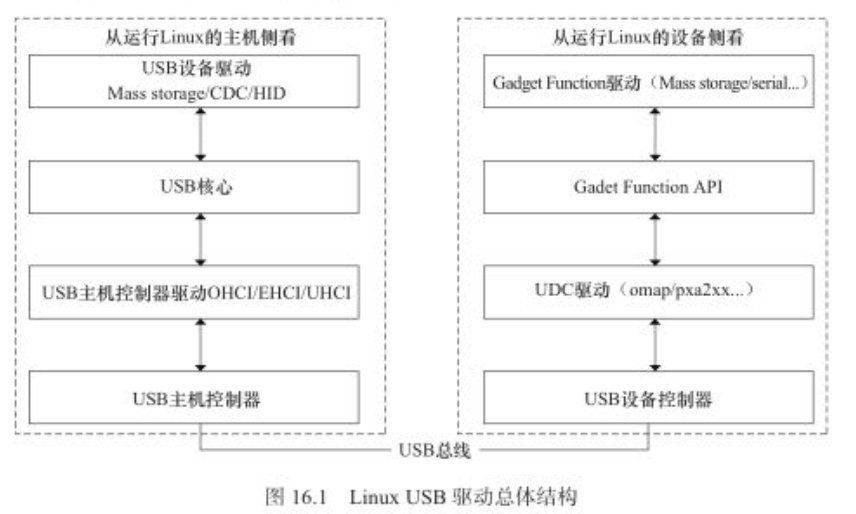
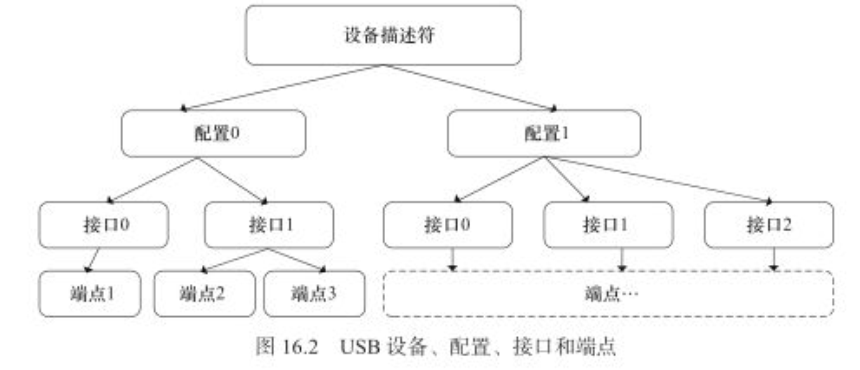
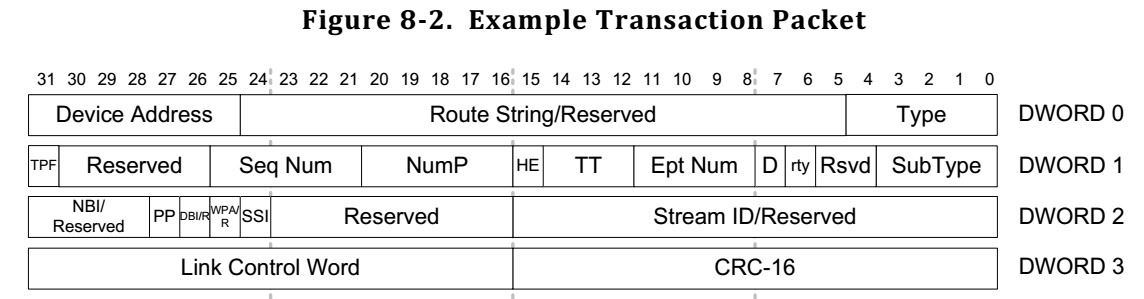
# 总结

|  |
| --- |
| **USB Core 模块：**  负责搭建整个USB框架: usb bus, hub driver, usb device driver  创建hub\_thread线程，处理hub上port的拔插事件(只针对于小机作为主控， 若小机作为usb 设备，则hub\_thread不会被wake\_up)  **Hub模块:**  负责hub上所有port的事件，当port由拔插事件，会产生INT中断传输，唤醒hub\_thread。  Hub\_thread会为port上的USB设备创建usb\_device设备并注册，唤醒generic\_probe  Generic\_probe会根据usb device的配置文件中的interface配置创建usb\_interface设备并注册，唤醒相应的driver，如uvcvideo驱动。  **Dwc3模块： usb3.0控制器**  做主控：  搭配xchi驱动  做设备：  搭配usb\_gadget驱动。dwc3\_gadget\_init()函数创建usb\_gadget驱动。 |
| **Dwc3 + gadget 框架**  **usb\_gadget\_driver和usb\_gadget**  **usb\_gadget**  : 由dwc3\_probe()创建，并挂载到udc\_list上  **usb\_gadget\_driver** : 由webcam\_init()创建  **match** : 查找udc\_list上挂载的usb\_gadget设备  **bind** : 由udc\_bind\_to\_driver()调用composite\_bind(), 创建**usb**\_**composite**\_**dev**，调用dwc3\_gadget\_start()  **usb\_composite\_driver和usb\_composite\_dev**  **usb\_composite\_driver:** 由webcam\_init()创建  **usb\_composite\_dev**  : 由usb\_gadget\_driver的composite\_bind()创建  **bind** : 由usb\_gadget\_driver的composite\_bind()调用: webcam\_bind()创建**uvc**\_**device**和**video**\_**device**，配置**usb**\_**configuration**  **wc3\_gadget\_start()**: 创建dwc3\_interrupt(),和dwc3\_thread\_interrupt()中断处理函数  **dwc3\_gadget\_ep\_queue()**： 将usb\_request发送到dwc3的endpoint上，启动endpoint发送数据给host端。  **中断处理函数：**  **dwc3\_interrupt():** 查询dwc3所有的event buffer, 若有event，则唤醒dwc3\_thread\_interrupt  **dwc3\_thread\_interrupt():** 处理端口数据事件和设备连接配置事件 |
| **Dwc3 + host 框架**  **platform\_device和platform\_driver: xhci-hcd**  **platform\_driver :** 由xhci\_hcd\_init()创建  **platform\_device** : 由dwc3\_host\_init()创建  **match**  : name = "xhci-hcd"  **probe**  : xhci\_plat\_probe()创建**usb**\_**device**  |--usb\_create\_hcd() : 创建usb\_hcd，其总线类型为xchi  |--usb\_add\_hcd() : 创建xchi下的第一个usb\_device：root\_hub  |-- usb\_hcd\_irq() ： xhci总线上的中断处理函数  |--rh\_timer\_func() : 创建hcd的定时器中断  **usb\_device和usb\_device\_driver: roothub**  **usb\_device\_driver :** 由usb\_init()创建  **usb\_device**  : 由xhci\_plat\_probe()创建USB2.0的第一个usb\_device和USB3.0的第一个usb\_device  **第一个usb\_device的devnum=1， portnum=0**  **match**  : type == usb\_device\_type  **probe**  : generic\_probe()根据usb\_device的配置文件创建**usb**\_**interface** 和 ep\_**device**  **usb\_driver和usb\_interface : roothub**  **usb\_driver** : 由usb\_init()注册hub\_driver  **usb\_interface :** 由generic\_probe()根据usb\_device的配置文件创建  **id\_table**  : USB\_CLASS\_HUB  **hub\_probe**  : 创建**usb**\_**hub**,**usb**\_**port**,，hub的中断传输线程**hub**\_**irq**()，  **hub中断事件处理**  **hub\_irq :** 将hub的中断urb添加到hub\_event\_list中  **hub\_thread() :** 处理port事件: 创建**usb\_device**和**ep**\_**device**，触发**generic**\_**probe**()  **usb\_device和usb\_device\_driver: usb camera**  **usb\_device\_driver** : 由usb\_init()创建  **usb\_device**  : 由hub\_thread()创建hub的port上的usb\_device  **match**  : type == usb\_device\_type  **probe**  : generic\_probe()根据usb\_device的配置文件创建**usb**\_**interface** 和 ep\_**device**  **usb\_driver和usb\_interface : usb camera**  **usb\_driver**  : 由uvc\_init()注册uvc\_driver  **usb\_interface** : 由generic\_probe()根据usb\_device的配置文件创建  **id\_table**  : USB\_CLASS\_VIDEO  **uvc\_probe** : 创建**uvc\_device** |
| **描述符:**  struct usb\_device  |--int devnum;  |--struct usb\_device\_descriptor descriptor; //设备描述符  |--struct usb\_host\_config \*config;  |--struct usb\_config\_descriptor desc; //配置描述符  |--struct usb\_interface \*interface[USB\_MAXINTERFACES];  |--struct usb\_host\_interface \*altsetting;  |--struct usb\_interface\_descriptor desc; //接口描述符  |--struct usb\_host\_endpoint \*endpoint; //端点描述符  |--unsigned char \*extra; //扩展描述符 |
| **主机控制器:**  struct usb\_hcd  |--unsigned long hcd\_priv[0]; //xhci\_hcd  |--const struct hc\_driver \*driver;  |--irqreturn\_t (\*irq) (struct usb\_hcd \*hcd);  |--int (\*start) (struct usb\_hcd \*hcd);  |--int (\*urb\_enqueue)(struct usb\_hcd \*hcd,struct urb \*urb, gfp\_t mem\_flags);  usb\_create\_hcd() : 创建HCD  usb\_add\_hcd() : 增加HCD  usb\_remove\_hcd() : 移除HCD  struct xhci\_hcd  |--struct usb\_hcd \*main\_hcd; //usb\_hcd    xhci\_plat\_setup() : 初始化XHCI主机控制器  xhci\_run() : 启动XHCI  xhci\_stop() : 停止XHCI  xhci\_reset() : 复位XHCI  hcd\_to\_xhci() : 根据USB HCD得到XHCI HCD  xhci\_to\_hcd() : 根据XHCI HCD得到USB HCD  cat /sys/kernel/debug/usb/devices ： 查看USB设备信息 |
| **USB驱动**  struct usb\_driver  |--int (\*probe) (struct usb\_interface \*intf,const struct usb\_device\_id \*id);  |--void (\*disconnect) (struct usb\_interface \*intf);  |--const struct usb\_device\_id \*id\_table;  usb\_register() : USB驱动的注册  usb\_deregister() : USB驱动的注销  probe() : USB设备的插入  disconnect() : USB设备的拔出  id\_table : 描述USB驱动支持的USB设备列表  |--USB\_DEVICE()  |--USB\_DEVICE\_VER()  |--USB\_DEVICE\_INFO()  |--USB\_INTERFACE\_INFO() |
| **UVC的URB:**  struct urb  usb\_alloc\_urb() : 创建URB  usb\_free\_urb() : 销毁URB  usb\_fill\_int\_urb(complete(), interval) : 创建int传输URB  usb\_fill\_bulk\_urb(complete()) : 创建bulk传输URB  usb\_fill\_control\_urb(complete(),setup\_packet) : 创建control传输URB  usb\_submit\_urb() : 提交URB给USB主机控制器  usb\_bulk\_msg(data) : 创建bulk传输msg  usb\_control\_msg(requset, requesttype, value, indexx, data) : 创建control传输msg |
| **设备控制器**  struct usb\_gadget  usb\_add\_gadget\_udc() : 添加设备控制器 |
| **UDC的REQ:**  struct usb\_request  usb\_ep\_enable() : 使能ep  usb\_ep\_disable() : 禁止ep  alloc\_ep\_req() : usb\_ep\_alloc\_request()  usb\_ep\_alloc\_request() : 分配依附于ep的req  usb\_ep\_free\_request() : 释放依附于ep的req  usb\_ep\_queue() : 提交req  usb\_ep\_dequeue() : 取消req  usb\_ep\_fifo\_status() : 返回FIFO中的字节数  usb\_ep\_fifo\_flush() : 冲刷掉FIFO中的数据 |



# Trace

usb\_gadget\_udc\_start()

|  |
| --- |
| [ 5.163874] [<ffffffc0000881f0>] dump\_backtrace+0x0/0x128  [ 5.169281] [<ffffffc000088328>] show\_stack+0x10/0x1c  [ 5.174343] [<ffffffc000734ebc>] dump\_stack+0x1c/0x28  [ 5.179406] [<ffffffc0000976d4>] warn\_slowpath\_common+0x74/0xa0  [ 5.185343] [<ffffffc0000977c0>] warn\_slowpath\_null+0x14/0x20  [ 5.191093] [<ffffffc000448928>] udc\_bind\_to\_driver+0xa8/0xec  [ 5.196843] [<ffffffc0004489d8>] usb\_gadget\_probe\_driver+0x6c/0xb4  [ 5.203031] [<ffffffbffc03f730>] usb\_composite\_probe+0xb8/0xdc [libcomposite]  [ 5.210156] [<ffffffbffc06d160>] init\_module+0x160/0x1ec [g\_android]  [ 5.216531] [<ffffffc0000814d8>] do\_one\_initcall+0xe0/0x14c  [ 5.222124] [<ffffffc0000f5278>] load\_module+0x1570/0x1e6c  [ 5.227624] [<ffffffc0000f5c30>] SyS\_init\_module+0xbc/0xf0 |

dwc3\_gadget\_stop

|  |
| --- |
| [ 142.007624] [<ffffffc0000881f0>] dump\_backtrace+0x0/0x128  [ 142.013030] [<ffffffc000088328>] show\_stack+0x10/0x1c  [ 142.018093] [<ffffffc0007350b8>] dump\_stack+0x1c/0x28  [ 142.023155] [<ffffffc0000976d4>] warn\_slowpath\_common+0x74/0xa0  [ 142.029093] [<ffffffc0000977c0>] warn\_slowpath\_null+0x14/0x20  [ 142.034843] [<ffffffc000426dc0>] dwc3\_gadget\_stop+0x30/0xe0  [ 142.040405] [<ffffffc000735c08>] dwc3\_gadget\_plugout.isra.6+0x6c/0x84  [ 142.046843] [<ffffffc0004236a8>] dwc3\_set\_plugstate+0x168/0x360  [ 142.052780] [<ffffffc00042ceec>] s\_dwc3\_set\_plugstate+0x28/0x3c  [ 142.058718] [<ffffffc00042ba90>] monitor\_handle\_plug\_in\_out\_msg+0x108/0x12c  [ 142.065686] [<ffffffc00042c688>] mon\_config\_store+0x380/0x3d0  [ 142.071436] [<ffffffc0002fbdcc>] kobj\_attr\_store+0x10/0x24  [ 142.076936] [<ffffffc0001e0ae4>] sysfs\_write\_file+0xd4/0x14c  [ 142.082593] [<ffffffc00017c358>] vfs\_write+0xa4/0x18c  [ 142.087655] [<ffffffc00017cb90>] SyS\_write+0x40/0x8c |

set\_connect\_type\_to\_pc\_mode()

|  |
| --- |
| [ 65.201406] [<ffffffc0000881f0>] dump\_backtrace+0x0/0x128  [ 65.206812] [<ffffffc000088328>] show\_stack+0x10/0x1c  [ 65.211874] [<ffffffc000734e70>] dump\_stack+0x1c/0x28  [ 65.216937] [<ffffffc0000976d4>] warn\_slowpath\_common+0x74/0xa0  [ 65.222874] [<ffffffc0000977c0>] warn\_slowpath\_null+0x14/0x20  [ 65.228624] [<ffffffc000426a80>] dwc3\_thread\_interrupt+0x654/0x6a4  [ 65.234812] [<ffffffc00010742c>] irq\_thread+0x104/0x158  [ 65.240031] [<ffffffc0000bba34>] kthread+0xb0/0xbc |

mon\_config\_store()

|  |
| --- |
| [ 65.018187] [<ffffffc0000881f0>] dump\_backtrace+0x0/0x128  [ 65.023593] [<ffffffc000088328>] show\_stack+0x10/0x1c  [ 65.028656] [<ffffffc000734e70>] dump\_stack+0x1c/0x28  [ 65.033718] [<ffffffc0000976d4>] warn\_slowpath\_common+0x74/0xa0  [ 65.039624] [<ffffffc0000977c0>] warn\_slowpath\_null+0x14/0x20  [ 65.045374] [<ffffffc00042c4b4>] mon\_config\_store+0x390/0x3d0  [ 65.051124] [<ffffffc0002fbdcc>] kobj\_attr\_store+0x10/0x24  [ 65.056624] [<ffffffc0001e0ae4>] sysfs\_write\_file+0xd4/0x14c  [ 65.062281] [<ffffffc00017c358>] vfs\_write+0xa4/0x18c  [ 65.067343] [<ffffffc00017cb90>] SyS\_write+0x40/0x8c |

# Usb\_core：注册usb总线，hub驱动，usb device驱动

|  |
| --- |
| 1. 创建usb\_bus总线，总线上的设备都是通过usb协议传输的。 2. 注册usb hub驱动和usb device驱动，当有usb\_hub设备或usb\_device设备插入时，触发相应的驱动。 3. 主控芯片内的usb控制器(XHCI)将作为root hub挂载到总线，相应的hub驱动会执行probe()。 |

## usb\_init()

1. 注册debugfs: Debugfs是linux内核的一种debug方式，debugfs最终会在系统内创建/sys/kernel/debug/usb，方便调试
2. 注册usb bus: /sys/bus/usb
3. 注册usb字符设备 ： file\_operations = usb\_fops

root@AD700A:/ # cat /proc/devices | grep usb

180 usb

1. 注册usbfs: usb\_driver usbfs\_driver /sys/bus/usb/drivers/usbfs
2. 注册usb\_device字符设备: file\_operations usbdev\_file\_operations

root@AD700A:/ # cat /proc/devices | grep usb

189 usb\_device

1. 注册hub\_driver : /sys/bus/usb/drivers/hub //usb\_interface\_driver
2. 注册usb\_device\_driver: /sys/bus/usb/drivers/usb //usb\_device\_driver

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **usb\_init**()  |--usb\_**debugfs**\_init(); //创建debugfs : /sys/kernel/debug/usb  1. 创建USB总线  |--**bus**\_**register**(&**usb**\_**bus**\_**type**);   |  | | --- | | //bus\_type: Driver.c  struct **bus**\_**type** **usb**\_**bus**\_**type** = {  .name = "usb",  .match = usb\_device\_match,  .uevent = usb\_uevent,  }; |   |--bus\_register\_notifier(&usb\_bus\_type, &usb\_bus\_nb);  |--usb\_major\_init();  |--**register**\_**chrdev**(USB\_MAJOR/\*180\*/, "usb", &**usb**\_**fops**); //注册usb:180字符设备   |  | | --- | | static const struct **file**\_**operations** **usb**\_**fops** = {  .owner = THIS\_MODULE,  .open = usb\_open,  .llseek = noop\_llseek,  }; |     |--**usb**\_**register**(&**usbfs**\_**driver**); //usb\_driver //注册usb\_driver : usbfs 到usb\_bus上  |--usb\_register\_driver(driver, THIS\_MODULE, KBUILD\_MODNAME)  |--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; //usbfs\_driver  |--new\_driver->drvwrap.driver.remove = usb\_unbind\_interface;  |--**driver**\_**register**(&new\_driver->drvwrap.driver); //device\_driver   |  | | --- | | //usb\_driver : Devio.c  struct **usb**\_**driver** **usbfs**\_**driver** = {  .name = "usbfs",  .probe = driver\_probe, //do nothing  .disconnect = driver\_disconnect,  .suspend = driver\_suspend,  .resume = driver\_resume,  }; |   |--usb\_devio\_init();  |--**cdev**\_**init**(&**usb**\_**device**\_**cdev**, &**usbdev**\_**file**\_**operations**);  |--**cdev**\_**add**(&usb\_device\_cdev, USB\_DEVICE\_DEV, USB\_DEVICE\_MAX); //注册usb\_device:189字符设备  |--usb\_register\_notify(&usbdev\_nb);   |  | | --- | | static struct **cdev** **usb**\_**device**\_**cdev**; | | const struct **file**\_**operations** **usbdev**\_**file**\_**operations** = {  .owner = THIS\_MODULE,  .llseek = usbdev\_lseek,  .read = usbdev\_read,  .poll = usbdev\_poll,  .unlocked\_ioctl = usbdev\_ioctl,  #ifdef CONFIG\_COMPAT  .compat\_ioctl = usbdev\_compat\_ioctl,  #endif  .open = usbdev\_open,  .release = usbdev\_release,  }; |   2. 注册Hub驱动到USB总线  |--**usb**\_**hub**\_**init**();  |--**usb**\_**register**(&**hub**\_**driver**)  |--usb\_register\_driver(driver, THIS\_MODULE, KBUILD\_MODNAME)  |--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**; //hub\_driver  |--new\_driver->drvwrap.driver.remove = usb\_unbind\_interface;  |--**driver**\_**register**(&new\_driver->drvwrap.driver); //device\_driver   |  | | --- | | static struct **usb**\_**driver** **hub**\_**driver** = {  .name = "hub",  .**probe** = **hub**\_**probe**, //插入hub设备时会调用  .disconnect = hub\_disconnect,  .suspend = hub\_suspend,  .resume = hub\_resume,  .reset\_resume = hub\_reset\_resume,  .pre\_reset = hub\_pre\_reset,  .post\_reset = hub\_post\_reset,  .unlocked\_ioctl = hub\_ioctl,  .id\_table = **hub**\_**id**\_**table**,  .supports\_autosuspend = 1,  }; | | static const struct **usb**\_**device**\_**id** **hub**\_**id**\_**table**[] = {  {  .match\_flags = USB\_DEVICE\_ID\_MATCH\_**VENDOR** | USB\_DEVICE\_ID\_MATCH\_INT\_CLASS,  .**idVendor** = USB\_VENDOR\_GENESYS\_LOGIC,  .bInterfaceClass = USB\_CLASS\_HUB,  .driver\_info = HUB\_QUIRK\_CHECK\_PORT\_AUTOSUSPEND  },  {  .match\_flags = USB\_DEVICE\_ID\_MATCH\_**DEV**\_**CLASS**,  .**bDeviceClass** = USB\_CLASS\_HUB  },  {  .match\_flags = USB\_DEVICE\_ID\_MATCH\_**INT**\_**CLASS**,  .**bInterfaceClass** = USB\_CLASS\_HUB  },  { } /\* Terminating entry \*/  }; |   3. 注册通用设备驱动到USB总线  |--**usb**\_**register**\_**device**\_**driver**(&**usb**\_**generic**\_**driver**, THIS\_MODULE); //usb\_device\_driver : device 到usb\_bus上  |--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**; //usb\_device\_driver  |--new\_udriver->drvwrap.driver.remove = usb\_unbind\_device;  |--driver\_register(&new\_udriver->drvwrap.driver); //device\_driver   |  | | --- | | struct **usb**\_**device**\_**driver** **usb**\_**generic**\_**driver** = {  .name = "usb",  .**probe** = **generic**\_**probe**, //插入device设备时会调用  .disconnect = generic\_disconnect,  .suspend = generic\_suspend,  .resume = generic\_resume,  .supports\_autosuspend = 1,  }; |   4. 创建USB线程，处理Hub的khubd\_wait事件  |--khubd\_task = **kthread**\_**run**(**hub**\_**thread**, NULL, "khubd"); |

## Usb\_bus 总线

### usb\_device\_match()

|  |
| --- |
| **usb\_device\_match** (struct **device** \*dev, struct **device**\_**driver** \*drv)  |--is\_**usb**\_**device**(dev) && is\_**usb**\_**device**\_**driver**(drv) ==> return 1;  |--is\_**usb**\_**interface**(dev) && !is\_usb\_device\_driver(drv)  |-- struct **usb**\_**driver** \*usb\_drv = to\_usb\_driver(drv);  |-- struct **usb**\_**interface** \*intf = to\_usb\_interface(dev);  |--usb\_match\_id(intf, usb\_drv->**id\_table**); return 1  |--usb\_match\_dynamic\_id(intf, **usb**\_**drv**); return 1 |

### is\_usb\_device()

|  |
| --- |
| **is\_usb\_device**(const struct **device** \*dev)  {  return dev->**type** == &**usb\_device\_type**; //usb\_alloc\_dev()  } |

### usb\_device\_type

|  |
| --- |
| 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  }; |

### is\_usb\_device\_driver()

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

### is\_usb\_interface

|  |
| --- |
| **is\_usb\_interface**(const struct **device** \*dev)  {  return dev->type == &**usb**\_**if\_device\_type;**  } |

### usb\_if\_device\_type

|  |
| --- |
| struct **device**\_**type** **usb\_if\_device\_type** = {  .name = "usb\_interface",  .release = usb\_release\_interface,  .uevent = usb\_if\_uevent,  }; |

## Hub驱动

|  |
| --- |
| 1. xchi主控包含root hub，当主控驱动挂载时会触发hub\_probe()。在probe中创建中断传输接口，并为hub的所有port创建usb\_port\_device\_type类型的设备。 2. 当USB设备连接到hub的port时，port状态发生变化，会产生INT传输，唤醒hub\_thread()线程 3. Hub\_thread会查询hub上的所有port的状态，当有设备链接的时候，会创建usb\_device设备并注册，唤醒usb device的generic\_probe()。 4. generic\_probe会根据usb\_device的配置文件中的接口配置创建并注册usb\_interface设备到usb bus上，usb bus会根据系统上的挂载的usb driver的id\_table寻找匹配的接口驱动。， |

### hub\_probe() : 创建usb\_hub, 及hub上的usb\_port设备

1. 创建usb\_hub设备

2. 获取hub的描述符，得到hub支持的port个数 : hub->descriptor->bNbrPorts

3. 为hub上的每一个port创建一个usb\_port设备并注册

4. 更新hub设备: hcd->driver->update\_hub\_device

|  |
| --- |
| **Hub\_Probe**(struct **usb**\_**interface** \*intf, const struct usb\_device\_id \*id)  1. 创建hub  |--struct **usb**\_**hub** \*hub = **kzalloc**(sizeof(\*hub), GFP\_KERNEL);  2. 配置hub  |--struct **usb**\_**host**\_**interface** \*desc = **intf**->**cur**\_**altsetting**;  |--struct **usb**\_**endpoint**\_**descriptor** \*endpoint = &desc->endpoint[0].desc;  |--**hub\_configure(hub, endpoint);** |
| **hub\_configure**(hub, endpoint)  1. 建立hub的中断传输通道  |--hub->urb = **usb**\_**alloc**\_**urb**(0, GFP\_KERNEL);  |--struct **usb**\_**device** \*hdev = hub->hdev;  |--pipe = usb\_rcvintpipe(hdev, endpoint->bEndpointAddress);  |--hub->buffer = kmalloc(sizeof(\*hub->buffer), GFP\_KERNEL);  |--maxp = usb\_maxpacket(hdev, pipe, usb\_pipeout(pipe));  |--**usb**\_**fill**\_**int**\_**urb**(hub->urb, hdev, pipe, \*hub->buffer, maxp, **hub**\_**irq**, hub, endpoint->bInterval);  2. 创建Hub的所有Port设备  |--**maxchild** = hub->descriptor->**bNbrPorts**;  |--for (i = 0; i < maxchild; i++)  |--ret = **usb\_hub\_create\_port\_device(hub, i + 1);**  |--hdev->maxchild = i; |
| **usb\_hub\_create\_port\_device**(struct usb\_hub \*hub, int port1)  1. 创建并注册port设备  |--struct **usb**\_**port** \*port\_dev = **kzalloc**(sizeof(\*port\_dev), GFP\_KERNEL);  |--port\_dev->portnum = i + 1;  |--port\_dev->dev.**type** = &usb\_**port**\_**device**\_**type**;  |--dev\_set\_name(&port\_dev->dev, "port%d", port1);  |--**device**\_**register**(&port\_dev->dev); |

### hub\_irq() ： hub的中断传输的回调函数

1. 将hub接收到的中断传输的urb事件添加到hub\_event\_list中

2. 触发中断传输事件 : wake\_up(&khubd\_wait);

|  |
| --- |
| **hub\_irq** (struct urb \*urb)  1. 处理HUB的INT传输URB  |--struct **usb**\_**hub** \*hub = urb->context;  |--kick\_khubd(hub);  |--list\_add\_tail(&hub->event\_list, &**hub**\_**event**\_**list**);  |--**wake**\_**up**(&**khubd**\_**wait**); //触发USB\_CORE创建的线程: Hub\_thread  2. 释放URB  |--**usb**\_**submit**\_**urb** (hub->urb, GFP\_ATOMIC); |

### hub\_thread() : 查看hub\_event\_list

1. 等待中断传输事件触发 : wait\_event\_freezable(khubd\_wait)

2. 查询hub的所有port上的status， 获取port状态: usb\_control\_msg(USB\_REQ\_GET\_STATUS)

3. 若端口有插拔事件：hub\_port\_connect\_change()

3.1 创建usb\_device设备，类型为usb\_device\_type: usb\_alloc\_dev(port1)

3.2 设置dev的状态为USB\_STATE\_POWERED

3.3 设置devnum ： hub\_set\_address()

3.4 获取设备描述符，配置描述符

3.5 注册usb\_device设备

3.6 创建ep\_device设备，类型为usb\_ep\_device\_type

3.7 注册ep\_device设备

|  |
| --- |
| **hub\_thread**()  1. 处理hub的事件  |--**wait\_event\_freezable**(khubd\_wait);  |--**hub**\_**events**(); |
| **hub\_events**();  1. 获取hub  |--tmp = **hub**\_**event**\_**list**.next;  |--hub = list\_entry(tmp, struct usb\_hub, event\_list);  2. 查询hub的所有端口状态  |--for (i = 1; i <= hub->descriptor->**bNbrPorts**; i++)  2.1 查询hub的端口状态  |--hub\_port\_status(hub, i, &portstatus, &portchange); //获取port1的status  |--get\_port\_status(hub->hdev, port1, &hub->status->port);  |--**usb**\_**control**\_**msg**(USB\_REQ\_GET\_STATUS); //发送control命令  |--\***status** = le16\_to\_cpu(hub->status->port.wPortStatus);  |--\***change** = le16\_to\_cpu(hub->status->port.wPortChange);  2.2 处理hub端口的事件 : hub的port存在插拔事件  |--**hub**\_**port**\_**connect**\_**change**(hub, i, portstatus, portchange); //usb\_hub + port1 |
| **hub\_port\_connect\_change**(struct usb\_hub \*hub, int port1, u16 portstatus, u16 portchange)  |--hub\_port\_debounce\_be\_stable(hub, port1); //去抖动  1. 创建usb\_device  |--struct **usb**\_**device** \*udev = usb\_alloc\_dev(hdev, hdev->bus, port1);  |--struct **usb**\_**device** \*dev = **kzalloc**(sizeof(\*dev), GFP\_KERNEL);  |--dev->dev.bus = &usb\_bus\_type;  |--dev->dev.type = &usb\_**device**\_type;  |--dev->state = USB\_STATE\_ATTACHED; //设置dev的状态  |--dev->portnum = port1;  2. 设置usb\_device的状态为POWERED  |--usb\_set\_device\_state(udev, USB\_STATE\_POWERED);  3.分配usb\_device的devnum: 一条usb总线下总共可以有128个设备，usb总线通过位图的形式来管理usb设备的地址，choose\_devnum是从usb 总线中找到一个usb地址，usb地址在1和128之间；  |--**choose**\_**devnum**(udev); //设置devnum  4. 设置port： 通过hub\_port\_reset来reset设备，reset机制为：通过set\_port\_feature来传输USB\_PORT\_FEAT\_RESET指令，设置成功后循环延时由之前确定的时间间隔后去读取port的status和change状态，要确保usb设备在reset后还能正常存在，如reset还能正常，则通过port的status状态位来确定usb设备的速度，循环延时总时间为500ms,而usb设备reset次数为5次，  |--hub\_port\_init(hub, udev, port1, i);  |--hub\_port\_reset(hub, port1, udev, delay, false);  |--**usb**\_**control**\_**msg**(USB\_REQ\_GET\_**DESCRIPTOR**)  |--hub\_port\_reset(hub, port1, udev, delay, false);  4.1 设置USB设备地址为devnum  |--**hub**\_**set**\_**address**(udev, devnum); //设置地址  |--usb\_get\_device\_descriptor(udev, 8);  4.2 获取USB设备描述符  |--**usb**\_**get**\_**device**\_**descriptor**(udev, USB\_DT\_DEVICE\_SIZE /\*18\*/);  |--hcd->driver->update\_device(hcd, udev); //xhci-plat.c  |--hub->ports[port1 - 1]->child = udev;  |--usb\_new\_device(udev);  |--usb\_enumerate\_device(udev);  4.3 获取USB配置描述符并解析  |--usb\_get\_configuration(udev);  |--**usb**\_**get**\_**descriptor**(USB\_DT\_**CONFIG**);  |--usb\_parse\_configuration()  |--udev->dev.devt = MKDEV(USB\_DEVICE\_MAJOR, (((udev->bus->busnum-1) \* 128) + (udev->devnum-1)));  4.4 注册USB设备  |--**device**\_**add**(&udev->dev); //触发probe : usb\_device\_type  4.5 创建ep\_device并注册  |--usb\_create\_ep\_devs(&udev->dev, &udev->ep0, udev);  |--struct **ep**\_**device** \*ep\_dev = kzalloc(sizeof(\*ep\_dev), GFP\_KERNEL);  |--ep\_dev->dev.type = &usb\_ep\_device\_type;  |--**device**\_**register**(&ep\_dev->dev); //触发probe ： usb\_ep\_device\_type  |--endpoint->ep\_dev = ep\_dev; |

## USB device驱动

|  |
| --- |
| generic\_probe会根据usb\_device的配置文件中的接口配置创建并注册usb\_interface设备到usb bus上，usb bus会根据系统上的挂载的usb driver的id\_table寻找匹配的接口驱动。 |

### generic\_probe()

1. 根据usb\_device选择一个configuration，默认一般选第一个。
2. 每个config中含有多个interface，为每一个interface创建usb\_interface设备并注册， interface.type = &usb\_if\_device\_type
3. 每个interface中含有多个endpoint，为每一个endpoint创建ep\_device设备并注册，endpoint.type = &usb\_ep\_device\_type

|  |
| --- |
| **generic**\_**probe**(struct **usb**\_**device** \*udev)  1. 选取usb\_device的配置文件，默认选第一个配置文件  |--c = **usb**\_**choose**\_**configuration**(udev); //choose first configuration  2. 根据配置文件配置相关interface  |--**usb**\_**set**\_**configuration**(udev, c); //usb\_device + configuration  3. 发出USB\_DEVICE\_ADD消息  |--usb\_**notify**\_**add**\_**device**(udev);  |--blocking\_notifier\_call\_chain(&usb\_notifier\_list, **USB**\_**DEVICE**\_**ADD**, udev); |
| int **usb**\_**set**\_**configuration**(struct usb\_device \*dev, int configuration)  1. 获取配置文件中支持的interface的个数  |--struct **usb**\_**host**\_**config** \*cp = &dev->config[i]  |--nintf = cp->desc.**bNumInterfaces**;  |--struct **usb**\_**interface** \*\*new\_interfaces = **kmalloc**(nintf \* sizeof(\*new\_interfaces), GFP\_NOIO);  2. 为每一个interface创建一个设备，并绑定对应驱动  |--for (; n < nintf; ++n) {  |--new\_interfaces[n] = **kzalloc**(sizeof(struct usb\_interface),GFP\_NOIO);  2.1 创建并配置usb\_interface  |--for (i = 0; i < nintf; ++i)  |--cp->interface[i] = intf = new\_interfaces[i];  |--intf->dev.bus = &usb\_bus\_type;  |--intf->dev.**type** = &**usb**\_**if**\_**device**\_**type**;  |--dev\_set\_name(&intf->dev, "%d-%s:%d.%d",dev->bus->busnum, dev->devpath,configuration, alt->desc.bInterfaceNumber);  2.2 注册usb\_interface  |--for (i = 0; i < nintf; ++i)  |--struct **usb**\_**interface** \*intf = cp->interface[i];  |--**device**\_**add**(&intf->dev);  2.3 创建interface中的endpoint\_device并注册  |--create\_intf\_ep\_devs(intf); //创建ep\_device  |--struct **usb**\_**host**\_**interface** \*alt = intf->cur\_altsetting;  |--for (i = 0; i < alt->desc.bNumEndpoints; ++i) //每个interface中含有多个endpoint  |--usb\_create\_ep\_devs(&intf->dev, &alt->endpoint[i], udev);  |--struct **ep**\_**device** \*ep\_dev = kzalloc(sizeof(\*ep\_dev), GFP\_KERNEL);  |--ep\_dev->dev.**type** = &**usb**\_**ep**\_**device**\_**type**;  |--**device**\_**register**(&ep\_dev->dev); //注册ep\_device， 触发ep驱动的probe()  |--endpoint->ep\_dev = ep\_dev;  3. 告诉USB设备，主控使用的是configuration对应的配置文件  |--**usb**\_**control**\_**msg**(dev, usb\_sndctrlpipe(dev, 0),USB\_REQ\_SET\_CONFIGURATION, 0, configuration, 0,NULL, 0, USB\_CTRL\_SET\_TIMEOUT);  |--**usb**\_**set**\_**device**\_**state**(dev, USB\_STATE\_CONFIGURED); |

### Usb\_device的创建过程: hub\_thread()

1. Hub上有中断，调用hub\_thread()
2. 查询hub上的所有port，看哪个port上有状态变化。
3. Debounce，等待usb port去抖动
4. 创建usb\_device设备，设定dev.type = &usb\_device\_type;
5. 设定usb\_device的状态: USB\_STATE\_POWERED
6. 给usb\_device设置一个有效的地址。
7. Hub的port执行reset
8. 获取usb\_device的描述符，得到bMaxPacketSize0
9. 设置地址，获取设备描述符，获取配置描述符，解析配置描述符
10. 注册usb\_device，触发usb\_device驱动的probe()
11. 创建usb\_device的ep\_device设备，ep0.type = &usb\_ep\_device\_type.
12. 注册ep\_device设备

### Usb interface的创建过程：generic\_probe()

1. 根据usb\_device选择一个configuration，默认一般选第一个。
2. 每个config中含有多个interface，为每一个interface创建usb\_interface设备并注册， interface.type = &usb\_if\_device\_type
3. 每个interface中含有多个endpoint，为每一个endpoint创建ep\_device设备并注册，endpoint.type = &usb\_ep\_device\_type

# DWC3：

|  |
| --- |
| gadget+dwc3构成usb device框架，如usb camera设备  host+dwc3构成usb host框架，如PC模式 |

## DTS: dwc3@e0400000

|  |
| --- |
| **owl\_dwc3@e040ce04** {  compatible = "actions,s700-dwc3";  reg = <0 0xe040ce04 0 0xc>;  ranges;  power-domains = <&powergate POWER\_DOMAIN\_USB3>;  clocks = <&clock CLK\_USB3\_480MPLL0>, <&clock CLK\_USB3\_480MPHY0>, <&clock CLK\_USB3\_5GPHY>, <&clock CLK\_USB3\_CCE>, <&clock CLK\_USB3\_MAC>;  clock-names = "usb3\_480mpll0", "usb3\_480mphy0", "usb3\_5gphy", "usb3\_cce", "usb3\_mac";  resets = <&reset RESET\_USB3>;  reset-names = "usb3";    dwc3@e0400000 {  **compatible = "synopsys,dwc3";**  reg = <0 0xe0400000 0 0xcd00>;  interrupts = <0 23 4>;  usb-phy = <&usb2\_phy>, <&usb3\_phy>;  };  }; |

## module\_platform\_driver()

|  |  |  |
| --- | --- | --- |
| **module\_platform\_driver**(**dwc3**\_**driver**);   |  | | --- | | static struct **platform**\_**driver** **dwc3**\_**driver** = {  .**probe** = **dwc3**\_**probe**,  .remove = dwc3\_remove,  .driver = {  .name = "dwc3",  .**of**\_**match**\_**table** = of\_match\_ptr(**of**\_**dwc3**\_**match**),  .pm = DWC3\_PM\_OPS,  },  }; | | static const struct of\_device\_id **of**\_**dwc3**\_**match**[] = {  {  .compatible = "**synopsys,dwc3"**  },  { },  }; | |

## dwc3\_probe(): dwc3作为device模式初始化

|  |
| --- |
| **dwc3\_probe**(struct platform\_device \*pdev)  1. 创建dwc3结构体并配置  |--struct **dwc3** \*dwc = **devm**\_**kzalloc**(dev, sizeof(\*dwc) + DWC3\_ALIGN\_MASK, GFP\_KERNEL);  |--dwc->maximum\_speed = DWC3\_DCFG\_SUPERSPEED;  2. 获取dwc3的硬件信息: dwc->hwparams  |--**dwc3**\_**cache**\_**hwparams**(dwc);  3. 配置event buffer：由dwc->hwparams决定  #define DWC3\_EVENT\_SIZE 4 /\* bytes \*/  #define DWC3\_EVENT\_MAX\_NUM 64 /\* 2 events/endpoint \*/  #define DWC3\_EVENT\_BUFFERS\_SIZE (DWC3\_EVENT\_SIZE \* DWC3\_EVENT\_MAX\_NUM)  |--**dwc3**\_**alloc**\_**event**\_**buffers**(dwc, DWC3\_EVENT\_BUFFERS\_SIZE);  |--num = DWC3\_NUM\_INT(dwc->hwparams.hwparams1);  |--dwc->num\_event\_buffers = num; //hwparams1: 获取dwc3支持的event\_buffer总数  |--dwc->ev\_buffs = devm\_kzalloc(dwc->dev, sizeof(\*dwc->ev\_buffs) \* num, GFP\_KERNEL);  4. 配置dwc3的in/out端点  |--**dwc3**\_**core**\_**init**(dwc);  |--dwc3\_core\_num\_eps(dwc); //获取dwc3的in/out端点数  |--dwc->num\_in\_eps = DWC3\_NUM\_IN\_EPS(parms);  |--dwc->num\_out\_eps = DWC3\_NUM\_EPS(parms) - dwc->num\_in\_eps;  5. 配置dwc3的模式为DEVICE, 并创建gadget驱动  |--mode = DWC3\_MODE\_DEVICE; //dwc3作为usb device设备进行环境配置  |--**dwc3**\_**set**\_**mode**(dwc, DWC3\_GCTL\_PRTCAP\_DEVICE);  |--**dwc3**\_**gadget**\_**init**(dwc) |

# Xchi Host Controller驱动 :

|  |
| --- |
| usb\_create\_hcd() : 创建usb\_hcd，若为USB2.0, has\_tt=1； 若为USB3.0，has\_tt=0  usb\_hcd包含:  |--usb\_bus： xchi  |--root\_hub：第一个usb\_device  |--speed : USB3.0  |--hc\_driver: xhci\_plat\_xhci\_driver  usb\_add\_hcd() : 创建usb\_bus下的第一个usb\_device：root\_hub  usb\_device包含:  |--bus：xhci  |--portnum：0 |

## xhci\_hcd\_init()

“xhci-hcd”对应的platform\_device在dwc3文件夹中创建

dwc3\_host\_init() //dwc3作为platfprm设备

|--struct platform\_device \*xhci;

|--xhci = platform\_device\_alloc("xhci-hcd", PLATFORM\_DEVID\_AUTO);

|  |  |  |
| --- | --- | --- |
| **module\_init**(xhci\_hcd\_init);  |--xhci\_register\_plat()  |--**platform**\_**driver**\_**register**(&**usb**\_**xhci**\_**driver**);   |  | | --- | | static struct **platform**\_**driver** **usb**\_**xhci**\_**driver** = {  **.probe = xhci\_plat\_probe,**  .remove = xhci\_plat\_remove,  .driver = {  .name = "xhci-hcd",  .pm = DEV\_PM\_OPS,  .of\_match\_table = of\_match\_ptr(**usb\_xhci\_of\_match**),  },  }; | | static const struct of\_device\_id **usb**\_**xhci**\_**of**\_**match**[] = {  { .compatible = "generic-xhci" },  { .compatible = "xhci-platform" },  { },  }; | |

## xhci\_plat\_probe()

1. 一个xhci支持2个hub,如一个usb3.0的hub和一个usb2.0的hub

2. 创建hc\_driver : xhci\_plat\_xhci\_driver

3. 创建第一个usb\_hcd结构体：usb\_create\_shared\_hcd(NULL)

4. 创建第二个usb\_hcd结构体：usb\_create\_shared\_hcd(roothub)

5. 配置每个hub的hc\_driver: hcd->driver = driver; //hc\_driver

6. 配置每个hub的定时器: hcd->rh\_timer.function = rh\_timer\_func;

7. 将usb\_hcd添加到usb\_bus\_list列表中

8. 创建usb\_device并注册，建立usb\_hcd和usb\_device的关系: hcd->self.root\_hub = rhdev;

9. 创建usb\_device的ep\_device设备并注册

|  |  |
| --- | --- |
| **xhci\_plat\_probe**()   1. 创建xchi协议的host controller驱动   |--driver = &**xhci**\_**plat**\_**xhci**\_**driver**;   |  | | --- | | static const struct **hc**\_**driver** **xhci**\_**plat**\_**xhci**\_**driver** = {  .description = "xhci-hcd",  .product\_desc = "xHCI Host Controller",  .hcd\_priv\_size = sizeof(struct xhci\_hcd \*),  .irq = xhci\_irq,  .flags = HCD\_MEMORY | HCD\_USB3 | HCD\_SHARED,  .**reset** = xhci\_plat\_setup,  .**start** = xhci\_run,  .stop = xhci\_stop,  .shutdown = xhci\_shutdown,  .urb\_enqueue = xhci\_urb\_enqueue,  .urb\_dequeue = xhci\_urb\_dequeue,  .alloc\_dev = xhci\_alloc\_dev,  .free\_dev = xhci\_free\_dev,  .alloc\_streams = xhci\_alloc\_streams,  .free\_streams = xhci\_free\_streams,  .add\_endpoint = xhci\_add\_endpoint,  .drop\_endpoint = xhci\_drop\_endpoint,  .endpoint\_reset = xhci\_endpoint\_reset,  .check\_bandwidth = xhci\_check\_bandwidth,  .reset\_bandwidth = xhci\_reset\_bandwidth,  .address\_device = xhci\_address\_device,  .enable\_device = xhci\_enable\_device,  .update\_hub\_device = xhci\_update\_hub\_device,  .reset\_device = xhci\_discover\_or\_reset\_device,  .get\_frame\_number = xhci\_get\_frame,  .hub\_control = xhci\_hub\_control,  .hub\_status\_data = xhci\_hub\_status\_data,  .bus\_suspend = xhci\_bus\_suspend,  .bus\_resume = xhci\_bus\_resume,  } |   2. 创建primary host controller驱动： HCD\_USB2  |--struct **usb**\_**hcd**\*hcd = **usb**\_**create**\_**hcd**(driver, &pdev->dev, dev\_name(&pdev->dev));  |--usb\_create\_shared\_hcd(driver, dev, bus\_name, **NULL**);  |--hcd->driver = driver; //hc\_driver  |--irq = platform\_get\_irq(pdev, 0);  3.创建primary的usb\_device： usb2.0 root hub, 第一个port number为0  |--**usb**\_**add**\_**hcd**(hcd, irq, IRQF\_SHARED);  3.1 创建xchi host controller并与primary host controller建立关系  |--**hcd**->**driver**->**reset**(hcd) //xhci\_plat\_setup()  |--struct **xhci**\_**hcd** \*xhci = kzalloc(sizeof(struct xhci\_hcd), GFP\_KERNEL);  |--**xhci->main\_hcd = hcd;**  |--**hcd->hcd\_priv = xhci;**  |--hcd->speed = HCD\_**USB2**;  |--**hcd**->**driver**->**start**(hcd); //xhci\_run()  4. 创建xchi的shared host controller驱动： HCD\_USB3  |--hcd = platform\_get\_drvdata(pdev);  |--xhci = hcd\_to\_xhci(hcd);  |--**xhci->shared\_hcd** = **usb**\_**create**\_**shared**\_hcd(driver, &pdev->dev, dev\_name(&pdev->dev), **hcd**);  |--hcd->driver = driver;  |--hcd->speed = driver->flags & HCD\_MASK; //HCD\_USB3  **|-- \*((struct xhci\_hcd \*\*) xhci->shared\_hcd->hcd\_priv) = xhci;**  5.创建shared hub的usb\_device： usb3.0 root hub, 第一个port number为0  |--**usb**\_**add**\_**hcd**(xhci->shared\_hcd, irq, IRQF\_SHARED); |
| **usb\_create\_shared\_hcd**(const struct hc\_driver \*driver, struct device \*dev, const char \*bus\_name, struct usb\_hcd \*primary\_hcd)  1. 创建host controller, 并于primary host controller建立关系  |--struct **usb**\_**hcd** \*hcd =**kzalloc**(sizeof(\*hcd) + driver->hcd\_priv\_size, GFP\_KERNEL);  |-- if (primary\_hcd ！= NULL)  |--hcd->primary\_hcd = primary\_hcd; //建立主从hub的对应关系  |--hcd->shared\_hcd = **primary**\_**hcd**;  |--primary\_hcd->primary\_hcd = primary\_hcd;  |--primary\_hcd->shared\_hcd = **hcd**;   1. 初始化host controller的 bus: usb2.0 , usb3.0   |-- **usb**\_**bus**\_**init**(&hcd->self);  |-- hcd->self.controller = dev;  |-- hcd->self.bus\_name = bus\_name;   1. 创建host controller的定时器   |--hcd->rh\_timer.**function** = **rh**\_**timer**\_**func**;  |--hcd->rh\_timer.data = (unsigned long) hcd;   1. 绑定host controller的上层驱动为hc\_driver   |--hcd->driver = driver;  |--hcd->speed = driver->flags & HCD\_MASK; //HCD\_USB3 |
| **usb\_add\_hcd**(hcd, irq, IRQF\_SHARED);  1. 创建host controller设备的的pool的内存空间: 4种大小的pool  |--**hcd**\_**buffer**\_**create**(hcd))  |--#define HCD\_BUFFER\_POOLS 4  |--size = pool\_max[i];  |--snprintf(name, sizeof name, "buffer-%d", size);  |--hcd->pool[i] = dma\_pool\_create(name, hcd->self.controller, size, size, 0);    2. 注册host controller的bus到usb bus上  |--**usb**\_**register**\_**bus**(&hcd->self)  |--list\_add (&bus->bus\_list, &**usb**\_**bus**\_**list**);    3. 为host controller创建第一个usb\_device设备，端口号为0  |--struct **usb**\_**device** \*rhdev = **usb**\_**alloc**\_**dev**(NULL, &hcd->self, **0**)  |--struct **usb**\_**device** \*dev = kzalloc(sizeof(\*dev), GFP\_KERNEL);  |--dev->dev.bus = &usb\_bus\_type;  |--dev->dev.type = &**usb**\_**device**\_**type**;  |--dev->state = USB\_STATE\_ATTACHED; //设置dev的状态  |--dev->portnum = **port1**; //0  |--dev->bus = bus;  |--dev->dev.parent = bus->controller;  |--usb\_enable\_endpoint(dev, &dev->ep0, false);  |--**hcd->self.root\_hub = rhdev;**  |--**hcd**->**driver**->**reset**(hcd) //xhci\_gen\_setup()  4. 绑定hub的irq中断号, 创建HC的中断函数  |--if (usb\_hcd\_is\_primary\_hcd(hcd) && irqnum)  |--**usb**\_**hcd**\_**request**\_**irqs**(hcd, **irqnum**, irqflags);  |--request\_irq(irqnum, &usb\_hcd\_irq, irqflags, hcd->irq\_descr, hcd);  |--hcd->irq = irqnum;  |--hcd->state = HC\_STATE\_RUNNING;  5. 启动xhci\_run  |--**hcd**->**driver**->**start**(hcd); //xhci\_run()  6. 注册roothub  |--**register**\_**root**\_**hub**(hcd);  |--sysfs\_create\_group(&rhdev->dev.kobj, &usb\_bus\_attr\_group); |
| **register\_root\_hub**(struct usb\_hcd \*hcd)  1. 配置第一个usb\_device：设备号为1  |--const int **devnum = 1;**  |--struct **usb**\_**device** \*usb\_dev = hcd->self.**root**\_**hub**;  |--usb\_dev->devnum = devnum;  |--usb\_dev->bus->devnum\_next = devnum + 1;  |--usb\_set\_device\_state(usb\_dev, USB\_STATE\_ADDRESS);  |--usb\_dev->ep0.desc.wMaxPacketSize = cpu\_to\_le16(64);  |--usb\_get\_device\_descriptor(usb\_dev, USB\_DT\_DEVICE\_SIZE);  |--usb\_get\_bos\_descriptor(usb\_dev); //只针对于USB\_SPEED\_SUPER    2. 获取roothub的配置描述符并解析  |--**usb**\_**new**\_**device**(udev);  |--usb\_enumerate\_device(udev);  2.1 获取USB配置描述符并解析  |--usb\_get\_configuration(udev);  |--**usb**\_**get**\_**descriptor**(**USB**\_**DT**\_**CONFIG**);  |--**usb**\_**parse**\_**configuration**()  |--udev->dev.devt = MKDEV(USB\_DEVICE\_MAJOR, (((udev->bus->busnum-1) \* 128) + (udev->devnum-1)));  2.2 注册USB设备  |--**device**\_**add**(&udev->dev); //触发probe : usb\_device\_type  2.3 创建ep\_device并注册  |--usb\_create\_ep\_devs(&udev->dev, &udev->ep0, udev);  |--struct **ep**\_**device** \*ep\_dev = **kzalloc**(sizeof(\*ep\_dev), GFP\_KERNEL);  |--ep\_dev->dev.type = &usb\_ep\_device\_type;  |--**device**\_**register**(&ep\_dev->dev); //触发probe ： usb\_ep\_device\_type  |--endpoint->ep\_dev = ep\_dev; |

### xhci\_plat\_setup

|  |
| --- |
| **xhci\_plat\_setup**(struct usb\_hcd \*hcd)  |--**xhci\_gen\_setup**(hcd, xhci\_plat\_quirks);  1. 若是第一个HC,创建xhci\_hcd, 建立xhci与hcd的连接  |--if (usb\_hcd\_is\_primary\_hcd(hcd))  |--struct **xhci**\_**hcd** \*xhci = **kzalloc**(sizeof(struct xhci\_hcd), GFP\_KERNEL);  |--xhci->main\_hcd = hcd;  |--hcd->hcd\_priv = xhci;  |--hcd->speed = HCD\_**USB2**;  |--hcd->self.root\_hub->speed = USB\_SPEED\_HIGH;  |--hcd->**has\_tt** = 1; //USB 2.0 roothub under xHCI has an integrated TT  |--else  |--return  2. 初始化xchi  |--**xhci**\_**halt**(xhci);  |--**xhci**\_**reset**(xhci);  |--**xhci**\_**init**(hcd); |

## rh\_timer\_func()

1. 获取hub的status数据
2. 调用中断传输的回调函数: hub\_irq()

|  |
| --- |
| **rh\_timer\_func**() //usb\_hcd   1. 查询host端的第一个usb\_device的状态: root\_hub   |--**usb\_hcd\_poll\_rh\_status**() //usb\_hcd  |--**length** = hcd->driver->hub\_status\_data(hcd, **buffer**); //获取hub的status数据  |--urb = hcd->**status**\_**urb**; //获取hub的status\_urb  |--clear\_bit(HCD\_FLAG\_POLL\_PENDING, &hcd->flags);  |--hcd->status\_urb = NULL;  |--urb->actual\_length = length;  2, 将status data拷贝到urb中  |--**memcpy**(urb->transfer\_buffer, **buffer**, **length**); //将hub的status数据拷贝到status\_urb中  |--usb\_hcd\_unlink\_urb\_from\_ep(dummy\_hcd\_to\_hcd(dum\_hcd), urb);  |--list\_del\_init(&urb->urb\_list); //断开status\_urb在链表中的链接  |--**usb\_hcd\_giveback\_urb**(dummy\_hcd\_to\_hcd(dum\_hcd), urb, status);  |--unmap\_urb\_for\_dma(hcd, urb);  |--usbmon\_urb\_complete(&hcd->self, urb, status); //monitor   1. 调用urb的回调函数，处理urb   |--**urb->complete** (**urb**); //hub\_irq() |
| **xhci\_hub\_status\_data**(struct usb\_hcd \*hcd, char \*buf)  1. 获取ports的status  |--max\_ports = xhci\_get\_ports(hcd, &port\_array);  |--if (hcd->speed == HCD\_USB3)  |--max\_ports = xhci->num\_usb3\_ports;  |--\*port\_array = xhci->usb3\_ports;  |--else  |--max\_ports = xhci->num\_usb2\_ports;  |--\*port\_array = xhci->usb2\_ports;  2. 将ports的status保存到buf中  |--for (i = 0; i < max\_ports; i++)  |--temp = readl(port\_array[i]);  |--buf[(i + 1) / 8] |= 1 << (i + 1) % 8;  |--buflength = (max\_ports + 8) / 8; |

# UVC驱动

## uvc\_init()

|  |  |  |
| --- | --- | --- |
| **module\_init**(uvc\_init);  |--**usb**\_**register**(&**uvc**\_**driver**.driver);   |  | | --- | | struct **uvc**\_**driver** **uvc**\_**driver** = {  .driver = { //struct usb\_driver  .name = "**uvcvideo**",  .**probe** = uvc\_probe,  .disconnect = uvc\_disconnect,  .suspend = uvc\_suspend,  .resume = uvc\_resume,  .reset\_resume = uvc\_reset\_resume,  .**id**\_**table** = **uvc**\_**ids**,  .supports\_autosuspend = 1,  },  }; | | static struct **usb**\_**device**\_**id** **uvc**\_**ids**[] = {  { USB\_INTERFACE\_INFO(USB\_CLASS\_**VIDEO**, 1, 0) },  {}  }; | |

## uvc\_probe

|  |
| --- |
| **uvc**\_**probe**(struct **usb**\_**interface** \*intf,const struct usb\_device\_id \*id)  1. 创建uvc\_device设备  |--struct **uvc**\_**device** \*dev = **kzalloc**(sizeof \*dev, GFP\_KERNEL);  |--dev->intfnum = intf->cur\_altsetting->desc.**bInterfaceNumber**;    2. 解析UVC的control描述符,为UVC描述符中的INPUT/OUTPUT/PROCESSING模块创建uvc\_entity  |--**uvc**\_**parse**\_**control**(dev)  |--uvc\_parse\_**standard**\_control(dev, buffer, buflen)  |--struct **uvc**\_**entity** \*term;  |--switch (buffer[2])  |--case UVC\_VC\_**INPUT**\_TERMINAL  |--term = uvc\_alloc\_entity(type | UVC\_TERM\_INPUT, buffer[3],1, n + p);  |--list\_add\_tail(&term->list, &dev->entities);  |--case UVC\_VC\_**OUTPUT**\_TERMINAL:  |--term = uvc\_alloc\_entity(type | UVC\_TERM\_OUTPUT, buffer[3], 1, 0);  |--list\_add\_tail(&term->list, &dev->entities);  |--case UVC\_VC\_**PROCESSING**\_UNIT:  |--unit = uvc\_alloc\_entity(buffer[2], buffer[3], 2, n);  |--list\_add\_tail(&unit->list, &dev->entities);    3. 注册v4l2设备  |--**v4l2**\_**device**\_**register**(&intf->dev, &dev->vdev)    4. 初始化uvc支持的控制命令  |--**uvc**\_**ctrl**\_**init**\_**device**(dev) //解析描述符中的bmControls && bControlSize字段    5. 创建uvc\_video\_chain : SU{0,1} -> PU{0,1} -> XU{0,n}  |--**uvc**\_**scan**\_**device**(dev)  |--struct **uvc**\_**entity** \*term;  |--list\_for\_each\_entry(term, &dev->entities, list)  |--struct **uvc**\_**video**\_**chain** \*chain = kzalloc(sizeof(\*chain), GFP\_KERNEL);  |--**uvc**\_**scan**\_**chain**(chain, term)  |--uvc\_scan\_chain\_entity(chain, entity)  |-- list\_add\_tail(&entity->chain, &chain->entities);  |--uvc\_scan\_chain\_forward(chain, entity, prev)  |-- list\_add\_tail(&forward->chain, &chain->entities);  |--uvc\_scan\_chain\_backward(chain, &entity)  |--list\_add\_tail(&chain->list, &dev->chains);    6. 创建video\_device并注册  |--**uvc**\_**register**\_**chains**(dev)  |--struct **uvc**\_**video**\_**chain** \*chain;  |--list\_for\_each\_entry(chain, &dev->chains, list)  |--uvc\_register\_terms(dev, chain);  |--struct **uvc**\_**entity** \*term;  |--list\_for\_each\_entry(term, &chain->entities, chain)  |--struct **uvc**\_**streaming** \*stream = uvc\_stream\_by\_id(dev, term->id);  |--stream->chain = chain;  |--**uvc**\_**register**\_**video**(dev, stream);  |--term->vdev = stream->vdev;    7. 设置中断传输处理函数  |--**uvc**\_**status**\_**init**(dev);  |--struct **usb**\_**host**\_**endpoint** \*ep = dev->int\_ep;  |--dev->**status** = **kzalloc**(UVC\_MAX\_STATUS\_SIZE, GFP\_KERNEL);  |--dev->**int**\_**urb** = **usb**\_**alloc**\_**urb**(0, GFP\_KERNEL);  |--pipe = usb\_rcvintpipe(dev->udev, ep->desc.bEndpointAddress);  |--interval = ep->desc.bInterval;  |--**usb**\_**fill**\_**int**\_**urb**(dev->**int**\_**urb**, dev->udev, pipe,dev->**status**, UVC\_MAX\_STATUS\_SIZE, **uvc**\_**status**\_**complete**, dev, interval); |
| **uvc**\_**register**\_**video**(struct uvc\_device \*dev,struct uvc\_streaming \*stream)  1. 初始化streaming interface  |--**uvc**\_**video**\_**init**(stream);  |--uvc\_queue\_init(&stream->queue, stream->type, !uvc\_no\_drop\_param);  |--uvc\_get\_video\_ctrl(stream, **probe**, 1, UVC\_GET\_**DEF**)  |--uvc\_set\_video\_ctrl(stream, probe, 1);  |--uvc\_get\_video\_ctrl(stream, **probe**, 1, UVC\_GET\_**CUR**);  |--if (stream->intf->num\_altsetting > 1)  |--**stream->decode = uvc\_video\_decode\_isoc;**  |--else  |--stream->decode = uvc\_video\_decode\_bulk;    2. 创建video\_device并注册  |--struct **video**\_**device** \*vdev = **video\_device\_alloc**();  |--vdev->v4l2\_dev = &dev->vdev;  |--vdev->fops = &**uvc**\_**fops**;  |--**video\_register\_device**(vdev, VFL\_TYPE\_GRABBER, uvc\_video\_device\_node\_number);  |--stream->vdev = vdev;   |  | | --- | | const struct **v4l2**\_**file**\_**operations** **uvc**\_**fops** = {  .owner = THIS\_MODULE,  .open = uvc\_v4l2\_open,  .release = uvc\_v4l2\_release,  .unlocked\_ioctl = uvc\_v4l2\_ioctl,  .compat\_ioctl32 = uvc\_v4l2\_compat\_ioctl32,  .read = uvc\_v4l2\_read,  .mmap = uvc\_v4l2\_mmap,  .poll = uvc\_v4l2\_poll,  }; | |

## uvc\_status\_complete: 中断传输函数的回调函数

|  |
| --- |
| **uvc**\_**status**\_**complete**(struct urb \*urb)  1. 处理urb  |--struct uvc\_device \*dev = urb->context;  |--len = urb->actual\_length;  |--switch (dev->status[0] & 0x0f)  |--case UVC\_STATUS\_TYPE\_**CONTROL**:  |--**uvc**\_**event**\_**control**(dev, dev->status, len);break;  |--case UVC\_STATUS\_TYPE\_**STREAMING**:  |--**uvc**\_**event**\_**streaming**(dev, dev->status, len);break;  2. 释放urb，再次使能endpoint的中断传输  |--urb->interval = dev->int\_ep->desc.**bInterval**;  |--**usb**\_**submit**\_**urb**(urb, GFP\_ATOMIC) |

## uvc\_video\_enable：启动vb2，设置isoc传输或bulk传输

|  |
| --- |
| **uvc\_video\_enable**(struct uvc\_streaming \*stream, int enable)  1. 启动vb2功能  |--**uvc**\_**queue**\_**enable**(&stream->queue, 1);  |--**vb2**\_**streamon**(&queue->queue, queue->queue.type);  |--queue->buf\_used = 0;  |--queue->framesdropped = 0;  2. 发送COMMIT消息，需要commit的数据放在data中  |--**uvc**\_**commit**\_**video**(stream, &stream->ctrl);  |--uvc\_set\_video\_ctrl(stream, ctrl, 0); //probe=0  |--\_\_uvc\_query\_ctrl(stream->dev, UVC\_**SET**\_**CUR**, 0, stream->intfnum,  probe ? UVC\_VS\_PROBE\_CONTROL : UVC\_VS\_**COMMIT**\_CONTROL, **data**,  size, uvc\_timeout\_param);  3. 配置isoc传输或bulk传输的URB  |--**uvc**\_**init**\_**video**(stream, GFP\_KERNEL); |
| **uvc\_init\_video**(struct uvc\_streaming \*stream, gfp\_t gfp\_flags);  |--struct usb\_interface \*intf = stream->intf;  1 isoc传输:  |--if (intf->num\_altsetting > 1)  1.1 寻找dwMaxPayloadTransferSize<= bandwidth <=UINT\_MAX的端点配置  |--unsigned int **best**\_**psize** = UINT\_MAX;  |--**bandwidth** = stream->ctrl.dwMaxPayloadTransferSize;  |--for (i = 0; i < intf->num\_altsetting; ++i)  |--struct usb\_host\_interface \*alts = &intf->altsetting[i];  |--ep = **uvc**\_**find**\_**endpoint**(alts, stream->header.bEndpointAddress);  |--**psize** = uvc\_endpoint\_max\_bpi(stream->dev->udev, ep);  |--if (psize >= bandwidth && psize <= best\_psize)  |--best\_psize = psize;  |--best\_ep = ep;  |--**altsetting** = alts->desc.bAlternateSetting;  1.2 将选定的端点配置告诉udc设备  |--**usb**\_**set**\_**interface**(stream->dev->udev, intfnum, altsetting);  1.3 配置isoc传输，回调函数为uvc\_video\_complete()  |--**uvc**\_**init**\_**video**\_**isoc**(stream, best\_ep, gfp\_flags);  |--size = stream->ctrl.**dwMaxVideoFrameSize**;  |--psize = uvc\_endpoint\_max\_bpi(stream->dev->udev, ep);  |--**npackets** = **uvc\_alloc\_urb\_buffers(**stream, size, psize, gfp\_flags);  |--**size** = npackets \* psize;  |--for (i = 0; i < **UVC**\_**URBS**; ++i)  |--struct urb \*urb = **usb\_alloc\_urb**(npackets, gfp\_flags);  |--urb->dev = stream->dev->udev;  |--urb->context = stream;  |--urb->interval = ep->desc.bInterval;  |--urb->transfer\_buffer = stream->urb\_buffer[i];  |--urb->complete = **uvc\_video\_complete**;  |--urb->number\_of\_packets = **npackets**;  |--urb->transfer\_buffer\_length = **size**;  |--for (j = 0; j < npackets; ++j)  |--urb->iso\_frame\_desc[j].offset = j \* psize;  |--urb->iso\_frame\_desc[j].length = psize;  |--stream->urb[i] = urb;  2 bulk传输  |--else  2.1 获取bEndpointAddress地址的端点配置  |--ep = **uvc\_find\_endpoint**(&intf->altsetting[0],stream->header.bEndpointAddress);  2.2 配置bulk传输，回调函数为uvc\_video\_complete()  |--**uvc\_init\_video\_bulk**(stream, ep, gfp\_flags);  |-- psize = usb\_endpoint\_maxp(&ep->desc) & 0x7ff;  |-- size = stream->ctrl.**dwMaxPayloadTransferSize**;  |--**npackets** = **uvc\_alloc\_urb\_buffers**(stream, size, psize, gfp\_flags);  |-- **size** = npackets \* psize;  |--for (i = 0; i < **UVC**\_**URBS**; ++i)  |--struct urb \*urb = usb\_alloc\_urb(0, gfp\_flags);  |--**usb\_fill\_bulk\_urb**(urb, stream->dev->udev, pipe,stream->urb\_buffer[i], **size**, **uvc\_video\_complete**,stream);  |--stream->urb[i] = urb;  3. 提交URB给  |--for (i = 0; i < **UVC**\_**URBS** /\* 5 \*/; ++i)  |--**usb\_submit\_urb**(stream->urb[i], gfp\_flags); |

# Usb gadget设备

|  |
| --- |
| request\_list链表： 存放webcam的req buffer  req\_queued链表: 存放dwc3需要处理的trb buffer  **启动流程:**   1. dwc3\_gadget\_start()： 创建dwc3\_interrupt()和dwc3\_thread\_interrupt()中断处理函数。dwc->gadget\_driver指向webcam驱动。   **数据处理流程:**   1. dwc3\_gadget\_ep\_queue()：webcam驱动将usb\_request数据添加到request\_list链表中，调用\_\_dwc3\_gadget\_kick\_transfer()。 2. \_\_dwc3\_gadget\_kick\_transfer()： 3. 向trb\_pool[]申请可用的trb，并将usb\_request信息存放到trb中，将usb\_request添加到req\_queued链表中。 4. 从req\_queued链表中取出usb\_request的trb，将trb的参数写到ep的相关寄存器中，设置ep的DWC3\_DEPCMD\_STARTTRANSFER命令，启动ep硬件功能。 5. 当ep的数据发送完之后，会产生中断，唤醒dwc3\_interrupt() 6. dwc3\_interrupt()查询DWC3\_GEVNTCOUNT，若event的count大于0，则有事件需要处理，唤醒dwc3\_thread\_interrupt()。 7. dwc3\_thread\_interrupt(): 若事件类型为ep事件，调用dwc3\_endpoint\_transfer\_complete ()。 8. dwc3\_endpoint\_transfer\_complete ()：更新usb\_request的信息，调用webcam的回调函数req->request.complete()。删除request\_list链表和req\_queued链表的挂载。 |

## dwc3\_gadget\_init

|  |  |  |  |
| --- | --- | --- | --- |
| **dwc3**\_**gadget**\_**init**(dwc);  1. 初始化usb\_gadget  |--dwc->**gadget.ops**  = &**dwc3\_gadget\_ops;**  |--dwc->gadget.max\_speed = USB\_SPEED\_SUPER;  |--dwc->gadget.name = **"dwc3-gadget";**   |  | | --- | | static const struct usb\_gadget\_ops dwc3\_gadget\_ops = {  .pullup = dwc3\_gadget\_pullup,  --| usb\_gadget\_connect()  --| usb\_gadget\_disconnect()  .**udc\_start = dwc3\_gadget\_start,**  --| usb\_gadget\_udc\_start()  --| usb\_gadget\_probe\_driver()  --| webcam\_init()  --| dwc3\_gadget\_plugin()  --| dwc3\_plug\_in() -- if(s == PLUGSTATE\_B\_IN)  .udc\_stop = dwc3\_gadget\_stop,  .get\_frame = dwc3\_gadget\_get\_frame,  .wakeup = dwc3\_gadget\_wakeup,  .set\_selfpowered = dwc3\_gadget\_set\_selfpowered,  }; |   2. 配置gadget的in/out endpoint  |--**dwc3**\_**gadget**\_**init**\_**endpoints**() //初始化gadget的in/out endpoint  |--dwc3\_gadget\_init\_hw\_endpoints(dwc, dwc->num\_out\_eps, **0**); //dwc3 + num + direction(0:out)  |--dwc3\_gadget\_init\_hw\_endpoints(dwc, dwc->num\_in\_eps, **1**); //dwc3 + num + direction(1:in)  |--for (i = 0; i < num; i++)  |--struct **dwc3**\_**ep** \*dep = **kzalloc**(sizeof(\*dep), GFP\_KERNEL);  |--snprintf(dep->name, sizeof(dep->name), "ep%d%s", epnum >> 1, (epnum & 1) ? "in" : "out");  |--if (epnum == 0 || epnum == 1)  |--dep->endpoint.maxpacket = 512;  |--dep->**endpoint.ops** = **&dwc3\_gadget\_ep0\_ops;**  |--dwc->gadget.ep0 = &dep->endpoint;  |--else  |--dep->endpoint.maxpacket = 1024;  |--dep->**endpoint.ops** = **&dwc3\_gadget\_ep\_ops;**  |--**dwc3**\_**alloc**\_**trb**\_**pool**(dep); //DWC3\_TRB\_NUM  |--list\_add\_tail(&dep->endpoint.ep\_list, &dwc->gadget.ep\_list);   |  | | --- | | static const struct usb\_ep\_ops **dwc3**\_**gadget**\_**ep**\_**ops** = {  .enable = dwc3\_gadget\_ep\_enable,  .disable = dwc3\_gadget\_ep\_disable,  .alloc\_request = dwc3\_gadget\_ep\_alloc\_request,  .free\_request = dwc3\_gadget\_ep\_free\_request,  .queue = dwc3\_gadget\_ep\_queue,  .dequeue = dwc3\_gadget\_ep\_dequeue,  .set\_halt = dwc3\_gadget\_ep\_set\_halt,  .set\_wedge = dwc3\_gadget\_ep\_set\_wedge,  }; | | static const struct usb\_ep\_ops **dwc3**\_**gadget**\_**ep0**\_**ops** = {  .enable = dwc3\_gadget\_ep0\_enable,  --| usb\_ep\_enable()  .disable = dwc3\_gadget\_ep0\_disable,  .alloc\_request = dwc3\_gadget\_ep\_alloc\_request,  --| usb\_ep\_alloc\_request()  .free\_request = dwc3\_gadget\_ep\_free\_request,  .queue = dwc3\_gadget\_ep0\_queue,  --| usb\_ep\_queue()  .dequeue = dwc3\_gadget\_ep\_dequeue,  --| usb\_ep\_dequeue()  .set\_halt = dwc3\_gadget\_ep0\_set\_halt,  .set\_wedge = dwc3\_gadget\_ep\_set\_wedge,  }; |   3. 注册gadget  |--**usb**\_**add**\_**gadget**\_**udc**(dwc->dev, &dwc->gadget); //device + usb\_gadget  |--usb\_add\_gadget\_udc\_release(parent, gadget, NULL); //device + usb\_gadget + release()  |--dev\_set\_name(&gadget->dev, "**gadget**");  |--INIT\_WORK(&gadget->work, usb\_gadget\_state\_work);  |--gadget->dev.release = usb\_udc\_nop\_release;  |--**device**\_**register**(&gadget->dev);  4. 创建udc\_class类，并将gadget驱动绑定到udc\_list中  |--struct **usb**\_**udc** \*udc = **kzalloc**(sizeof(\*udc), GFP\_KERNEL);  |--udc->dev.release = usb\_udc\_release;  |--udc->dev.**class** = **udc**\_**class**;  |--dev\_set\_name(&udc->dev, "%s", kobject\_name(&parent->kobj));  |--udc->**gadget** = **gadget**;  |--**list\_add\_tail(&udc->list, &udc\_list);**  |--**device**\_**add**(&udc->dev);  |--usb\_gadget\_set\_state(gadget, USB\_STATE\_NOTATTACHED); |

## dwc3\_gadget\_start() : 作为usb camera设备时会启动这个函数

|  |  |
| --- | --- |
| **dwc3\_gadget\_start**() //usb\_gadget + usb\_gadget\_driver   1. 创建中断线程   |--**request**\_**threaded**\_**irq**(irq, **dwc3**\_**interrupt**, **dwc3**\_**thread**\_**interrupt**, IRQF\_SHARED | IRQF\_ONESHOT, "dwc3", dwc);  |--dwc->gadget\_driver = driver;  |--dwc3\_gadget\_ep0\_desc.wMaxPacketSize = cpu\_to\_le16(512); //SuperSpeed   1. 使能endpoint，并设定端点描述符   |-- dep = dwc->**eps[0];**  |--\_\_dwc3\_gadget\_**ep**\_**enable**(dep, &**dwc3**\_**gadget**\_**ep0**\_**desc**, NULL, false); //ep[0]  |-- dep = dwc->**eps[1];**  |--\_\_dwc3\_gadget\_**ep**\_**enable**(dep, &**dwc3**\_**gadget**\_**ep0**\_**desc**, NULL, false); //ep[1]   |  | | --- | | static struct usb\_endpoint\_descriptor **dwc3**\_**gadget**\_**ep0**\_**desc** = {  .bLength = USB\_DT\_ENDPOINT\_SIZE,  .bDescriptorType = USB\_DT\_**ENDPOINT**,  .bmAttributes = USB\_ENDPOINT\_XFER\_**CONTROL**,  }; |  1. 允许接收SETUP数据包，使能irq.   |--dwc->ep0state = EP0\_**SETUP**\_PHASE;  |--**dwc3**\_ep0\_**out**\_**start**(dwc);  |--**dwc3**\_**gadget**\_**enable**\_**irq**(dwc); |

## dwc3\_interrupt() : dwc3的中断处理函数(查看是否有中断)

|  |
| --- |
| **dwc3\_interrupt**(irq + \_dwc) //dwc3有中断触发，读寄存器状态看是否有事件触发   1. 查询dwc3所有的event buffer, 若有event，则唤醒dwc3\_thread\_interrupt   |--for (i = 0; i < dwc->num\_event\_buffers; i++)  |--status = dwc3\_process\_event\_buf(dwc, i);  |--evt = dwc->ev\_buffs[buf];  |--count = **dwc3**\_**readl**(dwc->regs, **DWC3**\_**GEVNTCOUNT**(buf));  |--**if count > 0**  |--evt->count = count;  |--evt->flags |= DWC3\_EVENT\_PENDING; //设置evt  |-**-return IRQ\_WAKE\_THREAD**  |--if (status == IRQ\_WAKE\_THREAD)  |--**wakeup** : dwc3\_thread\_interrupt(irq + \_dwc) |

## dwc3\_thread\_interrupt()： dwc3的中断处理函数(处理中断)

|  |
| --- |
| **dwc3\_thread\_interrupt**(irq + \_dwc)  1. 若event buffer的PENDING状态位置起，且event count大于零，则表示有事件  |--for (i = 0; i < dwc->num\_event\_buffers; i++)  |--evt = dwc->ev\_buffs[i];  |--if evt->flags == DWC3\_EVENT\_**PENDING**  |--**if evt->count > 0**  |--event.raw = \*(u32 \*) (evt->buf + evt->lpos);  |--**dwc3**\_**process**\_**event**\_**entry**(dwc, &event);  |--evt->lpos = (evt->lpos + 4) % DWC3\_EVENT\_BUFFERS\_SIZE;  |--evt->count -= 4 |
| **dwc3\_process\_event\_entry**(dwc, &event);  1. 处理dwc3 endpoint event：isoc, bulk等传输事件  |--if (event->type.**is**\_**devspec** == 0)  |--**dwc3**\_**endpoint**\_**interrupt**()  1.1 ep0端口(in/out)：control传输(dwc3\_gadget\_ep0\_desc)  |--if (epnum == 0 || epnum == 1)  |-- dwc3\_ep0\_interrupt(dwc, event);  |--if event->endpoint\_event == DWC3\_DEPEVT\_**XFERCOMPLETE**  |--**dwc3**\_**ep0**\_**xfer**\_**complete**(dwc, event);  |--if event->endpoint\_event == DWC3\_DEPEVT\_**XFERNOTREADY**  |--**dwc3**\_**ep0**\_**xfernotready**(dwc, event);  1.2 非ep0端口: isoc传输或bulk传输  |--else //  |--if event->endpoint\_event == DWC3\_DEPEVT\_**XFERCOMPLETE**  |--if usb\_endpoint\_xfer != isoc  //非isoc传输，当端口收到complete事件后需要回传  //dwc3\_endpoint\_transfer\_complete(1)  |--**dwc3**\_**endpoint**\_**transfer**\_**complete**(dwc, dep, event, 1);  |--if event->endpoint\_event == DWC3\_DEPEVT\_**XFERINPROGRESS**  |--if usb\_endpoint\_xfer == isoc  //isoc传输，当端口收到progress事件后需要回传  //dwc3\_endpoint\_transfer\_complete(0)  |--**dwc3**\_**endpoint**\_**transfer**\_**complete**(dwc, dep, event, 0);  |--if event->endpoint\_event == DWC3\_DEPEVT\_**XFERNOTREADY**  |--if usb\_endpoint\_xfer == isoc  //当端口收到ready事件后需要start\_isoc()或kick\_transfer()  |--**dwc3**\_**gadget**\_**start**\_**isoc**(dwc, dep, event);  |--else  |--**\_\_dwc3\_gadget**\_**kick**\_**transfer**(dep, 0, 1);  2. 处理device event：设备断开，连接事件  |--if event->type.type == DWC3\_EVENT\_TYPE\_**DEV** //  |--**dwc3**\_**gadget**\_**interrupt**(dwc, &event->devt);  2.1 断开事件  |--if event->type == DWC3\_DEVICE\_EVENT\_**DISCONNECT**  |--dwc3\_gadget\_**disconnect**\_**interrupt**()  |--dwc->gadget\_driver->**disconnect**(&dwc->gadget);  2.2 复位事件  |--if event->type == DWC3\_DEVICE\_EVENT\_**RESET**  |--dwc3\_gadget\_**reset**\_**interrupt**()  2.3 连接事件  |--if event->type == DWC3\_DEVICE\_EVENT\_**CONNECT**\_**DONE**  |--dwc3\_gadget\_**conndone**\_**interrupt**()  |--dwc->speed = dwc3\_readl(dwc->regs, DWC3\_DSTS);  |--if speed == DWC3\_DCFG\_SUPERSPEED  |--dwc->gadget.ep0->maxpacket = 512;  |--dwc->gadget.speed = USB\_SPEED\_SUPER;  |--\_\_dwc3\_gadget\_**ep**\_**enable**(dep, &dwc3\_gadget\_ep0\_desc, NULL, true); //ep[0]  |--\_\_dwc3\_gadget\_**ep**\_**enable**(dep, &dwc3\_gadget\_ep0\_desc, NULL, true); //ep[1]  2.4 唤醒事件  |--if event->type == DWC3\_DEVICE\_EVENT\_**WAKEUP**  |--dwc3\_gadget\_**wakeup**\_**interrupt**()  |--dwc->gadget\_driver->**resume**(&dwc->gadget);  2.4 连接状态切换事件  |--if event->type == DWC3\_DEVICE\_EVENT\_**LINK**\_**STATUS**\_**CHANGE**  |--dwc3\_gadget\_**linksts**\_**change**\_**interrupt**() |

## \_\_dwc3\_gadget\_kick\_transfer : 启动端口传输

|  |
| --- |
| **\_\_dwc3\_gadget\_kick\_transfer**(struct dwc3\_ep \*dep, u16 cmd\_param,int start\_new)  1. 从req\_queued队列中取出dwc3\_request  |--dep->flags &= ~DWC3\_EP\_**PENDING**\_REQUEST;  |--if (start\_new)  |--if (list\_empty(&dep->req\_queued))  |--**dwc3**\_**prepare**\_**trbs**(dep, start\_new);  |--req = **next**\_**request**(&dep->req\_queued);  |--else  |--**dwc3**\_**prepare**\_**trbs**(dep, start\_new);  |--req = **next**\_**request**(&dep->req\_queued);  2. 若req\_queued队列为空，则置起 DWC3\_EP\_PENDING\_REQUEST 状态位，并返回  |--if (!req)  |--dep->flags |= DWC3\_EP\_**PENDING**\_REQUEST;  |--**return**  3. 将dwc3\_request中的trb地址和参数写到dwc3寄存器中，启动STARTTRANSFER命令  |--else  |--params.param0 = upper\_32\_bits(req->**trb**\_**dma**);  |--params.param1 = lower\_32\_bits(req->**trb**\_**dma**);  |--cmd = DWC3\_DEPCMD\_**STARTTRANSFER**;  |--ret = **dwc3\_send\_gadget\_ep\_cmd**(dwc, dep->number, **cmd**, &params);  |--if(ret < 0) //若发送命令失败，丢弃这个req  |--usb\_gadget\_unmap\_request(&dwc->gadget, &req->request, req->direction);  |--list\_del(&req->list);  |--else //若发送成功，则设置BUSY状态位，等待dec3\_thread中断  |--dep->**flags** |= DWC3\_EP\_**BUSY**; //设置BUSY标志  |--dep->**resource**\_**index** = dwc3\_gadget\_ep\_get\_transfer\_index(dwc, dep->number); |
| **dwc3\_prepare\_trbs**(dep, start\_new);  1. 若当前req与上个req是关联的，且当前trb已满，则丢弃  |--trbs\_left = (dep->busy\_slot - dep->free\_slot) & DWC3\_TRB\_MASK;  |--if (!usb\_endpoint\_xfer\_isoc(dep->endpoint.desc))  |--trbs\_left = DWC3\_TRB\_NUM - (dep->free\_slot & DWC3\_TRB\_MASK);    |--if (!trbs\_left)  |--if (!starting)  |--return //trb已满，但这个req与上个req是有关联的，则丢弃  |--if (usb\_endpoint\_xfer\_isoc(dep->endpoint.desc))  |--dep->busy\_slot = dep->free\_slot = 1; //isoc传输需要跳过第0个trb  |--else  |--dep->busy\_slot = dep->free\_slot = 0;    |--if ((trbs\_left <= 1) && usb\_endpoint\_xfer\_isoc(dep->endpoint.desc))  |--return //trb已满，且为isoc传输，则丢弃  2. 获取request\_list中所有的req  |--**list\_for\_each\_entry\_safe(req, n, &dep->request\_list, list)**  |--if (req->request.num\_mapped\_sgs > 0)  |--else  3. 将req的信息保存到trb\_pool中的trb中, free\_slot++  |--**dma** = req->request.dma;  |--length = req->request.length;  |--trbs\_left--;  |--if(trbs\_left == 0)  |--last\_one = 1;  |--if (list\_is\_last(&req->list, &dep->request\_list))  |--last\_one = 1;  |--**dwc3**\_**prepare**\_one\_trb(dep, req, **dma**, length, last\_one, false, 0);  |--**trb = &dep->trb\_pool[**dep->free\_slot & DWC3\_TRB\_MASK];  |--dep->**free\_slot++;**  |--trb->size = DWC3\_TRB\_SIZE\_LENGTH(length);  |--trb->**bpl** = lower\_32\_bits(**dma**);  |--trb->**bph** = upper\_32\_bits(**dma**);  |--trb->ctrl |= DWC3\_TRB\_CTRL\_HWO;  4. 将trb添加到req中，并将req挂载到req\_queued队列中, start\_slot = free\_slot  |--if (!req->trb)  |--dwc3\_gadget\_move\_request\_queued(req);  |--req->queued = true;  |--**list\_move\_tail(&req->list, &dep->req\_queued);**  |--**req->trb = trb;**  |--req->trb\_dma = dwc3\_trb\_dma\_offset(dep, trb);  |--req->**start\_slot** = dep->**free**\_**slot** & DWC3\_TRB\_MASK;  |--if(last\_one)  |--break //list\_for\_each\_entry\_safe(req, n, &dep->request\_list, list) |

## dwc3\_endpoint\_transfer\_complete () : 端口数据传输结束

|  |
| --- |
| **dwc3\_endpoint\_transfer\_complete**(dwc, dep, event, 0);  |--clean\_busy = **dwc3**\_**cleanup**\_**done**\_**reqs**(dwc, dep, event, status);  |--if (clean\_busy)  |--dep->flags &= **~DWC3\_EP\_BUSY;** |
| **dwc3\_cleanup\_done\_reqs(**dwc, dep, event, status);  1. 从 req\_queued队列中取出req, 根据req->start\_slot找到trb  |--req = **next**\_**request**(&dep->req\_queued);  |--slot = (req->**start**\_**slot** + i) % DWC3\_TRB\_NUM;  |--trb = &**dep->trb\_pool[slot];**  |--ret = \_\_**dwc3**\_**cleanup**\_**done**\_**trbs**(dwc, dep, req, trb,event, status);  |--count = trb->size & DWC3\_TRB\_SIZE\_MASK;  2. 更新req的actual值  |--req->request.**actual** += req->request.**length** - **count**;  3. 将req从req\_queue队列和request\_list队列中删除  |--**dwc3**\_**gadget**\_**giveback**(dep, req, status);  |--if (req->queued)  |--dep->**busy\_slot++;**  |--req->queued = false;  |--**list\_del(&req->list);**  |--req->trb = NULL;  |--usb\_gadget\_unmap\_request(&dwc->gadget, &req->request,req->direction);  4. 调用req的回调函数  |--**req**->**request**.**complete**(&dep->endpoint, &req->request);  5. 设置isoc的端点的状态  |--if (usb\_endpoint\_xfer\_isoc(dep->endpoint.desc) && list\_empty(&dep->req\_queued))  5.1 isoc传输 + req\_queued为空 + request\_list为空 --> EP PENDING  |--if (list\_empty(&dep->request\_list))  |--dep->flags = DWC3\_EP\_**PENDING**\_REQUEST;  5.2 isoc传输 + req\_queued为空 + request\_list不为空 --> EP ENABLE  |---else  |--dwc3\_stop\_active\_transfer(dwc, dep->number);  |--dep->flags = DWC3\_**EP**\_**ENABLED**; |

## dwc3\_gadget\_ep\_queue: 将req添加到端口上

|  |
| --- |
| **dwc3\_gadget\_ep\_queue**(struct usb\_ep \*ep, struct usb\_request \*request)  1.将req添加到request\_list链表中  |--**\_\_dwc3\_gadget\_ep\_queue**(dep, req); //dwc3\_ep + dwc3\_request  |--usb\_gadget\_map\_request(&dwc->gadget, &req->request, dep->direction);  |--**list\_add\_tail(&req->list, &dep->request\_list);**  //将req添加到request\_list队列中    |--if (dep->flags & DWC3\_EP\_**PENDING**\_REQUEST) //PENDING位置起  |--if (usb\_endpoint\_xfer\_isoc(dep->endpoint.desc)) //isoc传输  |--if (list\_empty(&dep->req\_queued))  |--dwc3\_stop\_active\_transfer(dwc, dep->number);  |--dep->flags = DWC3\_EP\_ENABLED;  |--return 0;  |--else //bulk传输  |--**\_\_dwc3\_gadget\_kick\_transfer**(dep, **0,** true); //START\_TRANSFER  |--if (usb\_endpoint\_xfer\_isoc(dep->endpoint.desc) && (dep->flags & DWC3\_EP\_BUSY) && !(dep->flags & DWC3\_EP\_MISSED\_ISOC))  |--**\_\_dwc3\_gadget\_kick\_transfer(**dep, **dep->resource\_index,** false); //UPDATE\_TRANSFER |

## dwc3\_gadget\_ep\_dequeue : 取消端口上的req传输

|  |
| --- |
| **dwc3\_gadget\_ep\_dequeue**()  |--**list\_for\_each\_entry(r, &dep->request\_list, list)**  |--if (r != req) //在request\_list没有找到指定的req，且换到req\_queued中找  |--list\_for\_each\_entry(r, &dep->req\_queued, list)  |--if (r == req) //在req\_queued中找到了req  |--dwc3\_stop\_active\_transfer(dwc, dep->number);  |--cmd=DWC3\_DEPCMD\_**ENDTRANSFER** | DWC3\_DEPCMD\_HIPRI\_FORCERM | DWC3\_DEPCMD\_CMDIOC | dep->resource\_index  |--**dwc3**\_**send**\_**gadget**\_**ep**\_**cmd**(dwc, dep->number, cmd, &params);  |--dep->flags &= **~DWC3\_EP\_BUSY**;  |--else  |--ret = -EINVAL;  |--**dwc3**\_**gadget**\_**giveback**(dep, req, -ECONNRESET);  |--**list\_del(&req->list);**  |--**req**->**request**.**complete**(&dep->endpoint, &req->request); |

## dwc3\_ep0\_xfer\_complete：CONTROL传输命令处理

|  |
| --- |
| **dwc3\_ep0\_xfer\_complete**(struct dwc3 \*dwc,const struct dwc3\_event\_depevt \*event)  |--switch (dwc->**ep0state**) {  1. SETUP事件  |--case EP0\_**SETUP**\_PHASE:  |--**dwc3**\_**ep0**\_**inspect**\_**setup**(dwc, event);break;  2. DATA事件  |--case EP0\_**DATA**\_PHASE:  |--dwc3\_ep0\_complete\_**data**(dwc, event);break;  |--ep0 = dwc->eps[0];  |--r = next\_request(&ep0->request\_list);  |--dwc3\_gadget\_giveback(ep0, r, 0); //调用req的回调函数  3. STATUS事件  |--case EP0\_**STATUS**\_PHASE:  |--dwc3\_ep0\_complete\_**status**(dwc, event);break; |
| **dwc3\_ep0\_inspect\_setup**(struct dwc3 \*dwc, const struct dwc3\_event\_depevt \*event)  |--struct usb\_ctrlrequest \*ctrl = dwc->ctrl\_req;  1. SETUP标准USB事件  |--if ((**ctrl**->**bRequestType** & USB\_TYPE\_MASK) == USB\_TYPE\_**STANDARD**)  |--ret = **dwc3**\_**ep0**\_**std**\_**request**(dwc, ctrl);  2. SETUP非标准USB事件  |--else  |--ret = dwc3\_ep0\_**delegate**\_**req**(dwc, ctrl); |

### dwc3\_ep0\_std\_request：处理标准USB的SETUP事件

|  |
| --- |
| **dwc3\_ep0\_std\_request**(struct dwc3 \*dwc, struct usb\_ctrlrequest \*ctrl)  switch (ctrl->**bRequest**) {  case USB\_REQ\_GET\_**STATUS**:  ret = dwc3\_ep0\_handle\_**status**(dwc, ctrl);break;  case USB\_REQ\_CLEAR\_**FEATURE**:  ret = dwc3\_ep0\_handle\_**feature**(dwc, ctrl, **0);**break;  case USB\_REQ\_SET\_**FEATURE**:  ret = dwc3\_ep0\_handle\_**feature**(dwc, ctrl, **1**);break;  case USB\_REQ\_SET\_**ADDRESS**:  ret = dwc3\_ep0\_set\_**address**(dwc, ctrl);break;  case USB\_REQ\_SET\_**CONFIGURATION**:  ret = dwc3\_ep0\_set\_**config**(dwc, ctrl);break;  case USB\_REQ\_SET\_**SEL**:  ret = dwc3\_ep0\_set\_**sel**(dwc, ctrl);break;  case USB\_REQ\_SET\_ISOCH\_**DELAY**:  ret = dwc3\_ep0\_set\_isoch\_**delay**(dwc, ctrl);break;  default:  ret = **dwc3**\_**ep0**\_**delegate**\_**req**(dwc, ctrl);break; |
| **dwc3\_ep0\_set\_config**(struct dwc3 \*dwc, struct usb\_ctrlrequest \*ctrl)  enum usb\_device\_state state = dwc->gadget.state;  switch (state) {  case USB\_STATE\_DEFAULT:  break;  case USB\_STATE\_**ADDRESS**:  ret = **dwc3**\_**ep0**\_**delegate**\_**req**(dwc, ctrl);  usb\_gadget\_set\_state(&dwc->gadget, USB\_STATE\_CONFIGURED);  reg = dwc3\_readl(dwc->regs, DWC3\_DCTL);  reg |= (DWC3\_DCTL\_ACCEPTU1ENA | DWC3\_DCTL\_ACCEPTU2ENA);  dwc3\_writel(dwc->regs, DWC3\_DCTL, reg);  break;  case USB\_STATE\_**CONFIGURED**:  ret = **dwc3**\_**ep0**\_**delegate**\_**req**(dwc, ctrl);  break; |

### dwc3\_ep0\_delegate\_req：处理非标准USB的SETUP事件

|  |
| --- |
| **dwc3\_ep0\_delegate\_req**(struct dwc3 \*dwc, struct usb\_ctrlrequest \*ctrl)   1. 非标准USB的SETUP事件，直接交给wencam驱动处理   |--**dwc->gadget\_driver->setup**(&dwc->gadget, ctrl); // composite\_setup |

# Usb Gadget驱动: webcam

## webcam\_init: 注册usb\_composite\_driver驱动

|  |  |  |
| --- | --- | --- |
| **webcam\_init**()   1. 创建并注册usb\_composite\_driver驱动   |--**usb**\_**composite**\_**probe**(&**webcam**\_**driver**);   |  | | --- | | static \_\_refdata struct **usb**\_**composite**\_**driver** **webcam**\_**driver** = {  .name = "g\_webcam",  .dev = &webcam\_device\_descriptor,  .strings = webcam\_device\_strings,  .max\_speed = USB\_SPEED\_SUPER,  .bind = webcam\_bind,  .unbind = webcam\_unbind,  }; |  1. 创建usb\_gadget\_driver驱动   |--driver->gadget\_driver = **composite**\_**driver**\_**template**;  |--gadget\_driver->function = (char \*) driver->name;  |--gadget\_driver->driver.name = driver->name;  |--gadget\_driver->max\_speed = driver->max\_speed;   |  | | --- | | static const struct **usb**\_**gadget**\_**driver** **composite**\_**driver**\_**template** = {  .bind = composite\_bind,  .unbind = composite\_unbind,  .setup = composite\_setup,  .disconnect = composite\_disconnect,  .suspend = composite\_suspend,  .resume = composite\_resume,  .driver = {  .owner = THIS\_MODULE,  },  }; |  1. 注册usb\_gadget\_driver驱动到udc\_list链表中   |--**usb**\_**gadget**\_**probe**\_**driver**(gadget\_driver);  |--list\_for\_each\_entry(udc, &**udc**\_**list**, list)  |--**udc**\_**bind**\_**to**\_**driver**(udc, driver); //usb\_udc + usb\_gadget\_driver   1. 绑定usb\_gadget驱动和usb\_gadget\_driver驱动   |--driver->bind(udc->gadget, driver); //composite\_bind()   1. 启动gadget设备   |--**usb**\_**gadget**\_**udc**\_**start**(udc->gadget, driver); //usb\_gadget + usb\_gadget\_driver  |--gadget->ops->udc\_start(gadget, driver); |
| **composite**\_**bind**()   1. 创建usb\_composite\_dev设备   |--struct **usb**\_**composite**\_**dev** cdev = **kzalloc**(sizeof \*cdev, GFP\_KERNEL);  |--cdev->gadget = gadget;   1. 创建usb\_composite\_dev设备的request buffer   |--composite\_dev\_prepare(composite, cdev); //usb\_composite\_driver + usb\_composite\_dev  |--cdev->req = usb\_ep\_alloc\_request(gadget->ep0, GFP\_KERNEL);  |--cdev->req->buf = kmalloc(USB\_COMP\_EP0\_BUFSIZ, GFP\_KERNEL);  |--cdev->req->complete = **composite**\_**setup**\_**complete**;  |--gadget->ep0->driver\_data = cdev;  |--cdev->driver = composite;   1. 绑定usb\_composite\_dev驱动和usb\_composite\_driver驱动   |--composite->bind(cdev); //webcam\_bind() |
| **webcam**\_**bind**(struct usb\_composite\_dev cdev)   1. 创建usb设备的配置文件: webcam\_config\_bind()   |--**usb**\_**add**\_**config**(cdev, &webcam\_config\_driver, **webcam**\_**config**\_**bind**);  |--usb\_add\_config\_only(cdev, config); //usb\_composite\_dev + usb\_configuration  |--list\_for\_each\_entry(c, &cdev->configs, list)  |--config->cdev = cdev;  |--list\_add\_tail(&config->list, &cdev->configs);  |--config->next\_interface\_id = 0;  |--bind(config); //webcam\_config\_bind()  |--usb\_ep\_autoconfig\_reset(cdev->gadget);  |--list\_for\_each\_entry (ep, &gadget->ep\_list, ep\_list);  |--ep->driver\_data = NULL;  |--gadget->in\_epnum = 0;  |--gadget->out\_epnum = 0;  |--usb\_composite\_overwrite\_options(cdev, &coverwrite); |
| **webcam**\_**config**\_**bind**(struct usb\_configuration \*c)  |--**uvc**\_**bind**\_**config**(c, uvc\_fs\_control\_cls, uvc\_ss\_control\_cls,uvc\_fs\_streaming\_cls, uvc\_hs\_streaming\_cls,uvc\_ss\_streaming\_cls); |
| **uvc**\_**bind**\_**config**(c, uvc\_fs\_control\_cls, uvc\_ss\_control\_cls, uvc\_fs\_streaming\_cls, uvc\_hs\_streaming\_cls, uvc\_ss\_streaming\_cls);    1. 创建uvc\_device,初始化uvc\_device的描述符  |--struct **uvc**\_**device** \*uvc = **kzalloc**(sizeof(\*uvc), GFP\_KERNEL);  |--uvc->state = UVC\_STATE\_DISCONNECTED;  |--uvc->desc.fs\_control = fs\_control;  |--uvc->desc.ss\_control = ss\_control;  |--uvc->desc.fs\_streaming = fs\_streaming;  |--uvc->desc.hs\_streaming = hs\_streaming;  |--uvc->desc.ss\_streaming = ss\_streaming;  2. 创建usb\_function  |--uvc->func.name = "uvc";  |--uvc->func.bind = uvc\_function\_bind;  |--**usb**\_**add**\_**function**(c, &uvc->func); //usb\_configuration + usb\_function  |--function->config = config;  |--list\_add\_tail(&function->list, &config->functions);  3. 绑定usb\_configuration和usb\_function  |--value = function->bind(config, function); //uvc\_function\_bind() |
| **uvc**\_**function**\_**bind**() //usb\_configuration + usb\_function  1. 根据uvc\_control\_ep分配dwcs的匹配端点作为gadget的control端点  |--ep = **usb**\_**ep**\_**autoconfig**(cdev->gadget, &uvc\_control\_ep); //usb\_gadget + usb\_endpoint\_descriptor  |--usb\_ep\_autoconfig\_ss(gadget, desc, NULL); //usb\_gadget + usb\_endpoint\_descriptor + usb\_ss\_ep\_comp\_descriptor  |--list\_for\_each\_entry (ep, &gadget->ep\_list, ep\_list)  |--**ep**\_**matches**(gadget, ep, desc, ep\_comp); //usb\_gadget + usb\_ep + usb\_endpoint\_descriptor + usb\_ss\_ep\_comp\_descriptor  |--uvc->control\_ep = ep;  2. 根据uvc\_ss\_streaming\_ep分配dwcs的匹配端点作为gadget的stream端点  |--usb\_ep\_autoconfig\_ss(cdev->gadget, &uvc\_ss\_streaming\_ep, &uvc\_ss\_streaming\_comp);  |--list\_for\_each\_entry (ep, &gadget->ep\_list, ep\_list)  |--**ep**\_**matches**(gadget, ep, desc, ep\_comp); //usb\_gadget + usb\_ep + usb\_endpoint\_descriptor + usb\_ss\_ep\_comp\_descriptor  |--uvc->video.ep = ep;  3. 创建gadget的所有的描述符  |--f->ss\_descriptors = **uvc**\_**copy**\_**descriptors**(uvc, USB\_SPEED\_SUPER);  4. 创建gadget的control传输，回调函数为uvc\_function\_ep0\_complete  |--uvc->control\_req = usb\_ep\_alloc\_request(cdev->gadget->ep0, GFP\_KERNEL);  |--uvc->control\_buf = kmalloc(UVC\_MAX\_REQUEST\_SIZE, GFP\_KERNEL);  |--uvc->control\_req->buf = uvc->control\_buf;  |--uvc->control\_req->complete = **uvc**\_**function**\_**ep0**\_**complete**;  |--uvc->control\_req->context = uvc;  5. 设置usb function的状态为deactivate  |--usb\_function\_deactivate(f); //usb\_function  |--struct usb\_composite\_dev \*cdev = function->config->cdev;  |--if (cdev->deactivations == 0) status = usb\_gadget\_disconnect(cdev->gadget);  |--gadget->ops->pullup(gadget, 0);  |--if (status == 0) cdev->deactivations++;  6. 创建uvc\_video  |--**uvc**\_**video**\_**init**(&uvc->video);  |--video->fcc = V4L2\_PIX\_FMT\_YUYV;  |--uvc\_queue\_init(&video->queue, V4L2\_BUF\_TYPE\_VIDEO\_OUTPUT); //uvc\_video\_queue  |--queue->queue.type = type; //V4L2\_BUF\_TYPE\_VIDEO\_OUTPUT  |--queue->queue.io\_modes = VB2\_MMAP | VB2\_USERPTR;  |--queue->queue.ops = &**uvc**\_**queue**\_**qops**;  |--queue->queue.mem\_ops = &vb2\_vmalloc\_memops;  |--vb2\_queue\_init(&queue->queue);   |  | | --- | | static struct **vb2**\_**ops** **uvc**\_**queue**\_**qops** = {  .queue\_setup = uvc\_queue\_setup,  .buf\_prepare = uvc\_buffer\_prepare,  .buf\_queue = uvc\_buffer\_queue,  }; |   7. 创建并注册video\_device  |--**uvc**\_**register**\_**video**(uvc);  |--struct **video**\_**device** \*video = video\_device\_alloc();  |--video->fops = &**uvc**\_**v4l2**\_**fops**;  |--video->release = video\_device\_release;  |--strlcpy(video->name, cdev->gadget->name, sizeof(video->name));  |--uvc->vdev = video;  |--**video**\_**register**\_**device**(video, VFL\_TYPE\_GRABBER, -1);   |  | | --- | | static struct **v4l2**\_**file**\_**operations** **uvc**\_**v4l2**\_**fops** = {  .owner = THIS\_MODULE,  .open = uvc\_v4l2\_open,  .release = uvc\_v4l2\_release,  .ioctl = uvc\_v4l2\_ioctl,  #ifdef CONFIG\_COMPAT  .compat\_ioctl32 = uvc\_v4l2\_ioctl,  #endif  .mmap = uvc\_v4l2\_mmap,  .poll = uvc\_v4l2\_poll,  #ifndef CONFIG\_MMU  .get\_unmapped\_area = uvc\_v4l2\_get\_unmapped\_area,  #endif  }; | |

# USB Monitor：切换dwc3的模式（device 🡨🡪 host）

## monitor\_handle\_plug\_in\_out\_msg:

|  |
| --- |
| static void monitor\_handle\_plug\_in\_out\_msg(char \*usb\_con\_msg)  {  if (!strncmp(usb\_con\_msg, "**USB\_B\_IN**", 8)) {  my\_mon->**dwc3\_set\_plugstate**(PLUGSTATE\_B\_IN); //s\_dwc3\_set\_plugstate()  my\_mon->xhci\_set\_plugstate(PLUGSTATE\_B\_IN); // do nothing  } else if (!strncmp(usb\_con\_msg, "**USB\_B\_OUT**", 9)) {  my\_mon->xhci\_set\_plugstate(PLUGSTATE\_B\_OUT);  my\_mon->dwc3\_set\_plugstate(PLUGSTATE\_B\_OUT);  } else if (!strncmp(usb\_con\_msg, "**USB\_A\_IN**", 8)) {  my\_mon->dwc3\_set\_plugstate(PLUGSTATE\_A\_IN);  my\_mon->xhci\_set\_plugstate(PLUGSTATE\_A\_IN);  } else if (!strncmp(usb\_con\_msg, "**USB\_A\_OUT**", 9)) {  my\_mon->xhci\_set\_plugstate(PLUGSTATE\_A\_OUT);  my\_mon->dwc3\_set\_plugstate(PLUGSTATE\_A\_OUT);  }  } |

## s\_dwc3\_set\_plugstate

|  |
| --- |
| s\_dwc3\_set\_plugstate(int s)  |--**dwc3\_set\_plugstate**(s);  |--\_\_dwc3\_set\_plugstate(&\_dwc\_plug,s);  |--if((s==PLUGSTATE\_A\_OUT)||(s==PLUGSTATE\_B\_OUT))  |--dwc3\_**plug**\_**out**(dwc\_plug,s);  |--if(s == PLUGSTATE\_**B\_OUT**)  1. USB device out:  |--**dwc3\_gadget\_plugout**(dwc\_plug);  |--if(s == PLUGSTATE\_**A\_OUT**)  2. USB host out:  |--**dwc3\_host\_exit**(dwc);  |--else if((s==PLUGSTATE\_A\_IN)||(s==PLUGSTATE\_B\_IN))  |--dwc3\_**plug**\_**in**(dwc\_plug,s);  |--dwc3\_core\_init(dwc); //更新dwc3的DWC3\_DCTL和DWC3\_GCTL  |--dwc3\_event\_buffers\_setup(dwc); //更新dwc3的event寄存器  |--if(s == PLUGSTATE\_**A\_IN**)  |--dwc3\_set\_host\_in\_auto\_retry(); //更新dwc3的DWC3\_GUCTL  |--dwc3\_set\_mode(dwc, DWC3\_GCTL\_PRTCAP\_HOST); //更新DWC3\_GCTL  3. USB host in:  |--**dwc3\_host\_init**(dwc);  |--else if(s == PLUGSTATE\_**B\_IN)**  |--dwc3\_set\_mode(dwc, DWC3\_GCTL\_PRTCAP\_DEVICE); //更新DWC3\_GCTL  4. USB device in：  |--**dwc3\_gadget\_plugin**(dwc\_plug); |

## dwc3\_gadget\_plugin: device mode enable

|  |
| --- |
| **dwc3\_gadget\_plugin**(struct dwc3\_plug \*dwc\_plug)  |--dwc3\_gadget\_plugin\_init(dwc); //DWC3\_DCFG |= DWC3\_DCFG\_LPM\_CAP  |--gadget->ops->**udc\_start**(gadget, dwc\_plug->dwc\_gadget\_driver); //参见dwc3\_gadget\_start,  |--dwc3\_gadget\_plug\_pullup(gadget, 1);  |--dwc3\_gadget\_run\_stop(dwc, is\_on); //is\_on = 1 //配置DWC3\_DCTL |

## dwc3\_gadget\_plugout: device mode disable

|  |
| --- |
| **dwc3\_gadget\_plugou**t(struct dwc3\_plug \*dwc\_plug)  |--dwc3\_gadget\_plug\_pullup(&dwc->gadget, 0);  |--dwc3\_gadget\_plug\_disconnect(dwc);  |--dwc3\_gadget\_run\_stop(dwc, is\_on); //is\_on = 0 //配置DWC3\_DCTL  |--gadget->ops->**udc**\_**stop**(&dwc->gadget, dwc->gadget\_driver); |

## dwc3\_host\_init: host mode enable

|  |
| --- |
| int **dwc3\_host\_init**(struct dwc3 \*dwc)  1. 创建xhci设备  struct platform\_device \*xhci = platform\_device\_alloc("xhci-hcd", PLATFORM\_DEVID\_AUTO);  dwc->xhci = xhci;  2. 注册xhci设备，触发驱动名为"xhci-hcd"的驱动xhci\_plat\_probe()  platform\_device\_add(xhci); |

## dwc3\_host\_exit:: host mode disable

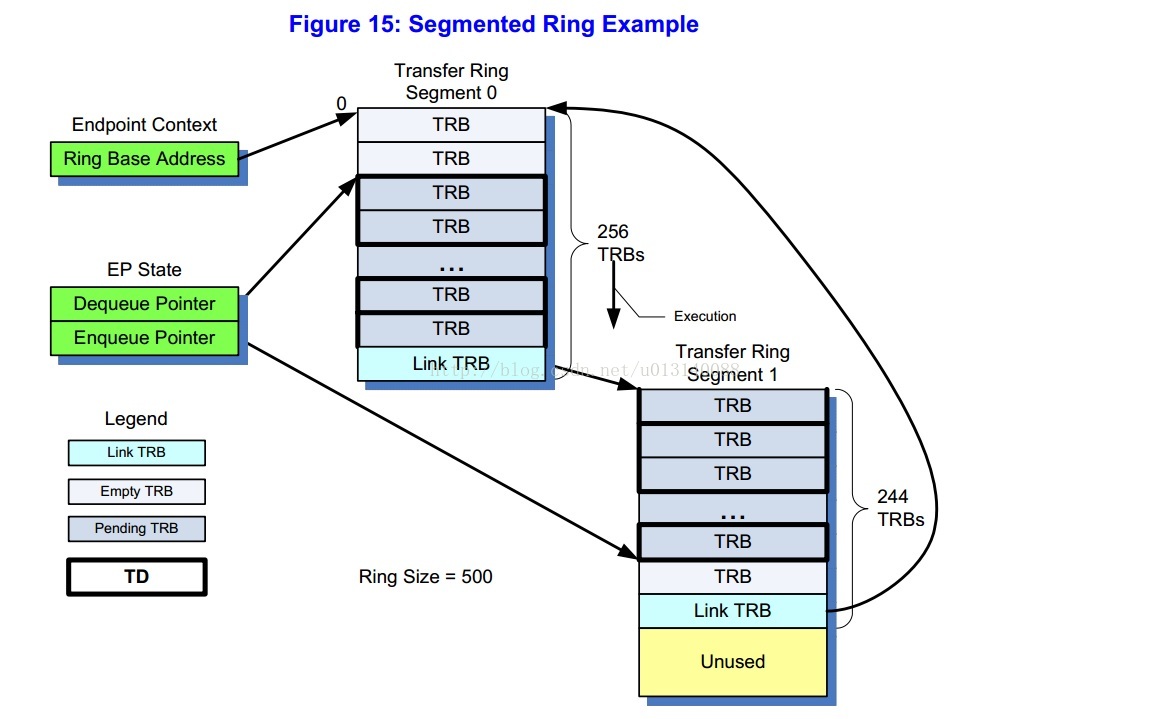
|  |
| --- |
| void **dwc3\_host\_exit**(struct dwc3 \*dwc)  |--platform\_device\_**unregister**(dwc->**xhci**); |

# UDC\_APP

|  |
| --- |
|  |
|  |
|  |
|  |
|  |
|  |
|  |
|  |

# 关键点解释

## XHCI RING: 循环buffer



USB XHCI HOST有三种RING

分别为COMMAND, EVENT 与TRANSFER

这里所说的是TRANSFER RING

一个RING可能由一个或者多个SEGMENT构成

每一个SEGMENT则有多个TRB构成

多个SEGMENT的链接是由LINK TRB来实现的，LINK就像一个单链表，最后一个LINK TRB指向第一个SEGMENT，由此，形成了一个环，RING

TRANSFER TRB有以下几种类型：

针对USB CONTROL TRANSFER的SETUP, DATA, STATUS TRBs

针对BULK, INTERRUPT, ISO TRANSFER的NORMAL TRB

针对ISO TRANSFER的ISOCH TRB (ISO TD由一个ISOCH TRB与0个或多个NORMAL TRB构成)

SW将需要硬件完成的USB传输，通过TRB的形式，将信息提交给硬件，放入RING当中，放入的位置为当前ENQUEUE PTR的位置，每放一个，ENQUEUE PTR向前跨一步，遇到LINK TRB，则跳转到LINK TRB指向的位置

而硬件则按DEQUEUE PTR指向的位置，取出TRB到CACHE当中，执行该TRB，同样，每执行一个，则ADVANCE 该 DEQUEUE PTR，遇LINK TRB，跳转。

TD表示一个USB TRANSFER（不同于USB TRANSACTION）

在TRB当中，有一个CH BIT，如果一处TD由多个TRB构成，则软件需要将除最后一个TRB的所有CH BIT置位。

# End