# Linux驱动

## 前言：

字符设备：字节读取

块设备：存储设备

网络设备：USB WIFI/SDIO WIFI

驱动：获取外设或传感器数据、控制外设

编写一个驱动、写一个简单测试应用程序APP（驱动和应用是完全分开的）

数据段、代码段、BSS段、堆、栈

用户态进入内核态：系统调用、异常和陷入

应用程序调用库API再调用系统调用

应用层：写open、write、read、close

驱动对应也写open、write、read、close

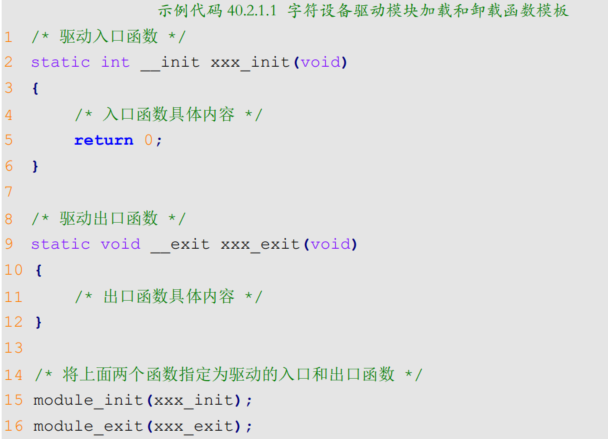
## 框架

Linux 内核文件 include/linux/fs.h 中有个叫做 file\_operations 的结构体，里面有open、read等函数

1. 驱动模块的加载和卸载

module\_init(xxx\_init); //注册模块加载函数

module\_exit(xxx\_exit); //注册模块卸载函数



加载模块：insmod和 modprobe drv.ko

1. 字符设备注册和注销

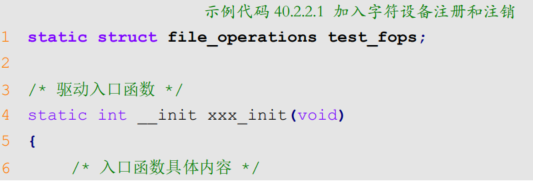
static inline int register\_chrdev(unsigned int major, const char \*name,const struct file\_operations \*fops)

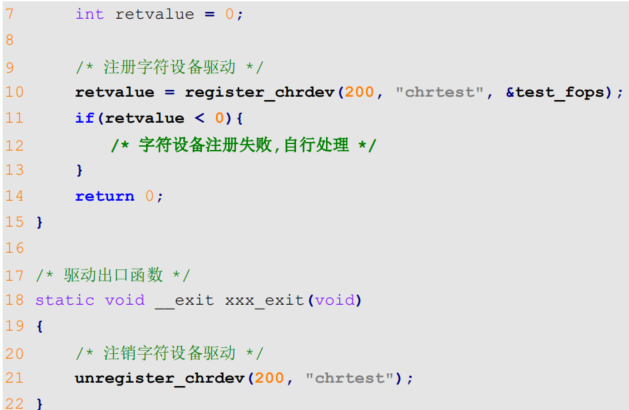
static inline void unregister\_chrdev(unsigned int major, const char \*name)

major：主设备号，Linux 下每个设备都有一个设备号，设备号分为主设备号和次设号两部分，关于设备号后面会详细讲解。

name：设备名字，指向一串字符串。

fops：结构体 file\_operations 类型指针，指向设备的操作函数集合变量。





“cat /proc/devices”可以查看当前已经被使用掉的设备号

1. 实现设备的具体操作函数

/\* 打开设备 \*/

static int chrtest\_open(struct inode \*inode, struct file \*filp)

{

/\* 用户实现具体功能 \*/

return 0;

}

/\* 从设备读取 \*/

static ssize\_t chrtest\_read(struct file \*filp, char \_\_user \*buf,

size\_t cnt, loff\_t \*offt)

{

/\* 用户实现具体功能 \*/

return 0;

}

/\* 向设备写数据 \*/

static ssize\_t chrtest\_write(struct file \*filp,const char \_\_user \*buf,

size\_t cnt, loff\_t \*offt)

{

/\* 用户实现具体功能 \*/

return 0;

}

/\* 关闭/释放设备 \*/

static int chrtest\_release(struct inode \*inode, struct file \*filp)

{

/\* 用户实现具体功能 \*/

return 0;

}

static struct file\_operations test\_fops = {

.owner = THIS\_MODULE,

.open = chrtest\_open,

.read = chrtest\_read,

.write = chrtest\_write,

.release = chrtest\_release,

};

/\* 驱动入口函数 \*/

static int \_\_init xxx\_init(void)

{

/\* 入口函数具体内容 \*/

int retvalue = 0;

/\* 注册字符设备驱动 \*/

retvalue = register\_chrdev(200, "chrtest", &test\_fops);

if(retvalue < 0){

/\* 字符设备注册失败,自行处理 \*/

}

return 0;

}

/\* 驱动出口函数 \*/

static void \_\_exit xxx\_exit(void)

{

/\* 注销字符设备驱动 \*/

unregister\_chrdev(200, "chrtest");

}

/\* 将上面两个函数指定为驱动的入口和出口函数 \*/

module\_init(xxx\_init);

module\_exit(xxx\_exit);

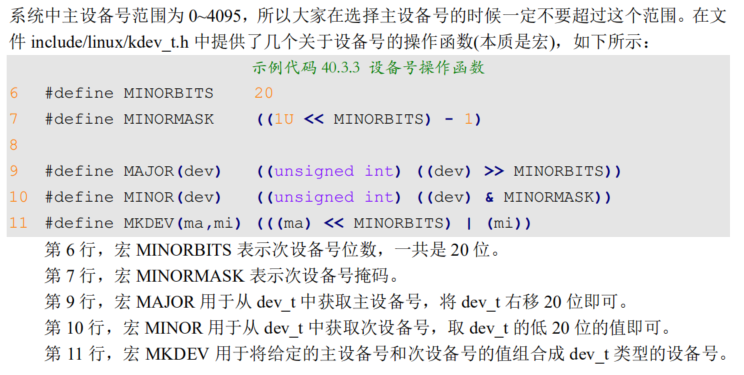
1. 添加 LICENSE 和作者信息

MODULE\_LICENSE() //添加模块 LICENSE 信息

MODULE\_AUTHOR() //添加模块作者信息

1. 设备号

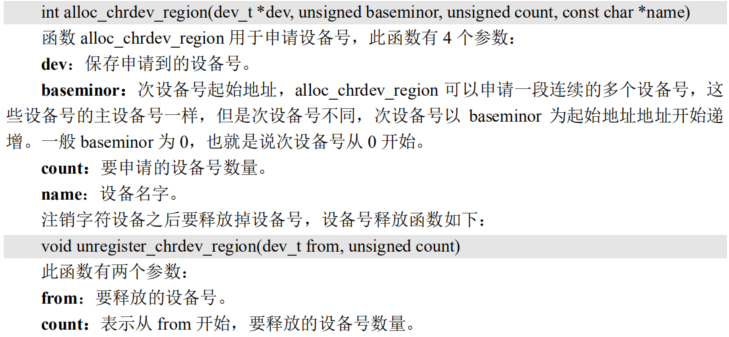
dev\_t是32位数据格式，高12位是主设备号，低20位是此设备号。主设备号范围为0~4095



设备号分配：

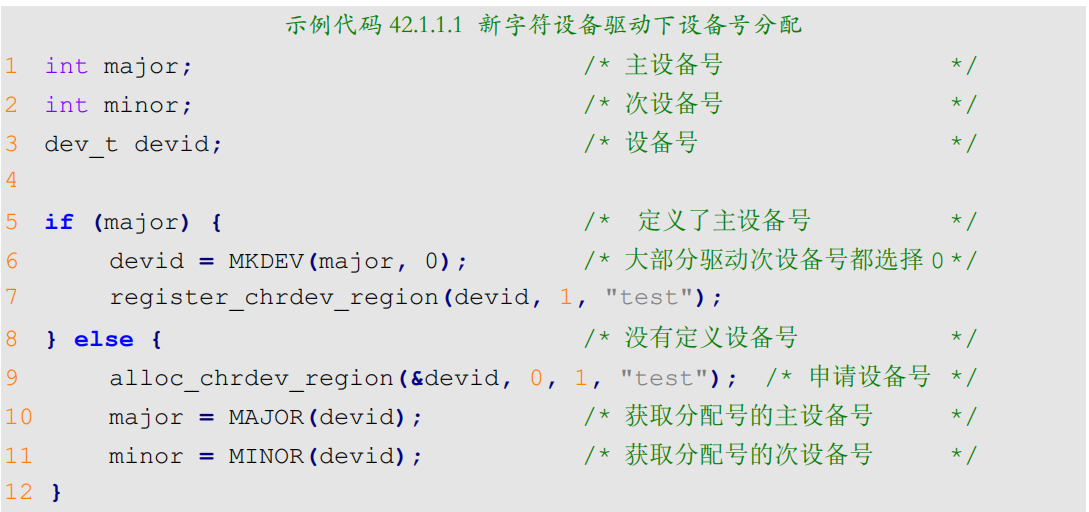
①静态分配

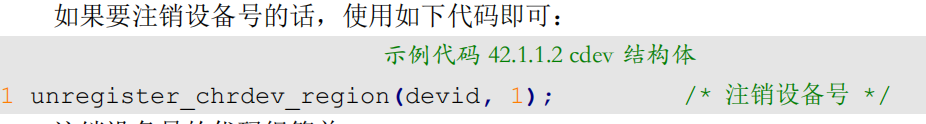
②动态分配



## 新字符设备驱动

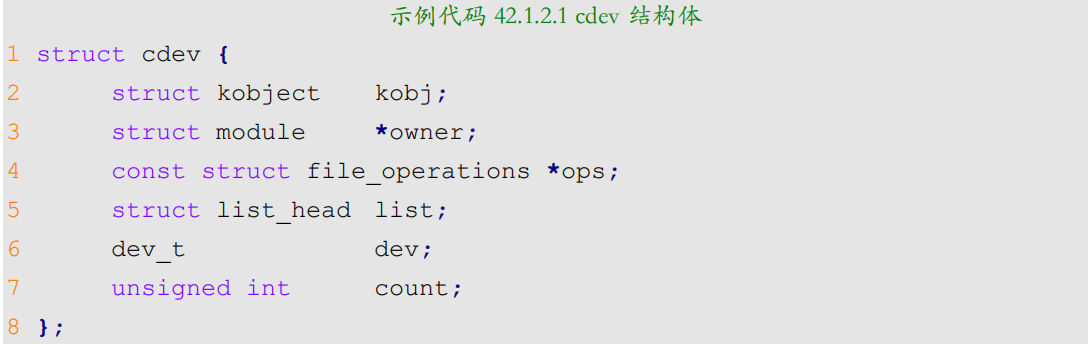
1. 设备号分配





1. 字符设备注册

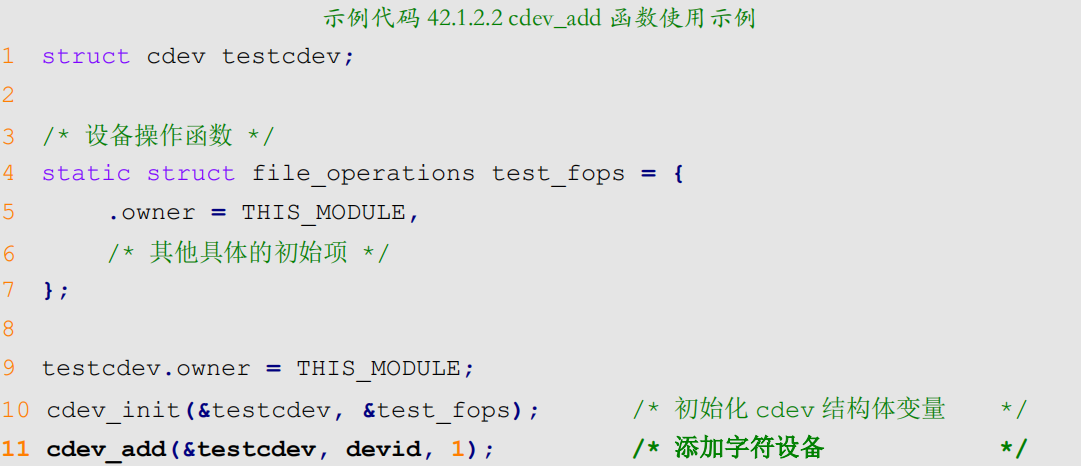
在 Linux 中使用 cdev 结构体表示一个字符设备，cdev 结构体在include/linux/cdev.h 文件中



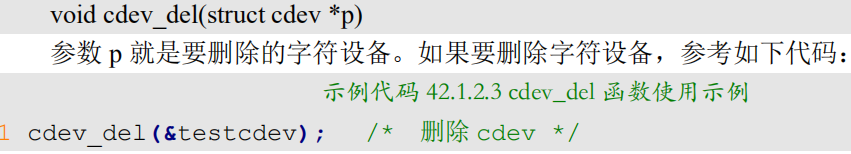
①定义字符设备：struct cdev test\_cdev;

②cdev\_init 函数初始化：void cdev\_init(struct cdev \*cdev, const struct file\_operations \*fops)  
③cdev\_add 函数：向Linux系统添加字符设备

int cdev\_add(struct cdev \*p, dev\_t dev, unsigned count)



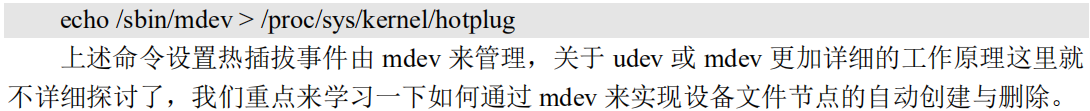
④cdev\_del 函数：卸载字符设备



1. 自动创建设备节点

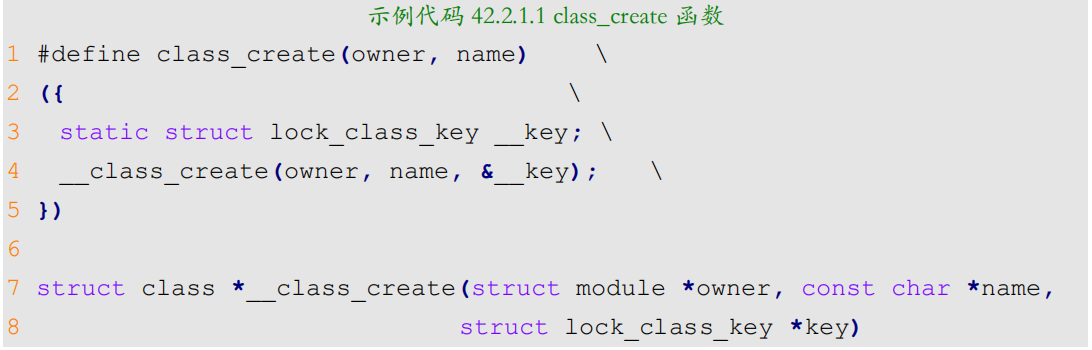
①设置

udev 是一个用户程序，在 Linux 下通过 udev 来实现设备文件的创建与删除，busybox 构建根文件系统的时候，busybox 会创建一个 udev 的简化版本—mdev，所以在嵌入式 Linux 中我们使用mdev 来实现设备节点文件的自动创建与删除，Linux 系统中的热插拔事件也由 mdev 管理，在/etc/init.d/rcS 文件中如下语句：



②创建和删除类

一般在 cdev\_add 函数后面添加自动创建设备节点相关代码。首先要创建一个 class 类，class 是个结构体，定义在文件include/linux/device.h 里面。

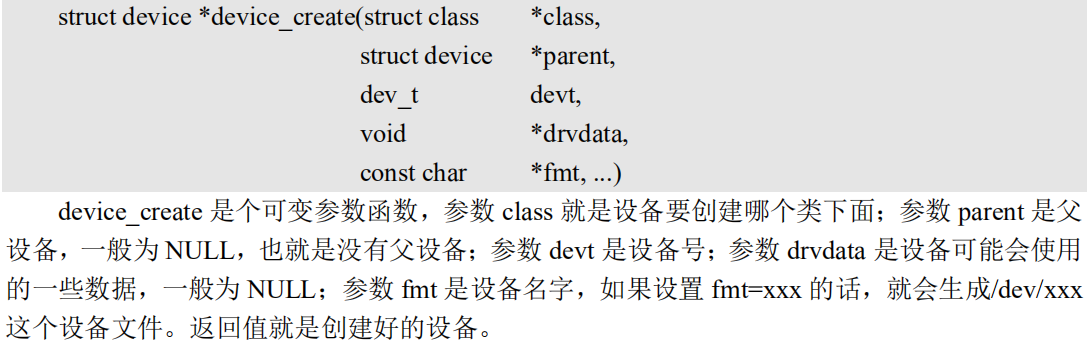


创建类函数：struct class \*class\_create (struct module \*owner, const char \*name)

删除类函数：void class\_destroy(struct class \*cls);

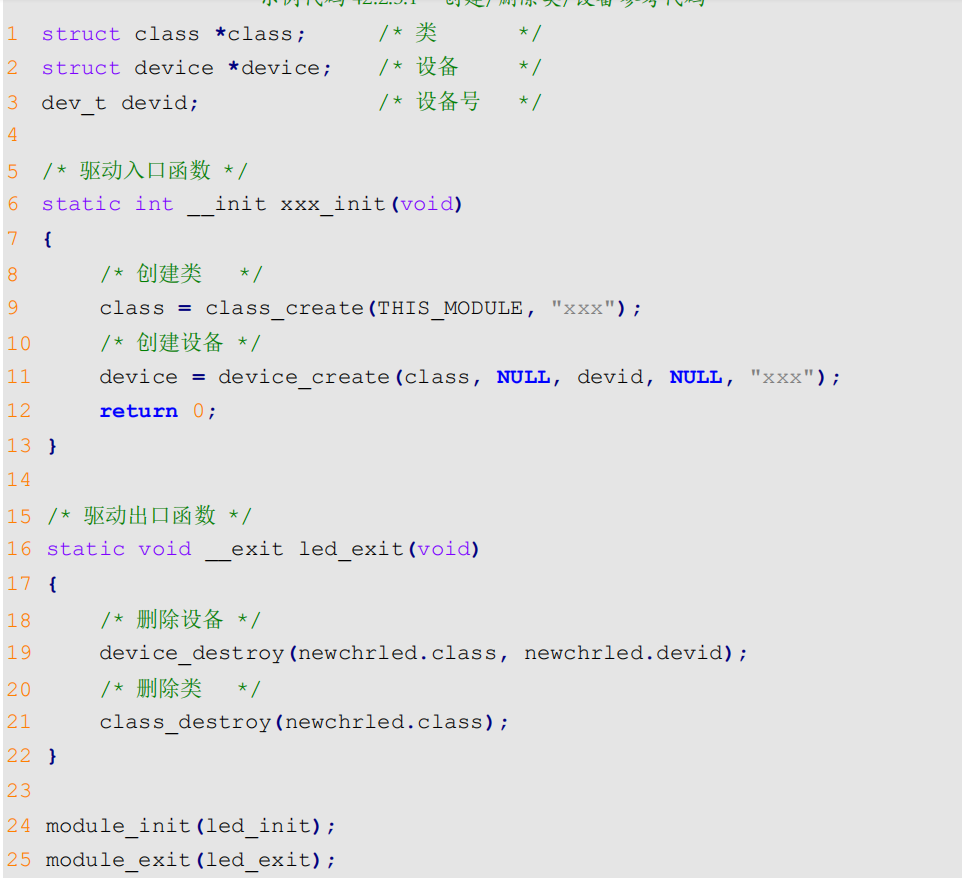
③创建设备

device\_create 函数：

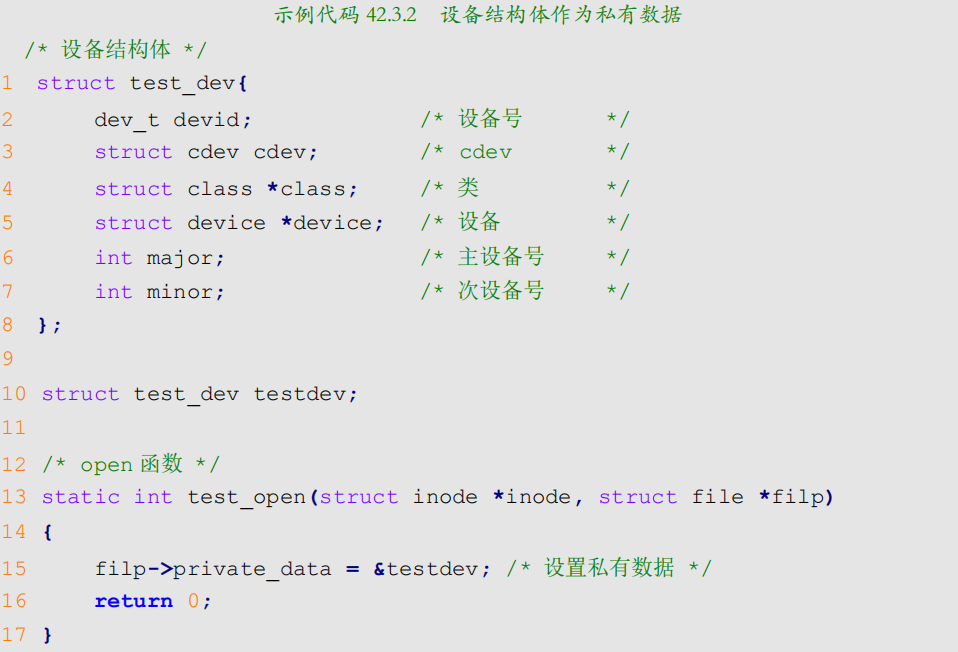


删除设备：device\_destroy--

void device\_destroy(struct class \*class, dev\_t devt)



1. 设置文件私有数据



## 设备树

简介：

DTS 文件采用树形结构描述板级设备，也就是开发板上的设备信息，比如CPU 数量、 内存基地址、IIC 接口上接了哪些设备、SPI 接口上接了哪些设备

在 3.x 版本(具体哪个版本笔者也无从考证)以前的 Linux 内核中 ARM 架构并没有采用设备树。在没有设备树的时候 Linux 是如何描述 ARM 架构中的板级信息呢？在 Linux 内核源码中大量的 arch/arm/mach-xxx 和 arch/arm/plat-xxx 文件夹

采用的设备树(Flattened Device Tree)，将这些描述板级硬件信息的内容都从 Linux 内中分离开来，用一个专属的文件格式来描述，这个专属的文件就叫做设备树，文件扩展名为.dts。

一个 SOC 可以作出很多不同的板子，这些不同的板子肯定是有共同的信息，将这些共同的信息提取出来作为一个通用的文件，其他的.dts 文件直接引用这个通用文件即可，这个通用文件就是.dtsi 文件，类似于 C 语言中的头文件。一般.dts 描述板级信息(也就是开发板上有哪些 IIC 设备、SPI 设备等)，.dtsi 描述 SOC 级信息(也就是 SOC 有几个 CPU、主频是多少、各个外设控制器信息等)。

DTS 是设备树源码文件，DTB 是将DTS 编译以后得到的二进制文件。将.c 文件编译为.o 需要用到 gcc 编译器，那么将.dts 编译为.dtb需要什么工具呢？需要用到 DTC 工具！DTC 工具源码在 Linux 内核的 scripts/dtc 目录下

每个板子都有一个对应的 DTS 文件，那么如何确定编译哪一个 DTS 文件呢？我们就以 I.MX6ULL 这款芯片对应的板子为例来看一下，打开 arch/arm/boot/dts/Makefile

1、DTS语法：关于设备树详 细 的 语 法 规 则 请 参 考 《 Devicetree SpecificationV0.2.pdf 》 和《Power\_ePAPR\_APPROVED\_v1.12.pdf》这两份文档，此两份文档已经放到了开发板光盘中，路 径 为 ： 4 、 参 考 资 料 ->Devicetree SpecificationV0.2.pdf、4、参 考 资 料 ->Power\_ePAPR\_APPROVED\_v1.12.pdf

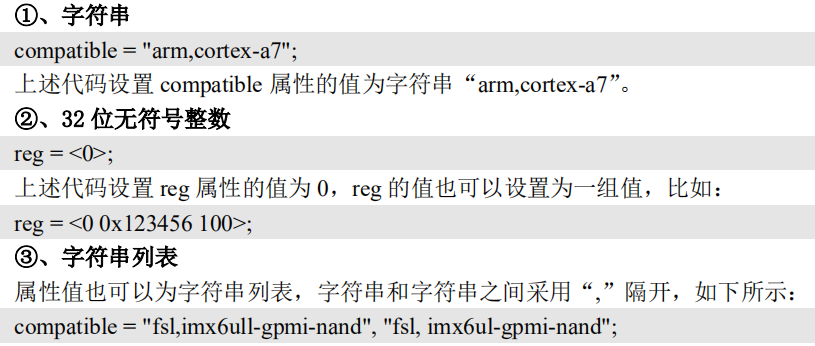
①dtsi 头文件：“#include”来引用.h、.dtsi 和.dts 文件

②设备节点

根结点

节点命名格式：node-name@unit-address或label: node-name@unit-address

数据形式：



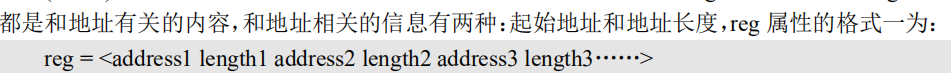
③标准熟悉

compatible 属性："manufacturer,model"，设备和驱动绑定

model 属性：设备模块信息，比如名字

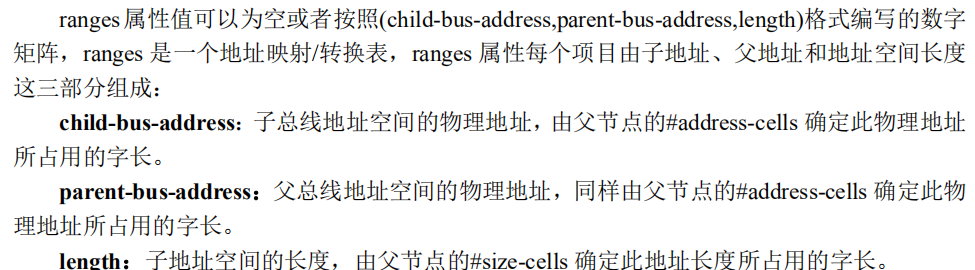
status 属性：设备的状态

#address-cells 和#size-cells 属性：无符号 32 位整形，#address-cells 属性值决定了子节点 reg 属性中地址信息所占用的字长(32 位)，#size-cells 属性值决定了子节点 reg 属性中长度信息所占的字节



reg属性：用于描述设备地址空间资源信息，一般都是某个外设的寄存器地址范围信息

Ranges属性：



Name属性

device\_type 属性（cpu和memory节点）

1. 设备匹配方法

①使用设备树之前：uboot 会向 Linux 内核传递一个叫做 machine id 的值，machine id也就是设备 ID，Linux内核都用MACHINE\_START和MACHINE\_END，arch/arm/include/asm/mach/arch.h

②使用设备树以后的设备匹配方法

当 Linux 内 核 引 入 设 备 树 以 后 就 不 再 使 用 MACHINE\_START 了 ， 而 是 换 为 了DT\_MACHINE\_START。DT\_MACHINE\_START 也定义在文件arch/arm/include/asm/mach/arch.h

machine\_desc 结构体中有个.dt\_compat 成员变量，此成员变量保存着本设备兼容属性，示例代码 43.3.4.5 中设置.dt\_compat = imx6ul\_dt\_compat，imx6ul\_dt\_compat 表里面有"fsl,imx6ul"和"fsl,imx6ull"这两个兼容值。



1. 节点追加或修改内容

imx6ull.dtsi 是设备树头文件，其他所有使用到 I.MX6ULL这颗 SOC 的板子都会引用 imx6ull.dtsi 这个文件，I.MX6U-ALPHA 开发板使用的设备树文件为 imx6ull-alientek-emmc.dts，因此我们需要在imx6ull-alientek-emmc.dts 文件中完成数据追加的内容

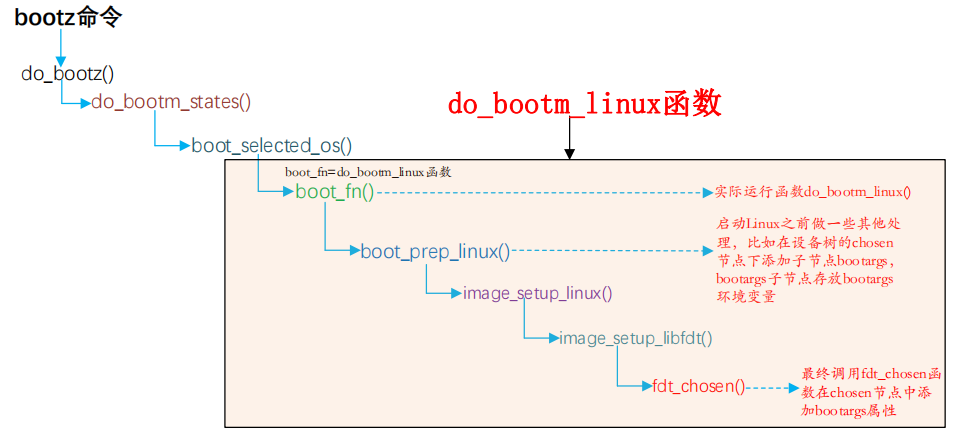
1. 设备树在系统中的体现

①根结点的所有属性和子节点：/proc/devicetree

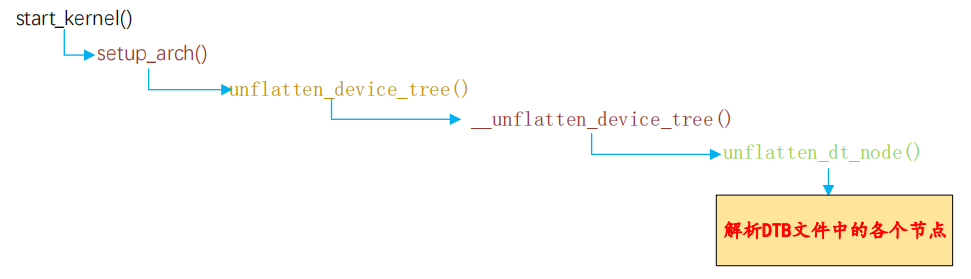
②根结点“/”各个属性：cat model/compatible

③根结点“/”各个子节点 cd ls

④特殊节点：aliases：定义别名 和 chosen：uboot向linux内核传递数据



1. Linux内核解析DTB文件



1. 设备树常用OF操作函数

Linux 内核给我们提供了一系列的函数来获取设备树中的节点或者属性信息，这一系列的函数都有一个统一的前缀“of\_”，所以在很多资料里面也被叫做 OF 函数。这些 OF 函数原型都定义在 include/linux/of.h 文件中。

查找节点：

①of\_find\_node\_by\_name：通过节点名字查找指定的节点

②of\_find\_node\_by\_type：通过 device\_type 属性查找指定的节点

③of\_find\_compatible\_node：根据 device\_type 和 compatible 这两个属性查找指定的节点

④of\_find\_matching\_node\_and\_match：通过 of\_device\_id 匹配表来查找指定的节点

⑤of\_find\_node\_by\_path：通过路径来查找指定的节点

查找父/子节点的OF函数：

①of\_get\_parent：获取指定节点的父节点

②of\_get\_next\_child：用迭代的查找子节点

提取属性值的OF函数：

①of\_find\_property 函数用于查找指定的属性

②of\_property\_count\_elems\_of\_size 函数用于获取属性中元素的数量

③of\_property\_read\_u32\_index 函数用于从属性中获取指定标号的 u32 类型数据值

④of\_property\_read\_u8\_array 函数

of\_property\_read\_u16\_array 函数

of\_property\_read\_u32\_array 函数

of\_property\_read\_u64\_array 函数

这 4 个函数分别是读取属性中 u8、u16、u32 和 u64 类型的数组数据

⑤of\_property\_read\_u8 函数

of\_property\_read\_u16 函数

of\_property\_read\_u32 函数

of\_property\_read\_u64 函数

有些属性只有一个整形值，这四个函数就是用于读取这种只有一个整形值的属性

⑥of\_property\_read\_string 函数用于读取属性中字符串值

⑦of\_n\_addr\_cells 函数用于获取#address-cells 属性值

⑧of\_size\_cells 函数用于获取#size-cells 属性值

其他常用OF函数：

①of\_device\_is\_compatible 函数用于查看节点的 compatible 属性是否有包含 compat 指定的字符串，也就是检查设备节点的兼容性

②of\_get\_address 函数用于获取地址相关属性

③of\_translate\_address 函数负责将从设备树读取到的地址转换为物理地址

④of\_address\_to\_resource 函数，此函数看名字像是从设备树里面提取资源值，但是本质上就是将 reg 属性值，然后将其转换为 resource 结构体类型

⑤of\_iomap 函数用于直接内存映射

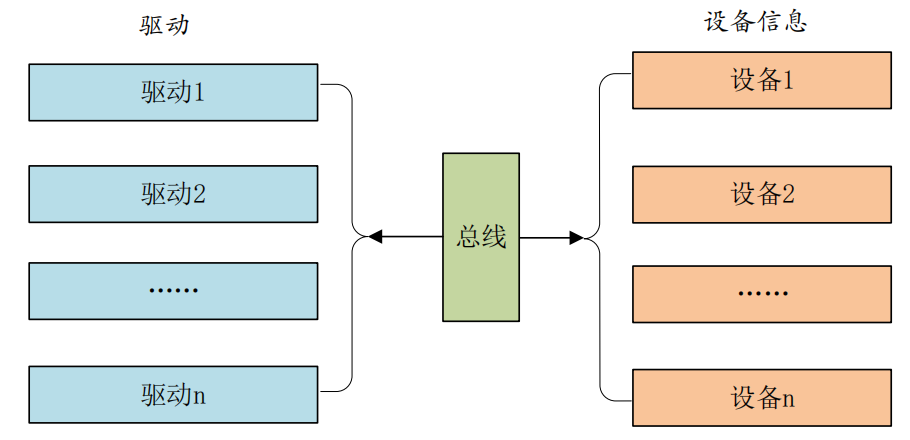
## Platform设备驱动设备

简介：

驱动的分隔，也就是将主机驱动和设备驱动分隔开来，主机控制器驱动已经由

半导体厂家编写好了，而设备驱动一般也由设备器件的厂家编写好了，我们只需要提供设备信息即可，比如 I2C 设备的话提供设备连接到了哪个 I2C 接口上，I2C 的速度是多少等等。相当于将设备信息从设备驱动中剥离开来，驱动使用标准方法去获取到设备信息(比如从设备树中获取到设备信息)，然后根据获取到的设备信息来初始化设备。

当我们向系统注册一个驱动的时候，总线就会在右侧的设备中查找，看看有没有与之匹配的设备，如果有的话就将两者联系起来。同样的，当向系统中注册一个设备的时候，总线就会在左侧的驱动中查找看有没有与之匹配的设备



Linux 提出了 platform 这个虚拟总线，相应的就有 platform\_driver 和 platform\_device。

1. platform总线

Linux系统内核使用bus\_type结构体表示总线，此结构体定义在文件include/linux/device.h

match 函数，完成设备和驱动之间匹配的，每一条总线都必须实现此函数

platform 总线是 bus\_type 的一个具体实例，定义在文件 drivers/base/platform.c



platform\_match 就是匹配函数，四种匹配方法：

①OF类型的匹配，就是设备树采用的匹配方式

②ACPI匹配

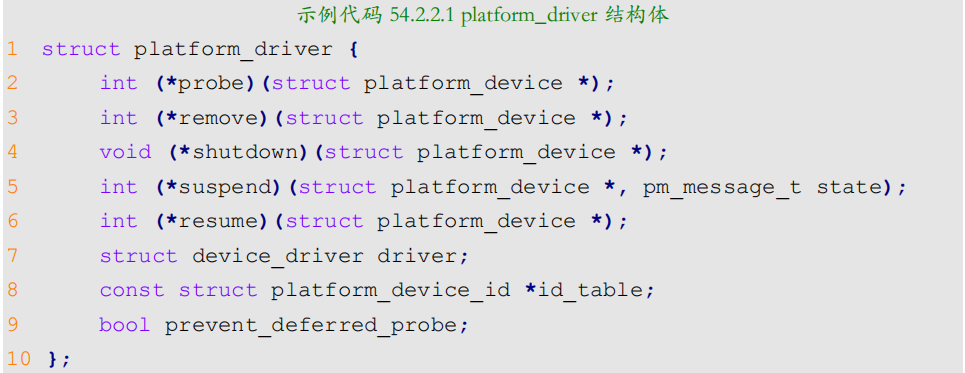
③id\_table匹配

④id\_table不存在直接比较驱动和设备name

1. Platform驱动

platform\_driver 结 构 体 表 示 platform 驱动，此结构体定义在文件

include/linux/platform\_device.h



probe 函数，当驱动与设备匹配成功以后 probe 函数就会执行，非常重要的函数！！

驱动流程：

①在编写 platform 驱动的时候，首先定义一个 platform\_driver 结构体变量，然后实现结构体中的各个成员变量，重点是实现匹配方法以及 probe 函数。当驱动和设备匹配成功以后 probe函数就会执行，具体的驱动程序在 probe 函数里面编写，比如字符设备驱动

②当我们定义并初始化好 platform\_driver 结构体变量以后，需要在驱动入口函数里面调用platform\_driver\_register 函数向 Linux 内核注册一个 platform 驱动

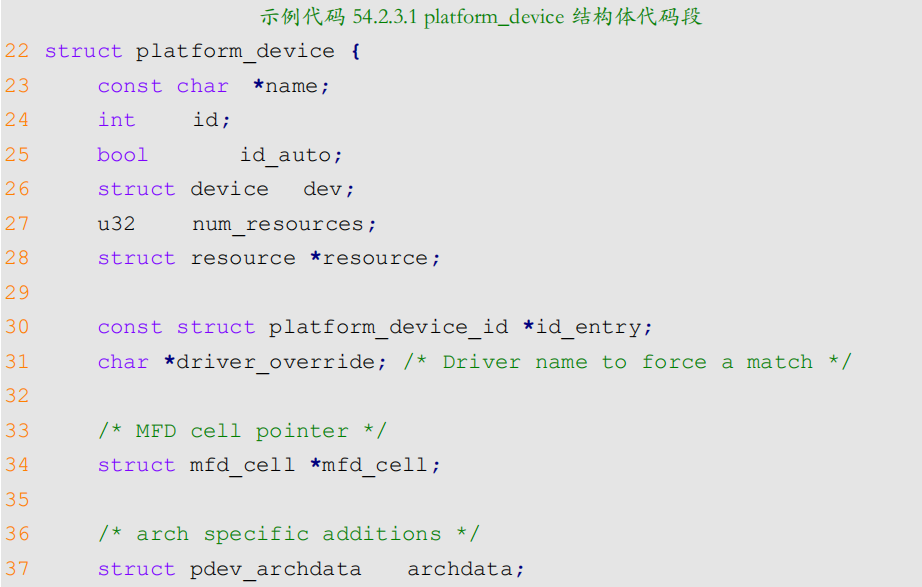
int platform\_driver\_register (struct platform\_driver \*driver)

卸载驱动：void platform\_driver\_unregister(struct platform\_driver \*drv)

1. Platform设备

platform\_device 这个结构体表示 platform 设备，这里我们要注意，如果内核支持设备树的话就不要再使用 platform\_device 来描述设备了，因为改用设备树去描述了。

一定要用 platform\_device 来描述设备信息的话也是可以的。platform\_device 结构体定义在文件include/linux/platform\_device.h 中



platform\_device\_register 函数将设备信息注册到 Linux 内核中：

int platform\_device\_register(struct platform\_device \*pdev)

通过 platform\_device\_unregister 函数注销掉相应的 platform设备：

void platform\_device\_unregister(struct platform\_device \*pdev)

1. 设备树下的platform驱动设备

①在设备树中创建设备节点：主要compatible属性

②编写platform驱动注意兼容属性

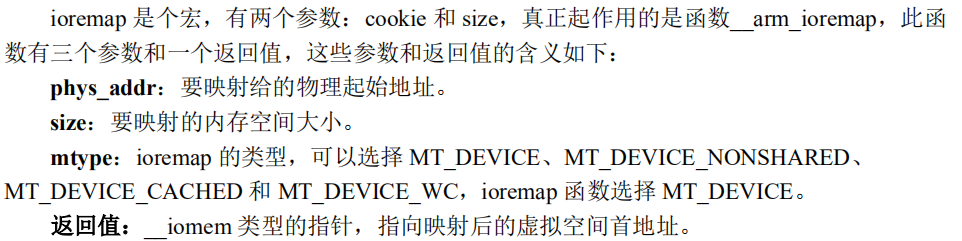
of\_match\_table

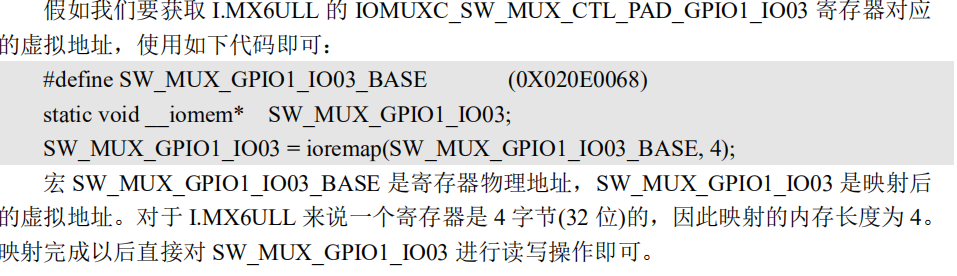
③编写platform驱动

## LED驱动

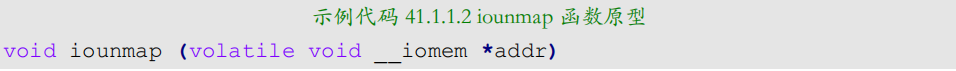
1. 内存映射

①ioremap 函数：获取指定物理地址空间对应的虚拟地址空间



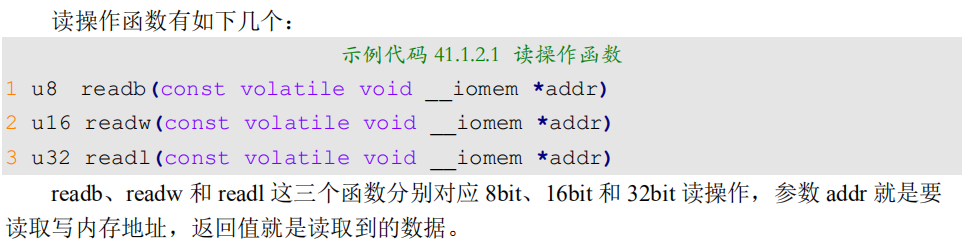


②卸载驱动的时候需要使用 iounmap 函数释放掉 ioremap 函数所做的映射



1. I/O内存访问函数

①读操作



②写操作

