hub流程

https://blog.csdn.net/smartvincent88/article/details/52512973

https://www.cnblogs.com/-glb/p/11568992.html

<https://blog.csdn.net/kunkliu/article/details/79407868>

<https://www.cnblogs.com/lxl-lennie/p/10184917.html>

<https://blog.csdn.net/w1107101310/article/details/79059155>

https://blog.csdn.net/cutter2002/article/details/69749825

https://www.cnblogs.com/LoTGu/p/5968042.html

<https://www.cnblogs.com/Ph-one/p/10938479.html>

https://www.cnblogs.com/cslunatic/p/3726053.html

# 概述

Lnx/ker./drivers/usb

atm/ class/ core/ gadget/ host/ image/ misc/ mon/ serial/ storage/ Kconfig Makefile README usb-skeleton.c

usb-skeleton.c 是一个简单的 USB driver 的框架，那么首先应该关注什么？那就是 Kconfig、Makefile、README

USB Core：Linux 内核开发人员们专门写了一些代码，负责实现一些核心的功能，为别的设 备驱动程序提供服务，比如申请内存，实现一些所有的设备都会需要的公共函数，并美其名曰 为“USB Core”。

Host，各主机控制器单独的代码移到host目录下，负责各种主机控制器的人来维护。主机控制器公共的代码任然保留在core目录下

USB gadget，配件，一些内部运行Linux的嵌入式设备，设备有USB设备控制器，可以将PC，也即是我们的主机作为master端，将设备作为slave端和主机通过USB进行通信，从主机的观点来看，主机系统的USB驱动程序控制插入其中的USB设备，而USB gadget的驱动程序控制外围设备作为一个USB设备和主机通信。（因为连接的设备端其实也是一个host controller，所以需要模拟，U盘、网卡等）。

gadget目录下大概分为两个模块：

* 一个udc驱动，针对具体CPU平台的，找不到现成的可以自己实现。
* 另一个就是gadget驱动，主要有file\_storage、ether、serial等。

另外还提供了 USB gadget API，即 USB 设备控制器硬件和 gadget 驱动通信的接口。PC 及 服务器只有 USB 主机控制器硬件，它们并不能作为 USB gadget 存在，而对于嵌入式设备，USB 设备控制器常被集成到处理器中，设备的各种功能，如 U 盘、网卡等常依赖这种 USB 设备控 制器来与主机连接，并且设备的各种功能之间可以切换，比如可以选择作为 U 盘或网卡等

image、input、media、net、serial、storage：剩下的几个目录分别存放各种USB设备的驱动，U盘驱动在storage目录下，触摸屏和USB键盘鼠标的驱动在input目录下。

class：如果前面几个目录没有该驱动，则在此目录下寻找。USB协议中除了通用的软硬件电气接口规范等，还包含了各种各样的Class协议，来为不同的功能定义各自的标准接口和具体总线上的数据交互格式和内容，例如支持U盘功能的Mass Storage Class。

misc：如果之前的目录都没有该设备的驱动，就在该目录下寻找。

usb-skeleton.c是一个简单的USB driver框架

分析代码时Kconfig和Makefile就是Linux kernel迷宫的地图

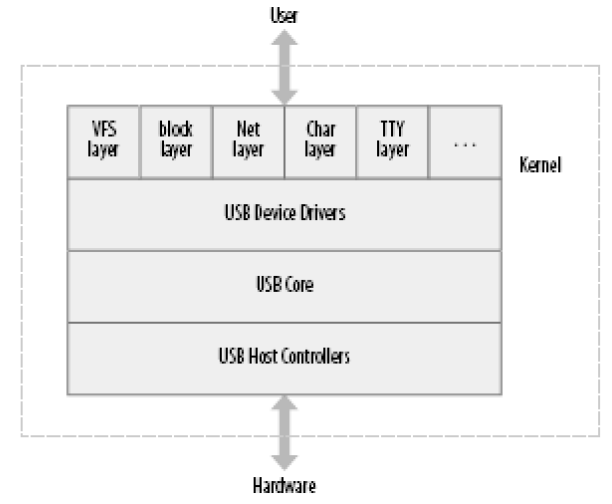
USB\_DEBUG:为啥android系统裁剪掉了。。这是USB 的调试tag，如果你在写USB 设备驱动的话，最好还是打开它

Makefile可比Kconfig简略多了，所以看起来也更亲切点，咱们总是拿的money越多越好，看的代码越少越好。这里之所以会出现CONFIG\_PCI，是因为通常USB的root hub包含在一个PCI设备中，前面也已经聊过了。hcd-pci和hcd顾名而思义就知道是说host controller的，它们实现了host controller公共部分，按协议里的说法它们就是HCDI（HCD的公共接口），host目录下则实现了各种不同的host controller

usb 子系统的初始化在文件drivers/usb/core/usb.c 里，因为咱们这里聊的主题就是usb core，所以如果日后牵涉到core 下面的哪个文件，就不再指明drivers/usb/core/这么一长串目录名了。

## 架构

core、host,controller 和driver 三者之间的关系



host controller 的驱动（HCD）必须位于USB 软件的最下一层, HCD 提供host controller 硬件的抽象，隐藏硬件的细节，在host controller 之下是物理的USB 及所有与之连接的USB 设备。而HCD只对USB core负责，USB core 将用户的请求映射到相关的HCD，用户不能直接访问HCD。

咱们写 USB 驱动的时候，只能调用core 的接口，core 会将咱们的请求发送给相应的HCD

# 3. usb core

drivers/usb/core/usb.c

## 3.1. 入口声明

subsystem\_initcall(usb\_init);

可以理解为module\_init，只不过因为这部分代码比较核心，开发人员把它看作一个子系统，而不仅仅是一个模块。usb\_init是真正的初始话函数。

module\_exit(usb\_exit);

usb\_exit是整个USB子系统结束时的清理函数。

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

\_\_init对于内核来说是一种暗示，表示这个函数仅在初始化期间使用，在模块被装载后，它占用的资源就会释放掉用于它处。

#define \_\_init \_\_attribute\_\_((\_\_section\_\_(".init.text")))

1

\_\_attribute\_\_是GNU C扩展，指示编译器进行特定方面的优化和更仔细的代码检查。GNU支持十几个属性，section就是其中之一。

通常编译器将函数放在.text节，变量放在.data节或.bss节，使用section属性，可以让编译器将函数或变量放在指定的节中。

链接器将相同节的代码或数据安排在一起，\_\_init修饰的所有代码都会被放在.init.text节中，初始化结束后就可以释放这部分内存。

#define subsys\_initcall(fn) \_\_define\_initcall("4",fn,4)

1

\_\_define\_initcall用于将指定的函数指针fn放到initcall.init节中，subsys\_initcall是把fn放到.initcall.init的子节.initcall4.init中。关于.initcall.init、.init.text和.initcall4.init，需要了解一些内核可执行文件相关的概念，说明如下：

内核可执行文件由许多链接在一起的对象文件组成，对象文件有许多节，如文本、数据、init数据、bss等。这些对象文件都是由一个称为链接器脚本的文件链接被装入的。这个链接器脚本的功能就是将输入对象文件的各节映射到输出文件中。

换句话说，它将所有输入对象的文件都链接到单一的可执行文件中，将该可执行文件的各节装入到指定地址处。

vmlinux.lds是存在于arch//目录中的内核链接器脚本，负责链接内核的各个节并将他们转入内存中特定的偏移量处。

\_\_initcall\_start = .;

.initcall.init : AT(ADDR(.initcall.init) -0xC0000000) {

\*(.initcall1.init)

\*(.initcall2.init)

\*(.initcall3.init)

\*(.initcall4.init)

\*(.initcall5.init)

\*(.initcall6.init)

\*(.initcall7.init)

}

\_\_initcall\_end = .;

1

subsys\_initcall将指定的函数指针放在.initcall4.init子节

core\_initcall将函数指针放在.initcall1.init子节

device\_initcall将函数指针放在.initcall6.init子节

各个子节中函数指针的调用顺序是确定的，\_\_init修饰的初始化函数在内核初始化过程中调用的顺序和.initcall.init节里函数指针的顺序有关，不同的初始化函数被放在不同的子节中，因此，也决定了他们的调用顺序。

实际执行函数调用的地方，就在/init/main.c文件中，do\_initcalls函数慧直接调用这里的\_\_initcall\_start、\_\_initcall\_end进行判断。

start\_kernel

rest\_init

kthread-kernel\_init

do\_basic\_setup

do\_initcalls

遍历\_\_initcall\_start到\_\_initcall\_end，执行每个函数指针

End

## 3.2. usb\_init概述

if(nousb) {

pr\_info("USB support disabled\n");

return 0;

}

1

nousb标识：启动内核时通过内核参数去掉USB子系统。

pr\_info只是一个打印信息的可变参数宏，即printk的变体，在include/linux/kernel.h中定义：

#define pr\_info(fmt, arg...) \

printk(KERN\_INFO fmt, ##arg)

C99规定了可变参数宏

#define debug(format, ...) fprintf (stderr, format, \_\_VA\_ARGS\_\_)

1

其中…表示可变参数，调用时替代\_\_VA\_ARGS\_\_

GCC支持更复杂的形式，可以给可变参数取名字，例如：

#define debug(format, args...) fprintf(stderr, format, args)

1

pr\_info定义中还有一个##，是为了处理特殊情况：没有可变参数，如果没有##，就会有一个多余的逗号，使用##使预处理器去掉这个多余的逗号。

这里简单说一下宏定义的特殊符号#和@

#表示将内容变为字符串，添加”“、##表示连接、#@表示将内容变为字符，添加”

##在变参宏\_\_VA\_ARGS\_\_中表示去掉多余的逗号

之后就是usb\_init最核心的工作了

ksuspend\_usb\_init电源管理相关

bus\_register注册USB总线

usb\_host\_init执行主机控制器相关初始化

usb\_major\_init一个总线也是一个设备，必须单独注册，USB是通过快速串行通信来读写数据的，这里把它当作字符设备来注册

usb\_register usbfs相关

usb\_devio\_init usbfs相关

usbfs\_init usbfs相关

usb\_hub\_init Hub初始化

usb\_register\_device\_driver 注册USB设备驱动，这里是USB device driver而不是USB driver，一个设备可以有多个接口，每个接口对应不同的驱动程序，这里的device driver对应的是整个设备，而不是接口。

## 3.3. Linux设备模型

模型的中心就是：总线、设备、驱动，bus、device和driver，都有自己的专属结构，定义在include/linux/device.h中：

struct bus\_type {

const char \* name;

struct module \* owner;

struct kset subsys;

struct kset drivers;

struct kset devices;

struct klist klist\_devices;

struct klist klist\_drivers;

struct blocking\_notifier\_head bus\_notifier;

struct bus\_attribute \* bus\_attrs;

struct device\_attribute \* dev\_attrs;

struct driver\_attribute \* drv\_attrs;

int (\*match)(struct device \* dev, struct device\_driver \* drv);

int (\*uevent)(struct device \*dev, struct kobj\_uevent\_env \*env);

int (\*probe)(struct device \* dev);

int (\*remove)(struct device \* dev);

void (\*shutdown)(struct device \* dev);

int (\*suspend)(struct device \* dev, pm\_message\_t state);

int (\*suspend\_late)(struct device \* dev, pm\_message\_t state);

int (\*resume\_early)(struct device \* dev);

int (\*resume)(struct device \* dev);

unsigned int drivers\_autoprobe:1;

};

struct device\_driver {

const char \* name;

struct bus\_type \* bus;

struct kobject kobj;

struct klist klist\_devices;

struct klist\_node knode\_bus;

struct module \* owner;

const char \* mod\_name; /\* used for built-in modules \*/

struct module\_kobject \* mkobj;

int (\*probe) (struct device \* dev);

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 device {

struct klist klist\_children;

struct klist\_node knode\_parent; /\* node in sibling list \*/

struct klist\_node knode\_driver;

struct klist\_node knode\_bus;

struct device \*parent;

struct kobject kobj;

char bus\_id[BUS\_ID\_SIZE]; /\* position on parent bus \*/

struct device\_type \*type;

unsigned is\_registered:1;

unsigned uevent\_suppress:1;

struct semaphore sem; /\* semaphore to synchronize calls to

\* its driver.

\*/

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 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;

spinlock\_t devres\_lock;

struct list\_head devres\_head;

/\* class\_device migration path \*/

struct list\_head node;

struct class \*class;

dev\_t devt; /\* dev\_t, creates the sysfs "dev" \*/

struct attribute\_group \*\*groups; /\* optional groups \*/

void (\*release)(struct device \* dev);

};

struct bus\_type中有struct kset drivers和struct kset devices；

struct device中有struct bus\_type \*bus和struct device\_driver \*driver；

struct device\_driver中有struct bus\_type \*bus和struct klist klist\_devices；

相互知道彼此的存在。

这里面包含了Linux设备模型中最基本的元素：kobject和kset。

kobject是所有内核对象的积累，所实现的只是一些公共的接口，kset是同种类型kobject对象的集合，可以说是对象的容器。

这样内核使用kobject将各个对象连接起来组成了一个分层的结构体系，kobject包含了parent成员，kset使用链表来实现。

struct kset {

struct kobj\_type \*ktype;

struct list\_head list;

spinlock\_t list\_lock;

struct kobject kobj;

struct kset\_uevent\_ops \*uevent\_ops;

};

struct kset\_uevent\_ops {

int (\*filter)(struct kset \*kset, struct kobject \*kobj);

const char \*(\*name)(struct kset \*kset, struct kobject \*kobj);

int (\*uevent)(struct kset \*kset, struct kobject \*kobj,

struct kobj\_uevent\_env \*env);

};

struct kobj\_type {

void (\*release)(struct kobject \*);

struct sysfs\_ops \* sysfs\_ops;

struct attribute \*\* default\_attrs;

};

1

struct bus\_type中drivers和devices表示了一条总线拥有的两条链表。

此外，klist包含了一个链表和一个自旋锁，暂且把它看成链表也可以。

struct klist {

spinlock\_t k\_lock;

struct list\_head k\_list;

void (\*get)(struct klist\_node \*);

void (\*put)(struct klist\_node \*);

};

struct list\_head {

struct list\_head \*next, \*prev;

};

#define LIST\_HEAD\_INIT(name) { &(name), &(name) }

#define LIST\_HEAD(name) \

struct list\_head name = LIST\_HEAD\_INIT(name)

LIST\_HEAD(xxx\_list) ==>

struct list\_head xxx\_list = {

&(xxx\_list),

&(xxx\_list)

}

设备与驱动的匹配过程：

新增驱动

创建struct device\_driver

插入bus\_type->drivers（kset类型）中，总线可以找到驱动

在bus\_type->devices（kset类型）链表中寻找每个还没有绑定驱动的设备（device->driver(device\_driver类型)指针为空）

匹配特征，如果支持则调用device\_bind\_driver函数

该函数，设置device->driver、添加到device\_driver->klist\_devices(klist类型)

互相关联完成

新增设备

创建struct device

插入bus\_type->devices（kset类型）中，总线可以找到设备

在bus\_type->drivers（kset类型）链表中寻找每个驱动

匹配特征，如果支持则调用device\_bind\_driver函数

该函数，设置device->driver、添加到device\_driver->klist\_devices(klist类型)

互相关联完成

## 3.4. USB总线type

在usb\_init的bus\_register函数调用过程中注册

retval = bus\_register(&usb\_bus\_type);

drivers/usb/core/driver.c中定义：

struct bus\_type usb\_bus\_type = {

.name = "usb",

.match = usb\_device\_match,

.uevent = usb\_uevent,

.suspend = usb\_suspend,

.resume = usb\_resume,

};

这里说一下C语言中结构体初始化语法（C99）：结构的指定初始化器使用点运算符和成员名标识特定的元素，例如只初始化book结构的value成员，可以这样写：

struct book {

char title[MAXTITL];

char author[AXAUTL];

float value;

};

struct book surprise = {.value = 10.99};

可以按照任意顺序使用指定初始化器：

struct book gift = { .value = 25.99,

.author = "James Broadfool",

.title = "Rue for the Toad"};

对特定成员的最后一次赋值才是它实际获得的值，例如：

struct book gift = { .value = 18.90,

.author = "Philionna Pestle",

0.25};

最终value的值就是0.25，因为在结构体声明中value紧跟在author成员之后，新值0.25取代了之前的18.9。

所以USB总线的name是“usb”，match就是总线设备和驱动之间的匹配函数，指向了usb\_device\_match函数

### 3.4.1. bus\_register(&usb\_bus\_type)

drivers/base/bus.c

下面看一下bus\_register的实现：

int bus\_register(struct bus\_type \* bus)

{

int retval;

BLOCKING\_INIT\_NOTIFIER\_HEAD(&bus->bus\_notifier);

//初始化struct bus\_type->bus\_notifier(struct blocking\_notifier\_head类型)

retval = kobject\_set\_name(&bus->subsys.kobj, "%s", bus->name);

//调用kmalloc申请内核空间，为struct bus\_type->subsys(kset类型).kobj(kobject类型).k\_name赋值

//这里赋值为"usb"

if (retval)

goto out;

bus->subsys.kobj.kset = &bus\_subsys;

//指定struct bus\_type->subsys(kset类型).kobj(kobject类型).kset（kset类型）

//这里没有找到bus\_subsys的定义位置

retval = subsystem\_register(&bus->subsys);

//调用kset\_register

//调用（kset\_init & kset\_add &kobject\_uevent(kset->kobj, KOBJ\_ADD)）

//kset\_init就是kobject\_init、kset->list（struct list\_head类型）、kset->list\_lock(spinlock\_t类型)初始化

//kset\_add，调用kobject\_add(&kset->kobj)

//kobject\_add：

//如果kobject->kset存在：

//将kobject->entry（list\_head类型）添加到kobject->kset->list中

//如果kobject->parent不存在，设置parent就是kobject->kset->kobject

//这里就是bus\_subsys->kobject

//create\_dir(kobject)->sysfs\_create\_dir(kobject)

//创建sysfs目录，如果存在parent就在该parent目录下创建，如果不存在就在sys根下创建

//这里就是在bus\_subsys下创建

//kobject\_uevent通知用户空间uevent，调用kobject\_uevent\_env(kobject,kobject\_action,NULL)

//后者发送uevent附带environmental data，单独分析

//这里分析时注意

//struct bus\_type->subsys(kset类型).kobj(kobject类型).kset（kset类型）

//并不是subsys（kset类型）自身，而是刚才设置bus\_subsys

if (retval)

goto out;

retval = bus\_create\_file(bus, &bus\_attr\_uevent);

//->sysfs\_create\_file(bus\_type->subsys.kobj, &bus\_attr\_uevent->attr)

//在kobject目录下创建bus\_attr\_uevent属性文件

if (retval)

goto bus\_uevent\_fail;

kobject\_set\_name(&bus->devices.kobj, "devices");

//设置bus\_type->devices(kset类型).kobj(kobject类型).k\_name为"devices"

bus->devices.kobj.parent = &bus->subsys.kobj;

//设置parent为bus\_type->subsys.kobj，就是刚才创建的名字为usb的kobject

retval = kset\_register(&bus->devices);

//过程同上文分析，核心就是初始化bus\_type->devices(kset类型).kobject对象，这里没有设置该kobject对象的kset，但是已经设置过了parent，调用sysfs\_create\_file进行文件创建

if (retval)

goto bus\_devices\_fail;

kobject\_set\_name(&bus->drivers.kobj, "drivers");

//设置bus\_type->drivers(kset类型).kobj(kobject类型).k\_name为"drivers"

bus->drivers.kobj.parent = &bus->subsys.kobj;

//设置parent为bus\_type->subsys.kobj，就是刚才创建的名字为usb的kobject

bus->drivers.ktype = &driver\_ktype;

//设置bus\_type->drivers(kset类型).ktype(kobj\_type)为预定义的driver\_ktype

retval = kset\_register(&bus->drivers);

//过程同上文分析，核心就是初始化bus\_type->drivers(kset类型).kobject对象，这里没有设置该kobject对象的kset，但是已经设置过了parent，调用sysfs\_create\_file进行文件创建

if (retval)

goto bus\_drivers\_fail;

klist\_init(&bus->klist\_devices, klist\_devices\_get, klist\_devices\_put);

klist\_init(&bus->klist\_drivers, NULL, NULL);

//初始化bus\_type->klist\_devices(klist类型)和bus\_type->klist\_drivers(klist类型)

bus->drivers\_autoprobe = 1;

//设置bus\_type->drivers\_autoprobe=1，字面含义自动检测

retval = add\_probe\_files(bus);

//调用bus\_create\_file（bus\_type, bus\_attr\_drivers\_probe）

//和bus\_create\_file(bus\_type, bus\_attr\_drivers\_autoprobe)

//核心就是调用sysfs\_create\_file，为bus\_type->subsys.kobj创建属性文件probe和autoprobe

if (retval)

goto bus\_probe\_files\_fail;

retval = bus\_add\_attrs(bus);

//如果bus\_type->bus\_attrs（struct bus\_attribute类型，第一个成员是attribute attr）不为空，则调用sysfs\_create\_file，在subsys.kobject下创建每个属性文件

if (retval)

goto bus\_attrs\_fail;

pr\_debug("bus type '%s' registered\n", bus->name);

return 0;

bus\_attrs\_fail:

remove\_probe\_files(bus);

bus\_probe\_files\_fail:

kset\_unregister(&bus->drivers);

bus\_drivers\_fail:

kset\_unregister(&bus->devices);

bus\_devices\_fail:

bus\_remove\_file(bus, &bus\_attr\_uevent);

bus\_uevent\_fail:

subsystem\_unregister(&bus->subsys);

out:

return retval;

}

97

其中bus\_attr\_uevent定义如下

static ssize\_t bus\_uevent\_store(struct bus\_type \*bus,

const char \*buf, size\_t count)

{

enum kobject\_action action;

if (kobject\_action\_type(buf, count, &action) == 0)

kobject\_uevent(&bus->subsys.kobj, action);

return count;

}

static BUS\_ATTR(uevent, S\_IWUSR, NULL, bus\_uevent\_store);

//BUS\_ATTR是一个宏定义，如下

#define BUS\_ATTR(\_name,\_mode,\_show,\_store) \

struct bus\_attribute bus\_attr\_##\_name = \_\_ATTR(\_name,\_mode,\_show,\_store)

//bus\_attribute定义如下

struct bus\_attribute {

struct attribute attr;

ssize\_t (\*show)(struct bus\_type \*, char \* buf);

ssize\_t (\*store)(struct bus\_type \*, const char \* buf, size\_t count);

};

//attribute定义如下

struct attribute {

const char \*name;

struct module \*owner;

mode\_t mode;

};

25

bus\_attr\_drivers\_probe和bus\_attr\_drivers\_autoprobe定义如下

static BUS\_ATTR(drivers\_probe, S\_IWUSR, NULL, store\_drivers\_probe);

static BUS\_ATTR(drivers\_autoprobe, S\_IWUSR | S\_IRUGO,

show\_drivers\_autoprobe, store\_drivers\_autoprobe);

//show\_drivers\_autoprobe展示的就是bus\_type->driver\_autoprobe的值

1

### 3.4.2. kobject\_uevent函数

//通知用户层一个uevent结束

//action表示正在发生的行为

//kobj表示正在发生行为的对象

//返回0表示完成，获取返回错误码

int kobject\_uevent(struct kobject \*kobj, enum kobject\_action action)

{

return kobject\_uevent\_env(kobj, action, NULL);

}

//其中kobject\_action定义如下

enum kobject\_action {

KOBJ\_ADD,

KOBJ\_REMOVE,

KOBJ\_CHANGE,

KOBJ\_MOVE,

KOBJ\_ONLINE,

KOBJ\_OFFLINE,

KOBJ\_MAX

};

8

下面来单独分析一下kobject\_uevent\_env函数实现

//发送一个uevent，附带环境数据

//action表示正在发生的事件

//kobj表示正在发生事件的对象

//envp\_ext指向环境数据的指针

int kobject\_uevent\_env(struct kobject \*kobj, enum kobject\_action action,

char \*envp\_ext[])

{

struct kobj\_uevent\_env \*env;

const char \*action\_string = kobject\_actions[action];//见下文

const char \*devpath = NULL;

const char \*subsystem;

struct kobject \*top\_kobj;

struct kset \*kset;

struct kset\_uevent\_ops \*uevent\_ops;

u64 seq;

int i = 0;

int retval = 0;

pr\_debug("%s\n", \_\_FUNCTION\_\_);

/\* search the kset we belong to \*/

top\_kobj = kobj;

while (!top\_kobj->kset && top\_kobj->parent)

top\_kobj = top\_kobj->parent;

//找到当前kobject所属的kset

if (!top\_kobj->kset) {

pr\_debug("kobject attempted to send uevent without kset!\n");

return -EINVAL;

}

kset = top\_kobj->kset;

uevent\_ops = kset->uevent\_ops;

//获取所属kset及其uevent\_ops

/\*定义如下：

struct kset\_uevent\_ops {

int (\*filter)(struct kset \*kset, struct kobject \*kobj);

const char \*(\*name)(struct kset \*kset, struct kobject \*kobj);

int (\*uevent)(struct kset \*kset, struct kobject \*kobj,

struct kobj\_uevent\_env \*env);

};

三个函数指针作用依次为：

filter:决定是否将事件传递到用户空间

name:负责将相应的字符串传递给用户空间的热插拔处理程序

uevent：将用户空间需要的参数添加到环境变量中\*/

/\* skip the event, if the filter returns zero. \*/

if (uevent\_ops && uevent\_ops->filter)

if (!uevent\_ops->filter(kset, kobj)) {

pr\_debug("kobject filter function caused the event to drop!\n");

return 0;

}

//调用所属kset附带的filter过滤函数，判断是否要放弃该事件

/\* originating subsystem \*/

if (uevent\_ops && uevent\_ops->name)

subsystem = uevent\_ops->name(kset, kobj);

else

subsystem = kobject\_name(&kset->kobj);

if (!subsystem) {

pr\_debug("unset subsystem caused the event to drop!\n");

return 0;

}

//调用所属kset附带的name函数获取子系统名称，或者使用所属kset->kobj（kobject类型）.k\_name

/\* environment buffer \*/

env = kzalloc(sizeof(struct kobj\_uevent\_env), GFP\_KERNEL);

if (!env)

return -ENOMEM;

/\*开辟空间struct kobj\_uevent\_env

struct kobj\_uevent\_env {

char \*envp[UEVENT\_NUM\_ENVP];

int envp\_idx;

char buf[UEVENT\_BUFFER\_SIZE];

int buflen;

};\*/

/\* complete object path \*/

devpath = kobject\_get\_path(kobj, GFP\_KERNEL);

if (!devpath) {

retval = -ENOENT;

goto exit;

}

//按照kobject->parent(kobject)父子关系进行遍历

//利用每一级的kobject->k\_name，中间用/分割，得到对象的路径

/\* default keys \*/

retval = add\_uevent\_var(env, "ACTION=%s", action\_string);

if (retval)

goto exit;

retval = add\_uevent\_var(env, "DEVPATH=%s", devpath);

if (retval)

goto exit;

retval = add\_uevent\_var(env, "SUBSYSTEM=%s", subsystem);

if (retval)

goto exit;

//调用了三个一样的函数add\_uevent\_var：ACTION、DEVPATH、SUBSYSTEM

//将键值信息添加到kobj\_uevent\_env的buf中

//envp是每个键值的单独索引，实际字符串还是只想buf中

/\* keys passed in from the caller \*/

if (envp\_ext) {

for (i = 0; envp\_ext[i]; i++) {

retval = add\_uevent\_var(env, envp\_ext[i]);

if (retval)

goto exit;

}

}

//将参数中环境变量数据，添加到kobj\_uevent\_env中

/\* let the kset specific function add its stuff \*/

if (uevent\_ops && uevent\_ops->uevent) {

retval = uevent\_ops->uevent(kset, kobj, env);

if (retval) {

pr\_debug ("%s - uevent() returned %d\n",

\_\_FUNCTION\_\_, retval);

goto exit;

}

}

//调用所属kset附带的uevent函数，添加所属kset特有的数据到kobj\_uevent\_env中

/\* we will send an event, so request a new sequence number \*/

spin\_lock(&sequence\_lock);

seq = ++uevent\_seqnum;

spin\_unlock(&sequence\_lock);

retval = add\_uevent\_var(env, "SEQNUM=%llu", (unsigned long long)seq);

if (retval)

goto exit;

//利用全局变量uevent\_seqnum获取新的序列号，SEQNUM=%llu 添加到kobj\_uevent\_env中

#if defined(CONFIG\_NET)

/\* send netlink message \*/

if (uevent\_sock) {

struct sk\_buff \*skb;

size\_t len;

/\* allocate message with the maximum possible size \*/

len = strlen(action\_string) + strlen(devpath) + 2;

skb = alloc\_skb(len + env->buflen, GFP\_KERNEL);

if (skb) {

char \*scratch;

/\* add header \*/

scratch = skb\_put(skb, len);

sprintf(scratch, "%s@%s", action\_string, devpath);

/\* copy keys to our continuous event payload buffer \*/

for (i = 0; i < env->envp\_idx; i++) {

len = strlen(env->envp[i]) + 1;

scratch = skb\_put(skb, len);

strcpy(scratch, env->envp[i]);

}

NETLINK\_CB(skb).dst\_group = 1;

netlink\_broadcast(uevent\_sock, skb, 0, 1, GFP\_KERNEL);

}

}

#endif

//调用netlink\_broadcast，socket通信，暂没有详细分析????

/\* call uevent\_helper, usually only enabled during early boot \*/

if (uevent\_helper[0]) {

char \*argv [3];

argv [0] = uevent\_helper;

argv [1] = (char \*)subsystem;

argv [2] = NULL;

retval = add\_uevent\_var(env, "HOME=/");

if (retval)

goto exit;

retval = add\_uevent\_var(env, "PATH=/sbin:/bin:/usr/sbin:/usr/bin");

if (retval)

goto exit;

call\_usermodehelper (argv[0], argv, env->envp, UMH\_WAIT\_EXEC);

}

//uevent\_helper，处理uevent的用户空间程序???

exit:

kfree(devpath);

kfree(env);

return retval;

}

182

其中kobject\_action是一个字符串数组，定义如下：

static const char \*kobject\_actions[] = {

[KOBJ\_ADD] = "add",

[KOBJ\_REMOVE] = "remove",

[KOBJ\_CHANGE] = "change",

[KOBJ\_MOVE] = "move",

[KOBJ\_ONLINE] = "online",

[KOBJ\_OFFLINE] = "offline",

};

这里用到了C语言的数组初始化器（C99）：可以初始化指定的数组元素，传统C初始化语法必须初始化最后一个元素之前的所有元素，才能初始化它：

int arr[6] = {0,0,0,0,0,212}; //传统的语法

1

C99新特性可以在初始化列表中使用带方括号的下标指明待初始化的元素：

int arr[6] = {[5] = 212}; //把arr[5]初始化为212,其他未初始化元素都会被设置为0

1

如果指定初始化器后面有更多的值，例如[4]=31,30,31，那么后面这些值将被用于初始化指定元素后面的元素。如果再次初始化指定的元素，那么最后初始化将会取代之前的初始化。

TODO：关于uevent事件机制的详细分析，需要后续继续梳理

### 3.4.3. usb\_bus\_type的match函数Part1

drivers/usb/core/driver.c

usb\_bus\_type 的match函数实现如下：

static int usb\_device\_match(struct device \*dev, struct device\_driver \*drv)

{

/\* devices and interfaces are handled separately \*/

if (is\_usb\_device(dev)) {

/\* interface drivers never match devices \*/

if (!is\_usb\_device\_driver(drv))

return 0;

/\* TODO: Add real matching code \*/

return 1;

} else {

struct usb\_interface \*intf;

struct usb\_driver \*usb\_drv;

const struct usb\_device\_id \*id;

/\* device drivers never match interfaces \*/

if (is\_usb\_device\_driver(drv))

return 0;

intf = to\_usb\_interface(dev);

usb\_drv = to\_usb\_driver(drv);

id = usb\_match\_id(intf, usb\_drv->id\_table);

if (id)

return 1;

id = usb\_match\_dynamic\_id(intf, usb\_drv);

if (id)

return 1;

}

return 0;

}

参数就是设备和驱动，对应的就是总线两条链表里的设备和驱动。总线上有新设备或驱动添加时，这个函数总是会被调用。

上来就是一个判断，判断该device是否是usb设备（else表示usb接口）

is\_usb\_device实现如下：

**static inline int** is\_usb\_device(**const struct** device \*dev)  
{  
 **return** dev->type == &usb\_device\_type;  
}

//usb\_device\_type定义如下

drivers/usb/core/usb.c

**struct** device\_type usb\_device\_type = {  
 .name = **"usb\_device"**,  
 .release = usb\_release\_dev,  
 .uevent = usb\_dev\_uevent,  
 .devnode = usb\_devnode,  
#ifdef CONFIG\_PM  
 .pm = &usb\_device\_pm\_ops,  
#endif  
};

//device\_type表示设备类型，嵌在struct device中，一个class或者bus可能包含不同的设备类型，例如partitions和disks、mouse和event。等价于kobject的kobj\_type，如果name指定了，uevent会在DEVTYPE变量中包含它

include/linux/device.h

**struct** device\_type {  
 **const char** \*name;  
 **const struct** attribute\_group \*\*groups;  
 **int** (\*uevent)(**struct** device \*dev, **struct** kobj\_uevent\_env \*env);  
 **char** \*(\*devnode)(**struct** device \*dev, umode\_t \*mode,  
 kuid\_t \*uid, kgid\_t \*gid);  
 **void** (\*release)(**struct** device \*dev);  
  
 **const struct** dev\_pm\_ops \*pm;  
};

如果是usb\_device，那么驱动必须是USB设备驱动，is\_usb\_device\_dirver实现如下：

**static inline int** is\_usb\_device\_driver(**struct** device\_driver \*drv)  
{  
 **return** container\_of(drv, **struct** usbdrv\_wrap, driver)->  
 for\_devices;  
}

//container\_of表示转换一个结构体成员到所在的结构体，就是将指针drv转换为struct usbdrv\_wrap

include/linux/usb.h

//usbdrv\_wrap定义如下

**struct** usbdrv\_wrap {  
 **struct** device\_driver driver;  
 **int** for\_devices;  
};

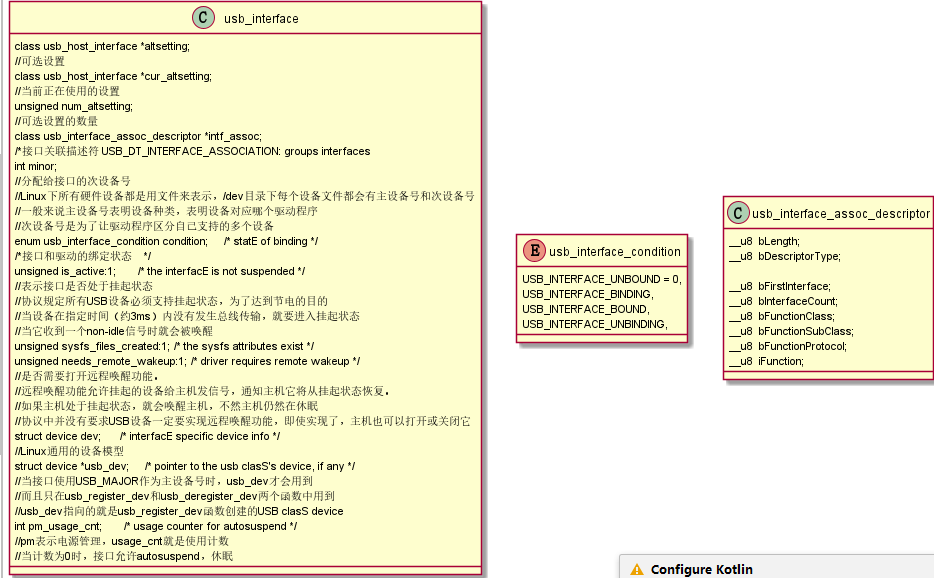
//成员for\_devices，0表示接口驱动，非0表示设备驱动

下面就得看接口相关的实现

### USB设备的四大部分

#### USB接口

Linux设备模型中的device落实在USB子系统，成了两个结构：一个是struct usb\_device，一个是struct usb\_interface。



这里提到的设置和之前说的配置不同，configuration是配置，setting是设置；配置是不同功能的选择，例如是手机作为摄像机还是作为U盘，接口对应功能，设置是具体功能内的设置，例如声音的大小。

#define USB\_MAJOR 180  
#define USB\_DEVICE\_MAJOR 189

其中USB\_DEVICE\_MAJOR是用于usbfs的，USB\_MAJOR是Linux为USB设备预留的主设备号。

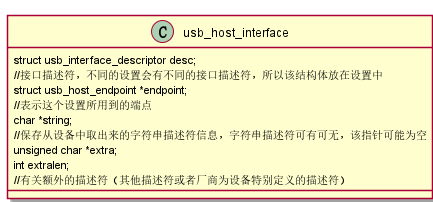
不过USB设备有很多种，并不都会用这个预留的主设备号，比如移动硬盘显示的主设备号是8,，大多数USB设备都会与input、video等子系统关联，并不单单只作为USB设备而存在。

USB接口对应一个USB驱动，当USB接口关联有其他子系统，则需要在对应驱动程序的probe函数中使用相应的注册函数，USB\_MAJOR也就用不到了，struct usb\_interface的字段minor可以忽略，minor只在USB\_MAJOR起作用时来起作用。

例如，USB键盘关联了input子系统，驱动对应drivers/hid/usbhid目录下的usbkbd.c文件，在它的probe函数中可以看到使用了input\_register\_device来注册一个输入设备。

TODO usb\_register\_dev device\_class

下面来看一下接口设置struct usb\_host\_interface



USB描述符是一个带有预定义格式的数据结构，里面保存了USB设备的各种属性还有相关信息，可以通过向设备请求获取他们的内容来了解和感知一个USB设备。分为四种：设备描述符、配置描述符、接口描述符和端点描述符，协议中规定一个USB设备必须支持这四种描述符。

这些描述符都放在USB设备的EEPROM内，EEPROM用来存储设备本身信息，即电可擦写可编程ROM，与Flash虽说都是要电擦除的，但它可以按字节擦除，Flash只能一次擦除一个block，如果要修改比较少的数据的话使用它还是比较合适的，但EEPROM成本较高，所以 一般USB设备只拿它存储一些本身特有的信息，数据的存储还是用Flash。



//该\_\_attribute\_\_告诉编译器这个结构的元素都是1字节对齐的，不要再添加填充位

#define USB\_DT\_INTERFACE\_SIZE 9

//接口描述符长度

lsusb打印出来的设备名称就是字符串描述符提供的，例如：

$lsusb

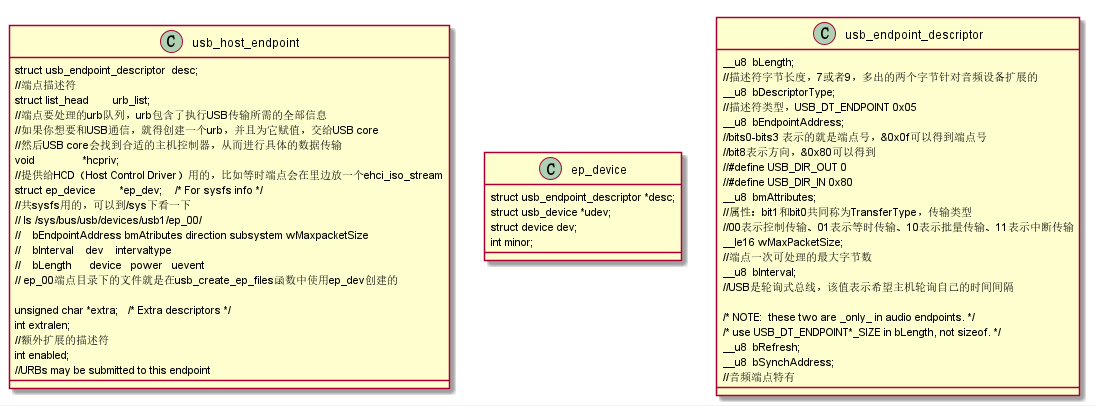
Bus 001 Device 013: ID 04b4:1081 Cypress Semiconductor Corp.

Bus 001 Device 001: ID 0000:0000

其中Cypress Semiconductor Corp.就是字符串描述符提供的，字符串描述符可以有多个，提供一些设备接口相关的描述性信息，比如厂商的名字，产品序列号等，接口描述符中的索引值就是用来区分它们的

#### USB端点

端点是USB数据传输的终点

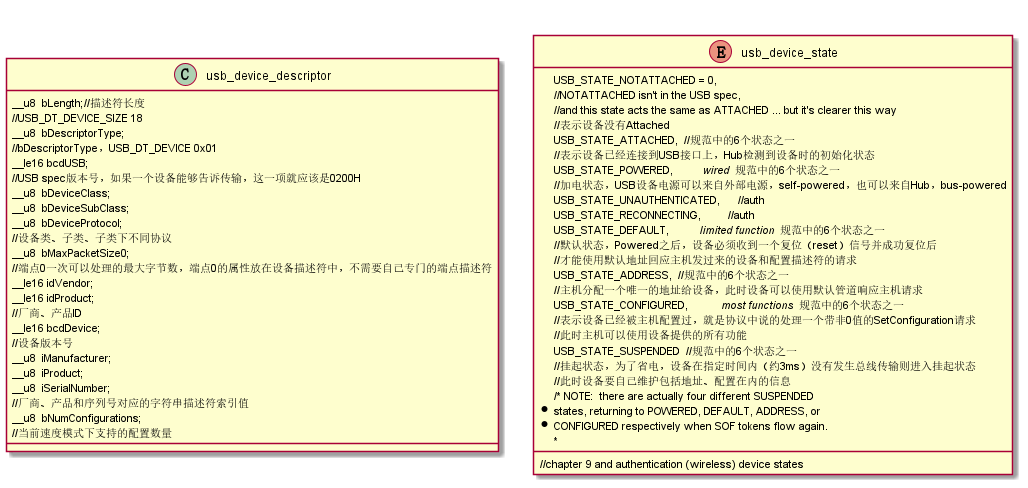


#define USB\_DT\_ENDPOINT\_SIZE 7

#define USB\_DT\_ENDPOINT\_AUDIO\_SIZE 9 /\* Audio extension \*/

#### USB设备





#### USB配置

Dfg



nterface Association Descriptors，前面接口usb\_interface中也有

#### 接口&设备驱动

每个接口对应一个独立的功能，需要专门的驱动进行交流，但是接口隶属于一个USB设备，设备还可以有不同的配置，可以为设备指定特定的配置，就需要USB设备驱动来完成相关处理，struct usb\_device\_dirver，下面分别说一下usb\_device\_driver（设备驱动）和usb\_driver（接口驱动）

struct usb\_driver {

const char \*name;

//驱动名字，对应/sys/bus/usb/drivers/下面子目录的名称，这里所有USB驱动名称必须唯一

int (\*probe) (struct usb\_interface \*intf,

const struct usb\_device\_id \*id);

//判断该驱动是否愿意接受某个接口，一个驱动往往可以支持多个接口

void (\*disconnect) (struct usb\_interface \*intf);

//接口失去联系，或使用rmmod卸载驱动将它和接口强行分开时调用

int (\*ioctl) (struct usb\_interface \*intf, unsigned int code,

void \*buf);

//驱动通过usbfs和用户空间交流时使用ioctl

int (\*suspend) (struct usb\_interface \*intf, pm\_message\_t message);

//挂起时调用

int (\*resume) (struct usb\_interface \*intf);

//唤醒时调用

int (\*reset\_resume)(struct usb\_interface \*intf);

int (\*pre\_reset)(struct usb\_interface \*intf);

//将要复位时调用

int (\*post\_reset)(struct usb\_interface \*intf);

//复位后调用

const struct usb\_device\_id \*id\_table;

//驱动支持的所有设备列表，不属于这张表的设备接口，驱动不支持

/\*

struct usb\_device\_id {

/\* which fields to match against?

\_\_u16 match\_flags;

/\* Used for product specific matches; range is inclusive

\_\_u16 idVendor;

\_\_u16 idProduct;

\_\_u16 bcdDevice\_lo;

\_\_u16 bcdDevice\_hi;

/\* Used for device class matches

\_\_u8 bDeviceClass;

\_\_u8 bDeviceSubClass;

\_\_u8 bDeviceProtocol;

/\* Used for interface class matches

\_\_u8 bInterfaceClass;

\_\_u8 bInterfaceSubClass;

\_\_u8 bInterfaceProtocol;

/\* not matched against

kernel\_ulong\_t driver\_info;

};

\*/

struct usb\_dynids dynids;

/\*动态id

每个驱动诞生时它的id在id\_table中已经确定了，通过动态id可以添加新的id给他

只要新id代表的设备存在，就会和它绑定起来

struct usb\_dynids {

spinlock\_t lock;

struct list\_head list;

};

通过sysfs可以添加新id，在/sys/bus/usb/drivers目录下，选择对应驱动进去，可以看到new\_id文件

使用echo将厂商和产品id写进去就可以了，例如：

echo 0557 2008 > /sys/bus/usb/drivers/foo\_driver/new\_id

\*/

struct usbdrv\_wrap drvwrap;

/\*嵌入struct device\_driver，但是多了一层包装

struct usbdrv\_wrap {

struct device\_driver driver;

int for\_devices; //非0表示device drivers，0表示interface drivers

};

\*/

unsigned int no\_dynamic\_id:1;

//禁止动态id

unsigned int supports\_autosuspend:1;

//对autosuspend的支持，设置为0表示不再允许绑定这个驱动的接口autosuspend

};

#define to\_usb\_driver(d) container\_of(d, struct usb\_driver, drvwrap.driver)

1

2

3

4

5

6

7

8

9

10

11

12

13

14

15

16

17

18

19

20

21

22

23

24

25

26

27

28

29

30

31

32

33

34

35

36

37

38

39

40

41

42

43

44

45

46

47

48

49

50

51

52

53

54

55

56

57

58

59

60

61

62

63

64

65

66

67

68

69

70

71

72

73

设备驱动定义如下，除了少了很多参数外，基本和接口驱动一样，函数参数用的是usb\_device，而不是usb\_interface，再有probe函数没有device\_id列表，表示接受所有的USB设备

struct usb\_device\_driver {

const char \*name;

int (\*probe) (struct usb\_device \*udev);

void (\*disconnect) (struct usb\_device \*udev);

int (\*suspend) (struct usb\_device \*udev, pm\_message\_t message);

int (\*resume) (struct usb\_device \*udev);

struct usbdrv\_wrap drvwrap;

unsigned int supports\_autosuspend:1;

};

#define to\_usb\_device\_driver(d) container\_of(d, struct usb\_device\_driver, \

drvwrap.driver)

1

2

3

4

5

6

7

8

9

10

11

12

13

内核定义了一个usb\_device\_driver，在drivers/usb/core/generic.c中，如下

struct usb\_device\_driver usb\_generic\_driver = {

.name = "usb",

.probe = generic\_probe,

.disconnect = generic\_disconnect,

#ifdef CONFIG\_PM

.suspend = generic\_suspend,

.resume = generic\_resume,

#endif

.supports\_autosuspend = 1,

};

这个对象在usb\_init中调用usb\_register\_device\_driver进行了注册

————————————————

### usb\_bus\_type的match函数Part2

原文链接：<https://blog.csdn.net/HJL_DLUT/article/details/80577921>

usb\_bus\_type 的match函数usb\_device\_match实现如下：

**static int** usb\_device\_match(**struct** device \*dev, **struct** device\_driver \*drv)  
{  
 */\* devices and interfaces are handled separately \*/* **if** (is\_usb\_device(dev)) {  
  
 */\* interface drivers never match devices \*/* **if** (!is\_usb\_device\_driver(drv))  
 **return** 0;  
  
 */\** ***TODO: Add real matching code*** *\*/* **return** 1;  
  
 } **else if** (is\_usb\_interface(dev)) {*//接口*  
 **struct** usb\_interface \*intf;  
 **struct** usb\_driver \*usb\_drv;  
 **const struct** usb\_device\_id \*id;  
*/如果驱动时usb\_device\_driver直接返回*  
 */\* device drivers never match interfaces \*/* **if** (is\_usb\_device\_driver(drv))  
 **return** 0;  
*//将struct device转换为struct usb\_interface*  
 intf = to\_usb\_interface(dev);

*//将struct device\_driver转换为struct usb\_driver*  
 usb\_drv = to\_usb\_driver(drv);  
*//遍历usb\_driver->id\_table(usb\_device\_id \*类型，数组)*

*//调用usb\_match\_one\_id(usb\_interface, usb\_device\_id)，见下文*  
 id = usb\_match\_id(intf, usb\_drv->id\_table);  
 **if** (id)  
 **return** 1;  
*//遍历usb\_driver->id\_table(usb\_device\_id \*类型，数组)*

*//调用usb\_match\_one\_id(usb\_interface, usb\_device\_id)，见下文*  
 id = usb\_match\_dynamic\_id(intf, usb\_drv);  
 **if** (id)  
 **return** 1;  
 }  
  
 **return** 0;  
}

下面看一下usb\_match\_one\_id的实现

int usb\_match\_one\_id(struct usb\_interface \*interface,

const struct usb\_device\_id \*id)

{

struct usb\_host\_interface \*intf;

struct usb\_device \*dev;

// proc\_connectinfo in devio.c may call us with id == NULL. \*/

if (id == NULL)

return 0;

intf = interface->cur\_altsetting;

//获得当前激活的设置usb\_host\_interface

dev = interface\_to\_usbdev(interface);

//利用device->parent获得usb\_interface所属的usb\_device

if (!usb\_match\_device(dev, id))

return 0;

//设备匹配，见下文

/\* The interface class, subclass, and protocol should never be

\* checked for a match if the device class is Vendor Specific,

\* unless the match record specifies the Vendor ID. \*/

if (dev->descriptor.bDeviceClass == USB\_CLASS\_VENDOR\_SPEC &&

!(id->match\_flags & USB\_DEVICE\_ID\_MATCH\_VENDOR) &&

(id->match\_flags & (USB\_DEVICE\_ID\_MATCH\_INT\_CLASS |

USB\_DEVICE\_ID\_MATCH\_INT\_SUBCLASS |

USB\_DEVICE\_ID\_MATCH\_INT\_PROTOCOL)))

return 0;

//如果usb\_device->descriptor.bDeviceClass是Vendor指定的

//并且usb\_device\_id的match\_flags没有指定要匹配VENDOR，但是要匹配Class、SubClass、Protocol，则直接返回0，表示不匹配

if ((id->match\_flags & USB\_DEVICE\_ID\_MATCH\_INT\_CLASS) &&

(id->bInterfaceClass != intf->desc.bInterfaceClass))

return 0;

//usb\_device\_id和接口描述符中的bInterfaceClass匹配

if ((id->match\_flags & USB\_DEVICE\_ID\_MATCH\_INT\_SUBCLASS) &&

(id->bInterfaceSubClass != intf->desc.bInterfaceSubClass))

return 0;

//usb\_device\_id和接口描述符中的bInterfaceSubClass匹配

if ((id->match\_flags & USB\_DEVICE\_ID\_MATCH\_INT\_PROTOCOL) &&

(id->bInterfaceProtocol != intf->desc.bInterfaceProtocol))

return 0;

//usb\_device\_id和接口描述符中的bInterfaceProtocol匹配

return 1;

}

1

2

3

4

5

6

7

8

9

10

11

12

13

14

15

16

17

18

19

20

21

22

23

24

25

26

27

28

29

30

31

32

33

34

35

36

37

38

39

40

41

42

43

44

45

usb\_match\_device实现如下：

如果usb\_device\_id->match\_flags & USB\_DEVICE\_ID\_MATCH\_VENDOR

usb\_device\_id->idVendor等于usb\_device->descriptor.idVendor

如果usb\_device\_id->match\_flags & USB\_DEVICE\_ID\_MATCH\_PRODUCT

usb\_device\_id->idProduct等于usb\_device->descriptor.idProduct

如果usb\_device\_id->match\_flags & USB\_DEVICE\_ID\_MATCH\_DEV\_LO

usb\_device\_id->bcdDevice\_lo不大于usb\_device->descriptor.bcdDevice

如果usb\_device\_id->match\_flags & USB\_DEVICE\_ID\_MATCH\_DEV\_HI

usb\_device\_id->bcdDevice\_hi不小于usb\_device->descriptor.bcdDevice

如果usb\_device\_id->match\_flags & USB\_DEVICE\_ID\_MATCH\_DEV\_CLASS

usb\_device\_id->bDeviceClass不小于usb\_device->descriptor.bDeviceClass

如果usb\_device\_id->match\_flags & USB\_DEVICE\_ID\_MATCH\_DEV\_SUBCLASS

usb\_device\_id->bDeviceSubClass不小于usb\_device->descriptor.bDeviceSubClass

如果usb\_device\_id->match\_flags & USB\_DEVICE\_ID\_MATCH\_DEV\_PROTOCOL

usb\_device\_id->bDeviceProtocol不小于usb\_device->descriptor.bDeviceProtocol

————————————————

## 3.6. usb\_init关键函数详细分析

### 3.6.1. ksuspend\_usb\_init电源管理相关-TODO

### 3.6.2. bus\_register注册USB总线

见5.4.1.bus\_register(&usb\_bus\_type)

### 3.6.3. usb\_host\_init执行主机控制器相关初始化 （模块+struct class）

static struct class \*usb\_host\_class;

int usb\_host\_init(void)

{

int retval = 0;

usb\_host\_class = class\_create(THIS\_MODULE, "usb\_host");

//字面意思，创建一个struct class结构体，见下文

if (IS\_ERR(usb\_host\_class))

retval = PTR\_ERR(usb\_host\_class);

return retval;

}

这里涉及了一个概念device class是设备的分类，定义在include/linux/device.h中如下

\*

\* device classes

\*/

struct class {

const char \* name;

struct module \* owner;

struct kset subsys;

struct list\_head children;

struct list\_head devices;

struct list\_head interfaces;

struct kset class\_dirs;

struct semaphore sem; /\* locks both the children and interfaces lists \*/

struct class\_attribute \* class\_attrs;

struct class\_device\_attribute \* class\_dev\_attrs;

struct device\_attribute \* dev\_attrs;

int (\*uevent)(struct class\_device \*dev, struct kobj\_uevent\_env \*env);

int (\*dev\_uevent)(struct device \*dev, struct kobj\_uevent\_env \*env);

void (\*release)(struct class\_device \*dev);

void (\*class\_release)(struct class \*class);

void (\*dev\_release)(struct device \*dev);

int (\*suspend)(struct device \*, pm\_message\_t state);

int (\*resume)(struct device \*);

};

1

2

3

4

5

6

7

8

9

10

11

12

13

14

15

16

17

18

19

20

21

22

23

24

25

26

27

28

class\_create函数实现如下：

struct class \*class\_create(struct module \*owner, const char \*name)

{

struct class \*cls;

int retval;

cls = kzalloc(sizeof(\*cls), GFP\_KERNEL);

if (!cls) {

retval = -ENOMEM;

goto error;

}

//开辟空间，以0初始化的kmalloc

cls->name = name;

cls->owner = owner;

cls->class\_release = class\_create\_release;

cls->release = class\_device\_create\_release;

//成员赋值

//本例class->name = "usb\_host"

//本例class->owner = THIS\_MODULE，见下文

retval = class\_register(cls);

//注册device class，见下文

if (retval)

goto error;

return cls;

error:

kfree(cls);

return ERR\_PTR(retval);

}

1

2

3

4

5

6

7

8

9

10

11

12

13

14

15

16

17

18

19

20

21

22

23

24

25

26

27

28

29

这里简单说一下THIS\_MODULE和Linux的模块

结构体struct module在内核中代表一个内核模块，通过insmod(实际执行init\_module系统调用)把自己编写的内核模块插入内核时，模块便与一个 struct module结构体相关联，并成为内核的一部分。

insmod调用了系统调用init\_module，在该系统调用函数中，首先调用 load\_module，把用户空间传入的整个内核模块文件创建成一个内核模块，返回一个struct module结构体。内核中便以这个结构体代表这个内核模块。

state是模块当前的状态。它是一个枚举型变量

可取的值为：MODULE\_STATE\_LIVE，MODULE\_STATE\_COMING，MODULE\_STATE\_GOING。

1

2

load\_module函数中完成模块的部分创建工作后，把状态置为 MODULE\_STATE\_COMING，sys\_init\_module函数中完成模块的全部初始化工作后（包括把模块加入全局的模块列表，调用模块本身的初始化函数)，把模块状态置为MODULE\_STATE\_LIVE。

最后，使用rmmod工具卸载模块时，会调用系统调用 delete\_module，会把模块的状态置为MODULE\_STATE\_GOING。这是模块内部维护的一个状态。

下面看一下module\_init的实现，include/linux/init.h

#ifndef MODULE

//见下文

#ifndef \_\_ASSEMBLY\_\_

...

/\*

\* A "pure" initcall has no dependencies on anything else, and purely

\* initializes variables that couldn't be statically initialized.

\*

\* This only exists for built-in code, not for modules.

\*/

#define pure\_initcall(fn) \_\_define\_initcall("0",fn,0)

#define core\_initcall(fn) \_\_define\_initcall("1",fn,1)

#define core\_initcall\_sync(fn) \_\_define\_initcall("1s",fn,1s)

#define postcore\_initcall(fn) \_\_define\_initcall("2",fn,2)

#define postcore\_initcall\_sync(fn) \_\_define\_initcall("2s",fn,2s)

#define arch\_initcall(fn) \_\_define\_initcall("3",fn,3)

#define arch\_initcall\_sync(fn) \_\_define\_initcall("3s",fn,3s)

#define subsys\_initcall(fn) \_\_define\_initcall("4",fn,4)

#define subsys\_initcall\_sync(fn) \_\_define\_initcall("4s",fn,4s)

#define fs\_initcall(fn) \_\_define\_initcall("5",fn,5)

#define fs\_initcall\_sync(fn) \_\_define\_initcall("5s",fn,5s)

#define rootfs\_initcall(fn) \_\_define\_initcall("rootfs",fn,rootfs)

#define device\_initcall(fn) \_\_define\_initcall("6",fn,6)

#define device\_initcall\_sync(fn) \_\_define\_initcall("6s",fn,6s)

#define late\_initcall(fn) \_\_define\_initcall("7",fn,7)

#define late\_initcall\_sync(fn) \_\_define\_initcall("7s",fn,7s)

#define \_\_initcall(fn) device\_initcall(fn)

#define module\_init(x) \_\_initcall(x);

#else /\* MODULE \*/

...

#endif

1

2

3

4

5

6

7

8

9

10

11

12

13

14

15

16

17

18

19

20

21

22

23

24

25

26

27

28

29

30

31

32

33

34

35

36

37

MODULE表示是否编译为模块：在make xxxconfig的时候，每一个模块都有三种选择：Y M N。选N不编译模块。选Y模块编译进内核。选M，就会在编译这个模块时候定义MODULE了。是在Makefile里面定义的。

在3.1章的时候，做过分析，显然module\_init就是将该函数指针放在.initcall6.init子节，start\_kernel的最后会调用这些函数

宏THIS\_MODULE表示当前模块，用来引用模块的struct module结构，下面看一下THIS\_MODULE的实现

#ifdef MODULE

extern struct module \_\_this\_module;

#define THIS\_MODULE (&\_\_this\_module)

#else /\* !MODULE \*/

#define THIS\_MODULE ((struct module \*)0)

#endif

1

2

3

4

5

6

指向的就是\_\_this\_module，编译模块后，会生成\*.mod.c这样的文件，打开文件后，出现如下类似的定义：

struct module \_\_this\_module

\_\_attribute\_\_((section(".gnu.linkonce.this\_module"))) = {

.name = KBUILD\_MODNAME,

.init = init\_module,

#ifdef CONFIG\_MODULE\_UNLOAD

.exit = cleanup\_module,

#endif

};

1

2

3

4

5

6

7

8

这个文件是调用modpost生成，所以生成模块的后就会出现\_\_this\_module这个变量，表示当前模块

最后分析一下class\_register

int class\_register(struct class \* cls)

{

int error;

pr\_debug("device class '%s': registering\n", cls->name);

INIT\_LIST\_HEAD(&cls->children);

INIT\_LIST\_HEAD(&cls->devices);

INIT\_LIST\_HEAD(&cls->interfaces);

kset\_init(&cls->class\_dirs);

init\_MUTEX(&cls->sem);

//相关成员的初始化

error = kobject\_set\_name(&cls->subsys.kobj, "%s", cls->name);

//设置class->subsys(kset类型).kobj(kobject类型).k\_name

//本例为"usb\_host"

if (error)

return error;

cls->subsys.kobj.kset = &class\_subsys;

//指定class->subsys(kset类型).kobj(kobject类型)的所属kset，class\_subsys为预定义的

//暂未找到定义

error = subsystem\_register(&cls->subsys);

//调用subsystem\_register进行kset的注册，在3.4.1分析过，总结来说就是：

//进行对应kset及包含kobject的初始化，将该kset->kobj添加到kset->kobj.kset的list中

//设置kset->kobj.parent = kset->kobj.kset->kobj

//创建该kset的sysfs，就是在parent目录下

//kobject\_uevent通知用户空间uevent

//这里注意kset->kobj.kset并不是自己，而是刚才设置的class\_subsys

if (!error) {

error = add\_class\_attrs(class\_get(cls));

class\_put(cls);

}

return error;

}

### 3.6.4. usb\_major\_init总线设备注册（register\_chrdev）

Gvbjh

一个总线也是一个设备，必须单独注册，USB是通过快速串行通信来读写数据的，这里把它当作字符设备来注册

int usb\_major\_init(void)

{

int error;

error = register\_chrdev(USB\_MAJOR, "usb", &usb\_fops);

//字面理解注册一个字符设备，详见下文

if (error)

err("unable to get major %d for usb devices", USB\_MAJOR);

return error;

}

1

2

3

4

5

6

7

8

9

10

11

其中usb\_fops定义如下，是字符设备的文件操作函数指针

static const struct file\_operations usb\_fops = {

.owner = THIS\_MODULE,

.open = usb\_open,

};

1

2

3

4

下面看一下register\_chrdev的实现

//字符设备结构体

struct cdev {

struct kobject kobj;

struct module \*owner;

const struct file\_operations \*ops;

struct list\_head list;

dev\_t dev;

unsigned int count;

};

//为字符设备注册一个major number

//@major：主设备号，如果是0则动态分配

//@name：设备名称

//@fops：设备相关的文件操作

int register\_chrdev(unsigned int major, const char \*name,

const struct file\_operations \*fops)

{

struct char\_device\_struct \*cd;

struct cdev \*cdev;

char \*s;

int err = -ENOMEM;

cd = \_\_register\_chrdev\_region(major, 0, 256, name);

//注册一个major，并指定minor范围，详见下文

if (IS\_ERR(cd))

return PTR\_ERR(cd);

cdev = cdev\_alloc();

//开辟一个cdev结构体，主要是开辟cdev空间，初始化kobject

if (!cdev)

goto out2;

cdev->owner = fops->owner;

cdev->ops = fops;

kobject\_set\_name(&cdev->kobj, "%s", name);

for (s = strchr(kobject\_name(&cdev->kobj),'/'); s; s = strchr(s, '/'))

\*s = '!';

//cdev结构体赋值

err = cdev\_add(cdev, MKDEV(cd->major, 0), 256);

//在系统中添加一个字符设备，详见下文

if (err)

goto out;

cd->cdev = cdev;

return major ? 0 : cd->major;

out:

kobject\_put(&cdev->kobj);

out2:

kfree(\_\_unregister\_chrdev\_region(cd->major, 0, 256));

return err;

}

1

2

3

4

5

6

7

8

9

10

11

12

13

14

15

16

17

18

19

20

21

22

23

24

25

26

27

28

29

30

31

32

33

34

35

36

37

38

39

40

41

42

43

44

45

46

47

48

49

50

51

52

主设备号注册函数实现如下

static struct char\_device\_struct {

struct char\_device\_struct \*next;

unsigned int major;

unsigned int baseminor;

int minorct;

char name[64];

struct file\_operations \*fops;

struct cdev \*cdev; /\* will die \*/

} \*chrdevs[CHRDEV\_MAJOR\_HASH\_SIZE];

#define CHRDEV\_MAJOR\_HASH\_SIZE 255

//全局的数组chrdevs，最多255个

static struct char\_device\_struct \*

\_\_register\_chrdev\_region(unsigned int major, unsigned int baseminor,

int minorct, const char \*name)

{

struct char\_device\_struct \*cd, \*\*cp;

int ret = 0;

int i;

cd = kzalloc(sizeof(struct char\_device\_struct), GFP\_KERNEL);

//开辟struct char\_device\_struct空间

if (cd == NULL)

return ERR\_PTR(-ENOMEM);

mutex\_lock(&chrdevs\_lock);

/\* temporary \*/

if (major == 0) {

for (i = ARRAY\_SIZE(chrdevs)-1; i > 0; i--) {

if (chrdevs[i] == NULL)

break;

}

if (i == 0) {

ret = -EBUSY;

goto out;

}

major = i;

ret = major;

}

//如果参数major为0，那么需要动态开辟，寻找数组chrdevs中目前没有被占用的最小索引作为major

cd->major = major;

cd->baseminor = baseminor;

cd->minorct = minorct;

strncpy(cd->name,name, 64);

//设置struct char\_device\_struct的成员

i = major\_to\_index(major);

for (cp = &chrdevs[i]; \*cp; cp = &(\*cp)->next)

if ((\*cp)->major > major ||

((\*cp)->major == major &&

(((\*cp)->baseminor >= baseminor) ||

((\*cp)->baseminor + (\*cp)->minorct > baseminor))))

break;

//首先获取当前major在chrdevs数组的位置，chrdevs[major%255]

//每个chrdevs[i]里面也是一个链表，next指针来遍历

//如果该链表中出现major大于当前major

//或者 major等于当前major并且（baseminor或者最大minor>=当前baseminor）

//意味着major相同，并且minor范围有交叉 或者 minor比当前minor范围整体大

//则停止遍历，得到对应的struct char\_device\_struct

//例如，当前major为258，chardevs[3]的链表中major为3，

/\* Check for overlapping minor ranges. \*/

if (\*cp && (\*cp)->major == major) {

int old\_min = (\*cp)->baseminor;

int old\_max = (\*cp)->baseminor + (\*cp)->minorct - 1;

int new\_min = baseminor;

int new\_max = baseminor + minorct - 1;

/\* New driver overlaps from the left. \*/

if (new\_max >= old\_min && new\_max <= old\_max) {

ret = -EBUSY;

goto out;

}

/\* New driver overlaps from the right. \*/

if (new\_min <= old\_max && new\_min >= old\_min) {

ret = -EBUSY;

goto out;

}

}

//如果存在major相同的项，并且minor范围有交叉，则跳转到out，错误

//最终得到的合适的插入位置

//将新创建的struct char\_device\_struct添加到chrdevs[major%255]的next链表中

cd->next = \*cp;

\*cp = cd;

mutex\_unlock(&chrdevs\_lock);

return cd;

out:

mutex\_unlock(&chrdevs\_lock);

kfree(cd);

return ERR\_PTR(ret);

}

1

2

3

4

5

6

7

8

9

10

11

12

13

14

15

16

17

18

19

20

21

22

23

24

25

26

27

28

29

30

31

32

33

34

35

36

37

38

39

40

41

42

43

44

45

46

47

48

49

50

51

52

53

54

55

56

57

58

59

60

61

62

63

64

65

66

67

68

69

70

71

72

73

74

75

76

77

78

79

80

81

82

83

84

85

86

87

88

89

90

91

92

93

94

95

96

97

下面我们通过例子来进行模拟，已加深理解

init：

数组每一项指针都为空

step1：

增加major=3，minor=15-20

结果chrdevs[3]

major=3(minor=15-20)

step2：

增加major=3，minor=0-10

结果chardevs[3]

major=3(minor=0-10)->major=3(minor=15-20)

step3：

增加major=3，minor=25-30

结果chardevs[3]

major=3(minor=0-10)->major=3(minor=15-20)->major=3(minor=25-30)

step4：

增加major=258，minor=10-30

结果chardevs[3]

major=3(minor=0-10)->major=3(minor=15-20)->major=3(minor=25-30) ->major=258(minor=10-30)

就是一个插入排序。

这里专门说一下

cd->next = \*cp;

\*cp = cd;

1

2

是如何进行链表插入的？

chrdevs数组元素就是指针，这里的cp是双层指针，&chardevs[i]，chardevs[i]->next指向的就是一个char\_device\_struct结构体，而cp取得就是next指针的地址，所以这里就是在为next指针赋值

下面看一下cdev\_add的实现，代码如下

err = cdev\_add(cdev, MKDEV(cd->major, 0), 256);

//MKDEV实现如下

#define MAJOR(dev) ((unsigned int) ((dev) >> MINORBITS))

#define MINOR(dev) ((unsigned int) ((dev) & MINORMASK))

#define MKDEV(ma,mi) (((ma) << MINORBITS) | (mi))

//后20位是次设备号，前面是主设备号，sizeof(unsinged long)-20

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

{

p->dev = dev;

p->count = count;

return kobj\_map(cdev\_map, dev, count, NULL, exact\_match, exact\_lock, p);

//核心调用kobj\_map，分析见下文

}

1

2

3

4

5

6

7

8

9

10

11

12

13

14

首先看一下kobj\_map的调用

kobj\_map(cdev\_map, dev, count, NULL, exact\_match, exact\_lock, p);

1

其中cdev\_map是一个全局变量，定义如下

static struct kobj\_map \*cdev\_map;

struct kobj\_map {

struct probe {

struct probe \*next;

dev\_t dev;

unsigned long range;

struct module \*owner;

kobj\_probe\_t \*get;

int (\*lock)(dev\_t, void \*);

void \*data;

} \*probes[255];

struct mutex \*lock;

};

1

2

3

4

5

6

7

8

9

10

11

12

13

函数具体实现如下：

int kobj\_map(struct kobj\_map \*domain, dev\_t dev, unsigned long range,

struct module \*module, kobj\_probe\_t \*probe,

int (\*lock)(dev\_t, void \*), void \*data)

{

unsigned n = MAJOR(dev + range - 1) - MAJOR(dev) + 1;

//表示设备号范围(dev, dev+range)中不同的主设备号的个数。通常n的值为1。

unsigned index = MAJOR(dev);

//主设备号

unsigned i;

struct probe \*p;

if (n > 255)

n = 255;

//主设备号最大也就是255

p = kmalloc(sizeof(struct probe) \* n, GFP\_KERNEL);

//开辟主设备数量的struct probe

if (p == NULL)

return -ENOMEM;

for (i = 0; i < n; i++, p++) {

p->owner = module;

p->get = probe;

p->lock = lock;

p->dev = dev;

p->range = range;

p->data = data;

}

//进行struct probe初始化

mutex\_lock(domain->lock);

for (i = 0, p -= n; i < n; i++, p++, index++) {

//kobj\_map中的probes数组中每个元素为一个struct probe链表的头指针。

struct probe \*\*s = &domain->probes[index % 255];

//每个链表中的probe对象有（MAJOR（probe.dev） % 255）值相同的关系

//若主设备号小于255， 则每个链表中的probe都有相同的主设备号。

while (\*s && (\*s)->range < range)

s = &(\*s)->next;

p->next = \*s;

\*s = p;

//链表中的元素是按照range值从小到大排列的。while循环即是找出该将p插入的位置。

}

//针对本例就是在全局kobj\_map：cdev\_map中添加对应major的struct probe结构

mutex\_unlock(domain->lock);

return 0;

}

1

2

3

4

5

6

7

8

9

10

11

12

13

14

15

16

17

18

19

20

21

22

23

24

25

26

27

28

29

30

31

32

33

34

35

36

37

38

39

40

41

42

43

44

45

46

总的来说就是在两个全局变量struct char\_device\_struct \*chrdevs[255] 和struct kobj\_map \*cdev\_map中添加该major对应的结构体

3.6.5. usbfs相关-TODO

usb\_register\_usbfs、usb\_devio\_init、usbfs\_init

3.6.6. usb\_hub\_init HUB初始化（usb\_register->usb\_register\_driver->driver\_register）

int usb\_hub\_init(void)

{

if (usb\_register(&hub\_driver) < 0) {

//注册usb接口驱动：hub\_driver，见下文

printk(KERN\_ERR "%s: can't register hub driver\n",

usbcore\_name);

return -1;

}

khubd\_task = kthread\_run(hub\_thread, NULL, "khubd");

//创建线程，名字为khubd，执行hub\_thread函数，见下文

if (!IS\_ERR(khubd\_task))

return 0;

/\* Fall through if kernel\_thread failed \*/

usb\_deregister(&hub\_driver);

printk(KERN\_ERR "%s: can't start khubd\n", usbcore\_name);

return -1;

}

1

2

3

4

5

6

7

8

9

10

11

12

13

14

15

16

17

18

19

20

首先分析一下usb\_register函数，就是用来注册接口驱动的，相关实现如下：

static struct usb\_driver hub\_driver = {

.name = "hub",

.probe = hub\_probe,

.disconnect = hub\_disconnect,

.suspend = hub\_suspend,

.resume = hub\_resume,

.reset\_resume = hub\_reset\_resume,

.pre\_reset = hub\_pre\_reset,

.post\_reset = hub\_post\_reset,

.ioctl = hub\_ioctl,

.id\_table = hub\_id\_table,

.supports\_autosuspend = 1,

};

//hub的usb\_driver接口驱动定义

static inline int usb\_register(struct usb\_driver \*driver)

{

return usb\_register\_driver(driver, THIS\_MODULE, KBUILD\_MODNAME);

}

//真正的实现如下

int usb\_register\_driver(struct usb\_driver \*new\_driver, struct module \*owner,

const char \*mod\_name)

{

int retval = 0;

if (usb\_disabled())

return -ENODEV;

new\_driver->drvwrap.for\_devices = 0;

new\_driver->drvwrap.driver.name = (char \*) new\_driver->name;

new\_driver->drvwrap.driver.bus = &usb\_bus\_type;

new\_driver->drvwrap.driver.probe = usb\_probe\_interface;

new\_driver->drvwrap.driver.remove = usb\_unbind\_interface;

new\_driver->drvwrap.driver.owner = owner;

new\_driver->drvwrap.driver.mod\_name = mod\_name;

spin\_lock\_init(&new\_driver->dynids.lock);

INIT\_LIST\_HEAD(&new\_driver->dynids.list);

//初始化struct usb\_driver成员，主要是struct usbdrv\_wrap中的device\_driver

retval = driver\_register(&new\_driver->drvwrap.driver);

//通用的驱动注册函数，见下文

if (!retval) {

pr\_info("%s: registered new interface driver %s\n",

usbcore\_name, new\_driver->name);

usbfs\_update\_special();

usb\_create\_newid\_file(new\_driver);

} else {

printk(KERN\_ERR "%s: error %d registering interface "

" driver %s\n",

usbcore\_name, retval, new\_driver->name);

}

return retval;

}

1

2

3

4

5

6

7

8

9

10

11

12

13

14

15

16

17

18

19

20

21

22

23

24

25

26

27

28

29

30

31

32

33

34

35

36

37

38

39

40

41

42

43

44

45

46

47

48

49

50

51

52

53

54

下面分析一下通用的驱动注册函数

int driver\_register(struct device\_driver \* drv)

{

if ((drv->bus->probe && drv->probe) ||

(drv->bus->remove && drv->remove) ||

(drv->bus->shutdown && drv->shutdown)) {

printk(KERN\_WARNING "Driver '%s' needs updating - please use bus\_type methods\n", drv->name);

}

klist\_init(&drv->klist\_devices, NULL, NULL);

//初始化device\_driver的klist\_devices链表，用于添加匹配的设备

return bus\_add\_driver(drv);

//在总线中添加驱动，见下文

}

int bus\_add\_driver(struct device\_driver \*drv)

{

struct bus\_type \* bus = bus\_get(drv->bus);

//获取所属总线

int error = 0;

if (!bus)

return -EINVAL;

pr\_debug("bus %s: add driver %s\n", bus->name, drv->name);

error = kobject\_set\_name(&drv->kobj, "%s", drv->name);

//设置驱动内核对象的名字

if (error)

goto out\_put\_bus;

drv->kobj.kset = &bus->drivers;

//设置驱动内核对象所属kset，bus\_type->drivers（kset类型）

error = kobject\_register(&drv->kobj);

//依次调用kobject\_init、kobject\_register和kobject\_uevent

//核心实现为，kobject初始化

//kobject->parent=所属kset.kobj（kobject类型）

//kobject添加到所属kset的list中

//sysfs操作：在parent下添加kobject文件夹

if (error)

goto out\_put\_bus;

if (drv->bus->drivers\_autoprobe) {

error = driver\_attach(drv);

if (error)

goto out\_unregister;

}

//如果所属bus支持自动检测，则调用driver\_attach进行驱动匹配，见下文

klist\_add\_tail(&drv->knode\_bus, &bus->klist\_drivers);

//将struct device\_driver添加到bus\_type->klist\_drivers链表中

module\_add\_driver(drv->owner, drv);

//将驱动模块添加到系统模块系统中

error = driver\_create\_file(drv, &driver\_attr\_uevent);

//sysfs操作：创建驱动对象的属性文件uevent

if (error) {

printk(KERN\_ERR "%s: uevent attr (%s) failed\n",

\_\_FUNCTION\_\_, drv->name);

}

error = driver\_add\_attrs(bus, drv);

//sysfs操作：遍历bus\_type->drv\_attrs[]数组，创建驱动对象的属性文件

if (error) {

/\* How the hell do we get out of this pickle? Give up \*/

printk(KERN\_ERR "%s: driver\_add\_attrs(%s) failed\n",

\_\_FUNCTION\_\_, drv->name);

}

error = add\_bind\_files(drv);

//sysfs操作：创建驱动对象的属性文件bind和unbind

if (error) {

/\* Ditto \*/

printk(KERN\_ERR "%s: add\_bind\_files(%s) failed\n",

\_\_FUNCTION\_\_, drv->name);

}

return error;

out\_unregister:

kobject\_unregister(&drv->kobj);

out\_put\_bus:

bus\_put(bus);

return error;

}

1

2

3

4

5

6

7

8

9

10

11

12

13

14

15

16

17

18

19

20

21

22

23

24

25

26

27

28

29

30

31

32

33

34

35

36

37

38

39

40

41

42

43

44

45

46

47

48

49

50

51

52

53

54

55

56

57

58

59

60

61

62

63

64

65

66

67

68

69

70

71

72

73

74

75

76

driver\_attach实现如下：

int driver\_attach(struct device\_driver \* drv)

{

return bus\_for\_each\_dev(drv->bus, NULL, drv, \_\_driver\_attach);

}

int bus\_for\_each\_dev(struct bus\_type \* bus, struct device \* start,

void \* data, int (\*fn)(struct device \*, void \*))

{

struct klist\_iter i;

struct device \* dev;

int error = 0;

if (!bus)

return -EINVAL;

klist\_iter\_init\_node(&bus->klist\_devices, &i,

(start ? &start->knode\_bus : NULL));

while ((dev = next\_device(&i)) && !error)

error = fn(dev, data);

klist\_iter\_exit(&i);

return error;

}

//就是遍历bus\_type->klist\_devices中每一个device，调用\_\_driver\_attach函数，如下

static int \_\_driver\_attach(struct device \* dev, void \* data)

{

struct device\_driver \* drv = data;

/\*

\* Lock device and try to bind to it. We drop the error

\* here and always return 0, because we need to keep trying

\* to bind to devices and some drivers will return an error

\* simply if it didn't support the device.

\*

\* driver\_probe\_device() will spit a warning if there

\* is an error.

\*/

if (dev->parent) /\* Needed for USB \*/

down(&dev->parent->sem);

down(&dev->sem);

if (!dev->driver)

driver\_probe\_device(drv, dev);

//核心实现，见下文

up(&dev->sem);

if (dev->parent)

up(&dev->parent->sem);

return 0;

}

int driver\_probe\_device(struct device\_driver \* drv, struct device \* dev)

{

int ret = 0;

if (!device\_is\_registered(dev))

return -ENODEV;

if (drv->bus->match && !drv->bus->match(dev, drv))

goto done;

//如果所属总线有match函数，则调用match函数进行驱动和设备的匹配

//本例就是调用usb\_bus\_type的match函数usb\_device\_match，前文分析过

pr\_debug("%s: Matched Device %s with Driver %s\n",

drv->bus->name, dev->bus\_id, drv->name);

ret = really\_probe(dev, drv);

//检测函数，详见下文

done:

return ret;

}

1

2

3

4

5

6

7

8

9

10

11

12

13

14

15

16

17

18

19

20

21

22

23

24

25

26

27

28

29

30

31

32

33

34

35

36

37

38

39

40

41

42

43

44

45

46

47

48

49

50

51

52

53

54

55

56

57

58

59

60

61

62

63

64

65

66

总的来说driver\_attach函数就是遍历bus\_type的klist\_devices中每个device，调用bus\_type的match函数（本例usb\_device\_match）进行驱动和设备的匹配

really\_probe函数实现如下：

static int really\_probe(struct device \*dev, struct device\_driver \*drv)

{

int ret = 0;

atomic\_inc(&probe\_count);

pr\_debug("%s: Probing driver %s with device %s\n",

drv->bus->name, drv->name, dev->bus\_id);

WARN\_ON(!list\_empty(&dev->devres\_head));

dev->driver = drv;

//设置sturct device->driver指针

if (driver\_sysfs\_add(dev)) {

//sysfs建立对象和驱动链接

printk(KERN\_ERR "%s: driver\_sysfs\_add(%s) failed\n",

\_\_FUNCTION\_\_, dev->bus\_id);

goto probe\_failed;

}

if (dev->bus->probe) {

//如果总线存在probe函数，则调用总线的probe函数

ret = dev->bus->probe(dev);

if (ret)

goto probe\_failed;

} else if (drv->probe) {

//如果总线没有probe函数，但是驱动有probe函数，就调用驱动的probe函数

ret = drv->probe(dev);

if (ret)

goto probe\_failed;

}

//本例中usb\_bus\_type没有probe函数，那就是调用驱动的probe函数hub\_probe

driver\_bound(dev);

//将struct device添加到匹配的struct device\_driver->klist\_devices中

ret = 1;

pr\_debug("%s: Bound Device %s to Driver %s\n",

drv->bus->name, dev->bus\_id, drv->name);

goto done;

probe\_failed:

devres\_release\_all(dev);

driver\_sysfs\_remove(dev);

dev->driver = NULL;

if (ret != -ENODEV && ret != -ENXIO) {

/\* driver matched but the probe failed \*/

printk(KERN\_WARNING

"%s: probe of %s failed with error %d\n",

drv->name, dev->bus\_id, ret);

}

/\*

\* Ignore errors returned by ->probe so that the next driver can try

\* its luck.

\*/

ret = 0;

done:

atomic\_dec(&probe\_count);

wake\_up(&probe\_waitqueue);

return ret;

}

1

2

3

4

5

6

7

8

9

10

11

12

13

14

15

16

17

18

19

20

21

22

23

24

25

26

27

28

29

30

31

32

33

34

35

36

37

38

39

40

41

42

43

44

45

46

47

48

49

50

51

52

53

54

55

56

57

58

59

hub\_thread函数实现如下：

static int hub\_thread(void \*\_\_unused)

{

set\_freezable();

do {

hub\_events();

wait\_event\_freezable(khubd\_wait,

!list\_empty(&hub\_event\_list) ||

kthread\_should\_stop());

} while (!kthread\_should\_stop() || !list\_empty(&hub\_event\_list));

pr\_debug("%s: khubd exiting\n", usbcore\_name);

return 0;

}

1

2

3

4

5

6

7

8

9

10

11

12

13

这里不进行深入分析，总述功能：其中的hub\_events()函数处理hub上面发生的事件。wait\_event\_freezable会在那边等待hub总线上面的事件，如果一直没有事件则函数不返回。hub\_thread第一次执行时会第一次调用hub\_events，发现无事可做，执行空，退出去执行wait\_event\_freezable函数，该函数等待usb总线上的事件，如usb设备插入、拔出等。没有事件不返回

————————————————

版权声明：本文为CSDN博主「HJL\_DLUT」的原创文章，遵循 CC 4.0 BY-SA 版权协议，转载请附上原文出处链接及本声明。

原文链接：<https://blog.csdn.net/HJL_DLUT/article/details/80577921>

### 3.6.7. usb\_register\_device\_driver 注册USB设备驱动（driver\_register）

其中usb\_generic\_driver之前已经简述过了，是一个来者不拒的设备驱动

usb\_register\_device\_driver(&usb\_generic\_driver, THIS\_MODULE);

struct usb\_device\_driver usb\_generic\_driver = {

.name = "usb",

.probe = generic\_probe,

.disconnect = generic\_disconnect,

#ifdef CONFIG\_PM

.suspend = generic\_suspend,

.resume = generic\_resume,

#endif

.supports\_autosuspend = 1,

};

1

2

3

4

5

6

7

8

9

10

11

12

下面看一下usb\_register\_device\_driver的实现，可以类比上文中usb\_register\_driver的实现

int usb\_register\_device\_driver(struct usb\_device\_driver \*new\_udriver,

struct module \*owner)

{

int retval = 0;

if (usb\_disabled())

return -ENODEV;

new\_udriver->drvwrap.for\_devices = 1;

new\_udriver->drvwrap.driver.name = (char \*) new\_udriver->name;

new\_udriver->drvwrap.driver.bus = &usb\_bus\_type;

new\_udriver->drvwrap.driver.probe = usb\_probe\_device;

new\_udriver->drvwrap.driver.remove = usb\_unbind\_device;

new\_udriver->drvwrap.driver.owner = owner;

//初始化usb\_device\_driver中成员，主要是struct usbdrv\_wrap的driver成员

retval = driver\_register(&new\_udriver->drvwrap.driver);

//调用通用的驱动注册函数，实现见上文3.3.6

if (!retval) {

pr\_info("%s: registered new device driver %s\n",

usbcore\_name, new\_udriver->name);

usbfs\_update\_special();

} else {

printk(KERN\_ERR "%s: error %d registering device "

" driver %s\n",

usbcore\_name, retval, new\_udriver->name);

}

return retval;

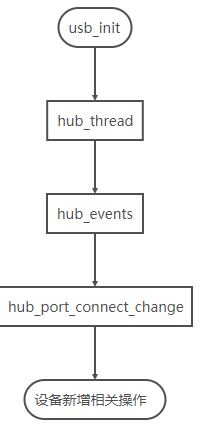
}

# 4. 设备生命线

## 4.1. Hub监听处理TODO

插到Hub到拔出。当USB设备连接到Hub某端口上时，Hub检测到有设备连接后，会分配一个struct usb\_device结构并初始化，调用设备模型提供的接口将设备添加到USB总线的设备列表中，然后USB总线会遍历驱动列表中的每个驱动，调用自己的match函数看是否和设备或接口匹配。

Hub检测过程如下



其中hub\_port\_connect\_change关键代码如下：

for (i = 0; i < SET\_CONFIG\_TRIES; i++) {

struct usb\_device \*udev;

/\* reallocate for each attempt, since references

\* to the previous one can escape in various ways

\*/

udev = usb\_alloc\_dev(hdev, hdev->bus, port1);

//Hub检测到端口有设备连接后，会调用core的usb\_alloc\_dev函数

//为struct usb\_device结构的对象申请内容，详见4.1.1

if (!udev) {

dev\_err (hub\_dev,

"couldn't allocate port %d usb\_device\n",

port1);

goto done;

}

usb\_set\_device\_state(udev, USB\_STATE\_POWERED);

udev->speed = USB\_SPEED\_UNKNOWN;

udev->bus\_mA = hub->mA\_per\_port;

udev->level = hdev->level + 1;

//完成struct usb\_device结构体创建后，hub会进行后续初始化工作

//将设备状态设置为powered（加电状态）

//将speed暂时设置为USB\_SPEED\_UNKNOWN，核心就是成员属性设置

//设备level设置为Hub level+1

//为设备能够从Hub那里获得的电流赋值，为了保证通信

/\* set the address \*/

choose\_address(udev);

if (udev->devnum <= 0) {

status = -ENOTCONN; /\* Don't retry \*/

goto loop;

}

//Hub会为设备在总线上选择独一无二的地址，设置usb\_device->devnum

/\* reset and get descriptor \*/

status = hub\_port\_init(hub, udev, port1, i);

//复位、设置地址、并获得设备描述符，详见下文

if (status < 0)

goto loop;

/\* consecutive bus-powered hubs aren't reliable; they can

\* violate the voltage drop budget. if the new child has

\* a "powered" LED, users should notice we didn't enable it

\* (without reading syslog), even without per-port LEDs

\* on the parent.

\*/

if (udev->descriptor.bDeviceClass == USB\_CLASS\_HUB

&& udev->bus\_mA <= 100) {

//如果设备类型是USB\_CLASS\_HUB进行的特殊处理，暂不关心

u16 devstat;

status = usb\_get\_status(udev, USB\_RECIP\_DEVICE, 0,

&devstat);

if (status < 2) {

dev\_dbg(&udev->dev, "get status %d ?\n", status);

goto loop\_disable;

}

le16\_to\_cpus(&devstat);

if ((devstat & (1 << USB\_DEVICE\_SELF\_POWERED)) == 0) {

dev\_err(&udev->dev,

"can't connect bus-powered hub "

"to this port\n");

if (hub->has\_indicators) {

hub->indicator[port1-1] =

INDICATOR\_AMBER\_BLINK;

schedule\_delayed\_work (&hub->leds, 0);

}

status = -ENOTCONN; /\* Don't retry \*/

goto loop\_disable;

}

}

/\* check for devices running slower than they could \*/

if (le16\_to\_cpu(udev->descriptor.bcdUSB) >= 0x0200

&& udev->speed == USB\_SPEED\_FULL

&& highspeed\_hubs != 0)

check\_highspeed (hub, udev, port1);

/\* Store the parent's children[] pointer. At this point

\* udev becomes globally accessible, although presumably

\* no one will look at it until hdev is unlocked.

\*/

status = 0;

/\* We mustn't add new devices if the parent hub has

\* been disconnected; we would race with the

\* recursively\_mark\_NOTATTACHED() routine.

\*/

spin\_lock\_irq(&device\_state\_lock);

if (hdev->state == USB\_STATE\_NOTATTACHED)

status = -ENOTCONN;

else

hdev->children[port1-1] = udev;

spin\_unlock\_irq(&device\_state\_lock);

/\* Run it through the hoops (find a driver, etc) \*/

if (!status) {

status = usb\_new\_device(udev);

//核心调用，初始化设备配置，详见4.1.4

if (status) {

spin\_lock\_irq(&device\_state\_lock);

hdev->children[port1-1] = NULL;

spin\_unlock\_irq(&device\_state\_lock);

}

}

if (status)

goto loop\_disable;

status = hub\_power\_remaining(hub);

if (status)

dev\_dbg(hub\_dev, "%dmA power budget left\n", status);

return;

loop\_disable:

hub\_port\_disable(hub, port1, 1);

loop:

ep0\_reinit(udev);

release\_address(udev);

usb\_put\_dev(udev);

if ((status == -ENOTCONN) || (status == -ENOTSUPP))

break;

}

1

设备要想从Powered状态发展到下一个Default状态，必须收到一个复位信号，并成功复位。Hub会复位设备，成功后，设备进入Default状态。

复位进入Default状态后，Hub也会获得设备真正的速度，依据速度，能过知道端口0一次能够处理的最大数据长度。

下面Hub使用Core中定义的usb\_control\_msg函数给设备发送SET\_ADDRESS请求，设备就进入Address状态了，设备的address就是上表中的Devnum

————————————————

### 4.1.1 usb\_alloc\_dev-USB设备对象创建

### 4.1.2. hub\_port\_init-设备复位、设备Address、设备描述符获取

### 4.1.3. usb\_control\_msg发送urb（端点管道）-TODO

### 4.1.4. usb\_new\_device-配置设备、设备注册、驱动匹配

#### 4.1.4.1 usb\_configure\_device

#### 4.1.4.1 device\_register->device\_add

## 4.2. 设备与驱动的match与probe

### 4.2.1. 设备驱动usb\_generic\_driver & usb\_bus\_type->match

### 4.2.2. 设备驱动usb\_generic\_driver中device\_driver->probe(usb\_probe\_device)

### 4.2.3. 设备驱动usb\_generic\_driver->probe(generic\_probe)

### 4.2.4. 接口驱动中struct device\_driver->probe(usb\_probe\_interface)

————————————————

版权声明：本文为CSDN博主「HJL\_DLUT」的原创文章，遵循 CC 4.0 BY-SA 版权协议，转载请附上原文出处链接及本声明。

原文链接：https://blog.csdn.net/HJL\_DLUT/article/details/80634630

# 参考

USB从入门到精通-1-Linux那些事儿之USB学习与扩展【重点参考】

<https://blog.csdn.net/HJL_DLUT/article/details/80577921>

<https://blog.csdn.net/HJL_DLUT/article/details/80634630>

# 源码分析PDF446

# drivers/usb/core

### 初始化usb\_init

subsys\_initcall(usb\_init);  
module\_exit(usb\_exit);  
MODULE\_LICENSE(**"GPL"**);

subsys\_initcall，它是一个宏，我们可以把它理解为module\_init，只不过因为这部分代码比较核心，开发者们把它看作一个子系统，而不仅仅是一个模块，这也很好理解，usbcore这个模块它代表的不是某一个设备，而是所有usb设备赖以生存的模块，Linux中，像这样一个类别的设备驱动被归结为一个子系统。比如pci子系统，比如scsi子系统，基本上，drivers/目录下面第一层的每个目录都算一个子系统，因为它们代表了一类设备。subsys\_initcall(usb\_init)的意思就是告诉我们usb\_init是我们真正的初始化函数，而usb\_exit()将是整个usb子系统的结束时的清理函数，于是我们就从usb\_init开始看起。

**static int** \_\_init usb\_init(**void**)

表明这个函数仅在初始化期间使用，在模块被装载之后，它占用的资源就会释放掉

#### Nousb

**if** (usb\_disabled()) {  
 pr\_info(**"%s: USB support disabled\n"**, usbcore\_name);  
 **return** 0;  
}

这里的nousb 是用来让我们在启动内核时通过内核参数去掉USB 子系统的，Linux 社会是一个很人性化的世界，它不会去逼迫我们接受USB，一切都只关乎我们自己的需要。不过我想我们一般是不会去指定nousb 的吧，毕竟它那么的讨人喜爱。如果你真的指定了nousb，那它就只会幽怨地说一句“USB support disabled”，然后退出usb\_init

**int** usb\_disabled(**void**)  
{  
 **return** nousb;  
}

pr\_info 只是一个打印信息的可变参数宏，即printk 的变体，在include/linux/kernel.h

中定义

#### 初始化

usb\_acpi\_register();  
*//注册USB 总线  
//只有成功地将USB 总线子系统注册到系统中  
//我们才可以向这个总线添加USB 设备*retval = bus\_register(&usb\_bus\_type);  
  
retval = bus\_register\_notifier(&usb\_bus\_type, &usb\_bus\_nb);  
retval = usb\_major\_init();  
retval = usb\_register(&usbfs\_driver);  
retval = usb\_devio\_init();  
retval = usb\_hub\_init();  
*//注册USB 设备驱动，看清楚了，是USB device driver 而不是USB driver*retval = usb\_register\_device\_driver(&usb\_generic\_driver, THIS\_MODULE);

#### 初始化usb\_bus\_type

Linux 设备模型中的总线落实在USB 子系统里就是usb\_bus\_type，它在usb\_init

注册bus\_register(&usb\_bus\_type)，在drivers/usb/core/driver.c 文件中定义：

**struct** bus\_type usb\_bus\_type = {  
 .name = **"usb"**,  
 .match = usb\_device\_match,  
 .uevent = usb\_uevent,  
};

name 自然就是USB 总线的绰号。match 这个函数指针就比较有意思了，它充当了一个红

娘的角色，在总线的设备和驱动之间“牵线搭桥”，但明显这里match 的条件不是那么苛刻，要

更为实际一些。match 指向了函数usb\_device\_match。

**static int** usb\_device\_match(**struct** device \*dev, **struct** device\_driver \*drv)  
{  
 */\* devices and interfaces are handled separately \*/* **if** (is\_usb\_device(dev)) {  
  
 */\* interface drivers never match devices \*/* **if** (!is\_usb\_device\_driver(drv))  
 **return** 0;  
  
 */\** ***TODO: Add real matching code*** *\*/* **return** 1;  
  
 } **else if** (is\_usb\_interface(dev)) {  
 **struct** usb\_interface \*intf;  
 **struct** usb\_driver \*usb\_drv;  
 **const struct** usb\_device\_id \*id;  
  
 */\* device drivers never match interfaces \*/* **if** (is\_usb\_device\_driver(drv))  
 **return** 0;  
  
 intf = to\_usb\_interface(dev);  
 usb\_drv = to\_usb\_driver(drv);  
  
 id = usb\_match\_id(intf, usb\_drv->id\_table);  
 **if** (id)  
 **return** 1;  
  
 id = usb\_match\_dynamic\_id(intf, usb\_drv);  
 **if** (id)  
 **return** 1;  
 }  
  
 **return** 0;  
}

这两个参数我们都已经很熟悉了，对应的就是总线两条链表里的设备和驱动。总线上有新设备或新的驱动添加时，这个函数总是会被调用，如果指定的驱动程序能够处理指定的设备，也就是匹配成功，函数返回0

### 设备模型

设备是通过总线连到计算机上的，而且还需要对应的驱动才能用，可是总线是如何发现设备的？设备又是如何和驱动对应，它们的关系如何，这些疑问的中心思想或中心词汇就是总线、设备和驱动

总线、设备、驱动，也就是bus、device、driver，既然是“名角”，在内核中都会有它们自己专属的结构，在include/linux/device.h 中定义：



struct bus\_type 中有成员struct kset

drivers 和struct kset devices

同时struct device 中有两个成员struct bus\_type \* bus 和struct device\_driver \*driver

struct device\_driver 中有两个成员struct bus\_type \* bus 和struct klist klist\_devices

我们可以知道struct device 中的bus表示这个设备连到哪个总线上，driver 表示这个设备的

驱动是什么。

struct device\_driver 中的bus 表示这个驱动属于哪个总线，klist\_devices 表示这个

驱动都支持哪些设备，因为这里device 是复数，又是list，更因为一个驱动可以支持多个设备，

而一个设备只能绑定一个驱动。当然，struct bus\_type 中的drivers 和devices 分别表示了这个总

线拥有哪些设备和哪些驱动

我们还需要看一看什么是klist 和kset。还有上面device 和driver 结构中出现的kobject 结

构是什么？我可以肯定地告诉你，kobject 和kset 都是Linux 设备模型中最基本的元素，总线、

设备、驱动是西瓜，kobjcet、klist 是种瓜的人，它们存在的意义在于把总线、设备和驱动这样的对象连接到设备模型上

整个Linux 的设备模型是一个OO 的体系结构，总线、设备和驱

动都是其中鲜活存在的对象，kobject 是它们的基类，所实现的只是一些公共的接口，kset 是同

种类型kobject 对象的集合，也可以说是对象的容器。只是因为C 语言里不可能会有C++语言

里类的class 继承、组合等的概念，只有通过kobject 嵌入到对象结构中来实现。这样，内核使

用kobject 将各个对象连接起来组成了一个分层的结构体系。kobject 结构中包含了parent 成员，

指向了另一个kobject 结构，也就是这个分层结构的上一层结点。而kset 是通过链表来实现的，

这样就可以明白，struct bus\_type 结构中的成员drivers 和devices 表示了一条总线拥有两条链表，

一条是设备链表，一条是驱动链表。我们知道了总线对应的数据结构，就可以找到这条总线关

联了多少设备，又有哪些驱动来支持这类设备。

那么klist 呢？其实它就包含了一个链表和一个自旋锁，我们暂且把它看成链表也无妨。本

来在2.6.11 内核中，struct device\_driver 结构的devices 成员就是一个链表类型。

那么总线、设备和驱动之间是如何和谐共处的呢？先说一说总线中的那两条链表是怎么形

成的。内核要求每次出现一个设备就要向总线汇报，或者说注册，每次出现一个驱动，也要向

总线汇报，或者说注册。比如系统初始化时，会扫描连接了哪些设备，并为每一个设备建立起

一个struct device 的变量，每一次有一个驱动程序，就要准备一个struct device\_driver 结构的变

量。把这些变量统统加入相应的链表，device 插入devices 链表，driver 插入drivers 链表。这

样通过总线就能找到每一个设备，每一个驱动。然而，假如计算机里只有设备却没有对应的驱

动，那么设备无法工作。反过来，倘若只有驱动却没有设备，驱动也起不了任何作用。

链表里的设备和驱动又是如何联系的？先有设备还是先有驱动？很久很久以前，先有的是设备，每一个要用的设备在计算机启动之前就已经插好了，插放在它应该在的位置上，然后计算机启动，操作系统开始初始化，总线

开始扫描设备，每找到一个设备，就为其申请一个struct device 结构，并且挂入总线中的devices链表中来。然后每一个驱动程序开始初始化，开始注册其struct device\_driver 结构，然后去总线的devices 链表中去寻找（遍历），去寻找每一个还没有绑定驱动的设备，即struct device 中的struct device\_driver 指针仍为空的设备，然后它会去观察这种设备的特征，看是否是它所支持的

设备，如果是，那么调用一个叫做device\_bind\_driver 的函数，把struct device 中的struct device\_driver driver 指向这个驱动，而struct device\_driver

driver 把struct device 加入它的那张struct klist klist\_devices 链表中来。就这样，bus、device 和

driver，这三者之间或者说他们中的两两之间，就给联系上了。知道其中之一，就能找到另外两个

#### 热插拔

但现在情况变了，出现了一种新的名词，叫热插拔。此时设备可以在计算机启动以后再插

入或者拔出计算机了。因此，很难再说是先有设备还是先有驱动了，因为都有可能。设备可以

在任何时刻出现，而驱动也可以在任何时刻被加载，所以，现在的情况就是，每当一个structdevice 诞生，它就会去bus 的drivers 链表中寻找自己的另一半。反之，每当一个一个struct

device\_driver 诞生，它就去bus 的devices 链表中寻找它的那些设备。如果找到了合适的，那么

和之前那种情况一样，调用device\_bind\_driver 绑定好。如果找不到，没有关系

#### 接口是设备的接口

设备可以有多个接口，每个接口代表一个功能，每个接口对应着一个驱动。Linux 设备模

型中的device 落实在USB 子系统，成了两个结构：一个是struct usb\_device，一个是struct

usb\_interface。

一个USB 键盘，上面带一个扬声器，因此有两个接口，那肯定得要两个驱动程序：一个是键盘驱动程序，一个是音频流驱动程序。“道”上的兄弟喜欢把这样两个整合在一起的东西叫做一个设备，我们用interface 来区分这两者。于是有了这里提到的数据结构，struct usb\_interface

如果你还是不明白什么是配置什么是设置的话，那就直接用它们大小关系来理解好了，毕

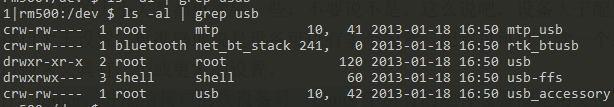
竟大家对互相之间的大小关系都更敏感一些，不要说不是。这么说吧，设备大于配置，配置大

于接口，接口大于设置。更准确地说是设备可以有多个配置，配置里可以包含一个或更多的接

口，而接口通常又具有一个或更多的设置。

minor，分配给接口的次设备号。Linux 下所有的硬件设备都是用文件来表示的，

俗称“设备文件”，在/dev 目录下面，为了显示自己并不是普通的文件，它们都会有一个主设备号和次设备号



一般来说，主设备号表明了设备的种类，也表明了设备对应着哪个驱动程序，而次设备号则是因为一个驱动程序要支持多个设备而

为了让驱动程序区分它们而设置的。也就是说，主设备号用来帮你找到对应的驱动程序，次设备号决定你的驱动对哪个设备进行操作

设备要想在Linux 里分得一个主设备号，有一个立足之地，也并不是那么容易的。主设备

号虽说不是什么特别稀缺的资源，但还是需要设备先在驱动里提出申请，获得系统的批准后才

能拥有一个。因为一部分的主设备号已经被静态地预先指定给了许多常见的设备，申请时要避

开它们。这些已经被分配掉的主设备号都列在Documentation/devices.txt 文件中。当然，如果你

是用动态分配的形式，就可以不去理会这些，直接让系统为你做主，替你选择一个即可。

很显然，USB 设备是很常见的，Linux 理应为它预留了一个主设备号。下面看一看

include/linux/usb.h 文件

#define USB\_MAJOR 180  
#define USB\_DEVICE\_MAJOR 189

usbfs 提供了在用户空间直接访问USB 硬件设备的接口，

它需要内核的大力支持，usbfs\_driver 就是用来完成这个光荣任务的。我们可以去usb\_devio\_init 函数中看一看，它在drivers/usb/devio.c 文件中定义：

int \_\_init usb\_devio\_init(void)  
{  
 int retval;  
  
 retval = register\_chrdev\_region(USB\_DEVICE\_DEV, USB\_DEVICE\_MAX,  
 "usb\_device");  
 if (retval) {  
 printk(KERN\_ERR "Unable to register minors for usb\_device\n");  
 goto out;  
 }  
 cdev\_init(&usb\_device\_cdev, &usbdev\_file\_operations);  
 retval = cdev\_add(&usb\_device\_cdev, USB\_DEVICE\_DEV, USB\_DEVICE\_MAX);  
 if (retval) {  
 printk(KERN\_ERR "Unable to get usb\_device major %d\n",  
 USB\_DEVICE\_MAJOR);  
 goto error\_cdev;  
 }  
 usb\_register\_notify(&usbdev\_nb);

register\_chrdev\_region 函数获得了设备usb\_device 对应的设备编号，设备usb\_device 对应

的驱动当然就是usbfs\_driver，参数USB\_DEVICE\_DEV 也在同一个文件中有定义

#define USB\_DEVICE\_DEV MKDEV(USB\_DEVICE\_MAJOR, 0)

终于再次见到了USB\_DEVICE\_MAJOR，也终于明白它是为了usbfs 而生的，为了让广大

人民群众能够在用户空间直接和USB 设备通信而生的。因此，它并不是我们所要寻找的。

2|rm500:/ # cat proc/devices

Character devices:

1 mem

4 ttyS

5 /dev/tty

5 /dev/console

rm500:/ # cat proc/devices | grep usb

180 usb

189 usb\_device

241 rtk\_btusb

/proc/devices 文件里显示了所有当前系统里已经分配出去的主设备号，当然上面只是列

出了字符设备

比如，

usb键盘关联了input子系统，驱动对应drivers/hid/usbhid目录下的usbkbd.c文件，在它

的probe函数里可以看到使用了input\_register\_device来注册一个输入设备。

说完了设备号，回到struct usb\_interface 的151 行，condition 字段表示接口和驱动的

绑定状态，enum usb\_interface\_condition 类型，在include/linux/usb.h 里定义

**enum** usb\_interface\_condition {  
 USB\_INTERFACE\_UNBOUND = 0,  
 USB\_INTERFACE\_BINDING,  
 USB\_INTERFACE\_BOUND,  
 USB\_INTERFACE\_UNBINDING,  
};

前面说linux设备模型的时候说了，设备和驱动是相生相依的关系，

，所有的 usb 设备都必须支 持挂起状态，就是说为了达到节电的目的，当设备在指定的时间内，3ms 吧，如果没有发 生总线传输，就要进入挂起状态。当它收到一个 non-idle 的信号时，就会被唤醒。152 行 is\_active 表示接口是不是处于挂起状态。153 行 needs\_remote\_wakeup 表示是否需要 打开远程唤醒功能。远程唤醒允许挂起的设备给主机发信号，通知主机它将从挂起状态恢复， 注意如果此时主机处于挂起状态，就会唤醒主机，不然主机仍然在睡着，设备自个醒过来干 吗用。协议里并没有要求 USB 设备一定要实现远程唤醒的功能，即使实现了，从主机这边 儿也可以打开或关闭它。157 行 pm\_usage\_cnt，pm 就是电源管理，usage\_cnt 就是使 用计数，当它为 0 时，接口允许 autosuspend。什么叫 autosuspend

接下来就剩下 155 行的struct device dev和 156 行的struct device \*usb\_dev，看到 struct device没，它们就是linux设备模型里的device嵌在这儿的对象，我们的心中要时时 有个模型。不过这么想当然是不正确的，两个里面只有dev才是模型里的device嵌在这儿的， usb\_dev则不是。当接口使用 USB\_MAJOR作为主设备号时，usb\_dev才会用到，你找遍 整个内核，也只在usb\_register\_dev和usb\_deregister\_dev两个函数里能够看到它， usb\_dev指向的就是usb\_register\_dev函数里创建的usb class device。

### desc，接口的描述符

struct usb\_interface 里表示接口设置的 struct， usb\_host\_interface同样在 include/linux/usb.h 文件里定义。

desc，接口的描述符。什么叫描述符

比如 USB 的描述符。 实际上，usb 的描述符是一个带有预定义格式的数据结构，里面保存了 usb 设备的各种属 性还有相关信息，姓甚名谁啊，哪儿生产的啊等等，我们可以通过向设备请求获得它们的内 容来深刻的了解感知一个 usb 设备。主要有四种 usb 描述符，设备描述符，配置描述符， 接口描述符和端点描述符，协议里规定一个 usb 设备是必须支持这四大描述符的，当然也 有其它一些描述符来让设备可以显得个性些，但这四大描述符是一个都不能少的。

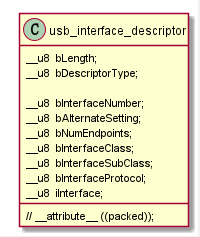
这些描述符放哪儿？当然是在设备里。usb 设备里都会有一个叫 EEPROM 的东东，没错，就是放在它那儿，它就是用来存储设备 本身信息的。如果你的脑海里还残存着一些大学里的美好时光的话，应该还会记得 EEPROM，就是电可擦写的可编程ROM，它与Flash虽说都是要电擦除的，但它可以按字节擦除，Flash 只能一次擦除一个 block，所以如果要改动比较少的数据的话，使用它还是比较合适的，但 是世界上没有完美的东西，此物成本相对 Flash 比较高，所以一般来说 usb 设备里只拿它 来存储一些本身特有的信息，要想存储数据，还是用 Flash 吧。

具体到接口描述符，它当然就是描述接口本身的信息的。一个接口可以有多个设置，使用不 同的设置，描述接口的信息会有些不同，所以接口描述符并没有放在 struct usb\_interface 结构里，而是放在表示接口设置的 struct usb\_host\_interface 结构里。定义在 include/linux/usb/ch9.h 文件里

又看到了 \_\_attribute\_\_，不过这里改头换面成了 \_\_attribute\_\_ ((packed))，意思就是 告诉编译器，这个结构的元素都是 1 字节对齐的，不要再添加填充位了。因为这个结构和 spec里的Table 9.12 是完全一致的，包括字段的长度，如果不给编译器这么个暗示，编译 器就会依据你平台的类型在结构的每个元素之间添加一定的填充位，如果你拿这个添加了填 充位的结构去向设备请求描述符，你想想会是什么结果。

296 行，bLength，描述符的字节长度。协议里规定，每个描述符必须以一个字节打头来 表明描述符的长度。那可以扳着指头数一下，接口描述符的 bLength 应该是 9，两个巴掌 就数完了，没错，ch9.h 文件里紧挨着接口描述符的定义就定义了这个长度

include/uapi/linux/usb/ch9.h



又看到了 \_\_attribute\_\_，不过这里改头换面成了 \_\_attribute\_\_ ((packed))，意思就是 告诉编译器，这个结构的元素都是 1 字节对齐的，不要再添加填充位了。因为这个结构和 spec里的Table 9.12 是完全一致的，包括字段的长度，如果不给编译器这么个暗示，编译 器就会依据你平台的类型在结构的每个元素之间添加一定的填充位，如果你拿这个添加了填 充位的结构去向设备请求描述符