# Linux设备模型(9)_device resource management

作者：wowo 发布于：2014-9-24 23:28 分类：统一设备模型

### 1. 前言

**蜗蜗建议，每一个Linux驱动工程师，都能瞄一眼本文。**

之所以用“瞄”，因此它很简单，几乎不需要花费心思就能理解。之所有这建议，是因为它非常实用，可以解答一些困惑，可以使我们的代码变得简单、简洁。先看一个例子：

```
1: /* drivers/media/platform/soc_camera/mx1_camera.c, line 695 */
2: static int __init mx1_camera_probe(struct platform_device *pdev)
3: {
4:     ...
5:
6:     res = platform_get_resource(pdev, IORESOURCE_MEM, 0);
7:     irq = platform_get_irq(pdev, 0);
8:     if (!res || (int)irq <= 0) {
9:         err = -ENODEV;
10:        goto exit;
11:    }
12:
13:    clk = clk_get(&pdev->dev, "csi_clk");
```

相信每一个写过Linux driver的工程师，都在probe函数中遇到过上面的困惑：要顺序申请多种资源（IRQ、Clock、memory、regions、ioremap、dma、等等），只要任意一种资源申请失败，就要回滚释放之前申请的所有资源。于是函数的最后，一定会出现很多的goto标签（如上面的exit_free_irq、exit_free_dma、等等），并在申请资源出错时，小心翼翼的goto到正确的标签上，以便释放已申请资源。

正像上面代码一样，整个函数被大段的、重复的“if (condition) { err = xxx; goto xxx; }”充斥，浪费精力，容易出错，不美观。有困惑，就有改善的余地，最终，Linux设备模型借助device resource management（设备资源管理），帮我们解决了这个问题。就是：driver你只管申请就行了，不用考虑释放，我设备模型帮你释放。最终，我们的driver可以这样写：

```
1: static int __init mx1_camera_probe(struct platform_device *pdev)
2: {
3:     ...
4:
5:     res = platform_get_resource(pdev, IORESOURCE_MEM, 0);
6:     irq = platform_get_irq(pdev, 0);
7:     if (!res || (int)irq <= 0) {
8:         return -ENODEV;
9:     }
10:
11:    clk = devm_clk_get(&pdev->dev, "csi_clk");
12:    if (IS_ERR(clk)) {
13:        return PTR_ERR(clk);
```

怎么做到呢？注意上面“devm_”开头的接口，答案就在那里。不要再使用那些常规的资源申请接口，用devm_xxx的接口代替。为了保持兼容，这些新接口和旧接口的参数保持一致，只是名字前加了“devm_”，并多加一个struct device指针。

### 2. devm_xxx

下面列举一些常用的资源申请接口，它们由各个framework（如clock、regulator、gpio、等等）基于device resource management实现。使用时，直接忽略“devm_”的前缀，后面剩下的部分，driver工程师都很熟悉。只需记住一点，driver可以只申请，不释放，设备模型会帮忙释放。不过如果为了严谨，在driver remove时，可以主动释放（也有相应的接口，这里没有列出）。

站内搜索

请输入关键词 搜索

功能

留言板
评论列表
支持者列表

最新评论

ctwillson
后续的是不是没了?
callme_friend
@pixiandouban：visio
callme_friend
@hit201j：太忙，没时间回复，需要的朋友可统一去以下链...
callme_friend
@icecoder：太忙，没时间回复，需要的朋友可统一去以下...
callme_friend
@就爱吃泡芙：太忙，没时间回复，需要的朋友可统一去以下链接下...
callme_friend
@今雨轩：太忙，没时间回复，需要的朋友可统一去以下链接下载：...

文章分类

Linux内核分析(11)
统一设备模型(15)
电源管理子系统(42)
中断子系统(15)
进程管理(19)
内核同步机制(18)
GPIO子系统(5)
时间子系统(14)
通信类协议(7)
内存管理(27)
图形子系统(1)
文件系统(4)
TTY子系统(6)
u-boot分析(3)
Linux应用技巧(13)
软件开发(6)
基础技术(13)
蓝牙(16)
ARMv8A Arch(13)
显示(3)
USB(1)
基础学科(10)
技术漫谈(12)
项目专区(0)
X Project(28)

随机文章

Concurrency Managed Workqueue之（一）：workqueue的基本概念

```
1: extern void *devm_kzalloc(struct device *dev, size_t size, gfp_t gfp);
2:
3: void __iomem *devm_ioremap_resource(struct device *dev,
4:   struct resource *res);
5: void __iomem *devm_ioremap(struct device *dev, resource_size_t offset,
6:   unsigned long size);
7:
8: struct clk *devm_clk_get(struct device *dev, const char *id);
9:
10: int devm_gpio_request(struct device *dev, unsigned gpio,
11:   const char *label);
12:
13: static inline struct pinctrl * devm_pinctrl_get_select(
```

**3. 什么是“设备资源”**

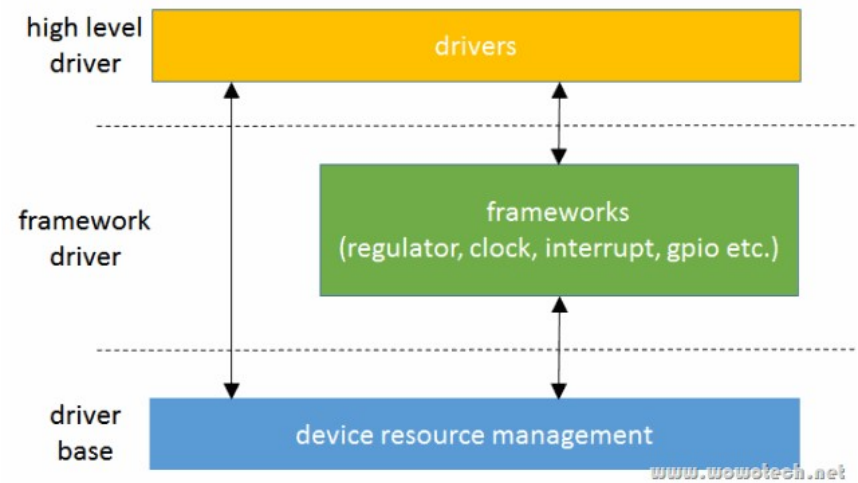
一个设备能工作，需要依赖很多的外部条件，如供电、时钟等等，这些外部条件称作设备资源（device resouce）。对于现代计算机的体系结构，可能的资源包括：

> a）power，供电。
> b）clock，时钟。
> c）memory，内存，在kernel中一般使用kzalloc分配。
> d）GPIO，用户和CPU交换简单控制、状态等信息。
> e）IRQ，触发中断。
> f）DMA，无CPU参与情况下进行数据传输。
> g）虚拟地址空间，一般使用ioremap、request_region等分配。
> h）等等

而在Linux kernel的眼中，“资源”的定义更为广义，比如PWM、RTC、Reset，都可以抽象为资源，供driver使用。

在较早的kernel中，系统还不是特别复杂，且各个framework还没有成型，因此大多的资源都由driver自行维护。但随着系统复杂度的增加，driver之间共用资源的情况越来越多，同时电源管理的需求也越来越迫切。于是kernel就将各个resource的管理权收回，基于“device resource management”的框架，由各个framework统一管理，包括分配和回收。

**4. device resource management的软件框架**




device resource management位于“drivers/base/devres.c”中，它的实现非常简单，为什么呢？因为资源的种类有很多，表现形式也多种多样，而devres不可能——知情，也就不能进行具体的分配和回收。因此，devres能做的（也是它的唯一功能），就是：

> 提供一种机制，将系统中某个设备的所有资源，以链表的形式，组织起来，以便在driver detach的时候，自动释放。

而更为具体的事情，如怎么抽象某一种设备，则由上层的framework负责。这些framework包括：regulator framework（管理power资源），clock framework（管理clock资源），interrupt framework（管理中断资源）、gpio framework（管理gpio资源），pwm framework（管理PWM），等等。

其它的driver，位于这些framework之上，使用它们提供的机制和接口，开发起来就非常方便了。


X-021-ROOTFS-基于busybox的最简rootfs的制作
蓝牙协议中LQ和RSSI的原理及应用场景
启动regulator framework分析任务
玩转BLE(1)_Eddystone beacon

**文章存档**

2018年10月(1)
2018年8月(1)
2018年6月(1)
2018年5月(1)
2018年4月(7)
2018年2月(4)
2018年1月(5)
2017年12月(2)
2017年11月(2)
2017年10月(1)
2017年9月(5)
2017年8月(4)
2017年7月(4)
2017年6月(3)
2017年5月(3)
2017年4月(1)
2017年3月(8)
2017年2月(6)
2017年1月(5)
2016年12月(6)
2016年11月(11)
2016年10月(9)
2016年9月(6)
2016年8月(9)
2016年7月(5)
2016年6月(8)
2016年5月(8)
2016年4月(7)
2016年3月(5)
2016年2月(5)
2016年1月(6)
2015年12月(6)
2015年11月(9)
2015年10月(9)
2015年9月(4)
2015年8月(3)
2015年7月(7)
2015年6月(3)
2015年5月(6)
2015年4月(9)
2015年3月(9)
2015年2月(6)
2015年1月(6)
2014年12月(17)
2014年11月(8)
2014年10月(9)
2014年9月(7)
2014年8月(12)
2014年7月(6)
2014年6月(6)
2014年5月(9)
2014年4月(9)
2014年3月(7)
2014年2月(3)
2014年1月(4)

## 5. 代码分析

### 5.1 数据结构

先从struct device开始吧！该结构中有一个名称为“devres_head”的链表头，用于保存该设备申请的所有资源，如下：

```
1: struct device {
2:         ...
3:         spinlock_t              devres_lock;
4:         struct list_head        devres_head;
5:         ...
6: }
7:
```

那资源的数据结构呢？在“drivers/base/devres.c”中，名称为struct devres，如下：

```
1: struct devres {
2:         struct devres_node              node;
3:         /* -- 3 pointers */
4:         unsigned long long              data[]; /* guarantee ull alignment */
5: };
```

咋一看非常简单，一个struct devres_node的变量node，一个零长度数组data，但其中有无穷奥妙，让我们继续分析。

node用于将devres组织起来，方便插入到device结构的devres_head链表中，因此一定也有一个list_head（见下面的entry）。另外，资源的存在形式到底是什么，device resource management并不知情，因此需要上层模块提供一个release的回调函数，用于release资源，如下：

```
1: struct devres_node {
2:         struct list_head                entry;
3:         dr_release_t                    release;
4: #ifdef CONFIG_DEBUG_DEVRES
5:         const char                      *name;
6:         size_t                          size;
7: #endif
8: };
```

抛开用于debug的变量不说，也很简单，一个entry list_head，一个release回调函数。看不出怎么抽象资源啊！别急，奥妙都在data这个零长度数组上面呢。

注1：不知道您是否注意到，devres有关的数据结构，是在devres.c中定义的（是C文件哦！）。换句话说，是对其它模块透明的。这真是优雅的设计（尽量屏蔽细节）！

### 5.2 一个无关话题：零长度数组

零长度数组的英文原名为Arrays of Length Zero，是GNU C的规范，主要用途是用来作为结构体的最后一个成员，然后用它来访问此结构体对象之后的一段内存（通常是动态分配的内存）。什么意思呢?

以struct devres为例，node变量的长度为3个指针的长度，而struct devres的长度也是3个指针的长度。而data只是一个标记，当有人分配了大于3个指针长度的空间并把它转换为struct devres类型的变量后，我们就可以通过data来访问多出来的memory。也就是说，有了零长度数组data，struct devres结构的长度可以不定，完全依赖于你分配的空间的大小。有什么用呢?

以本文的应用场景为例，多出来的、可通过data访问的空间，正是具体的device resource所占的空间。资源的类型不同，占用的空间的多少也不同，但devres模块的主要功能又是释放资源所占的资源。这是就是零长度数组的功能之一，因为整个memory空间是连续的，因此可以通过释devres指针，释放所有的空间，包括data所指的那片不定长度的、具体资源所用的空间。

零长度数组（data[0]），在不同的C版本中，有不同的实现方案，包括1长度数组（data[1]）和不定长度数组（data[]，本文所描述就是这一种），具体可参考GCC的规范：

https://gcc.gnu.org/onlinedocs/gcc/Zero-Length.html

### 5.3 向上层framework提供的接口：devres_alloc/devres_free、devres_add/devres_remove

先看一个使用device resource management的例子（IRQ模块）：

```
1: /* include/linux/interrupt.h */
2: static inline int __must_check
3: devm_request_irq(struct device *dev, unsigned int irq, irq_handler_t handler,
4:                  unsigned long irqflags, const char *devname, void *dev_id)
5: {
6:         return devm_request_threaded_irq(dev, irq, handler, NULL, irqflags,
7:                                          devname, dev_id);
8: }
9:
```

前面我们提过，上层的IRQ framework，会提供两个和request_irq/free_irq基本兼容的接口，这两个接口的实现非常简单，就是在原有的实现之上，封装一层devres的操作，如要包括：

1）一个自定义的数据结构（struct irq_devres），用于保存和resource有关的信息（对中断来说，就是IRQ num），如下：

```
/*
 * Device resource management aware IRQ request/free implementation.
 */
struct irq_devres {
        unsigned int irq;
        void *dev_id;
};
```

2）一个用于release resource的回调函数（这里的release，和memory无关，例如free IRQ），如下：

```
static void devm_irq_release(struct device *dev, void *res)
{
        struct irq_devres *this = res;

        free_irq(this->irq, this->dev_id);
}
```

> 因为回调函数是由devres模块调用的，由它的参数可知，struct irq_devres变量就是实际的“资源”，但对devres而言，它并不知道该资源的实际形态，因而是void类型指针。也只有这样，devres模块才可以统一的处理所有类型的资源。

3）以回调函数、resource的size为参数，调用devres_alloc接口，为resource分配空间。该接口的定义如下：

```
void * devres_alloc(dr_release_t release, size_t size, gfp_t gfp)
{
        struct devres *dr;

        dr = alloc_dr(release, size, gfp);
        if (unlikely(!dr))
                return NULL;
        return dr->data;
}
```

调用alloc_dr，分配一个struct devres类型的变量，并返回其中的data指针（5.2小节讲过了，data变量实际上是资源的代表）。alloc_dr的定义如下：

```
static __always_inline struct devres * alloc_dr(dr_release_t release,
                                                 size_t size, gfp_t gfp)
{
        size_t tot_size = sizeof(struct devres) + size;
        struct devres *dr;

        dr = kmalloc_track_caller(tot_size, gfp);
        if (unlikely(!dr))
                return NULL;

        memset(dr, 0, tot_size);
        INIT_LIST_HEAD(&dr->node.entry);
        dr->node.release = release;
```

看第一句就可以了，在资源size之前，加一个struct devres的size，就是total分配的空间。除去struct devres的，就是资源的（由data指针访问）。之后是初始化struct devres变量的node。

4）调用原来的中断注册接口（这里是request_threaded_irq），注册中断。该步骤和device resource management无关。

5）注册成功后，以设备指针（dev）和资源指针（dr）为参数，调用devres_add，将资源添加到设备的资源链表头（devres_head）中，该接口定义如下：

```
void devres_add(struct device *dev, void *res)
{
        struct devres *dr = container_of(res, struct devres, data);
        unsigned long flags;

        spin_lock_irqsave(&dev->devres_lock, flags);
        add_dr(dev, &dr->node);
        spin_unlock_irqrestore(&dev->devres_lock, flags);
}
```

从资源指针中，取出完整的struct devres指针，调用add_dr接口。add_dr也很简单，把struct devres指针挂到设备的devres_head中即可：

```
1: static void add_dr(struct device *dev, struct devres_node *node)
2: {
3:         devres_log(dev, node, "ADD");
4:         BUG_ON(!list_empty(&node->entry));
5:         list_add_tail(&node->entry, &dev->devres_head);
6: }
```

6）如果失败，可以通过devres_free接口释放资源占用的空间，devm_free_irq接口中，会调用devres_destroy接口，将devres从devres_head中移除，并释放资源。这里就不详细描述了。

#### 5.4 向设备模型提供的接口：devres_release_all

这里是重点，用于自动释放资源。

先回忆一下设备模型中probe的流程（可参考“Linux设备模型(5)_device和device driver”），devres_release_all接口被调用的时机有两个：

1）probe失败时，调用过程为（就不详细的贴代码了）：

__driver_attach/__device_attach-->driver_probe_device—>really_probe，really_probe调用driver或者bus的probe接口，如果失败（返回值非零，可参考本文开头的例子），则会调用devres_release_all。

2）deriver dettach时（就是driver remove时）

driver_detach/bus_remove_device-->__device_release_driver-->devres_release_all

devres_release_all的实现如下：

```
1: int devres_release_all(struct device *dev)
2: {
3:         unsigned long flags;
4:
5:         /* Looks like an uninitialized device structure */
6:         if (WARN_ON(dev->devres_head.next == NULL))
7:                 return -ENODEV;
8:         spin_lock_irqsave(&dev->devres_lock, flags);
9:         return release_nodes(dev, dev->devres_head.next, &dev->devres_head,
10:                              flags);
11: }
```

以设备指针为参数，直接调用release_nodes：

```
1: static int release_nodes(struct device *dev, struct list_head *first,
2:                          struct list_head *end, unsigned long flags)
3:         __releases(&dev->devres_lock)
4: {
5:         LIST_HEAD(todo);
6:         int cnt;
7:         struct devres *dr, *tmp;
8:
9:         cnt = remove_nodes(dev, first, end, &todo);
10:
11:         spin_unlock_irqrestore(&dev->devres_lock, flags);
12:
13:         /* Release.  Note that both devres and devres_group are
```

release_nodes会先调用remove_nodes，将设备所有的struct devres指针从设备的devres_head中移除。然后，调用所有资源的release回调函数（如5.3小节描述的devm_irq_release），回调函数会回收具体的资源（如free_irq）。最后，调用free，释放devres以及资源所占的空间。

*原创文章，转发请注明出处。蜗窝科技，www.wowotech.net。*

**标签: Linux 设备模型 设备资源管理 devres**


« Linux电源管理(11)_Runtime PM之功能描述 | Linux kernel中断子系统之（五）：驱动申请中断API»

**评论：**

**寂寞沙洲冷**
2016-12-11 23:54

从上周发现这个网站到周末看了好几篇文章，觉得楼主实在是太伟大了，能把Linux的内核部分讲得实在是如此详细又易懂还能认真回复每一个读者提出的问题，在边看边学习的过程中不知不觉想眼泪留下来。我是一个嵌入式linux新手，刚毕业入职不久，之前没接触过Linux，然后上司突然给一个任务是要实现一个芯片的I2C总线驱动（因为好像总线驱动都是公司提供的？）并用一个温度传感器验证它是否能正常工作（也就是还要写一个设备驱动+测试Case），当时简直一头雾水，还照着很多S3C2440的书为例子来看那些驱动分析然而并没有什么卵用（其实本质是因为不同芯片采用的I2C的IP设计不同导致思路不同），问团队小组长他一边敷衍我说I2C总线驱动很简单只需要Probe一下什么的就行了，一遍又不对为什么要Probe说不出个所以然（也有可能是不愿意说，所以我现在怀疑他压根没做过I2C的驱动，可能就是拿做过的别的驱动过来硬套I2C了），直到看到博主的文章才知道为什么在probe里面申请clk，ioremap，申请IRQ和memory等等……想起之前工作几个月走过的弯路，不禁感慨万千要是早一点发现博主的博客，就不会走那么多弯路和忍受委屈了！感谢楼主的付出，我会继续努力学习的！

Ps.又偷偷吐槽一下I2C驱动任务完成了领导又来了一个任务是GPIO驱动，然而在高版本内核源码中找到了模板驱动在低版本（3.0.8）内核里面并没有，然后我把GPIO的驱动（drivers\gpio目录下）复制到低版本中经过source insight同步后发现缺少一些必要的库函数，跟IRQ相关的，然后我问小组长说怎么办，他说你不能改动kernel你只能改动你的驱动代码，然后我和他说有一些东西必须要用又没有到怎么办（比如irq_data结构体中有hwirq这个成员，但是低版本没有；高版本有irq_map机制可以把GPIO中断集中映射只使用芯片的一个中断号但是低版本的话是为每一个能产生GPIO中断的端口都分配中断号）……他说那你就自己想办法绕开。。。可是博主这东西真的能绕开吗？难不成我在自己的驱动里面写一个函数自行完成中断映射？T.T

回复

**wowo**
2016-12-12 10:23

@寂寞沙洲冷：我觉得啊，想GPIO、I2C这种驱动的porting，不要照搬原来的代码，把framework的设计思路理解了之后，自己重写一份（当然，硬件操作部分的代码可以完全复用，拿来就是了），就不会出现你这里所说的问题。而且经过这样几个驱动之后，以后做其他的，就不会有障碍了。
（PS：善待自己最先接触的几个任务，这是良机！不要急，不要快……）

回复

**寂寞沙洲冷**
2016-12-12 14:36

@wowo：感谢wowo大神的指点，我也觉得这个不能急不能快，可是BOSS催得着急呀（我也不知道他们为啥这么着急）所以我都是下班回家之后继续深入的看。然后想咨询一下就是您说的framework涉及思路指的是啥，比如I2C的话，指的是总线驱动代码中probe的那一套吗（比如probe中ioremap，申请irq等等但是不包括实现I2C的读写函数等等和硬件相关的操作）？

回复

**wowo**
2016-12-12 14:46

@寂寞沙洲冷：boss当然急（哈哈），差不多就是这些吧。

回复

**寂寞沙洲冷**
2016-12-12 16:14

@wowo：感谢WOWO，我会跟着您的微博好好学习哒

回复

**hony**
2017-01-14 22:56

@寂寞沙洲冷：wowo有微博？

回复

**wowo**
2017-01-16 09:07

@hony：没有微博啊，前面的兄弟估计是说错了，呵呵。

**jiang**
2017-06-16 10:35

@寂寞沙洲冷：总线驱动是控制器驱动 这种东西一般都是芯片厂商提供写好的 例如三星 高通等 跟着soc一起的 里面实现了usb或i2c协议等等 自己写难度很大 你们是芯片原厂？你们老大居然把这种活交给新手？

回复

**wowo**
2017-06-19 08:49

@jiang：呵呵，在原厂这样的事情很正常……

回复

**wanyafe**
2016-06-08 13:03

楼主是否可以补充一下关于devres_group的内容？

回复

**wowo**
2016-06-08 14:01

@wanyafe：devres位于设备模型比较底层的地方，driver开发者基本上不直接使用，因此本文也就是简单介绍一下。
对于Devres group，你可以参考“Documentation/driver-model/devres.txt”，简单的说，它的目的就是把很多的devres打包。
但具体怎么使用呢？kernel中有关的应用场景不多，其中一个场景是：
分配并获取多个资源的时候，如果中间某一个资源获取失败，则需要在错误返回的地方一个一个的把之前成功的统统释放，比较麻烦，使用Devres group则可以解决这样的问题。
先调用devres_open_group打开一个group，然后获取一个个的devres。如果某一个devres获取失败，可以直接调用devres_release_group，release所有成功的资源，就不用一个一个处理了。

回复

**sibulini**

2017-05-11 16:08

@wowo：感觉要学好linux，英语特别重要，好像大牛都要看Documentation文档

回复

**wowo**

2017-05-12 08:34

@sibulini：英语document就是要多看，看多了就自然了 ~

回复

---

**GhostaR**

2016-05-18 09:49

博主文章确实写得很好，从中学到不少，感谢博主的无私分享！

回复

**wowo**

2016-05-19 09:35

@GhostaR：多谢鼓励，大家一起学习~

回复

---

**madang**

2016-02-01 11:59

问一个问题：devm_ioremap_resource 接口和 devm_ioremap的区别。
看了一下代码，为一个区别就是一个接口devm_ioremap_resource（）接口会调用
devm_request_mem_region（）函数来检查 地址重复的问题。
但是看到内核里面 确实也有驱动调用devm_ioremap接口来做ioremap，
所以，想问一下，什么情况下可以调用 devm_ioremap ?

回复

---

**linuxer2010**

2015-08-02 13:06

一口气看完了驱动模型系列，收获颇丰，谢谢了。

回复

---

**Barry**

2015-07-07 17:48

hi，您好。看DT相关的资料找到了您的博客。感觉很好，国内好像很少如此深入分析驱动和内核的博客或文章。我相信不是国内确实大牛，只是有些可能觉得时间不够，有些觉得自己知道就行何苦分享给他人。我代表所有同学感谢博主的分享精神，也希望有更多的人加入这个行动中来。现在我看了一部分文章了，感觉写得好，但由于经验问题没能及时领会。希望此网站将来也不关闭，我已收藏，将持续关注博主，相信其他的同学也如我一般。最后提个建设性意见：就是希望网站能加入 “快速返回顶部”的功能。

回复

**linuxer**

2015-07-07 19:32

@Barry：多谢鼓励！我们会继续的，分享很快乐

回复

**wowo**

2015-07-08 09:08

@Barry：谢谢您的意见，有空的话我加一个。

回复

**tcutee**

2015-09-07 10:00

@Barry：很多机器还是用的老的 Linux内核，机器比较稳定，虽然 DT推出了好几年了，内核都奔四了，多数还是采用 2.6 原始方式

回复

---

**twenty**

2015-04-02 18:33

博主开头这句："之所以用“瞄”，因此它很简单，几乎不需要花费心思就能理解。" 令在下深受打击…… 作为初学者，一口气看完了设备模型系列文章，觉得这篇是最难理解.
ps: wowo能不能给想从事linux驱动开发的菜鸟(me)些学习建议，或者说学习方法？ 本人刚毕业，大学学过c语言，模电、数电、单片机等课程，但都没有学精……（大神见笑了） 现在想从新开始，立志搞linux底层驱动开发。
另外，我也看过一些这方面书、文档、教程，但都感觉看过不久就忘，不能真正吸收。 依然是个linux驱动开发门外汉，我想改变这种学习方法！ 但又不知如何实践？苦于无人指点，于是迷失在了linux茫茫代码中，实在痛苦！ 希望大神闲暇时能不吝赐教！！ 跪谢了 orz

回复

**wowo**

2015-04-02 21:49

@twenty：抱歉！开头那句话，我表达的不清晰，所谓的“好理解”，是指devm_xxx接口和之前的接口的区别，以及使用这些接口的好处。
当然，device resource意义及实现，确实很难理解。但这又如何呢？对一个驱动工程师来说，为什么一定需要理解device resource呢？只要会使用这些devm_xxx接口，就足够了啊！
所以不用纠结，这也正是我给你的建议：

学习的过程是一个螺旋式上升的过程，同样的一个事物，你今天看，和你5年后看，肯定是不一样的。这不是能力问题，而是经验问题。因此看不懂又如何呢？先放放，总有一天回过头来就懂了。
立志搞linux底层，就找一份相关的工作，做就行了。复杂驱动不会，就先做简单的。简单的也不会？那就从别处抄嘛！总有一天什么都会了。
因此重要的不是看了多少文档、书籍，而是有没有行动起来，去做就是了！

回复

**twenty**
2015-04-03 08:56

@wowo：感谢回复！！ 我会改变学习方法的，实践，实践，do it,do it,do it.....

回复

**Issues**
2015-02-05 08:54

谢谢楼主的解答，可这些头文件，在安装内核树之前应该已经存在了啊(/usr/src/ 下本来就有内核的头文件)。内核树构建的的是source文件和编译后的vmlinux.o镜像文件。如果编译新内核时这些文件没用，哪这些文件是干什么的。而且只有头文件，编译新model的时候引用的内核符号在哪？？

回复

**wowo**
2015-02-05 09:53

@Issues：如果你目标kernel和主机kernel是相同版本的话，确实可以使用主机的header……

回复

**Issues**
2015-02-05 16:28

@wowo：按楼主的意思，如果我编译和的module是和主机kernel相同版本的话，那不用构建内核源码树应该也能编译成功。可这样是会失败的啊，编译时会提示要求安装源码树

回复

**wowo**
2015-02-05 21:14

@Issues：先确认是否有header文件。
再确认是否有模块编译目录：/lib/modules/$(uanem -r)/build
以及header的搜索路径是否正确。
等等。

回复

**Issues**
2015-02-07 08:12

@wowo：最后还是在ldd 3rd上找到答案了，2.6内核的模块要和内核源码树中的目标文件连接，这样可以得到更加健壮的模块加载器，因此就需要这些目标文件存在于内核目录树中。以前版本的内核是只需要一套头文件的。第一遍看书的时候根本没理解是啥意思。现在发现，这句话是解释的最明白的。

谢谢楼主的耐心解答。网站的内容非常不错，有很多我需要学习的文章。希望以后窝窝可以变成linux爱好者的乐园。

回复

**wowo**
2015-02-07 21:46

@Issues：非常感谢您的分享，以后多交流。

回复

**hony**
2017-01-14 23:10

@Issues：没有吧，直接到/lib/modules/2.6.32-642.13.1.el6.x86_64/build，直接make M=<模块源码路径>，我这边是可以编译的。

而没有安装内核源码。

回复

**wowo**
2017-01-16 09:15

@hony：“/lib/modules/2.6.32-642.13.1.el6.x86_64”，这里面应该有所需的东西了。

**Issues**
2015-01-28 15:41

刚转入linux驱动开发，从楼主的系列文章学到了很多。不过鉴于个人还是内核菜鸟，有几个困扰多时，网上也没有找到确切答案的问题，想请楼主百忙之中拨冗解答一下。

1.在已安装linux系统的pc上进行驱动开发（面向本机，如给嵌入式开发，还比较好理解），为何还要安装内核源码树。已安装的linux系统没有内核么，两者有何区别。个人理解，是不是因为操作系统的内核是一个完整的可执行程序，因此，新建module是无法链接其中的目标文件，因此需要安装源码树。源码树安装并make后，形成目标文件，是不是只生成了一个vmlinux.o(ubuntu)，之后新建的module文件就可以通过/linux/*.h链接到此目标文件完成编译。

2.新建的moudle编译完成后，insmod加载模块，新module是加载到当前的linux操作系统中，还是新安装的内核源码树上。

希望楼主能详细解说一下，非常感谢。

回复

**wowo**
2015-01-29 10:37

@Issues：我想你问的是有关"编译linux kernel module"的问题。
首先，编译kernel module（无论target machine是Host还是嵌入式本机），kernel source tree（从kernel.org下载的）不是必须的。而 linux kernel header才是必须的。
那么，什么是 linux kernel header呢？打个比方，我们使用一个第三方库（静态库或者动态库）的时候，除了库文件，往往还需要一些头文件，以便知道这些库文件中符号（其实这些也不是必须的，如果我们知道所有的符号的话，但这对linux kernel是不可能的）。
module编译也类似，我们编写驱动，需要用到很多很多kernel已有的机制（如各种framework，其它driver等等，这些表现出来的就是一个一个符号，位于各个模块的头文件中)。这也是为什么需要linux kernel header。
最后，没有安装在"内核源码树"上这一说法。module文件就像一个可执行文件（windows下的EXE，Linux的EFL，等等），需要由操作系统，加载到内存，解析并执行之。

回复

**perr**
2014-09-29 10:59

linuxer辛苦了,讲的真好.device tree的设备节点所占用的资源如何反映到struct device中?

回复

**linuxer**
2014-09-29 12:34

@perr：你没有发现吗？这个网站主要是由2个人维护的，这份文档是蜗蜗同学写的，呵呵～～～

回复

**forion**
2014-09-29 12:35

@linuxer：嘿嘿，最开始我以为是有人格分裂的一个人。

回复

**linuxer**
2014-09-29 12:49

@forion：的确，我和蜗窝性格还是有些不一样的，我比较温和，他比较犀利，如果是一个人的话的确是有人格分裂的嫌疑，呵呵～～～

工作不忙的时候别忘了也写写文章哦～～～蜗窝是大家的蜗窝，呵呵～～～

回复

**forion**
2014-09-30 14:01

@linuxer：还在积累阶段，怕写出来误导大家，另外最近项目太忙，天天加班到很晚，都没精力看文档了。

回复

**perr**
2014-09-29 12:38

@linuxer：挺有默契的嘛.不会写同一个地方

回复

**linuxer**
2014-09-29 12:46

@perr：我们曾经在一个公司，有相同的理念，对linux kernel有相同的热情，自然也就很容易成为朋友，一起做事情。
我也是从蜗蜗那里学到很多，我很少见到有一个年轻人能象蜗蜗那么有大局观。
我们经常通话，确定短期内的方向，当然不会写重复的，呵呵～～～

回复

**perr**
2014-09-29 12:52

@linuxer：希望有时间能把clocksource和clk给讲讲

回复

**linuxer**
2014-09-29 12:59

@perr：clocksource是linux时间子系统的一部分，我搞完中断子系统这部分，下一个专题就是时间子系统了。
clk是和clock distribution相关的内容，属于内核common clock framework，可以归入电源管理子系统，蜗蜗同学应该会写这部分内容的

回复

**蜗蜗**
2014-09-29 22:09

@linuxer：linuxer过奖了，实在不敢当的，需要向你学习的还很多，以前在学，现在和将来也在。

回复

**WL**
2017-01-08 19:43

@linuxer：能够找到一个志同道合的朋友，值得珍惜！

回复

**perr**
2014-09-29 12:51

@perr：希望有时间能把clocksource和clk给讲讲

回复

**蜗蜗**
2014-09-29 22:35

@perr：device tree的设备节点，准确的说，是用来描述资源，例如device的I/O memory、IRQ number等。系统启动后，device tree会把这些节点统统挂到struct device的of_node指针上。
另外，对于一些标准资源（IO、IRQ、DMA、GPIO、等等），device tree会帮助解析（需要和对应的framework交互），并以platform resource（struct resource）的形式保存下来，因此driver可以通过platform_get_xxx系列接口获取。
而对于一些非标准节点（如自定义的配置参数），则需要调用of_xxx系列接口，自行解析。
要完全分析清楚，估计也得一篇不短的文章啊，Linuxer写了三篇DTS的文章，把DTS的原理分析的很透彻了。
如果能从这个角度，再写一篇，阐述从Linux驱动开发者的角度，怎么使用，就perfect了。哈哈。

回复

**wowo**

2014-09-30 08:39

@蜗蜗：补充一点，这里的解析，只是解析描述，具体的资源，还是需要使用devm_xxx接口获取的。

回复

**发表评论：**

昵称

邮件地址 (选填)

个人主页 (选填)

发表评论