_match_id(pdrv->id_table, pdev) != NULL;
        /* fall-back to driver name match */
        return (strcmp(pdev->name, drv->name) == 0);
}
```

从上面的定义可以知道platform总线的匹配函数手下是比较id_table是匹配的首选项。

probe 函数称之为探针函数，用于检测总线上有该驱动能够处理的设备，而 remove 函数则是为了说明总线上该驱动能够处理的设备被移除。

因此这两个函数是在平台设备中一定要被实现的函数。

其他的函数则不一样要求实现。

平台驱动的设计主要是完成平台驱动结构体的填充和注册。

通常的平台驱动结构体实现如下：

```
static struct platform_driver my_driver =
{
                /*平台驱动的 probe 函数实现*/
                .probe = my_probe,
                /*平台驱动的 remove 函数实现*/
                .remove = my_remove,
                /*实现设备驱动的 name 和 owner 变量*/
                .driver =
                {
                                /*该参数主要实现总线中的匹配函数调用
*/
                                .name = "my_dev",
                                /*该函数表示模块的拥有者*/
                                .owner = THIS_MODULE,
                },
};
```

其中的 my_probe 和 my_remove 是自己定义的 probe 和 remove 函数。

最主要的是内嵌设备驱动结构体的填充，主要的填充包括 name 和 owner 两个，当然也可以包括其他的。由于没有填充 id_table,那么 name 就是总线匹配操作的第一选择。因此如果没有填充好 id_table,那么 name 元素是一定要实现的，不然不能完成相应的设备驱动匹配操作。

完成 platform_driver 结构体的填充过后就是完成驱动的在初始化阶段的注册以及退出阶段的释放操作，基本的实现函数为：

注册函数，通常在驱动初始化函数中调用：

```
int platform_driver_register(struct platform_driver *drv)
```

释放函数，通常在驱动退出函数调用：

```
void platform_driver_unregister(struct platform_driver *drv)
```

完成相关的注册以后总线、设备、驱动的大概框架就完成啦。

但是这只是常用的框架，还不能在应用程序中使用。

基于平台驱动的设备驱动都是基于总线架构的，基本的实现过程与之前的简单字符设备存在较大的差别，主要的区别在驱动的初始化不在是平台设备驱动的初始化函数中实现，而是在 probe 函数中实现。而驱动的卸载函数则是在 remove 函数中实现。probe 函数是平台总线实现匹配以后首先被调用的函数，因此在其中实现字符设备、块设备、网络设备驱动的初始化是有意义的，这样的设备驱动就是基于平台总线的设备驱动，便于维护。

平台总线驱动的注册过程分析：

```
int platform_driver_register(struct platform_driver *drv)
{
        /*第一步，仍然是完成结构体的填充操作*/
        /*驱动的总线类型*/
        drv->driver.bus = &platform_bus_type;
        /*将自己定义的probe函数赋值给平台驱动中设备驱动的probe函数,其他函数类似*/
        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;
        if (drv->suspend)
                drv->driver.suspend = platform_drv_suspend;
        if (drv->resume)
                drv->driver.resume = platform_drv_resume;
        /*第二步，仍然是完成一般设备驱动的注册操作*/
        /*然手就是一般驱动的注册，这样就完成了设备的注册*/
        return driver_register(&drv->driver);
}
/*设备驱动的 probe 函数的赋值过程*/
static int platform_drv_probe(struct device *_dev)
{
        /*得到设备对应的平台驱动*/
        struct platform_driver *drv = to_platform_driver(_dev->driver);
```

```
        /*得到设备的平台设备*/
          struct platform_device *dev = to_platform_device(_dev);
        /*下面的 probe 是自己实现的 probe 函数。具体的实现思路：
            根据一般设备找对应的平台设备，同时根据设备的驱动找到平台驱动。
          然后返回平台驱动的 probe 函数(自己实现通常是初始化操作)地址。
      */
          return drv->probe(dev);
}
```

实现的总线平台驱动模型的最简单源码:

平台设备的实现：device.c

```
#include
#include
#include
#include
#include
#include
/*平台模型驱动的平台设备对象*/
static struct platform_device *my_device;
/*初始化函数*/
static int __init my_device_init(void)
{
                int ret = 0;
                /*采用 platform_device_alloc 分配一个 platform_device 对
象
                    参数分别为 platform_device 的 name,和 id。
                */
                my_device = platform_device_alloc("my_dev",-1);
                /*注册设备,注意不是 platform_device_register，将平台设
备注册到内核中*/
                ret = platform_device_add(my_device);
                /*如果出错释放相关的内存单元*/
                if(ret)
                {
```

```
                platform_device_put(my_device);
        }
        return ret;
}
/*卸载处理函数*/
static void __exit my_device_exit(void)
{
        platform_device_unregister(my_device);
}
module_init(my_device_init);
module_exit(my_device_exit);
MODULE_LICENSE("GPL");
MODULE_AUTHOR("GP-");
```

平台驱动的实现：driver.c

```
#include
#include
#include
#include
#include
```

/*平台驱动中的 probe 和 remove 函数是必须实现的函数*/

```
/*设备驱动的探测函数，主要实现检测总线上是否有该驱动对应的设备*/
static my_probe(struct device *dev)
{
        /*
```

如果添加实际的设备到该平台总线设备驱动模型中，则可以在该函数

中实现具体的设备驱动函数的初始化操作，包括设备号的申请，设备

的初始化，添加。自动设备文件创建函数的添加等操作。

```
                或者是混杂字符设备的相关初始化操作。当然结构体的相关处理仍
                然采取全局变量的形式。
        */
        printk("Driver found devices which this driver can be handle\n");
        return 0;
}
/*设备驱动的移除函数，主要检测该驱动支持设备的移除活动检测*/
static my_remove(struct device *dev)
{
        /*
                如果添加实际的设备到该平台总线设备驱动模型中，则可以在该函数
                中实现具体的设备的释放，包括设备的删除，设备号的注销等操作。
        */
        printk("Driver found device unpluded\n");
        return 0;
}
static struct platform_driver my_driver =
{
        /*平台驱动的 probe 函数实现*/
        .probe = my_probe,
        /*平台驱动的 remove 函数实现*/
        .remove = my_remove,
        /*实现设备驱动的 name 和 owner 变量*/
        .driver =
        {
                /*该参数主要实现总线中的匹配函数调用*/
                .name = "my_dev",
                /*该函数表示模块的拥有者*/
                .owner = THIS_MODULE,
```

```
                },
};
/*初始化函数*/
static int __init my_driver_init(void)
{
                /*注册平台驱动*/
                return platform_driver_register(&my_driver);
}
/*退出函数*/
static void __exit my_driver_exit(void)
{
                /*注销平台驱动*/
                return platform_driver_unregister(&my_driver);
}
/*加载和卸载*/
module_init(my_driver_init);
module_exit(my_driver_exit);
MODULE_LICENSE("GPL");
MODULE_AUTHOR("GP-");
```

将一般设备驱动加入到总线设备模型中的相关操作是后期总结和学习的内容。设备驱动实现的实现原理还是之前的那些操作，但是初始化和推出函数发生了改变。

总结：

platform 总线的驱动模型只是在一般总线模型的基础上做了相关的延伸，实质上只要弄清除总线模型的一般原理，学习 platform 总线也就简单不少。但是毕竟还是学习阶段。