# Linux驱动\_USB

## 1. USB基础知识

### 1.1 USB传输线结构

一条USB的传输线分别由地线、电源线、D+、D-四条线构成，D+和D-是差分输入线(抗干扰)，它使用的是3.3V的电压，而电源线和地线可向设备提供5V电压，最大电流为500MA。OTG的做法就是增来一个ID pin来判断设备是接入设备的是主还是从。vbus主要是供电，D+/D-则是用来传输数据，就是我们前面所讲的主设备和从设备间唯一的一条铁路。

### 1.2 USB版本

USB1.0版本速度1.5Mbps（低速USB）USB1.1版本速度12Mbps（全速USB） USB2.0版本速度480Mbps（高速USB）。

OHCI、UHCI都是USB1.1的接口标准，而EHCI是对应USB2.0的接口标准，最新的xHCI是USB3.0的接口标准。

(1)OHCI（Open Host Controller Interface）是支持USB1.1的标准，但它不仅仅是针对USB，还支持其他的一些接口，比如它还支持Apple的火线（Firewire，IEEE 1394）接口。与UHCI相比，OHCI的硬件复杂，硬件做的事情更多，所以实现对应的软件驱动的任务，就相对较简单。主要用于非x86的USB，如扩展卡、嵌入式开发板的USB主控。  
(2)UHCI（Universal Host Controller Interface），是Intel主导的对USB1.0、1.1的接口标准，与OHCI不兼容。UHCI的软件驱动的任务重，需要做得比较复杂，但可以使用较便宜、较简单的硬件的USB控制器。Intel和VIA使用UHCI，而其余的硬件提供商使用OHCI。  
(3)EHCI（Enhanced Host Controller Interface），是Intel主导的USB2.0的接口标准。EHCI仅提供USB2.0的高速功能，而依靠UHCI或OHCI来提供对全速（full-speed）或低速（low-speed）设备的支持。  
(4)xHCI（eXtensible Host Controller Interface），是最新最火的USB3.0的接口标准，它在速度、节能、虚拟化等方面都比前面3中有了较大的提高。xHCI支持所有种类速度的USB设备（USB 3.0 SuperSpeed, USB 2.0 Low-, Full-, and High-speed, USB 1.1 Low- and Full-speed）。xHCI的目的是为了替换前面3种(UHCI/OHCI/EHCI)。

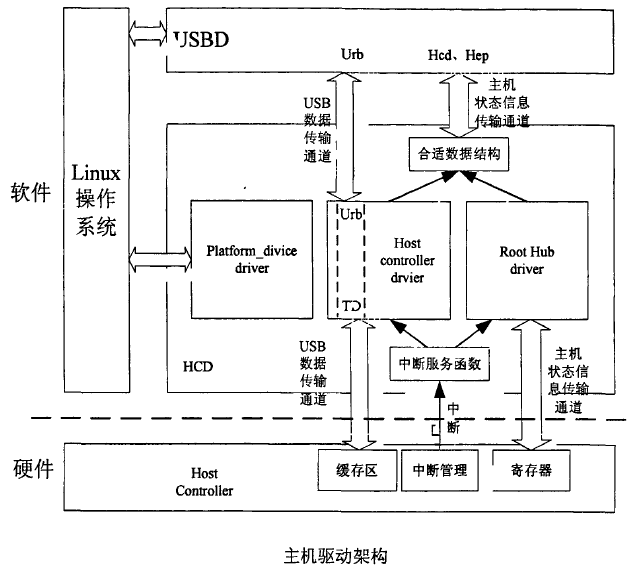
### 1.3 USB热插拔原理

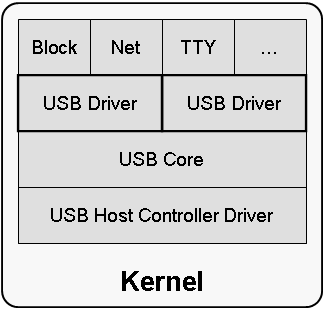
USB主机是如何检测到设备的插入的呢？首先，在USB集线器的每个下游端口的D+和D-上，分别接了一个15K欧姆的下拉电阻到地。这样，在集线器的端口悬空时，就被这两个下拉电阻拉到了低电平。而在USB设备端，在D+或者D-上接了1.5K欧姆上拉电阻。对于全速和高速设备，上拉电阻是接在D+上；而低速设备则是上拉电阻接在D-上。这样，当设备插入到集线器时，由1.5K的上拉电阻和15K的下拉电阻分压，结果就将差分数据线中的一条拉高了。集线器检测到这个状态后，它就报告给USB主控制器（或者通过它上一层的集线器报告给USB主控制器），这样就检测到设备的插入了。USB高速设备先是被识别为全速设备，然后通过HOST和DEVICE两者之间的确认，再切换到高速模式的。在高速模式下，是电流传输模式，这时将D+上的上拉电阻断开。

### 1.4 USB主机控制器

USB主机控制器属于南桥芯片的一部分，通过PCI总线和处理器通信。USB主机控制器分为UHCI（英特尔提出）、OHCI（康柏和微软提出）、EHCI。其中OHCI驱动程序用来为非PC系统上以及带有SiS和ALi芯片组的PC主办上的USB芯片提供支持。UHCI驱动程序多用来为大多数其他PC主板（包括Intel和Via）上的USB芯片提供支持。ENCI兼容OHCI和UHCI。UHCI的硬件线路比OHCI简单，所以成本较低，但需要较复杂的驱动程序，CPU负荷稍重。主机控制器驱动程序完成的功能主要包括：解析和维护URB，根据不同的端点进行分类缓存URB；负责不同USB传输类型的调度工作；负责USB数据的实际传输工作；实现虚拟跟HUB的功能。

下面为USB主机驱动架构图：





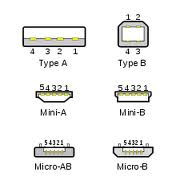
(1)USB 驱动都是跨kernel子系统的，因为最终USB设备是要通过BLCOCK或CHAR设备的方式呈现给我们的，所以我们从图中看到USB Driver之上还有一层。

(2)USB driver利用USB Core提供的API来简单优雅的完成驱动工作，这里USB Core抽象了复杂的USB协议。

(3)主机控制器驱动位于USB软件的最下层，提供主机控制器硬件的抽象，隐藏硬件的细节，在主机控制器之下是物理的USB及所有与之连接的USB设备。主机控制器驱动只和USB Core进行关联，USB Core将用户的请求映射到相关的主机控制器驱动，从而使用户无需去访问主机控制器。

(4)USB Core和USB主机控制器驱动就构成了我们的USB子系统，USB Core负责实现一些核心的功能，例如协议之类，提供一个用于访问和控制USB硬件的接口，使设备驱动不用去考虑系统当前使用哪种主机控制器。自从有了USB子系统，写USB驱动的时候，只需要调用USB Core export的接口，就几乎能完成所有工作。

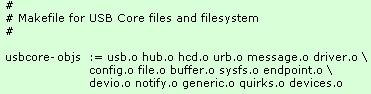
### 1.5 USB物理接口类型



## 2.USB主机驱动

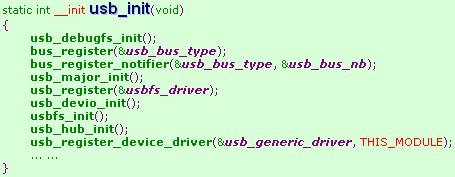
### 2.1 USB HOST驱动整体分析

#### 2.1.1 USB core

USB core源码位于./drivers/usb/core，其中的Makefile摘要如下：

usbcore这个模块代表的不是某一个设备，而是所有USB设备赖以生存的模块，它就是USB子系统。

./drivers/usb/core/usb.c里实现了初始化，伪代码如下：



usbcore注册了USB总线，USB文件系统，USB Hub以及USB的设备驱动usb generic driver等。

#### 2.1.2 USB总线

注册USB总线通过bus\_register(&usb\_bus\_type);

struct bus\_type usb\_bus\_type **=** **{**

**.**name **=** "usb"**,**

**.**match **=** usb\_device\_match**,** // 这是个很重要的函数，用来匹配USB设备和驱动。

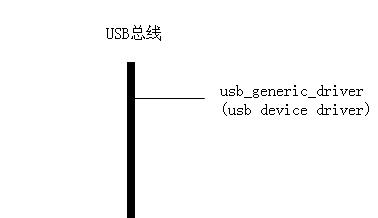
**.**uevent **=** usb\_uevent**,**

**.**pm **=** **&**usb\_bus\_pm\_ops**,**

**};**

#### 2.1.3 USB设备和驱动匹配的全过程

**-> step 1 - usb device driver**

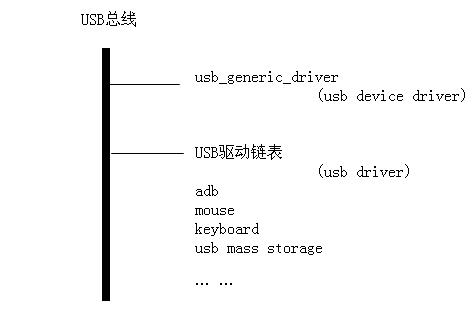


USB子系统初始化的时候就会注册usb\_generic\_driver, 它的结构体类型是usb\_device\_driver，它是USB世界里唯一的一个USB设备驱动，区别于struct usb\_driver USB驱动。

USB设备驱动(usb device driver)就只有一个，即usb\_generice\_driver这个对象，所有USB设备都要绑定到usb\_generic\_driver上，它的使命可以概括为：为USB设备选择一个合适的配置，让设备进入configured状态。

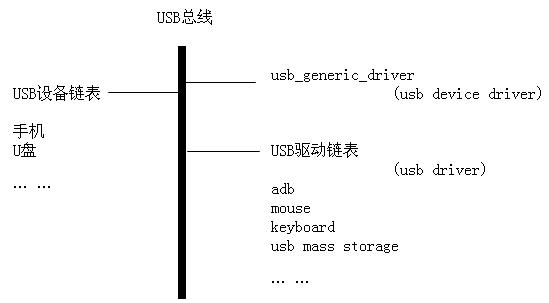
USB驱动(usb driver)就是USB设备的接口驱动程序，比如adb驱动程序，u盘驱动程序，鼠标驱动程序等等。

**-> step 2 - usb driver**



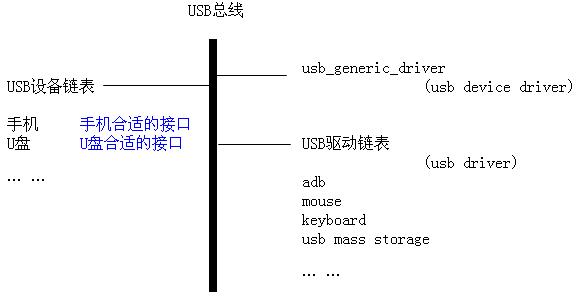
Linux启动时注册USB驱动，在xxx\_init()里通过**usb\_register()**将USB驱动提交个设备模型，添加到USB总线的驱动链表里。

**-> step 3 - usb device**



USB设备连接在Hub上，Hub检测到有设备连接进来，为设备分配一个struct usb\_device结构体对象，并将设备添加到USB总线的设备列表里。

**-> step 4 - usb interface**



USB设备各个配置的详细信息在USB core里的漫漫旅途中已经被获取并存放在相关的几个成员里。

usb\_generic\_driver得到了USB设备的详细信息，然后把准备好的接口送给设备模型，Linux设备模型将接口添加到设备链表里，然后去轮询USB总线另外一条驱动链表，针对每个找到的驱动去调用USB总线的match函数，完成匹配。

#### 2.1.4 USB Request Block (urb)

USB主机与设备间的通信以数据包(packet)的形式传递，Linux的思想就是把这些遵循协议的数据都封装成数据块(block)作统一调度，USB的数据块就是urb，结构体struct urb，定义在<linux/usb.h>，其中的成员unsigned char \*setup\_packet指针指向SETUP数据包。下面总结下使用urb完成一次完整的USB通信需要经历的过程。

-> step 1 - usb\_alloc\_urb()

创建urb，并指定USB设备的目的端点。

-> step 2 - usb\_control\_msg()

将urb提交给USB core, USB core将它交给HCD主机控制器驱动。

-> step 3 - usb\_parse\_configuration()

HCD解析urb,拿到数据与USB设备通信。

-> step 4

HCD把urb的所有权交还给驱动程序。

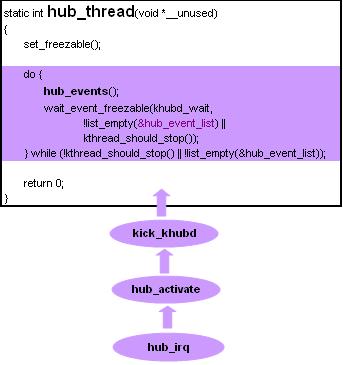
协议层里最重要的函数就是usb\_control/bulk/interrupt\_msg()，这里就简单地理一条线索，

usb\_control\_msg() => usb\_internal\_control\_msg() => usb\_start\_wait\_urb() => usb\_submit\_urb() => usb\_hcd\_submit\_urb => hcd->driver->urb\_enqueue() HCD主控制器驱动根据具体平台实现USB数据通信。

#### 2.1.5 USB Hub

Hub集线器用来连接更多USB设备，硬件上实现了USB设备的总线枚举过程，软件上实现了USB设备与接口在USB总线上的匹配。

USB子系统初始化时，usb\_hub\_init()开启一个名为"khubd"的内核线程。



内核线程khubd从Linux启动后就自始至终为USB Hub服务，没有Hub事件时khubd进入睡眠，有USB Hub事件触发时将会经由hud\_irq() => hub\_activate() => kick\_khubd() 最终唤醒khubd，将事件加入hub\_event\_list列表，并执行hub\_events()。hub\_events()会不停地轮询hub\_events\_list列表去完成hub触发的事件，直到这个列表为空时退出结束，回到wait\_event\_xxx继续等待。

**处理hub事件的全过程大致可分为两步：**

第一步 判断端口状态的变化

通过hub\_port\_status()得到hub端口的状态。

源码里类似像hub\_port\_status(), hub\_hub\_status()等功能函数，都调用了核心层的usb\_control\_msg()去实现主控制器与USB设备间的通信。

第二步 处理端口的变化

hub\_port\_connect\_change()是核心函数，以端口发现有新的USB设备插入为例，USB Hub为USB设备做了以下几步重要的工作，注意这里所谓的USB设备是指插入USB Hub的外接USB设备(包括Hub和Functions)，接下来Hub都在为USB设备服务。

(1)usb\_alloc\_dev() 为USB设备申请一个sturct usb\_device结构。

(2)usb\_set\_device\_state() 设置USB设备状态为上电状态。(硬件上设备已进入powered状态)。

(3)choose\_address()为USB设备选择一个地址，利用一个轮询算法为设备从0-127里选择一个地址号。

(4)hub\_port\_init() 端口初始化，实质就是获取设备描述符device descriptor。

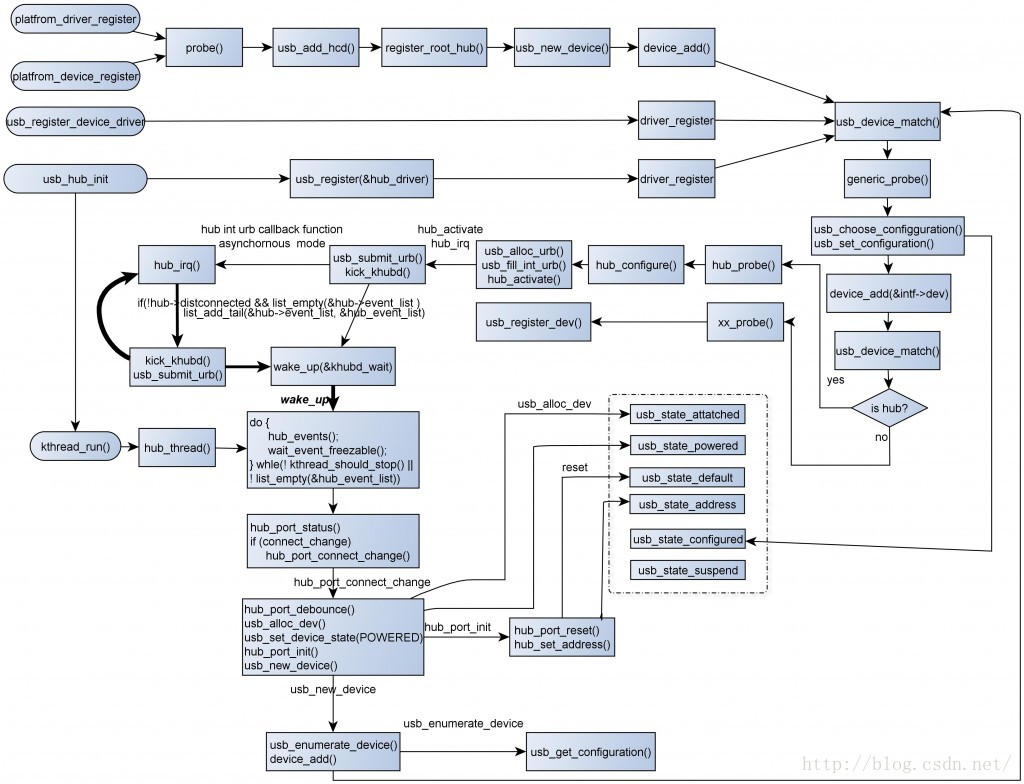
(5)usb\_get\_status() 这个有点特殊，它是专门给Hub又外接Hub而准备的。

(6)usb\_new\_device() 这时USB设备已经进入了Configured状态，调用device\_add()在USB总线上寻找驱动，若匹配成功，则加载对应的驱动程序。

#### 2.1.6 USB HOST工作示意图

无论是hub还是普通的usb设备，它们注册到usb\_bus\_type都会经历两次Match，因为第一次注册进来时，是将整个设备作为一个 device注册，然后在通用的devices 驱动程序usb\_generic\_driver的 generic\_probe函数中，将该设备的所有接口进行设置并将这些接口注册到usb\_bus\_type。

第二次match则是比较接口驱动，也就是usb\_register注册的驱动，如果是Hub设备的接口，则会调用 hub\_probe，如果是其他设备则调用 xx\_probe函数。如果是Hub的话，usb主机会监测hub端口变化，如果有变化会分配一个usb\_devices注册到 usb\_bus\_type重复前边的步骤。



### 2.2 S3C2440 USB HOST代码分析

#### 2.2.1 USB platform平台总线

整个驱动框架从platform总线开始。

static struct platform\_driver ohci\_hcd\_s3c2410\_driver **=** **{**

**.**probe **=** ohci\_hcd\_s3c2410\_drv\_probe**,**

**.**remove **=** ohci\_hcd\_s3c2410\_drv\_remove**,**

**.**shutdown **=** usb\_hcd\_platform\_shutdown**,**

/\*.suspend = ohci\_hcd\_s3c2410\_drv\_suspend, \*/

/\*.resume = ohci\_hcd\_s3c2410\_drv\_resume, \*/

**.**driver **=** **{**

**.**owner **=** THIS\_MODULE**,**

**.**name **=** "s3c2410-ohci"**,**

**},**

**};**

/\* USB Host Controller \*/

static struct resource s3c\_usb\_resource**[]** **=** **{**

**[**0**]** **=** **{**

**.**start **=** S3C24XX\_PA\_USBHOST**,**

**.**end **=** S3C24XX\_PA\_USBHOST **+** S3C24XX\_SZ\_USBHOST **-** 1**,**

**.**flags **=** IORESOURCE\_MEM**,**

**},**

**[**1**]** **=** **{**

**.**start **=** IRQ\_USBH**,**

**.**end **=** IRQ\_USBH**,**

**.**flags **=** IORESOURCE\_IRQ**,**

**}**

**};**

struct platform\_device s3c\_device\_usb **=** **{**

**.**name **=** "s3c2410-ohci"**,**

**.**id **=** **-**1**,**

**.**num\_resources **=** ARRAY\_SIZE**(**s3c\_usb\_resource**),**

**.**resource **=** s3c\_usb\_resource**,**

**.**dev **=** **{**

**.**dma\_mask **=** **&**s3c\_device\_usb\_dmamask**,**

**.**coherent\_dma\_mask **=** 0xffffffffUL

**}**

**};**

当usb的platform driver和platform device匹配上时，ohci\_hcd\_s3c2410\_drv\_probe函数被调用。

#### 2.2.2 ohci\_hcd\_s3c2410\_drv\_probe函数

static int ohci\_hcd\_s3c2410\_drv\_probe**(**struct platform\_device **\***pdev**)**

**{**

**return** usb\_hcd\_s3c2410\_probe**(&**ohci\_s3c2410\_hc\_driver**,** pdev**);**

**}**

static int usb\_hcd\_s3c2410\_probe **(**const struct hc\_driver **\***driver**,**

struct platform\_device **\***dev**)**

**{**

struct usb\_hcd **\***hcd **=** **NULL;**

int retval**;**

//这里没有设置platform\_data，所以该代码无用

s3c2410\_usb\_set\_power**(**dev**->**dev**.**platform\_data**,** 1**,** 1**);**

//创建usb\_hcd 绑定usb\_driver等

hcd **=** usb\_create\_hcd**(**driver**,** **&**dev**->**dev**,** "s3c24xx"**);**

**if** **(**hcd **==** **NULL)**

**return** **-**ENOMEM**;**

//主机控制寄存器 起始地址 结束地址

hcd**->**rsrc\_start **=** dev**->**resource**[**0**].**start**;**

hcd**->**rsrc\_len **=** dev**->**resource**[**0**].**end **-** dev**->**resource**[**0**].**start **+** 1**;**

//申请IO空间

**if** **(!**request\_mem\_region**(**hcd**->**rsrc\_start**,** hcd**->**rsrc\_len**,** hcd\_name**))** **{**

dev\_err**(&**dev**->**dev**,** "request\_mem\_region failed"**);**

retval **=** **-**EBUSY**;**

**goto** err\_put**;**

**}**

//获得usb host时钟

clk **=** clk\_get**(&**dev**->**dev**,** "usb-host"**);**

//获得usb bus host时钟

usb\_clk **=** clk\_get**(&**dev**->**dev**,** "usb-bus-host"**);**

//使能时钟

s3c2410\_start\_hc**(**dev**,** hcd**);**

//ioremap usb host相关寄存器

hcd**->**regs **=** ioremap**(**hcd**->**rsrc\_start**,** hcd**->**rsrc\_len**);**

//初始化ohci

ohci\_hcd\_init**(**hcd\_to\_ohci**(**hcd**));**

//加载hcd驱动

retval **=** usb\_add\_hcd**(**hcd**,** dev**->**resource**[**1**].**start**,** IRQF\_DISABLED**);**

**if** **(**retval **!=** 0**)**

**goto** err\_ioremap**;**

**return** 0**;**

err\_ioremap**:**

s3c2410\_stop\_hc**(**dev**);**

iounmap**(**hcd**->**regs**);**

clk\_put**(**usb\_clk**);**

err\_clk**:**

clk\_put**(**clk**);**

err\_mem**:**

release\_mem\_region**(**hcd**->**rsrc\_start**,** hcd**->**rsrc\_len**);**

err\_put**:**

usb\_put\_hcd**(**hcd**);**

**return** retval**;**

**}**

probe的主要工作就是分配一个 usb\_hcd 结构、设置然后 usb\_add\_hcd 。

##### 2.2.2.1 usb\_create\_hcd

usb\_hcd——USB Host Controller Driver，同时，一个主机控制器驱动对应一条 usb\_bus。

struct usb\_hcd **\***usb\_create\_hcd **(**const struct hc\_driver **\***driver**,**

struct device **\***dev**,** char **\***bus\_name**)**

**{**

struct usb\_hcd **\***hcd**;**

//分配一个 usb\_hcd + driver->hcd\_priv\_size 空间

//这里的hcd\_priv\_size = sizeof(struct ohci\_hcd)

//hcd\_priv成员在struct usb\_hcd被定义成了0项数组的形式

//而大小则是由hc\_driver的hcd\_priv\_size项来指定的

hcd **=** kzalloc**(sizeof(\***hcd**)** **+** driver**->**hcd\_priv\_size**,** GFP\_KERNEL**);**

dev\_set\_drvdata**(**dev**,** hcd**);**

kref\_init**(&**hcd**->**kref**);**

//初始化 usb\_bus ,一个主机控制器对应一个 usb\_bus

usb\_bus\_init**(&**hcd**->**self**);**

hcd**->**self**.**controller **=** dev**;**

hcd**->**self**.**bus\_name **=** bus\_name**;**

hcd**->**self**.**uses\_dma **=** **(**dev**->**dma\_mask **!=** **NULL);**

//设置 usb\_bus

init\_timer**(&**hcd**->**rh\_timer**);**

hcd**->**rh\_timer**.**function **=** rh\_timer\_func**;**

hcd**->**rh\_timer**.**data **=** **(**unsigned long**)** hcd**;**

#ifdef CONFIG\_PM

INIT\_WORK**(&**hcd**->**wakeup\_work**,** hcd\_resume\_work**);**

#endif

//绑定 hc\_driver

hcd**->**driver **=** driver**;**

hcd**->**product\_desc **=** **(**driver**->**product\_desc**)** **?** driver**->**product\_desc **:**

"USB Host Controller"**;**

**return** hcd**;**

**}**

##### 2.2.2.2 hc\_driver

hc就是host control的意思.即为主机控制器驱动。

从代码看2410采用的OCHI标准，OCHI标准下的USB控制器都是一样的，遵循统一的标准。

static const struct hc\_driver ohci\_s3c2410\_hc\_driver **=** **{**

**.**description **=** hcd\_name**,**

**.**product\_desc **=** "S3C24XX OHCI"**,**

**.**hcd\_priv\_size **=** **sizeof(**struct ohci\_hcd**),**

/\*

\* generic hardware linkage

\*/

**.**irq **=** ohci\_irq**,**

**.**flags **=** HCD\_USB11 **|** HCD\_MEMORY**,**

/\*

\* basic lifecycle operations

\*/

**.**start **=** ohci\_s3c2410\_start**,**

**.**stop **=** ohci\_stop**,**

**.**shutdown **=** ohci\_shutdown**,**

/\*

\* managing i/o requests and associated device resources

\*/

**.**urb\_enqueue **=** ohci\_urb\_enqueue**,**

**.**urb\_dequeue **=** ohci\_urb\_dequeue**,**

**.**endpoint\_disable **=** ohci\_endpoint\_disable**,**

/\*

\* scheduling support

\*/

**.**get\_frame\_number **=** ohci\_get\_frame**,**

/\*

\* root hub support

\*/

**.**hub\_status\_data **=** ohci\_s3c2410\_hub\_status\_data**,**

**.**hub\_control **=** ohci\_s3c2410\_hub\_control**,**

**.**hub\_irq\_enable **=** ohci\_rhsc\_enable**,**

#ifdef CONFIG\_PM

**.**bus\_suspend **=** ohci\_bus\_suspend**,**

**.**bus\_resume **=** ohci\_bus\_resume**,**

#endif

**.**start\_port\_reset **=** ohci\_start\_port\_reset**,**

**};**

其中：

ohci\_run：ohci控制器的初始化

ohci\_urb\_enqueue：OHCI数据传输

目前整个OHCI还不是很了解，后续补充。

##### 2.2.2.3 usb\_add\_hcd

int usb\_add\_hcd**(**struct usb\_hcd **\***hcd**,**

unsigned int irqnum**,** unsigned long irqflags**)**

**{**

int retval**;**

struct usb\_device **\***rhdev**;**

dev\_info**(**hcd**->**self**.**controller**,** "%s\n"**,** hcd**->**product\_desc**);**

set\_bit**(**HCD\_FLAG\_HW\_ACCESSIBLE**,** **&**hcd**->**flags**);**

**if** **((**retval **=** hcd\_buffer\_create**(**hcd**))** **!=** 0**)** **{**

dev\_dbg**(**hcd**->**self**.**controller**,** "pool alloc failed\n"**);**

**return** retval**;**

**}**

**if** **((**retval **=** usb\_register\_bus**(&**hcd**->**self**))** **<** 0**)**

**goto** err\_register\_bus**;**

**if** **((**rhdev **=** usb\_alloc\_dev**(NULL,** **&**hcd**->**self**,** 0**))** **==** **NULL)** **{**

dev\_err**(**hcd**->**self**.**controller**,** "unable to allocate root hub\n"**);**

retval **=** **-**ENOMEM**;**

**goto** err\_allocate\_root\_hub**;**

**}**

rhdev**->**speed **=** **(**hcd**->**driver**->**flags **&** HCD\_USB2**)** **?** USB\_SPEED\_HIGH **:**

USB\_SPEED\_FULL**;**

hcd**->**self**.**root\_hub **=** rhdev**;**

/\* wakeup flag init defaults to "everything works" for root hubs,

\* but drivers can override it in reset() if needed, along with

\* recording the overall controller's system wakeup capability.

\*/

device\_init\_wakeup**(&**rhdev**->**dev**,** 1**);**

/\* "reset" is misnamed; its role is now one-time init. the controller

\* should already have been reset (and boot firmware kicked off etc).

\*/

**if** **(**hcd**->**driver**->**reset **&&** **(**retval **=** hcd**->**driver**->**reset**(**hcd**))** **<** 0**)** **{**

dev\_err**(**hcd**->**self**.**controller**,** "can't setup\n"**);**

**goto** err\_hcd\_driver\_setup**;**

**}**

/\* NOTE: root hub and controller capabilities may not be the same \*/

**if** **(**device\_can\_wakeup**(**hcd**->**self**.**controller**)**

**&&** device\_can\_wakeup**(&**hcd**->**self**.**root\_hub**->**dev**))**

dev\_dbg**(**hcd**->**self**.**controller**,** "supports USB remote wakeup\n"**);**

/\* enable irqs just before we start the controller \*/

**if** **(**hcd**->**driver**->**irq**)** **{**

snprintf**(**hcd**->**irq\_descr**,** **sizeof(**hcd**->**irq\_descr**),** "%s:usb%d"**,**

hcd**->**driver**->**description**,** hcd**->**self**.**busnum**);**

**if** **((**retval **=** request\_irq**(**irqnum**,** **&**usb\_hcd\_irq**,** irqflags**,**

hcd**->**irq\_descr**,** hcd**))** **!=** 0**)** **{**

dev\_err**(**hcd**->**self**.**controller**,**

"request interrupt %d failed\n"**,** irqnum**);**

**goto** err\_request\_irq**;**

**}**

hcd**->**irq **=** irqnum**;**

dev\_info**(**hcd**->**self**.**controller**,** "irq %d, %s 0x%08llx\n"**,** irqnum**,**

**(**hcd**->**driver**->**flags **&** HCD\_MEMORY**)** **?**

"io mem" **:** "io base"**,**

**(**unsigned long long**)**hcd**->**rsrc\_start**);**

**}** **else** **{**

hcd**->**irq **=** **-**1**;**

**if** **(**hcd**->**rsrc\_start**)**

dev\_info**(**hcd**->**self**.**controller**,** "%s 0x%08llx\n"**,**

**(**hcd**->**driver**->**flags **&** HCD\_MEMORY**)** **?**

"io mem" **:** "io base"**,**

**(**unsigned long long**)**hcd**->**rsrc\_start**);**

**}**

**if** **((**retval **=** hcd**->**driver**->**start**(**hcd**))** **<** 0**)** **{**

dev\_err**(**hcd**->**self**.**controller**,** "startup error %d\n"**,** retval**);**

**goto** err\_hcd\_driver\_start**;**

**}**

/\* starting here, usbcore will pay attention to this root hub \*/

rhdev**->**bus\_mA **=** min**(**500u**,** hcd**->**power\_budget**);**

**if** **((**retval **=** register\_root\_hub**(**hcd**))** **!=** 0**)**

**goto** err\_register\_root\_hub**;**

**if** **(**hcd**->**uses\_new\_polling **&&** hcd**->**poll\_rh**)**

usb\_hcd\_poll\_rh\_status**(**hcd**);**

**return** retval**;**

err\_register\_root\_hub**:**

hcd**->**driver**->**stop**(**hcd**);**

err\_hcd\_driver\_start**:**

**if** **(**hcd**->**irq **>=** 0**)**

free\_irq**(**irqnum**,** hcd**);**

err\_request\_irq**:**

err\_hcd\_driver\_setup**:**

hcd**->**self**.**root\_hub **=** **NULL;**

usb\_put\_dev**(**rhdev**);**

err\_allocate\_root\_hub**:**

usb\_deregister\_bus**(&**hcd**->**self**);**

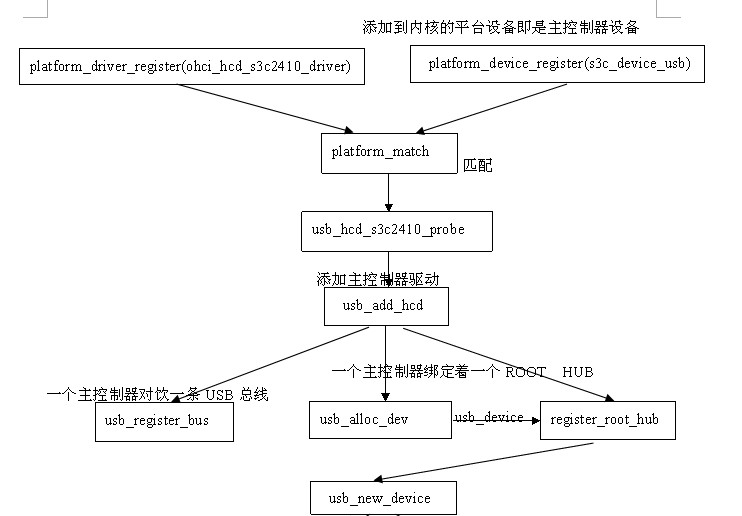
err\_register\_bus**:**

hcd\_buffer\_destroy**(**hcd**);**

**return** retval**;**

**}**

如下图可以描述usb\_add\_hcd完成的功能。



(1)注册一条USB总线

(2)注册root\_hub

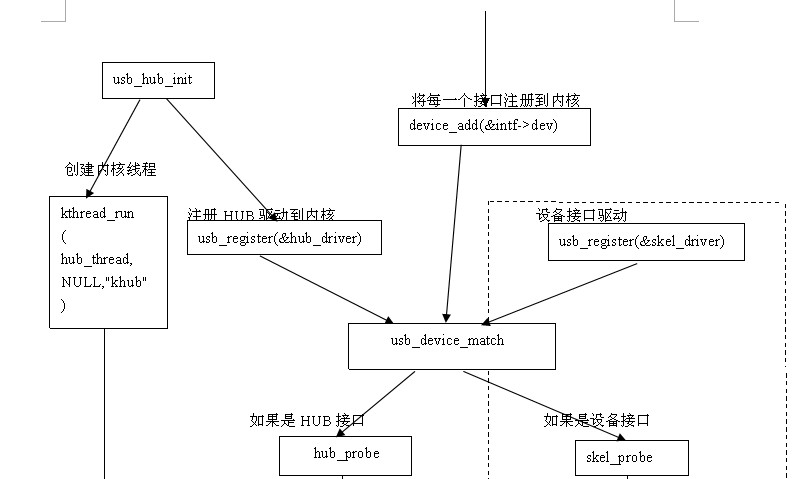
#### 2.2.3 OCHI插入检测流程

##### 2.2.3.1 USB Root HUB初始化

root hub是第一个usb设备，也是必须的usb设备。

在usb\_hub\_init中注册了usb\_hub驱动到USB子系统中。在usb\_add\_hcd中通过register\_root\_hub注册到usb\_root\_hub设备，注册driver或device都会调用usb\_device\_match函数，如果匹配成功，则hub\_probe函数调用。

相关流程见下图：



##### 2.2.3.2 USB设备插入检测

在usb\_hub\_init还创建了一个线程hub\_thread，在线程hub\_thread中主要做的事是循环调用hub\_events，hub\_events只执行一遍便进入休眠等待状态，需要wake\_up(&khubd\_wait)将其唤醒从而再次调用hub\_events。

那么这个wakeup(&khubd\_wait)是在哪里被调用的呢？这就涉及到从设备插入到host去枚举设备的整个流程了。

(1)初始化urb并进行submit

static int hub\_configure**(**struct usb\_hub **\***hub**,** struct usb\_endpoint\_descriptor **\***endpoint**)**

**{**

hub**->**urb **=** usb\_alloc\_urb**(**0**,** GFP\_KERNEL**);**

usb\_fill\_int\_urb**(**hub**->**urb**,** hdev**,** pipe**,** **\***hub**->**buffer**,** maxp**,** hub\_irq**,** hub**,** endpoint**->**bInterval**);**

hub\_activate**(**hub**,** HUB\_INIT**);**

usb\_submit\_urb**(**hub**->**urb**,** GFP\_NOIO**);**

usb\_hcd\_submit\_urb**(**urb**,** mem\_flags**);**

rh\_urb\_enqueue**(**hcd**,** urb**);**

rh\_queue\_status **(**hcd**,** urb**);**

hcd**->**status\_urb **=** urb**;**

上面创建了一个hub的urb，设置urb的complete回调函数为hub\_irq，然后注册这个urb。hub\_irq后面会讲到当roothub检测到roothub status change的中断函数中最终会调用hub\_irq，hub\_irq函数通过wakeup(&khubd\_wait)将睡眠等待的hub\_thread唤醒，从而进一步对设备进行枚举识别。

(2)设备插入中断来了

static irqreturn\_t ohci\_irq **(**struct usb\_hcd **\***hcd**)**

**{**

**if** **(**ints **&** OHCI\_INTR\_RHSC**)** **{**

ohci\_vdbg**(**ohci**,** "rhsc\n"**);**

ohci**->**next\_statechange **=** jiffies **+** STATECHANGE\_DELAY**;**

ohci\_writel**(**ohci**,** OHCI\_INTR\_RD **|** OHCI\_INTR\_RHSC**,** **&**regs**->**intrstatus**);**

ohci\_writel**(**ohci**,** OHCI\_INTR\_RHSC**,** **&**regs**->**intrdisable**);**

usb\_hcd\_poll\_rh\_status**(**hcd**);**

**}**

**}**

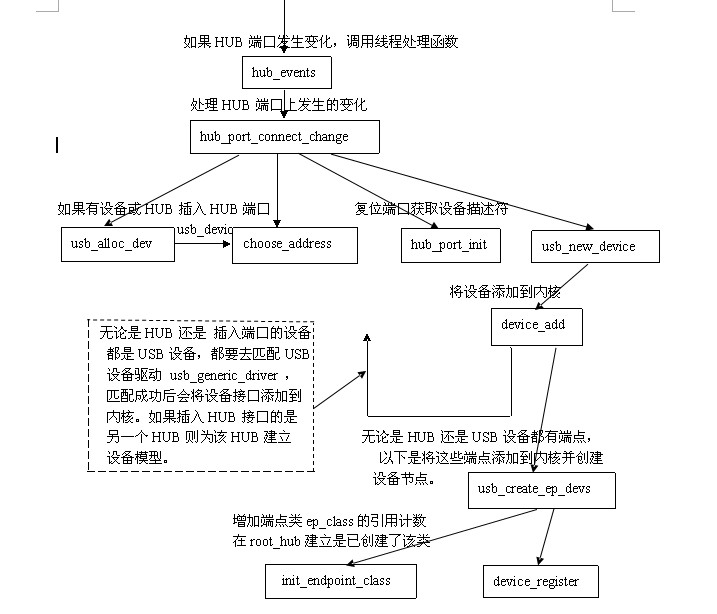
当roothub检测到OHCI\_INTR\_RHSC中断的时候，他会调用usb\_hcd\_poll\_rh\_status去确认roothub status是否真的有变化，并且通过urb->complete即hub\_irq来通知usb/core中的hub\_thread底层设备有插入。

另外在底层创建usb\_hcd的usb\_create\_hcd->usb\_create\_shared\_hcd函数中会创建一个hcd->rh\_timer，这个timer会不断的调用usb\_hcd\_poll\_rh\_status来polling roothub的状态。

总的来说，usb\_hcd\_poll\_rh\_status一方面会在roothub的OHCI\_INTR\_RHSC中断中调用，另一方面又被rh\_timer不断的polling。

回到hub\_thread和hub\_events，来看看设备的枚举，当urb->complete函数hub\_irq被调用时，他会去唤醒睡眠等待的hub\_thread，然后执行hub\_events。

**剩下的过程请参照2.1.5。**



#### 2.2.4 USB鼠标驱动分析

##### 2.2.4.1 USB设备插入识别过程

(1)root hub检测到有设备插入

driver**/**usb**/**core**/**hub**.**c

usb**->**hun\_thread**()->**hub**->**events**()->**hub\_port\_connect\_change**()**

(2)寻找匹配的驱动程序

driver**/**usb**/**core**/**hub**.**c

hub\_port\_connect\_change**()**   //检测到新设备连接

usb\_new\_device**(**udev**)**        //注册新设备

usb\_get\_configuration**(**udev**)** //获得设备各种描述符

device\_add**(&**udev**->**dev**)**      //把这个设备注册到usb系统中

bus\_attach\_device**(**dev**)**   //把这个设备添加到相应bus的设备列表中

device\_attach**()**         //为设备找到相应的设备驱动程序

bus\_for\_each\_drv**(**dev**->**bus**,NULL,**dev**,**\_\_device\_attach**)**

//从总线上已注册的所有驱动中找出匹配的驱动程序

(3) \_\_device\_attach

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

**{**

struct device **\*** dev **=** data**;**

**return** driver\_probe\_device**(**drv**,** dev**);**

**}**

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**;**

ret **=** really\_probe**(**dev**,** drv**);**

done**:**

**return** ret**;**

**}**

(4) usb\_device\_match

从\_\_device\_attach函数中，可以看到先进行match的操作，再调用really\_probe函数。

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

**{**

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

**{**

    /\*dev代表整个usb设备\*/

**...**

**}**

**else**

**{**

/\*dev代表一个usb设备interface\*/

**...**

usb\_match\_id**();**

**...**

usb\_match\_dynamic\_id**();**

**...**

**}**

**}**

这个函数只是做一些粗略的匹配, 如果匹配成功则返回1,然后由really\_probe来做进一步的匹配。

(5)probe分支1

分支1: dev drv代表的是设备级别:

此时的drv肯定是usb\_generic\_driver。因为在当前的usb系统中只有这个driver是代表整个设备的驱动,它是在usb\_init中被注册的，而我们通常写的usb驱动都是代表一个interface的。

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

**...**

**.**probe **=** generic\_probe**,**

**...**

**}**

static int generic\_probe**(**struct usb\_device **\***udev**)**

**{**

**...**

c **=** choose\_configuration**(**dev**);**

**if(**c **>=** 0**)** **{**

err **=** usb\_set\_configuration**(**udev**,** c**);**  //设置配置,并注册interface.

**...**

**}**

**...**

**}**

该函数为这个usb设备选择一个合适的配置,并注册这个配置下面的interface.

它为当前配置下的每个interface调用device\_add()函数。

int usb\_set\_configuration**(**struct usb\_device **\***dev**,** int configuration**)**

**{**

**...**

**for(**I **=** 0**;** I **<** nintf**;** i**++)** **{**

struct usb\_interface **\***intf **=** cp**->**interface**[**i**];**

**...**

device\_add**(&**intf**->**dev**);**

**...**

**}**

**...**

**}**

device\_add又会回到上面的函数，下面将进入分支2。

(6)probe分支2

分支2：dev drv代表的是interface级别

此时的dev代表着一个interface, 而drv就代表了我们自己的usb驱动. 但是我们应当看到drv是device\_driver类型, 而我们写的usb驱动的类型一般是usb\_driver, 因此这里的probe和我们自己写的probe显然不是同一个. 实际上这里的drv是我们的驱动对象里内嵌的一个子对象(因为linux下所以的驱动都必须用device\_driver来代表,). 那这个子对象的probe函数是在哪里赋值的呢? 这就要看usb\_register函数了

跟踪这个函数我们可以看到这里的probe 函数实际上是usb\_probe\_interface   
/\*usb/core/driver.c\*/ (所有的usb interface驱动都是一样的)。

static int usb\_probe\_interface**(**struct device **\***dev**)**

**{**

struct driver **=** to\_usb\_driver**(**dev**->**driver**);**  //dev->driver  在really\_probe中设置.

**...**

error **=** driver**->**probe**(**intf**,** id**);**   //这个就是我们自己写的probe函数了.

**...**

**}**

driver->probe(intf, id); 这就调用到我们自己写的代码里面了。

2.2.4.2 USB包的种类

1.令牌包：用于启动一次USB传输，通过8位包标识(PID)来区分令牌包的类型

IN 令牌包用于通知设备返回一个数据包；

OUT令牌包用于通知设备将要输出一个数据包；

SETUP令牌包用在控制传输中。

以上三种令牌包的结构如下：

| 同步域 | 8位包标识 | 7位地址 | 4位端点号 | 5位CRC5校验 | EOP |

SOF包在每帧开始时以广播方式发送，发送帧号，不跟随数据：

| 同步域 | 8位包标识符 | 11位帧号 | 5位CRC5校验 | EOP |

2.数据包：用于传输数据。当设备或主机端成功发送或接收数据时，会切换数据包的类型。有DATA0, DATA1, DATA2, MDATA类型。结构为：

| 同步域 | 8位包标识符 | 字节0 | 字节1 | …… | 字节N | 16位CRC16校验 | EOP |

3.握手包。握手包数据结构简单， 用于表示对方是否确认传输。

| 同步域 | 包标识 | EOP |

握手包的类型有ACK NAK STALL NYET

4.先导包。特殊场合使用的包，主要包括PRE, ERR, SPLIT, PING.

PRE ： 只用在全速中，通知集线器打开低速端口。

PING ： 等待设备返回ACK或NAK，判断设备是否能够进行传输；

SPLIT ：通知集线器将高速数据包转化为全速或低速数据包下发；

ERR ： 分裂事物中的错误表示。