# TODO

上拉电阻

<https://www.cnblogs.com/ct20150811/p/10523559.html>

hub流程

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

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

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

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

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

# 物理接口

针与孔， 凸与凹，公与母，插与被插

插的为公 被插的为母

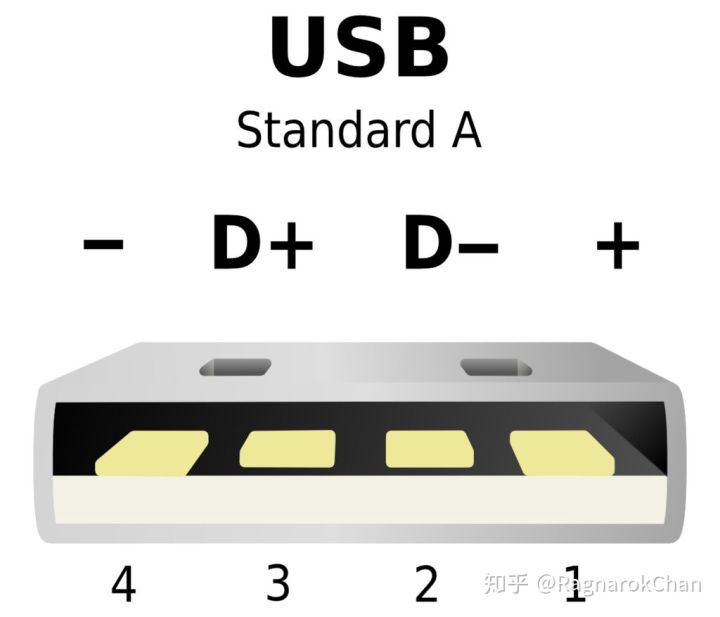




## 1.1 - USB Type-A

USB Type-A又可称为USB-A。直到现在不少PC、PC周边、手机充电器等等都依然采用了这种接口，是目前普及度最高的USB接口了。而USB-A亦有分为公座与母座。常见的USB-A数据线的A端就是公座，而充电器上的则是母座。这个就不用多说了。



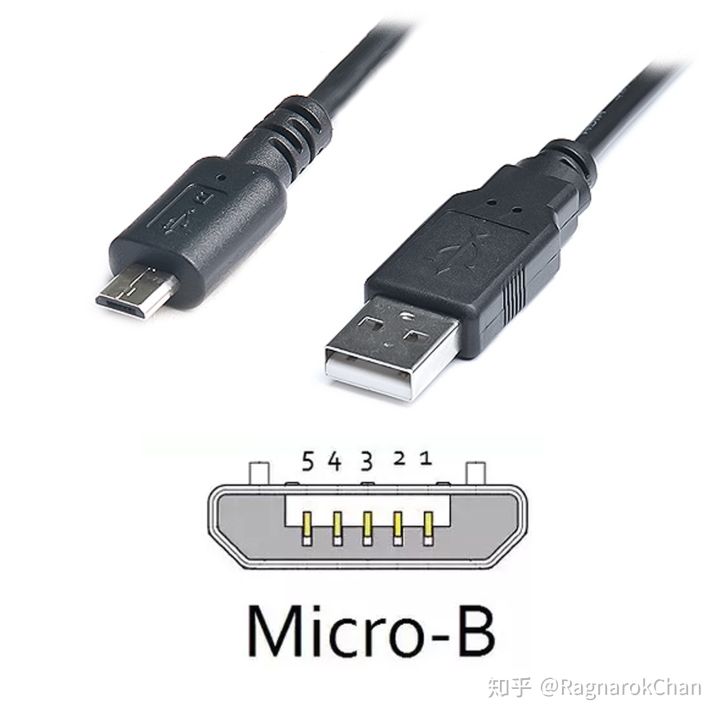


### - USB Type-A的结构

USB-A公座上下设计了4个凹口，与母座4个突出的弹片相互配合进行固定。所以多次拔插的USB-A公座上会有明显的划动轨迹

## – MicroUSB

USB-A的体积相对较大，对于一些便携或者体积较小的设备并不友好。MicroUSB因需求而诞生了。MicroUSB的Pin脚定义与各类USB相近，但多了一Pin为ID空白设计。而正如刚才所说，MicroUSB更多用在一些手机（目前已经全面转向USB-C接口）、各种类型的USB小型设备（如USB风扇）。目前MicroUSB的成本较低的原因，大量单价较低的USB设备依然选择这种USB接口。



### - MicroUSB的结构

MicroUSB使用2个突出的小卡口与母座上的镂空位置进行卡扣配合，由于卡扣使用类似弹簧的形式，多次拔插后当卡扣回复幅度越来越小，固定性能则会越来越差。到最后，将会无法卡紧接口导致接触不良甚至接口脱出。

## 1.3 - USB-A / MicroUSB高速接口

## 1.4 - USB Type-C

专题讲解

# 2章 - 传输接口版本

## 2.1 - 前言：混乱的命名方式

## 2.2 - USB 2.0简析

## 2.3 - USB 3.2简析

### 2.3.1 - USB 3.0 / 3.1 / 3.2有什么区别呢？

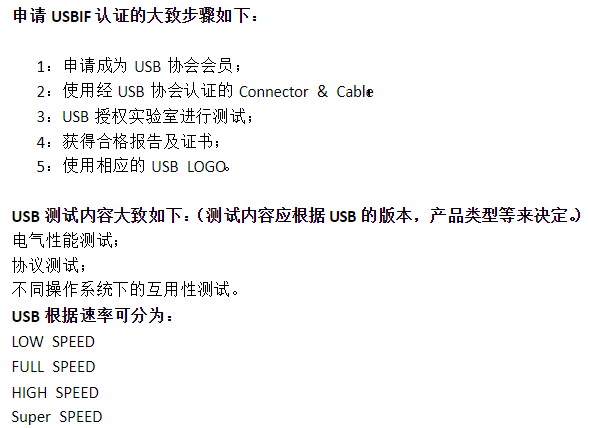
### 2.3.2 - 又爱又恨，USB 3.2与WIFI、移动网络的冲突

# USB认证

USBIF认证是由USB开发者论坛提出的一种非强制认证，一般为客户自愿申请，USB开发者论坛(USB Implementers Forum，USBIF) 负责 USB 标准制订，其成员目前包括HP、Intel

USBIF是一个非盈利组织，主要的活动在推广、营销USB标准，并维护规格及认证程序









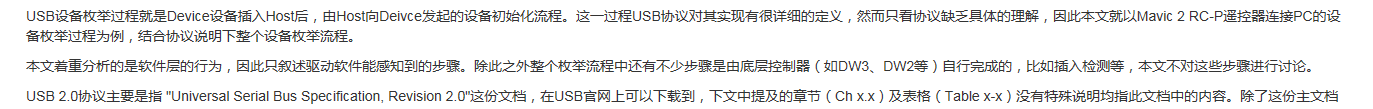
## 测试





## USB2.0协议





# 源码分析PDF446

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

什么 是 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”。

USB Core：

早期的 Linux 内核，其结构并不是如今天这般有层次感，远不像今天这般 错落有致，那时候 drivers/usb/目录下放了很多文件，USB Core 与其他各种设备驱动程序代码都 堆砌在这里，后来，在 drivers/usb/目录下面出来了一个 core 目录，就专门放一些核心的代码， 比如初始化整个 USB 系统，初始化 Root Hub，初始化主机控制器的代码，再后来甚至把主机 控制器相关的代码也单独建了一个目录，叫 host 目录。这是因为 USB 主机控制器随着时代的 发展，也开始有了好几种，不再像刚开始那样只有一种。所以，设计者们把一些主机控制器公 共的代码仍然留在 core 目录下，而一些各主机控制器单独的代码则移到 host 目录下面负责各种 主机控制器的人去维护。

那么 USB gadget 呢？gadget 说白了就是配件的意思，主要就是一些内部运行 Linux 的嵌入 式设备，比如 PDA，设备本身有 USB 设备控制器（USB Device Controller），可以将 PC，也就 是我们的主机作为 master 端，将这样的设备作为 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。比如，我们的嵌入式主板上支持 SD 卡，如果我们 希望将主板通过 USB 连接到 PC 之后，这个 SD 卡被模拟成 U 盘，那么就要通过 USB gadget 架构的驱动

gadget 目录下大概可以分为两个模块：一个是 udc 驱动，这个驱动是针对具体 CPU 平台的， 如果找不到现成的，就要自己实现；另外一个就是 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。

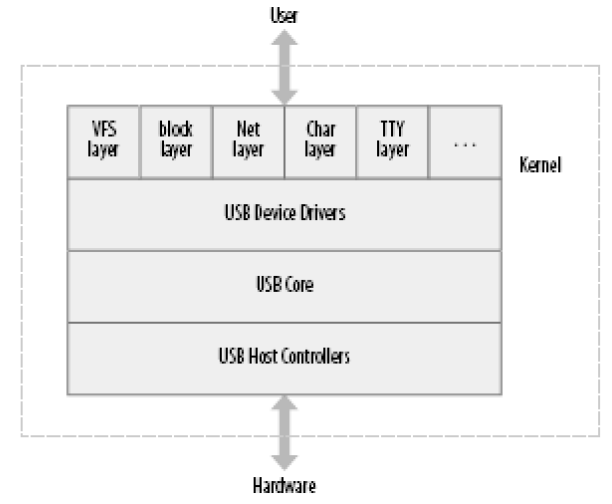
剩下的几个目录分门别类地放了各种 USB 设备的驱动，U 盘的驱动在 storage 目录下

触 摸屏和 USB 键盘鼠标的驱动在 input 目录下等。

另外，在 USB 协议中，除了通用的软硬件电气 接口规范等，还包含各种各样的 Class 协议，用来为不同的功能定义各自的标准接口和具体的 总线上的数据交互格式和内容。这些 Class 协议的数量非常多，比如常见的支持 U 盘功能的 Mass Storage Class，以及通用的数据交换协议 CDC Class。此外，还包括 Audio Class、Print Class 等。理论上讲，即使没有这些 Class，通过专用驱动也能够实现各种各样的应用功能。但是，正 是 Mass Storage Class 的使用，使得各个厂商生产的 U 盘都能通过操作系统自带的统一驱动程 序来使用，对 U 盘的普及起了极大的推动作用，制定其他的 Class 也是同样的目的。

## 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

## drivers/usb/core

### Kconfig

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。

### 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

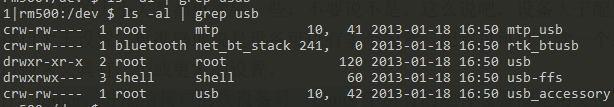
竟大家对互相之间的大小关系都更敏感一些，不要说不是。这么说吧，设备大于配置，配置大

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。

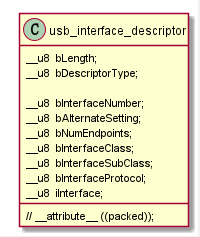
这些描述符放哪儿？当然是在设备里。usb 设备里都会有一个叫 EEPROM 的东东，没错，就是放在它那儿，它就是用来存储设备 本身信息的。如果你的脑海里还残存着一些大学里的美好时光的话，应该还会记得 EEPROM，就是电可擦写的可编程ROM，它与Flash虽说都是要电擦除的，但它可以按字节擦除，Flash 只能一次擦除一个 block，所以如果要改动比较少的数据的话，使用它还是比较合适的，但 是世界上没有完美的东西，此物成本相对 Flash 比较高，所以一般来说 usb 设备里只拿它 来存储一些本身特有的信息，要想存储数据，还是用 Flash 吧。

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

# OTG

E:\K\kernel\msm-android-msm-marlin-3.18-pie-qpr2\drivers\usb\gadget\configfs.c



设置type C空盒子

gadget\_dev\_desc\_UDC\_store

sys.usb.controller

/config/usb\_gadget/g1/UDC

configs.c

sys.usb.controller

## typeA口的逻辑

默认值为device

首先系统的默认值定义在device/rockship/kj001/kj001.mk **sys.usb.mode peripheral**

自动切换：插线为device，否则为host

开机系统服务起来之后5s内，检查到typeA连接则为自动切换peripheral，测试就可以和电脑连接了

手动切换**device**

**private static final** String USB\_MODE\_HOST = **"host"**;  
**private static final** String USB\_MODE\_PERIPHERAL = **"peripheral"**;  
**private static final** String PROP\_USB\_MODE = **"sys.usb.mode"**;

SystemProperties.set(PROP\_USB\_MODE, usbMode);

如果要连接电脑：

setprop **sys.usb.mode peripheral**

**然后触发init事件**

### setprop sys.usb.mode peripheral经历了什么

# USB CDC介绍

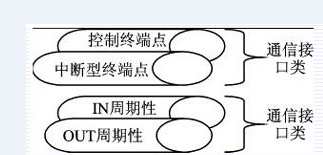
USB的CDC类是USB通信设备类（Communication Device Class）的简称。CDC类是USB组织定义的一类专门给各种通信设备（电信通信设备和中速网络通信设备）使用的USB子类。

       USB CDC具备两个类接口：通信接口类（Communication Interface Class）和数据接口类(Data Interface Class)。

协议原文：Two classes of interfaces are described in this specification:Communication Class interfaces and Data Class interfaces.The CommunicationClass interface is a management interface and is required of all communicationdevices. The Data Class interface can be used to transport data whose structureand usage is not defined by any other class, such as Audio.The format of thedata moving over this interface can be identified using the associatedCommunication Class interface.

       个人见解：通常一个CDC类又由两个接口子类组成通信接口类（Communication Interface Class）和数据接口类(Data Interface Class)。通信接口类是一个管理接口，是所有通信设备的都需要具备的接口。数据接口类和用于数据的传输，它的结构和用途与其他(USB)定义的类不一样，如音频接口类。数据接口类传输的数据格式可以由通信接口类确定

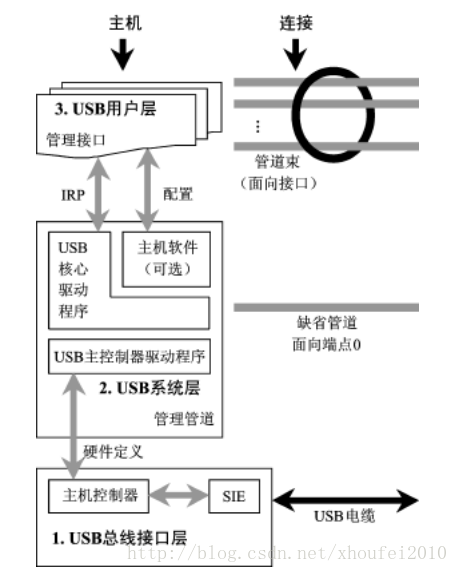
USB CDC类结构如图1所示：



通信接口类是一个主要通过通信接口类对设备进行管理和控制，而通过数据接口类传送数据。这两个接口子类占有不同数量和类型的终端点（Endpoints），如图1所示。

       通信接口类需要一个控制终端点（Control Endpoint）和一个可选的中断（Interrupt）型终端点。数据接口子类需要一个方向为输入（IN）的周期性（Isochronous）型终端点和一个方向为输出（OUT）的周期性型终端点。其中控制终端点主要用于USB设备的枚举和虚拟串口的波特率和数据类型（数据位数、停止位和起始位）设置的通信。输出方向的非同步终端点用于主机（Host）向从设备（Slave）发送数据，相当于传统物理串口中的TXD线（如果从单片机的角度看），输入方向的非同步终端点用于从设备向主机发送数据，相当于传统物理串口中的RXD线。

## USB驱动结构



对于TI的Cortex-M4芯片，第1层“USB总线接口层”、第2层“USB系统层”的“USB主控制驱动程序”已经由驱动库“SW-TM4C-2.0.1.11577.exe”完成，用户安装之后，“usblib”文件夹有相应的驱动。

用户需要完成主要为“USB核心驱动程序”和“管理接口”。

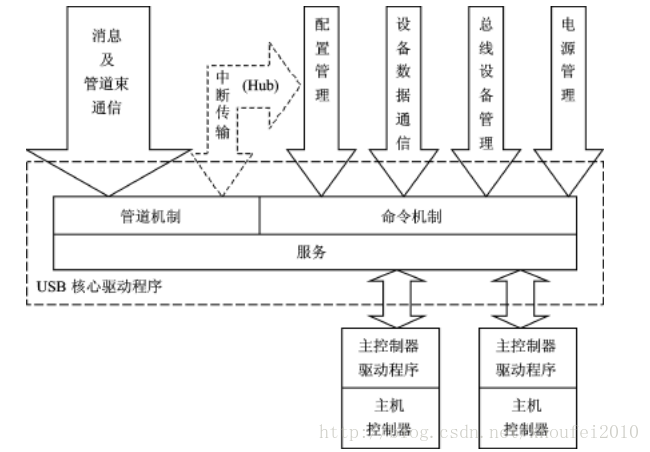


图3 USB核心驱动程序USB的结构

    对于USB CDC类传输，依靠为管道传输。TI cortex-M4系列芯片共有14个管道：7个IN管道、7个OUT管道（详细介绍可查看datasheet的USB章节）。

   USB的管道机制完全支持USB的4种管道类型(也就是数据传输的类型),即批量传输、同步传输、中断传输和控制传输。对于USB CDC的数据传输接口类，采用的为批量传输：BULK IN、BULK OUT。通信接口类采用的是中断传输。

## 2、相关资料推荐

      （1）     USB CDC标准协议网址：http://www.usb.org/developers/devclass\_docs

      （2）     USB CDC标准协议：“usbcdc11.pdf”(Universal Serial Bus Class Definitions for Communication Devices)

               CSDN下载网址：http://download.csdn.net/detail/xhoufei2010/7434447

      （3）     USB书籍资料推荐（PDF版）：《计算机USB系统原理及其主从机设计》

                     CSDN下载网址：http://download.csdn.net/detail/xhoufei2010/7434465

      （4）     程序开发环境：CCS5.5

      （5）     硬件芯片：TM4C1236D5PM

      （6）     驱动软件包（TI cortex-M4芯片驱动包）：SW-TM4C-2.0.1.11577

      （7）     工程源代码链接：

原文链接：<https://blog.csdn.net/xhoufei2010/article/details/28010617>

# USB Type-C

USB Type-C，又称USB-C。是目前USB接口中最新推出的物理接口，虽然体积并没有MicroUSB小，但功能与性能上吊打了MicroUSB。并且加入了正反盲插功能更为人性化。而智能手机开始全面淘汰MicroUSB亦证明了USB-C的设计的强大性。而USB-C的设计考虑到未来众多发展上的方向，各项性能提升至于亦加强了部分功能上的内容。从而使其从过往USB只有4 - 10Pin暴增到最多24Pin（公头满Pin为22Pin）。为了让所有设备进行大统一，USB-C的设计非常先进，是目前性能、功能最强的USB接口



图 1: USB Type-C接头外形

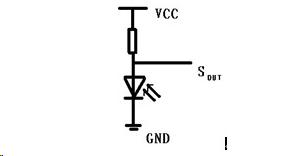
1. 基准设计电流3A，配备USB-C连接器的标准规格连接线可通过3A电流，同时还支持超出现有USB供电能力的USB Power Delivery，可以最大提供100W电力（20V/5A）
2. 正反盲插设计，更为人性化；支持正反均可插入的“正反插”功能
3. 高达40Gbps的传输速度（仅ThunderBolt3）；最大传输速度10Gb/s,即是USB 3.1 Gen2标准
4. 支持DP，可为显示器提供高清视频、音频信号输出；
5. 可通过Pin / EMaker的选用，实现不同的功能。
6. 接口插座的尺寸与原来的Micro-USB规格一样小，约为8.3mm X 2.5mm
7. 可承受1万次反复插拔

## 相关参数/名词解释

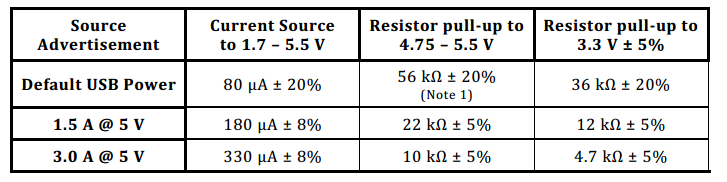
### 上拉电阻Rp

上拉电阻和下拉电阻2者共同的作用是：避免电压的“悬浮”，造成电路的不稳定

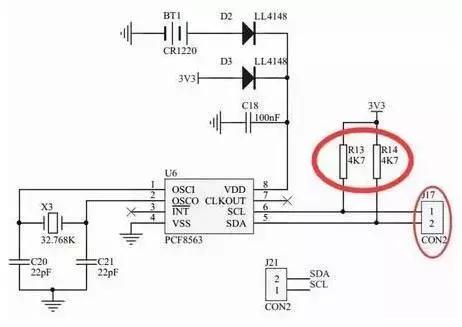
* 概念：将一个不确定的信号，通过一个电阻与电源VCC相连，固定在高电平；
* 上拉是对器件注入电流；灌电流；
* 当一个接有上拉电阻的IO端口设置为输入状态时，它的常态为高电平



Rp有6个参数（5V档位和3.3V档位各3个），指示着不同的供电能力。



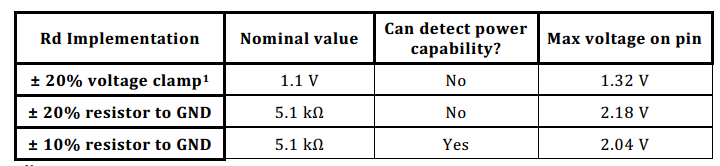
电阻一端接VCC，一端接逻辑电平接入引脚(如单片机引脚)



如上图，R13和R14，一端接到了3.3V，一端通过J17连接到单片机引脚，这两个电阻就是上拉电阻

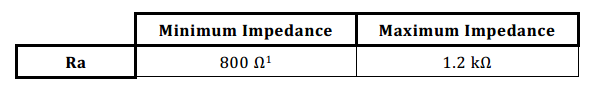
### 下拉电阻Rd

都是5.1K电阻下地，能否检测电源供电能力，取决于电阻的精度。



### 线材中的下拉电阻Ra

最小值800 ohm，最大值1.2K



### DFP

Downstream Facing Port(DFP)：下行端口，可以理解为Host，即为HOST或者HUB下行端口

DFP提供VBUS，也可以提供数据。典型的DFP设备是电源适配器，因为它永远都只是提供电源。

### UFP

Upstream Facing Port(UFP)：上行端口，即为Device或者HUB的上行端口

UFP从VBUS中取电，并可提供数据。典型设备是U盘，移动硬盘，因为它们永远都是被读取数据和从VBUS取电，当然不排除未来可能出现可以作为主机的U盘

### DRP

DRP(Dual Role Port): 双角色端口，DRP既可以做DFP(Host)，也可以做UFP(Device)，也可以在DFP与UFP间动态切换。

典型的DRP设备是电脑(电脑可以作为USB的主机，也可以作为被充电的设备（苹果新推出的MAC Book Air）)，具OTG功能的手机(手机可以作为被充电和被读数据的设备，也可以作为主机为其他设备提供电源或者读取U盘数据)，移动电源(放电和充电可通过一个USB Type-C，即此口可以放电也可以充电)。

Dual-Role Power (DRP)：能做为Sink/Source

### **CC**

**CC（Configuration Channel）**：配置通道，这是USB Type-C里新增的关键通道，它的作用有检测USB连接，检测正反插，USB设备间数据与VBUS的连接建立与管理等。

Configuration Channel (CC)：配置通道，用于识别、控制

### **USB PD(USB Power Delivery)**

Type-C中的PD的意思指是:USB Power Delivery功率传输协议。

USB PD 协议基于USB3.1，是USB3.1 中即type-c端口后提出的功率传输概念。可以为这种技术带来更大的灵活性，将充电能力扩大为目前的10倍：最高可达100瓦

<https://zhidao.baidu.com/question/1772307446156134100.html>

PD是一种通信协议，它是一种新的电源和通讯连接方式，它允许USB设备间传输最高至100W（20V/5A）的功率，同时它可以改变端口的属性，也可以使端口在DFP与UFP之间切换，它还可以与电缆通信，获取电缆的属性。

### E-Marker

**E**[lec](http://www.eeworld.com.cn/tags/LEC)**tronically Marked Cable**: [封装](http://www.eeworld.com.cn/LED/packaging/)有E-Marker芯片的USB Type-C有源电缆，DFP和UFP利用PD协议可以读取该电缆的属性：电源传输能力，数据传输能力，ID等信息。所有全功能的Type-C电缆都应该封装有E-Marker，但USB2.0 Type-C电缆可以不封装E-Marker

### 名词

Alternate Mode Adapter(AMA)：支持PD USB交替模式的设备，作为UFP存在

Alternate Mode Controller (AMC)：支持PD USB交替模式的主机，作为DFP存在

Augmented Power Data Object(APDO)：体现Source端的供电能力或者Sink的耗电能力，是一个数据对象

Atomic Message Sequence(AMS)：一个固定的信息序列，一般作为PE\_SRC\_Ready, PE\_SNK\_Ready or PE\_CBL\_Ready的开始或者结束

Binary Frequency Shift Keying (BFSK)：二进制频移键控

Biphase Mark Coding(BMC)：双相位标识编码，通过CC通信

Constant Voltage (CV)：恒定电压，不随负载变化而变化

Current Limit (CL)：电流限制

Device Policy Manager(DPM)：设备策略管理器

Dual-Role Data (DRD)：能作为DFP/UFP

End of Packet (EOP):结束包

IR Drop：在Sink和Source之间的电压降

Over-Current Protection(OCP)：过流保护

Over-Temperature Protection(OTP)：过温保护

Over-Voltage Protection(OVP)：过压保护

PD Power (PDP)：Source的功耗输出，由制造商在PDOs字段中展示

Power Data Object (PDO)：用来表示Source的输出能力和Sink的消耗能力的数据对象

Programmable Power Supply (PPS)：电源输出受程序控制

PSD：一种吃电但是没有数据的设备，如充电宝

SOP Packet：Start of Packet，所有的PD传输流程，都是以SOP Packet开始，SOP\*代表SOP，SOP’，SOP’’

Standard ID(SID)：标准ID

Standard or Vendor ID(SVID)：标准或产商ID

System Policy Manager(SPM)：系统策略管理，运行在Host端。

VCONN Powered Accessory(VPA)：由VCONN供电的附件

VCONN Powered USB Device(VPD)：由VCONN供电的设备

Vendor Data Object (VDO)：产商特定信息数据对象

Vendor Defined Message(VDM)：产商定义信息

Vendor ID (VID)：产商ID

### 状态

Disabled State：从CC引脚移除终端，如果不支持该状态，那么该端口在上电后直接是Unattached.SNK或Unattached.SRC，该状态端口不会驱动VBUS或VCONN，CC1和CC2会呈现高阻到地

ErrorRecovery State：从CC1和CC2引脚移除终端，接下来会根据端口类型转化为Unattached.SNK或Unattached.SRC，这相当于强制断开连接事件，并寻找一个新的连接。如果该状态不支持，则转化为支持的disabled状态，如果disabled状态也不支持，则转化Unattached.SNK或Unattached.SRC。，该状态端口不会驱动VBUS或VCONN，CC1和CC2会呈现高阻到地

Unattached.SNK State：端口等待检测到Source的出现，一个端口Dead Battery不供电时候进入这个状态，端口不能驱动VBUS和VCONN，CC1和CC2分别地通过Rd终止到地，当Source连接检测到会转化为AttachWait.SNK，意味着在一个CC引脚上有SNK.Rp。USB 2.0不支持USB PD可能在VBUS检测到直接转化到Attached.SNK

AttachWait.SNK State：端口检测到SNK.Rp状态在一个CC引脚上，并等待VBUS。端口不驱动VBUS或VCONN

Attached.SNK State：端口连接上了，并作为Sink操作，如果初始化进入这个状态同样作为UFP操作，Power和Data的状态改变可以通过USB PD Command。直接从Unattached.SNK转化过来是通过检测VBUS，不确定方向和可用的高于默认的USB Power

Try.SRC State：端口查询决定伙伴端口是否支持Sink，不驱动VBUS和VCONN，端口要在CC1和CC2上分别Source电流

TryWait.SNK State：端口成为Source失败，准备连接成Sink，不支持VBUS或VCONN，CC1和CC2分别通过Rd终止

Try.SNK State：端口查询决定伙伴端口是否支持Source

TryWait.SRC State：端口成为Sink失败，准备连接成Source

————————————————

版权声明：本文为CSDN博主「Chris.Cheung」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/a6588621/article/details/85165562

5.4 名词/连接状态解释

名词：

Alternate Mode Adapter(AMA)：支持PD USB交替模式的设备，作为UFP存在

Alternate Mode Controller (AMC)：支持PD USB交替模式的主机，作为DFP存在

Augmented Power Data Object(APDO)：体现Source端的供电能力或者Sink的耗电能力，是一个数据对象

Atomic Message Sequence(AMS)：一个固定的信息序列，一般作为PE\_SRC\_Ready, PE\_SNK\_Ready or PE\_CBL\_Ready的开始或者结束

Binary Frequency Shift Keying (BFSK)：二进制频移键控

Biphase Mark Coding(BMC)：双相位标识编码，通过CC通信

Configuration Channel (CC)：配置通道，用于识别、控制等

Constant Voltage (CV)：恒定电压，不随负载变化而变化

Current Limit (CL)：电流限制

Device Policy Manager(DPM)：设备策略管理器

Downstream Facing Port(DFP)：下行端口，即为HOST或者HUB下行端口

Upstream Facing Port(UFP)：上行端口，即为Device或者HUB的上行端口

Dual-Role Data (DRD)：能作为DFP/UFP

End of Packet (EOP):结束包

IR Drop：在Sink和Source之间的电压降

Over-Current Protection(OCP)：过流保护

Over-Temperature Protection(OTP)：过温保护

Over-Voltage Protection(OVP)：过压保护

PD Power (PDP)：Source的功耗输出，由制造商在PDOs字段中展示

Power Data Object (PDO)：用来表示Source的输出能力和Sink的消耗能力的数据对象

Programmable Power Supply (PPS)：电源输出受程序控制

PSD：一种吃电但是没有数据的设备，如充电宝

SOP Packet：Start of Packet，所有的PD传输流程，都是以SOP Packet开始，SOP\*代表SOP，SOP’，SOP’’

Standard ID(SID)：标准ID

Standard or Vendor ID(SVID)：标准或产商ID

System Policy Manager(SPM)：系统策略管理，运行在Host端。

VCONN Powered Accessory(VPA)：由VCONN供电的附件

VCONN Powered USB Device(VPD)：由VCONN供电的设备

Vendor Data Object (VDO)：产商特定信息数据对象

Vendor Defined Message(VDM)：产商定义信息

Vendor ID (VID)：产商ID

状态：

Disabled State：从CC引脚移除终端，如果不支持该状态，那么该端口在上电后直接是Unattached.SNK或Unattached.SRC，该状态端口不会驱动VBUS或VCONN，CC1和CC2会呈现高阻到地

ErrorRecovery State：从CC1和CC2引脚移除终端，接下来会根据端口类型转化为Unattached.SNK或Unattached.SRC，这相当于强制断开连接事件，并寻找一个新的连接。如果该状态不支持，则转化为支持的disabled状态，如果disabled状态也不支持，则转化Unattached.SNK或Unattached.SRC。，该状态端口不会驱动VBUS或VCONN，CC1和CC2会呈现高阻到地

Unattached.SNK State：端口等待检测到Source的出现，一个端口Dead Battery不供电时候进入这个状态，端口不能驱动VBUS和VCONN，CC1和CC2分别地通过Rd终止到地，当Source连接检测到会转化为AttachWait.SNK，意味着在一个CC引脚上有SNK.Rp。USB 2.0不支持USB PD可能在VBUS检测到直接转化到Attached.SNK

AttachWait.SNK State：端口检测到SNK.Rp状态在一个CC引脚上，并等待VBUS。端口不驱动VBUS或VCONN

Attached.SNK State：端口连接上了，并作为Sink操作，如果初始化进入这个状态同样作为UFP操作，Power和Data的状态改变可以通过USB PD Command。直接从Unattached.SNK转化过来是通过检测VBUS，不确定方向和可用的高于默认的USB Power

Try.SRC State：端口查询决定伙伴端口是否支持Sink，不驱动VBUS和VCONN，端口要在CC1和CC2上分别Source电流

TryWait.SNK State：端口成为Source失败，准备连接成Sink，不支持VBUS或VCONN，CC1和CC2分别通过Rd终止

Try.SNK State：端口查询决定伙伴端口是否支持Source

TryWait.SRC State：端口成为Sink失败，准备连接成Source

————————————————

版权声明：本文为CSDN博主「Chris.Cheung」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/a6588621/article/details/85165562

### Data Role/Power Role

#### Type-C的 Data Role

在USB2.0端口，USB根据数据传输的方向定义了HOST/Device/OTG三种角色，其中OTG即可作为HOST，也可作为Device，在Type-C中，也有类似的定义，只是名字有了些许修改。如下所示：

（1）DFP(Downstream Facing Port)：

下行端口，可以理解为Host或者是HUB，DFP提供VBUS、VCONN，可以接收数据。在协议规范中DFP特指数据的下行传输，笼统意义上指的是数据下行和对外提供电源的设备。

（2）UFP（Upstream Facing Port）：

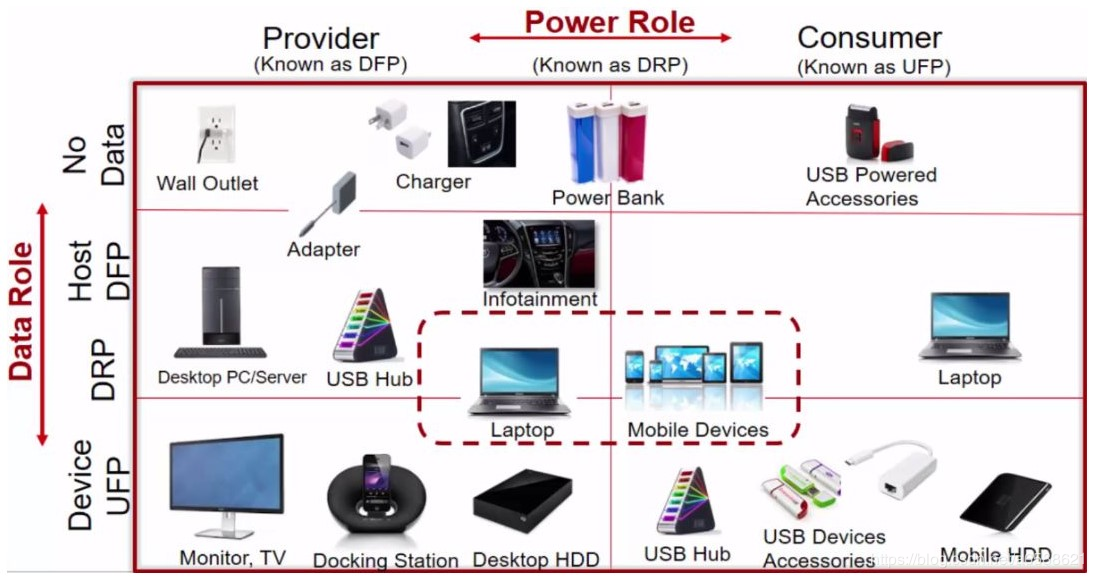
上行端口，可以理解为Device，UFP从VBUS中取电，并可提供数据。典型设备是U盘，移动硬盘。

（3）DRP（Dual Role Port）：请注意DRP分为DRD(Dual Role Data)/DRP(Dual Role Power)

双角色端口，类似于以前的OTG，DRP既可以做DFP(Host)，也可以做UFP(Device)，也可以在DFP与UFP间动态切换。典型的DRP设备是笔记本电脑。设备刚连接时作为哪一种角色，由端口的Power Role（参考后面的介绍）决定；后续也可以通过switch过程更改（如果支持USB PD协议的话）。

#### Power Role

根据USB PORT的供电（或者受电）情况，USB Type-C将port划分为Source、Sink等power角色,如下图显示常用设备的Data Role和Power Role.



Power Role 详细可以分为：

a）Source Only

b）默认Source，但是偶尔能够通过PD SWAP切换为SINK模式

c）Sink Only

d）默认SINK，但是偶尔能够通过PD SWAP切换为Source模式

e）Source/SINK 轮换

 f）Sourcing Device （能供电的Device，显示器）

g）Sinking Host（吃电的Host，笔记本电脑）

## 发展历史

自1998年以来，USB发布至今，USB已经走过20个年头有余了。在这20年间，USB-IF组织发布N种接口状态，包括A口、B口、MINI-A、MINI-B、Micro-A、Micro-B等等接口形态，由于各家产品的喜好不同，不同产品使用不同类型的插座，因此悲剧来了，我们也要常备N中不明用途的接口转接线材。

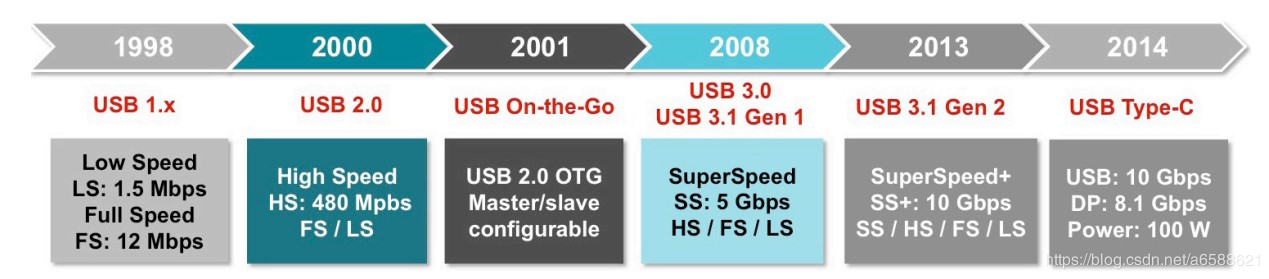
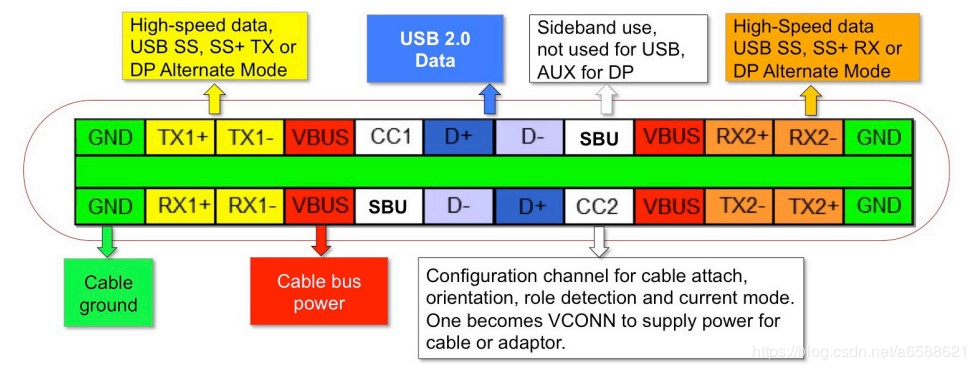


图1   USB协议发布时间节点

而对于Type-C来说，看起来USB标准化组织也是意识到统一和标准化问题，在定义标准时，除了硬件接口定义上，还增加了一部分“个性化”特点

### 定义了全新的接口形态

接口大小跟Micro USB相近，约为8.3mm x 2.5mm，支持正反插，同时也规范了对应的线材，接口定义如下（线材端只有一对USB2.0 DATA）



#### 两种插座

在插座定义上，定义了如下两种插座：

a）全功能的Type-C插座，可以用于支持USB2.0、USB3.1、等特性的平台和设备。

b）USB 2.0 Type-C插座，只可以用在支持USB2.0的平台和设备上。

#### 三种插头

在插头定义上，定义了如下三种插头：

a）全功能的Type-C插头，可以用于支持USB2.0、USB3.1、等特性的平台和设备。

b）USB 2.0 Type-C插头，只可以用在支持USB2.0的平台和设备上。

c）USB Type-C Power-Only插头，用在那些只需要供电设备上（如充电器）。

#### 三种线缆

在线缆定义上，定义了如下三种线缆：

a）两端都是全功能Type-C插头的全功能Type-C线缆。

b）两端都是USB 2.0 Type-C插头的USB 2.0 Type-C线缆。

c）只有一端是Type-C插头（全功能Type-C插头或者USB 2.0 Type-C插头）的线缆。

#### 兼容旧设备

还定义了N种为了兼容旧设备的线缆：

a）一种线缆，一端是全功能的Type-C插头，另一端是USB 3.1 Type-A插头。

b）一种线缆，一端是USB 2.0 Type-C插头，另一端是USB 2.0 Type-A插头。

c）一种线缆，一端是全功能的Type-C插头，另一端是USB 3.1 Type-B插头。

d）一种线缆，一端是USB 2.0 Type-C插头，另一端是USB 2.0 Type-B插头。

e）一种线缆，一端是USB 2.0 Type-C插头，另一端是USB 2.0 Mini-B插头。

f）一种线缆，一端是全功能的Type-C插头，另一端是USB 3.1 Micro-B插头。

g）一种线缆，一端是USB 2.0 Type-C插头，另一端是USB 2.0 Micro-B插头。

h）一种适配器，一端是全功能的Type-C插头，另一端是USB 3.1 Type-A插座。

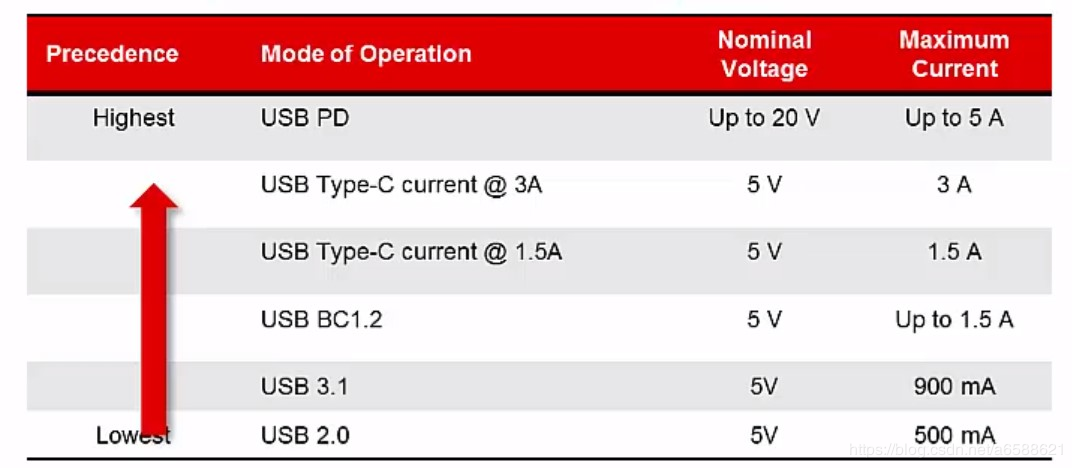
i）一种适配器，一端是USB 2.0 Type-C插头，另一端是USB 2.0 Micro-B插座

以上这些线材，我们知道，Type-A接的是HOST，所以转接线中，CC引脚需要接上拉电阻。Type-B接的是Device，因此CC引脚需要接下拉电阻。

其中，具备全功能的Type-C应该具备E-Marker功能，由于具备E-Marker，线缆能够被读到其带电流的能力、特性、线材ID等等。E-Marker的供电电源来自于VCONN，如何知道线缆需要VCONN呢？线缆会通过下拉的电阻Ra，Source检测到之后会提供VCONN。

### 传输速率，供电效能

最大传输速度10Gb/s,即是USB 3.1 Gen2标准，也支持4 Lane DP模式，传输高清图像，在供电部分，最大可以支持100W（20V/5A



### 个性化”协商机制

由于端口一致，线材两端接口也一直，为了能够区分两端USB设备的角色（Host/Device），必须有一套协商机制，便于进行角色确认，这部分通过CC(Configuration Channel)管脚进行设置。后面随着PD规范的面世，CC脚开始被用来做简单的半双工通信，用来完成POWER供给的协商

### 强悍的一统天下的态势

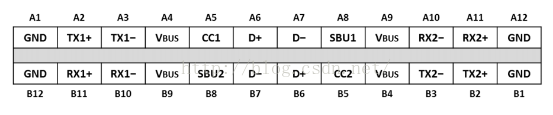
由于Type-C的扩展功能（SBU1/SBU2），大部分配件诸如耳机、视频接口、Debug接口等等都可以实现兼容设计，成功逆袭以往所有的USB标准，成功上位

## 硬件原理

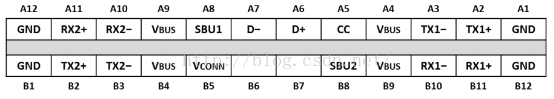
### 引脚

引脚解释，如图2所示：

母口：



公头：



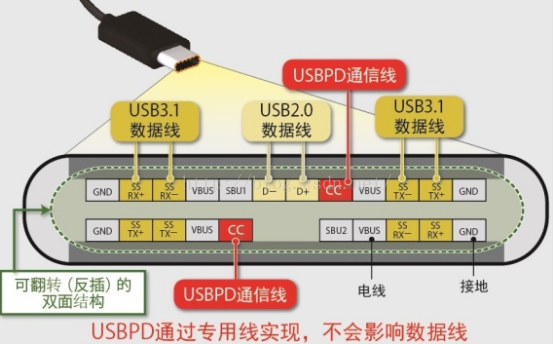
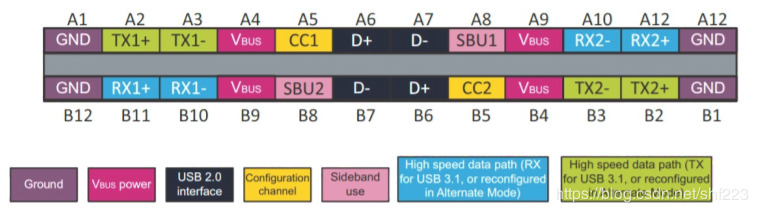


图2 引脚映射图



辅助信号sub1和sub2(Side band use)，在特定的一些传输模式时才用

d+和d-是来兼容USB之前的标准的

这里说一下，USB3.0只有一组RX/TX，速度是5Gb，USB Type-C为了保证正反都可以插就用了两组，但实际上数据传输还是只用了一组RX/TX,速度就已经达到10Gb了。如果后面升级协议，两组都传的话就和DisplayPort一样20Gb了

数据传输主要有TX/RX两组差分信号，CC1和CC2是两个关键引脚，作用很多：

• 探测连接，区分正反面，区分DFP和UFP，也就是主从

• 配置Vbus，有USB Type-C和USB Power Delivery两种模式

• 配置Vconn，当线缆里有芯片的时候，一个cc传输信号，一个cc变成供电Vconn

• 配置其他模式，如接音频配件时，dp，pcie时

电源和地都有4个，这就是为什么可以支持到100W的原因

### **工作流程**

上图DFP (Downstream Facing Port)也就是主，UFP (Upstream Facing Port)为从。除了DFP、UFP，还有个DRP (Dual Role port)，DRP可以做DFP也可以做UFP。当DPR接到UFP，DRP转化为DFP。当DRP接到DFP，DRP转化为UFP。两个DRP接在一起，这时就是任意一方为DFP，另一方为UFP。

在DFP的CC pin有上拉电阻Rp，在UFP有下拉电阻Rd。未连接时，DFP的VBUS是无输出的。连接后，CC pin相连，DFP的CC pin会检测到UFP的下拉电阻Rd，说明连接上了，DFP就打开Vbus电源开关，输出电源给UFP。而哪个CC pin(CC1，CC2)检测到下拉电阻就确定接口插入的方向，顺便切换RX/TX。

电阻Rd=5.1k，电阻Rp为不确定的值，根据前面的图看到USB Type-C有几种供电模式，靠什么来甄别?就靠Rp的值，Rp的值不一样，CC pin检测到的电压就不一样，然后来控制DFP端执行哪种供电模式。

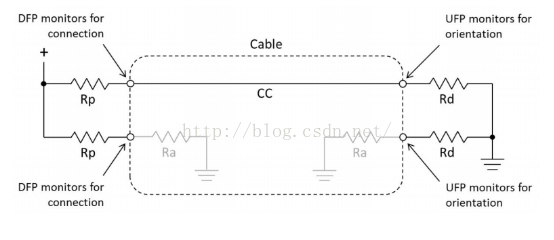
需要注意的是，上图里画了两个CC，实际上在不含芯片的线缆里只有一根cc线。

含芯片的线缆也不是两根cc线，而是一根cc，一根Vconn，用来给线缆里的芯片供电(3.3V或5V)，这时就cc端没有下拉电阻Rd，而是下拉电阻Ra，800-1200欧。

当CC pin两个都接了下拉电阻<=Ra，DFP进入音频配件模式，左右声道，mic都俱全

### USB-C设备识别方法

USB-C设备识别方法如图3所示



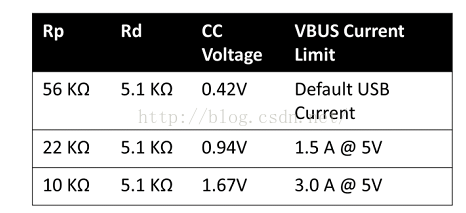


表1：USB-C Rp和Rd具体值(以上拉电压5V为例)  
正反可插如图4所示。

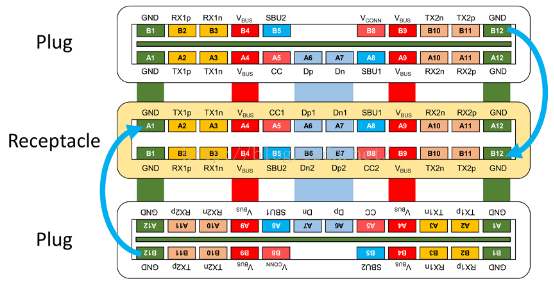


图4：正反可插图示

### DRP设备是如何工作的

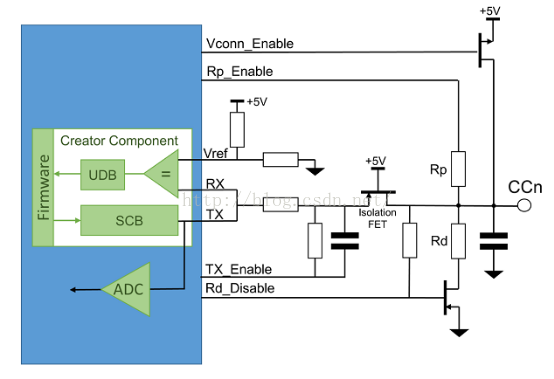


图5 DRP设备电路原理

DRP状态机伪代码示例如下所示(该示例伪代码状态机来源于P124 Figure 4-15 of USB Type-C Specification Release 1.1)，为便于理解下一章节讲述的USB PD，所以也加入了USB PD的简单状态。

drp\_toggle\_timeout = 50ms;

tc\_sm(void \*arg)

{

    switch (state) {

    case UFP\_STATE\_DETACHED:

        if (DFP连接) {

            state = UFP\_STATE\_ATTACHED\_DEBOUNCE;

            debounce\_timeout = 100ms;

        } else if (drp\_toggle\_timeout) {

            set\_drp\_mode(CC电阻上拉);

            state = DFP\_STATE\_DETACHED;

        }

        break;

    case UFP\_STATE\_ATTACHED\_DEBOUNCE:

        if (DFP仍然连接) {

            state = UFP\_STATE\_ATTACHED;

        } else {

            state = UFP\_STATE\_DETACHED;

        }

        break;

    case UFP\_STATE\_ATTACHED:

        // TODO

        state = UFP\_STATE\_DISCOVERY;

        break;

    case UFP\_STATE\_DISCOVERY:

        // TODO: PD negotiate

        break;

    [...]

    case DFP\_STATE\_DETACHED:

        if (UFP连接) {

            state = DFP\_STATE\_ATTACHED\_DEBOUNCE;

            debounce\_timeout = 100ms;

        } else if (drp\_toggle\_timeout) {

            set\_drp\_mode(CC电阻下拉);

            state = UFP\_STATE\_DETACHED;

        }

        break;

    case DFP\_STATE\_ATTACHED\_DEBOUNCE:

        if (UFP仍然连接) {

            state = DFP\_STATE\_ATTACHED;

        } else {

            state = DFP\_STATE\_DETACHED;

        }

        break;

    case DFP\_STATE\_ATTACHED:

        // TODO

        state = DFP\_STATE\_DISCOVERY;

        break;

    case DFP\_STATE\_DISCOVERY:

        // TODO: PD negotiate

        break;

    } // end of switch

}

防伪以及线缆验证

————————————————

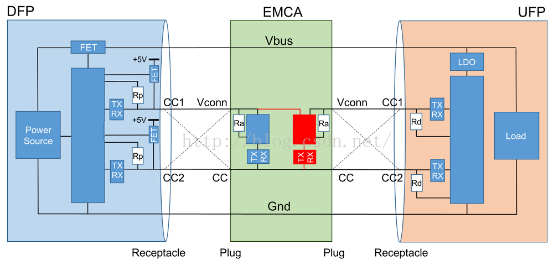


图 6：eMarker(EMCA)原理简介

### QA

问题1：USB-C的接口是否一定要IC支持？  
Reference docs:  
Type-C离一统江山差的只是一个液态金属  
USB发展的新趋势 USB PD解读，<http://www.itocp.com/htmls/36/n-4336-2.html>

2 Abbreviations

ARC：Argonant RISC Core

AT91SAM9260：SAM means Smart ARM-based Microcontroller

ATMEL SAMBA：ATMEL Smart ARM-based Microcontroller Boot Assistant

CC2530：TI ChipCon2530

DWC2：Design Ware Controller 2，Apple的嵌入式设备，包括iPad和iPhone都是使用的DWC2

ISP1161：Philips' Integrated host Solution Pairs 1161，“Firms introduce USB host controllers”，https://www.eetimes.com/document.asp?doc\_id=1290054

SL811HS：Cypress/ScanLogic 811 Host/Slave，性能上与ISP1161（Integrated host Solution Pairs 1161）相当

TDI：TransDimension Inc.，该公司首先发明了将TT集成到EHCI RootHub中的方法，这样对于嵌入式系统来说，就省去了OHCI/UHCI的硬件，同时降低了成本，作为对该公司的纪念，Linux内核定义了宏ehci\_is\_TDI(ehci)

TLV：TI Low Value，高性价比

TPS：TI Performance Solution

TT：Transaction Translator（事务转换器，将USB2.0的包转换成USB1.1的包）

## 握手协商/Alt Mode

USB Type-C的插座中有两个CC脚，以下的角色检测，都是通过CC脚进行的，但是对于插头、或者线缆正常只有一个CC引脚，两个端口连接在一起之后，只存在一个CC引脚连接，通过检测哪一个CC有连接，就可以判断连接的方向。如果USB线缆中有需供电的器件，其中一个CC引脚将作为VCONN供电。

### CC引脚有如下作用

a）检测USB Type-C端口的插入，如Source接入到Sink

b）用于判断插入方向，翻转数据链路

c）在两个连接的Port之间，建立对应的Data Role

d）配置VBUS，通过下拉电阻判断规格，在PD协商中使用，为半双工模式

e）配置VCONN

 f）检测还有配置其他可选的配置模式，如耳机或者其他模式

### 连接方向、Data Role、Power Role角色检测

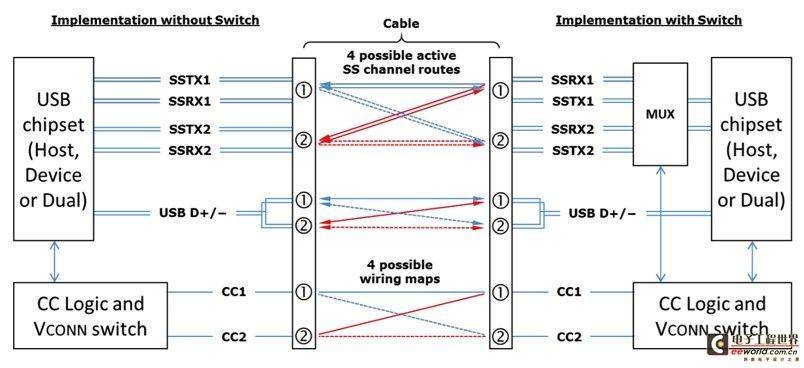
#### **设备连接与分开检测**

DFP需要检测到CC管脚上有某个电压时，判断UFP设备已插入或拔出，来提供和管理VBUS。当没有UFP设备插入时，必须关闭VBUS。因此所有的DFP设备需要CC逻辑检测与控制芯片

#### **插入方向检测**

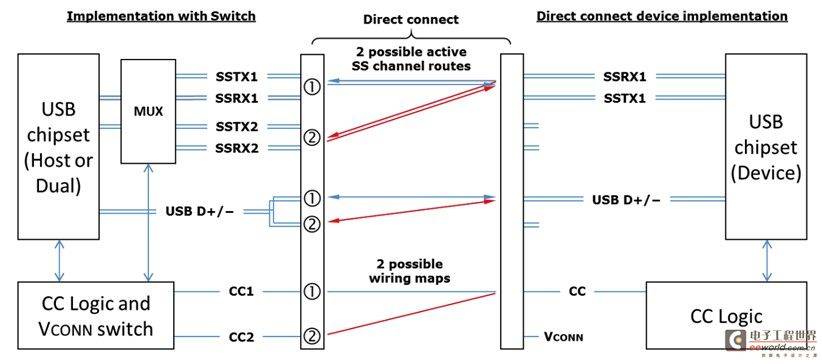
如图1，虽然USB Type-C插座和插头的两排管脚上下对称，USB数据信号都有两组重复的通道，但主控芯片通常只有一组TX/RX和D+/-通道。由于USB2.0的数据率最高只有480Mbps, 可以不考虑信号走线的阻抗连续性而得到较好地数据传输质量，因此USB2.0的D+/-信号可以不被MUX控制而直接从主控芯片一分二连接至USB Type-C插座的两组D+/-管脚上。但USB3.0或者USB3.1的数据率高达5Gbps或者10Gbps，如果信号线还是被简单地一分二的话，不连续的信号线阻抗将严重破坏数据传输质量，因此必须由MUX切换来保证信号路径阻抗的一致性，以确保信号传输质量。下图中右侧所示的MUX从TX1/RX1和TX2/RX2中选择一路连接至主控芯片，而这个MUX就必须被CC Logic控制。

    因此，在USB2.0应用中，无需考虑方向检测问题，但USB3.0或者USB3.1应用中，必须考虑方向检测问题



但必须注意的是在USB3.0/USB3.1的应用中，有一种情况也可以不需要MUX，即不需要方向检测，如图2所示，不管是正插还是反插，左侧主机都可以根据CC管脚上的状态来切换MUX来连通USB3.0/USB3.1信号。此场景发生在右侧设备永远是UFP的情况下，比如U盘，移动硬盘等。

因此，USB3.0/USB3.1应用中，除UFP设备以外的所有设备都需要CC逻辑检测与控制芯片。



USB Type-C直接连接数据走线逻辑模型

* [收藏](http://news.eeworld.com.cn/xfdz/2015/0323/article_40868.html)
* [评论](http://news.eeworld.com.cn/xfdz/2015/0323/article_40868.html#comment)0

**分享到**

[微博](javascript:;)

[QQ](javascript:;)

[微信](javascript:;)

[LinkedIn](javascript:;)

   USB Type-C凭借其自身强大的功能，在Apple, Intel, Google等厂商的强势推动下，必将迅速引发一场USB接口的革命，并将积极影响我们日常生活的方方面面。本文讨论一个重要的专业问题：USB Type-C设备到底是否需要CC逻辑检测与控制芯片？

    要回答这个问题，我们得先从基本概念谈起。

**DFP(Downstream Facing Port)**: 下行端口，可以理解为Host，DFP提供VBUS，也可以提供数据。典型的DFP设备是电源适配器，因为它永远都只是提供电源。

**UFP(Upstream Facing Port)**: 上行端口，可以理解为Device，UFP从VBUS中取电，并可提供数据。典型设备是U盘，移动硬盘，因为它们永远都是被读取数据和从VBUS取电，当然不排除未来可能出现可以作为主机的U盘。

**DRP(Dual Role Port)**: 双角色端口，DRP既可以做DFP(Host)，也可以做UFP(Device)，也可以在DFP与UFP间动态切换。典型的DRP设备是电脑(电脑可以作为USB的主机，也可以作为被充电的设备（苹果新推出的MAC Book Air）)，具OTG功能的手机(手机可以作为被充电和被读数据的设备，也可以作为主机为其他设备提供电源或者读取U盘数据)，移动电源(放电和充电可通过一个USB Type-C，即此口可以放电也可以充电)。

**CC（Configuration Channel）**：配置通道，这是USB Type-C里新增的关键通道，它的作用有检测USB连接，检测正反插，USB设备间数据与VBUS的连接建立与管理等。

**USB PD(USB Power Delivery)**: PD是一种通信协议，它是一种新的电源和通讯连接方式，它允许USB设备间传输最高至100W（20V/5A）的功率，同时它可以改变端口的属性，也可以使端口在DFP与UFP之间切换，它还可以与电缆通信，获取电缆的属性。

**Electronically Marked Cable**: 封装有E-Marker芯片的USB Type-C有源电缆，DFP和UFP利用PD协议可以读取该电缆的属性：电源传输能力，数据传输能力，ID等信息。所有全功能的Type-C电缆都应该封装有E-Marker，但USB2.0 Type-C电缆可以不封装E-Marker。

    USB Type-C设备DFP-to-UFP配置流程与VBUS管理有如下主要流程：

**设备连接与分开检测：**DFP需要检测到CC管脚上有某个电压时，判断UFP设备已插入或拔出，来提供和管理VBUS。当没有UFP设备插入时，必须关闭VBUS。因此所有的DFP设备需要CC逻辑检测与控制芯片。

**插入方向检测：**如图1，虽然USB Type-C插座和插头的两排管脚上下对称，USB数据信号都有两组重复的通道，但主控芯片通常只有一组TX/RX和D+/-通道。由于USB2.0的数据率最高只有480Mbps, 可以不考虑信号走线的阻抗连续性而得到较好地数据传输质量，因此USB2.0的D+/-信号可以不被MUX控制而直接从主控芯片一分二连接至USB Type-C插座的两组D+/-管脚上。但USB3.0或者USB3.1的数据率高达5Gbps或者10Gbps，如果信号线还是被简单地一分二的话，不连续的信号线阻抗将严重破坏数据传输质量，因此必须由MUX切换来保证信号路径阻抗的一致性，以确保信号传输质量。下图中右侧所示的MUX从TX1/RX1和TX2/RX2中选择一路连接至主控芯片，而这个MUX就必须被CC Logic控制。

    因此，在USB2.0应用中，无需考虑方向检测问题，但USB3.0或者USB3.1应用中，必须考虑方向检测问题。

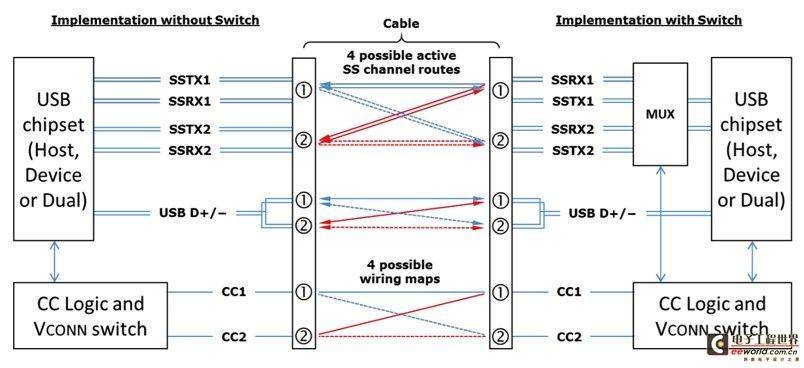
[](http://8.eewimg.cn/news/uploadfile/2015/0323/20150323095205419.jpg)

图1 USB Type-C数据走线逻辑模型

    但必须注意的是在USB3.0/USB3.1的应用中，有一种情况也可以不需要MUX，即不需要方向检测，如图2所示，不管是正插还是反插，左侧主机都可以根据CC管脚上的状态来切换MUX来连通USB3.0/USB3.1信号。此场景发生在右侧设备永远是UFP的情况下，比如U盘，移动硬盘等。

因此，USB3.0/USB3.1应用中，除UFP设备以外的所有设备都需要CC逻辑检测与控制芯片。

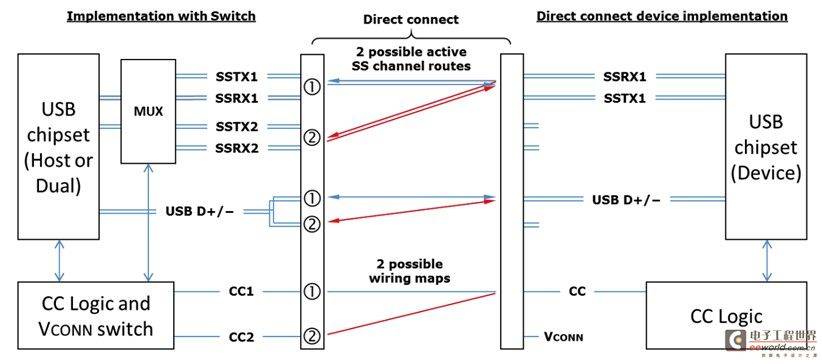
[](http://8.eewimg.cn/news/uploadfile/2015/0323/20150323095358139.jpg)

图2 USB Type-C直接连接数据走线逻辑模型

#### **建立DFP-to-UFP和VBUS管理与检测**

    DRP在待机模式下每50ms在DFP和UFP间切换一次。当切换至DFP时，CC管脚上必须有一个上拉至VBUS的电阻Rp或者输出一个电流源，当切换至UFP时，CC管脚上必须有一个下拉至GND的电阻Rd。此切换动作必须由CC Logic芯片来完成。

当DFP检测到UFP插入之后才可以输出VBUS，当UFP拔出以后必须关闭VBUS。此动作必须由CC Logic芯片来完成。

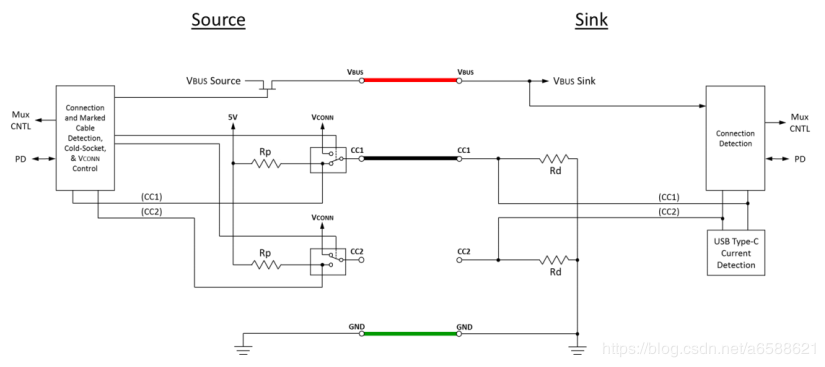
#### **USB Type-C VBUS电流检测与使用**

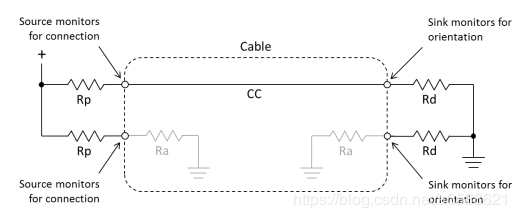
    USB Type-C中新增了电流检测与使用功能，新增三种电流模式：默认的USB电源模式(500mA/900mA)，1.5A，3.0A。三种电流模式由CC管脚来传输和检测，对于需要广播电流输出能力的DFP而言，需要通过不同值的CC上拉电阻Rp来实现；对于UFP而言，需要检测CC管脚上的电压值来获取对方DFP的电流输出能力。

#### **USB PD通信**

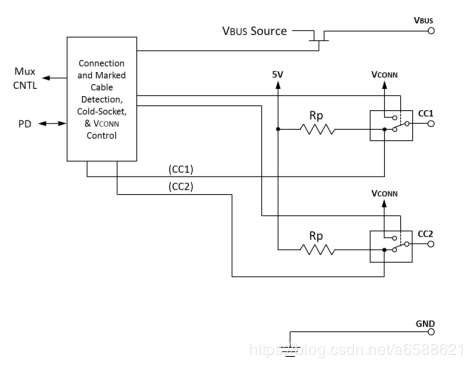
    USB PD看似只是电源传输与管理的协议，实际上它可改变端口角色，可与有源电缆通讯，允许DFP成为受电设备等诸多高级功能，因此支持PD的设备必须采用CC Logic芯片。

#### SourceSink Connection





如图所示，Source端CC引脚为上拉，Sink端CC引脚为下拉。握手过程为接入后检测到有效连接（即一端为Host一端为Device），随后检测线材供电能力，再进行USB枚举。



a）Source端使用一个MOSFET去控制电源，初始状态下，FET为关闭状态

b）Source端CC1/CC2均上拉至高电平，同时检测是否有Sink插入，当检测到有Rd下拉电阻时，说明Sink被检测到。Rp的阻值表明Host能够提供的功率水平。

c）Source端根据Cable中哪一个CC引脚为Rd下拉，去翻转USB的数据链路，同时决定另外一个CC引脚为VCONN

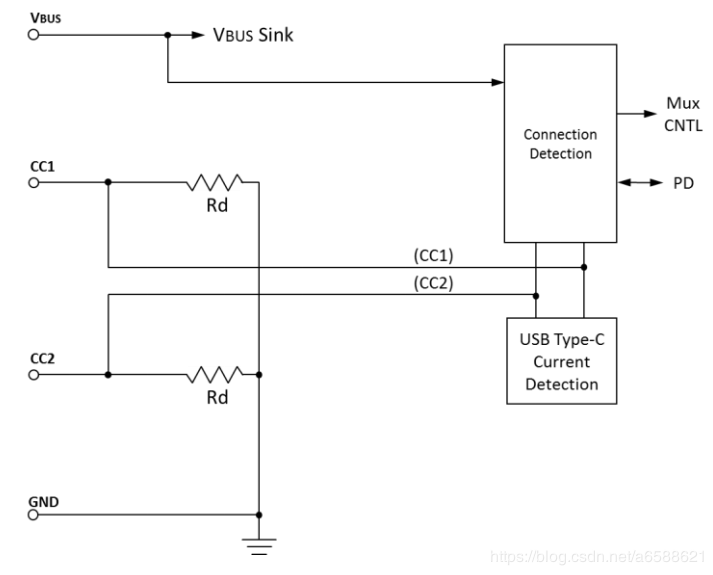
d）在此之后，Source打开VBUS，同时VCONN供电

e）Source可以动态调整Rp的值，去表示给Sink的电流发送变化，告知SINK最大可以使用的电流

 f）Source会持续检测Rd的存在，一旦连接断开，电源将会被关闭

g）如果Source支持高级功能（PD或者Alternate Mode），将通过CC引脚进行通信

如下图指示了SINK端CC1和CC2框架：



a）SINK的两个CC引脚均通道Rd下拉到GND

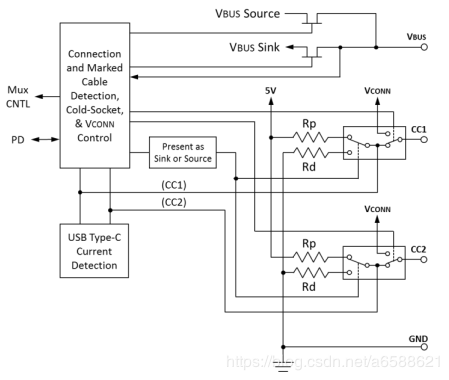
b）SINK通过检测VBUS，来判断Source的连接与否

c）SINK通过CC引脚上拉的特性，来检测目前的USB通信链路（翻转）

d）SINK可选地去检测Rp的值，去判断Source可提供的电流。同时管理自身的功耗，保证不超过Source提供的最大范围

e）同样的，如果支持高级功能，通过CC引脚进行通信。

如下图指示DRP的CC引脚在链接之前的架构

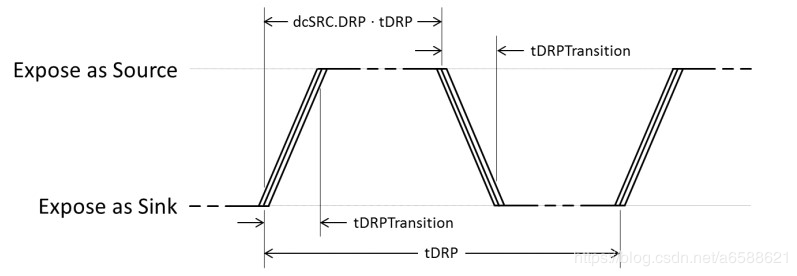


A 当作为Source存在的时候，DRP使用MOSFET控制VBUS供电与否

b）DRP使用Switch去切换自身身份作为Source，或者是SINK

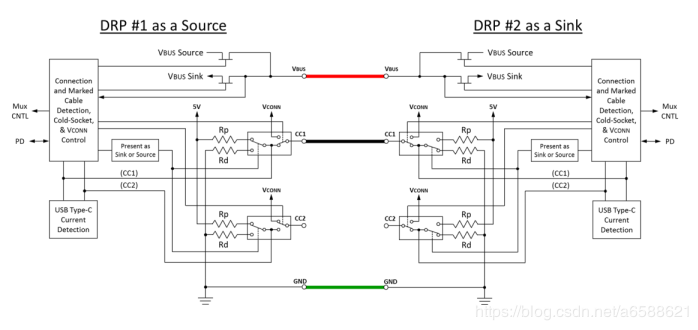
c）DRP存在一套机制，分三种情况，去决定自身是SINK或者是Source，去建立两者间彼此的角色。

情况1：不使用PD SWAP，随机变成Source/SINK中的任意一个，CC脚波形为方波



情况2：自身倾向于作为Source，执行Try.SRC，问对面能不能做SINK呀，我做Source

情况3：与情况2相反，自身倾向作为SINK，执行Try.SNK，你做Source，我做小弟



当然还存在Source&Source，SINK&SINK这种搞基模式，唯一的结果就是一直停留在Unattached.SNK/Unattached.SRC，无法终成眷属。

————————————————

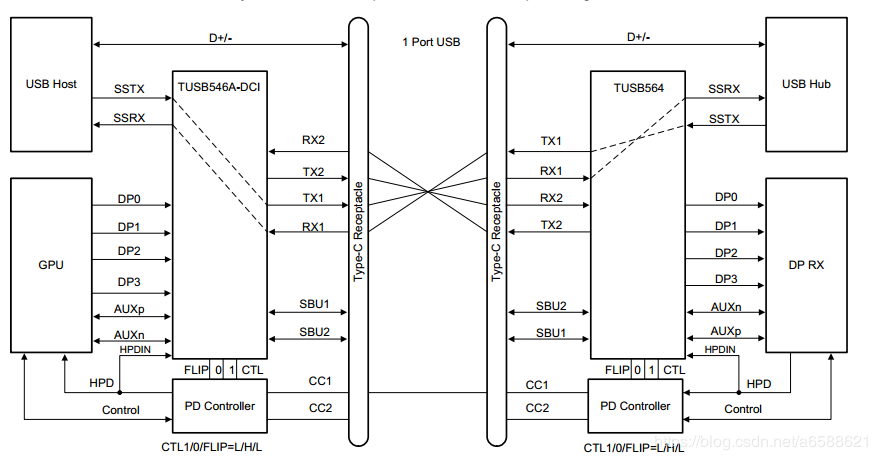
## 如何进行数据链路的切换

以TUSB546(DFP)，TUSB564(UFP)为例子

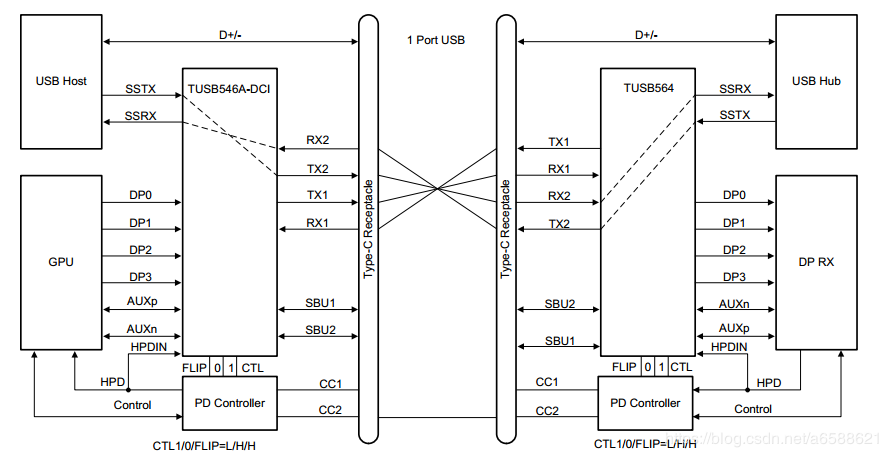
### 纯USB3.0

前者的使用例子如笔记本电脑、后者的使用例子如Monitor

如下图，两端设备会根据插入方向，切换数据链路。图X插入连接为CC1，因此TUSB564切换到TX1/RX1

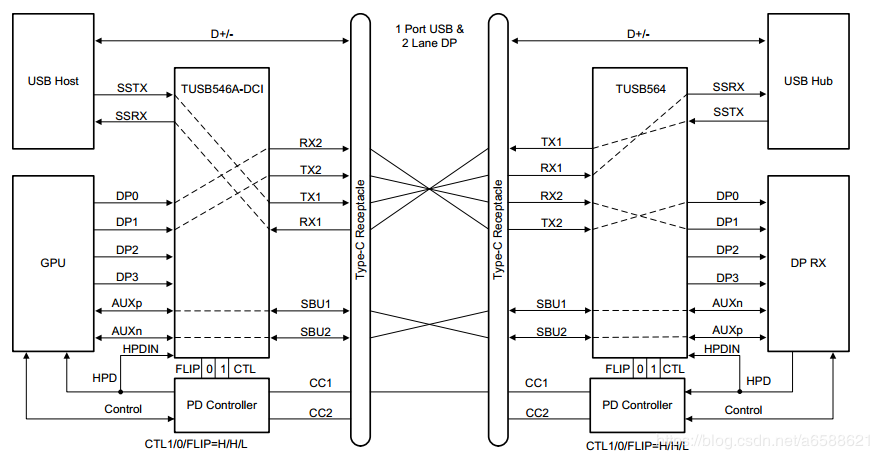


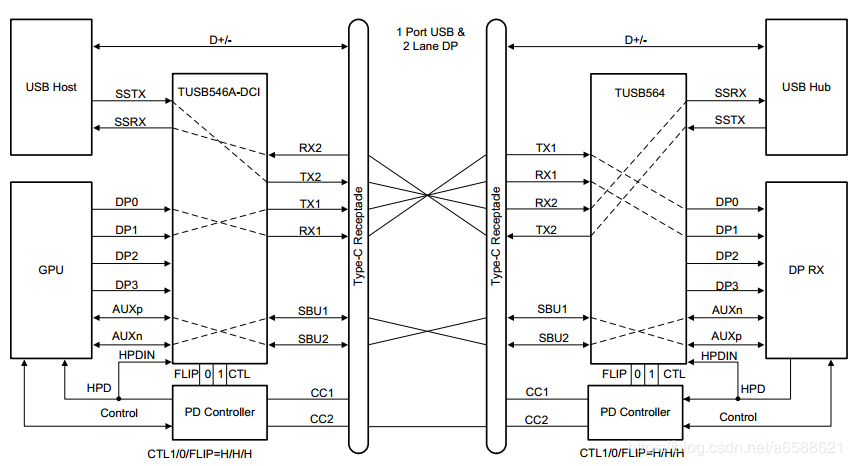
图中插入连接为CC2，因此TUSB564切换到TX2/RX2，也就是根据CC引脚插入，识别插入方向



### USB3.1和2 LANE of DisplayPort

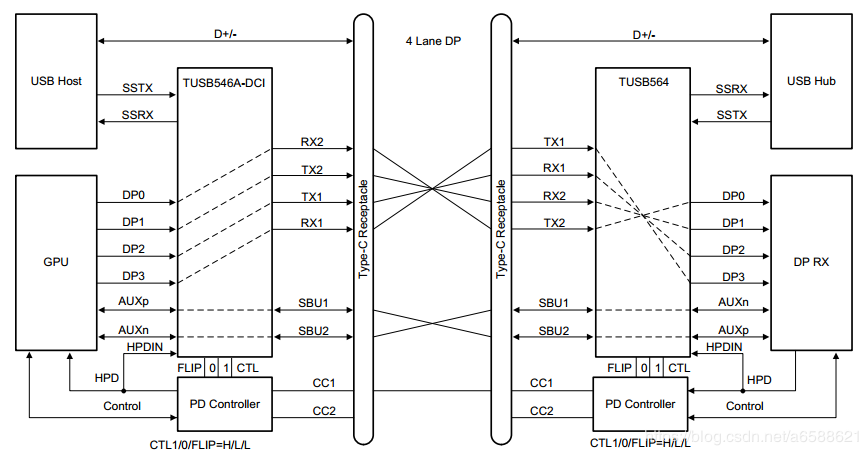
切换原理如上，需要注意的是，DP信号是使用SBUx进行传输

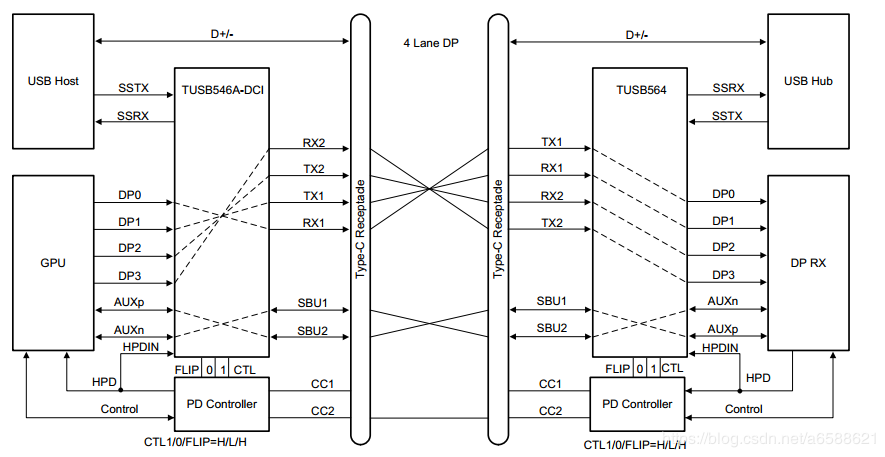




### 纯DP模式 4 lane

4lane是什么意思，通道？？





问题思考：如何确定是DP 4 lane模式或者是DP 2 lane+USB3.0 模式？

通过CC引脚，利用PD协议沟通，协商，PD Controler 发起请求，并得到回应

## PD协议

PD协议是Power Delivery，简单来说是一种快速充电标准。

允许设备通过Type-C端口协商功率，如：

1. 5V 3A

2. 9V 3A

3. 15V 3A

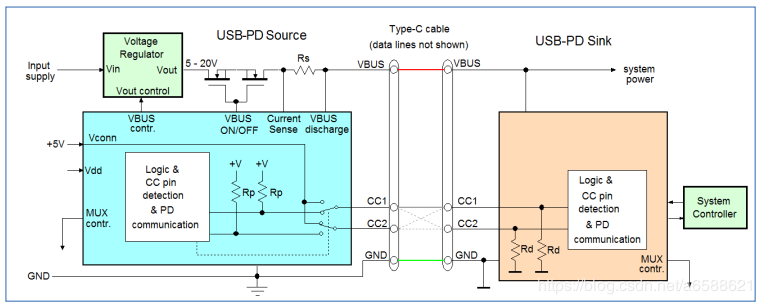
4. 20V 5A

最大可到100W的输出

PD协商通过功率提供者（provider）和功率消费者（Consumer）之间的信息交换完成的，功率提供者如手机充电器，功率消费者如手机

### 系统框图

包含PD协议的Type-C 系统从Source到SINK的系统框图大致如下。

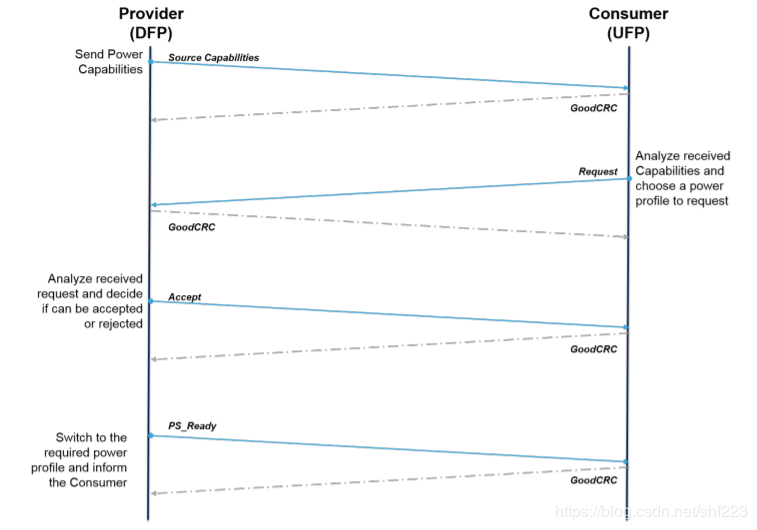


在Source的内部包含了一个电压转换器，且受到PD控制器控制，他会根据输入电压的条件以及最高可输出规格需求，此电压转换器可以是BUCK、Boost、Buck-Boost或者反激转换器。整个通信过程都在PD控制器的管控之下，USB PD还有一个开关，用于切换VCONN电源（电缆包含电子标签时用到）。

当电缆接通之后，PD协议的SOP通信就开始在CC线上进行，以此来选择电源传输的规格，此部分由Sink端向Source端询问能够提供的电源配置参数（5V/9V/12V/15V/20V）

### 时序

USB PD协议规定CC（Configuration Channel）作为PD协商信息交换的引脚



下面的描述Provider简化为P，Consumer简化为C）

1. P首先发起PD协商，向C发送P具有的Power能力的消息，也就是P支持哪些功率类型

2. C收到P发送的Power能力的消息后，分析P的Power能力并选择其中一个Power配置发送给P

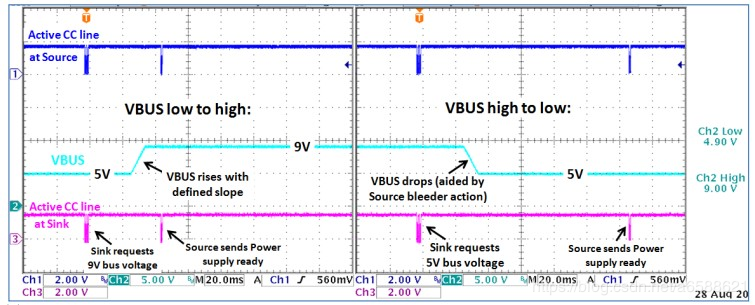
3. P收到C请求的power配置，决定是否接受这个请求

4. 切换到C请求的Power配置并通知C

整个过程在都需要做循环冗余校验，如果校验失败，消息会被忽略。如果通信错误持续，通信过程将会被软重置并重新建立连接，如果错误仍然持续，那么系统将会被硬重置。

### 申请9V电压协议

如下波形为SINK 控制器申请一个9V电压输出的例子。



（1）SINK端发起SOP，申请获取Source能提供的规格资料

（2）Source回复能提供的规格列表

（3）SINK回复选择的电压规格，并带上所需要的电流参数，并发出相应的请求

（4）Source接受请求，并且把VBUS由5V抬升到9V

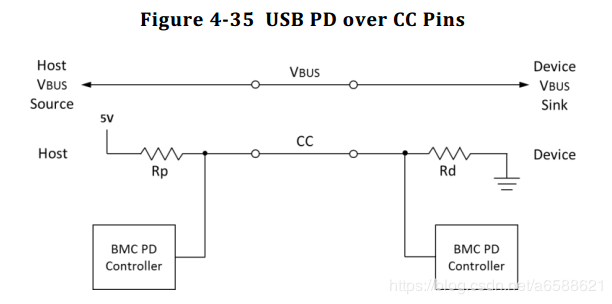
（5）在电压变化期间，SINK的电流会保持尽可能小，Source端VBUS到达9V并稳定之后，会发出Ready信号

（6）SINK端电流逐步抬升，若SINK需要降低电压，会重复以上过程

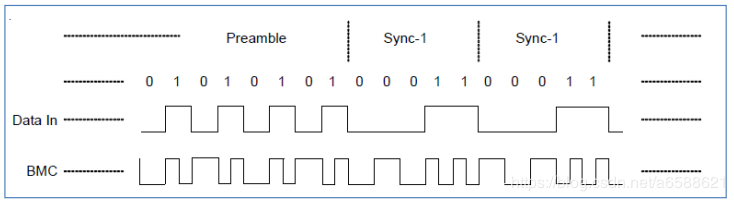
需要注意的是，在电压下降期间，Source为了让电压快速降低，Source会打开放电电路，达到额定值之后，Source会等待一段时间，电压稳定之后再发出Ready信号给SINK。

这种沟通方式的好处就是能确保任何电源的变化都能在SINK和SOURCE的规格范围内，避免出现不可控情况

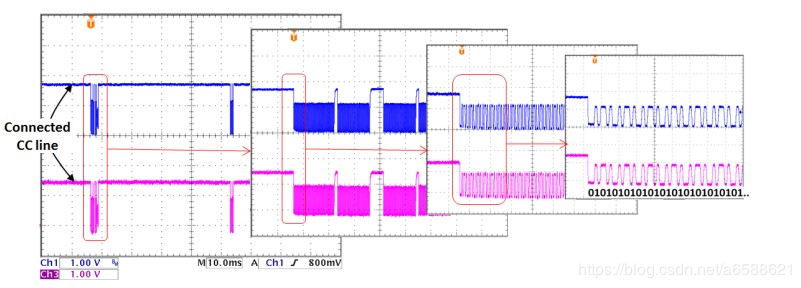
PD协议的通信编码为Bi-phase Mark Coded (BMC)，通过CC脚进行通信



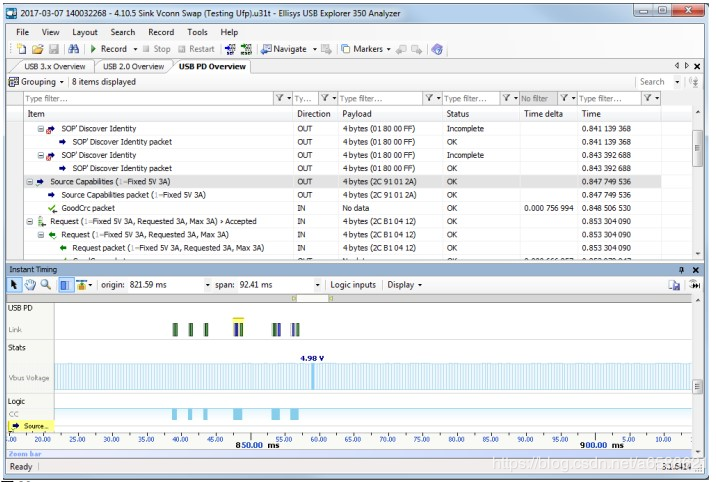
BMC码是一种单线通信编码，数据1的传输，需要有一次高/低电平之间的切换过程，而0的传输则是固定的高电平或者低电平。每一个数据包都包含有0/1交替的前置码，起始码（SOP），报文头，数据位，CRC以及结束码（EOP）



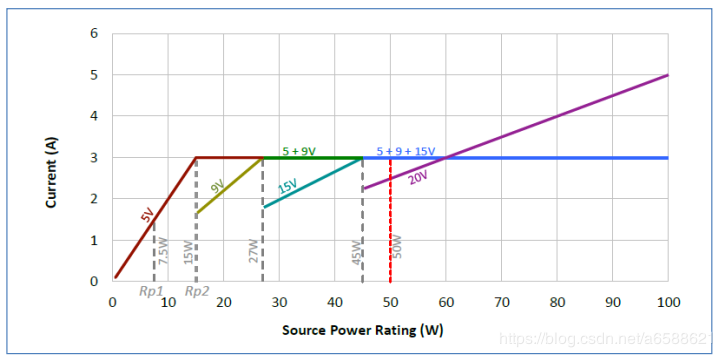
如下图所示，展开后的CC脚PD通信波形



BMC编码的通信，也可以使用分析仪进行分析，用来抓取每个数据包，并且获得数据包的作用，如电压电流等



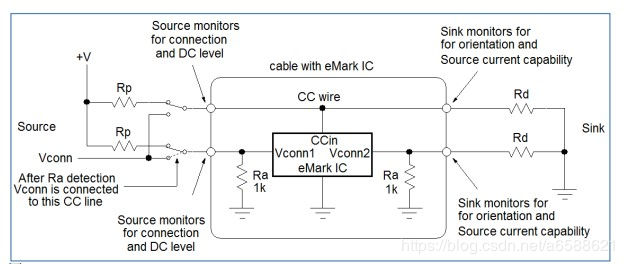
PD3.0规范中，定义了以下电源配置清单：



对于5V/9V/15V来说，最大的电流为3A，在20V的配置当中，如果是普通的电流，则最大能够支持20V/3A，即60W，如果使用的是带了E-Marker的线缆，则供电能达到20V5A，即100W。

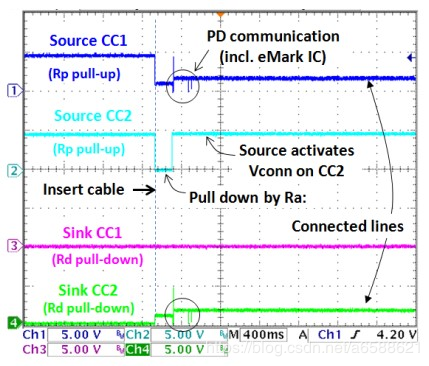
### E-Marker

支持超高速数据传输（USB3.1）或者是供电电流超过3A，电缆都必须使用E-Marker进行标识。线缆中有IC，他们需要从VCONN获得电源。



我们注意到，线缆中有1K的下拉电阻Ra，这样在线缆插入的时候，Source会识别到CC1和CC2电压下降的情况，具体的电压会告诉主机那个端子被Sink的5.1K下拉，那个端子被线缆的1K电阻下拉。因此线缆的插入方向也可以被识别到。Source就可以通过开关，给E-Marker提供VCONN。

如下图为带E-Marker的情况

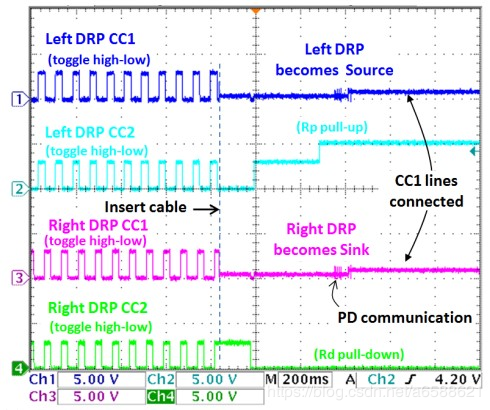


电缆接通之后，Source的一根CC线被来自VCONN的1K拉低

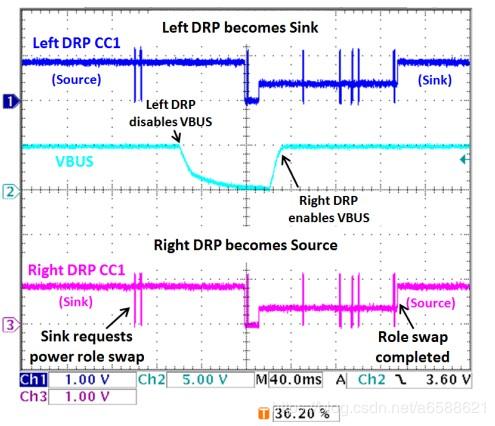
（2）Source检测到此电压，知道电缆中有E-Marker，因此切换VCONN到对应的CC引脚

（3）在之后，PD通信将会包含Source和E-Marker之间的通信（SOP'&SOP''）Source和Sink之间为SOP

当设备为DRP时，设备的CC1和CC2为方波，一旦连接，CC端都会发生改变。



在本次连接当中，左边的DRP做了Source，右边的DRP作为SINK，也有可能翻转过来。也可以本来就设定SOURCE或者SINK优先。连接之后，想翻转也是允许的，只要发起角色变换请求就可以了。



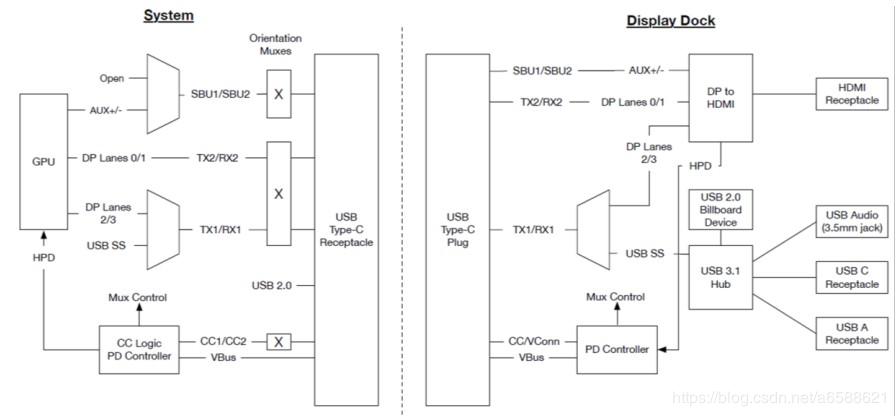
## Type-C的其他模式

USB Type-C支持语音附件以及Debug模式，USB Type-C接口的耳机如果只作为UFP且因为其功耗较小而无需检测DFP的供电能力时，无需CC Logic芯片。

    综上，所有的DFP(如电源适配器)，所有的DRP(如电脑，手机，平板，移动电源), 所有需要检测DFP电流输出能力的UFP，所有支持PD的设备，都需要CC逻辑检测与端口控制芯片。换句话说，只有因为功耗较低而不需要检测电流能力的UFP(U盘，耳机，鼠标等)才不需要CC逻辑检测端口控制芯片

### Display Port Alternate Mode

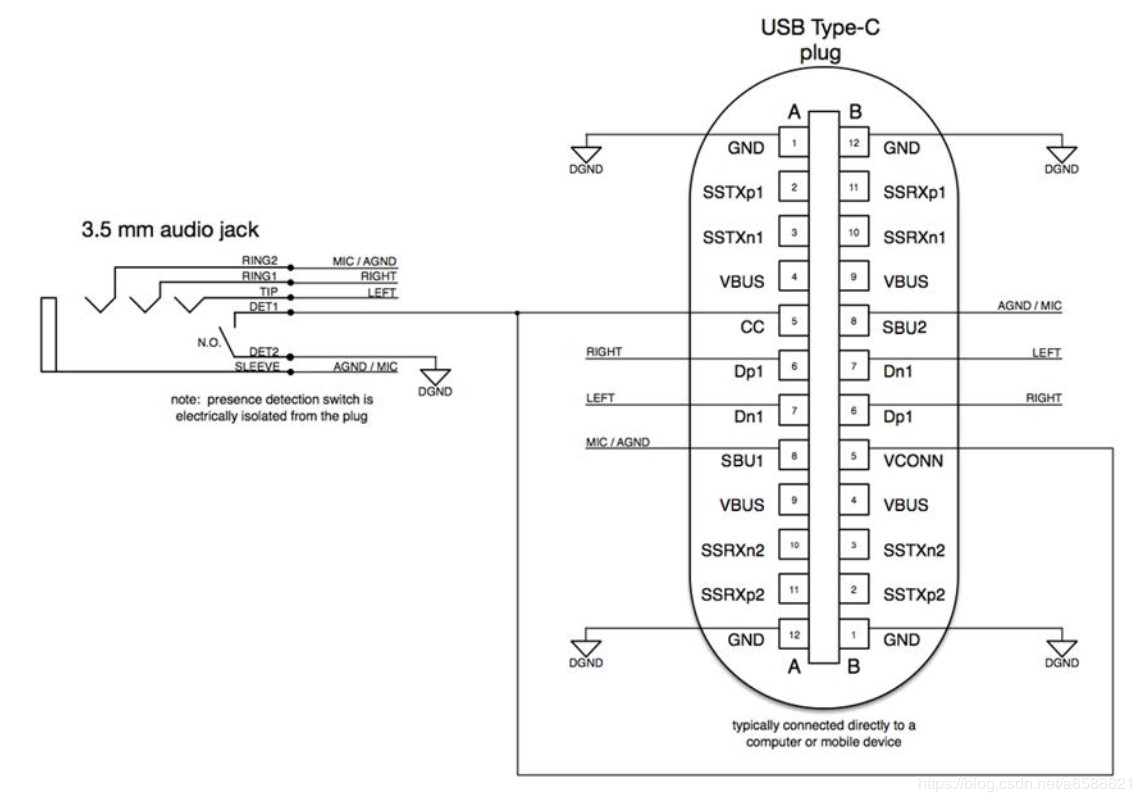
系统会通过USB PD协议中VDMs的信息通信（CC引脚通信），去告知支持Display Port模式。在这个模式当中，USB SuperSpeed 信号允许部分传输USB，部分传输DP信号。



### 3.3.2  Audio Adapter Accessory Mode

       如下图，为3.5mm音频输入口转Type-C端口，USB2.0链路被用来传输模拟音频信号，若带MIC，MIC信号则连接在SBU引脚上，在这个模式当中，电源可以提供到500mA电流。

       Host端如何识别到音频模式呢？把CC引脚和VCON连接，并且下拉电阻小于Ra/2(则小于400ohm)，或者分别对地，下拉电阻小于Ra(小于800ohm)，则Host会识别为音频模式。



### 3.3.3 Debug Accessory Mode （DAM）

       在DAM下，连接软体和硬体提供可视化调试和控制的系统，使用较少。

## 参考

细谈Type-C、PD原理（一）

<https://blog.csdn.net/a6588621/article/details/85049539>

USB Type-C的基本原理（连载一）

<https://blog.csdn.net/zoosenpin/article/details/49963031>

USB Type-C设备是否需要CC逻辑芯片

<http://news.eeworld.com.cn/xfdz/2015/0323/article_40868.html>

# 调试方法

linux kernel usb调试方法：

  sysfs/debugfs

  usbmon

 Dynamic debug interface

 Tracepoints

调试信息文件路径：

ls  /sys/bus/usb/devices/

cat  /sys/kernel/debug/usb/devices

cat  /proc/bus/input/devices  # 查找系统的usb输入设备

## sysfs/debugfs

### ls  /sys/bus/usb/devices/

kg411:/sys/bus/usb/devices $ ls

1-0:1.0 2-0:1.0 2-1 2-1:1.0 3-0:1.0 3-1 3-1:1.0 3-1:1.1 3-1:1.2 3-1:1.3 3-1:1.4 3-1:1.5 4-0:1.0 usb1 usb2 usb3 usb4

kg411:/sys/bus/usb/devices $ ls

1-0:1.0 2-1 3-0:1.0 3-1:1.0 3-1:1.2 3-1:1.4 4-0:1.0 5-1 5-1.4:1.0 5-1:1.0 usb1 usb3 usb5

2-0:1.0 2-1:1.0 3-1 3-1:1.1 3-1:1.3 3-1:1.5 5-0:1.0 5-1.4 5-1.4:1.1 6-0:1.0 usb2 usb4 usb6

### cat  /sys/kernel/debug/usb/devices

2|kg411:/sys/kernel/debug/usb $ cat devices

T: Bus=06 Lev=00 Prnt=00 Port=00 Cnt=00 Dev#= 1 Spd=5000 MxCh= 0

B: Alloc= 0/800 us ( 0%), #Int= 0, #Iso= 0

D: Ver= 3.00 Cls=09(hub ) Sub=00 Prot=03 MxPS= 9 #Cfgs= 1

P: Vendor=1d6b ProdID=0003 Rev= 4.04

S: Manufacturer=Linux 4.4.83 xhci-hcd

S: Product=xHCI Host Controller

S: SerialNumber=xhci-hcd.8.auto

C:\* #Ifs= 1 Cfg#= 1 Atr=e0 MxPwr= 0mA

I:\* If#= 0 Alt= 0 #EPs= 1 Cls=09(hub ) Sub=00 Prot=00 Driver=(none)

E: Ad=81(I) Atr=03(Int.) MxPS= 4 Ivl=256ms

T: Bus=05 Lev=00 Prnt=00 Port=00 Cnt=00 Dev#= 1 Spd=480 MxCh= 1

B: Alloc= 0/800 us ( 0%), #Int= 0, #Iso= 0

D: Ver= 2.00 Cls=09(hub ) Sub=00 Prot=01 MxPS=64 #Cfgs= 1

P: Vendor=1d6b ProdID=0002 Rev= 4.04

S: Manufacturer=Linux 4.4.83 xhci-hcd

S: Product=xHCI Host Controller

S: SerialNumber=xhci-hcd.8.auto

C:\* #Ifs= 1 Cfg#= 1 Atr=e0 MxPwr= 0mA

I:\* If#= 0 Alt= 0 #EPs= 1 Cls=09(hub ) Sub=00 Prot=00 Driver=hub

E: Ad=81(I) Atr=03(Int.) MxPS= 4 Ivl=256ms

T: Bus=05 Lev=01 Prnt=01 Port=00 Cnt=01 Dev#= 2 Spd=480 MxCh= 4

D: Ver= 2.00 Cls=09(hub ) Sub=00 Prot=01 MxPS=64 #Cfgs= 1

P: Vendor=05e3 ProdID=0608 Rev=85.37

S: Product=USB2.0 Hub

C:\* #Ifs= 1 Cfg#= 1 Atr=e0 MxPwr=100mA

I:\* If#= 0 Alt= 0 #EPs= 1 Cls=09(hub ) Sub=00 Prot=00 Driver=hub

E: Ad=81(I) Atr=03(Int.) MxPS= 1 Ivl=256ms

**T: Bus=05 Lev=02 Prnt=02 Port=03 Cnt=01 Dev#= 3 Spd=12 MxCh= 0**

**D: Ver= 2.00 Cls=02(comm.) Sub=00 Prot=00 MxPS=64 #Cfgs= 1**

**P: Vendor=fff0 ProdID=0008 Rev= 0.01**

**S: Manufacturer=KJI VCPSerial**

**S: Product=KJI VCPSerial**

**C:\* #Ifs= 2 Cfg#= 1 Atr=c0 MxPwr=100mA**

**I:\* If#= 0 Alt= 0 #EPs= 1 Cls=02(comm.) Sub=02 Prot=00 Driver=cdc\_acm**

**E: Ad=81(I) Atr=03(Int.) MxPS= 16 Ivl=8ms**

**I:\* If#= 1 Alt= 0 #EPs= 2 Cls=0a(data ) Sub=00 Prot=00 Driver=cdc\_acm**

**E: Ad=82(I) Atr=02(Bulk) MxPS= 64 Ivl=0ms**

**E: Ad=03(O) Atr=02(Bulk) MxPS= 64 Ivl=0ms**

T: Bus=04 Lev=00 Prnt=00 Port=00 Cnt=00 Dev#= 1 Spd=12 MxCh= 1

B: Alloc= 0/900 us ( 0%), #Int= 0, #Iso= 0

D: Ver= 1.10 Cls=09(hub ) Sub=00 Prot=00 MxPS=64 #Cfgs= 1

P: Vendor=1d6b ProdID=0001 Rev= 4.04

S: Manufacturer=Linux 4.4.83 ohci\_hcd

S: Product=Generic Platform OHCI controller

S: SerialNumber=fe3a0000.usb

C:\* #Ifs= 1 Cfg#= 1 Atr=e0 MxPwr= 0mA

I:\* If#= 0 Alt= 0 #EPs= 1 Cls=09(hub ) Sub=00 Prot=00 Driver=**hub**

E: Ad=81(I) Atr=03(Int.) MxPS= 2 Ivl=255ms

T: Bus=03 Lev=00 Prnt=00 Port=00 Cnt=00 Dev#= 1 Spd=480 MxCh= 1

B: Alloc= 0/800 us ( 0%), #Int= 1, #Iso= 0

D: Ver= 2.00 Cls=09(hub ) Sub=00 Prot=00 MxPS=64 #Cfgs= 1

P: Vendor=1d6b ProdID=0002 Rev= 4.04

S: Manufacturer=Linux 4.4.83 ehci\_hcd

S: Product=EHCI Host Controller

S: SerialNumber=fe380000.usb

C:\* #Ifs= 1 Cfg#= 1 Atr=e0 MxPwr= 0mA

I:\* If#= 0 Alt= 0 #EPs= 1 Cls=09(hub ) Sub=00 Prot=00 Driver=**hub**

E: Ad=81(I) Atr=03(Int.) MxPS= 4 Ivl=256ms

T: Bus=03 Lev=01 Prnt=01 Port=00 Cnt=01 Dev#= 2 Spd=480 MxCh= 0

D: Ver= 2.00 Cls=00(>ifc ) Sub=00 Prot=00 MxPS=64 #Cfgs= 1

P: Vendor=2ca3 ProdID=001f Rev=ff.ff

S: Manufacturer=KJI

S: Product=KJI

S: SerialNumber=0123456789ABCDEF

C:\* #Ifs= 6 Cfg#= 1 Atr=c0 MxPwr= 0mA

A: FirstIf#= 0 IfCount= 2 Cls=e0(wlcon) Sub=01 Prot=03

A: FirstIf#= 4 IfCount= 2 Cls=02(comm.) Sub=02 Prot=01

I:\* If#= 0 Alt= 0 #EPs= 1 Cls=e0(wlcon) Sub=01 Prot=03 Driver=rndis\_hos

E: Ad=82(I) Atr=03(Int.) MxPS= 8 Ivl=32ms

I:\* If#= 1 Alt= 0 #EPs= 2 Cls=0a(data ) Sub=00 Prot=00 Driver=rndis\_hos

E: Ad=81(I) Atr=02(Bulk) MxPS= 512 Ivl=0ms

E: Ad=08(O) Atr=02(Bulk) MxPS= 512 Ivl=0ms

I:\* If#= 2 Alt= 0 #EPs= 2 Cls=08(stor.) Sub=06 Prot=50 Driver=usb-stora

E: Ad=83(I) Atr=02(Bulk) MxPS= 512 Ivl=0ms

E: Ad=09(O) Atr=02(Bulk) MxPS= 512 Ivl=125us

I:\* If#= 3 Alt= 0 #EPs= 2 Cls=ff(vend.) Sub=43 Prot=01 Driver=(none)

E: Ad=84(I) Atr=02(Bulk) MxPS= 512 Ivl=0ms

E: Ad=0a(O) Atr=02(Bulk) MxPS= 512 Ivl=0ms

I:\* If#= 4 Alt= 0 #EPs= 1 Cls=02(comm.) Sub=02 Prot=01 Driver=cdc\_acm

E: Ad=86(I) Atr=03(Int.) MxPS= 10 Ivl=32ms

I:\* If#= 5 Alt= 0 #EPs= 2 Cls=0a(data ) Sub=00 Prot=00 Driver=cdc\_acm

E: Ad=85(I) Atr=02(Bulk) MxPS= 512 Ivl=0ms

E: Ad=0b(O) Atr=02(Bulk) MxPS= 512 Ivl=0ms

T: Bus=02 Lev=00 Prnt=00 Port=00 Cnt=00 Dev#= 1 Spd=12 MxCh= 1

B: Alloc= 15/900 us ( 2%), #Int= 1, #Iso= 0

D: Ver= 1.10 Cls=09(hub ) Sub=00 Prot=00 MxPS=64 #Cfgs= 1

P: Vendor=1d6b ProdID=0001 Rev= 4.04

S: Manufacturer=Linux 4.4.83 ohci\_hcd

S: Product=Generic Platform OHCI controller

S: SerialNumber=fe3e0000.usb

C:\* #Ifs= 1 Cfg#= 1 Atr=e0 MxPwr= 0mA

I:\* If#= 0 Alt= 0 #EPs= 1 Cls=09(hub ) Sub=00 Prot=00 Driver=hub

E: Ad=81(I) Atr=03(Int.) MxPS= 2 Ivl=255ms

T: Bus=02 Lev=01 Prnt=01 Port=00 Cnt=01 Dev#= 2 Spd=12 MxCh= 0

D: Ver= 2.00 Cls=ff(vend.) Sub=47 Prot=d0 MxPS=64 #Cfgs= 1

P: Vendor=2ca3 ProdID=1501 Rev= 1.01

S: Manufacturer=KJI

S: Product=KJI Virtual Joystick

S: SerialNumber=1234567890

C:\* #Ifs= 1 Cfg#= 1 Atr=c0 MxPwr=500mA

I:\* If#= 0 Alt= 0 #EPs= 2 Cls=ff(vend.) Sub=47 Prot=d0 Driver=xpad

E: Ad=01(O) Atr=03(Int.) MxPS= 64 Ivl=7ms

E: Ad=81(I) Atr=03(Int.) MxPS= 64 Ivl=7ms

T: Bus=01 Lev=00 Prnt=00 Port=00 Cnt=00 Dev#= 1 Spd=480 MxCh= 1

B: Alloc= 0/800 us ( 0%), #Int= 0, #Iso= 0

D: Ver= 2.00 Cls=09(hub ) Sub=00 Prot=00 MxPS=64 #Cfgs= 1

P: Vendor=1d6b ProdID=0002 Rev= 4.04

S: Manufacturer=Linux 4.4.83 ehci\_hcd

S: Product=EHCI Host Controller

S: SerialNumber=fe3c0000.usb

C:\* #Ifs= 1 Cfg#= 1 Atr=e0 MxPwr= 0mA

I:\* If#= 0 Alt= 0 #EPs= 1 Cls=09(hub ) Sub=00 Prot=00 Driver=hub

E: Ad=81(I) Atr=03(Int.) MxPS= 4 Ivl=256ms

### cat  /proc/bus/input/devices  XXX

# 查找系统的usb输入设备

没啥用的

## usbmon**抓包分析**

usbmon源码路径 Kernel\drivers\usb\mon

### 环境配置

在Kconfig默认配置

default y 就可以编译了

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

usbmon的相关命令行：

$ sudo mount -t debugfs none\_debugs /sys/kernel/debug

$ sudo modprobe usbmon

#Verify that bus sockets are present. 我的电脑有3个usb插口。可以看到每个插口有usbmon

$ sudo ls /sys/kernel/debug/usb/usbmon

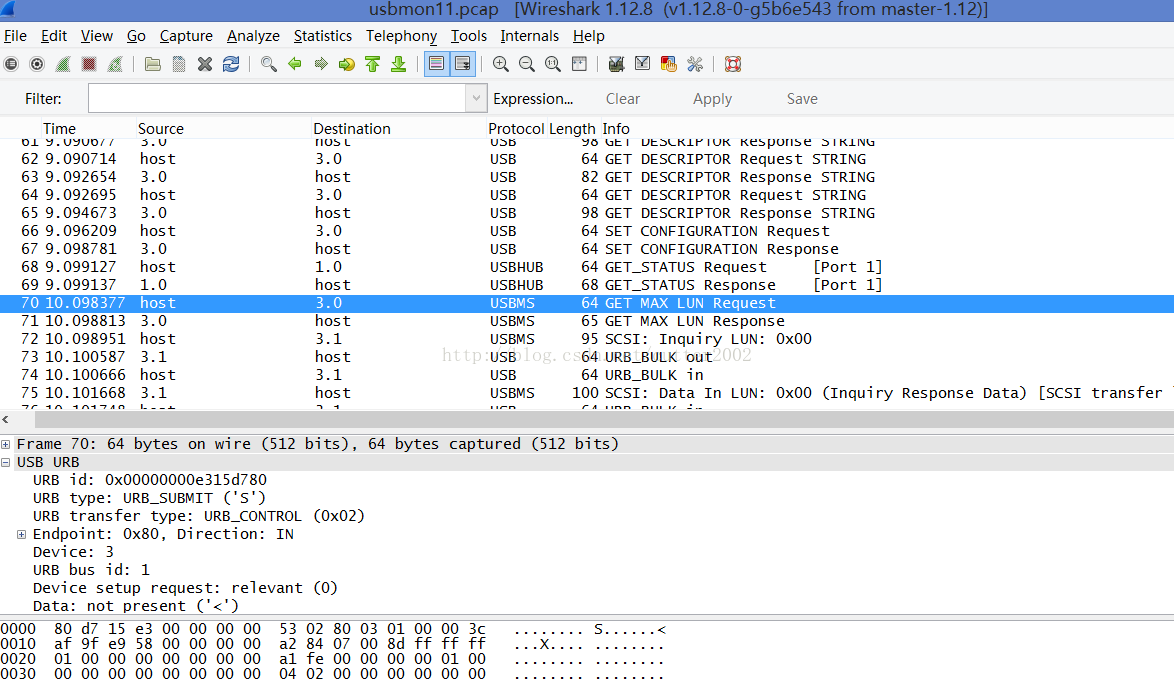
0s 0u 1s 1t 1u 2s 2t 2u

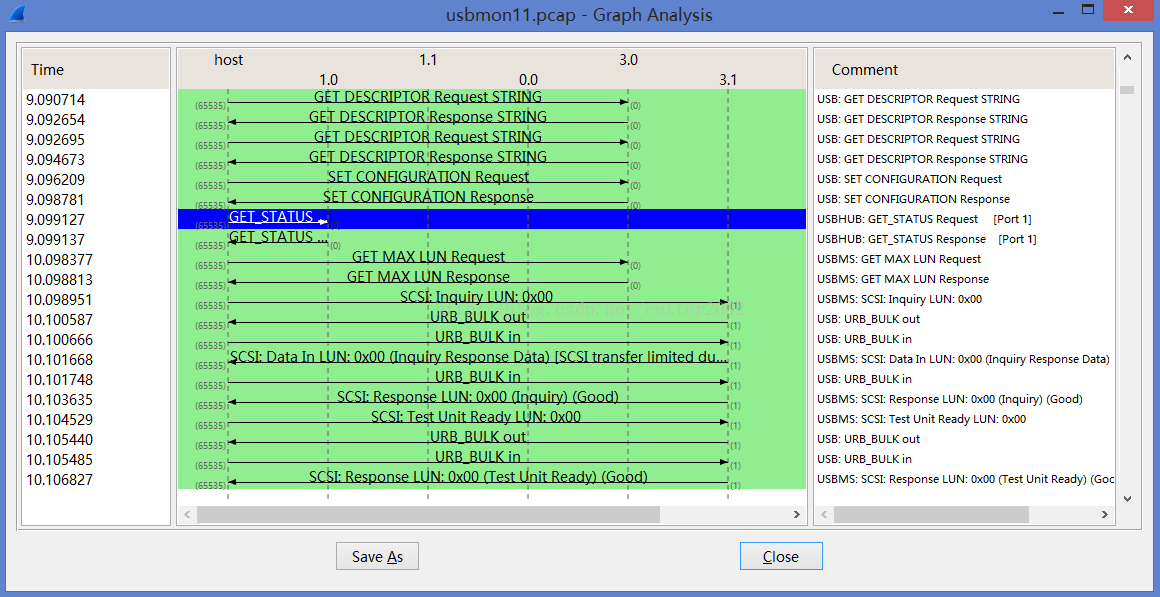
$ls /dev | grep usb

usbmon0 usbmon1 usbmon2

$sudo tcpdump -i usbmon1 -w usbmon11.pcap # usb抓包

抓包U盘插入usb接口过程数据包，Wireshark的分析结果。





### 实测

2|ag411:/sys/kernel/debug/usb $ ls

devices ohci uvcvideo

## 错误码

arch/arm64/include/generated/asm/errno.h

#include **<asm-generic/errno.h>**

### error-110

解决方式：1.这里的error-110，实际上式连接超时的错误，至于为什么会出现这个错误，还不明所以，后来搜索到资料说USB2.0的驱动有些问题，会导致这样的问题，和USB相关的模块是ehci\_hcd模块，你可以通过lsmod命令查看你当前的kernel是否加载ehci\_hcd模块，若加载了你可以使用modeprobe -r ehci\_hcd把该模块卸载掉，之后你再试试看设备能否被发现；如果还是不行，你找到系统的开机脚本在开机的时候不加载ehci\_hcd模块，我便是这样在在开机的时候不加载ehci\_hcd的模块，以后热插拔便恢复了正常

————————————————

原文链接：<https://blog.csdn.net/huangxiaohu_coder/article/details/13267525>

## 参考

<https://blog.csdn.net/cutter2002/article/details/69808345>

# device descriptor read64, error -62解决方法总结

android linux 开发usb错误-device descriptor read64, error -62解决方法总结

在目录kernel26\_h/drivers/usb/host/ohci-s3c2410.c文件中将下面函数的内同修改为  
  
  
  
static void s3c2410\_start\_hc(struct platform\_device \*dev, struct usb\_hcd \*hcd)  
  
{  
  
       struct s3c2410\_hcd\_info \*info = dev->dev.platform\_data;  
  
  
  
       //======  
  
       unsigned long upllvalue =  (0x38<<12)|(0x02<<4)|(0x01);  
  
       unsigned long upllvalue1 = (0x38<<12)|(0x02<<4)|(0x02);  
  
       dev\_dbg(&dev->dev, "s3c2410\_start\_hc:\n");  
  
//=====  
  
       \_\_raw\_writel(upllvalue,S3C2410\_UPLLCON);  
  
       mdelay(20);  
  
       \_\_raw\_writel(upllvalue1,S3C2410\_UPLLCON);  
  
       mdelay(20);  
  
//=====  
  
       clk\_enable(clk);  
  
       mdelay(10);  
  
       if (info != NULL) {  
  
              info->hcd       = hcd;  
  
              info->report\_oc = s3c2410\_hcd\_oc;  
  
  
  
              if (info->enable\_oc != NULL) {  
  
                     (info->enable\_oc)(info, 1);  
  
              }  
  
       }  
  
}  
  
主要功能是在USB寄存器初始化的之前给它较高的频率，等初始化完成后在将频率恢复为正常的48MHZ。

**linux-2.6.28系统移植s3c6410开发板USB不能识别的处理**

初始化OK,一插上usb就报如下错误：

/ # usb 1-1: new full speed USB device using s3c2410-ohci and address 2  
usb 1-1: device descriptor read/64, error -62  
usb 1-1: device descriptor read/64, error -62  
usb 1-1: new full speed USB device using s3c2410-ohci and address 3  
usb 1-1: device descriptor read/64, error -62  
usb 1-1: device descriptor read/64, error -62  
usb 1-1: new full speed USB device using s3c2410-ohci and address 4  
usb 1-1: device not accepting address 4, error -62  
usb 1-1: new full speed USB device using s3c2410-ohci and address 5  
usb 1-1: device not accepting address 5, error -62  
hub 1-0:1.0: unable to enumerate USB device on port 1

解决方法：

根据终端打印的错误

cd include/asm-generic/errno.h  
 u-boot/include/asm-arm/errno.H

找到:  
#define ETIME 62 /\*timer expired\*/  
再由:  
error-codes.txt 去找usb error code  
http://ftp.gnu.org/tmp/linux-libre-fsf2\_2.6.28/linux-2.6.28/Documentation/usb/error-codes.txt

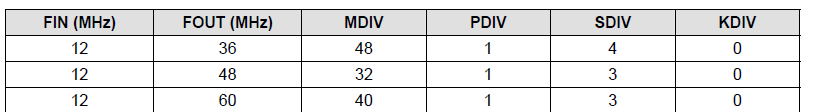
-ETIME (\*\*)      No response packet received within the prescribed  
           bus turn-around time.  This error may instead be  
            reported as -EPROTO or -EILSEQ.

由此可以判断,这个错误与USB设备超时有关。报告这个错误的地方在drivers/usb/core/hub.c中的hub\_port\_init部分，由于usb\_get\_device\_descriptor获取usb设备资讯的时候产生了超时，这样基本可以确定三种情况，1.USB设备及介面有问题，2、usbcore有问题3、usb driver有问题。

我们可以很容易的派出1.2的可能性，问题应该在usb driverimplement部分造成。2.6的内核usb driver把usb规范中对usb的操作集中到了core里面，针对不同设备的implement 分别归为host、gadget、storage等。基本确定问题就在ohci-s3c2410.c中。

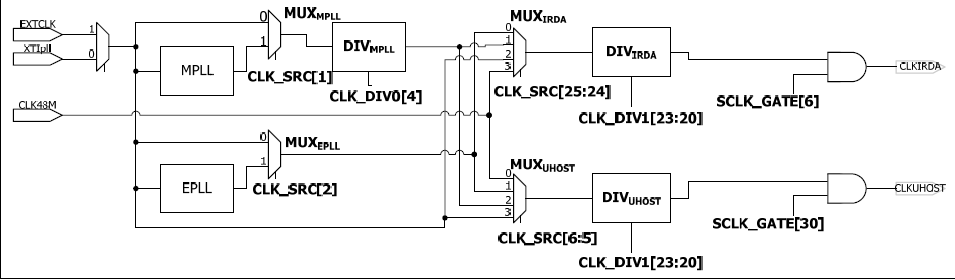
原来是USB Host的48MHz时钟没有起来。

s3c6410支持三个PLL分别是APLL，MPLL和EPLL。APLL为ARM提供时钟，产生ARMCLK，MPLL为所有和AXI/AHB/APB相连的模块提供时钟，产生HCLK和PCLK，EPLL为特殊的外设提供时钟，产生SCLK。



如图所示为EPLL\_CON的 M、 P 和 S的取值。

根据s3c6410的数据手册：



如图所示，描述的是用于IrDA和USB host 的时钟发生器，通常USB借口需要48M的操作时钟。

从图中可也以说明,HCLK\_GATE,PCLK\_GATE和SCL\_GATE控制时钟操作。如果一个位设置，则通过每个时钟分频器相应的时钟将会被提供，否则，将被屏蔽。

HCLK\_GATE控制HCLK，用于每个Ips。每个IP的AHB接口逻辑被独立地屏蔽，以减少动态电力消耗。PCLK\_GATE控制PCLK。通过SCLK\_GATE时钟被控制。

根据上图EPLL通道写出一下程序。

58 #define EPLL\_CON01               0

359

360 #define UPLL\_SRC\_MASK    ((1<<2)|(3<<5))

361 #define UPLL\_SRC         ((1<<2)|(1<<5))

362 #define UPLL\_DIV1\_MASK   (0xf<<20)

363 #define UPLL\_DIV1        (0<<20)

364 #define UPLL\_GATE\_MASK   (1<<30)

365 #define UPLL\_GATE        (1<<30)

void set\_upll(void)

368 {

369         unsigned int tmp;

370

371         while(\_\_raw\_readl(S3C\_EPLL\_CON0)!=EPLL\_CON00)

372         \_\_raw\_writel(EPLL\_CON00,S3C\_EPLL\_CON0);

373

374         while(\_\_raw\_readl(S3C\_EPLL\_CON1)!=EPLL\_CON01)

375         \_\_raw\_writel(EPLL\_CON01,S3C\_EPLL\_CON1);

376

377         while(((tmp= \_\_raw\_readl(S3C\_CLK\_SRC))&UPLL\_SRC\_MASK)!=UPLL\_SRC)

378                 \_\_raw\_writel((tmp&UPLL\_SRC\_MASK)|UPLL\_SRC,S3C\_CLK\_SRC);

379         while(((tmp=\_\_raw\_readl(S3C\_CLK\_DIV1))&UPLL\_DIV1\_MASK)!=UPLL\_DIV1)

380                 \_\_raw\_writel((tmp&UPLL\_DIV1\_MASK)|UPLL\_DIV1,S3C\_CLK\_DIV1);

381         while(((tmp=\_\_raw\_readl(S3C\_SCLK\_GATE))&UPLL\_GATE\_MASK)!=UPLL\_GATE)

382                 \_\_raw\_writel((tmp&UPLL\_GATE\_MASK)|UPLL\_GATE,S3C\_SCLK\_GATE);

383 }

在probe中加入上面的函数修改USB host的时钟：

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

387                                   struct platform\_device \*dev)

388 {

389         struct usb\_hcd \*hcd = NULL;

390         int retval;

391

392 #if !defined(CONFIG\_ARCH\_2410)

393         usb\_host\_clk\_en();

394 #endif

395

396         set\_upll();

然后编译内核。

USB的不能识别的错误就解决了