华清远见 FARSIGHT 始于2004 | 嵌入式培训专家

年口碑积累，成功培养50000多名研发工程师，铸就专业品牌形象

华清远见的企业理念是不仅要做良心教育、做专业教育，更要做受人尊敬的职业教育。

《Linux 那些事儿之我是 USB》

作者：华清远见

专业始于专注 卓识源于远见

第 3 章 Linux 那些事儿之我是 UHCI

专业始于专注 卓识源于远见

1、引子

UHCI，Universal Host Controller Interface，是一种 USB 主机控制器的接口规范，。它是 Intel 公司提出来的江湖中把遵守它的硬件称为 UHCI 主机控制器。在 Linux 中，把这种硬件叫做 HC，或者说 Host Controller，而把与它对应的软件叫做 HCD，即 HC Driver。Linux 中的 HCD 所对应的模块叫做 uhci-hcd。

当我们看一个模块时，首先是看 Kconfig 和 Makefile 文件。在 drivers/usb/host/Kconfig 文件中：

```
161 config USB_UHCI_HCD
162     tristate "UHCI HCD (most Intel and VIA) support"
163     depends on USB && PCI
164     ---help---
165       The Universal Host Controller Interface is a standard by Intel for
166       accessing the USB hardware in the PC (which is also called the USB
167       host controller). If your USB host controller confor ms to this
168       standard, you may want to say Y, but see below. All recent boards
169       with Intel PCI chipsets (like intel 430TX, 440FX, 440LX, 440BX,
170       i810, i820) conform to this standard. Also all VIA PCI chipsets
171      (like VIA VP2, VP3, MVP3, Apollo Pro, Apollo Pro II or Apollo Pro
172       133). If unsure, say Y.
173
174       To compile this driver as a module, choose M here: the
175       module will be called uhci-hcd.
```

请注意第 163 句 depends on USB && PCI，意即这个选项依赖于另外两个选项：CONFIG_USB 和 CONFIG_PCI。很显然这两个选项分别代表着 Linux 中 USB 和 PCI 的核心代码。

UHCI 作为 USB 主机控制器的接口，依赖于 USB 核心，很正常，但为何它也依赖于 PCI 核心代码呢？理由很简单，UHCI 主机控制器本身通常是 PCI 设备，即通常它会插在 PCI 插槽里，或者直接就集成在主板上。但总之，大多数 UHCI 主机控制器是连在 PCI 总线上的。所以，写 UHCI 驱动程序就不得不了解一点 PCI 设备驱动程序。

先用 lspci 命令看一下：

```
localhost:/usr/src/linux-2.6.22.1/drivers/usb/host # lspci | grep USB
00:1d.0 USB Controller: Intel Corporation Enterprise Southbridge UHCI USB #1 (rev 09)
00:1d.1 USB Controller: Intel Corporation Enterprise Southbridge UHCI USB #2 (rev 09)
00:1d.2 USB Controller: Intel Corporation Enterprise Southbridge UHCI USB #3 (rev 09)
00:1d.7 USB Controller: Intel Corporation Enterprise Southbridge EHCI USB (rev 09)
```

比如在我的计算机里，就有三个 UHCI 主机控制器，以及另一个主机控制器和 EHCI 主机控制器。它们都是 PCI 设备。

接着来看 Makefile：

```
localhost:/usr/src/linux-2.6.22.1/drivers/usb/host # cat Makefile
#
# Makefile for USB Host Controller Drivers
#

ifeq ($(CONFIG_USB_DEBUG),y)
       EXTRA_CFLAGS          += -DDEBUG
endif

obj-$(CONFIG_PCI)             += pci-quirks.o

obj-$(CONFIG_USB_EHCI_HCD)      += ehci-hcd.o
obj-$(CONFIG_USB_ISP116X_HCD)   += isp116x-hcd.o
obj-$(CONFIG_USB_OHCI_HCD)      += ohci-hcd.o
obj-$(CONFIG_USB_UHCI_HCD)      += uhci-hcd.o
obj-$(CONFIG_USB_SL811_HCD)     += sl811-hcd.o
obj-$(CONFIG_USB_SL811_CS)      += sl811_cs.o
obj-$(CONFIG_USB_U132_HCD)      += u132-hcd.o
```

很显然，我们要的就是与 CONFIG_USB_UHCI_HCD 对应的 uhci-hcd.o 模块。而与 uhci-hcd.o 最相关的就是与之同名的 C 文件。这是它的源文件。在 drivers/usb/host/uhci-hcd.c 的最后 7 行，可以看到：

```
969 module_init(uhci_hcd_init);
970 module_exit(uhci_hcd_cleanup);
971
```

```
MODULE_AUTHOR(DRIVER_AUTHOR);
MODULE_DESCRIPTION(DRIVER_DESC);
MODULE_LICENSE("GPL");
```

正如每个女人都应该有一支口红一样，每个模块都应该有两个宏： module_init 和 module_exit，分别用来初始化和注销自己。而这两行代码的意思就是说 uhci_hcd_init 函数将会在加载这个模块时被调用，uhci_hcd_cleanup 则是将会在卸载这个模块时被执行。

所以，只能从 uhci_hcd_init 开始我们的故事，

```
static int __init uhci_hcd_init(void)
{
 int retval = -ENOMEM;

      printk(KERN_INFO DRIVER_DESC " " DRIVER_VERSION "%s\n",
                     ignore_oc ? ", overcurrent ignored" : "");

      if (usb_disabled())
          return -ENODEV;

      if (DEBUG_CONFIGURED) {
          errbuf = kmalloc(ERRBUF_LEN, GFP_KERNEL);
          if (!errbuf)
              goto errbuf_failed;
          uhci_debugfs_root = debugfs_create_dir("uhci", NULL);
          if (!uhci_debugfs_root)
              goto debug_failed;
      }

      uhci_up_cachep = kmem_cache_create("uhci_urb_priv",
              sizeof(struct urb_priv), 0, 0, NULL, NULL);
      if (!uhci_up_cachep)
          goto up_failed;

      retval = pci_register_driver(&uhci_pci_driver);
      if (retval)
          goto init_failed;

      return 0;

init_failed:
      kmem_cache_destroy(uhci_up_cachep);

up_failed:
      debugfs_remove(uhci_debugfs_root);

debug_failed:
      kfree(errbuf);

errbuf_failed:

      return retval;
}
```

2、开户和销户

之所以说 uhci_hcd_init 有技术含量，并不是说它包含多么精巧的算法，包含多么复杂的数据结构，而是因为这其中涉及了很多东西。首先 924 行，usb_disable 涉及了 Linux 中的内核参数的概念，928 行的 kmalloc 和 936 行的 kmem_cache_create 涉及了 Linux 内核中内存申请的问题，931 行 debugfs_create_dir 则涉及了，一个虚拟的文件系统 debugfs，而 941 行 pci_register_driver 则涉及 Linux 中 PCI 设备驱动程序的注册。

这么多东西往这里一堆，其复杂程度立马就上来了。

内核参数，什么是内核参数？看一下 grub 文件：

```
    title SUSE Linux Enterprise Server 10 (kdb enabled)
       kernel (hd0,2)/boot/vmlinuz-2.6.22.1-test root=/dev/hda3 resume=/dev/hda2  splash=silent
showopts
       initrd /boot/initrd-2.6.22.1-test
```

kernel 那行都是内核参数，比如 root， resume， splash， showopts。其中“root”=代表的是 root 文件系统的位置，“resume”=代表的是用于 software suspend 恢复的分区。而 USB 子系统也准备了 nousb 参数。所以如果往这一行后面加上 nousb，则意味着系统不需要支持 USB，即把 USB 子系统给“disable”掉了。换句话说，usb_disabled 返回的就是 nousb 的值。在 drivers/usb/core/usb.c 中也能看到这个函数：

```
852 /*
853  * for external read access to <nousb>
854  */
855 int usb_disabled(void)
856 {
857         return nousb;
858 }
```

申请内存，用来申请内存的两个函数分别是 kmalloc 和 kmem_cache_create。应该很熟悉 kmalloc。而 kmem_cache_create 则是传说中的 slab“现身”了。传统上，kmem_cache_create 是 slab 分配器的接口函数，用于创建一个“内存池”cache 创建了一个 cache 之后，就可以用另一个函数 kmem_cache_zalloc 来申请内存，使用 kmem_cache_free 来释放内存。可以使用 kmem_cache_destroy 来彻底释放这个内存池。

这里重是每次用 kmem_cache_zalloc 申请内存的大小是一样的，即在 kmem_cache_create 中的第二个参数所指定的，比如 sizeof(struct urb_priv)，即以后用 kmem_cache_zalloc 申请的内存总是这么大，而这里 kmem_cache_create 的返回值就是创建好的那个 cache。这里返回值被赋给了 uhci_up_cachep。它是一个 struct kmem_cache 的结构体指针。所以以后用 kmem_cache_zalloc 时只要把 uhci_up_cachep 作为参数即可得到想要的内存。对于 kmem_cache_free 和 kmem_cache_destroy 也一样。

理解这些函数最简单的比喻就是，去沃尔玛超市购物，超市里给你提供了篮子，你可以把你需要的东西装在篮子里，但大家都知道超市里的篮子数量是有限的，但是虽然超市篮子不够的情况存在，但是沃尔玛在开张之前肯定会准备了足够多的篮子，比如它订做了一个仓库的篮子，每个篮子都一样大。即一开始沃尔玛方就调用了 kmem_cache_create 做了一池的篮子，而你每次去就是使用 kmem_cache_zalloc 去拿一个篮子即可，而当你付款之后你要离开了，你又调用 kmem_cache_free 去归还篮子，每个人都这样做的话，你下次去了要用篮子又可以用 kmem_cache_zalloc 再拿一个。

而一旦哪天沃尔玛宣武门分店连续亏损，店子倒闭，它就可以调用 kmem_cache_destroy 把篮子全都毁掉，当然更形象的例子是它把篮子转移到别的店去，比如知春路分店。那么从整个沃尔玛公司来看，可以供来装东西的容器总容积还是没有变，正如你的计算机总的内存是不会变的。假如公司将来又打算在国贸开一家分店，那么它可以再次调用 kmem_cache_create。而对你来说，你并不需要知道一池内存到底有多少，就像你永远不用知道沃尔玛知春路店究竟有多少个篮子一样。

调试信息

好了，第三，debugfs_create_dir，传说中的 debugfs 也“现身了”。很多事情都是早已注定的，原本以为写设备驱动的只要懂一些硬件规范就可以了，后来终于在眼泪中明白，有些人一旦写代码就会越写越复杂。如今的内核代码早已不那么单纯了。

以前我一直以为，Linux 中 PCI 子系统和 USB 子系统的掌门人 Greg 同志只是一个花拳绣腿的家伙，只是每天忙着到处演讲、开会，而不干正经事。后来我发现，其实不是的，Greg 其实还是干了很多有意义的事情，不得不承认，Greg 是条汉子！debugfs 就是他开发的一个虚拟的文件系统，专门用于输出调试信息。这个文件系统默认是被挂载在/sys/kernel/debug 下的，比如：

```
localhost:~ # mount
/dev/hda3 on / type reiserfs (rw,acl,user_xattr)
proc on /proc type proc (rw)
sysfs on /sys type sysfs (rw)
debugfs on /sys/kernel/debug type debugfs (rw)
udev on /dev type tmpfs (rw)
devpts on /dev/pts type devpts (rw,mode=0620,gid=5)
```

这个文件系统是专门为开发人员准备的，在配置内核时可以编译也可以不编译进去。其对应的 Kconfig 文件是 lib/Kconfig.debug：

```
50 config DEBUG_FS
```

```
51   bool "Debug Filesystem"
52   depends on SYSFS
53   help
54     debugfs is a virtual file system that kernel developers use to put
55     debugging files into.  Enable this option to be able to read and
56     write to these files.
57
58     If unsure, say N.
```

很有意思的是，这个文件系统居然依赖于另一个文件系统——sysfs。如果用 make menuconfig 命令编译内核，则在 Kernel hacking 下面找到它，即 DEBUG_FS。我们不去深入研究这个文件系统，但是对于它的接口函数是有必要了解一下的。

首先，debugfs_create_dir 就是创建一个目录。像这里这么一行的作用就是在/sys/kernel/debug 下面创建一个叫做 uhci 的目录，比如加载了 uhci-hcd 这个模块，就能看到 uhci 目录：

```
localhost:/usr/src/linux-2.6.22.1 # ls /sys/kernel/debug/
kprobes  uhci
```

这个函数的返回值是文件系统里最经典的一个 struct dentry 结构体指针。而这里我们把返回值赋给了 struct dentry 指针 uhci_debugfs_root。它被定义在 drivers/usb/host/uhci-debug.c 中：

```
20 static struct dentry *uhci_debugfs_root;
```

显然这个指针对 uhci-hcd 模块来说是到处可以引用的。且可用 debugfs_create_file 函数在 uhci 目录下创建文件。这在 uhci_start 函数中也会介绍。而以后删除这个目录的任务就在 uhci_hcd_cleanup 中，它只要调用 debugfs_remove 函数即可。

注册 PCI

现在剩下第四个问题，pci_register_driver，其实一路走来应该多少有感觉，虽然没见过这个函数，但是能感觉出它和 usb_register_driver 注册 USB 驱动作用相同，注册 PCI 驱动。请注意参数 uhci_pci_driver。

```
894 static const struct pci_device_id uhci_pci_ids[] = { {
895       /* handle any USB UHCI controller */
896       PCI_DEVICE_CLASS(PCI_CLASS_SERIAL_USB_UHCI, ~0),
897       .driver_data =  (unsigned long) &uhci_driver,
898       }, { /* end: all zeroes */ }
899 };
900
901 MODULE_DEVICE_TABLE(pci, uhci_pci_ids);
902
903 static struct pci_driver uhci_pci_driver = {
904       .name =         (char *)hcd_name,
905       .id_table =     uhci_pci_ids,
906
907       .probe =        usb_hcd_pci_probe,
908       .remove =       usb_hcd_pci_remove,
909       .shutdown =     uhci_shutdown,
910
911 #ifdef  CONFIG_PM
912       .suspend =      usb_hcd_pci_suspend,
913       .resume =       usb_hcd_pci_resume,
914 #endif  /* PM */
915 };
```

这里 PCI_CLASS_SERIAL_USB_UHCI 是 0x0c0300，03 表示类别为 03，代表 USB；而 00 代表 UHCI。OHCI 是 0x0c0310，而 EHCI 则是 0x0c0320。而 PCI spec 规定：最前面两位的 0c 代表所有串行总线控制器。

所以不难知道，真正的故事将从哪个函数开始：probe 函数，即 usb_hcd_pci_probe。在讲这个函数之前在本节的最后把 uhci_hcd_cleanup 也给贴出来：

```
961 static void __exit uhci_hcd_cleanup(void)
962 {
963       pci_unregister_driver(&uhci_pci_driver);
964       kmem_cache_destroy(uhci_up_cachep);
965       debugfs_remove(uhci_debugfs_root);
966       kfree(errbuf);
967 }
```

总之，一个模块就是这样，写一个注册函数，再写一个注销函数即可。Linux 中模块机制的便利就是：当想用它时可以用 modprobe 或者 insmod 加载它，当不想用它时，可以用 rmmod 卸载它。加载就是注册，卸载就是注销，就像银行里的开户和销户，但至少这种服务你可以随意享受。

3、PCI，我们来了！

神圣的 PCI 设备驱动程序——usb_hcd_pci_probe 即将带领我们开启新的篇章，开始 PCI 世界之旅，也将开始一段全新的体验。

细心的你或许注意到了，关于 HCD 的代码，被分布于两个目录：drivers/usb/core/和 drivers/usb/host/，其中前者包含三个相关的文件：hcd-pci.c，hcd.c 和 hcd.h，这是一些公共的代码。因为 USB 主机控制器有很多种，光从接口来说，目前就有 UHCI，OHCI 和 EHCI，谁知道以后还会不会有更多呢。而这些主机控制器的驱动程序有一些代码是相同的，所以就把它提取出来，专门写在某几个文件中，因此有了这种格局。光就某一种具体的 HCD 代码，还是在 drivers/usb/host/下面，比如与 UHCI 相关的代码就是以下几个文件：

```
localhost:/usr/src/linux-2.6.22.1/drivers/usb/host # ls uhci-*
uhci-debug.c  uhci-hcd.c  uhci-hcd.h  uhci-hub.c  uhci-q.c
```

就 UHCI 的驱动来说，其四大函数指针 probe/remove/suspend/resume 都是指向一些公共函数，都定义于 drivers/usb/core/hcd-pci.c 中。只有一个 shutdown 指针指向的函数 uhci_shutdown 是它自己定义的来自 drivers/usb/host/uhci-hcd.c 中。

probe 函数，即 usb_hcd_pci_probe 来自 drivers/usb/core/hcd-pci.c：

```
58 int usb_hcd_pci_probe (struct pci_dev *dev, const struct pci_device_id *id)
59 {
60      struct hc_driver       *driver;
61      struct usb_hcd         *hcd;
62      int                    retval;
63
64      if (usb_disabled())
65          return -ENODEV;
66
67      if (!id || !(driver = (struct hc_driver *) id->driver_data))
68          return -EINVAL;
69
70      if (pci_enable_device (dev) < 0)
71          return -ENODEV;
72      dev->current_state = PCI_D0;
73      dev->dev.power.power_state = PMSG_ON;
74
75      if (!dev->irq) {
76          dev_err (&dev->dev,
77                  "Found HC with no IRQ.  Check BIOS/PCI %s setup!\n",
78                  pci_name(dev));
79          retval = -ENODEV;
80          goto err1;
81      }
82
83      hcd = usb_create_hcd (driver, &dev->dev, pci_name(dev));
84      if (!hcd) {
85          retval = -ENOMEM;
86          goto err1;
87      }
88
89      if (driver->flags & HCD_MEMORY) {      // EHCI, OHCI
90          hcd->rsrc_start = pci_resource_start (dev, 0);
91          hcd->rsrc_len = pci_resource_len (dev, 0);
92          if (!request_mem_region (hcd->rsrc_start, hcd->rsrc_len,
93                                   driver->description)) {
94              dev_dbg (&dev->dev, "controller already in use\n");
95              retval = -EBUSY;
96              goto err2;
97      }
98          hcd->regs = ioremap_nocache (hcd->rsrc_start, hcd->rsrc_len);
99          if (hcd->regs == NULL) {
100             dev_dbg (&dev->dev, "error mapping memory\n");
```

```
101             retval = -EFAULT;
102             goto err3;
103         }
104
105     } else {                            // UHCI
106         int    region;
107
108         for (region = 0; region < PCI_ROM_RESOURCE; region++) {
109             if (!(pci_resource_flags (dev, region) &
110                                 IORESOURCE_IO))
111                 continue;
112
113             hcd->rsrc_start = pci_resource_start (dev, region);
114             hcd->rsrc_len = pci_resource_len (dev, region);
115             if (request_region (hcd->rsrc_start, hcd->rsrc_len,
116                                 driver->description))
117                 break;
118         }
119         if (region == PCI_ROM_RESOURCE) {
120             dev_dbg (&dev->dev, "no i/o regions available\n");
121             retval = -EBUSY;
122             goto err1;
123     }
124     }
125
126     pci_set_master (dev);
127
128     retval = usb_add_hcd (hcd, dev->irq, IRQF_SHARED);
129     if (retval != 0)
130         goto err4;
131     return retval;
132
133  err4:
134     if (driver->flags & HCD_MEMORY) {
135         iounmap (hcd->regs);
136  err3:
137         release_mem_region (hcd->rsrc_start, hcd->rsrc_len);
138     } else
139         release_region (hcd->rsrc_start, hcd->rsrc_len);
140  err2:
141     usb_put_hcd (hcd);
142  err1:
143     pci_disable_device (dev);
144     dev_err (&dev->dev, "init %s fail, %d\n", pci_name(dev), retval);
145     return retval;
146 }
```

PCI 设备驱动程序肯定要比 USB 设备驱动程序复杂。其他我不说，光凭这幅经典的 PCI 标准配置寄存器的图就够我们这些新手们研究半天的（如图 3.3.1 所示）。

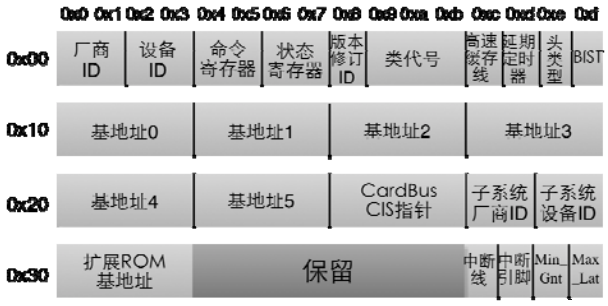

图 3.3.1　PCI 配置寄存器

看明白这张图就算对 PCI 设备驱动有一点认识，看不明白的话就说明还没入门。不过不要慌，我也不懂，让我陪着你一起结合代码来看。不过从此刻开始，这张图将被我们无数次地提起。为了便于称呼，我们给这张图取个好记的名字，就叫“清明上坟图”吧，简称“上坟图”。这张图在整个 PCI 世界里的作用就相当于我们学习化学的教材中最后几页里附上的那个化学元素周期表。写 PCI 设备驱动的人对于这张图的熟悉程度就要达到我们当时那种随口就能喊出“氢氦锂铍硼碳氮氧氟氖钠镁铝硅磷硫氯氩钾钙”的境界。

70 行，pci_enable_device()，在使用一个 PCI 设备之前，必须调用 pci_enable_device 激活它，该函数会调用底层代码激活 PCI 设备上的 I/O 资源和内存资源。而 143 行那个 pci_disable_device 则恰恰是做一些与之相反的事情。任何一个 PCI 设备驱动程序都会调用这两个函数。只有在激活了设备之后，驱动程序才可以访问它的资源。

72 行，73 行，这里的 dev 是 struct pci_dev 结构体指针，它有一个成员：pci_power_t current_state，用来记录该设备的当前电源状态，这个世界上除了我们熟知的 PCI spec 以外，还有一个规范叫做 PCI Power Management spec，它专门为 PCI 设备定义那些电源管理方面的接口。按这个规范，PCI 设备一共可以有四种电源状态：D0，D1，D2，D3。正常的工作状态就是 D0，而 D3 是耗电最少的状态，也就意味着设备“Power off”了。在 include/linux/pci.h 中有关于这些状态的定义：

```
74 #define PCI_D0        ((pci_power_t __force) 0)
75 #define PCI_D1        ((pci_power_t __force) 1)
76 #define PCI_D2        ((pci_power_t __force) 2)
77 #define PCI_D3hot     ((pci_power_t __force) 3)
78 #define PCI_D3cold    ((pci_power_t __force) 4)
79 #define PCI_UNKNOWN   ((pci_power_t __force) 5)
80 #define PCI_POWER_ERROR ((pci_power_t __force) -1)
```

继续看 usb_hcd_pci_probe()的 75 行，dev->irq，struct pci_dev 有这么一个成员：unsigned int irq。这个意思很明显，中断号，它来自哪里？好，让我们第一次说一下这张“清明上坟图”了，每一个 PCI 设备都有一堆寄存器，厂商就是按着这张图来设计自己的设备。这张图里全都是寄存器，但是并非所有设备都拥有全部寄存器，其中有些是必选的，有些是可选的，就好比我们大学里面的必修课和选修课。比如，厂商 ID、设备 ID 和类代号这就是必选的，它们就用来标志一个设备，而很多厂商也是会利用子系统厂商 ID 和子系统设备 ID 的，因为可以进一步地细分设备。

仔细数一数，这张图里一共是 64 个字节。而其中倒数第 4 个字节，即 byte 60，记录的正是该设备可以使用的中断号。在系统初始化时这个值就已经被写进去了，所以对于写设备驱动的人来说，不需要考虑太多。这就是 dev->irq 这行的意思。USB 主机控制器必须有中断号，否则没法正常工作。

接下来，usb_create_hcd()，才是正式进入 HCD 的概念。这个函数来自 drivers/usb/core/hcd.c：

```
1493 struct usb_hcd *usb_create_hcd (const struct hc_driver *driver,
1494                 struct device *dev, char *bus_name)
1495 {
1496     struct usb_hcd *hcd;
1497
1498     hcd = kzalloc(sizeof(*hcd) + driver->hcd_priv_size, FP_KERNEL);
1499     if (!hcd) {
1500         dev_dbg (dev, "hcd alloc failed\n");
1501         return NULL;
1502     }
1503     dev_set_drvdata(dev, hcd);
1504     kref_init(&hcd->kref);
1505
1506     usb_bus_init(&hcd->self);
1507     hcd->self.controller = dev;
1508     hcd->self.bus_name = bus_name;
1509     hcd->self.uses_dma = (dev->dma_mask != NULL);
1510
1511     init_timer(&hcd->rh_timer);
1512     hcd->rh_timer.function = rh_timer_func;
1513     hcd->rh_timer.data = (unsigned long) hcd;
1514 #ifdef CONFIG_PM
1515     INIT_WORK(&hcd->wakeup_work, hcd_resume_work);
1516 #endif
1517
1518     hcd->driver = driver;
1519     hcd->product_desc = (driver->product_desc) ? driver->product_desc :
1520                     "USB Host Controller";
1521
1522     return hcd;
1523 }
```

usb_create_hcd()的第一个参数 struct hc_driver，这个结构体掀开了我们对 USB 主机控制器驱动的认识，它来自 drivers/usb/core/hcd.h：

```
149 struct hc_driver {
150     const char     *description;   /* "ehci-hcd" etc */
151     const char     *product_desc;  /* product/vendor string */
152     size_t         hcd_priv_size;  /* size of private data */
153
154     /* irq handler */
155     irqreturn_t    (*irq) (struct usb_hcd *hcd);
156
157     int    flags;
158 #define HCD_MEMORY     0x0001       /* HC regs use memory (else I/O) */
159 #define HCD_USB11      0x0010       /* USB 1.1 */
160 #define HCD_USB2       0x0020       /* USB 2.0 */
161
162     /* called to init HCD and root hub */
163     int    (*reset) (struct usb_hcd *hcd);
164     int    (*start) (struct usb_hcd *hcd);
165
166     /* NOTE:  these suspend/resume calls relate to the HC as
167      * a whole, not just the root hub; they're for PCI bus glue.
168      */
169     /* called after suspending the hub, before entering D3 etc */
170     int    (*suspend) (struct usb_hcd *hcd, pm_message_t message);
171
172     /* called after entering D0 (etc), before resuming the hub */
173     int    (*resume) (struct usb_hcd *hcd);
174
175     /* cleanly make HCD stop writing memory and doing I/O */
176     void   (*stop) (struct usb_hcd *hcd);
177
178     /* shutdown HCD */
179     void   (*shutdown) (struct usb_hcd *hcd);
180
181     /* return current frame number */
182     int    (*get_frame_number) (struct usb_hcd *hcd);
183
184     /* manage i/o requests, device state */
185     int    (*urb_enqueue) (struct usb_hcd *hcd,
186                                struct usb_host_endpoint *ep,
187                                struct urb *urb,
188                                gfp_t mem_flags);
189     int    (*urb_dequeue) (struct usb_hcd *hcd, struct urb *urb);
190
191     /* hw synch, freeing endpoint resources that urb_dequeue can't */
192     void   (*endpoint_disable)(struct usb_hcd *hcd,
193                     struct usb_host_endpoint *ep);
194
195     /* root hub support */
196     int           (*hub_status_data) (struct usb_hcd *hcd, char *buf);
197     int           (*hub_control) (struct usb_hcd *hcd,
198                             u16 typeReq, u16 wValue, u16 wIndex,
199                             char *buf, u16 wLength);
200     int           (*bus_suspend)(struct usb_hcd *);
201     int           (*bus_resume)(struct usb_hcd *);
202     int       (*start_port_reset)(struct usb_hcd *, unsigned port_num);
203     void          (*hub_irq_enable)(struct usb_hcd *);
204     /* Needed only if port-change IRQs are level-triggered */
205 };
```

说句良心话，你说这么长的一个结构体，要我怎么看？现在制药的都知道要制良心药，你们这些写代码的就不能写良心代码？反正吧，每个 HCD 都得对应这么一个结构体变量。比如 UHCI，在 drivers/usb/host/uhci-hcd 中就有这么一段：

```
862 static const char hcd_name[] = "uhci_hcd";
863
864 static const struct hc_driver uhci_driver = {
865     .description =        hcd_name,
866     .product_desc =       "UHCI Host Controller",
867     .hcd_priv_size =      sizeof(struct uhci_hcd),
868
869     /* Generic hardware linkage */
870     .irq =                uhci_irq,
871     .flags =              HCD_USB11,
872
873     /* Basic lifecycle operations */
```

```
874     .reset =             uhci_init,
875     .start =             uhci_start,
876 #ifdef CONFIG_PM
877     .suspend =           uhci_suspend,
878     .resume =            uhci_resume,
879     .bus_suspend =       uhci_rh_suspend,
880     .bus_resume =        uhci_rh_resume,
881 #endif
882     .stop =              uhci_stop,
883
884     .urb_enqueue =       uhci_urb_enqueue,
885     .urb_dequeue =       uhci_urb_dequeue,
886
887     .endpoint_disable =  uhci_hcd_endpoint_disable,
888     .get_frame_number =  uhci_hcd_get_frame_number,
889
890     .hub_status_data =   uhci_hub_status_data,
891     .hub_control =       uhci_hub_control,
892 };
```

其实就是一堆指针，也没什么了不起。不过我得提醒你了，在咱们整个故事中也就只有一个 struct hc_driver 变量：uhci_driver。以后凡是提到 hc_driver，指的就是 uhci_driver。probe 函数是 PCI 那边的接口，而 hc_driver 是 USB 这边的接口，这两概念咋扯到一块去了呢？呵呵，usb_hcd_pci_probe 函数 67 行，看见没有，driver 在这里被赋值了，而等号右边那个 id->driver_data 又是什么？继续回去看，在 uhci_pci_ids 这张表里写得很清楚，driver_data 就是被赋值为&uhci_driver，所以说一切都是有因才有果的，不会无缘无故地出现一个变量。

继续看 usb_create_hcd()，1496 行，一个变态的数据结构还不够，还得来一个更变态的。struct usb_hcd，有一个 HCD 就得有这么一个结构体，也来自 drivers/usb/core/hcd.h：

```
58 struct usb_hcd {
59
60      /*
61       * housekeeping
62       */
63      struct usb_bus        self;         /* hcd is-a bus */
64      struct kref           kref;         /* reference counter */
65
66      const char            *product_desc;  /* product/vendor string */
67      char                  irq_descr[24];  /* driver + bus # */
68
69      struct timer_list      rh_timer;     /* drives root-hub polling */
70      struct urb            *status_urb;   /* the current status urb */
71 #ifdef CONFIG_PM
72      struct work_struct     wakeup_work;   /* for remote wakeup */
73 #endif
74
75      /*
76       * hardware info/state
77       */
78      const struct hc_driver  *driver;      /* hw-specific hooks */
79
80      /* Flags that need to be manipulated atomically */
81      unsigned long         flags;
82 #define HCD_FLAG_HW_ACCESSIBLE  0x00000001
83 #define HCD_FLAG_SAW_IRQ       0x00000002
84
85      unsigned              rh_registered:1;/* is root hub registered? */
86
87      /* The next flag is a stopgap, to be removed when all the HCDs
88       * support the new root-hub polling mechanism. */
89      unsigned              uses_new_polling:1;
90      unsigned              poll_rh:1;     /* poll for rh status? */
91      unsigned              poll_pending:1; /* status has changed? */
92      unsigned              wireless:1;    /* Wireless USB HCD */
93
94      int                   irq;          /* irq allocated */
95      void __iomem           *regs;        /* device memory/io */
96      u64                   rsrc_start;    /* memory/io resource start */
97      u64                   rsrc_len;     /* memory/io resource length */
98      unsigned              power_budget;   /* in mA, 0 = no limit */
99
```

```
100 #define HCD_BUFFER_POOLS       4
101     struct dma_pool        *pool [HCD_BUFFER_POOLS];
102
103     int                 state;
104 #define  __ACTIVE              0x01
105 #define  __SUSPEND             0x04
106 #define  __TRANSIENT           0x80
107
108 #define  HC_STATE_HALT          0
109 #define  HC_STATE_RUNNING       (__ACTIVE)
110 #define  HC_STATE_QUIESCING     (__SUSPEND|__TRANSIENT|__ACTIVE)
111 #define  HC_STATE_RESUMING      (__SUSPEND|__TRANSIENT)
112 #define  HC_STATE_SUSPENDED     (__SUSPEND)
113
114 #define HC_IS_RUNNING(state) ((state) & __ACTIVE)
115 #define HC_IS_SUSPENDED(state) ((state) & __SUSPEND)
116
117     /* more shared queuing code would be good; it should support
118      * smarter scheduling, handle transaction translators, etc;
119      * input size of periodic table to an interrupt scheduler.
120      * (ohci 32, uhci 1024, ehci 256/512/1024).
121      */
122
123     /* The HC driver's private data is stored at the end of
124      * this structure.
125      */
126     unsigned long hcd_priv[0]
127                 __attribute__ ((aligned (sizeof(unsigned long))));
128 };
```

所以 usb_create_hcd 函数就是用来为 struct usb_hcd 申请内存空间的，并且初始化。我们来看它具体如何初始化的。

1498 行，申请内存，并且初值为 0。

接下来得注意了，usb_create_hcd 中的 dev 可是 struct device 结构体指针，而刚才的 usb_hcd_pci_probe 中的 dev 是 struct pci_dev 结构体指针，struct pci_dev 表示的就是一个 PCI 设备，它有一个成员 struct device dev，所以实际上在调用 usb_create_hcd 时第二个参数是&dev->dev。而这里 1503 行的 dev_set_drvdata 就是一简单的内联函数，来自 include/linux/device.h：

```
491 static inline void
492 dev_set_drvdata (struct device *dev, void *data)
493 {
494     dev->driver_data = data;
495 }
```

这个结构体中有一个成员 void *driver_data，其效果就是令 dev->driver_data 等于申请好的 hcd。

而 1504 行，初始化一个引用计数，可以看到 struct usb_hcd 有一个成员 struct kref kref，即引用计数的变量。

1506 行，struct usb_hcd 中有一个成员 struct usb_bus self，一个主机控制器就意味着一条总线，所以这里又出来另一个结构体：struct usb_bus。

```
276 struct usb_bus {
277     struct device *controller;     /* host/master side hardware */
278     int busnum;                   /* Bus number (in order of reg) */
279     char *bus_name;                /* stable id (PCI slot_name etc) */
280     u8 uses_dma;                  /* Does the host controller use DMA? */
281     u8 otg_port;                   /* 0, or number of OTG/HNP port */
282     unsigned is_b_host:1;         /* true during some HNP roleswitches */
283     unsigned b_hnp_enable:1;       /* OTG: did A-Host enable HNP? */
284
285     int devnum_next;              /* Next open device number in
286                                    * round-robin allocation */
287
288     struct usb_devmap devmap;     /* device address allocation map */
289     struct usb_device *root_hub;   /* Root hub */
290     struct list_head bus_list;     /* list of busses */
291
292     int bandwidth_allocated;       /* on this bus: how much of the time
293                                    * reserved for periodic (intr/iso)
```

```
294                                     * requests is used, on average?
295                                     * Units: microseconds/frame.
296                                     * Limits: Full/low speed reserve 90%,
297                                     * while high speed reserves 80%.
298                                     */
299     int bandwidth_int_reqs;          /* number of Interrupt requests */
300     int bandwidth_isoc_reqs;         /* number of Isoc. requests */
301
302 #ifdef CONFIG_USB_DEVICEFS
303     struct dentry *usbfs_dentry; /* usbfs dentry entry for the bus */
304 #endif
305     struct class_device *class_dev; /* class device for this bus */
306
307 #if defined(CONFIG_USB_MON)
308     struct mon_bus *mon_bus;          /* non-null when associated */
309     int monitored;                   /* non-zero when monitored */
310 #endif
311 };
```

4、I/O 内存和 I/O 端口

usb_bus_init 来自 drivers/usb/core/hcd.c，很显然，它就是初始化 struct usb_bus 结构体指针。而这个结构体变量 hcd->self 的内存已经在刚才为 HCD 申请内存时一并申请了。

```
695 static void usb_bus_init (struct usb_bus *bus)
696 {
697     me mset (&bus->devmap, 0, sizeof(struct usb_devmap));
698
699     bus->devnum_next = 1;
700
701     bus->root_hub = NULL;
702     bus->busnum = -1;
703     bus->bandwidth_allocated = 0;
704     bus->bandwidth_int_reqs  = 0;
705     bus->bandwidth_isoc_reqs = 0;
706
707     INIT_LIST_HEAD (&bus->bus_list);
708 }
```

当初在 Hub 驱动中就讲过，devnum_next 在总线初始化时会被设为 1，说的就是这里。

回到 usb_create_hcd 中来，又是几行赋值。

倒是 1511 行引起了我的注意，又是可恶的时间机制：init_timer，这个函数我们也见过多次了，斑驳的陌生终于在时间的抚摸下变成了今日的熟悉。这里设置的函数是 rh_timer_func，而传递给这个函数的参数是 hcd。

1515 行，INIT_WORK 也在 Hub 驱动里见过了，这里 hcd_resume_work 什么时候会被调用也到时候再看。

剩下两行赋值。1518 行，struct usb_hcd 有一个 struct hc_driver 的结构体指针成员，所以就这样把它和咱们这个 uhci_driver 给联系起来了。而在 uhci_driver 中可看到，其中有一个 product_desc 被赋值为“UHCI Host Controller”，所以这里也赋给 hcd->product_desc，因为 struct hc_driver 和 struct usb_hcd 这两个结构体中都有一个成员 const char *product_desc。

至此，usb_create_hcd 结束了，返回了申请好赋好值的 hcd。我们继续回到 probe 函数中来。

89 到 124 这个 if-else 的确让我开心，因为 if 里说的是 EHCI 和 OHCI 的情况，else 里针对的才是 UHCI，鉴于 EHCI 将由某人来写，而 OHCI 和 UHCI 性质一样，我们不会讲，所以这就意味着这个 if-else 只要从 105 行开始看，即直接看到 UHCI 那部分的代码。爽！

不过也得知道这里为何要判断 HCD_MEMORY，这个宏是表明该 HC 的寄存器是使用 Memory 的，而没有设置 flag 的 HC 的寄存器是使用 I/O 的。这些寄存器俗称 I/O 端口（I/O ports），这个 I/O 端口可以被映射在 Memory Space，也可以被映射在 I/O Space。UHCI 是属于后者，而 EHCI/OHCI 属于前者。

这里看上去必须多说几句，否则很难说清楚。以我们家 Intel 为代表的 i386 系列处理器中，内存和外部 I/O 是独立编址、独立寻址的，于是有一个地址空间叫做内存空间，另有一个地址空间叫做 I/O 空间。也就是说，从处理器的角度来说，i386 提供了一些单独的指令用来访问 I/O 空间。换言之，访问 I/O 空间和访问普通的内存得使用不同的指令。而在一些嵌入式的处理器中，比如 PowerPC 只使用一个空间，即内存空间。那像这种情况，外设的 I/O 端口的物理地址就被映射到内存地址空间中，这就是内存映射（Memory-mapped）。而外设的 I/O 端口的物理地址就被映射到 I/O 地址空间中，这就是 I/O-mapped，即 I/O 映射。

那么 EHCI/OHCI 除了有寄存器以外，还有内存。而它们把这些统统映射到内存空间中去，而 UHCI 只使用寄存器来通信，所以它只需要映射寄存器，即 I/O 端口。而 spec 规定，它是映射到 I/O 空间的。Linux 中 I/O 内存和 I/O 端口都被视为一种资源，分别被记录在/proc/iomem 和/proc/ioports 中。

所以可以在这里看到 uhci-hcd：

```
localhost:~ # cat /proc/ioports
0000-001f : dma1
0020-0021 : pic1
……
bca0-bcbf : 0000:00:1d.2
  bca0-bcbf : uhci_hcd
bcc0-bcdf : 0000:00:1d.1
  bcc0-bcdf : uhci_hcd
bce0-bcff : 0000:00:1d.0
  bce0-bcff : uhci_hcd
c000-cfff : PCI Bus #10
  cc00-ccff : 0000:10:0d.0
d000-dfff : PCI Bus #0e
  dcc0-dcdf : 0000:0e:00.1
    dcc0-dcdf : e1000
  dce0-dcff : 0000:0e:00.0
    dce0-dcff : e1000
e000-efff : PCI Bus #0c
  e800-e8ff : 0000:0c:00.1
    e800-e8ff : qla2xxx
  ec00-ecff : 0000:0c:00.0
    ec00-ecff : qla2xxx
fc00-fc0f : 0000:00:1f.1
  fc00-fc07 : ide0
```

而在这里看到 ehci-hcd：

```
localhost:~ # cat /proc/iomem
00000000-0009ffff : System RAM
  00000000-00000000 : Crash kernel
……
d8000000-d80fffff : PCI Bus #01
  d8000000-d80fffff : PCI Bus #02
    d80f0000-d80fffff : 0000:02:0e.0
      d80f0000-d80fffff : megasas: LSI Logic
d8100000-d81fffff : PCI Bus #0c
  d8100000-d813ffff : 0000:0c:00.1
e0000000-efffffff : reserved
f2000000-f7ffffff : PCI Bus #06
  f4000000-f7ffffff : PCI Bus #07
    f4000000-f7ffffff : PCI Bus #08
      f4000000-f7ffffff : PCI Bus #09
        f4000000-f5ffffff : 0000:09:00.0
          f4000000-f5ffffff : bnx2
f8000000-fbffffff : PCI Bus #04
  f8000000-fbffffff : PCI Bus #05
    f8000000-f9ffffff : 0000:05:00.0
      f8000000-f9ffffff : bnx2
……
fca00400-fca007ff : 0000:00:1d.7
  fca00400-fca007ff : ehci_hcd
fe000000-ffffffff : reserved
100000000-22fffffff : System RAM
```

使用 I/O 内存要先申请，再映射。使用 I/O 端口也要先申请，也可称为请求，即让内核知道你要访问这个端口，这样内核知道了以后它就不会再让别人也访问这个端口了。申请 I/O 端口的函数是 request_region，这个函数来自 include/linux/ioport.h：

```
116 /* Convenience shorthand with allocation */
117 #define request_region(start,n,name)  \
            __request_region(&ioport_resource, (start), (n), (name))
118 #define request_mem_region(start,n,name)  \
            __request_region(&iomem_resource, (start), (n), (name))
119 #define rename_region(region, newname)  \
            do { (region)->name = (newname); } while (0)
120
121 extern struct resource * __request_region(struct resource *,
122                         resource_size_t start,
123                             resource_size_t n, const char *name);
```

这里看到的 request_mem_region 用来申请 I/O 内存。申请了之后，还需要使用 ioremap 或者 ioremap_nocache 函数来映射。

request_region 的三个参数 start，n，name 意味着你想使用从 start 开始的 size 为 n 的 I/O port 资源，name 自然就是你的名字了。这三个概念在的 cat /proc/ioports 里面显示得很清楚，name 就是 uhci-hcd。

那么对于 uhci-hcd，我们究竟需要请求哪些地址，需要多少空间呢？嗯，又要提到那张上坟图了。PCI 设备本身有一堆的地址空间、内存空间和 I/O 空间。那么用什么把这些空间映射到总线上来呢？寄存器。每个设备都有 6 个地址空间，这叫做 6 个基址寄存器，有的设备还有一个 ROM，所以又有一个 Expansion ROM Base Address，它对应第 7 个区间，或者说区间 6，而在“上坟图”上对应的就叫做扩展 ROM 基址寄存器。每个寄存器都是 4 个字节。而在 include/linux/pci.h 中定义了：

```
235 #define PCI_ROM_RESOURCE        6
```

所以看到循环条件就是从 0 到 PCI_ROM_RESOURCE 之前，即循环 6 次，因为有 6 个区间，区间也叫 region。那么这些寄存器究竟取的什么值呢？这就是在 PCI 总线初始化时做的事情了，它会把每个基址寄存器赋上值，而实际上就是映射于总线上的地址，总线驱动的作用就是让各个设备需要的地址资源都得到满足，并且设备与设备之间的地址不发生冲突。PCI 总线驱动做了这些之后，我们 PCI 设备驱动就简单了，在需要使用时直接请求即可，正如这里的 request_region。那么传递给 request_region 的具体参数是什么呢？

两个函数，pci_resource_start 和 pci_resource_len，去获得一个区间的起始地址和长度，所以就很好理解这段代码了。至于 109 行这个 if 判断，pci_resource_flags 是用来判断一个资源是哪种类型，include/linux/ioport.h 中一共定义了 4 种资源：

```
36 #define IORESOURCE_IO           0x00000100      /* Resource type */
37 #define IORESOURCE_MEM          0x00000200
38 #define IORESOURCE_IRQ          0x00000400
39 #define IORESOURCE_DMA          0x00000800
```

它们是 I/O、内存、中断和 DMA。对应在/proc 下的 ioports，iomem，interrupt 和 dma 4 个文件。所以这里就判断如果不是 I/O 端口资源，那么就不予理睬。因为 UHCI 主机控制器只需要理睬 I/O 端口。

request_region 函数执行成功将返回非 NULL，否则返回 NULL。所以一旦成功就跳出循环。反之，如果循环都结束了还未能请求到，那就说明出错了。那么你说为何一旦成功就跳出循环？老实说，这个问题足足困扰了我 13 秒钟，别小看 13 秒钟，有这么长时间刘翔都已经完成一次 110 米跨栏了。让 spec 来告诉你。

看到没有，20~23h，4 个字节，这里正好对应 UHCI 的 I/O 空间基址寄存器。换言之，UHCI 就定义了一个基址寄存器，所以只要使用一个基址寄存器就可以映射所需要的地址了。所以，成功一次就可以结束循环了。

又一次回到 usb_hcd_pci_probe 中，126 行，pci_set_master 函数。还是看那张“上坟图”，注意到第三个寄存器，即命令寄存器。让我们用 PCI spec 来告诉你这个命令寄存器的格局：

看到 Bus Master 了么？没错，就是那个 Bit 2。用毛德操先生的话说就是“PCI 设备要进行 DMA 操作就得具有竞争成为总线主的能力”。而这个 Bus Master 位就是用来打开或关闭 PCI 设备竞争成为总线主的能力的。在完成 PCI 总线的初始化时，所有 PCI 设备的 DMA 功能都是关闭的，所以这里要调用 pci_set_master 启用 USB 主机控制器竞争成为总线主的能力。就是说 PCI 设备有没有这种能力是可以设置的。

USB spec 2.0 中 10.2.9 节在讲到 USB Host Interface 时，说了 USB HC 是应该具备这种能力的：“The Host Controller provides a high-speed bus-mastering interface to and from main system memory. The physical transfer between memory and the USB wire is performed automatically by the Host Controller.”

同时在 UHCI spec 里面我们也能找到这么一句话：“For the implementation example in this document, the Host Controller is a PCI device. PCI Bus master capability in the Host Controller permits high performance data transfers to system memory.”

不过，究竟什么是 PCI 的 Bus Master？连接到 PCI 总线上的设备有两种：主控设备和目标设备，即 master 设备和 target-only 设备。这两者最直接的区别就是 target-only 最少需要 47 根 pin，而 master 最少需要 49 根 pin，它们所必须支持的总线信号就是不一样的。

PCI 设备如果以 target-only 的方式工作，那么它就完全是在主机的 CPU 的控制之下工作，比如设备接收到某一个外部事件，中断主机，主机 CPU 读写设备，这样设备就可以工作了。这样的设备也被少数人称为 slave 设备，或者说从设备。这就是典型的奴才型的设备，主说什么就是什么，完全没有自己的见解。而 master 设备就比这个要复杂了，master 设备能够不在主机 CPU 的干预下访问主机的地址空间，包括主存和其他 PCI 设备。很显然，DMA 就属于这种情况，即不需要主机 CPU 干涉的情况下 USB 主机控制器通过 DMA 直接读写内存。所以需要启用这种能力。

不过 PCI 总线在同一时刻只能供一对设备完成传输。至于有了竞争力能不能竞争得到 Master，那就得看人品了。上天给了你一幅天使的面孔和魔鬼的身材，但你能不能成为明星就得看造化了，当然只要你遵守圈中的潜规则，你离成功就不远了。

5、传说中的 DMA

下一个函数，usb_add_hcd，定义在 drivers/usb/core/hcd.c 中:

```
1558 int usb_add_hcd(struct usb_hcd *hcd,
1559                unsigned int irqnum, unsigned long irqflags)
1560 {
1561     int retval;
1562     struct usb_device *rhdev;
1563
1564     dev_info(hcd->self.controller, "%s\n", hcd->product_desc);
1565
1566     set_bit(HCD_FLAG_HW_ACCESSIBLE, &hcd->flags);
1567
1568     /* HC is in reset state, but accessible.  Now do the one-time init,
1569      * bottom up so that hcds can customize the root hubs before khubd
1570      * starts talking to them.  (Note, bus id is assigned early too.)
1571      */
1572     if ((retval = hcd_buffer_create(hcd)) != 0) {
1573         dev_dbg(hcd->self.controller, "pool alloc failed\n");
1574         return retval;
1575     }
1576
1577     if ((retval = usb_register_bus(&hcd->self)) < 0)
1578         goto err_register_bus;
1579
1580     if ((rhdev = usb_alloc_dev(NULL, &hcd->self, 0)) == NULL) {
1581         dev_err(hcd->self.controller, "unable to allocate root hub\n");
1582         retval = -ENOMEM;
1583         goto err_allocate_root_hub;
1584     }
1585     rhdev->speed = (hcd->driver->flags & HCD_USB2) ? USB_SPEED_HIGH :
1586                     USB_SPEED_FULL;
1587     hcd->self.root_hub = rhdev;
```

```
1588
1589    /* wakeup flag init defaults to "everything works" for root hubs,
1590     * but drivers can override it in reset() if needed, along with
1591     * recording the overall controller's system wakeup capability.
1592     */
1593    device_init_wakeup(&rhdev->dev, 1);
1594
1595    /* "reset" is misnamed; its role is now one-time init. the controller
1596     * should already have been reset (and boot firmware kicked off etc).
1597     */
1598    if (hcd->driver->reset && (retval = hcd->driver->reset(hcd)) < 0) {
1599        dev_err(hcd->self.controller, "can't setup\n");
1600        goto err_hcd_driver_setup;
1601    }
1602
1603    /* NOTE: root hub and controller capabilities may not be the same */
1604    if (device_can_wakeup(hcd->self.controller)
1605                && device_can_wakeup(&hcd->self.root_hub->dev))
1606        dev_dbg(hcd->self.controller, "supports USB remote wakeup\n");
1607
1608    /* enable irqs just before we start the controller */
1609    if (hcd->driver->irq) {
1610        snprintf(hcd->irq_descr, sizeof(hcd->irq_descr), "%s:usb%d",
1611                hcd->driver->description, hcd->self.busnum);
1612        if ((retval = request_irq(irqnum, &usb_hcd_irq, irqflags,
1613                        hcd->irq_descr, hcd)) != 0) {
1614            dev_err(hcd->self.controller,
1615                        "request interrupt %d failed\n", irqnum);
1616            goto err_request_irq;
1617        }
1618        hcd->irq = irqnum;
1619        dev_info(hcd->self.controller, "irq %d, %s 0x%08llx\n", irqnum,
1620                        (hcd->driver->flags & HCD_MEMORY) ?
1621                        "io mem" : "io base",
1622                        (unsigned long long)hcd->rsrc_start);
1623    } else {
1624        hcd->irq = -1;
1625        if (hcd->rsrc_start)
1626        dev_info(hcd->self.controller, "%s 0x%08llx\n",
1627                        (hcd->driver->flags & HCD_MEMORY) ?
1628                        "io mem" : "io base",
1629                    (unsigned long long)hcd->rsrc_start);
1630    }
1631
1632    if ((retval = hcd->driver->start(hcd)) < 0) {
1633        dev_err(hcd->self.controller, "startup error %d\n", retval);
1634        goto err_hcd_driver_start;
1635    }
1636
1637    /* starting here, usbcore will pay attention to this root hub */
1638    rhdev->bus_mA = min(500u, hcd->power_budget);
1639    if ((retval = register_root_hub(hcd)) != 0)
1640        goto err_register_root_hub;
1641
1642    if (hcd->uses_new_polling && hcd->poll_rh)
1643        usb_hcd_poll_rh_status(hcd);
1644    return retval;
1645
1646 err_register_root_hub:
1647    hcd->driver->stop(hcd);
1648 err_hcd_driver_start:
1649    if (hcd->irq >= 0)
1650        free_irq(irqnum, hcd);
1651 err_request_irq:
1652 err_hcd_driver_setup:
1653    hcd->self.root_hub = NULL;
1654    usb_put_dev(rhdev);
1655 err_allocate_root_hub:
1656    usb_deregister_bus(&hcd->self);
1657 err_register_bus:
1658    hcd_buffer_destroy(hcd);
1659    return retval;
1660 }
```

1566 行，设置一个 flag，至于作用，等遇到了再说。

1572 行，hcd_buffer_create，初始化一个 buffer 池。现在是时候说一说 DMA 了。我们知道一个 USB 主机控制器控制着一条 USB 总线，而 USB 主机控制器的一项重要工作是在内存和 USB 总线之间传输数据。这个过程可以使用 DMA 也可以不使用 DMA，不使用 DMA 的方式即所谓的 PIO 方式。DMA 代表着 Direct Memory Access，即直接内存访问。即不需要 CPU 干预，比如有一个 UHCI 控制器，我告诉它，内存中某个地方放了一堆数据，你去取吧，然后它就自己去取，取完了它就跟我说一声，告诉我它取完了。

那么在整个 USB 子系统中是如何处理这些事情的呢？好，苦等了这么久，我终于有机会来向你解释这个问题了，现在我终于可以说电视剧中男主角对女主角常说的那句话，“你听我解释，你听我解释呀！”在 Hub 驱动中，调用过一个函数 usb_buffer_alloc，当时很多网友问我这个函数究竟是如何处理 DMA 或不 DMA 的？

关于 DMA，合理的做法是先创建一个内存池，然后每次都从池子里要内存。具体说就是先由 HCD 这边建池子，然后设备驱动就直接索取。来看一下来自 drivers/usb/core/buffer.c 中的 hcd_buffer_create：

```
52 int hcd_buffer_create(struct usb_hcd *hcd)
53 {
54      char          name[16];
55      int           i, size;
56
57      if (!hcd->self.controller->dma_mask)
58          return 0;
59
60      for (i = 0; i < HCD_BUFFER_POOLS; i++) {
61          if (!(size = pool_max [i]))
62              continue;
63          snprintf(name, sizeof name, "buffer-%d", size);
64          hcd->pool[i] = dma_pool_create(name, hcd->self.controller,
65                          size, size, 0);
66          if (!hcd->pool [i]) {
67              hcd_buffer_destroy(hcd);
68              return -ENOMEM;
69          }
70      }
71      return 0;
72 }
```

先看 64 行，调用 dma_pool_create 函数，这个函数是真正去创建内存池的函数，或者更准确地讲，创建一个 DMA 池，内核中定义了一个结构体：struct dma_pool 来专门代表一个 DMA 池。而这个函数的返回值就是生成的那个 DMA 池。如果创建失败就调用 hcd_buffer_destroy，还是来自同一个文件：

```
82 void hcd_buffer_destroy(struct usb_hcd *hcd)
83 {
84      int           i;
85
86      for (i = 0; i < HCD_BUFFER_POOLS; i++) {
87          struct dma_pool *pool = hcd->pool[i];
88          if (pool) {
89              dma_pool_destroy(pool);
90              hcd->pool[i] = NULL;
91          }
92      }
93 }
```

看得出这里调用的是 dma_pool_destroy，其作用不言自明。

那么我们知道了创建池子和销毁池子的函数，如何从池子里索取或者把索取的释放回去呢？这对应的两个函数分别是，dma_pool_alloc 和 dma_pool_free，而这两个函数正是与我们说的 usb_buffer_alloc 以及 usb_buffer_free 相联系的。于是来看这两个函数的代码，来自 drivers/usb/core/usb.c：

```
589 void *usb_buffer_alloc(
590     struct usb_device *dev,
591     size_t size,
592     gfp_t mem_flags,
593     dma_addr_t *dma
594 )
595 {
596     if (!dev || !dev->bus)
597         return NULL;
```

```
598     return hcd_buffer_alloc(dev->bus, size, mem_flags, dma);
599 }
612 void usb_buffer_free(
613     struct usb_device *dev,
614     size_t size,
615     void *addr,
616     dma_addr_t dma
617 )
618 {
619     if (!dev || !dev->bus)
620         return;
621     if (!addr)
622         return;
623     hcd_buffer_free(dev->bus, size, addr, dma);
624 }
```

很显然，它们调用的就是 hcd_buffer_alloc 和 hcd_buffer_free，于是进一步跟踪，来自 drivers/usb/core/buffer.c：

```
100 void *hcd_buffer_alloc(
101     struct usb_bus  *bus,
102     size_t              size,
103     gfp_t               mem_flags,
104     dma_addr_t          *dma
105 )
106 {
107     struct usb_hcd        *hcd = bus_to_hcd(bus);
108     int                 i;
110     /* some USB hosts just use PIO */
111     if (!bus->controller->dma_mask) {
112         *dma = ~(dma_addr_t) 0;
113         return kmalloc(size, mem_flags);
114     }
116     for (i = 0; i < HCD_BUFFER_POOLS; i++) {
117         if (size <= pool_max [i])
118             return dma_pool_alloc(hcd->pool [i], mem_flags, dma);
119     }
120     return dma_alloc_coherent(hcd->self.controller, size, dma, 0);
121 }
123 void hcd_buffer_free(
124     struct usb_bus  *bus,
125     size_t              size,
126     void                *addr,
127     dma_addr_t          dma
128 )
129 {
130     struct usb_hcd        *hcd = bus_to_hcd(bus);
131     int                 i;
133     if (!addr)
134         return;
136     if (!bus->controller->dma_mask) {
137         kfree(addr);
138         return;
139     }
141     for (i = 0; i < HCD_BUFFER_POOLS; i++) {
142         if (size <= pool_max [i]) {
143             dma_pool_free(hcd->pool [i], addr, dma);
144             return;
145         }
146     }
147     dma_free_coherent(hcd->self.controller, size, addr, dma);
148 }
```

看见了吧，最终调用的就是 dma_pool_alloc 和 dma_pool_free。那么主机控制器到底支不支持 DMA 操作呢？看见上面这个 dma_mask 了么？默认情况下，dma_mask 在总线枚举时被函数 pci_scan_device 中设置为了 0xffffffff。struct device 结构体有一个成员 u64 *dma_mask，如果一个 PCI 设备不能支持 DMA，那么应该在 probe 函数中调用 pci_set_dma_mask 把这个 dma_mask 设置为 NULL。不过一个没有精神分裂症的 PCI 设备通常是支持 DMA 的。这个掩码更多的作用是，比如你的设备只能支持 24 位的寻址，那你就得通过设置 dma_mask 来告诉 PCI 层，就需要把 dma_mask 设置为 0x00ffffff。因为标准的 PCI 设备都是 32 位寻址的，所以标准情况就是设置的 0xffffffff。不过开发人员们的建议是不要直接使用这些数字，而是使用它们定义在 include/linux/dma-mapping.h 中的这些宏：

```
16 #define DMA_64BIT_MASK  0xffffffffffffffffULL
17 #define DMA_48BIT_MASK  0x0000ffffffffffffULL
18 #define DMA_40BIT_MASK  0x000000ffffffffffULL
19 #define DMA_39BIT_MASK  0x0000007fffffffffULL
20 #define DMA_32BIT_MASK  0x00000000ffffffffULL
21 #define DMA_31BIT_MASK  0x000000007fffffffULL
22 #define DMA_30BIT_MASK  0x000000003fffffffULL
23 #define DMA_29BIT_MASK  0x000000001fffffffULL
24 #define DMA_28BIT_MASK  0x000000000fffffffULL
25 #define DMA_24BIT_MASK  0x0000000000ffffffULL
```

不过目前在 drivers/usb/目录下面没有哪个驱动会调用 pci_set_dma_mask，因为现代总线上的大部分设备都能够处理 32 位地址，换句话说大家的设备都还算“正经”，但如果你们家生产出来一个不伦不类的设备，那么你就别忘了在 probe 阶段用 pci_set_dma_mask 设置一下，否则你就甭指望设备能够正确进行 DMA 传输。关于这个函数的使用，可以参考 drivers/net 下面的那些驱动，很多网卡驱动都调用了这个函数，虽然其中很多其实就是设置 32 位。

要不，总结一下？以上这几个 DMA 函数就不细讲了。但需要对某些地方单独拿出来讲。

第一，hcd_buffer_alloc 函数中，111 行，判断：如果 dma_mask 为 NULL，说明这个主机控制器不支持 DMA，那么使用原始的方法申请内存，即 kmalloc。然后申请好了就直接返回，而不会继续去执行下面的 dma_pool_alloc 函数。同样的判断在 hcd_buffer_free 中也是一样的，没有 DMA 的就直接调用 kfree 释放内存，而不需要调用 dma_pool_free 了。

第二，你应该注意到这里还有另外两个函数我们根本没提起：dma_alloc_coherent 和 dma_free_coherent。这两个函数也是用来申请 DMA 内存的，但是它们适合申请比较大的内存，比如 N 个 page 的那种。而 DMA 池的作用本来就是提供给小打小闹式的内存申请的。当前的 USB 子系统里在 drivers/usb/core/buffer.c 中定义了一个数组：

```
26 static const size_t    pool_max [HCD_BUFFER_POOLS] = {
27       /* platfor ms without dma-friendly caches might need to
28        * prevent cacheline sharing...
29        */
30       32,
31       128,
32       512,
33       PAGE_SIZE / 2
34       /* bigger --> allocate pages */
35 };
```

HCD_BUFFER_POOLS 这个宏的值为 4。结合 hcd_buffer_alloc 函数中面那个循环来看，可以知道，你要是申请 32 个字节以内、128 个字节以内、512 个字节以内，或者最多二分之一个 PAGE_SIZE 以内的，就直接使用这个内存池了。否则的话，就得用那个 dma_alloc_coherent 了。释放时也一样。

第三，struct usb_hcd 结构体有这么一个成员，struct dma_pool *pool [HCD_BUFFER_POOLS]，这个数组的值是在 hcd_buffer_create 中赋上的，即当时以这个 pool_max 为模型创建了 4 个池子，所以 hcd_buffer_alloc 里就可以这样用。

第四，至于像 dma_pool_create/dma_pool_alloc/dma_alloc_coherent 这些函数具体怎么实现的我想任何一个写设备驱动程序的都不用关心吧，倒是有人会比较感兴趣，因为他们是研究内存管理的。

Ok，讲完了 hcd_buffer_create，让我们还是回到 usb_add_hcd 中来，继续往下走。

6、来来，我是一条总线，线线线线线线

请注意，下一个函数，1577 行，usb_register_bus()。我们说过，一个 USB 主机控制器就意味着一条 USB 总线，因为主机控制器控制的正是一条总线。古人说，猫走不走直线，完全取决于耗子，而数据走不走总线，完全取决于主机控制器。

所以这里作为主机控制器的驱动，我们必须从软件的角度来说，注册一条总线。来自 drivers/usb/core/hcd.c：

```
720 static int usb_register_bus(struct usb_bus *bus)
721 {
722     int busnum;
724     mutex_lock(&usb_bus_list_lock);
725     busnum = find_next_zero_bit (busmap.busmap, USB_MAXBUS, 1);
726     if (busnum < USB_MAXBUS) {
727         set_bit (busnum, busmap.busmap);
728         bus->busnum = busnum;
729     } else {
730         printk (KERN_ERR "%s: too many buses\n", usbcore_name);
731         mutex_unlock(&usb_bus_list_lock);
732         return -E2BIG;
733     }
735   bus->class_dev=class_device_create(usb_host_class,NULL,MKDEV(0,0),
736                              bus->controller, "usb_host%d", busnum);
737     if (IS_ERR(bus->class_dev)) {
738         clear_bit(busnum, busmap.busmap);
739         mutex_unlock(&usb_bus_list_lock);
740         return PTR_ERR(bus->class_dev);
741     }
743     class_set_devdata(bus->class_dev, bus);
745     /* Add it to the local list of buses */
746     list_add (&bus->bus_list, &usb_bus_list);
747     mutex_unlock(&usb_bus_list_lock);
749     usb_notify_add_bus(bus);
751     dev_info (bus->controller,
          "new USB bus registered, assigned bus number %d\n", bus->busnum);
752     return 0;
753 }
```

Linux 中名字里带一个 register 的函数那是数不胜数，随着你对 Linux 内核渐渐的熟悉，慢慢就会觉得其实叫 register 的函数都很简单。甚至你会发现，Linux 内核中的模块没有不用 register 函数的。

这个函数首先让我们想起了在 Hub 驱动中讲的那个 choose_address。当时有一个 devicemap，而现在有一个 busmap。很显然，原理是一样的。在 drivesr/usb/core/hcd.c 中有定义：

```
89 #define USB_MAXBUS              64
90 struct usb_busmap {
91     unsigned long busmap [USB_MAXBUS / (8*sizeof (unsigned long))];
92 };
93 static struct usb_busmap busmap;
```

和当时我们在 Hub 驱动中对 devicemap 的分析一样，当时的结论是该 map 一共有 128 位，同理可知这里 busmap 则一共有 64 位。也就是说一共可以有 64 条 USB 总线。我想，对我们这些凡夫俗子来说，这么多条足够了吧。

735 行，class_device_create 函数是 Linux 设备模型中一个很基础的函数。Intel 的企业文化中有六大价值观（去 Intel 面试时我特逗的一点就是把那六大价值观给背了下来，然后面试时跟面试官一条一条说，把人家逗乐了），这六大价值观中有一个叫做 Result Orientation，用中文说就是以结果为导向。那现在我想是该使用这一价值观时了，Linux 2.6 设备模型中提供了大把大把的基础函数，它们都在 drivers/base 目录下面，这里的函数你如果有兴趣当然可以看一看，不过我不推荐你这么做，除非你的目的就是要彻底研究这个设备模型是如何实现的。对这些基础函数，我觉得比较好的认识方法就是以结果为导向，查看它们执行之后具体有什么效果，用直观的效果来体现它们的作用。

那么这里 class_device_create 的效果是什么？我们知道设备模型和 sysfs 结合相当紧密，最能反映设备模型的效果的就是 sysfs。所以凭一种男人的直觉，我们应该到/sysfs 下面去看效果。不过我现在更愿意给你提供更多的背景知识。

首先，什么是 class？C++的高手们一定不会不知道 class 吧，虽说我从未写过 C++程序，可是好歹也看过两遍《Thinking in C++》的第一卷，所以 class 还是知道的。class 就是类，设备模型中引入类的意义在于让一些模糊的东西变得更加清晰、直观，比如同样是 SCSI 设备，可能磁盘和磁带，都属于 scsi_device。于是就可以在/sys/class 下面建立一个文件夹，从这里来体现同一个类别的各种设备。比如，某台机器里/sys/class 下面可以看到这些类，此时此刻还没有加载 usbcore：

```
localhost:~ # ls /sys/class/
backlight  graphics  mem   net     spi_master  vc
```

```
dma       input     misc  pci_bus  tty         vtconsole
```

而 class_device_create 的第一个参数是 usb_host_class，它是什么呢？

让我们把镜头切给 usbcore 的初始化函数，usb_init()。我们在 Hub 驱动中已经说过，usb_init 是 Linux 中整个 USB 子系统的起点，一切的一切都从这里开始。而这个函数中有这么一段：

```
877     retval = usb_host_init();
878     if (retval)
879     goto host_init_failed;
```

来看它具体做了什么事情，usb_host_init()来自 drivers/usb/core/hcd.c:

```
671 static struct class *usb_host_class;
672
673 int usb_host_init(void)
674 {
675     int retval = 0;
676
677     usb_host_class = class_create(THIS_MODULE, "usb_host");
678     if (IS_ERR(usb_host_class))
679         retval = PTR_ERR(usb_host_class);
680     return retval;
681 }
```

让我们在上面提到的那台机器机器中加载 usbcore，查看在/sys/class/下面会发生点什么：

```
localhost:~ # modprobe usbcore
localhost:~ # ls /sys/class/
backlight   dma   graphics   input   mem   misc   net   pci_bus   spi_master   tty   usb_device
usb_host  vc  vtconsole
```

看出区别了么？多了两个目录，usb_host 和 usb_device，换言之，多了两个类。所以不难知道，这里 class_create 函数的效果就是在/sys/class/下面创建一个 usb_host 的目录。而 usb_device 是在另一个函数 usb_devio_init()中创建的。创建方法是一样的，均是调用 class_create 函数。

继续看 usb_host，调用 class_create 将返回值赋给了 usb_host_class，而这正是我们传递给 class_device_create 的第一个参数，所以不看代码也应该知道我们的目标是在/sys/class/usb_host/下面建立一个文件或者一个目录，那么结合代码来看就不难发现我们要建立的是一个叫做 usb_hostn 的文件或者是目录，具体是什么，让我们用结果来说话。没有加载 uhci-hcd 时，可以看出这个目录是空的。

```
localhost:~ # ls /sys/class/usb_host/
```

把这个模块加载，再来查看效果：

```
localhost:~ # modprobe uhci-hcd
localhost:~ # ls -l /sys/class/usb_host/
total 0
drwxr-xr-x 2 root root 0 Oct  4 22:26 usb_host1
drwxr-xr-x 2 root root 0 Oct  4 22:26 usb_host2
drwxr-xr-x 2 root root 0 Oct  4 22:26 usb_host3
drwxr-xr-x 2 root root 0 Oct  4 22:26 usb_host4
```

因为这台机器有 4 个 UHCI 主机控制器，所以可以看出，分别为每个主机控制器建立了一个目录。而 usb_host 后面的这个 1，2，3，4 就是刚才说的 busnum，即总线编号，因为一个主机控制器控制着一条总线。

同时我们把 class_device_create 的返回值赋给了 bus->class_dev。struct usb_bus 中有一个成员 struct class_device *class_dev，这个成员被称作 class device。这个结构体对写驱动的人来说意义不大，但是从设备模型的角度来说是必要的，实际上对写驱动的人来说，你完全可以不理睬设备模型中 class 这个部分，可以尽可能少地支持设备模型，因为这对访问设备没有太多影响。甚至可以让设备根本就不在/sysfs 下面体现出来，用代码去支持设备模型，将来你使用设备时就能享受到设备模型为你提供的方便。相反，如果不支持设备模型，那么使用设备时会发现有很多不便。所以 743 行做了这么一个举动，调用 class_set_devdata()，这就算是写代码的人对设备模型的支持，因为 struct class_device 中也有一个成员 void *class_data，被称为 class-specific data，而在 include/linux/device.h 中定义了 class_set_devdata()和一个与之对应的函数 class_get_devdata()。

```
279 static inline void *
```

```
280 class_get_devdata (struct class_device *dev)
281 {
282     return dev->class_data;
283 }
284
285 static inline void
286 class_set_devdata (struct class_device *dev, void *data)
287 {
288     dev->class_data = data;
289 }
```

结合这里具体对这个函数调用的代码可知，最终这个主机控制器对应的 class_device 的 class_data 被赋值为 bus。这样有朝一日我们要通过 class_device 找到对应的 bus 时只要调用 class_get_devdata 即可。设备模型的精髓就在于把一个设备相关联的种种元素都给联系起来，设备模型提供了大量建立这种纽带的函数。我们要做的就是调用这些函数。

好，继续，746 行，很显然又是队列操作。usb_bus_list 是一个全局队列，在 drivers/usb/core/hcd.c 中定义：

```
85 LIST_HEAD (usb_bus_list);
86 EXPORT_SYMBOL_GPL (usb_bus_list);
```

每次注册一条总线就是往这个队列里添加一个元素。struct usb_bus 中有一个成员 struct list_head bus_list。所以这里直接调用 list_add 即可。

用一句话解释 749 行，usb_notify_add_bus，按 Intel 以结果为导向的理论来说，这个函数在此情此景执行的结果是/proc/bus/usb 下面会多一些文件，比如：

```
localhost:~ # ls /proc/bus/usb/
001  002  003  004  devices
```

好了，usb_register_bus 算是看完了，再一次回到 usb_add_hcd 中来，1580 行，usb_alloc_dev 被调用，这个函数我们可不陌生。不过我们下节再看吧。这节剩下的时间我们为 usb_notify_add_bus 再多说两句话，这个函数牵涉到 Linux 中的 notify 机制，这是 Linux 内核中一种常用的事件回调处理机制。传说中的那个“神奇”的内核调试工具 KDB 中就是利用这种机制进入 KDB 的。

这种机制在网络设备驱动中的应用，那就像“成都小吃”在北京满大街都是一样。而在 USB 子系统中，以前并没有使用这种机制，只是 Linux 中 USB 掌门人 Greg 在 2005 年底提出来要加进来的。

7、主机控制器的初始化

好了，usb_alloc_dev，多么熟悉啊，这里做的就是为 Root Hub 申请了一个 struct usb_device 结构体，并且初始化，将返回值赋给指针 rhdev。回顾这个函数可以知道，Root Hub 的 parent 指针指向了主机控制器本身。

1585 行，确定 rhdev 的 speed，UHCI 和 OHCI 都是源于曾经的 USB 1.1，而 EHCI 才是来自 USB 2.0。只有 USB 2.0 才定义了高速的设备，以前的设备只有两种速度、低速和全速。也只有 EHCI 的驱动才定义了一个 flag：HCD_USB 2。所以这里记录的 rhdev->speed 就是 USB_SPEED_FULL。

1593 行，device_init_wakeup，也在 Hub 驱动中见过。第 2 个参数是 1 就意味着把 Root Hub 的 Wakeup 能力打开了。正如注释里说的那样，可以在自己的驱动里面把它关掉。

1598 行，如果 hc_driver 中有 reset 函数，就调用它，uhci_driver 显然是有的。回去看它的定义，可知 uhci_init。所以这时候就要调用这个函数了，显然顾名思义，它的作用是做一些初始化。它来自 drivers/usb/host/uhci-hcd.c：

```
483 static int uhci_init(struct usb_hcd *hcd)
484 {
485     struct uhci_hcd *uhci = hcd_to_uhci(hcd);
486     unsigned io_size = (unsigned) hcd->rsrc_len;
487     int port;
488
489     uhci->io_addr = (unsigned long) hcd->rsrc_start;
```

```
490
491     /* The UHCI spec says devices must have 2 ports, and goes on to say
492      * they may have more but gives no way to determine how many there
493      * are.  However according to the UHCI spec, Bit 7 of the port
494      * status and control register is always set to 1.  So we try to
495      * use this to our advantage.  Another common failure mode when
496      * a nonexistent register is addressed is to return all ones, so
497      * we test for that also.
498      */
499     for (port = 0; port < (io_size - USBPORTSC1) / 2; port++) {
500         unsigned int portstatus;
501
502         portstatus = inw(uhci->io_addr + USBPORTSC1 + (port * 2));
503         if (!(portstatus & 0x0080) || portstatus == 0xffff)
504             break;
505     }
506     if (debug)
507         dev_info(uhci_dev(uhci), "detected %d ports\n", port);
508
509     /* Anything greater than 7 is weird so we'll ignore it. */
510     if (port > UHCI_RH_MAXCHILD) {
511         dev_info(uhci_dev(uhci), "port count misdetected? "
512                         "forcing to 2 ports\n");
513         port = 2;
514     }
515     uhci->rh_numports = port;
516
517     /* Kick BIOS off this hardware and reset if the controller
518      * isn't already safely quiescent.
519      */
520     check_and_reset_hc(uhci);
521     return 0;
522 }
```

可恶！又出来一个新的结构体：struct uhci_hcd，很显然，这是 UHCI 特有的。

```
371 struct uhci_hcd {
372
373     /* debugfs */
374     struct dentry *dentry;
375
376     /* Grabbed from PCI */
377     unsigned long io_addr;
378
379     struct dma_pool *qh_pool;
380     struct dma_pool *td_pool;
381
382     struct uhci_td *term_td;       /* Terminating TD, see UHCI bug */
383     struct uhci_qh *skelqh[UHCI_NUM_SKELQH];       /* Skeleton QHs */
384     struct uhci_qh *next_qh;       /* Next QH to scan */
385
386     spinlock_t lock;
387
388     dma_addr_t frame_dma_handle;    /* Hardware frame list */
389     __le32 *frame;
390     void **frame_cpu;            /* CPU's frame list */
391
392     enum uhci_rh_state rh_state;
393     unsigned long auto_stop_time;         /* When to AUTO_STOP */
394
395     unsigned int frame_number;           /* As of last check */
396     unsigned int is_stopped;
397 #define UHCI_IS_STOPPED       9999         /* Larger than a frame # */
398     unsigned int last_iso_frame;         /* Frame of last scan */
399     unsigned int cur_iso_frame;          /* Frame for current scan */
400
401     unsigned int scan_in_progress:1;    /* Schedule scan is running */
402     unsigned int need_rescan:1;          /* Redo the schedule scan */
403     unsigned int dead:1;                 /* Controller has died */
404     unsigned int working_RD:1;           /* Suspended root hub doesn't
405                                             need to be polled */
406     unsigned int is_initialized:1;      /* Data structure is usable */
407     unsigned int fsbr_is_on:1;           /* FSBR is turned on */
408     unsigned int fsbr_is_wanted:1;       /* Does any URB want FSBR? */
409     unsigned int fsbr_expiring:1;        /* FSBR is timing out */
410
```

```
411     struct timer_list fsbr_timer;         /* For turning off FBSR */
412
413     /* Support for port suspend/resume/reset */
414     unsigned long port_c_suspend;         /* Bit-arrays of ports */
415     unsigned long resuming_ports;
416     unsigned long ports_timeout;        /* Time to stop signalling */
417
418     struct list_head idle_qh_list;      /* Where the idle QHs live */
419
420     int rh_numports;                    /* Number of root-hub ports */
421
422     wait_queue_head_t waitqh;         /* endpoint_disable waiters */
423     int num_waiting;                    /* Number of waiters */
425     int total_load;                     /* Sum of array values */
426     short load[MAX_PHASE];               /* Periodic allocations */
427 };
```

写代码的人永远都不会体会到读代码人的痛苦，我真的觉得如果这些变态的数据结构再多出现几个的话就不想看了。我的忍耐是有底线的。

看 uhci_init 的 485 行，hcd_to_uhci，来自 drivers/usb/host/uhci-hcd.h，

```
430 static inline struct uhci_hcd *hcd_to_uhci(struct usb_hcd *hcd)
431 {
432     return (struct uhci_hcd *) (hcd->hcd_priv);
433 }
434 static inline struct usb_hcd *uhci_to_hcd(struct uhci_hcd *uhci)
435 {
436     return container_of((void *) uhci, struct usb_hcd, hcd_priv);
437 }
```

很显然，这两个函数完成的就是 uhci_hcd 和 usb_hcd 之间的转换。至于 hcd->hcd_priv 是什么？首先我们看到 struct usb_hcd 中有一个成员 unsigned long hcd_priv[0]。这就是传说中的零长度数组！

执行这个命令：

```
localhost:~ # info gcc "c ext" zero
```

就会知道什么是 GCC 中所谓的零长度数组。这是 GCC 对 C 的扩展。在标准 C 中我们定义数组时其长度至少为 1，而在 Linux 内核中结构体的定义里却经常看到最后一个元素定义为这种零长度数组，不占结构的空间，但它意味着这个结构体的长度允满了变数，即 sizeof(hcd_priv)==0 的作用就相当于一个占位符。申请空间时可使用它：

```
hcd = kzalloc(sizeof(*hcd) + driver->hcd_priv_size, GFP_KERNEL);
```

实际上，这就是 usb_create_hcd 中的一行，只不过当时没讲，现在知道，逃避不是办法，必须面对。driver->hcd_priv_size 可以在 uhci_driver 中找到，即 sizeof(struct uhci_hcd)，所以最终 hcd->hcd_priv 代表的就是 struct uhci_hcd，只不过需要一个强制转换。这样就能理解 hcd_to_uhci 了。当然，反过来，uhci_to_hcd 的作用就更显而易见了。不过我倒是想友情提醒一下，现在是在讲 UHCI 主机控制器的驱动程序，那么在这个故事里，有一些结构体变量是唯一的，比如 struct uhci_hcd 的结构体指针，即 uhci；而 struct usb_hcd 的结构体指针即 hcd。即以后我们如果见到某个指针名字为 uhci 或者 hcd，那就不用再多解释了。此 uhci 即彼 uhci，此 hcd 即彼 hcd。

接下来，io_size 和 io_addr 的赋值都很好懂。

然后是决定这个 Root Hub 到底有几个端口。端口号是从 0 开始的，UHCI 的 Root Hub 最多不能超过 8 个端口，即端口号不能超过 7。这段代码的含义注释里面说得很清楚，首先 UHCI 定义了 PORTSC 寄存器，全称为 PORT STATUS AND CONTROL REGISTER，即端口状态和控制寄存器，只要有一个端口就有一个寄存器。而每个这类寄存器都是 16 个 bits，即两个 bytes，因此从地址安排上来说，每一个端口占两个 bytes，而 spec 规定端口 1 的地址位于基址开始的第 10h 和 11h，端口 2 的地址向后顺推，即位于基址开始的第 12h 和 13h，依此类推，USBPORTSC1 这个宏的值是 16，USBPORTSC2 的宏值为 18，这两个宏用来标志端口的偏移量，显然 16 就是 10h，而 18 就是 12h。UHCI 定义的寄存器中，PORTSC 是最后一类寄存器，它后面没有更多的寄存器了，但是它究竟有几个 PORTSC 寄存器就不知道了，否则也就不用判断有多少个端口了。

于是这段代码就是从 USBPORTSC1 开始往后走，一直循环下去，读取这个寄存器的值，即 portstatus，按 spec 规定，这个寄存器的值中 bit7 是一个保留位，应该一直为 1。所以如果不为 1，那么就不用往下走了，说明已经没有寄存器了。另一种常见的错误是读出来都为 1，经验表明，这种情况也表示没有寄存器了。说明一下，inw 就是读 IO 端口的函数，w 就表示按 word 读。很显然这里要按 word 读，因为 PORTSC 寄存器是 16bits，一个字（word）。inw 所接的参数就是具体的 I/O 地址，即基址加偏移量。

510 行，UHCI_RH_MAXCHILD 就是 7，port 不能大于 7，否则出错了，于是设置 port 为 2，因为 UHCI spec 规定每个 Root Hub 最少有两个端口。

于是，uhci->rh_numports 被用来记录 Root Hub 的端口数。

520 行，check_and_reset_hc()函数特“虚伪”，看似“超简单”其实“超复杂”。定义于 drivers/usb/host/uhci-hcd.c 中：

```
169 static void check_and_reset_hc(struct uhci_hcd *uhci)
170 {
171     if (uhci_check_and_reset_hc(to_pci_dev(uhci_dev(uhci)),
                                    uhci->io_addr))
172         finish_reset(uhci);
173 }
```

看上去就两行，可这两行足以让我看了半个小时。首先第一个函数，uhci_check_and_reset_hc 来自 drivers/usb/host/pci-quirks.c，

```
91 int uhci_check_and_reset_hc(struct pci_dev *pdev, unsigned long base)
92 {
93      u16 legsup;
94      unsigned int cmd, intr;
95
96      /*
97       * When restarting a suspended controller, we expect all the
98       * settings to be the same as we left them:
99       *
100      *     PIRQ and SMI disabled, no R/W bits set in USBLEGSUP;
101      *     Controller is stopped and configured with EGSM set;
102      *     No interrupts enabled except possibly Resume Detect.
103      *
104      * If any of these conditions are violated we do a complete reset.
105      */
106     pci_read_config_word(pdev, UHCI_USBLEGSUP, &legsup);
107     if (legsup & ~(UHCI_USBLEGSUP_RO | UHCI_USBLEGSUP_RWC)) {
108         dev_dbg(&pdev->dev, "%s: legsup = 0x%04x\n",
109                             __FUNCTION__, legsup);
110         goto reset_needed;
111     }
112
113     cmd = inw(base + UHCI_USBCMD);
114     if ((cmd & UHCI_USBCMD_RUN) || !(cmd & UHCI_USBCMD_CONFIGURE) ||
115                     !(cmd & UHCI_USBCMD_EGSM)) {
116         dev_dbg(&pdev->dev, "%s: cmd = 0x%04x\n",
117                             __FUNCTION__, cmd);
118         goto reset_needed;
119     }
120
121     intr = inw(base + UHCI_USBINTR);
122     if (intr & (~UHCI_USBINTR_RESUME)) {
123         dev_dbg(&pdev->dev, "%s: intr = 0x%04x\n",
124                             __FUNCTION__, intr);
125         goto reset_needed;
126     }
127     return 0;
129 reset_needed:
130     dev_dbg(&pdev->dev, "Performing full reset\n");
131     uhci_reset_hc(pdev, base);
132     return 1;
133 }
```

而第二个函数 finish_reset 来自 drivers/usb/host/uhci-hcd.c：

```
129 static void finish_reset(struct uhci_hcd *uhci)
130 {
```

```
131     int port;
133     /* HCRESET doesn't affect the Suspend, Reset, and Resume Detect
134      * bits in the port status and control registers.
135      * We have to clear them by hand.
136      */
137     for (port = 0; port < uhci->rh_numports; ++port)
138         outw(0, uhci->io_addr + USBPORTSC1 + (port * 2));
140     uhci->port_c_suspend = uhci->resuming_ports = 0;
141     uhci->rh_state = UHCI_RH_RESET;
142     uhci->is_stopped = UHCI_IS_STOPPED;
143     uhci_to_hcd(uhci)->state = HC_STATE_HALT;
144     uhci_to_hcd(uhci)->poll_rh = 0;
146     uhci->dead = 0;       /* Full reset resurrects the controller */
147 }
```

一起看这两个函数。pci_read_config_word 函数的作用正如同它的字面意思一样：读寄存器，读什么寄存器？就是那张“上坟图”呗。

在 drivers/usb/host/quirks.c 中有一打关于这些宏的定义：

```
20 #define UHCI_USBLEGSUP         0xc0          /* legacy support */
21 #define UHCI_USBCMD            0             /* command register */
22 #define UHCI_USBINTR           4             /* interrupt register */
23 #define UHCI_USBLEGSUP_RWC     0x8f00         /* the R/WC bits */
24 #define UHCI_USBLEGSUP_RO      0x5040        /* R/O and reserved bits */
25 #define UHCI_USBCMD_RUN        0x0001         /* RUN/STOP bit */
26 #define UHCI_USBCMD_HCRESET    0x0002        /* Host Controller reset */
27 #define UHCI_USBCMD_EGSM       0x0008         /* Global Suspend Mode */
28 #define UHCI_USBCMD_CONFIGURE   0x0040         /* Config Flag */
29 #define UHCI_USBINTR_RESUME    0x0002     /* Resume interrupt enable */
```

UHCI_USBLEGSUP 是一个比较特殊的寄存器，LEGSUP 全称为 LEGACY SUPPORT REGISTER，UHCI spec 的第五章第二节专门介绍了这个寄存器。一调用 pci_read_config_word 函数就是把这个寄存器的值读出来，而把结果保存在变量 legsup 中，接着下一行就来判断它的值。

这里首先介绍三个概念。我们注意到寄存器的每一位都有一个属性，比如 RO（Read Only，只读），RW（Read/Write，可读可写），RWC（Read/Write Clear）。寄存器的某位如果是 RWC 的属性，那么表示该位可以被读可以被写，然而，与 RW 不同的是，如果写了一个 1 到该位，将会把该位清为 0。倘若你写的是一个 0，则什么也不会发生，对这个世界不产生任何改变。（UHCI Spec 中是这么说的：“R/WC Read/Write Clear. A register bit with this attribute can be read and written. However, a write of a 1 clears (sets to 0) the corresponding bit and a write of a 0 has no effect.”）

而 UHCI_USBLEGSUP_RO 为 0x5040，即 0101 0000 0100 0000，标志 LEGSUP 寄存器的 bit 6，bit 12 和 bit 14 三位为 1，这几位是只读的（bit 14 是保留位）。UHCI_USBLEGSUP_RWC 为 0x8f00，即 1000 1111 0000 0000，即标志着 LEGSUP 寄存器的 bit 8，bit 9，bit 10，bit 11 和 bit 15 为 1，这几位的属性是 RWC 的。这里让这两个宏“或”一下，按位去反，然后让 legsup 和它们相与，其效果就是判断 LEGSUP 寄存器的 bit 0，bit 1，bit 2，bit 3，bit 4，bit 5，bit 7 和 bit 13 是否为 1，这几位其实就是 RW 的。

这里的注释说，这几位任何一位为 1 则表示需要“reset”，其实这种注释是不太负责任的，仔细看一下 UHCI spec 会发现，RW 的这些位并不是因为它们是 RW 位就应该被 reset，而是因为 bit0~bit5，bit7，bit13 实际上都是一些 enable/disable 的开关，特别是中断相关的使能位，当我们还没有准备就绪时，我们理应把它们关掉。就相当于我新买了一个手机，而还没有号码，那我出于省电的考虑，基本上会选择把手机先关掉，等到我有号了，才会去把手机打开。而其他位都是一些状态位，它们为 0 还是为 1 只是表明不同的状态。状态位影响不大。

113 行， UHCI_USBCMD 表征 UHCI 的命令寄存器，这也是 UHCI spec 中定义的寄存器。这个寄存器为 00h 和 01h 处，所以 UHCI_USBCMD 的值为 0。

关于这个寄存器，需要考虑几位。首先是 bit 0，这一位被称作 RS bit，即 Run/Stop，当这一位被设置为 1，表示 HC 开始处理执行调度，调度什么？比如，传说中的 urb。显然，现在时机还未成熟，所以这一位必须设置为 0。这里代码的意思是如果它为 1，就执行 reset。UHCI_USBCMD_RUN 的值为 0x0001，即表征 bit 0。

另两个需要考虑的是 bit 3 和 bit 6。bit 3 被称为 EGSM，即 Enter Global Suspend Mode，这一位为 1 表示 HC 进入 Global Suspend Mode，这期间是不会有 USB 交易的。把这一位设置为 0 则是跳出这个模式，显然咱们这里的判断是如果这一位被设置为了 0，就执行 reset，否则就没有必要。因为 reset 的目的是为了清除当前存在于总线上的任何交易，让主机控制器和设备都“忘了过去，重新开始新的生活”。

而 bit 6 被称为 CF，即 Configure Flag，设置了这一位表示主机控制器当前正在被配置的过程中，显然如果主机控制器还在这个阶段就没有必要 reset 了。从逻辑上来说，关于这个寄存器的判断，我们的理念是，如果 HC 是停止挂起的，并且它还是配置的。没有必要做“reset”的。

接下来，121 行，读另一个寄存器，UHCI_USBINTR，值为 4。UHCI spec 中定义了一个中断使能寄存器。其 I/O 地址位于 04h 和 05h。很显然一开始我们得关中断。关于这个寄存器的描述如表 3.7.1 所示：

表 3.7.1 UHCI_USBINTR 寄存器

Bit	描述
15:4	保留
3	Short Packet Interrupt（短包中断）Enable. 1=Enable. 0=Disable.
2	Interrupt On Complete（完成时中断，IOC）Enable. 1=Enable. 0=Disable.
1	Resume Interrupt（唤醒中断）Enable. 1=Enable. 0=Disable.
0	Timeout/CRC Interrupt（超时/CRC）Enable. 1=Enable. 0=Disable.

谢天谢地，bit 15 到 bit 4 是保留位，并且默认应该是 0，所以无需理睬。而剩下几位在现阶段应该要 disable 掉，或者说应该设置为 0。唯有 bit 1 是个例外，正如 uhci_check_and_reset_hc 函数前的注释里说的一样，调用这个函数有两种可能的上下文：一种是主机控制器刚刚被发现时，这是一次性的工作；另一种是电源管理中的“resume”之时。虽然此时此刻调用这个函数是处于第一种上下文，但显然第二种情景发生的频率更高，可能性更大。

而对于“resume”的情况，显然这个唤醒中断使能寄存器必须被 enable。（这里也能解释为何不应该清掉 LEGSUP 寄存器里面的状态位，还应该尽量保持它们。因为如果从“suspend”回到“resume”，当然希望之前的状态得到保留，否则状态改变了那不就乱了么？那也就不叫恢复了。）

最终，127 行，如果前面的三条 goto 语句都没有执行，那么说明并不需要执行“reset”，这里就直接返回了，返回值为 0。反之如果前面任何一条 goto 语句执行了，那么就往下走，执行 uhci_reset_hc，然后返回 1。

函数 uhci_reset_hc 也来自 drivers/usb/host/pci-quirks.c：

```
59 void uhci_reset_hc(struct pci_dev *pdev, unsigned long base)
60 {
61      /* Turn off PIRQ enable and SMI enable.  (This also turns off the
62       * BIOS's USB Legacy Support.)  Turn off all the R/WC bits too.
63       */
64      pci_write_config_word(pdev, UHCI_USBLEGSUP, UHCI_USBLEGSUP_RWC);
65
66      /* Reset the HC - this will force us to get a
67       * new notification of any already connected
68       * ports due to the virtual disconnect that it
69       * implies.
70       */
71      outw(UHCI_USBCMD_HCRESET, base + UHCI_USBCMD);
72      mb();
73      udelay(5);
74      if (inw(base + UHCI_USBCMD) & UHCI_USBCMD_HCRESET)
75          dev_warn(&pdev->dev, "HCRESET not completed yet!\n");
76
77      /* Just to be safe, disable interrupt requests and
78       * make sure the controller is stopped.
79       */
80      outw(0, base + UHCI_USBINTR);
81      outw(0, base + UHCI_USBCMD);
82 }
```

这个函数其实就是一堆寄存器操作：读寄存器或者写寄存器。这种代码完全就是纸老虎，看上去挺恐怖，一堆的宏啊，寄存器啊，其实这些东西对我们完全就是小菜一碟。

首先 64 行，pci_write_config_word 就是写寄存器，写的还是 UHCI_USBLEGSUP 寄存器，即 LEGSUP 寄存器，写入 UHCI_USBLEGSUP_RWC，根据对 RWC 的解释，这样做的后果就是让这几位都清零。凡是 RWC 的 bit 其实都是在传达某种事件的发生，而清零往往代表的是认可这件事情。

然后 71 行，outw 的作用就是写端口地址，这里写的是 UHCI_USBCMD，即写命令寄存器，而 UHCI_USBCMD_HCRESET 表示什么意思呢？UHCI spec 中是这样说的：“Host Controller Reset (HCRESET). When this bit is set, the Host Controller module resets its internal timers, counters, state machines, etc. to their initial value. Any transaction currently in progress on USB is immediately terminated. This bit is reset by the Host Controller when the reset process is complete.”

显然，这就是真正的执行硬件上的“reset”。把计时器、计数器和状态机全都给复位。当然这件事情是需要一段时间的，所以这里调用 udelay 来延时 5 ms。最后再读这一位，因为正如上面这段英文里所说的那样，当“reset”完成了之后，这一位会被硬件“reset”，即这一位应该为 0。

最后 80 行和 81 行，写寄存器，把 0 写入中断使能寄存器和命令寄存器，这就是彻底的“reset”，关掉中断请求，并且停止 HC。

终于，我们结束了 uhci_check_and_reset_hc，如果执行了 reset，返回值应该为 1。则将执行 finish_reset 函数。这个函数的代码前面已经贴出来了，因为它只是做一些简单的赋值。唯一有一行写寄存器的循环操作，其含义又在注释里写得很明白。至于这些赋值究竟意味着什么，之后遇到了再说。

于是又跳出了 check_and_reset_hc(uhci)，回到了 uhci_init，不过幸运的是，这个函数也该结束了，返回值为 0，就来个三级跳，回到了 usb_add_hcd。

8、有一种资源,叫中断

结束了 uhci_init 回到亲爱的 usb_add_hcd 之后，1604 行到 1606 行是调试语句，飘过。

有一种液体叫眼泪，曾经以为，闭上眼睛，眼泪就不会流出来了。的确，眼泪流回了心里。

有一种资源叫中断，曾经以为，关掉中断，USB 主机就不会工作了。的确，USB 主机没有中断基本就“挂了”……

1609 行，中断，判断 driver 有没有一个 irq 函数，如果没有，就简单地记录 hcd->irq 为-1（hcd->irq 就是用来记录传说中的中断号的）。如果有，那就有事情要做了。显然 uhci_driver 的赋值就是 uhci_irq。当然，现在不用执行它，只要为它做点事情。最重要的是 request_irq 这个函数。我们先来查看它直观的效果。

加载 uhci-hcd 之前：

```
localhost:~ # cat /proc/interrupts
           CPU0
  0:      29906   IO-APIC-edge      timer
  1:         10   IO-APIC-edge      i8042
  8:          2   IO-APIC-edge      rtc
  9:          3   IO-APIC-fasteoi   acpi
 12:        115   IO-APIC-edge      i8042
 14:       6800   IO-APIC-edge      ide0
 16:        780   IO-APIC-fasteoi   eth0
NMI:          0
LOC:       1403
ERR:          0
MIS:          0
```

加载 uhci-hcd 模块：

```
localhost:~ # modprobe usbcore
localhost:~ # modprobe uhci-hcd
```

加载 uhci-hcd 模块之后：

```
localhost:~ # cat /proc/interrupts
           CPU0
  0:      32625   IO-APIC-edge      timer
```

```
  1:         10   IO-APIC-edge      i8042
  8:          2   IO-APIC-edge      rtc
  9:          3   IO-APIC-fasteoi   acpi
 12:        115   IO-APIC-edge      i8042
 14:       6915   IO-APIC-edge      ide0
 16:        870   IO-APIC-fasteoi   eth0, uhci_hcd:usb1
 17:          0   IO-APIC-fasteoi   uhci_hcd:usb4
 18:          0   IO-APIC-fasteoi   uhci_hcd:usb2
 19:          0   IO-APIC-fasteoi   uhci_hcd:usb3
NMI:          0
LOC:       1403
ERR:          0
MIS:          0
```

众所周知，/proc/interrupts 列出了计算机中中断资源的使用情况。这其中 uhci_hcd：usb1/usb2/usb3/usb4 这几个字符串就是 request_irq 中的倒数第二个参数，即实参 hcd->irq_descr，而这个字符串的赋值就是 1610 行那个 snprintf 语句的职责。hcd->driver->description 就是“uhci-hcd”，hcd->self.busnum 就是 1，2，3，4。因为这台机器一共 4 个 usb 主机控制器，它们的编号分别是 1，2，3，4。

request_irq 的具体作用是请求中断资源，更准确地说是安装中断处理函数或者叫中断句柄（interrupt handler)。

这个函数的第 1 个参数就是中断号，irqnum 是一路传下来的，即最初的那个 dev->irq。

第 2 个参数就是中断句柄。这里传递的是 usb_hcd_irq 函数，它将会在响应中断时被调用。

第 3 个参数，irqflags，在 probe 中调用 usb_add_hcd 时，传递的第三个参数是 IRQF_SHARED，这个参数也被传递给了 request_irq，所以这里的 irqflags 为 IRQF_SHARED，这表示该中断可以被多个设备共享。这也是为什么 uhci-hcd：usb1 和网卡驱动 eth0 用的是同一个中断号。之所以要共享，是因为在操作系统中，也要节约资源，尽量和别的设备共享中断号。

第 4 个参数就是字符串。第 5 个参数主要是用来标志使用中断的设备，它是一个指针，这个参数只是用来区分不同的设备，可以把它设置为 NULL，不用它。但更多的情况是它被设置为指向驱动程序私有的数据。传递的是 hcd 本身，这样在 usb_hcd_irq 函数中就可以使用它，因为事实上在 usb_hcd_irq 中把它当成一个参数来用。

这样 request_irq 的作用就明白了，以后释放中断资源时我们只要调用 free_irq 函数即可。这个函数将会在 usb_remove_hcd 时被调用。

最后 1618 行，也把 irqnum 记录在了 hcd->irq 中。

再强调一下，usb_hcd_irq 是这个故事中很重要的角色，它将在未来主机控制器需要中断时被调用。希望不要把她忘怀。作为一个中断函数，如果不是它以后有一定的利用价值，咱们现在完全没有必要为它注册，这非常符合常理！

9、一个函数引发的故事

9.1 故事的开始

接下来，1632 行，下一个函数，driver->start 被调用。对于 uhci_driver，其 start 指针指向的是 uhci_start 函数，经过了“人间大炮一级准备、二级准备”之后，这个函数基本上就算介于三级准备和发射之间了。这个函数算是整个故事中最重要的一个函数，理解它是理解整个 uhci 的关键，它来自文件 drivers/usb/host/uhci-hcd.c：

```
556 static int uhci_start(struct usb_hcd *hcd)
557 {
558     struct uhci_hcd *uhci = hcd_to_uhci(hcd);
559     int retval = -EBUSY;
560     int i;
561     struct dentry *dentry;
562
```

```
563     hcd->uses_new_polling = 1;
564
565     spin_lock_init(&uhci->lock);
566     setup_timer(&uhci->fsbr_timer, uhci_fsbr_timeout,
567                     (unsigned long) uhci);
568     INIT_LIST_HEAD(&uhci->idle_qh_list);
569     init_waitqueue_head(&uhci->waitqh);
570
571     if (DEBUG_CONFIGURED) {
572         dentry = debugfs_create_file(hcd->self.bus_name,
573                         S_IFREG|S_IRUGO|S_IWUSR, uhci_debugfs_root,
574                         uhci, &uhci_debug_operations);
575         if (!dentry) {
576             dev_err(uhci_dev(uhci), "couldn't create uhci "
577                                 "debugfs entry\n");
578             retval = -ENOMEM;
579             goto err_create_debug_entry;
580         }
581         uhci->dentry = dentry;
582     }
583
584     uhci->frame = dma_alloc_coherent(uhci_dev(uhci),
585                     UHCI_NUMFRAMES * sizeof(*uhci->frame),
586                     &uhci->frame_dma_handle, 0);
587     if (!uhci->frame) {
588         dev_err(uhci_dev(uhci), "unable to allocate "
589                         "consistent memory for frame list\n");
590         goto err_alloc_frame;
591     }
592     me mset(uhci->frame, 0, UHCI_NUMFRAMES * sizeof(*uhci->frame));
593
594    uhci->frame_cpu = kcalloc(UHCI_NUMFRAMES, sizeof(*uhci->frame_cpu),
595                     GFP_KERNEL);
596     if (!uhci->frame_cpu) {
597         dev_err(uhci_dev(uhci), "unable to allocate "
598                         "memory for frame pointers\n");
599         goto err_alloc_frame_cpu;
600     }
601
602     uhci->td_pool = dma_pool_create("uhci_td", uhci_dev(uhci),
603                     sizeof(struct uhci_td), 16, 0);
604     if (!uhci->td_pool) {
605         dev_err(uhci_dev(uhci), "unable to create td dma_pool\n");
606         goto err_create_td_pool;
607     }
608
609     uhci->qh_pool = dma_pool_create("uhci_qh", uhci_dev(uhci),
610                     sizeof(struct uhci_qh), 16, 0);
611     if (!uhci->qh_pool) {
612         dev_err(uhci_dev(uhci), "unable to create qh dma_pool\n");
613         goto err_create_qh_pool;
614     }
615
616     uhci->term_td = uhci_alloc_td(uhci);
617     if (!uhci->term_td) {
618         dev_err(uhci_dev(uhci), "unable to allocate terminating TD\n");
619         goto err_alloc_term_td;
620     }
621
622     for (i = 0; i < UHCI_NUM_SKELQH; i++) {
623         uhci->skelqh[i] = uhci_alloc_qh(uhci, NULL, NULL);
624         if (!uhci->skelqh[i]) {
625             dev_err(uhci_dev(uhci), "unable to allocate QH\n");
626             goto err_alloc_skelqh;
627         }
628     }
629
630     /*
631      * 8 Interrupt queues; link all higher int queues to int1 = async
632      */
633     for (i = SKEL_ISO + 1; i < SKEL_ASYNC; ++i)
634         uhci->skelqh[i]->link = LINK_TO_QH(uhci->skel_async_qh);
635     uhci->skel_async_qh->link = UHCI_PTR_TERM;
636     uhci->skel_term_qh->link = LINK_TO_QH(uhci->skel_term_qh);
637
```

```
638     /* This dummy TD is to work around a bug in Intel PIIX controllers*/
639     uhci_fill_td(uhci->term_td, 0, uhci_explen(0) |
640                     (0x7f << TD_TOKEN_DEVADDR_SHIFT) | USB_PID_IN, 0);
641     uhci->term_td->link = UHCI_PTR_TERM;
642     uhci->skel_async_qh->element = uhci->skel_term_qh->element =
643                     LINK_TO_TD(uhci->term_td);
644
645     /*
646      * Fill the frame list: make all entries point to the proper
647      * interrupt queue.
648      */
649     for (i = 0; i < UHCI_NUMFRAMES; i++) {
650
651         /* Only place we don't use the frame list routines */
652         uhci->frame[i] = uhci_frame_skel_link(uhci, i);
653     }
654
655     /*
656      * Some architectures require a full mb() to enforce completion of
657      * the memory writes above before the I/O transfers in configure_hc().
658      */
659     mb();
660
661     configure_hc(uhci);
662     uhci->is_initialized = 1;
663     start_rh(uhci);
664     return 0;
665
666 /*
667  * error exits:
668  */
669 err_alloc_skelqh:
670     for (i = 0; i < UHCI_NUM_SKELQH; i++) {
671         if (uhci->skelqh[i])
672             uhci_free_qh(uhci, uhci->skelqh[i]);
673     }
674
675     uhci_free_td(uhci, uhci->term_td);
676
677 err_alloc_term_td:
678     dma_pool_destroy(uhci->qh_pool);
679
680 err_create_qh_pool:
681     dma_pool_destroy(uhci->td_pool);
683 err_create_td_pool:
684     kfree(uhci->frame_cpu);
686 err_alloc_frame_cpu:
687     dma_free_coherent(uhci_dev(uhci),
688                     UHCI_NUMFRAMES * sizeof(*uhci->frame),
689                     uhci->frame, uhci->frame_dma_handle);
691 err_alloc_frame:
692     debugfs_remove(uhci->dentry);
693
694 err_create_debug_entry:
695     return retval;
696 }
```

这个函数简直就是一个大杂烩，所有“变态”的代码全都集中在这一个函数中边了。天下没有轻松的成功，成功要付代价。在这个浮躁的社会中，也许很难再有人能够静下心来看代码了。都说现在的程序员是做一天程序撞一天钟，我们这些读程序的又何尝不是这种心态呢？

面对这越来越枯燥的代码，我想我们不能再像过去那样分析了。记得一位朋友曾经教育过我，读懂 Linux 内核代码和读懂女人的心一样，不是不可能，只是需要你多下点功夫，多用点时间，多多沟通，多多了解。这套理论我觉得很有道理，所以我想从现在开始我决定多用点时间多下点功夫来读代码，要和代码多多沟通才能对它有多多了解。所以我决定用出我的杀手锏 KDB。也许你不熟悉 KDB，没有关系，我只是通过 KDB 来展示一些函数调用关系。我主要会使用 KDB 的 bp 命令来设置断点，用 bt 命令来显示函数调用堆栈。很显然，了解了函数的调用关系对读懂代码是很有帮助的。

首先在加载 uhci-hcd 时设置断点 uhci_start。于是会在 uhci_start 被调用时进入 KDB，用 bt 命令看一下堆栈的 traceback。

```
kdb>bt
Stack traceback for pid 3498
0xdd5ac550 3498 3345 1 0 R 0xdd5ac720 *modprobe
esp        eip         Function (args)
0xd4a89d40 0xc0110000 lapic_resume+0x185
0xd4a89d48 0xe0226e41 [uhci_hcd]uhci_start
0xd4a89d54 0xe0297132 [usbcore]usb_add_hcd+0x3fb
0xd4a89da0 0xe02a0a9b [usbcore]usb_hcd_pci_probe+0x263
```

其实堆栈中还有更多的函数，篇幅原因，与此没有太多关系的函数就不列出来了。但是至少从这个 traceback 中可以很清楚地知道目前的处境，在 uhci_start 中，而调用它的函数是 usb_add_hcd，后者则是被 usb_hcd_pci_probe 函数调用，而 usb_hcd_pci_probe 函数正是我们故事的真正开始处。

但单单是 KDB 还不足以显示我们的决心。有人说，女人如画，不同的画中有不同的风景，代码也是如此，左看右看，上看下看，角度不同风景各异。对于 uhci-hcd 这样变态的模块，用常规的方法恐怕是很难看明白了，我们必须引入一种新的方法——图解法。从 uhci_start 函数开始，我们将接触到一堆乱七八糟的庞大而复杂的数据结构，这些数据结构的关系如果不能理解，那么我们很难读懂这代码。所以决定把 uhci_start 作为实验，通过图解法把众多复杂的结构众多的链表之间的关系给描绘出来。

9.2 图解法

571 行之前全是些初始化的代码，用到了再回来看。

571 行到 582 行，上次我们看到 DEBUG_CONFIGURED 是在 UHCI 的初始化代码中，即 uhci_hcd_init 函数中，这是一个编译开关，打开开关就是 1，不打开就是 0，这里假设打开。因为有必要多了解一下 debugfs，毕竟当初已经接触过 debugfs 了。而且当时已经看到函数 debugfs_create_dir 在/sys/kernel/debug 下面建立了一个 uhci 的目录，而现在看到的这个 debugfs_create_file 很自然，就是在/sys/kernel/debug/uhci 下面建立文件，比如：

```
localhost:~ # ls -l /sys/kernel/debug/uhci/
total 0
-rw-r--r-- 1 root root 0 Oct  8 13:18 0000:00:1d.0
-rw-r--r-- 1 root root 0 Oct  8 13:18 0000:00:1d.1
-rw-r--r-- 1 root root 0 Oct  8 13:18 0000:00:1d.2
-rw-r--r-- 1 root root 0 Oct  8 13:18 0000:00:1d.3
```

可以看见，在这个目录下面针对每个 UHCI 主机控制器各建立了一个文件，文件名就是该设备的 PCI 名，即域名+总线名+插槽号+功能号。

接下来从 584 行到 628 行，全都是内存申请相关的，包括可恶的 DMA。不过这些函数已经不再陌生，dma_alloc_coherent，dma_pool_create 都已经讲过，但要说清楚这些实际的代码，则必须借助一张来自 UHCI spec 的经典图，如图 3.9.1 所示。

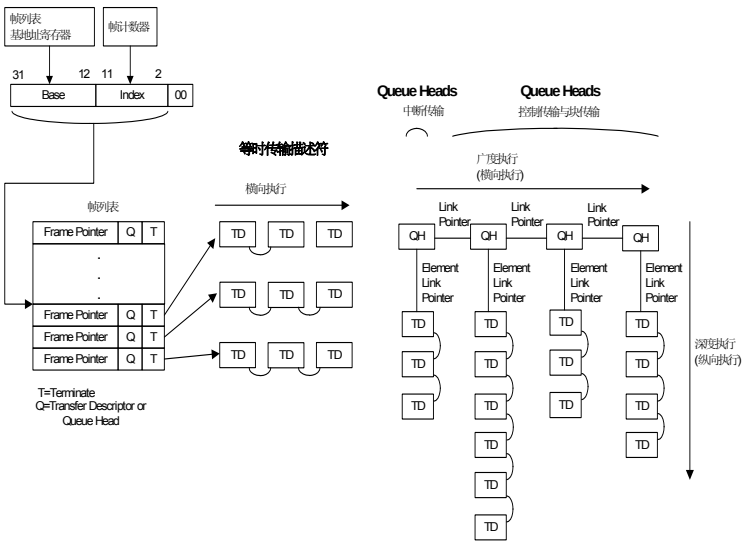

图 3.9.1 调度图

有了上面这张图，你就可以运筹帷幄之内决胜千里之外。这张图对于我们研究 UHCI 的意义，就好比 1993 年那部何家劲版的少年张三丰中那张藏宝图，所有人都想得到它，因为得到了它就意味着得到一切。而对于所有写 UHCI 主机驱动的人来说，他们对于这幅图的共同心声是：输了你，赢了世界又如何？

之所以这幅图如此重要，是因为 UHCI 主机控制器的原理完全集中在这幅图中。

主机控制器最重要的一个职责是调度。那么它如何调度呢？首先你得把帧列表（Frame List）的起始地址告诉它，由这个 List 将会牵出许多的 TD 和 QH 来。

首先 Frame List 是一个数组，最多 1024 个元素。每一个元素代表一个 Frame，时间上来说就是 1 ms。而和每一个 Frame 相联系的是“transaction”，即交易。比如说一次传输，就可以算作一笔交易。而 TD 和 QH 就是用来表征这些交易的。Frame List 的每一个元素会包含指针指向 TD 或者 QH，实际上 Frame List 的每一个元素被称作一个 Frame List Pointer，它一共有 32 个 bit，其中 bit31 到 bit4 则表示 TD 或者 QH 的地址，bit1 则表示指向的到底是 QH 还是 TD。

而从硬件上来说，访问这个 Frame List 的方法是使用一个基址寄存器和一个计数器，即图中我们看到的那个帧列表基地址寄存器（Frame List Base Address Register）和帧计数器（Frame Counter）。下面的图 3.9.2 也许更能说明它们的作用。

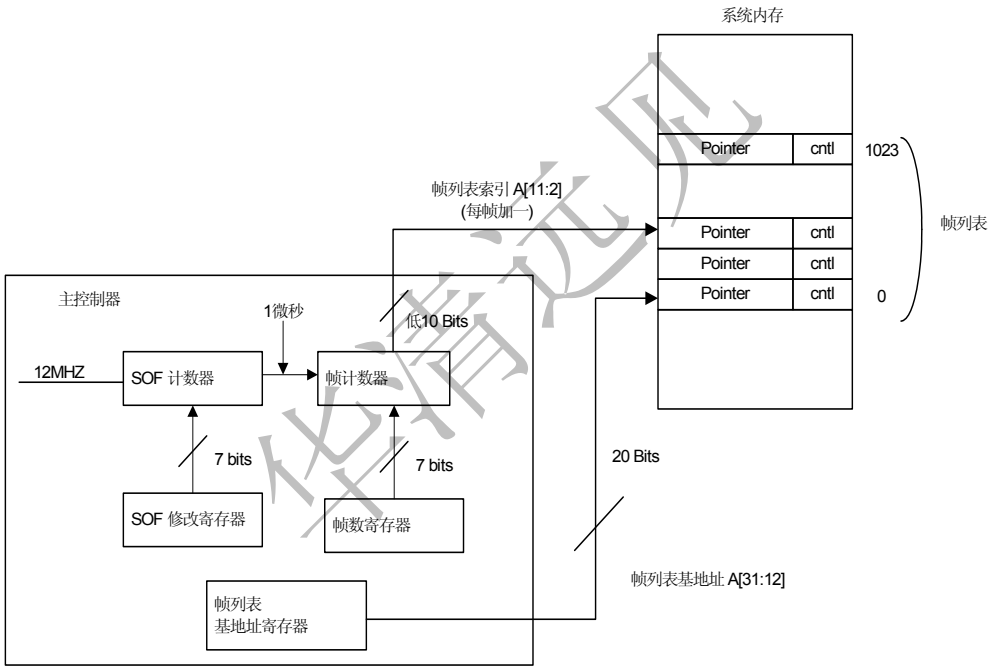

图 3.9.2　帧列表基地址寄存器和帧计数器

实际上在主机控制器中有一个时钟，用我们电子专业的术语来说就是主机控制器内部肯定有一个晶体振荡器，从而实现一个周期为 1 msec 的信号，于是帧列表基地址寄存器和帧计数器就去遍历这张 Frame List，它们在这张表里一毫秒前进一格，走到 1023 了就又再次归零。

那么驱动程序的责任是什么呢？为图 3.9.1 那张调度图准备好相应的数据结构。填充 Fra

TD 实际上描述的就是最终要在 USB 总线上传输的数据包，学名为 Transaction Descriptor，是主机控制器的基本执行单位。UHCI spec 定义了两类 TD：ISO TD 和 non-ISO TD。即等时 TD 和非等时 TD。我们知道 USB 一共四种传输方式：中断、批量、控制和等时。这其中等时传输的 TD 虽然从数据结构的格式来说是一样的，但是作用不一样。从调度图来看，等时的 TD 也是专门被列出来了。主机控制器驱动程序负责填充 TD，主机控制器将会去获取 TD，然后执行相应的数据传输。

QH 就是队列头（Queue Head）。从这张调度图里也能看到，QH 实际上把各个非等时 TD 给连接了起来，组织成若干队列。

从图中我们看到一个现象，对主机控制器来说，4 种传输方式是有优先级的区别的，等时传输总是最先被满足，最先被执行，中断传输，再控制传输和批量传输。等时传输和中断传输都叫做周期传输，或者说定期传输。

再强调一下，驱动程序的任务就是填充这张图，硬件的作用则按照这张图去执行，这种分工是很明确的。

Ok，现在让我们结合代码和结构体定义来查看。

首先 584 行，使用 dma_alloc_coherent 申请内存，赋给 uhci->frame，而与此同时建立了 DMA 映射。frame_dma_handle 和 frame 是以后从软件方面来指代这个 Frame List 的，而 frame_dma_handle 因为是物理地址，要让它和真正的硬件联系起来，稍后在一个叫做 configure_hc 的函数中你会看到，我们会把它写入 Frame List 的基址寄存器。这样以后操作 uhci->frame 就等于真正的操作硬件了。这更意味着以后只要把申请的 TD、QH 和 uhci->frame 联系起来就可以了。这里我们也注意到，UHCI_NUMFRAMES 就是一个宏，它的值正是 1024。到目前为止，一切看起来都那么和谐。

而 594 行这个 frame_cpu 则是和 frame 相对应的，它是一个纯粹软件意义上的 Frame List。即 Frame 身上承担着硬件的使命，而 frame_cpu 则属于从软件角度来说记录这张 Frame List 的。

609 行，这两个 dma_poll_create 的作用就是创建内存池，为 TD 和 QH 申请内存带来方便。

616 行调用的这个 uhci_alloc_td 以及 623 行调用的 uhci_alloc_qh 则均来自 drivers/usb/host/uhci-q.c，先看 uhci_alloc_td，及其搭档 uhci_free_td。

```
106 static struct uhci_td *uhci_alloc_td(struct uhci_hcd *uhci)
107 {
108     dma_addr_t dma_handle;
109     struct uhci_td *td;
110
111     td = dma_pool_alloc(uhci->td_pool, GFP_ATOMIC, &dma_handle);
112     if (!td)
113         return NULL;
114
115     td->dma_handle = dma_handle;
116     td->frame = -1;
117
118     INIT_LIST_HEAD(&td->list);
119     INIT_LIST_HEAD(&td->fl_list);
120
121     return td;
122 }
123
124 static void uhci_free_td(struct uhci_hcd *uhci, struct uhci_td *td)
125 {
126     if (!list_empty(&td->list)) {
127         dev_warn(uhci_dev(uhci), "td %p still in list!\n", td);
128         WARN_ON(1);
129     }
130     if (!list_empty(&td->fl_list)) {
131         dev_warn(uhci_dev(uhci), "td %p still in fl_list!\n", td);
132         WARN_ON(1);
133     }
134
135     dma_pool_free(uhci->td_pool, td, td->dma_handle);
136 }
```

这意思很明白。再来看后者，uhci_alloc_qh 及其搭档 uhci_free_qh。

```
247 static struct uhci_qh *uhci_alloc_qh(struct uhci_hcd *uhci,
248                 struct usb_device *udev, struct usb_host_endpoint *hep)
249 {
250     dma_addr_t dma_handle;
251     struct uhci_qh *qh;
252
253     qh = dma_pool_alloc(uhci->qh_pool, GFP_ATOMIC, &dma_handle);
254     if (!qh)
255         return NULL;
256
257     me mset(qh, 0, sizeof(*qh));
```

```
258     qh->dma_handle = dma_handle;
259
260     qh->element = UHCI_PTR_TERM;
261     qh->link = UHCI_PTR_TERM;
262
263     INIT_LIST_HEAD(&qh->queue);
264     INIT_LIST_HEAD(&qh->node);
265
266     if (udev) {           /* Normal QH */
267         qh->type = hep->desc.bmAttributes & USB_ENDPOINT_XFERTYPE_MASK;
268         if (qh->type != USB_ENDPOINT_XFER_ISOC) {
269             qh->dummy_td = uhci_alloc_td(uhci);
270             if (!qh->dummy_td) {
271                 dma_pool_free(uhci->qh_pool, qh, dma_handle);
272                 return NULL;
273             }
274         }
275         qh->state = QH_STATE_IDLE;
276         qh->hep = hep;
277         qh->udev = udev;
278         hep->hcpriv = qh;
279
280         if (qh->type == USB_ENDPOINT_XFER_INT ||
281                 qh->type == USB_ENDPOINT_XFER_ISOC)
282             qh->load = usb_calc_bus_time(udev->speed,
283                             usb_endpoint_dir_in(&hep->desc),
284                             qh->type == USB_ENDPOINT_XFER_ISOC,
285                             le16_to_cpu(hep->desc.wMaxPacketSize))
286                            / 1000 + 1;
287
288     } else {              /* Skeleton QH */
289         qh->state = QH_STATE_ACTIVE;
290         qh->type = -1;
291     }
292     return qh;
293 }
294
295 static void uhci_free_qh(struct uhci_hcd *uhci, struct uhci_qh *qh)
296 {
297     WARN_ON(qh->state != QH_STATE_IDLE && qh->udev);
298     if (!list_empty(&qh->queue)) {
299         dev_warn(uhci_dev(uhci), "qh %p list not empty!\n", qh);
300         WARN_ON(1);
301     }
302
303     list_del(&qh->node);
304     if (qh->udev) {
305         qh->hep->hcpriv = NULL;
306         if (qh->dummy_td)
307             uhci_free_td(uhci, qh->dummy_td);
308     }
309     dma_pool_free(uhci->qh_pool, qh, qh->dma_handle);
310 }
```

这个就明显复杂一些了。uhci_alloc_qh 的执行有两条路径：一种是 udev 有所指，一种是 udev 为空。我们这里传进来的是 NULL，所以暂时可以不看另一种路径，到时候需要看时再去看。这么一来就意味着此时此刻不需要看 266 到 287 这些行了。而这意味着能看到的只是一些简单的赋值操作而已。但是我们必须理解为什么这里有两种情况，因为这正是 uhci-hcd 这个驱动的精妙之处。

为了堆砌出那幅调度图，有一个很简单的方法，即每次有传输任务，建立一个或者几个 TD，把 urb 转换成 urb，然后把 TD 挂入 Frame List Pointer，不就可以了么？朋友，如果生活真的是这么简单，如果世界真的是这么单纯，那么也许现在的我依然是一张洁白的宣纸，绝不是现在这张被上海的梅雨湿润过的废纸。

9.3 TD 和 QH

从调度图可以看出，等时传输不需要 QH，只要把几个 TD 连接起来，让 Frame List Pointer 指向第一个 TD 就可以了。换言之，需要为等时传输准备一个队列，然后每一个 Frame 都让 Frame List Pointer 指向队列的头部。

那么应该如何操作中断传输呢？实际上为中断传输建立了 8 个队列，不同的队列代表了不同的周期，这 8 个队列分别代表的是 1 ms，2 ms，4 ms，8 ms，16 ms，32 ms，64 ms 和 128 ms。USB spec 里规定，对于全速/低速设备来说，其渴望周期撑死也不能超过 255 ms。这 8 个队列的队列头就叫做 Skeleton QH，而对于正儿八经的传输来说，我们需要另外专门的建立 QH，并往该 QH 中连接上相关的 TD，这类 QH 就是这里所称的 Normal QH。有人偏向于把往 QH 上连接 TD 称之为“为 QH 装备上 TD”。而这几个 Skeleton QH 是不会被装备任何 TD 的，要让别的中断 QH 知道自己该放置在何处。不过 Skeleton QH 并非只是为中断传输准备的，实际上，我们准备了 11 个 Skeleton QH，除了中断传输的 8 个以外，还有一个为等时传输准备的 QH，一个为表征“大部队结束”的 QH，一个为处理 unlink 而设计的 QH。这三个都有点特殊，我们遇到了再讲。

回到代码中来，从 uhci_start 函数的角度来看，我们注意到 uhci_alloc_qh 返回值是 qh，而 qh 是一个 struct uhci_qh 结构体变量，而刚才 uhci_alloc_td 函数的返回值 td，是一个 struct uhci_td 结构体变量。TD 和 QH 这两个概念说起来轻松，可是化成代码来表示的这两个结构体绝对不是省油的灯。先看 struct uhci_td，来自 drivers/usb/host/uhci-hcd.h 中：

```
242 struct uhci_td {
243     /* Hardware fields */
244     __le32 link;
245     __le32 status;
246     __le32 token;
247     __le32 buffer;
249     /* Software fields */
250     dma_addr_t dma_handle;
252     struct list_head list;
254     int frame;                      /* for iso: what frame? */
255     struct list_head fl_list;
256 } __attribute__((aligned(16)));
```

再看 struct uhci_qh，依然来自 drivers/usb/host/uhci-hcd.h：

```
126 struct uhci_qh {
127     /* Hardware fields */
128     __le32 link;                    /* Next QH in the schedule */
129     __le32 element;                 /* Queue element (TD) pointer */
131     /* Software fields */
132     dma_addr_t dma_handle;
134     struct list_head node;          /* Node in the list of QHs */
135     struct usb_host_endpoint *hep;  /* Endpoint information */
136     struct usb_device *udev;
137     struct list_head queue;         /* Queue of urbps for this QH */
138     struct uhci_td *dummy_td;       /* Dummy TD to end the queue */
139     struct uhci_td *post_td;        /* Last TD completed */
141     struct usb_iso_packet_descriptor *iso_packet_desc;
142                                   /* Next urb->iso_frame_desc entry */
143     unsigned long advance_jiffies;  /* Time of last queue advance */
144     unsigned int unlink_frame;      /* When the QH was unlinked */
145     unsigned int period;    /* For Interrupt and Isochronous QHs */
146     short phase;                    /* Between 0 and period-1 */
147     short load;                 /* Periodic time requirement, in us */
148     unsigned int iso_frame;       /* Frame # for iso_packet_desc */
149     int iso_status;                /* Status for Isochronous URBs */
151     int state;                     /* QH_STATE_xxx; see above */
152     int type;                    /* Queue type (control, bulk, etc) */
153     int skel;                      /* Skeleton queue number */
155     unsigned int initial_toggle:1; /* Endpoint's current toggle value */
156     unsigned int needs_fixup:1;  /* Must fix the TD toggle values */
157     unsigned int is_stopped:1;   /* Queue was stopped by error/unlink */
158     unsigned int wait_expired:1;  /* QH_WAIT_TIMEOUT has expired */
159     unsigned int bandwidth_reserved:1;    /* Periodic bandwidth has been allocated */
161 } __attribute__((aligned(16)));
```

某种意义上来说，struct usb_hcd，struct uhci_hcd 这些结构体和 struct uhci_td，struct uhci_qh 之间的关系就好比宏观经济学与微观经济学的关系。它们都是为了描述主机控制器，只是分别是从宏观角度和微观角度。从另一个角度来说，这些宏观的数据结构实际上是软件意义上的，而这些微观的数据结构倒是和硬件有着对应关系。从硬件上来说， Frame List，Isochronous Transfer Descriptors（简称 TD），Queue Heads（简称 QH），以及 queued Transfer Descriptors（也简称 TD）都是 UHCI spec 定义的数据结构。

先看 struct uhci_td 结构体。uhci_alloc_td 函数定义了两个局部变量：dma_handle 和 td，其中 dma_handle 传递给了 dma_pool_alloc 函数，于是知道它记录的是 td 的 DMA 地址。td 内部有一个成员 dma_addr_t dma_handle，它被赋值为 dma_handle。td 内部另一个成员 int frame，用来表征这个 td 所对应的 frame，目前初始化 frame 为-1。另外，td 还有两个成员，struct list_head list 和 struct list_head fl_list。这是两个队列。uhci_alloc_td 函数中用 INIT_LIST_HEAD 把它们俩进行了初始化。而反过来 uhci_free_td 函数的工作就是反其道而行之，调用 dma_pool_free 函数去释放这个内存。在释放之前检查了一下这两个队列是否为空，如果不为空会先提出警告。

再来看 struct uhci_qh 结构体，在 uhci_alloc_qh 函数中，首先也是定义两个局部变量，qh 和 dma_handle。使用的也是同样的套路，qh 调用 dma_pool_alloc 来申请，然后用 me mset 给它清零。dma_handle 同样传递给了 dma_pool_alloc，并且之后也赋值给了 qh->dma_handle。qh 同样有一个成员 dma_addr_t dma_handle。qh 中也有两个成员是队列：struct list_head node 和 struct list_head queue，同样也是在这里被初始化。此外，还有四个成员被赋了值，即 element、link、state 和 type。关于这四个赋值，我们暂时不提，用到了再说。不过我们应该回到 uhci_start 的上下文去看一下 uhci_alloc_qh 被调用的具体情况，622 行这里有一个循环，UHCI_NUM_SKELQH 是一个宏，这个宏的值为 11，所以这里就是申请了 11 个 QH，这正是我们前面介绍过的那个 11 个 Skeleton QH。与此同时我们注意到 struct uhci_hcd 中有一个成员 struct uhci_qh *skelqh[UHCI_NUM_SKELQH]，即有这么一个数组，数组 11 个元素，而这里就算是为这 11 个元素申请了内存空间。

接下来，要具体解释这里的代码。还得看下来自 drivers/usb/host/uhci-hcd.h 的一些宏：

```
317 #define UHCI_NUM_SKELQH         11
318 #define SKEL_UNLINK             0
319 #define skel_unlink_qh          skelqh[SKEL_UNLINK]
320 #define SKEL_ISO                1
321 #define skel_iso_qh             skelqh[SKEL_ISO]
322         /* int128, int64, ..., int1 = 2, 3, ..., 9 */
323 #define SKEL_INDEX(exponent)    (9 - exponent)
324 #define SKEL_ASYNC              9
325 #define skel_async_qh           skelqh[SKEL_ASYNC]
326 #define SKEL_TERM               10
327 #define skel_term_qh            skelqh[SKEL_TERM]
328
329 /* The following entries refer to sublists of skel_async_qh */
330 #define SKEL_LS_CONTROL         20
331 #define SKEL_FS_CONTROL         21
332 #define SKEL_FSBR               SKEL_FS_CONTROL
333 #define SKEL_BULK               22
```

好家伙，光注释就看得我一愣一愣的，可惜还是没看懂。但基本上我们能感觉出，当前我们的目标是为了建立 QH 数据结构，并把相关的队列给连接起来。

633 行，SKEL_ISO 是 1，SKEL_ASYNC 是 9，所以这里就是循环 7 次。实际上，在这 11 个元素的数组中，2 到 9 就是对应于中断传输的那 8 个 Skeleton QH，所以这里就是为这 8 个 QH 的 link 元素赋值。对于这 8 个 QH，周期为 128 ms 的那个 QH 被称为 int128，周期为 64 ms 的被称为 int64，于是就有了 int128，int64，…，int1，分别对应这个数组的 2，3，…，9 号元素。今后我们对这几个 QH 的称呼也是如此，skel int128 QH，skel int64 QH，…，skel int2 QH，skel int1 QH。

而这里我们还看到另一个家伙，skel_async_qh。它表示 async queue，指的是 low-speed control，full-speed control，bulk 这三种异步队列。与之对应的就是刚才的 SKEL_ASYNC 宏， SKEL_ASYNC 等于 9，而我们同时知道 skel int1 QH 实际上也是 skelqh[]数组的 9 号元素，所以实际上 skel_async_qh 和 skel int1 QH 是共用了同一个 QH，这是因为 skel int1 QH 表示中断传输的周期为 1 ms，而控制传输和批量传输也是每一个 ms 或者说每一个 Frame 都会被调度的，当然前提是带宽足够。所以这里的做法就是把 skel int128 QH 到 skel int2 QH 的 link 指针全都赋为 LINK_TO_QH(uhci->skel_async_qh)。

LINK_TO_QH 是一个宏，定义于 drivers/usb/host/uhci-hcd.h：

```
174 #define LINK_TO_QH(qh)   \
            (UHCI_PTR_QH | cpu_to_le32((qh)->dma_handle))
```

UHCI_PTR_QH 等一系列宏也来自同一文件：

```
76 #define UHCI_PTR_BITS          __constant_cpu_to_le32(0x000F)
77 #define UHCI_PTR_TERM          __constant_cpu_to_le32(0x0001)
78 #define UHCI_PTR_QH            __constant_cpu_to_le32(0x0002)
79 #define UHCI_PTR_DEPTH         __constant_cpu_to_le32(0x0004)
80 #define UHCI_PTR_BREADTH       __constant_cpu_to_le32(0x0000)
```

这样我们就要看 struct uhci_qh 这个结构体中的成员__le32 link 了。这是一个指针，这个指针指向下一个 QH，换言之，它包含着下一个 QH 或者下一个 TD 的地址。不过它一共 32 个 bits，其中只有 bit31 到 bit4 这些位是用来记录地址的，bit3 和 bit2 是保留位，bit1 则用来表示该指针指向的是一个 QH 还是一个 TD。bit1 如果为 1，表示本指针指向的是一个 QH，如果为 0，表示本指针指向的是一个 TD。（刚才这个宏 UHCI_PTR_QH 正是起到这个作用的，实际上 QH 总是 16 字节对齐的，即它的低四位总是为 0，所以我们总是把低四位拿出来利用，比如这里的 LINK_TO_QH 就是把这个 struct uhci_qh 的 bit1 给设置成 1，以表明它指向的是一个 QH。）而 bit0 表示本 QH 是否是最后一个 QH，如果 bit0 位 1，则说明本 QH 是最后一个 QH 了，所以这个指针实际上是无效的，而 bit0 为 0 才表示本指针有效，因为至少 QH 后面还有 QH 或者还有 TD。我们看到 skel_async_qh 的 link 指针被赋予了 UHCI_PTR_TERM。

另外这里还为 skel_term_qh 的 link 给赋了值，我们看到它就指向自己。skel_term_qh 是 skelqa[]数组的第 10 个元素，其作用暂时还不明了。但以后自然会知道的。

struct uhci_td 里面同样也有个指针，__le32 link，它同样指向另一个 TD 或者 QH，而 bit1 和 bit0 的作用和 struct uhci_qh 中的 link 是一模一样的，bit1 为 1 表示指向 QH，为 0 表示指向 TD。bit0 为 1 表示指针无效，即本 TD 是最后一个 TD 了，bit0 为 0 表示指针有效。

639 行 uhci_fill_td，来自 drivers/usb/host/uhci-q.c：

```
138 static inline void uhci_fill_td(struct uhci_td *td, u32 status,
139                 u32 token, u32 buffer)
140 {
141     td->status = cpu_to_le32(status);
142     td->token = cpu_to_le32(token);
143     td->buffer = cpu_to_le32(buffer);
144 }
```

实际上就是填充 struct uhci_td 中的三个成员：__le32 status，__le32 token 和__le32 buffer。咱们来看传递给它的参数，status 和 buffer 都是 0，只有一个 token 不为 0，uhci_explen 来自 drivers/usb/host/uhci-hcd.h：

```
211 #define td_token(td)            le32_to_cpu((td)->token)
212 #define TD_TOKEN_DEVADDR_SHIFT  8
213 #define TD_TOKEN_TOGGLE_SHIFT   19
214 #define TD_TOKEN_TOGGLE         (1 << 19)
215 #define TD_TOKEN_EXPLEN_SHIFT   21
216 #define TD_TOKEN_EXPLEN_MASK  0x7FF /* expected length, encoded as n-1*/
217 #define TD_TOKEN_PID_MASK       0xFF
219 #define uhci_explen(len)    ((((len) - 1) & TD_TOKEN_EXPLEN_MASK) << \
220                                 TD_TOKEN_EXPLEN_SHIFT)
222 #define uhci_expected_length(token) ((((token) >> TD_TOKEN_EXPLEN_SHIFT) + \
223                                 1) & TD_TOKEN_EXPLEN_MASK)
224 #define uhci_toggle(token)      (((token) >> TD_TOKEN_TOGGLE_SHIFT) & 1)
225 #define uhci_endpoint(token)    (((token) >> 15) & 0xf)
226 #define uhci_devaddr(token) (((token) >> TD_TOKEN_DEVADDR_SHIFT) & 0x7f)
227 #define uhci_devep(token)  (((token) >> TD_TOKEN_DEVADDR_SHIFT) & 0x7ff)
228 #define uhci_packetid(token)    ((token) & TD_TOKEN_PID_MASK)
229 #define uhci_packetout(token)   (uhci_packetid(token) != USB_PID_IN)
230 #define uhci_packetin(token)    (uhci_packetid(token) == USB_PID_IN)
```

真是一波未平一波又起。麻烦的东西一个又一个地跳出来。让我一次次的感觉到心力交瘁。关于 token，UHCI spec 为 TD 定义了 4 个双字，即四个 DWord，其中第三个 DWord 叫做 TD TOKEN。一个 DWord 一共 32 个 bits。在这 32 个 bits 中，bit31 到 bit21 表示 Maximum Length，即这次传输的最大允许字节。bit20 是保留位，bit19 表示 Data Toggle，bit18 到 bit15 表示端点的地址，即我们曾经说的端点号；bit14 到 bit8 表示设备地址，bit7 到 bit0 表示 PID，即 Packet ID。以上这些宏就是为了提取出一个 Token 的各个部分，比如 uhci_toggle，就是 token 右移 19 位，然后和 1 相与，结果当然就是 token 的 bit19，正是刚才说的 Data Toggle。而 uhci_expected_length 则是获取 bit31 到 bit21，即 Length 这一段（加上 1 是因为 Spec 规定，这

一段为 0 表示 1 个 byte，为 1 表示 2 个 bytes，为 2 表示 3 个 bytes……）

于是我们很快就能明白这个 uhci_fill_td 具体做了什么。(0x7f << TD_TOKEN_DEVADDR_SHIFT)表示把 7f 左移 8 位，USB_PID_IN 等于 0x69，UHCI spec 规定这就表示 PID IN。然后 uhci_explen(len)的作用和 uhci_expected_length 的作用恰恰相反，它把一个 length 转换成 bit31 到 bit21，这样三块“或”一下，就构造了一个新的 token。

至于这个 token 构造好了之后填充给这 TD 究竟有什么用，我们看不出来，实际上注释说了，这是为了修复一个 Bug，若干年前，Intel PIIX 控制器有一个 Bug，当时为了绕开这个 Bug，引入了这么一段。

关于这个 Bug 的详情，我们在后面会讲，它和一个叫做 FSBR 的东西有关。只不过我们现在看到的是 term_td 的 link 指针被设置为了 UHCI_PTR_TERM，和 skel_term_qh 的 link 赋值一样，又是那个休止符。其实这里的道理很简单，就相当于我们每次申请一个链表时总是把最后一个指针设置成 NULL 一样。只不过这里不是叫作 NULL，是叫作 UHCI_PTR_TERM，但其作用都是一样，就相当于五线谱中的休止符。注意 uhci->term_td 正是我们一开始调用 uhci_alloc_td 时申请并且做的初始化。

642 行，struct uhci_qh 中另一个成员为__le32 element。它指向一个队列中的第一个元素。LINK_TO_TD 来自 drivesr/usb/host/uhci-hcd.h：

```
269 #define LINK_TO_TD(td)          (cpu_to_le32((td)->dma_handle))
```

理解了 LINK_TO_QH 自然就能理解 LINK_TO_TD。这里咱们令 skel_async_qh 和 skel_term_qh 的 element 等于这个 uhci->term_td。

649 行，这个循环可够夸张的，UHCI_NUMFRAMES 的值为 1024，所以这个循环就是惊世骇俗的 1024 次。仿佛写代码的人受了北京大学经济学院院长刘伟的熏陶。既然你刘伟说：“我把堵车看成是一个城市繁荣的标志，是一件值得欣喜的事情。如果一个城市没有堵车，那它的经济也可能凋零衰败。1998 年特大水灾刺激了需求，拉动增长，光水毁房屋就几百万间，所以水灾拉动中国经济增长 1.35%。”于是写代码的人说：“我把循环次数看成是一个程序高效的标志，是一件值得欣喜的事情。如果一个程序没有循环，那它的效率也可能惨不忍睹……”

9.4 用队列诠释调度

uhci_frame_skel_link 函数来自 drivers/usb/host/uhci-hcd.c：

```
97 static __le32 uhci_frame_skel_link(struct uhci_hcd *uhci, int frame)
98 {
99      int skelnum;
100
101     /*
102      * The interrupt queues will be interleaved as evenly as possible.
103      * There's not much to be done about period-1 interrupts; they have
104      * to occur in every frame.  But we can schedule period-2 interrupts
105      * in odd-numbered frames, period-4 interrupts in frames congruent
106      * to 2 (mod 4), and so on.  This way each frame only has two
107      * interrupt QHs, which will help spread out bandwidth utilization.
108      *
109      * ffs (Find First bit Set) does exactly what we need:
110      * 1,3,5,...  => ffs = 0 => use period-2 QH = skelqh[8],
111      * 2,6,10,... => ffs = 1 => use period-4 QH = skelqh[7], etc.
112      * ffs >= 7 => not on any high-period queue, so use
113      *     period-1 QH = skelqh[9].
114      * Add in UHCI_NUMFRAMES to insure at least one bit is set.
115      */
116     skelnum = 8 - (int) __ffs(frame | UHCI_NUMFRAMES);
117     if (skelnum <= 1)
118         skelnum = 9;
119     return LINK_TO_QH(uhci->skelqh[skelnum]);
120 }
```

俗话说，彪悍的人生不需要解释，彪悍的代码不需要注释。但是像这个函数这样，仅仅几行代码，却用了一堆的注释，不可谓不彪悍也。

首先__ffs 是一个位操作函数，其意思已经在注释里说得很清楚了，给它一个输入参数，它为你找到这

个输入参数的第一个被 set 的位，被 set 就是说为 1。这个函数涉及汇编代码，对我这个汇编语言不会编、微机原理闹危机的人来说，显然是不愿意仔细去看这个函数具体是怎么实现的，只是知道，每个体系结构都实现了自己的这个函数__ffs。比如，i386 的就在 include/asm-i386/bitops.h 中，而 x86 64 位的就在 include/asm-x86_64/bitops.h 中。

在 Intel 以结果为导向的理念指导下，我们来看一下这个函数的返回值，很显然，正如注释里说的那样，如果输入参数是 1，3，5，7，9 等奇数，那么返回值必然是 0，因为 bit0 肯定是 1。如果参数是 2，6，10，14，18，22，26 这一系列的数，那么返回值就是 1，因为 bit0 一定是 0，而 bit1 一定是 1。如果参数是 4，12，20，28，36 这一系列的数，那么返回值就是 2，因为 bit0 和 bit1 一定是 0，而 bit2 一定是 1。

不难看出，其实第一组数就是除以 2 余数为 1 的数列，第二组数就是除以 4 余数为 2 的数列，第三组就是除以 8 余数为 4 的数列，用当年奥赛的术语取模符号（mod）来说，就是第一组是 1 mod 2，第二组是 2 mod 4，第三组是 4 mod 8，如此下去，返回值为 0 的一共有 512 个，返回值为 1 的一共有 256 个，返回值为 2 的一共有 128 个，返回值为 3 的一共有 64 个，返回值为 4 的一共有 32 个，返回值为 5 的一共有 16 个，返回值为 6 的一共有 8 个，返回值为 7 的一共有 4 个，返回值为 8 的一共有 2 个，返回值为 9 的一共有 1 个(即 512)。*N* 年前我们就知道，1+2+4+…+512=1023。

结合咱们这里代码的 116 行，frame 为 0 的话，__ffs 的返回值为 10，所以 skelnum 就应该为-2，frame 为 1 到 1023 这个过程中，skelnum 为 8 的次数为 512，为 7 的次数为 256，为 6 的次数为 128，为 5 的次数为 64，为 4 的次数为 32，为 3 的次数为 16，为 2 的次数为 8，为 1 的次数为 4，为 0 的次数为 2，为-1 的次数为 1。而 117 行这个 if 语句就使得 skelnum 小于等于 1 的那几次都把 skelnum 设置为 9，这总共有 8 次。（为 1 的 4 次，为 0 的 2 次，为-1 的 1 次，为-2 的一次。）

因此我们就知道 skelnum 的取值范围是 2 到 9，而这也就意味着这里 uhci_frame_skel_link 函数的返回值实际上就是 uhci->skelqh[]这个数组中的 7 个元素。前面已经知道这个数组一共有 11 个元素，除了 skelqh[2]到 skelqh[9]这 8 个元素以外，skelqh[1]是为等时传输准备的，skel[10]是休止符（即 skel_term_qh），skel[0]表示没有连接的状态。

要深刻理解这个数组需要时间的沉淀，但是很明显，uhci_frame_skel_link 的效果就是为 skelqh[2]和 skelqh[9]找到了归宿。在 1024 个 frame 中，有 8 个 frame 指向了 skelqh[2]，即 skel int128 QH，1024 除以 8 等于 128，岂不正是每隔 128 ms 这个 qh 会被访问到么？同理，16 个 frame 指向了 skelqh[3]， 即 skel int64 QH，1024 除以 16 等于 64，也正意味着每隔 64 ms 会被访问到。一直到 skelqh[8]，即 skel int2 QH，有 512 个 frame 指向了它，所以这就代表每隔 2 ms 会被访问的那个队列。剩下的 skelqh[9]，即 skel int1 QH，总共也有 8 次，不过你别误会，skel int1 QH 代表的是 1 ms 周期，显然应该是 1024 个 frame 都指向它。可是你别忘了，刚才我们不是把 skel int2 QH 到 skel int8 QH 的 link 指针都指向了 skel int1 QH 了么？

还没明白？我们说了要用图解法来理解这个问题，所以不妨先画出此时此刻这整个框架。

```
framelist[]
[  0 ]----> QH -------\
[  1 ]----> QH --------> QH ----> UHCI_PTR_TERM
 …    ----> QH -------\
[1023]----> QH ------/
            ^^         ^^              ^^
          7 QHs for     1 QH for   f    End Chain
           INT (2-128 ms)  1 ms-INT(plus CTRL Chain,BULK Chain)
```

skel 实际上就是 Skeleton，框架或者骨骼的意思。skelqh 数组扮演着一个框架的作用。实际上一个 QH 对应着一个端点，即从主机控制器的角度来说，它为每一个端点建立一个队列，这就要求每个队列有一个队列头，而许多个端点的队列如何组织呢？正如上面显示的框架一样，有了一个端点，就为它建立相应的队列，根据需要来建立，然后把它插入到框架中的对应位置。

9.5 一个函数引发的故事（五）

接着走，661 行，configure_hc，来自 drivers/usb/host/uhci-hcd.c，

```
178 static void configure_hc(struct uhci_hcd *uhci)
```

```
179 {
180     /* Set the frame length to the default: 1  ms exactly */
181     outb(USBSOF_DEFAULT, uhci->io_addr + USBSOF);
182
183     /* Store the frame list base address */
184     outl(uhci->frame_dma_handle, uhci->io_addr + USBFLBASEADD);
185
186     /* Set the current frame number */
187     outw(uhci->frame_number & UHCI_MAX_SOF_NUMBER,
188                     uhci->io_addr + USBFRNUM);
189
190     /* Mark controller as not halted before we enable interrupts */
191     uhci_to_hcd(uhci)->state = HC_STATE_SUSPENDED;
192     mb();
193
194     /* Enable PIRQ */
195     pci_write_config_word(to_pci_dev(uhci_dev(uhci)), USBLEGSUP,
196                     USBLEGSUP_DEFAULT);
197 }
```

USBSOF_DEFAULT 和 USBSOF 定义于 drivers/usb/host/uhci-hcd.h：

```
43 #define USBFRNUM       6
44 #define USBFLBASEADD    8
45 #define USBSOF         12
46 #define   USBSOF_DEFAULT       64     /* Frame length is exactly 1  ms */
```

UHCI spec 中定义了一个 START OF FRAME(SOF) MODIFY REGISTER，这里称作 SOF 寄存器，其地址位于 Base+(0Ch)处，0Ch 即这里的 12。这个寄存器值的修改意味着 Frame 周期的改变，通常我们没有必要修改这个寄存器，直接设置为默认值 64 即可，按照 UHCI spec 中 2.1.6 的陈述，这意味着对于常见的 12MHz 的时钟输入的情况，Frame 周期将为 1 ms。(The default value is decimal 64 which gives a SOF cycle time of 12000. For a 12 MHz SOF counter clock input, this produces a 1 ms Frame period.)

紧接着 USBFLBASEADD 用来表示另一个寄存器，UHCI spec 中称之为 FLBASEADD， 即 Frame List 基地址寄存器，它一共有 32 个 bits，位于 Base+(08-0Bh)，这个寄存器应该包含 Frame List 在系统内存中的起始地址。其中，bit31 到 bit12 对应于内存地址信号[31：12]，而 bit11 到 bit0 则是保留位，必须全为 0。frame_dma_handle 正是前面调用 dma_alloc_coherent 为 uhci->frame 申请内存时映射的 DMA 地址，显然这个地址需要写到这个寄存器里，这样主机控制器才会知道去怎样访问这个 Frame List。

接下来，UHCI_MAX_SOF_NUMBER 是定义于 drivers/usb/host/uhci-hcd.h 的宏，值为 2047，用二进制来表示就是 11 个 1，即 111 1111 1111，而从 USBFRNUM 这里我们看到了是 6，它对应于 UHCI spec 中的寄存器 FRNUM（Frame Number Register），地址为 Base+(06-07h)，这个寄存器一共 16 个 bits，这其中 bit10 到 bit0 包含了当前的 Frame 号，所以把 uhci->frame_number 与 UHCI_MAX_SOF_NUMBER 相与就得到它的 bit10 到 bit0 这 11 个 bits，即得到 Frame 号然后写入到 FRNUM 寄存器中去。unsigned int frame_number 是 struct uhci_hcd 的一个成员。

然后，设置 uhci_to_hcd(uhci)->state 为 HC_STATE_SUSPENDED，注意我们当初在 finish_reset 中也有设置过这个状态，只不过当时是设置成了 HC_STATE_HALT。凡事都是有因有果的，我们做了这些设置，到时候自然会用到的。

195 行，pci_write_config_word，写寄存器，USBLEGSUP，不过这次写的是 USBLEGSUP_DEFAULT，这个宏的值为 0x2000，这是 UHCI spec 中规定的默认值。

这样我们就算是配置好了 HC，到这里就算万事俱备，只欠东风了。662 行就设置 uhci->is_initialized 为 1，这意图再明显不过了。

回到 uhci_start 中，还剩下最后一个函数，start_rh()，rh 表示 Root Hub。这个函数来自 drivers/usb/host/uhci-hcd.c：

```
324 static void start_rh(struct uhci_hcd *uhci)
325 {
326     uhci_to_hcd(uhci)->state = HC_STATE_RUNNING;
327     uhci->is_stopped = 0;
328
```

```
329     /* Mark it configured and running with a 64-byte max packet.
330      * All interrupts are enabled, even though RESUME won't do anything.
331      */
332     outw(USBCMD_RS | USBCMD_CF | USBCMD_MAXP, uhci->io_addr + USBCMD);
333     outw(USBINTR_TIMEOUT | USBINTR_RESUME | USBINTR_IOC | USBINTR_SP,
334                     uhci->io_addr + USBINTR);
335     mb();
336     uhci->rh_state = UHCI_RH_RUNNING;
337     uhci_to_hcd(uhci)->poll_rh = 1;
338 }
```

又一次设置了 uhci_to_hcd(uhci)->state，只不过这次设置的是 HC_STATE_RUNNING。然后设置 is_stopped 为 0。

然后是写寄存器 USBCMD，这次写的是什么呢？先看 drivers/usb/host/uhci-hcd.h 中关于这个命令寄存器定义的宏：

```
16 #define USBCMD          0
17 #define   USBCMD_RS           0x0001  /* Run/Stop */
18 #define   USBCMD_HCRESET       0x0002  /* Host reset */
19 #define   USBCMD_GRESET        0x0004  /* Global reset */
20 #define   USBCMD_EGSM          0x0008  /* Global Suspend Mode */
21 #define   USBCMD_FGR           0x0010  /* Force Global Resume */
22 #define   USBCMD_SWDBG         0x0020  /* SW Debug mode */
23 #define   USBCMD_CF            0x0040  /* Config Flag (sw only) */
24 #define   USBCMD_MAXP          0x0080  /* Max Packet (0 = 32, 1 = 64) */
```

结合 spec 和这里的注释来看，USBCMD_RS 表示 RUN/STOP，1 表示 RUN，0 表示 STOP。USBCMD_CF 表示 Configure Flag，在配置阶段结束时，应该把这个 flag 设置好。USBCMD_MAXP 则表示 FSBR 最大包的 size，这位为 1 表示 64bytes，为 0 表示 32bytes。关于 FSBR 我们以后会知道。

然后写另一个寄存器，USBINTR，表示中断使能寄存器。这个寄存器我们前面曾经提过。当时我们贴出了关于它的图片，知道它的 bit3，bit2，bit1，bit0 分别表示四个开关，为 1 就是使能，为 0 就是使不能，drivers/usb/host/uhci-hcd.h 中也定义了这些相关的宏：

```
37 #define USBINTR         4
38 #define   USBINTR_TIMEOUT      0x0001  /* Timeout/CRC error enable */
39 #define   USBINTR_RESUME       0x0002  /* Resume interrupt enable */
40 #define   USBINTR_IOC          0x0004  /* Interrupt On Complete enable */
41 #define   USBINTR_SP           0x0008  /* Short packet interrupt enable */
```

显而易见的是，咱们这里就是把这四个开关全部打开。这四种中断的意思都在注释里说得很清楚了：第一种是超时，第二种是从 Suspended 进入 Resume，第三种是完成了一个交易，第四种是接收到的包小于预期的长度。在这四种情况下，USB 主机控制器可以向系统主机或者说向系统的 CPU 发送中断请求。

最后设置 uhci->rh_state 为 UHCI_RH_RUNNINNG，并设置 uhci_to_hcd(uhci)->poll_rh 为 1。 到这里 uhci_start 就可以返回了，没什么意外的话就返回 0。于是咱们还是回到 usb_add_hcd 中去。

10、寂寞在唱歌

接下来就该是 usb_hcd_poll_rh_status 函数了，这个函数在咱们整个故事将出现多次，甚至可以说在任何一个 HCD 的故事中都将出现多次。为了继续走下去，我们必须做一个伟大的假设，假设现在 Root Hub 上还没有连接任何设备，也就是说此时此刻，USB 设备树上只有 Root Hub 形单影只。我们以此为上下文开始往下看。

usb_hcd_poll_rh_status 来自 drivers/usb/core/hcd.c：

```
541 void usb_hcd_poll_rh_status(struct usb_hcd *hcd)
542 {
543     struct urb     *urb;
544     int           length;
545     unsigned long  flags;
546     char          buffer[4];     /* Any root hubs with > 31 ports? */
547
548     if (unlikely(!hcd->rh_registered))
549         return;
```

```
550     if (!hcd->uses_new_polling && !hcd->status_urb)
551         return;
552
553     length = hcd->driver->hub_status_data(hcd, buffer);
554     if (length > 0) {
555
556         /* try to complete the status urb */
557         local_irq_save (flags);
558         spin_lock(&hcd_root_hub_lock);
559         urb = hcd->status_urb;
560         if (urb) {
561             spin_lock(&urb->lock);
562             if (urb->status == -EINPROGRESS) {
563                 hcd->poll_pending = 0;
564                 hcd->status_urb = NULL;
565             urb->status = 0;
566                 urb->hcpriv = NULL;
567                 urb->actual_length = length;
568                 memcpy(urb->transfer_buffer, buffer, length);
569             } else          /* urb has been unlinked */
570             length = 0;
571             spin_unlock(&urb->lock);
572         } else
573             length = 0;
574         spin_unlock(&hcd_root_hub_lock);
575
576         /* local irqs are always blocked in completions */
577         if (length > 0)
578             usb_hcd_giveback_urb (hcd, urb);
579         else
580             hcd->poll_pending = 1;
581         local_irq_restore (flags);
582     }
583
584     /* The USB 2.0 spec says 256  ms.  This is close enough and won't
585      * exceed that limit if HZ is 100. */
586     if (hcd->uses_new_polling ? hcd->poll_rh :
587                 (length == 0 && hcd->status_urb != NULL))
588         mod_timer (&hcd->rh_timer, jiffies +  msecs_to_jiffies(250));
589 }
```

这个函数天生是为了中断传输而活的。

前面两个 if 肯定是不满足的。rh_registered 刚刚才被设置为 1， uses_new_polling 也在 uhci_start()中设置为了 1。所以，继续昂首挺胸地往前走。

553 行，driver->hub_status_data 是每个驱动自己定义的，对于 UHCI 来说，定义了 uhci_hub_status_data 这么一个函数，它来自 drivers/usb/host/uhci-hub.c 中：

```
184 static int uhci_hub_status_data(struct usb_hcd *hcd, char *buf)
185 {
186     struct uhci_hcd *uhci = hcd_to_uhci(hcd);
187     unsigned long flags;
188     int status = 0;
189
190     spin_lock_irqsave(&uhci->lock, flags);
191
192     uhci_scan_schedule(uhci);
193     if (!test_bit(HCD_FLAG_HW_ACCESSIBLE, &hcd->flags) || uhci->dead)
194         goto done;
195     uhci_check_ports(uhci);
196
197     status = get_hub_status_data(uhci, buf);
198
199     switch (uhci->rh_state) {
200     case UHCI_RH_SUSPENDING:
201     case UHCI_RH_SUSPENDED:
202         /* if port change, ask to be resumed */
203         if (status)
204             usb_hcd_resume_root_hub(hcd);
205         break;
206
207     case UHCI_RH_AUTO_STOPPED:
208         /* if port change, auto start */
```

```
209         if (status)
210             wakeup_rh(uhci);
211         break;
212
213     case UHCI_RH_RUNNING:
214         /* are any devices attached? */
215         if (!any_ports_active(uhci)) {
216             uhci->rh_state = UHCI_RH_RUNNING_NODEVS;
217             uhci->auto_stop_time = jiffies + HZ;
218         }
219         break;
220
221     case UHCI_RH_RUNNING_NODEVS:
222         /* auto-stop if nothing connected for 1 second */
223         if (any_ports_active(uhci))
224             uhci->rh_state = UHCI_RH_RUNNING;
225         else if (time_after_eq(jiffies, uhci->auto_stop_time))
226             suspend_rh(uhci, UHCI_RH_AUTO_STOPPED);
227         break;
228
229     default:
230         break;
231     }
232
233 done:
234     spin_unlock_irqrestore(&uhci->lock, flags);
235     return status;
236 }
```

坦白说，这个函数一下子就把咱们这个故事的技术含量给拉高了上去。尤其是那个 uhci_scan_schedule，一下子就让故事变得复杂了起来，原本清晰的故事，渐渐变得模糊。

uhci_scan_schedule 来自 drivers/usb/host/uhci-q.c：

```
1708 static void uhci_scan_schedule(struct uhci_hcd *uhci)
1709 {
1710     int i;
1711     struct uhci_qh *qh;
1712
1713     /* Don't allow re-entrant calls */
1714     if (uhci->scan_in_progress) {
1715         uhci->need_rescan = 1;
1716         return;
1717     }
1718     uhci->scan_in_progress = 1;
1719 rescan:
1720     uhci->need_rescan = 0;
1721     uhci->fsbr_is_wanted = 0;
1722
1723     uhci_clear_next_interrupt(uhci);
1724     uhci_get_current_frame_number(uhci);
1725     uhci->cur_iso_frame = uhci->frame_number;
1726
1727     /* Go through all the QH queues and process the URBs in each one */
1728     for (i = 0; i < UHCI_NUM_SKELQH - 1; ++i) {
1729         uhci->next_qh = list_entry(uhci->skelqh[i]->node.next,
1730                         struct uhci_qh, node);
1731         while ((qh = uhci->next_qh) != uhci->skelqh[i]) {
1732             uhci->next_qh = list_entry(qh->node.next,
1733                             struct uhci_qh, node);
1734
1735             if (uhci_advance_check(uhci, qh)) {
1736                 uhci_scan_qh(uhci, qh);
1737                 if (qh->state == QH_STATE_ACTIVE) {
1738                     uhci_urbp_wants_fsbr(uhci,
1739                   list_entry(qh->queue.next, struct urb_priv, node));
1740                 }
1741             }
1742         }
1743     }
1744
1745     uhci->last_iso_frame = uhci->cur_iso_frame;
1746     if (uhci->need_rescan)
1747         goto rescan;
1748     uhci->scan_in_progress = 0;
```

```
1749
1750    if (uhci->fsbr_is_on && !uhci->fsbr_is_wanted &&
1751                    !uhci->fsbr_expiring) {
1752        uhci->fsbr_expiring = 1;
1753        mod_timer(&uhci->fsbr_timer, jiffies + FSBR_OFF_DELAY);
1754    }
1755
1756    if (list_empty(&uhci->skel_unlink_qh->node))
1757        uhci_clear_next_interrupt(uhci);
1758    else
1759        uhci_set_next_interrupt(uhci);
1760 }
```

这个函数的复杂性让我哭都哭不出来。我做梦也没想到这个函数竟然会牵出一打函数来。

scan_in_progress 初值为 0，只有在这个函数中才会改变它的值。因为它本来就是被用来表征我们处于这个函数中，注释中也说了，使用这个变量的目的就是为了避免这个函数被嵌套调用。所以如果这里判断为 0，则 1718 行就立刻设置它为 1。反之如果已经不为 0 了，就设置 need_rescan 为 1，并且函数返回。

1720 行和 1721 行，设置 need_rescan 和 fsbr_is_wanted 都为 0。

1723 行，uhci_clear_next_interrupt()来自 drivers/usb/host/uhci-q.c：

```
35 static inline void uhci_clear_next_interrupt(struct uhci_hcd *uhci)
36 {
37     uhci->term_td->status &= ~cpu_to_le32(TD_CTRL_IOC);
38 }
```

清中断。如果一个 TD 设置了 TD_CTRL_IOC 这个 flag，就表示该 TD 所在的 Frame 结束之后，USB 主机控制器将向 CPU 发送一次中断。在这里我们实际上不希望 term_td 结束所在的 Frame 发生任何中断，因为我们现在正在处理整个调度队列。

而接下来的另一个函数 uhci_get_current_frame_number()则来自 drivers/usb/host/uhci-hcd.c：

```
441 static void uhci_get_current_frame_number(struct uhci_hcd *uhci)
442 {
443     if (!uhci->is_stopped) {
444         unsigned delta;
445
446         delta = (inw(uhci->io_addr + USBFRNUM) - uhci->frame_number) &
447                         (UHCI_NUMFRAMES - 1);
448         uhci->frame_number += delta;
449     }
450 }
```

正如注释里说的那样，把寄存器中的值更新给 uhci->frame_number。

我们结合 1729 行和 1731 行来看，注意到这里判断的就是 uhci->skelqh[]数组的每个成员的 node 队列。我们知道 struct uhci_qh 结构体有一个成员是 node，它代表的是一支队伍，在 uhci_alloc_qh 中我们事实上用 INIT_LIST_HEAD 宏初始化了这个队列，这个宏就是初始化一个空队列，即一个节点的 next 和 prev 指针都指向自己。所以很显然，uhci->next_qh 就等于 uhci->skelqh[i]。于是 1731 这个 while 循环就不会执行，因此，1728 开始的这个 for 循环也就没有什么作用。或者至少说，现在还不是它做贡献的时刻。待到山花浪漫时，skelqh[]的 node 队列有内容了，我们自然还会再次回来看这个函数。

飘过了这个 for 循环 uhci_scan_schedule 函数。1748 行又把 scan_in_progress 设置为 0。

1750 行自然也不用说，fsbr_is_on 默认也是 0。所以暂时这里也不会执行。

至于 1756 行这段 if-else，skel_unlink_qh 实际上就是 skelqh[0]，同样，此时此刻它的 node 队列也是空的，故 list_empty 是满足的，因此 uhci_clear_next_interrupt 会再次被调用。

结束了 uhci_scan_schedule，我们继续回到 uhci_hub_status_data 中来。

193 行，HCD_FLAG_HW_ACCESSIBLE 这个 flag 我们是设置过的，就在 usb_add_hcd 中设置的。而 uhci->dead 在 finish_reset 中设置为 0。

接下来 uhci_check_ports 函数，来自 drivers/usb/host/uhci-hub.c：

```
136 static void uhci_check_ports(struct uhci_hcd *uhci)
```

```
137 {
138     unsigned int port;
139     unsigned long port_addr;
140     int status;
141
142     for (port = 0; port < uhci->rh_numports; ++port) {
143         port_addr = uhci->io_addr + USBPORTSC1 + 2 * port;
144         status = inw(port_addr);
145         if (unlikely(status & USBPORTSC_PR)) {
146             if (time_after_eq(jiffies, uhci->ports_timeout)) {
147                 CLR_RH_PORTSTAT(USBPORTSC_PR);
148                 udelay(10);
149
150                 /* HP's server management chip requires
151                  * a longer delay. */
152                 if (to_pci_dev(uhci_dev(uhci))->vendor ==
153                                         PCI_VENDOR_ID_HP)
154                     wait_for_HP(port_addr);
155
156                 /* If the port was enabled before, turning
157                  * reset on caused a port enable change.
158                  * Turning reset off causes a port connect
159                  * status change.  Clear these changes. */
160                 CLR_RH_PORTSTAT(USBPORTSC_CSC | USBPORTSC_PEC);
161                 SET_RH_PORTSTAT(USBPORTSC_PE);
162             }
163         }
164         if (unlikely(status & USBPORTSC_RD)) {
165             if (!test_bit(port, &uhci->resuming_ports)) {
166
167                 /* Port received a wakeup request */
168                 set_bit(port, &uhci->resuming_ports);
169                 uhci->ports_timeout = jiffies +
170                                         msecs_to_jiffies(20);
171
172                 /* Make sure we see the port again
173                  * after the resuming period is over. */
174                 mod_timer(&uhci_to_hcd(uhci)->rh_timer,
175                                         uhci->ports_timeout);
176             } else if (time_after_eq(jiffies,
177                                         uhci->ports_timeout)) {
178                 uhci_finish_suspend(uhci, port, port_addr);
179             }
180         }
181     }
182 }
```

这个函数倒是挺清晰的，遍历 Root Hub 的各个端口，读取它们对应的端口寄存器。和这个端口寄存器相关的宏又是很多，来自 drivers/usb/host/uhci-hcd.h：

```
49 #define USBPORTSC1     16
50 #define USBPORTSC2     18
51 #define   USBPORTSC_CCS        0x0001  /* Current Connect Status
52                                         * ("device present") */
53 #define   USBPORTSC_CSC        0x0002  /* Connect Status Change */
54 #define   USBPORTSC_PE         0x0004  /* Port Enable */
55 #define   USBPORTSC_PEC        0x0008  /* Port Enable Change */
56 #define   USBPORTSC_DPLUS      0x0010  /* D+ high (line status) */
57 #define   USBPORTSC_DMINUS     0x0020  /* D- high (line status) */
58 #define   USBPORTSC_RD         0x0040  /* Resume Detect */
59 #define   USBPORTSC_RES1       0x0080  /* reserved, always 1 */
60 #define   USBPORTSC_LSDA       0x0100  /* Low Speed Device Attached */
61 #define   USBPORTSC_PR         0x0200  /* Port Reset */
62 /* OC and OCC from Intel 430TX and later (not UHCI 1.1d spec) */
63 #define   USBPORTSC_OC         0x0400  /* Over Current condition */
64 #define   USBPORTSC_OCC        0x0800  /* Over Current Change R/WC */
65 #define   USBPORTSC_SUSP       0x1000  /* Suspend */
66 #define   USBPORTSC_RES2       0x2000  /* reserved, write zeroes */
67 #define   USBPORTSC_RES3       0x4000  /* reserved, write zeroes */
68 #define   USBPORTSC_RES4       0x8000  /* reserved, write zeroes */
```

首先要看的是状态位 USBPORTSC_PR，为 1 表示此时此刻该端口正处于 reset 的状态。

其次我们看状态位 USBPORTSC_RD，这位为 1 表示端口探测到了 resume 信号。

显然以上两种情况都不是我们需要考虑的，至少不是现在需要考虑的。

于是下一个函数，get_hub_status_data，来自 drivers/usb/host/uhci-hub.c：

```
55 static inline int get_hub_status_data(struct uhci_hcd *uhci, char *buf)
56 {
57      int port;
58      int mask = RWC_BITS;
59
60      /* Some boards (both VIA and Intel apparently) report bogus
61       * overcurrent indications, causing massive log spam unless
62       * we completely ignore them.  This doesn't seem to be a problem
63       * with the chipset so much as with the way it is connected on
64       * the motherboard; if the overcurrent input is left to float
65       * then it may constantly register false positives. */
66      if (ignore_oc)
67          mask &= ~USBPORTSC_OCC;
68
69      *buf = 0;
70      for (port = 0; port < uhci->rh_numports; ++port) {
71          if ((inw(uhci->io_addr + USBPORTSC1 + port * 2) & mask) ||
72                      test_bit(port, &uhci->port_c_suspend))
73              *buf |= (1 << (port + 1));
74      }
75      return !!*buf;
76 }
```

这里 RWC_BITS 就是用来屏蔽端口寄存器中的 RWC 的 bits。这三个都是状态改变位。

这里的 ignore_oc 又是一个模块参数，uhci-hcd 和 ehci-hcd 这两个模块都会使用这个参数。注释里说得很清楚为何要用这个，有些主板喜欢谎报军情，对于这种情况，可以使用一个 ignore_oc 参数来忽略之。

接下来又是遍历端口，读取每个端口的寄存器，如果有戏，就把信息存在 buf 中，直到这时我们才注意到 usb_hcd_poll_rh_status 函数中定义了一个 char buffer[4]，这个 buffer 被一次次地传递下来。buf 一共 32 个 bits，这里凡是一个端口的寄存器里有东西（除了状态改变位以外），就在 buf 里把相应的位设置为 1。如果设置了 port_c_suspend 也需要这样，port_c_suspend 到时候我们在电源管理部分会看到，现在当然没有被设置。

不过这个函数最酷的就是最后这句居然有两个感叹号。这保证返回值要么是 0，要么是非 0。

接下来判断 uhci->rh_state 了，我们前面在 start_rh 中设置了它为 UHCI_RH_RUNNING， 所以这里就是执行 any_ports_active。

这个 any_ports_active 也来自 drivers/usb/host/uhci-hub.c：

```
42 static int any_ports_active(struct uhci_hcd *uhci)
43 {
44      int port;
45
46      for (port = 0; port < uhci->rh_numports; ++port) {
47          if ((inw(uhci->io_addr + USBPORTSC1 + port * 2) &
48                      (USBPORTSC_CCS | RWC_BITS)) ||
49                  test_bit(port, &uhci->port_c_suspend))
50              return 1;
51      }
52      return 0;
53 }
```

这时再次读端口寄存器。其中 CCS 表示端口连接有变化。我们假设现在没有变化。那么这里什么也不干，直接返回 0。这样 uhci_hub_status_data 最终返回 status。这个 status 正是 get_hub_status_data 的返回值，即那个要么是 0，要么是非 0 的。

如果为 0，那么很好办，直接“凌波微步”来到 586 行，判断 hcd->uses_new_polling，咱们在 uhci_start 中设置为了 1。所以这里继续判断 hcd->poll_rh，在 start_rh 中也把它设置为了 1。所以，这里 mod_timer 会被执行。这个函数意味着时间到了某件事情就会发生，咱们曾经在 usb_create_hcd 中初始化过 hcd->rh_timer，并且为它绑定了函数 rh_timer_func，所以不妨来看一下 rh_timer_func，来自 drivers/usb/core/hcd.c：

```
593 static void rh_timer_func (unsigned long _hcd)
594 {
595     usb_hcd_poll_rh_status((struct usb_hcd *) _hcd);
596 }
```

原来就是调用 usb_hcd_poll_rh_status。所以 usb_hcd_poll_rh_status 函数是一个被多次调用的函数，只不过多次之间是有个延时的，而咱们这里调用 mod_timer 设置的是 250 ms。而每次所做的就是去询问 Root Hub 的状态。实际上这就是 poll 的含义——轮询。

那么咱们这个故事基本上就结束了。我们能看到的是 usb_hcd_poll_rh_status 这个函数，每隔 250 ms 这样被执行一遍，重复一遍又一遍，可是它忙忙碌碌却什么也不做，但即便如此也比我要好，要知道我的人生就像复印机，每天都在不停的重复，可问题是有时候还他妈的卡纸。

11、Root Hub 的控制传输

虽然最伟大的 probe 函数就这样结束了。但是，我们的道路还很长，困难还很多，最终的结局是未知数。

在剩下的篇幅中，我们将围绕 usb_submit_urb()函数展开。子曾经曰过：不吃饭的女人这世上也许还有好几个，不吃醋的女人却连一个也没有。我也曾经曰过：不遵循 USB spec 的 USB 设备这世上也许还有好几个，不调用 usb_submit_urb()的 USB 设备驱动却连一个也没有。想必一路走来的兄弟们早就想知道神秘的 usb_submit_urb()函数究竟是怎么“混”的吧？

不管是控制传输，还是中断传输，或是批量传输，又或者等时传输，设备驱动都一定会调用 usb_submit_urb 函数，只有通过它才能提交 urb。所以接下来就分类来看这个函数，查看四种传输分别是如何处理的。

不过我们仍然先假设还没有设备插入 Root Hub 吧。因为 Root Hub 始终是一个特殊的角色，它的特殊地位决定了我们必须特殊对待。Hub 只涉及两种传输方式：控制传输和中断传输。我们先来看控制传输，确切地说是先看 Root Hub 的控制传输。

还记得刚才在 register_root_hub 中那个 usb_get_device_descriptor 么？在 Hub 驱动中讲过，它会调用 usb_get_descriptor，而后者会调用 usb_control_ msg，而 usb_control_ msg 则调用 usb_internal_control_ msg，然后 usb_start_wait_urb 会被调用，但最终会被调用的是 usb_submit_urb()。于是我们就来看一下 usb_submit_urb()究竟何德何能让大家如此景仰，我们来看这个设备描述符究竟是如何获得的。

这个函数显然分量比较重，它来自 drivers/usb/core/urb.c：

```
220 int usb_submit_urb(struct urb *urb, gfp_t mem_flags)
221 {
222     int                 pipe, temp, max;
223     struct usb_device       *dev;
224     int                 is_out;
225
226     if (!urb || urb->hcpriv || !urb->complete)
227         return -EINVAL;
228     if (!(dev = urb->dev) ||
229         (dev->state < USB_STATE_DEFAULT) ||
230         (!dev->bus) || (dev->devnum <= 0))
231         return -ENODEV;
232     if (dev->bus->controller->power.power_state.event != PM_EVENT_ON
233                 || dev->state == USB_STATE_SUSPENDED)
234         return -EHOSTUNREACH;
235
236     urb->status = -EINPROGRESS;
237     urb->actual_length = 0;
238
239     /* Lots of sanity checks, so HCDs can rely on clean data
240      * and don't need to duplicate tests
241      */
242     pipe = urb->pipe;
243     temp = usb_pipetype(pipe);
244     is_out = usb_pipeout(pipe);
```

```
245
246     if (!usb_pipecontrol(pipe) && dev->state < USB_STATE_CONFIGURED)
247         return -ENODEV;
248
249     /* FIXME there should be a sharable lock protecting us against
250      * config/altsetting changes and disconnects, kicking in here.
251      * (here == before maxpacket, and eventually endpoint type,
252      * checks get made.)
253      */
254
255     max = usb_maxpacket(dev, pipe, is_out);
256     if (max <= 0) {
257         dev_dbg(&dev->dev,
258             "bogus endpoint ep%d%s in %s (bad maxpacket %d)\n",
259                 usb_pipeendpoint(pipe), is_out ? "out" : "in",
260                 __FUNCTION__, max);
261         return -EMSGSIZE;
262     }
263
264     /* periodic transfers limit size per frame/uframe,
265      * but drivers only control those sizes for ISO.
266      * while we're checking, initialize return status.
267      */
268     if (temp == PIPE_ISOCHRONOUS) {
269         int    n, len;
270
271         /* "high bandwidth" mode, 1-3 packets/uframe? */
272         if (dev->speed == USB_SPEED_HIGH) {
273             int    mult = 1 + ((max >> 11) & 0x03);
274             max &= 0x07ff;
275             max *= mult;
276         }
277
278         if (urb->number_of_packets <= 0)
279             return -EINVAL;
280         for (n = 0; n < urb->number_of_packets; n++) {
281             len = urb->iso_frame_desc[n].length;
282             if (len < 0 || len > max)
283                 return -EMSGSIZE;
284             urb->iso_frame_desc[n].status = -EXDEV;
285             urb->iso_frame_desc[n].actual_length = 0;
286         }
287     }
288
289     /* the I/O buffer must be mapped/unmapped, except when length=0 */
290     if (urb->transfer_buffer_length < 0)
291         return -EMSGSIZE;
292
293 #ifdef DEBUG
294     /* stuff that drivers shouldn't do, but which shouldn't
295      * cause proble ms in HCDs if they get it wrong.
296      */
297     {
298     unsigned int    orig_flags = urb->transfer_flags;
299     unsigned int    allowed;
300
301     /* enforce simple/standard policy */
302     allowed = (URB_NO_TRANSFER_DMA_MAP | URB_NO_SETUP_DMA_MAP |
303                     URB_NO_INTERRUPT);
304     switch (temp) {
305     case PIPE_BULK:
306         if (is_out)
307             allowed |= URB_ZERO_PACKET;
308         /* FALLTHROUGH */
309     case PIPE_CONTROL:
310         allowed |= URB_NO_FSBR; /* only affects UHCI */
311         /* FALLTHROUGH */
312     default:                    /* all non-iso endpoints */
313         if (!is_out)
314             allowed |= URB_SHORT_NOT_OK;
315         break;
316     case PIPE_ISOCHRONOUS:
317         allowed |= URB_ISO_ASAP;
318         break;
319     }
```

```
320     urb->transfer_flags &= allowed;
321
322     /* fail if submitter gave bogus flags */
323     if (urb->transfer_flags != orig_flags) {
324         err("BOGUS urb flags, %x --> %x",
325                 orig_flags, urb->transfer_flags);
326         return -EINVAL;
327     }
328     }
329 #endif
330     /*
331      * Force periodic transfer intervals to be legal values that are
332      * a power of two (so HCDs don't need to).
333      *
334      * FIXME want bus->{intr,iso}_sched_horizon values here.  Each HC
335      * supports different values... this uses EHCI/UHCI defaults (and
336      * EHCI can use smaller non-default values).
337      */
338     switch (temp) {
339     case PIPE_ISOCHRONOUS:
340     case PIPE_INTERRUPT:
341         /* too small? */
342         if (urb->interval <= 0)
343             return -EINVAL;
344         /* too big? */
345         switch (dev->speed) {
346         case USB_SPEED_HIGH:    /* units are microframes */
347             // NOTE usb handles 2^15
348             if (urb->interval > (1024 * 8))
349                 urb->interval = 1024 * 8;
350             temp = 1024 * 8;
351             break;
352         case USB_SPEED_FULL:    /* units are frames/ msec */
353         case USB_SPEED_LOW:
354             if (temp == PIPE_INTERRUPT) {
355                 if (urb->interval > 255)
356                     return -EINVAL;
357                 // NOTE ohci only handles up to 32
358                 temp = 128;
359             } else {
360                 if (urb->interval > 1024)
361                     urb->interval = 1024;
362                 // NOTE usb and ohci handle up to 2^15
363                 temp = 1024;
364             }
365             break;
366         default:
367             return -EINVAL;
368         }
369         /* power of two? */
370         while (temp > urb->interval)
371             temp >>= 1;
372         urb->interval = temp;
373     }
374
375     return usb_hcd_submit_urb(urb, mem_flags);
376 }
```

天哪，这个函数绝对够让你我看得“七窍流血”。这种变态已经不能用语言来形容了，鲁迅先生看了一定会说我已经出离愤怒了！南唐的李煜在看完这段代码之后感慨道：问君能有几多愁，恰似太监上青楼！

这个函数的核心变量就是那个 temp。很明显，它表示的就是传输管道的类型。我们说了现在考虑的是 Root Hub 的控制传输。那么很明显的事实是，usb_hcd_submit_urb 会被调用，而 268 行这个 if 语段和 338 行这个 switch 都没有什么意义。所以我们来看 usb_hcd_submit_urb 吧，来自 drivers/usb/core/hcd.c：

```
921 int usb_hcd_submit_urb (struct urb *urb, gfp_t mem_flags)
922 {
923     int                 status;
924     struct usb_hcd        *hcd = bus_to_hcd(urb->dev->bus);
925     struct usb_host_endpoint *ep;
926     unsigned long         flags;
927
928     if (!hcd)
929         return -ENODEV;
```

```
930
931     usbmon_urb_submit(&hcd->self, urb);
932
933     /*
934      * Atomically queue the urb,  first to our records, then to the HCD.
935      * Access to urb->status is controlled by urb->lock ... changes on
936      * i/o completion (normal or fault) or unlinking.
937      */
938
939     // FIXME:  verify that quiescing hc works right (RH cleans up)
940
941     spin_lock_irqsave (&hcd_data_lock, flags);
942     ep = (usb_pipein(urb->pipe) ? urb->dev->ep_in : urb->dev->ep_out)
943                     [usb_pipeendpoint(urb->pipe)];
944     if (unlikely (!ep))
945         status = -ENOENT;
946     else if (unlikely (urb->reject))
947              status = -EPERM;
948     else switch (hcd->state) {
949     case HC_STATE_RUNNING:
950     case HC_STATE_RESUMING:
951 doit:
952         list_add_tail (&urb->urb_list, &ep->urb_list);
953         status = 0;
954         break;
955     case HC_STATE_SUSPENDED:
956         /* HC upstream links (register access, wakeup signaling) can work
957          * even when the downstream links (and DMA etc) are quiesced; let
958          * usbcore talk to the root hub.
959          */
960         if(hcd->self.controller->power.power_state.event==PM_EVENT_ON
961                     && urb->dev->parent == NULL)
962             goto doit;
963         /* FALL THROUGH */
964     default:
965         status = -ESHUTDOWN;
966         break;
967     }
968     spin_unlock_irqrestore (&hcd_data_lock, flags);
969     if (status) {
970         INIT_LIST_HEAD (&urb->urb_list);
971         usbmon_urb_submit_error(&hcd->self, urb, status);
972         return status;
973     }
974
975     /* increment urb's reference count as part of giving it to the HCD
976      * (which now controls it).  HCD guarantees that it either returns
977      * an error or calls giveback(), but not both.
978      */
979     urb = usb_get_urb (urb);
980     atomic_inc (&urb->use_count);
981
982     if (urb->dev == hcd->self.root_hub) {
983         /* NOTE:  requirement on hub callers (usbfs and the hub
984          * driver, for now) that URBs' urb->transfer_buffer be
985          * valid and usb_buffer_{sync,unmap}() not be needed, since
986          * they could clobber root hub response data.
987          */
988         status = rh_urb_enqueue (hcd, urb);
989         goto done;
990     }
991
992     /* lower level hcd code should use *_dma exclusively,
993      * unless it uses pio or talks to another transport.
994      */
995     if (hcd->self.uses_dma) {
996         if (usb_pipecontrol (urb->pipe)
997                 && !(urb->transfer_flags & URB_NO_SETUP_DMA_MAP))
998             urb->setup_dma = dma_map_single (
999                             hcd->self.controller,
1000                                 urb->setup_packet,
1001                                 sizeof (struct usb_ctrlrequest),
1002                                 DMA_TO_DEVICE);
1003             if (urb->transfer_buffer_length != 0
1004                 && !(urb->transfer_flags & URB_NO_TRANSFER_DMA_MAP))
```

```
1005                 urb->transfer_dma = dma_map_single (
1006                                 hcd->self.controller,
1007                                 urb->transfer_buffer,
1008                                 urb->transfer_buffer_length,
1009                                 usb_pipein (urb->pipe)
1010                                     ? DMA_FROM_DEVICE
1011                                     : DMA_TO_DEVICE);
1012     }
1013
1014     status = hcd->driver->urb_enqueue (hcd, ep, urb, mem_flags);
1015 done:
1016     if (unlikely (status)) {
1017         urb_unlink (urb);
1018         atomic_dec (&urb->use_count);
1019         if (urb->reject)
1020             wake_up (&usb_kill_urb_queue);
1021         usbmon_urb_submit_error(&hcd->self, urb, status);
1022         usb_put_urb (urb);
1023     }
1024     return status;
1025 }
```

凡是名字中带着 usbmon 的函数都甭管，它是一个 USB 的监控工具，启用与否取决于一个编译选项：CONFIG_USB_MON，咱们假设不打开它，这样它的这些函数实际上就都是些空函数。就比如 931 行的 usbmon_urb_submit，以及下面的这个 usbmon_urb_submit_error。

942 行得到与这个 urb 相关的 struct usb_host_endpoint 结构体指针 ep。事实上 struct urb 和 struct usb_host_endpoint 这两个结构体中都有一个成员 struct list_head urb_list，每个端点都维护着一个队列，所有与它相关的 urb 都被放入到这个队列中，而 952 行所做的就是这件事。当然，之所以我们现在会执行 952 行，是因为我们的 hcd->state 在 start_rh 中被设置成了 HC_STATE_RUNNING。

接着，我们发现，对于 Root Hub， rh_urb_enqueue 会被执行；对于非 Root Hub，即一般的 Hub，driver->urb_enqueue 会被执行；对于 UHCI 来说，就是 uhci_urb_enqueue 会被执行。先来看 Root Hub。

rh_urb_enqueue 来自 drivers/usb/core/hcd.c：

```
629 static int rh_urb_enqueue (struct usb_hcd *hcd, struct urb *urb)
630 {
631     if (usb_pipeint (urb->pipe))
632         return rh_queue_status (hcd, urb);
633     if (usb_pipecontrol (urb->pipe))
634         return rh_call_control (hcd, urb);
635     return -EINVAL;
636 }
```

12、Root Hub 的控制传输（二）

“医生，请把孩子取出来之后，顺便给我吸吸脂。”广州一妇女在剖腹产手术前对医生说。

对于控制传输，rh_call_control 会被调用。我也特别希望能有人给这个函数吸吸脂。我们的上下文是为了获取设备描述符，即当初那个 usb_get_device_descriptor 领着我们来到了这个函数，为了完成这件事情，实际上只需要很少的代码，但是 rh_call_control 这个函数涉及了所有与 Root Hub 相关的控制传输，以至于我们不得不顺便查看其他代码。当然了，既然是顺便，那么我们也就不会详细的去讲解每一行。这个函数定义于 drivers/usb/core/hcd.c：

```
344 static int rh_call_control (struct usb_hcd *hcd, struct urb *urb)
345 {
346     struct usb_ctrlrequest *cmd;
347     u16             typeReq, wValue, wIndex, wLength;
348     u8              *ubuf = urb->transfer_buffer;
349     u8              tbuf [sizeof (struct usb_hub_descriptor)]
350             __attribute__((aligned(4)));
351     const u8        *bufp = tbuf;
352     int             len = 0;
353     int             patch_wakeup = 0;
354     unsigned long   flags;
355     int             status = 0;
```

```
356     int           n;
357
358     cmd = (struct usb_ctrlrequest *) urb->setup_packet;
359     typeReq  = (cmd->bRequestType << 8) | cmd->bRequest;
360     wValue   = le16_to_cpu (cmd->wValue);
361     wIndex   = le16_to_cpu (cmd->wIndex);
362     wLength  = le16_to_cpu (cmd->wLength);
363
364     if (wLength > urb->transfer_buffer_length)
365         goto error;
366
367     urb->actual_length = 0;
368     switch (typeReq) {
369
370     /* DEVICE REQUESTS */
371
372     /* The root hub's remote wakeup enable bit is implemented using
373      * driver model wakeup flags.  If this system supports wakeup
374      * through USB, userspace may change the default "allow wakeup"
375      * policy through sysfs or these calls.
376      *
377      * Most root hubs support wakeup from downstream devices, for
378      * runtime power management (disabling USB clocks and reducing
379      * VBUS power usage).  However, not all of them do so; silicon,
380      * board, and BIOS bugs here are not uncommon, so these can't
381      * be treated quite like external hubs.
382      *
383      * Likewise, not all root hubs will pass wakeup events upstream,
384      * to wake up the whole system.  So don't assume root hub and
385      * controller capabilities are identical.
386      */
387
388     case DeviceRequest | USB_REQ_GET_STATUS:
389         tbuf [0] = (device_may_wakeup(&hcd->self.root_hub->dev)
390                                 << USB_DEVICE_REMOTE_WAKEUP)
391                             | (1 << USB_DEVICE_SELF_POWERED);
392         tbuf [1] = 0;
393         len = 2;
394         break;
395     case DeviceOutRequest | USB_REQ_CLEAR_FEATURE:
396         if (wValue == USB_DEVICE_REMOTE_WAKEUP)
397             device_set_wakeup_enable(&hcd->self.root_hub->dev, 0);
398         else
399             goto error;
400         break;
401     case DeviceOutRequest | USB_REQ_SET_FEATURE:
402         if (device_can_wakeup(&hcd->self.root_hub->dev)
403                         && wValue == USB_DEVICE_REMOTE_WAKEUP)
404             device_set_wakeup_enable(&hcd->self.root_hub->dev, 1);
405         else
406             goto error;
407         break;
408     case DeviceRequest | USB_REQ_GET_CONFIGURATION:
409         tbuf [0] = 1;
410         len = 1;
411         /* FALLTHROUGH */
412     case DeviceOutRequest | USB_REQ_SET_CONFIGURATION:
413         break;
414     case DeviceRequest | USB_REQ_GET_DESCRIPTOR:
415         switch (wValue & 0xff00) {
416         case USB_DT_DEVICE << 8:
417             if (hcd->driver->flags & HCD_USB2)
418                 bufp = usb2_rh_dev_descriptor;
419             else if (hcd->driver->flags & HCD_USB11)
420                 bufp = usb11_rh_dev_descriptor;
421             else
422                 goto error;
423             len = 18;
424             break;
425         case USB_DT_CONFIG << 8:
426             if (hcd->driver->flags & HCD_USB2) {
427                 bufp = hs_rh_config_descriptor;
428                 len = sizeof hs_rh_config_descriptor;
429             } else {
430                 bufp = fs_rh_config_descriptor;
```

```
431                 len = sizeof fs_rh_config_descriptor;
432             }
433             if (device_can_wakeup(&hcd->self.root_hub->dev))
434                 patch_wakeup = 1;
435             break;
436         case USB_DT_STRING << 8:
437             n = rh_string (wValue & 0xff, hcd, ubuf, wLength);
438             if (n < 0)
439                 goto error;
440             urb->actual_length = n;
441             break;
442         default:
443             goto error;
444         }
445         break;
446     case DeviceRequest | USB_REQ_GET_INTERFACE:
447         tbuf [0] = 0;
448         len = 1;
449         /* FALLTHROUGH */
450     case DeviceOutRequest | USB_REQ_SET_INTERFACE:
451         break;
452     case DeviceOutRequest | USB_REQ_SET_ADDRESS:
453         // wValue == urb->dev->devaddr
454         dev_dbg (hcd->self.controller, "root hub device address %d\n",
455                     wValue);
456         break;
457
458     /* INTERFACE REQUESTS (no defined feature/status flags) */
459
460     /* ENDPOINT REQUESTS */
461
462     case EndpointRequest | USB_REQ_GET_STATUS:
463         // ENDPOINT_HALT flag
464         tbuf [0] = 0;
465         tbuf [1] = 0;
466         len = 2;
467         /* FALLTHROUGH */
468     case EndpointOutRequest | USB_REQ_CLEAR_FEATURE:
469     case EndpointOutRequest | USB_REQ_SET_FEATURE:
470         dev_dbg (hcd->self.controller, "no endpoint features yet\n");
471         break;
472
473     /* CLASS REQUESTS (and errors) */
474
475     default:
476         /* non-generic request */
477         switch (typeReq) {
478         case GetHubStatus:
479         case GetPortStatus:
480         len = 4;
481             break;
482         case GetHubDescriptor:
483             len = sizeof (struct usb_hub_descriptor);
484             break;
485         }
486         status = hcd->driver->hub_control (hcd,
487                     typeReq, wValue, wIndex,
488                     tbuf, wLength);
489         break;
490 error:
491         /* "protocol stall" on error */
492         status = -EPIPE;
493     }
494
495     if (status) {
496         len = 0;
497         if (status != -EPIPE) {
498             dev_dbg (hcd->self.controller,
499                             "CTRL: TypeReq=0x%x val=0x%x "
500                             "idx=0x%x len=%d ==> %d\n",
501                             typeReq, wValue, wIndex,
502                             wLength, status);
503         }
504     }
505     if (len) {
```

```
506         if (urb->transfer_buffer_length < len)
507             len = urb->transfer_buffer_length;
508         urb->actual_length = len;
509         // always USB_DIR_IN, toward host
510         memcpy (ubuf, bufp, len);
511
512         /* report whether RH hardware supports remote wakeup */
513         if (patch_wakeup &&
514             len > offsetof (struct usb_config_descriptor,
515                             bmAttributes))
516             ((struct usb_config_descriptor *)ubuf)->bmAttributes
517                         |= USB_CONFIG_ATT_WAKEUP;
518     }
519
520     /* any errors get returned through the urb completion */
521     local_irq_save (flags);
522     spin_lock (&urb->lock);
523     if (urb->status == -EINPROGRESS)
524         urb->status = status;
525     spin_unlock (&urb->lock);
526     usb_hcd_giveback_urb (hcd, urb);
527     local_irq_restore (flags);
528     return 0;
529 }
```

看到这样近 200 行的函数，真是有一种“叫天天不灵叫地地不应”的感觉。不幸中的万幸，这个函数的结构还是很清晰的。自上而下地看过来就可以了。

对于控制传输，首先要获得它的 setup_packet，来自 urb 结构体，正如我们当初在 usb-storage 中看到的那样。这里把这个 setup_packet 赋给 cmd 指针。把其中的各个成员都给取出来，分别放在临时变量 typeReq，wValue，wIndex，wLength 中，然后来判断这个 typeReq。

如果是设备请求并且方向是 IN，而且是 USB_REQ_GET_STATUS，则设置 len 为 2。

如果是设备请求并且方向是 OUT，而且是 USB_REQ_CLEAR_FEATURE，则如何如何。

如果是设备请求并且方向是 OUT，而且是 USB_REQ_SET_FEATURE，则如何如何。

如果是设备请求并且方向是 IN，而且是 USB_REQ_GET_CONFIGURATION，则设置 len 为 1。

如果是设备请求并且方向是 OUT，而且是 USB_REQ_SET_CONFIGURATION， 则啥也不做。

如果是设备请求并且方向是 IN，而且是 USB_REQ_GET_DESCRIPTOR，则继续判断，wValue 值来决定究竟是要获得什么描述符。如果是 USB_DT_DEVICE，则说明要获得的是设备描述符，这正是咱们的上下文。（传递给 usb_get_descriptor 的第二个参数就是 USB_DT_DEVICE，传递给 usb_control_ msg 的第三个参数正是 USB_REQ_GET_DESCRIPTOR。）如果是 USB_DT_CONFIG，则说明要获得的是配置描述符；如果是 USB_DT_STRING，则说明要获得的是字符串描述符。实际上，对于 Root Hub 来说，这些东西都是一样的，在 drivers/usb/core/hcd.c 中都预先定义好了，usb2_rh_dev_descriptor 是针对 USB 2.0 的；而 usb11_rh_dev_descriptor 是针对 USB 1.1 的。 HCI 驱动里面设置了 flags 的 HCD_USB11。

```
116 #define KERNEL_REL      ((LINUX_VERSION_CODE >> 16) & 0x0ff)
117 #define KERNEL_VER      ((LINUX_VERSION_CODE >> 8) & 0x0ff)
118
119 /* usb 2.0 root hub device descriptor */
120 static const u8 usb2_rh_dev_descriptor [18] = {
121     0x12,       /*  __u8  bLength; */
122     0x01,       /*  __u8  bDescriptorType; Device */
123     0x00, 0x02, /*  __le16 bcdUSB; v2.0 */
124
125     0x09,       /*  __u8  bDeviceClass; HUB_CLASSCODE */
126     0x00,       /*  __u8  bDeviceSubClass; */
127     0x01,       /*  __u8  bDeviceProtocol; [ usb 2.0 single TT ]*/
128     0x40,       /*  __u8  bMaxPacketSize0; 64 Bytes */
129
130     0x00, 0x00, /*  __le16 idVendor; */
131     0x00, 0x00, /*  __le16 idProduct; */
132     KERNEL_VER, KERNEL_REL, /*  __le16 bcdDevice */
133
134     0x03,       /*  __u8  iManufacturer; */
```

```
135     0x02,       /*  __u8  iProduct; */
136     0x01,       /*  __u8  iSerialNumber; */
137     0x01        /*  __u8  bNumConfigurations; */
138 };
139
140 /* no usb 2.0 root hub "device qualifier" descriptor: one speed only */
141
142 /* usb 1.1 root hub device descriptor */
143 static const u8 usb11_rh_dev_descriptor [18] = {
144     0x12,       /*  __u8  bLength; */
145     0x01,       /*  __u8  bDescriptorType; Device */
146     0x10, 0x01, /*  __le16 bcdUSB; v1.1 */
147
148     0x09,       /*  __u8  bDeviceClass; HUB_CLASSCODE */
149     0x00,       /*  __u8  bDeviceSubClass; */
150     0x00,       /*  __u8  bDeviceProtocol; [ low/full speeds only ] */
151     0x40,       /*  __u8  bMaxPacketSize0; 64 Bytes */
152
153     0x00, 0x00, /*  __le16 idVendor; */
154     0x00, 0x00, /*  __le16 idProduct; */
155     KERNEL_VER, KERNEL_REL, /*  __le16 bcdDevice */
156
157     0x03,       /*  __u8  iManufacturer; */
158     0x02,       /*  __u8  iProduct; */
159     0x01,       /*  __u8  iSerialNumber; */
160     0x01        /*  __u8  bNumConfigurations; */
161 };
162
168 static const u8 fs_rh_config_descriptor [] = {
169
170     /* one configuration */
171     0x09,       /*  __u8  bLength; */
172     0x02,       /*  __u8  bDescriptorType; Configuration */
173     0x19, 0x00, /*  __le16 wTotalLength; */
174     0x01,       /*  __u8  bNumInterfaces; (1) */
175     0x01,       /*  __u8  bConfigurationValue; */
176     0x00,       /*  __u8  iConfiguration; */
177     0xc0,       /*  __u8  bmAttributes;
178                         Bit 7: must be set,
179                                6: Self-powered,
180                                5: Remote wakeup,
181                                4..0: resvd */
182     0x00,       /*  __u8  MaxPower; */
183
184     /* USB 1.1:
185      * USB 2.0, single TT organization (mandatory):
186      *     one interface, protocol 0
187      *
188      * USB 2.0, multiple TT organization (optional):
189      *     two interfaces, protocols 1 (like single TT)
190      *     and 2 (multiple TT mode) ... config is
191      *     sometimes settable
192      *     NOT IMPLEMENTED
193      */
194
195     /* one interface */
196     0x09,       /*  __u8  if_bLength; */
197     0x04,       /*  __u8  if_bDescriptorType; Interface */
198     0x00,       /*  __u8  if_bInterfaceNumber; */
199     0x00,       /*  __u8  if_bAlternateSetting; */
200     0x01,       /*  __u8  if_bNumEndpoints; */
201     0x09,       /*  __u8  if_bInterfaceClass; HUB_CLASSCODE */
202     0x00,       /*  __u8  if_bInterfaceSubClass; */
203     0x00,    /*  __u8  if_bInterfaceProtocol; [usb1.1 or single tt] */
204     0x00,       /*  __u8  if_iInterface; */
205
206     /* one endpoint (status change endpoint) */
207     0x07,       /*  __u8  ep_bLength; */
208     0x05,       /*  __u8  ep_bDescriptorType; Endpoint */
209     0x81,       /*  __u8  ep_bEndpointAddress; IN Endpoint 1 */
210     0x03,       /*  __u8  ep_bmAttributes; Interrupt */
211     0x02, 0x00,/*  __le16 ep_wMaxPacketSize; 1 + (MAX_ROOT_PORTS / 8) */
212     0xff        /*  __u8  ep_bInterval; (255 ms -- usb 2.0 spec) */
213 };
214
```

```
215 static const u8 hs_rh_config_descriptor [] = {
216
217     /* one configuration */
218     0x09,       /*  __u8  bLength; */
219     0x02,       /*  __u8  bDescriptorType; Configuration */
220     0x19, 0x00, /*  __le16 wTotalLength; */
221     0x01,       /*  __u8  bNumInterfaces; (1) */
222     0x01,       /*  __u8  bConfigurationValue; */
223     0x00,       /*  __u8  iConfiguration; */
224     0xc0,       /*  __u8  bmAttributes;
225                         Bit 7: must be set,
226                             6: Self-powered,
227                             5: Remote wakeup,
228                             4..0: resvd */
229     0x00,       /*  __u8  MaxPower; */
230
231     /* USB 1.1:
232      * USB 2.0, single TT organization (mandatory):
233      *     one interface, protocol 0
234      *
235      * USB 2.0, multiple TT organization (optional):
236      *     two interfaces, protocols 1 (like single TT)
237      *     and 2 (multiple TT mode) ... config is
238      *     sometimes settable
239      *     NOT IMPLEMENTED
240      */
241
242     /* one interface */
243     0x09,       /*  __u8  if_bLength; */
244     0x04,       /*  __u8  if_bDescriptorType; Interface */
245     0x00,       /*  __u8  if_bInterfaceNumber; */
246     0x00,       /*  __u8  if_bAlternateSetting; */
247     0x01,       /*  __u8  if_bNumEndpoints; */
248     0x09,       /*  __u8  if_bInterfaceClass; HUB_CLASSCODE */
249     0x00,       /*  __u8  if_bInterfaceSubClass; */
250     0x00,    /*  __u8  if_bInterfaceProtocol; [usb1.1 or single tt] */
251     0x00,       /*  __u8  if_iInterface; */
252
253     /* one endpoint (status change endpoint) */
254     0x07,       /*  __u8  ep_bLength; */
255     0x05,       /*  __u8  ep_bDescriptorType; Endpoint */
256     0x81,       /*  __u8  ep_bEndpointAddress; IN Endpoint 1 */
257     0x03,       /*  __u8  ep_bmAttributes; Interrupt */
258                   /* __le16 ep_wMaxPacketSize; 1 + (MAX_ROOT_PORTS / 8)
259                    * see hub.c:hub_configure() for details. */
260     (USB_MAXCHILDREN + 1 + 7) / 8, 0x00,
261     0x0c        /*  __u8  ep_bInterval; (256 ms -- usb 2.0 spec) */
262 };
```

如果是设备请求且方向为 IN，而且是 USB_REQ_GET_INTERFACE，则设置 len 为 1。

如果是设备请求且方向为 OUT，而且是 USB_REQ_SET_INTERFACE，则如何如何。

如果是设备请求且方向为 OUT，而且是 USB_REQ_SET_ADDRESS，则如何如何。

如果是端点请求且方向为 IN，而且是 USB_REQ_GET_STATUS，则如何如何。

如果是端点请求且方向为 OUT，而且是 USB_REQ_CLEAR_FEATURE 或者 USB_REQ_SET_FEATURE，则如何如何。

以上这些设置，统统是和 USB spec 中规定的东西相匹配的。

如果是 Hub 特定的类请求，而且是 GetHubStatus 或者是 GetPortStatus，则设置 len 为 4。

如果是 Hub 特定的类请求，而且是 GetHubDescriptor，则设置 len 为 usb_hub_descriptor 结构体的大小。

最后对于 Hub 特定的类请求需要调用主机控制器驱动程序的 hub_control 函数，对于 uhci_driver 来说，这个指针被赋值为 uhci_hub_control，来自 drivers/usb/host/uhci-hub.c：

```
239 static int uhci_hub_control(struct usb_hcd *hcd, u16 typeReq, u16 wValue,
240                     u16 wIndex, char *buf, u16 wLength)
241 {
242     struct uhci_hcd *uhci = hcd_to_uhci(hcd);
243     int status, lstatus, retval = 0, len = 0;
```

```
244     unsigned int port = wIndex - 1;
245     unsigned long port_addr = uhci->io_addr + USBPORTSC1 + 2 * port;
246     u16 wPortChange, wPortStatus;
247     unsigned long flags;
248
249     if (!test_bit(HCD_FLAG_HW_ACCESSIBLE, &hcd->flags) || uhci->dead)
250         return -ETIMEDOUT;
251
252     spin_lock_irqsave(&uhci->lock, flags);
253     switch (typeReq) {
254
255     case GetHubStatus:
256         *(__le32 *)buf = cpu_to_le32(0);
257         OK(4);          /* hub power */
258     case GetPortStatus:
259         if (port >= uhci->rh_numports)
260             goto err;
261
262         uhci_check_ports(uhci);
263         status = inw(port_addr);
264
265         /* Intel controllers report the OverCurrent bit active on.
266          * VIA controllers report it active off, so we'll adjust the
267          * bit value.  (It's not standardized in the UHCI spec.)
268          */
269         if (to_pci_dev(hcd->self.controller)->vendor ==
270                         PCI_VENDOR_ID_VIA)
271             status ^= USBPORTSC_OC;
272
273         /* UHCI doesn't support C_RESET (always false) */
274         wPortChange = lstatus = 0;
275         if (status & USBPORTSC_CSC)
276             wPortChange |= USB_PORT_STAT_C_CONNECTION;
277         if (status & USBPORTSC_PEC)
278             wPortChange |= USB_PORT_STAT_C_ENABLE;
279         if ((status & USBPORTSC_OCC) && !ignore_oc)
280             wPortChange |= USB_PORT_STAT_C_OVERCURRENT;
281
282         if (test_bit(port, &uhci->port_c_suspend)) {
283             wPortChange |= USB_PORT_STAT_C_SUSPEND;
284             lstatus |= 1;
285         }
286         if (test_bit(port, &uhci->resuming_ports))
287             lstatus |= 4;
288
289         /* UHCI has no power switching (always on) */
290         wPortStatus = USB_PORT_STAT_POWER;
291         if (status & USBPORTSC_CCS)
292             wPortStatus |= USB_PORT_STAT_CONNECTION;
293         if (status & USBPORTSC_PE) {
294             wPortStatus |= USB_PORT_STAT_ENABLE;
295             if (status & SUSPEND_BITS)
296                 wPortStatus |= USB_PORT_STAT_SUSPEND;
297         }
298         if (status & USBPORTSC_OC)
299             wPortStatus |= USB_PORT_STAT_OVERCURRENT;
300         if (status & USBPORTSC_PR)
301             wPortStatus |= USB_PORT_STAT_RESET;
302         if (status & USBPORTSC_LSDA)
303             wPortStatus |= USB_PORT_STAT_LOW_SPEED;
304
305         if (wPortChange)
306             dev_dbg(uhci_dev(uhci), "port %d portsc %04x,%02x\n",
307                                 wIndex, status, lstatus);
308
309         *(__le16 *)buf = cpu_to_le16(wPortStatus);
310         *(__le16 *)(buf + 2) = cpu_to_le16(wPortChange);
311          OK(4);
312     case SetHubFeature:           /* We don't implement these */
313     case ClearHubFeature:
314         switch (wValue) {
315         case C_HUB_OVER_CURRENT:
316         case C_HUB_LOCAL_POWER:
317             OK(0);
318         default:
```

```
319                 goto err;
320             }
321             break;
322         case SetPortFeature:
323             if (port >= uhci->rh_numports)
324                 goto err;
325 
326             switch (wValue) {
327             case USB_PORT_FEAT_SUSPEND:
328                 SET_RH_PORTSTAT(USBPORTSC_SUSP);
329                 OK(0);
330             case USB_PORT_FEAT_RESET:
331                 SET_RH_PORTSTAT(USBPORTSC_PR);
332 
333                 /* Reset terminates Resume signalling */
334                 uhci_finish_suspend(uhci, port, port_addr);
335 
336                 /* USB v2.0 7.1.7.5 */
337                 uhci->ports_timeout = jiffies +  msecs_to_jiffies(50);
338                 OK(0);
339             case USB_PORT_FEAT_POWER:
340                 /* UHCI has no power switching */
341                 OK(0);
342             default:
343                 goto err;
344             }
345             break;
346         case ClearPortFeature:
347             if (port >= uhci->rh_numports)
348                 goto err;
349 
350             switch (wValue) {
351             case USB_PORT_FEAT_ENABLE:
352             CLR_RH_PORTSTAT(USBPORTSC_PE);
353 
354                 /* Disable terminates Resume signalling */
355                 uhci_finish_suspend(uhci, port, port_addr);
356                 OK(0);
357             case USB_PORT_FEAT_C_ENABLE:
358                 CLR_RH_PORTSTAT(USBPORTSC_PEC);
359                 OK(0);
360             case USB_PORT_FEAT_SUSPEND:
361                 if (!(inw(port_addr) & USBPORTSC_SUSP)) {
362 
363                     /* Make certain the port isn't suspended */
364                     uhci_finish_suspend(uhci, port, port_addr);
365                 } else if (!test_and_set_bit(port,
366                                         &uhci->resuming_ports)) {
367                     SET_RH_PORTSTAT(USBPORTSC_RD);
368 
369                     /* The controller won't allow RD to be set
370                      * if the port is disabled.  When this happens
371                      * just skip the Resume signalling.
372                      */
373                     if (!(inw(port_addr) & USBPORTSC_RD))
374                         uhci_finish_suspend(uhci, port,
375                                                 port_addr);
376                     else
377                         /* USB v2.0 7.1.7.7 */
378                         uhci->ports_timeout = jiffies +
379                                         msecs_to_jiffies(20);
380                 }
381                 OK(0);
382             case USB_PORT_FEAT_C_SUSPEND:
383                 clear_bit(port, &uhci->port_c_suspend);
384                 OK(0);
385             case USB_PORT_FEAT_POWER:
386                 /* UHCI has no power switching */
387                 goto err;
388             case USB_PORT_FEAT_C_CONNECTION:
389                 CLR_RH_PORTSTAT(USBPORTSC_CSC);
390                 OK(0);
391             case USB_PORT_FEAT_C_OVER_CURRENT:
392                 CLR_RH_PORTSTAT(USBPORTSC_OCC);
393                 OK(0);
```

```
394         case USB_PORT_FEAT_C_RESET:
395         /* this driver won't report these */
396             OK(0);
397         default:
398             goto err;
399         }
400         break;
401     case GetHubDescriptor:
402         len = min_t(unsigned int, sizeof(root_hub_hub_des), wLength);
403         memcpy(buf, root_hub_hub_des, len);
404         if (len > 2)
405             buf[2] = uhci->rh_numports;
406         OK(len);
407     default:
408 err:
409         retval = -EPIPE;
410     }
411     spin_unlock_irqrestore(&uhci->lock, flags);
412
413     return retval;
414 }
```

249 行，struct usb_hcd 结构体的成员 unsigned long flags，当初在 usb_add_hcd 中调用 set_bit 函数设置了这么一个 flag，HCD_FLAG_HW_ACCESSIBLE，基本上这个 flag 在我们的故事中是被设置了的。另外，struct uhci_hcd 结构体有一个成员 unsigned int dead，它如果为 1 就表明控制器“宕机”了。

然后用一个 switch 来处理 Hub 特定的类请求。OK 居然也是一个宏，定义于 drivers/usb/host/uhci-hub.c：

```
78 #define OK(x)                   len = (x); break
```

所以如果请求是 GetHubStatus，则设置 len 为 4。

如果请求是 GetPortStatus，则调用 uhci_check_ports，然后读端口寄存器。USBPORTSC_CSC 表示端口连接有变化，USBPORTSC_PEC 表示端口 Enable 有变化。USBPORTSC_OCC 表示 Over Current 有变化。struct uhci_hcd 的两个成员，port_c_suspend 和 resuming_ports 都是电源管理相关的。

但无论如何，以上所做的这些都是为了获得：wPortStatus 和 wPortChange，以此来响应 GetPortStatus 这个请求。

但是 SetPortFeature 就有事情要做了。wValue 表明具体是什么特征。

SET_RH_PORTSTAT 这个宏就是专门用于设置 Root Hub 的端口特征的。

```
80 #define CLR_RH_PORTSTAT(x) \
81       status = inw(port_addr); \
82       status &= ~(RWC_BITS|WZ_BITS); \
83       status &= ~(x); \
84       status |= RWC_BITS & (x); \
85       outw(status, port_addr)
86
87 #define SET_RH_PORTSTAT(x) \
88       status = inw(port_addr); \
89       status |= (x); \
90   status &= ~(RWC_BITS|WZ_BITS); \
91       outw(status, port_addr)
```

对于 USB_PORT_FEAT_RESET，还需要调用 uhci_finish_suspend。

如果是 USB_PORT_FEAT_POWER，则什么也不做，因为 UHCI 不吃这一套。

如果请求是 ClearPortFeature，基本上也是一样的做法。除了调用的宏变成了 CLR_RH_PORTSTAT。

如果请求是 GetHubDescriptor，那就满足它呗。root_hub_hub_des 是早就在 drivers/usb/host/uhci-hub.c 中定义好的：

```
15 static __u8 root_hub_hub_des[] =
    16 {
17       0x09,                  /*  __u8  bLength; */
18       0x29,                  /*  __u8  bDescriptorType; Hub-descriptor */
19       0x02,                  /*  __u8  bNbrPorts; */
20       0x0a,                  /* __u16  wHubCharacteristics; */
21       0x00,                  /*   (per-port OC, no power switching) */
```

```
    22      0x01,               /*  __u8  bPwrOn2pwrGood; 2 ms */
    23      0x00,               /*  __u8  bHubContrCurrent; 0 mA */
    24      0x00,           /*  __u8  DeviceRemovable; *** 7 Ports max *** */
    25      0xff            /*  __u8  PortPwrCtrlMask; *** 7 ports max *** */
    26 };
```

回到 rh_call_control，switch 结束了，下面判断 status 和 len。

然后调用 usb_hcd_giveback_urb()，来自 drivers/usb/core/hcd.c：

```
1385 void usb_hcd_giveback_urb (struct usb_hcd *hcd, struct urb *urb)
1386 {
1387     int at_root_hub;
1388
1389     at_root_hub = (urb->dev == hcd->self.root_hub);
1390     urb_unlink (urb);
    1391
1392     /* lower level hcd code should use *_dma exclusively if the
1393      * host controller does DMA */
1394     if (hcd->self.uses_dma && !at_root_hub) {
1395         if (usb_pipecontrol (urb->pipe)
1396                 && !(urb->transfer_flags & URB_NO_SETUP_DMA_MAP))
1397             dma_unmap_single (hcd->self.controller, urb->setup_dma,
1398                               sizeof (struct usb_ctrlrequest),
1399                               DMA_TO_DEVICE);
1400         if (urb->transfer_buffer_length != 0
1401                 && !(urb->transfer_flags & URB_NO_TRANSFER_DMA_MAP))
1402         dma_unmap_single (hcd->self.controller,
1403                               urb->transfer_dma,
1404                               urb->transfer_buffer_length,
1405                               usb_pipein (urb->pipe)
1406                                  ? DMA_FROM_DEVICE
1407                                  : DMA_TO_DEVICE);
1408     }
1409
1410     usbmon_urb_complete (&hcd->self, urb);
1411     /* pass ownership to the completion handler */
1412     urb->complete (urb);
1413     atomic_dec (&urb->use_count);
1414     if (unlikely (urb->reject))
1415         wake_up (&usb_kill_urb_queue);
1416     usb_put_urb (urb);
1417 }
1418 EXPORT_SYMBOL (usb_hcd_giveback_urb);
```

这里最重要最有意义的一行当然就是 1412 行，调用 urb 的 complete 函数，这正是在 usb-storage 里期待的那个函数。从此 rh_call_control 函数也该返回了，以后设备驱动又获得了控制权。事实上令人欣喜的是对于 Root Hub，1394 行开始的这一段 if 是不会被执行的，因为 at_root_hub 显然是为真。不过就算这段要执行也没什么可怕的，无非就是把之前为 urb 建立的 DMA 映射给取消掉。而另一方面，对于 Root Hub 来说，complete 函数基本上是什么也不做，只不过是让咱们再次回到 usb_start_wait_urb 去，而控制传输需要的数据也已经 copy 到 urb->transfer_buffer 中去了。至此，Root Hub 的控制传输就算结束了，即我们的 usb_get_device_descriptor 函数取得了空前绝后的圆满成功。

13、非 Root Hub 的批量传输

看完了控制传输，再来看批量传输，Root hub 没有批量传输，所以只需要关注非 Root Hub。

当然还是从 usb_submit_urb()开始。和控制传输一样，可以直接跳到 usb_hcd_submit_urb()。由于我们在 start_rh()中设置了 hcd->state 为 HC_STATE_RUNNING，所以这里 list_add_tail 会被执行，本 urb 会被加入到 ep 的 urb_list 队列中去。

然后还是老套路，driver->urb_enqueue 会被执行，即又一次进入了 uhci_urb_enqueue。没啥好说的，uhci_alloc_urb_priv 和 uhci_alloc_qh 会被再次执行以申请 urbp 和 QH，但这次 uhci_submit_control 不会被调用了，取而代之的是 uhci_submit_bulk()。这个函数来自 drivers/usb/host/uhci-q.c：

```
1043 static int uhci_submit_bulk(struct uhci_hcd *uhci, struct urb *urb,
1044              struct uhci_qh *qh)
```

```
1045 {
1046     int ret;
1047
1048     /* Can't have low-speed bulk transfers */
1049     if (urb->dev->speed == USB_SPEED_LOW)
1050         return -EINVAL;
1051
1052     if (qh->state != QH_STATE_ACTIVE)
1053         qh->skel = SKEL_BULK;
1054     ret = uhci_submit_common(uhci, urb, qh);
1055     if (ret == 0)
1056     uhci_add_fsbr(uhci, urb);
1057     return ret;
1058 }
```

又是一个很“赤裸裸”的函数，除了设置 qh->skel 为 SKEL_BULK 以外，就是调用 uhci_submit_common 了，而这个函数也是我们今后将在中断传输中调用的。因为批量传输和中断传输一样，就是一个阶段，直接传递数据就可以了，不用那么多废话。如果成功返回的话在调用 uhci_add_fsbr 把 urbp->fsbr 设置为 1。我们来看一下 uhci_submit_common()，这是一个公共的函数，批量传输和中断传输都会调用它，来自 drivers/usb/host/uhci-q.c：

```
927 static int uhci_submit_common(struct uhci_hcd *uhci, struct urb *urb,
928               struct uhci_qh *qh)
929 {
930     struct uhci_td *td;
931     unsigned long destination, status;
932     int maxsze = le16_to_cpu(qh->hep->desc.wMaxPacketSize);
933     int len = urb->transfer_buffer_length;
934     dma_addr_t data = urb->transfer_dma;
935     __le32 *plink;
936     struct urb_priv *urbp = urb->hcpriv;
937     unsigned int toggle;
938
939     if (len < 0)
940         return -EINVAL;
941
942     /* The "pipe" thing contains the destination in bits 8--18 */
943    destination = (urb->pipe & PIPE_DEVEP_MASK)|usb_packetid(urb->pipe);
944     toggle = usb_gettoggle(urb->dev, usb_pipeendpoint(urb->pipe),
945                     usb_pipeout(urb->pipe));
946
947     /* 3 errors, dummy TD remains inactive */
948     status = uhci_maxerr(3);
949     if (urb->dev->speed == USB_SPEED_LOW)
950         status |= TD_CTRL_LS;
951     if (usb_pipein(urb->pipe))
952         status |= TD_CTRL_SPD;
953
954     /*
955      * Build the DATA TDs
956      */
957     plink = NULL;
958     td = qh->dummy_td;
959     do {    /* Allow zero length packets */
960         int pktsze = maxsze;
961
962         if (len <= pktsze) {          /* The last packet */
963             pktsze = len;
964             if (!(urb->transfer_flags & URB_SHORT_NOT_OK))
965                 status &= ~TD_CTRL_SPD;
966         }
967
968         if (plink) {
969             td = uhci_alloc_td(uhci);
970             if (!td)
971                 goto nomem;
972             *plink = LINK_TO_TD(td);
973         }
974         uhci_add_td_to_urbp(td, urbp);
975         uhci_fill_td(td, status,
976                     destination | uhci_explen(pktsze) |
977                     (toggle << TD_TOKEN_TOGGLE_SHIFT),
978                     data);
```

```
979         plink = &td->link;
980         status |= TD_CTRL_ACTIVE;
981
982         data += pktsze;
983         len -= maxsze;
984         toggle ^= 1;
985     } while (len > 0);
986
987     /*
988      * URB_ZERO_PACKET means adding a 0-length packet, if direction
989      * is OUT and the transfer_length was an exact multiple of maxsze,
990      * hence (len = transfer_length - N * maxsze) == 0
991      * however, if transfer_length == 0, the zero packet was already
992      * prepared above.
993      */
994     if ((urb->transfer_flags & URB_ZERO_PACKET) &&
995                     usb_pipeout(urb->pipe) && len == 0 &&
996                     urb->transfer_buffer_length > 0) {
997         td = uhci_alloc_td(uhci);
998         if (!td)
999             goto nomem;
1000        *plink = LINK_TO_TD(td);
1001
1002        uhci_add_td_to_urbp(td, urbp);
1003        uhci_fill_td(td, status,
1004                        destination | uhci_explen(0) |
1005                        (toggle << TD_TOKEN_TOGGLE_SHIFT),
1006                        data);
1007        plink = &td->link;
1008
1009        toggle ^= 1;
1010    }
1012    /* Set the interrupt-on-completion flag on the last packet.
1013     * A more-or-less typical 4 KB URB (= size of one memory page)
1014     * will require about 3  ms to transfer; that's a little on the
1015     * fast side but not enough to justify delaying an interrupt
1016     * more than 2 or 3 URBs, so we will ignore the URB_NO_INTERRUPT
1017     * flag setting. */
1018    td->status |= __constant_cpu_to_le32(TD_CTRL_IOC);
1020    /*
1021     * Build the new dummy TD and activate the old one
1022     */
1023    td = uhci_alloc_td(uhci);
1024    if (!td)
1025        goto nomem;
1026    *plink = LINK_TO_TD(td);
1028    uhci_fill_td(td, 0, USB_PID_OUT | uhci_explen(0), 0);
1029    wmb();
1030qh->dummy_td->status |= __constant_cpu_to_le32(TD_CTRL_ACTIVE);
1031    qh->dummy_td = td;
1032
1033    usb_settoggle(urb->dev, usb_pipeendpoint(urb->pipe),
1034                    usb_pipeout(urb->pipe), toggle);
1035    return 0;
1037 nomem:
1038    /* Remove the dummy TD from the td_list so it doesn't get freed */
1039    uhci_remove_td_from_urbp(qh->dummy_td);
1040    return -ENOMEM;
1041 }
```

几经曲折之后我终于看明白了这个函数，虽然这个函数很“无耻”，但是它却给了我们一丝亲切的感觉，我们曾经熟悉的 urb，曾经熟悉的 transfer_buffer_length 以及 transfer_dma 又一次映入了我们的眼帘。这里我们看到它们俩被赋给了 len 和 data。

932 行，令 maxsze 等于端点描述符里记录的 wMaxPacketSize，即包的最大“size”。

接下来又是一堆的赋值：

第一个，destination，urb->pipe 由几个部分组成，这里的两个宏无非就是提取其中的 destination，它们都来自 drivers/usb/host/uhci-hcd.h：

```
7 #define usb_packetid(pipe)   \
            (usb_pipein(pipe) ? USB_PID_IN : USB_PID_OUT)
8 #define PIPE_DEVEP_MASK        0x0007ff00
```

显然，PIPE_DEVEP_MASK 就是用来获取 bit8 到 bit18，而 usb_packtid 就是为了获得传输的方向，IN 还是 OUT，usb_pipein 就是获取 pipe 的 bit7。

第二个，toggle，usb_gettoggle 就是获得这个 toggle 位。

第三个，status，等式右边的 uhci_maxerr 来自 drivers/usb/host/uhci-hcd.h：

```
203 #define uhci_maxerr(err)                ((err) << TD_CTRL_C_ERR_SHIFT)
```

在同一个文件中还定义了这样一些宏：

```
184 #define TD_CTRL_SPD           (1 << 29)      /* Short Packet Detect */
185 #define TD_CTRL_C_ERR_MASK     (3 << 27)      /* Error Counter bits */
186 #define TD_CTRL_C_ERR_SHIFT    27
187 #define TD_CTRL_LS             (1 << 26)      /* Low Speed Device */
188 #define TD_CTRL_IOS            (1 << 25)      /* Isochronous Select */
189 #define TD_CTRL_IOC           (1 << 24)      /* Interrupt on Complete */
190 #define TD_CTRL_ACTIVE         (1 << 23)      /* TD Active */
191 #define TD_CTRL_STALLED        (1 << 22)      /* TD Stalled */
192 #define TD_CTRL_DBUFERR        (1 << 21)      /* Data Buffer Error */
193 #define TD_CTRL_BABBLE         (1 << 20)      /* Babble Detected */
194 #define TD_CTRL_NAK            (1 << 19)      /* NAK Received */
195 #define TD_CTRL_CRCTIMEO       (1 << 18)      /* CRC/Time Out Error */
196 #define TD_CTRL_BITSTUFF       (1 << 17)      /* Bit Stuff Error */
197 #define TD_CTRL_ACTLEN_MASK  0x7FF /* actual length, encoded as n - 1 */
198
199 #define TD_CTRL_ANY_ERROR     (TD_CTRL_STALLED | TD_CTRL_DBUFERR | \
200                               TD_CTRL_BABBLE | TD_CTRL_CRCTIME | \
201                               TD_CTRL_BITSTUFF)
```

这些宏看似很莫名其妙，其实是有来历的，UHCI spec 中对 TD 有明确的描述，硬件上来看，它一共有四个双字（DWORD）。这其中第二个双字被称为 TD CONTROL AND STATUS，就是专门记录控制和状态信息的，一个双字就是 32 个 bits，bit0 到 bit31。而其中咱们这里的 TD_CTRL_C_ERR_MASK 就是为了提取 bit28 和 bit27 的。spec 中说，这两位是一个计数器，记录的是这个 TD 在执行过程中被探测到出现错误的次数，比如一开始设置它为 3，那么它每次出现错误就减 1，三次错误之后这个计数器就从 1 变成了 0，于是主机控制器就会把这个 TD 设置为 inactive，即给这个 TD 宣判“死刑”。咱们这里设置的就是 3 次。

接下来几行还是设置这个 status，status 的 bit26 标志着这个设备是低速设备还是全速设备。如果为 1 则表示是低速设备，为 0 则表示是全速设备。bit29 表示 Short Packet Detect(SPD)，其含义为，如果这一位为 1，则当一个包是输入包，并且成功完成了传输，但是实际长度比最大长度要短，则这个 TD 将会被标为 inactive。如果是输出包，则这一位没有任何意义。所以这里判断的是这个管道是不是输入管道。另外，如果传输出现了错误，则这一位也没有任何意义，汇报 SPD 的前提是数据必须成功地被传输了。

在做好这些前奏工作之后，957 行开始干正经事了。

如果传输的长度比 pktsze，或者说小于 maxsze，则说明这个包是最后一个包了。URB_SHORT_NOT_OK 是 urb 的 transfer_flags 中众多个标志位的一个，如果设置了这一个 flag 就表明 short 包是不能够接受的。反之则说明确实是一个短包，这种情况就把 status 中 SPD 这一位给清掉。

接着看，plink 一开始被设置为 NULL。所以第一次进入循环的话就直接执行 974 行，uhci_add_td_to_urbp()，

然后调用 uhci_fill_td 函数，咱们已经讲过了，无非就是设置一个 TD 的 status，token 和 buffer 这三个成员。

设置了 TD 之后，令 plink 等于 td->link，TD 的 link 也是 UHCI spec 明确规定的 4 个 DWORD 之一，被称为 Link Pointer，物理上，正是它把各个 TD 给连接起来的。

设置好这些之后，再把 status 中 bit23 给设置为 1，这一位如果为 1，则表示激活这个传输了。TD_CTRL_ACTIVE 这一位用来表征 TD 是一个待执行的活跃交互，主机控制器驱动在调度一个交互请求时将这一位设成 1，而硬件（主机控制器）在完成了一次交互之后，或者成功，或者彻底失败，就将这一位改成 0。这样驱动程序只要扫描各个 uhci_td 数据结构，发现某个 uhci_td 数据结构的 TD_CTRL_ACTIVE 位变成了 0，就说明这个交互已经完成。

最后增加 data，减小 len，并且把 toggle 位置反。如果数据还没传输完，就开始下一轮的循环。

第二次循环的区别在于 plink 这时候已经有值了，所以这次 969 行 uhci_alloc_td 会被执行，这次就将申请一个 TD。然后让 plink 里的内容赋为这个 TD 的 DMA 地址，这样就把这个 TD 和之前的 TD 给连接了起来。而其他事情则和第一次循环时一样。

不过有人问了这么一个问题，这里貌似有两个队列，一个是 td->list，一个是 td->link，这是什么原因？我们看到 struct uhci_td 中有__le32 link 和 struct list_head list，后者就是一个经典的队列头，而前者是一个链接指针，实际上它们构成了两个队列，或者说两个链表，前者使用的物理地址，后者使用的是虚拟地址。因为 USB 主机控制器显然不认识虚拟地址。所以我们要让 USB 主机控制器能够顺着各个 TD 来执行，就得为它准备一个物理地址链接起来的队列，但是同时，从软件角度来说，要保证 CPU 能够访问各个 TD，则又必须以虚拟地址的方式组建一个队列，从而使得 CPU 可以对 uhci_td 数据结构进行常规的队列操作。在我们的故事中出现了两个队列。uhci_submit_common 函数结束后，各个 TD 就组成了一个 QH。

这个循环结束之后，主机控制器的驱动工作就算完成了，我们知道处理器的基本职责是取指令和执行指令，类似地，UHCI 主机控制器的基本职责就是取 TD 和执行 TD，这里因为 TD 也建好了，也连入该连接的地方了，剩下的具体执行就是硬件的事情了。

其实建立 TD 队列的过程是很简单的，反反复复的就是在调用三个函数：uhci_alloc_td，uhci_add_td_to_urbp，uhci_fill_td。其意图很明显，基本上就是三步走，申请 TD，将其加入大部队，填充好。其中每一次调用了 uhci_alloc_td 之后都要判断是否申请成功，如果不成功就直接 goto nomem。

然后还有一些细节的工作，994 行，判断 urb 的另一个 transfer_flags，URB_ZERO_PACKET 是否设置了，如果设置了，并且传输方向是输出，len 等于 0，需要传输的数据长度是大于 0 的（这说明最初 len 并不是 0，而现在是 0，即说明 transfer_buffer_length 的长度恰好等于整数个 maxsze）。这个 flag 的含义是这个传输最后需要有一个零长度的包。对于这种情况，没啥好说的，申请一个 TD 连接、填充好，然后把 toggle 位置反即可。

1018 行，设置 status 的 bit24（nterrupt on Complete(IOC)）。这一位如果为 1，则表示主机控制器会在这个 TD 执行的 Frame 结束时触发中断。当初咱们在 uhci_submit_control 中也给状态阶段的 TD 设置了这一位。

接下来的这一小段代码基本上就是处理那个 dummy_td。当年在 uhci_alloc_qh 中曾经刻意为 qh->dummy_td 给申请了空间。TD 是用来结束一个队列的，或者说它表征队列的结束。

这里结束之后，这个函数就结束了，返回 0。只有刚才申请 TD 时失败了才会跳到下面去执行 uhci_remove_td_from_urbp()，把 dummy_td 从 td_list 中删除。这个函数也是很简单的，来自 drivers/usb/host/uhci-q.c：

```
151 static void uhci_remove_td_from_urbp(struct uhci_td *td)
152 {
153     list_del_init(&td->list);
154 }
```

然后，uhci_submit_common 结束之后回到 uhci_submit_bulk，并进而回到 uhci_urb_enqueue 中，而剩下的代码和控制传输就一样了，无需多说。这样，传说中的批量传输就这么被轻松搞定了。同样，在数据真的执行完之后，会执行 urb 的 complete 函数，控制权会转移给设备驱动。

这一切听上去都很完美，似乎天衣无缝，可问题是，不管是之前说的控制传输还是现在说的批量传输，urb->complete 究竟是被谁调用的？前面在讲 Root Hub 时咱们看到了 usb_hcd_giveback_urb 被调用，而它会调用 urb->complete。那么对于非 Root hub 呢？

还记得咱们注册了中断函数吧？中断函数不会吃闲饭，咱们为控制传输和批量传输中的最后一个 TD 设置了 IOC，于是该 TD 完成之后的那个 Frame 结束时分，主机控制器会向 CPU 发送中断，于是中断函数会被调用。

14、传说中的中断服务程序（ISR）

想当初在 usb_add_hcd 中使用 request_irq 注册了中断函数，每注册一个函数都是为了日后能够利用该函数，当初注册了 usb_hcd_irq，这会儿就该调用这个函数了。这个函数来自 drivers/usb/core/hcd.c：

```
1431 irqreturn_t usb_hcd_irq (int irq, void *__hcd)
1432 {
1433struct usb_hcd        *hcd = __hcd;
1434    int                 start = hcd->state;
1435
1436    if (unlikely(start == HC_STATE_HALT ||
1437        !test_bit(HCD_FLAG_HW_ACCESSIBLE, &hcd->flags)))
1438        return IRQ_NONE;
1439    if (hcd->driver->irq (hcd) == IRQ_NONE)
1440        return IRQ_NONE;
1441
1442    set_bit(HCD_FLAG_SAW_IRQ, &hcd->flags);
1443
1444    if (unlikely(hcd->state == HC_STATE_HALT))
1445        usb_hc_died (hcd);
1446    return IRQ_HANDLED;
1447 }
```

对于 UHCI 来说，driver->irq 就是 uhci_irq()函数。来自 drivers/usb/host/uhci-hcd.c：

```
377 static irqreturn_t uhci_irq(struct usb_hcd *hcd)
378 {
379     struct uhci_hcd *uhci = hcd_to_uhci(hcd);
380     unsigned short status;
381     unsigned long flags;
382
383     /*
384      * Read the interrupt status, and write it back to clear the
385      * interrupt cause.  Contrary to the UHCI specification, the
386      * "HC Halted" status bit is persistent: it is RO, not R/WC.
387      */
388     status = inw(uhci->io_addr + USBSTS);
389     if (!(status & ~USBSTS_HCH))    /* shared interrupt, not mine */
390         return IRQ_NONE;
391     outw(status, uhci->io_addr + USBSTS);         /* Clear it */
392
393     if (status & ~(USBSTS_USBINT | USBSTS_ERROR | USBSTS_RD)) {
394         if (status & USBSTS_HSE)
395             dev_err(uhci_dev(uhci), "host system error, "
396                                     "PCI proble ms?\n");
397         if (status & USBSTS_HCPE)
398             dev_err(uhci_dev(uhci), "host controller process "
399                                     "error, something bad happened!\n");
400         if (status & USBSTS_HCH) {
401             spin_lock_irqsave(&uhci->lock, flags);
402             if (uhci->rh_state >= UHCI_RH_RUNNING) {
403                 dev_err(uhci_dev(uhci),
404                             "host controller halted, "
405                             "very bad!\n");
406                 if (debug > 1 && errbuf) {
407                     /* Print the schedule for debugging */
408                     uhci_sprint_schedule(uhci,
409                                 errbuf, ERRBUF_LEN);
410                     lprintk(errbuf);
411                 }
412                 uhci_hc_died(uhci);
413
414                 /* Force a callback in case there are
415                  * pending unlinks */
416                 mod_timer(&hcd->rh_timer, jiffies);
417             }
418             spin_unlock_irqrestore(&uhci->lock, flags);
419         }
420     }
421
422     if (status & USBSTS_RD)
423         usb_hcd_poll_rh_status(hcd);
424     else {
425     spin_lock_irqsave(&uhci->lock, flags);
```

```
426             uhci_scan_schedule(uhci);
427             spin_unlock_irqrestore(&uhci->lock, flags);
428         }
429
430         return IRQ_HANDLED;
431 }
```

USBSTS 就是 UHCI 的状态寄存器，而 USBSTS_USBINT 标志状态寄存器的 bit0，按照 UHCI spec 的规定，bit0 对应于 IOC。USBSTS_ERROR 对应于 bit 1，这一位如果为 1，表示传输出现了错误，USBSTS_RD 则对应于 bit2，RD 就是 Resume Detect 的意思，主机控制器在收到 resume 的信号时会把这一位设置为 1。所以很快就知道我们应该关注的就是 426 这么一行代码，即 uhci_scan_schedule 这个最熟悉的陌生人。

当我们再一次踏入 uhci_scan_schedule 时，曾经那段被我们略过的 while 循环现在就不得不面对了，uhci_advance_check 会被调用，它来自 drivers/usb/host/uhci-q.c：

```
1636 static int uhci_advance_check(struct uhci_hcd *uhci, struct uhci_qh *qh)
1637 {
1638     struct urb_priv *urbp = NULL;
1639     struct uhci_td *td;
1640     int ret = 1;
1641     unsigned status;
1642
1643     if (qh->type == USB_ENDPOINT_XFER_ISOC)
1644         goto done;
1645
1646     /* Treat an UNLINKING queue as though it hasn't advanced.
1647      * This is okay because reactivation will treat it as though
1648      * it has advanced, and if it is going to become IDLE then
1649      * this doesn't matter anyway.  Furthermore it's possible
1650      * for an UNLINKING queue not to have any URBs at all, or
1651      * for its first URB not to have any TDs (if it was dequeued
1652      * just as it completed).  So it's not easy in any case to
1653      * test whether such queues have advanced. */
1654     if (qh->state != QH_STATE_ACTIVE) {
1655     urbp = NULL;
1656         status = 0;
1657
1658     } else {
1659         urbp = list_entry(qh->queue.next, struct urb_priv, node);
1660         td = list_entry(urbp->td_list.next, struct uhci_td, list);
1661         status = td_status(td);
1662         if (!(status & TD_CTRL_ACTIVE)) {
1663
1664             /* We're okay, the queue has advanced */
1665             qh->wait_expired = 0;
1666             qh->advance_jiffies = jiffies;
1667            goto done;
1668         }
1669         ret = 0;
1670     }
1671
1672     /* The queue hasn't advanced; check for timeout */
1673     if (qh->wait_expired)
1674         goto done;
1675
1676     if (time_after(jiffies, qh->advance_jiffies + QH_WAIT_TIMEOUT)) {
1677
1678         /* Detect the Intel bug and work around it */
1679         if (qh->post_td && qh_element(qh) == LINK_TO_TD(qh->post_td)) {
1680             qh->element = qh->post_td->link;
1681             qh->advance_jiffies = jiffies;
1682             ret = 1;
1683             goto done;
1684         }
1685
1686         qh->wait_expired = 1;
1687
1688         /* If the current URB wants FSBR, unlink it temporarily
1689          * so that we can safely set the next TD to interrupt on
1690          * completion.  That way we'll know as soon as the queue
1691          * starts moving again. */
1692         if (urbp && urbp->fsbr && !(status & TD_CTRL_IOC))
1693             uhci_unlink_qh(uhci, qh);
```

```
1694
1695    } else {
1696        /* Unmoving but not-yet-expired queues keep FSBR alive */
1697        if (urbp)
1698            uhci_urbp_wants_fsbr(uhci, urbp);
1699    }
1700
1701 done:
1702    return ret;
1703 }
```

从 urbp 中的 td_list 里面取出一个 TD，读取它的状态，最初设置了 TD_CTRL_ACTIVE，如果一个 TD 被执行完了，主机控制器会把它的 TD_CTRL_ACTIVE 给取消掉。所以这里 1662 行判断，如果已经没有了 TD_CTRL_ACTIVE，说明这个 TD 已经被执行完了，于是执行 goto 语句跳出去，从而 uhci_advance_check 函数就返回值了，对于这种情况，返回值为 1。uhci_advance_check 顾名思义，就是检查咱们的队列有没有前进，如果一个 TD 从 ACTIVE 变成了非 ACTIVE，这就说明队列前进了，因为主机控制器只有执行完一个 TD 才会把一个 TD 的 ACTIVE 取消，然后它就会前进去获取下一个 QH 或者 TD。

而如果 uhci_advance_check 返回了 1，那么接下来 uhci_scan_qh 会被调用，它来自 drivers/usb/host/uhci-q.c：

```
1536 static void uhci_scan_qh(struct uhci_hcd *uhci, struct uhci_qh *qh)
1537 {
1538    struct urb_priv *urbp;
1539    struct urb *urb;
1540    int status;
1541
1542    while (!list_empty(&qh->queue)) {
1543        urbp = list_entry(qh->queue.next, struct urb_priv, node);
1544    urb = urbp->urb;
1545
1546        if (qh->type == USB_ENDPOINT_XFER_ISOC)
1547        status = uhci_result_isochronous(uhci, urb);
1548        else
1549            status = uhci_result_common(uhci, urb);
1550        if (status == -EINPROGRESS)
1551            break;
1552
1553        spin_lock(&urb->lock);
1554        if (urb->status == -EINPROGRESS)       /* Not dequeued */
1555            urb->status = status;
1556        else
1557            status = ECONNRESET;          /* Not -ECONNRESET */
1558        spin_unlock(&urb->lock);
1559
1560    /* Dequeued but completed URBs can't be given back unless
1561         * the QH is stopped or has finished unlinking. */
1562        if (status == ECONNRESET) {
1563        if (QH_FINISHED_UNLINKING(qh))
1564                qh->is_stopped = 1;
1565            else if (!qh->is_stopped)
1566                return;
1567        }
1568
1569        uhci_giveback_urb(uhci, qh, urb);
1570        if (status < 0 && qh->type != USB_ENDPOINT_XFER_ISOC)
1571            break;
1572    }
1573
1574    /* If the QH is neither stopped nor finished unlinking (normal case),
1575     * our work here is done. */
1576    if (QH_FINISHED_UNLINKING(qh))
1577        qh->is_stopped = 1;
1578    else if (!qh->is_stopped)
1579        return;
1580
1581    /* Otherwise give back each of the dequeued URBs */
1582 restart:
1583    list_for_each_entry(urbp, &qh->queue, node) {
1584        urb = urbp->urb;
1585        if (urb->status != -EINPROGRESS) {
1586
```

```
1587            /* Fix up the TD links and save the toggles for
1588             * non-Isochronous queues.  For Isochronous queues,
1589             * test for too-recent dequeues. */
1590            if (!uhci_cleanup_queue(uhci, qh, urb)) {
1591                qh->is_stopped = 0;
1592                return;
1593            }
1594            uhci_giveback_urb(uhci, qh, urb);
1595            goto restart;
1596        }
1597    }
1598    qh->is_stopped = 0;
1599
1600    /* There are no more dequeued URBs.  If there are still URBs on the
1601     * queue, the QH can now be re-activated. */
1602    if (!list_empty(&qh->queue)) {
1603        if (qh->needs_fixup)
1604            uhci_fixup_toggles(qh, 0);
1605
1606        /* If the first URB on the queue wants FSBR but its time
1607         * limit has expired, set the next TD to interrupt on
1608         * completion before reactivating the QH. */
1609        urbp = list_entry(qh->queue.next, struct urb_priv, node);
1610        if (urbp->fsbr && qh->wait_expired) {
1611        struct uhci_td *td = list_entry(urbp->td_list.next,
1612                                struct uhci_td, list);
1613
1614            td->status |= __cpu_to_le32(TD_CTRL_IOC);
1615        }
1616
1617        uhci_activate_qh(uhci, qh);
1618    }
1619
1620    /* The queue is empty.  The QH can become idle if it is fully
1621     * unlinked. */
1622    else if (QH_FINISHED_UNLINKING(qh))
1623        uhci_make_qh_idle(uhci, qh);
1624 }
```

可以看到，不管是控制传输还是批量传输，下一个被调用的函数都是 uhci_result_common()，来自 drivers/usb/host/uhci-q.c：

```
1151 static int uhci_result_common(struct uhci_hcd *uhci, struct urb *urb)
1152 {
1153    struct urb_priv *urbp = urb->hcpriv;
1154    struct uhci_qh *qh = urbp->qh;
1155    struct uhci_td *td, *tmp;
1156    unsigned status;
1157    int ret = 0;
1158
1159    list_for_each_entry_safe(td, tmp, &urbp->td_list, list) {
1160    unsigned int ctrlstat;
1161        int len;
1162
1163        ctrlstat = td_status(td);
1164        status = uhci_status_bits(ctrlstat);
1165        if (status & TD_CTRL_ACTIVE)
1166            return -EINPROGRESS;
1167
1168        len = uhci_actual_length(ctrlstat);
1169        urb->actual_length += len;
1170
1171        if (status) {
1172        ret = uhci_map_status(status,
1173                        uhci_packetout(td_token(td)));
1174            if ((debug == 1 && ret != -EPIPE) || debug > 1) {
1175                /* Some debugging code */
1176                dev_dbg(&urb->dev->dev,
1177                            "%s: failed with status %x\n",
1178                            __FUNCTION__, status);
1179
1180                if (debug > 1 && errbuf) {
1181                    /* Print the chain for debugging */
1182                uhci_show_qh(uhci, urbp->qh, errbuf,
1183                                        ERRBUF_LEN, 0);
```

```
1184                    lprintk(errbuf);
    1185                }
1186            }
1187
1188        } else if (len < uhci_expected_length(td_token(td))) {
1190        /* We received a short packet */
1191        if (urb->transfer_flags & URB_SHORT_NOT_OK)
1192                ret = -EREMOTEIO;
1194            /* Fixup needed only if this isn't the URB's last TD */
1195            else if (&td->list != urbp->td_list.prev)
1196                ret = 1;
1197        }
1199        uhci_remove_td_from_urbp(td);
1200        if (qh->post_td)
1201            uhci_free_td(uhci, qh->post_td);
1202        qh->post_td = td;
1204        if (ret != 0)
1205            goto err;
1206    }
1207    return ret;
1209 err:
1210    if (ret < 0) {
1211        /* In case a control transfer gets an error
1212         * during the setup stage */
1213        urb->actual_length = max(urb->actual_length, 0);
1214
1215        /* Note that the queue has stopped and save
1216         * the next toggle value */
1217        qh->element = UHCI_PTR_TERM;
1218        qh->is_stopped = 1;
1219        qh->needs_fixup = (qh->type != USB_ENDPOINT_XFER_CONTROL);
1220        qh->initial_toggle = uhci_toggle(td_token(td)) ^
1221                             (ret == -EREMOTEIO);
1223    } else        /* Short packet received */
1224        ret = uhci_fixup_short_transfer(uhci, qh, urbp);
1225    return ret;
1226 }
```

首先 list_for_each_entry_safe 就相当于传说中的 list_for_each_entry，其作用都是遍历 urbp 的 td_list，一个一个 TD 地处理。

1163 行，td_status 是一个很简单的宏，来自 drivers/usb/host/uhci-hcd.h：

```
262 static inline u32 td_status(struct uhci_td *td) {
263     __le32 status = td->status;
264
265     barrier();
266     return le32_to_cpu(status);
267 }
```

其实就是获取 struct uhci_td 结构体指针的 status 成员。

而 uhci_status_bits 亦是来自同一个文件中的宏：

```
204 #define uhci_status_bits(ctrl_sts)     ((ctrl_sts) & 0xF60000)
```

要看懂这个宏需要参考下面的图 3.14.1。

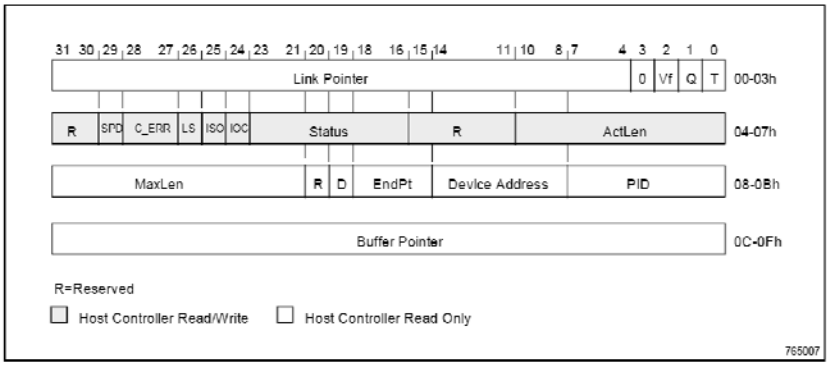

图 3.14.1　TD 结构定义

这就是 UHCI spec 中对 TD 的结构体的定义，我们注意到它有 4 个 DWORD，而 uhci_td 中的成员 status 实际上指的是这里的 04-07h 这整个双字，请注意这幅图中 04-07h 这个双字中，bit16 到 bit23 那一段被称为 Status，即这几位表示状态，uhci_status_bits 则是为了获得这几个 bits，把 ctrl_sts 和 0xF60000 相与得到 bit17 到 bit23，因为 UHCI spec 中规定了 bit16 是保留位，意义不大。

这其中，bit 23 被称为 Active，其实它就是我们一直说的那个 TD_CTRL_ACTIVE。如果这一位还设置了那么就说明这个 TD 还是活的，就不要去碰它。如果没有设置，那么继续往下走。

下一个宏是 uhci_actual_length，依然来自 drivers/usb/host/uhci-hcd.h：

```
205 #define uhci_actual_length(ctrl_sts)    (((ctrl_sts) + 1) & \
206                     TD_CTRL_ACTLEN_MASK)    /* 1-based */
```

这里 TD_CTRL_ACTLEN_MASK 是 0x7FF，我们注意到 TD 的定义中，在 04~07h 中，bit0 到 bit10 这 11 个 bits 被称之为 ActLen，这个 field 是由主机控制器来写的，表示实际传输了多少个 bytes，它被以 *n*-1 的方式进行编码，所以这里解码就要加 1。在 usb-storage 那个故事中看到的 urb->actual_length 就是这么计算出来的，即每次处理一个 TD 就加上 len。

顺便提一下，我们注意到在 uhci_submit_control 中设置了 urb->actual_length 为-8，实际上用 urb->actual_length 小于 0 来表示控制传输的 Setup 阶段没能取得成功，至于它具体是负多少并不重要，取为负 8 只是图个吉利。

如果一切正常的话，status 实际上应该是 0，不为 0 就表示出错了。1171 这一段就是为错误打印一些调试信息。

1188，如果虽然没有啥异常状态，但是 len 比期望值要小，那么首先判断是不是在 urb 的 transfer_flags 中设置了 URB_SHORT_NOT_OK，如果设置了，那就返回汇报错误。如果没有设置，继续判断，查看这个 TD 是不是咱们整个队伍中最后一个 TD，否则就有问题，设置返回值为 1。

1199 行，既然 TD 完成了使命，那么就可以“过河拆桥、卸磨杀驴”了。

1200 行，第一次见 qh->post_td，它当然是空的。如果不为空就调用 uhci_free_td 来释放它。struct uhci_qh 结构体中的成员 post_td 是用来记录刚刚完成了的那个 TD。它的赋值恰恰就是在 1202 这一行，即令 qh->post_td 等于现在这个 TD，因为这个 TD 就是刚刚完成的 TD。

正常的话，应该返回 0。如果不正常，那就跳到 1209 下面去。

如果 ret 小于 0，则需要对 QH 的一些成员进行赋值。

如果 ret 不小于 0，实际上就是对应于刚才那个 ret 为 1 的情况，即传输长度小于预期长度，这种情况就调用 uhci_fixup_short_transfer()这个专门为此而设计的函数。来自 drivers/usb/host/uhci-q.c：

```
1104 static int uhci_fixup_short_transfer(struct uhci_hcd *uhci,
1105                 struct uhci_qh *qh, struct urb_priv *urbp)
1106 {
1107     struct uhci_td *td;
1108     struct list_head *tmp;
1109     int ret;
1110
1111     td = list_entry(urbp->td_list.prev, struct uhci_td, list);
1112     if (qh->type == USB_ENDPOINT_XFER_CONTROL) {
1113
1114         /* When a control transfer is short, we have to restart
1115          * the queue at the status stage transaction, which is
1116          * the last TD. */
1117         WARN_ON(list_empty(&urbp->td_list));
1118         qh->element = LINK_TO_TD(td);
1119         tmp = td->list.prev;
1120         ret = -EINPROGRESS;
1121
1122     } else {
1123
1124         /* When a bulk/interrupt transfer is short, we have to
1125          * fix up the toggles of the following URBs on the queue
1126          * before restarting the queue at the next URB. */
```

```
1127        qh->initial_toggle = uhci_toggle(td_token(qh->post_td)) ^ 1;
1128        uhci_fixup_toggles(qh, 1);
1129
1130        if (list_empty(&urbp->td_list))
1131            td = qh->post_td;
1132        qh->element = td->link;
1133        tmp = urbp->td_list.prev;
1134        ret = 0;
1135    }
1136
1137    /* Remove all the TDs we skipped over, from tmp back to the start */
1138    while (tmp != &urbp->td_list) {
1139        td = list_entry(tmp, struct uhci_td, list);
1140        tmp = tmp->prev;
1141
1142        uhci_remove_td_from_urbp(td);
1143        uhci_free_td(uhci, td);
1144    }
1145    return ret;
1146 }
```

这里对于控制传输和对于批量传输有着不同的处理方法。

如果是控制传输，那么令 tmp 等于本 urb 的 td_list 中的倒数第 2 个 TD，然后一个一个往前走，见一个删一个，并且把 ret 设置为-EINPROGRESS 然后返回 ret，这样做的后果就是留下了最后一个 TD，而其他 TD 统统撤了。而对于控制传输，我们知道其最后一个 TD 就是状态阶段的 TD。

而对于批量传输或者中断传输的做法是从最后一个 TD 开始往前走，全都删除。uhci_fixup_toggles() 来自 drivers/usb/host/uhci-q.c：

```
377 static void uhci_fixup_toggles(struct uhci_qh *qh, int skip_first)
378 {
379     struct urb_priv *urbp = NULL;
380     struct uhci_td *td;
381     unsigned int toggle = qh->initial_toggle;
382     unsigned int pipe;
383
384     /* Fixups for a short transfer start with the second URB in the
385      * queue (the short URB is the first). */
386     if (skip_first)
387         urbp = list_entry(qh->queue.next, struct urb_priv, node);
388
389     /* When starting with the first URB, if the QH element pointer is
390      * still valid then we know the URB's toggles are okay. */
391     else if (qh_element(qh) != UHCI_PTR_TERM)
392         toggle = 2;
393
394     /* Fix up the toggle for the URBs in the queue.  Normally this
395      * loop won't run more than once: When an error or short transfer
396      * occurs, the queue usually gets emptied. */
397     urbp = list_prepare_entry(urbp, &qh->queue, node);
398     list_for_each_entry_continue(urbp, &qh->queue, node) {
399
400         /* If the first TD has the right toggle value, we don't
401          * need to change any toggles in this URB */
402         td = list_entry(urbp->td_list.next, struct uhci_td, list);
403         if (toggle > 1 || uhci_toggle(td_token(td)) == toggle) {
404             td = list_entry(urbp->td_list.prev, struct uhci_td,
405                             list);
406             toggle = uhci_toggle(td_token(td)) ^ 1;
407
408         /* Otherwise all the toggles in the URB have to be switched */
409         } else {
410             list_for_each_entry(td, &urbp->td_list, list) {
411                 td->token ^= __constant_cpu_to_le32(
412                                     TD_TOKEN_TOGGLE);
413                 toggle ^= 1;
414             }
415         }
416     }
417
418     wmb();
419     pipe = list_entry(qh->queue.next, struct urb_priv, node)->urb->pipe;
420     usb_settoggle(qh->udev, usb_pipeendpoint(pipe),
```

```
421                     usb_pipeout(pipe), toggle);
422     qh->needs_fixup = 0;
423 }
```

哇赛，这一段美妙的队列操作，足以让我等看得眼花缭乱头晕目眩了。

看这个函数之前，注意两点：

第一，在调用 uhci_fixup_toggles 之前的那句话，qh->initial_toggle 被赋了值，而且还就是 post_td 的 toggle 位取反。

第二，咱们传递进来的第 2 个参数是 1，即 skip_first 是 1，因此 387 行会被执行，urbp 是 QH 的 queue 队列中的第 2 个。因为第一个必然是刚才处理的那个，即那个出现短包问题的 urb。

然后 397，398 行从第 2 个 urbp 开始遍历 QH 的 queue 队列。首先是获得 urbp 里的第 1 个 TD。注意到 toggle 要么为 1 要么为 0。（除非 skip_first 为 0，执行了 392 行，那么 toggle 将等于 2。）如果这个 TD 的 toggle 位和 qh->initial_toggle 相同，即它和那个 post_td 的 toggle 相反，那么 TD 是正确的，直接让 TD 走到 td_list 的最后一个元素去，再把 toggle 置为反。

反之，如果 TD 的 toggle 和 qh->initial_toggle 不同，即它和之前那个 post_td 的 toggle 相同，那么说明整个 urb 中的所有的 TD 的 toggle 位都反了，都得翻一次。

最后调用 usb_settoggle 来设置一次，设置 qh->needs_fixup 为 0。

显然，这么说谁都会，关键是得理解，也许现在是时候去理解 USB 世界里的同步问题了。USB spec 中是为了实现同步，定义了 Data0 和 Data1 这两种序列位，如果发送者要发送多个包给接收者，则给每个包编上号，让 Data0 和 Data1 间隔着发送出去。发送方和接受方都维护着一张绪列位，在一次传输开始之前，发送方和接受方的这个序列位必须同步，或者说相同，即要么同为 Data0，要么同为 Data1，这种机制称之为 Data Toggle Synchronization。

举例来说，假设一开始双方都是 Data0，当接收方成功地接收到了一个包，会把自己的同步位翻转，即所谓的 toggle，它变成了 Data1，然后发送一个 ACK 给发送方，告诉对方，已成功的接收到了你的包，而发送方在接收到这个 ACK 之后，也会翻转自己的同步位，于是也跟着变成了 Data1。下一个包也是一样的做法。所以我们看到 uhci_submit_common()函数中没填充一个 TD，就翻转一次 toggle 位，即 984 行那个“toggle^=1”。同样在 uhci_submit_control()中也能看到对于 toggle 的处理。回过头去看 uhci_submit_control ()中 879 行，来看一下我们是如何为控制传输设置 toggle 位的。

首先是 Setup 阶段，让 toggle 位为 0。（A SETUP always uses a DATA0 PID for the data field of the SETUP transaction. ---USB spec 2.0, 8.5.3）

其次是数据阶段，在填充每一个 TD 之前翻转 toggle 位，即 850 行那个 destination^=TD_TOKEN_TOGGLE，第一次翻转之后 toggle 位是 Data1。

最后是状态阶段，879 行，为状态阶段的 toggle 位设置为 Data1，这是依据 USB spec 中规定的来设置的。（A Status stage is delineated by a change in direction of data flow from theprevious stage and always uses a DATA1 PID. ---USB spec 2.0, 8.5.3）

那么为何在 uhci_submit_common 中调用了 usb_gettoggle()和 usb_settoggle，而 uhci_submit_control 中没有调用呢？struct usb_device 这个结构体有这么一个成员 toggle[]。

```
336 struct usb_device {
337     int           devnum;       /* Address on USB bus */
338     char          devpath [16]; /* Use in messages: /port/port/... */
339     enum usb_device_state  state;/* configured, not attached, etc */
340     enum usb_device_speed  speed;  /* high/full/low (or error) */
341
342     struct usb_tt   *tt;      /* low/full speed dev, highspeed hub */
343     int           ttport;       /* device port on that tt hub */
344
345     unsigned int toggle[2];       /* one bit for each endpoint
346                                    * ([0] = IN, [1] = OUT) */
347
```

toggle 数组的第一个元素是针对 IN 类型端点的，第二个元素是针对 OUT 类型端点的，每个端点都在这张表里占有一个 bit。于是咱们就可以用它来记录端点的 toggle，以保证传输的同步，但是，实际上在这个故事里，真正使用这个数组的只有两种端点，即批量端点和中断端点，另外两种端点并不需要这个数组。首先，等时端点是不需要使用 toggle bits 来进行同步的，这是 USB spec 中规定的。Data Toggle 同步对等时传输没有意义。其次，控制传输的 toggle 位的 Setup 阶段总是 Data0，数据阶段总是从 Data1 开始，Status 阶段总是 Data1。

USB spec 已经为控制传输规定好了，必须遵守它，所以就没有必要另外使用这个数组来记录端点的 toggle 位了。这就是为什么操作这个 toggle 数组的两个函数 usb_gettoggle/usb_settoggle 不会出现在提交控制 urb 的函数 uhci_submit_control 中。而对于批量传输和中断传输，恰恰是因为每次在设置好一个 urb 的各个 TD 之后调用 usb_settoggle 来设置这个 toggle，下一次为新 urb 的第一个 TD 设置 toggle 位时才可以直接调用 usb_gettoggle。这样就保证了前一个 urb 的 TD 的 toggle 位和后一个 urb 的 TD 的 toggle 位刚好相反，即所谓的交叉顺序，以保证了和设备内部的 toggle 位相同步。

了解了这些 toggle 位的设置之后，再来看我们的这段代码，来看一下这个 uhci_fixup_toggles 究竟是怎么“fixup”的。根据前面看到的对 qh->initial_toggle 的赋值可以知道，initial_toggle 实际上就是接收到 short 包的那个 TD 的 toggle 位取反，即 post_td 的 toggle 取反（函数 uhci_fixup_short_transfer 中 1127 行），而 403 行所比较的就是第二个 urb 的第一个 TD 的 token 是否和现在这个一样，如果不一样，就把该 urb 的所有的 TD 翻转一下，如果一样，则说明没有问题，但无论哪种情况，都要记录 toggle 本身，因为注意到在 420 行还调用了 usb_settoggle 来设置了该管道的 toggle 位的。

那么如何理解这个一样就说明没有问题呢？我们知道，主机控制器处理的 TD 总是 QH 中的第一个 TD，当然其所属的 urb 也一定是 QH 的第一个 urb，而且该 TD 的 toggle 位是和端点同步的，假设它们之前都是 Data0，那么现在该 TD 结束之后，端点那边的 toggle 位就该变成了 Data1。

另一方面，根据 UHCI spec，我们知道，如果一个 urb 的 TD 被检测到了短包，则该 urb 剩下的 TD 就不会被处理了，而下一个 urb 的第一个 TD 的 toggle 得和现在 urb 的这个被处理的 TD 的 toggle 相反就说明它的 toggle 位也是 Data1，即它是和端点同步的。这样就可以理直气壮地重新开启下一个 urb。反之，如果第一个 TD 和端点的 toggle 位相反，就把整个队列的所有 TD 都给反一下，这个工程不可谓不浩大，但是没有办法，谁叫设备不争气发送出这种短包来呢？这就叫成长的代价。

另外提一下，和 uhci_submit_common()函数一样，也可以理解为什么在 uhci_fixup_toggles 最后，即 420 行，会再次调用 usb_settoggle 了。注意一下，403 至 415 这一段，toggle 的两种赋值：第一种，由于整个“队伍”是出于正确的同步状况，所以不用改任何一个 TD 的 toggle 位，404 行直接让 td 等于本 urb 队列中的最后一个 TD，然后 toggle 是它的 toggle 位取反。而对于整个队伍都得翻转的情况，看到 411 让每一个 td 进行翻转，而 413 行 toggle 也跟着一次次翻转，以保证 toggle 最终等于最后一个 td 的 toggle 位的翻转。

最后再来看一下 TD_CTRL_SPD 这个 flag 的使用。这个 flag 对应于 TD 那 4 个双字中的第二个双字中的 bit29，在 UHCI spec 中关于 bit 是这么介绍的：

```
    Short Packet Detect (SPD). 1=Enable. 0=Disable. When a packet has this bit set to 1 and the
packet:
    1. is an input packet;
    2. is in a queue; and
    3. successfully completes with an actual length less than the maximum length;
    then the TD is marked inactive, the Queue Header is not updated and the USBINT status bit (Status
Register) is set at the end of the frame. In addition, if the interrupt is enabled, the interrupt
will be sent at the end of the frame.
    Note that any error (e.g., babble or FIFO error) prevents the short packet from being reported.
The behavior is undefined when this bit is set with output packets or packets outside of queues.
```

所以，对于 IN 方向的数据包，如果设置了这个 flag，那么主机控制器读到一个短包之后，它就会触发中断。因此注意到 uhci_submit_common 函数中，951 行和 952 行，就对 IN 管道设置了这个 flag。即对于接下来的每一个数据包，都会检测一下看是否收到了短包，是的话就及时发送中断向上级汇报。而 965 行又取消掉这个 flag 了，因为这是最后一个包，最后一个包当然是有可能是短包的。同样，在

uhci_submit_control 中也是如法炮制，835 行设置了 TD_CTRL_SPD，即保证数据阶段能够准确地汇报“险情”，而 881 行又取消掉，因为这已经是状态阶段了，最后一个包当然是允许短包的。

最后注意到，uhci_fixup_toggles 最后一行设置了 qh->needs_fixup 为 0。稍后会看到对这个变量是否为 0 的判断，目前这个上下文当然就是 0。

回到 uhci_fixup_short_transfer 来，一个需要解释的问题是，为何要设置 qh->element。正如上面从 UHCI spec 中摘取过来的那段对 SPD 的解释中所说的，当遇到短包时，QH 不会被更新“update”，这也是为什么一个 TD 出现了短包下一个 TD 就不会被执行的原因。所以这里咱们就需要手工的“update”这个 QH。对于控制传输，QH 的 element 指向了状态传输的 TD，因为要让状态阶段重新执行一次，就算是短包也得汇报一下，所以最后返回的是-EINPROGRESS。而对于批量/中断传输， TD 是本 urbp 的 td_list 中最后一个 TD（看 1111 行的赋值）。element 指向了该 TD 的 link 指针，也就是指向了下一个 urb，所以最后返回的是 0。

到这里就很明白，uhci_fixup_short_transfer()中 1138 行 1144 这一段 while 循环的意义了。把那个有问题的 urb 的前面那些 TD 统统删掉，把内存也释放掉。

至此，我们结束了 uhci_fixup_short_transfer()。因而，uhci_result_common 也就结束了。回到了 uhci_scan_qh 中，仍然在 QH 中按照 urb 一个一个地循环。如果 status 是-EINPROGRESS，则结束循环，继续执行该 urb。

没什么故障的话，urb->status 应该还是-EINPROGRESS，这是最初提交 urb 时设置的。于是设置 urb->status 为 status，这就是执行之后的结果。

最后 1569 行，既然 status 不是-EINPROGRESS，那么 uhci_giveback_urb 被调用。

```
1485 static void uhci_giveback_urb(struct uhci_hcd *uhci, struct uhci_qh *qh,
1486             struct urb *urb)
1487 __releases(uhci->lock)
1488 __acquires(uhci->lock)
1489 {
1490     struct urb_priv *urbp = (struct urb_priv *) urb->hcpriv;
1491
1492     /* When giving back the first URB in an Isochronous queue,
1493      * reinitialize the QH's iso-related members for the next URB. */
1494     if (qh->type == USB_ENDPOINT_XFER_ISOC &&
1495                 urbp->node.prev == &qh->queue &&
1496                 urbp->node.next != &qh->queue) {
1497         struct urb *nurb = list_entry(urbp->node.next,
1498                         struct urb_priv, node)->urb;
1499
1500         qh->iso_packet_desc = &nurb->iso_frame_desc[0];
1501         qh->iso_frame = nurb->start_frame;
1502         qh->iso_status = 0;
1503     }
1504
1505     /* Take the URB off the QH's queue.  If the queue is now empty,
1506      * this is a perfect time for a toggle fixup. */
1507     list_del_init(&urbp->node);
1508     if (list_empty(&qh->queue) && qh->needs_fixup) {
1509         usb_settoggle(urb->dev, usb_pipeendpoint(urb->pipe),
1510                     usb_pipeout(urb->pipe), qh->initial_toggle);
1511         qh->needs_fixup = 0;
1512     }
1514     uhci_free_urb_priv(uhci, urbp);
1516     spin_unlock(&uhci->lock);
1517     usb_hcd_giveback_urb(uhci_to_hcd(uhci), urb);
1518     spin_lock(&uhci->lock);
1519
1520     /* If the queue is now empty, we can unlink the QH and give up its
1521      * reserved bandwidth. */
1522     if (list_empty(&qh->queue)) {
1523         uhci_unlink_qh(uhci, qh);
1524         if (qh->bandwidth_reserved)
1525             uhci_release_bandwidth(uhci, qh);
1526     }
1527 }
```

首先 1494 行这一段 if 是针对等时传输的，暂时略过。

然后把这个 urbp 从 QH 的队伍中删除掉。如果队列因此就空了，并且 needs_fixup 设置为了 1。那就调用 usb_settoggle。不过上下文里 needs_fixup 是 0，所以暂不管。把 urbp 的各个 TD 全部给删除，把 TD 的内存给释放掉，把 urbp 本身的内存释放掉。

接下来调用 usb_hcd_giveback_urb 把控制权交回给设备驱动程序。这个函数已经不再陌生了。

最后，如果 QH 整个队伍已经空了，那么就调用 uhci_unlink_qh 把 QH 给撤掉。这个函数来自 drivers/usb/host/uhci-q.h：

```
555 static void uhci_unlink_qh(struct uhci_hcd *uhci, struct uhci_qh *qh)
556 {
557     if (qh->state == QH_STATE_UNLINKING)
558         return;
559     WARN_ON(qh->state != QH_STATE_ACTIVE || !qh->udev);
560     qh->state = QH_STATE_UNLINKING;
561
562     /* Unlink the QH from the schedule and record when we did it */
563     if (qh->skel == SKEL_ISO)
564         ;
565     else if (qh->skel < SKEL_ASYNC)
566         unlink_interrupt(uhci, qh);
567     else
568         unlink_async(uhci, qh);
569
570     uhci_get_current_frame_number(uhci);
571     qh->unlink_frame = uhci->frame_number;
572
573     /* Force an interrupt so we know when the QH is fully unlinked */
574     if (list_empty(&uhci->skel_unlink_qh->node))
575         uhci_set_next_interrupt(uhci);
576
577     /* Move the QH from its old list to the end of the unlinking list */
578     if (qh == uhci->next_qh)
579         uhci->next_qh = list_entry(qh->node.next, struct uhci_qh,
580                         node);
581     list_move_tail(&qh->node, &uhci->skel_unlink_qh->node);
582 }
```

对于批量传输或者控制传输，要调用的是 unlink_async()，依然是来自 drivers/usb/host/uhci-q.c：

```
537 static void unlink_async(struct uhci_hcd *uhci, struct uhci_qh *qh)
538 {
539     struct uhci_qh *pqh;
540     __le32 link_to_next_qh = qh->link;
541
542     pqh = list_entry(qh->node.prev, struct uhci_qh, node);
543     pqh->link = link_to_next_qh;
544
545     /* If this was the old first FSBR QH, link the terminating skeleton
546      * QH to the next (new first FSBR) QH. */
547     if (pqh->skel < SKEL_FSBR && qh->skel >= SKEL_FSBR)
548         uhci->skel_term_qh->link = link_to_next_qh;
549     mb();
550 }
```

“打江山难而毁江山容易”，这一句话从 link_async 和 unlink_async 这两个函数对比一下就会明白其真义。540 行，542 行，543 行的结果就是经典的删除队列节点的操作。让 pqh 等于 qh 的前一个节点，然后让 pqh 的 link 等于原来 qh 的 link，这样 qh 就没有利用价值了。

547 行这个 if 也不难理解，如果刚才的 qh 是第一个 FSBR 的 qh，那么就令 skel_term_qh 的 link 指向下一个 qh，因为前面说过，skel_term_qh 总是要被设置为第一个 FSBR qh。

调用 uhci_get_current_frame_number 获得当前的 Frame，记录在 unlink_frame 中。

调用 uhci_set_next_interrupt，来自 drivers/usb/host/uhci-q.c：

```
28 static void uhci_set_next_interrupt(struct uhci_hcd *uhci)
29 {
30     if (uhci->is_stopped)
31         mod_timer(&uhci_to_hcd(uhci)->rh_timer, jiffies);
```

```
    32      uhci->term_td->status |= cpu_to_le32(TD_CTRL_IOC);
    33 }
```

这个函数的行为显然是和 uhci_clear_next_interrupt 相反的，等于是开启中断。

如果这个 qh 是 uhci->next_qh，那么就让 next_qh 顺延至下一个 QH。

最后把刚才 unlink 的这个 QH 插入到另外一支队伍中去，这支队伍就是 uhci->skel_unlink_qh，所有被 unlink 的 QH 都会被招入这支“革命队伍”中去。很显然这是一支“无产阶级革命队伍”，因为进来的 qh 都是一无所有的。

uhci_giveback_urb 结束了，回到 uhci_scan_qh 中。uhci_cleanup_queue 被调用，来自 drivers/usb/host/uhci-q.c：

```
319 static int uhci_cleanup_queue(struct uhci_hcd *uhci, struct uhci_qh *qh,
320              struct urb *urb)
321 {
322     struct urb_priv *urbp = urb->hcpriv;
323     struct uhci_td *td;
324     int ret = 1;
325
326     /* Isochronous pipes don't use toggles and their TD link pointers
327      * get adjusted during uhci_urb_dequeue().  But since their queues
328      * cannot truly be stopped, we have to watch out for dequeues
329      * occurring after the nominal unlink frame. */
330     if (qh->type == USB_ENDPOINT_XFER_ISOC) {
331         ret = (uhci->frame_number + uhci->is_stopped !=
332                         qh->unlink_frame);
333         goto done;
334     }
335
336     /* If the URB isn't first on its queue, adjust the link pointer
337      * of the last TD in the previous URB.  The toggle doesn't need
338      * to be saved since this URB can't be executing yet. */
339     if (qh->queue.next != &urbp->node) {
340         struct urb_priv *purbp;
341         struct uhci_td *ptd;
342
343     purbp = list_entry(urbp->node.prev, struct urb_priv, node);
344         WARN_ON(list_empty(&purbp->td_list));
345         ptd = list_entry(purbp->td_list.prev, struct uhci_td,
346                         list);
347         td = list_entry(urbp->td_list.prev, struct uhci_td,
348                         list);
349         ptd->link = td->link;
350         goto done;
351     }
352
353     /* If the QH element pointer is UHCI_PTR_TERM then then currently
354      * executing URB has already been unlinked, so this one isn't it. */
355     if (qh_element(qh) == UHCI_PTR_TERM)
356         goto done;
357     qh->element = UHCI_PTR_TERM;
358
359     /* Control pipes don't have to worry about toggles */
360     if (qh->type == USB_ENDPOINT_XFER_CONTROL)
361         goto done;
362
363     /* Save the next toggle value */
364     WARN_ON(list_empty(&urbp->td_list));
365     td = list_entry(urbp->td_list.next, struct uhci_td, list);
366     qh->needs_fixup = 1;
367     qh->initial_toggle = uhci_toggle(td_token(td));
368
369 done:
370     return ret;
371 }
```

最后，uhci_make_qh_idel 被调用，来自 drivers/usb/host/uhci-q.c：

```
590 static void uhci_make_qh_idle(struct uhci_hcd *uhci, struct uhci_qh *qh)
591 {
592     WARN_ON(qh->state == QH_STATE_ACTIVE);
594     if (qh == uhci->next_qh)
```

```
595             uhci->next_qh = list_entry(qh->node.next, struct uhci_qh,
596                             node);
597         list_move(&qh->node, &uhci->idle_qh_list);
598         qh->state = QH_STATE_IDLE;
600         /* Now that the QH is idle, its post_td isn't being used */
601         if (qh->post_td) {
602             uhci_free_td(uhci, qh->post_td);
603             qh->post_td = NULL;
604         }
606         /* If anyone is waiting for a QH to become idle, wake them up */
607         if (uhci->num_waiting)
608             wake_up_all(&uhci->waitqh);
609 }
```

目的就一个：设置 qh->state 为 QH_STATE_IDLE。

uhci_make_qh_idle 结束之后，uhci_scan_qh 也就结束了，回到了 uhci_scan_schedule 中。

最后判断 uhci->skel_unlink_qh 领衔的“队伍”是否为空，如果为空，就调用 uhci_clear_next_interrupt 清中断，否则又有 QH 加入了这支“队伍”，就调用 uhci_set_next_interrupt 去产生下一次中断，从而再次把 qh->state 设置为 QH_STATE_IDLE。于是 uhci_scan_schedule 也结束了。而 uhci_irq 也就结束了。

15、Root Hub 的中断传输

中断传输和等时传输无疑要比之前的那两种传输复杂些，至少它们具有周期性。我们看代码时看不懂也不用灰心，歌手林志炫的成名曲《单身情歌》也就是把自己看代码的亲身经历唱了出来：“为了看代码孤军奋斗，早就吃够了看代码的苦，在代码中失落的人到处有，而我只是其中一个”。

和前面那个控制传输一样，中断传输的代码也分为两部分：一个是针对 Root Hub 的，这部分相当简单；另一个是针对非 Root Hub 的，这一部分明显复杂许多。先来看 Root Hub 的。跟踪 urb 的入口多少年也不会变，依然是 usb_submit_urb。

还记得在 Hub 驱动中讲的那个 hub_probe 吗？在 Hub 驱动的探测过程中，最终会提交一个 urb，即中断 urb。那么来看调用 usb_submit_urb()来 submit 这个 urb 之后究竟会发生什么？

但是与之前控制传输批量传输不同的是，之前是“凌波微步”来到了最后一行 usb_hcd_submit_urb，而现在在这一行之前还有几行是需要关注的。

首先注意到 243 行对临时变量 temp 赋了值，看到它被赋值为 usb_pipetype(pipe)，我相信即使是普通人也知道，从此以后 temp 就是管道类型。

于是飞一样飞到 338 行，这里有一个 switch。看到这里你明白当初在讲控制传输和 Bulk 传输时为何跳过了这一段了吧，没错，这里只有两个 case，即 PIPE_ISOCHRONOUS 和 PIPE_INTERRUPT，这两个 case 就是等时管道和中断管道。而控制传输和 Bulk 传输根本不在这一个选择的考虑范畴之内。所以当时我们很幸福地“飘”了过去。但现在不行了。实际上这里对于等时传输和对于中断传输处理方法是一样的。

首先判断 urb->interval，在 Hub 驱动中已经讲过它的作用，它当然不能小于等于 0。

其次根据设备是高速还是全速低速，再一次设置 interval。当时在 Hub 驱动中也说过，对于高速设备和全速低速设备，这个 interval 的单位是不一样的。前者的单位是微帧，后者的单位是帧，所以这里对它们有不同的处理方法，但是总的来说，可以看到 temp 无论如何会是 2 的整数次方，所以 372 行这么一赋值的效果是，如果你的期待值是介于 2 的 *n* 次方和 2 的 *n*+1 次方之间，那么就把它设置成 2 的 *n* 次方。因为最终设置成 2 的整数次方对我们来说软件上便于实现，而硬件上来说也无所谓，因为 USB spec 中也是允许的，比如，USB spec 2.0 5.7.4 中有这么一段：

“The period provided by the system may be shorter than that desired by the device up to the shortest period defined by the USB (125 µs microframe or 1 ms frame). The client software and device can depend only on the fact that the host will ensure that the time duration between two transaction attempts with the endpoint will be no longer than the desired period.”

现在进入 usb_hcd_submit_urb。而对于 Root Hub，我们又再一次进入了 rh_urb_enqueue；对于中断传输，我们进入了 rh_queue_status，这个函数来自 drivers/usb/core/hcd.c：

```
600 static int rh_queue_status (struct usb_hcd *hcd, struct urb *urb)
601 {
602     int           retval;
603     unsigned long   flags;
604     int           len = 1 + (urb->dev->maxchild / 8);
605
606     spin_lock_irqsave (&hcd_root_hub_lock, flags);
607     if (urb->status != -EINPROGRESS)       /* already unlinked */
608         retval = urb->status;
609     else if (hcd->status_urb || urb->transfer_buffer_length < len) {
610         dev_dbg (hcd->self.controller, "not queuing rh status urb\n");
611         retval = -EINVAL;
612     } else {
613         hcd->status_urb = urb;
614         urb->hcpriv = hcd;     /* indicate it's queued */
616         if (!hcd->uses_new_polling)
617             mod_timer (&hcd->rh_timer, jiffies + msecs_to_jiffies(250));
620         /* If a status change has already occurred, report it ASAP */
621         else if (hcd->poll_pending)
622             mod_timer (&hcd->rh_timer, jiffies);
623         retval = 0;
624     }
625     spin_unlock_irqrestore (&hcd_root_hub_lock, flags);
626     return retval;
627 }
```

还好这个函数不那么变态，由于设置了 hcd->uses_new_polling 为 1，而 hcd->poll_pending 只有在一个地方被改变，即 usb_hcd_poll_rh_status()，如果这个函数被调用了而 Hub 端口处没什么变化，那么 poll_pending 就会设置为 1。但当第一次来到这个函数时，poll_pending 还没有被设定过，则它只能是 0。

假设第一次执行 usb_hcd_poll_rh_status 时，Root Hub 的端口确实没有什么信息，即没有连接任何 USB 设备并且没有任何需要汇报的信息，那么 poll_pending 就会设置为 1。所以下一次当来到这个函数时，622 行的 mod_timer 会被执行。所以将再一次执行 usb_hcd_poll_rh_status，并且是立即执行！

但关于 usb_hcd_poll_rh_status，咱们也没什么好讲的，当初我们已经详细地讲过了。所以基本上我们就知道了，如果 Root Hub 的端口没有什么改变的话，usb_submit_urb 为 Root Hub 而提交的中断 urb 也不干什么正经事，我们能看到的是 rh_queue_status，rh_urb_enqueue，usb_hcd_submit_urb，usb_submit_urb 这四个函数像多米诺骨牌一样一个一个地依次返回 0。即使 Hub 端口里永远不接入任何设备，驱动程序也仍然大呼道“我们还活着”。

不过我最后想提醒一点，由于 Hub 驱动中的 usb_submit_urb 是在 hub_probe 的过程中被执行的，而这时候实际上正处在 register_root_hub 中，也就是说，咱们是在 usb_add_hcd 中，回过去看这个函数你会发现，1639 行调用 register_root_hub，而 1643 行调用 usb_hcd_poll_rh_status。这也就是说，尽管很早之前就讲过了 usb_hcd_poll_rh_status 函数，但是实际上第一次调用 usb_hcd_poll_rh_status 发生在 rh_queue_status 之后。这也就是为什么这里第一次进入 rh_queue_status 时，poll_pending 的值为 0，因为只有调用了 usb_hcd_poll_rh_status 之后，poll_pending 才有可能变成 1。

16、非 Root Hub 的中断传输

再来看非 Root hub 的中断传输，usb_submit_urb 还是那个 usb_submit_urb，usb_hcd_submit_urb 还是那个 usb_hcd_submit_urb，但是很显然 rh_urb_enqueue 不会再被调用。取而代之的是 1014 行，driver->urb_enqueue 的被调用，即 uhci_urb_enqueue 函数。这个函数在讲控制传输时已经讲过了，后来在讲批量传输时又讲过，但是当时的上下文是控制传输或者批量传输，当然和现在的中断传输不一样。

我们回过头来看 uhci_urb_enqueue，很快就会发现对于中断传输，执行 1415 行，会调用 uhci_submit_interrupt 函数。于是有人建议我们立即去看 uhci_submit_interrupt，不过“有时候看到的不一定是真的，真的不一定看得到”。1415 行只是表面现象。

其实在看 uhci_submit_interrupt 之前，需要注意的是 1401 行，uhci_alloc_qh 函数。虽然大家都调用了它，可是不同的上下文里它做的事情大不一样。让我们再次回到 uhci_alloc_qh 中来，来自 drivers/usb/host/uhci-q.c 的函数不长，所以不妨再一次贴出来：

```
247 static struct uhci_qh *uhci_alloc_qh(struct uhci_hcd *uhci,
248              struct usb_device *udev, struct usb_host_endpoint *hep)
249 {
250     dma_addr_t dma_handle;
251     struct uhci_qh *qh;
252
253     qh = dma_pool_alloc(uhci->qh_pool, GFP_ATOMIC, &dma_handle);
254     if (!qh)
255         return NULL;
256
257     me mset(qh, 0, sizeof(*qh));
258     qh->dma_handle = dma_handle;
259
260     qh->element = UHCI_PTR_TERM;
261     qh->link = UHCI_PTR_TERM;
262
263     INIT_LIST_HEAD(&qh->queue);
264     INIT_LIST_HEAD(&qh->node);
265
266     if (udev) {           /* Normal QH */
267     qh->type = hep->desc.bmAttributes & USB_ENDPOINT_XFERTYPE_MASK;
268         if (qh->type != USB_ENDPOINT_XFER_ISOC) {
269             qh->dummy_td = uhci_alloc_td(uhci);
270             if (!qh->dummy_td) {
271                 dma_pool_free(uhci->qh_pool, qh, dma_handle);
272                 return NULL;
273             }
274         }
275         qh->state = QH_STATE_IDLE;
276         qh->hep = hep;
277         qh->udev = udev;
278         hep->hcpriv = qh;
279
280         if (qh->type == USB_ENDPOINT_XFER_INT ||
281                         qh->type == USB_ENDPOINT_XFER_ISOC)
282             qh->load = usb_calc_bus_time(udev->speed,
283                             usb_endpoint_dir_in(&hep->desc),
284                             qh->type == USB_ENDPOINT_XFER_ISOC,
285                             le16_to_cpu(hep->desc.wMaxPacketSize))
286                         / 1000 + 1;
287
288     } else {              /* Skeleton QH */
289         qh->state = QH_STATE_ACTIVE;
290         qh->type = -1;
291     }
292     return qh;
293 }
```

很显然，280 行，不管是中断传输还是等时传输，都需要执行 282 行至 286 行这一小段。这一段其实就是调用了 usb_calc_bus_time()函数。

这个函数来自 drivers/usb/core/hcd.c：

```
860 long usb_calc_bus_time (int speed, int is_input, int isoc, int bytecount)
861 {
862     unsigned long   tmp;
863
864     switch (speed) {
865     case USB_SPEED_LOW:     /* INTR only */
866         if (is_input) {
867             tmp = (67667L * (31L + 10L * BitTime (bytecount))) / 1000L;
868             return (64060L + (2 * BW_HUB_LS_SETUP) + BW_HOST_DELAY + tmp);
869         } else {
870             tmp = (66700L * (31L + 10L * BitTime (bytecount))) / 1000L;
871             return (64107L + (2 * BW_HUB_LS_SETUP) + BW_HOST_DELAY + tmp);
872         }
873     case USB_SPEED_FULL:    /* ISOC or INTR */
874         if (isoc) {
875             tmp = (8354L * (31L + 10L * BitTime (bytecount))) / 1000L;
876             return (((is_input) ? 7268L : 6265L) + BW_HOST_DELAY + tmp);
```

```
877             } else {
878                 tmp = (8354L * (31L + 10L * BitTime (bytecount))) / 1000L;
879                 return (9107L + BW_HOST_DELAY + tmp);
880             }
881         case USB_SPEED_HIGH:    /* ISOC or INTR */
882             // FIXME adjust for input vs output
883             if (isoc)
884                 tmp = HS_NSECS_ISO (bytecount);
885             else
886                 tmp = HS_NSECS (bytecount);
887             return tmp;
888         default:
889             pr_debug ("%s: bogus device speed!\n", usbcore_name);
890             return -1;
891         }
892 }
```

这俨然就是一道小学数学题。传递进来的四个参数都很直白。speed 表征设备的速度，is_input 表征传输的方向，isoc 表征是不是等时传输，为 1 就是等时传输，为 0 则是中断传输。bytecount 更加直白，要传输多少个 bytes 的字节。

以前我一直只知道什么是贷款，因为我们 80 后中的大部分都不得不贷款去做房奴，但我从不知道究竟什么是带宽，看到了这个 usb_calc_bus_time()函数和我们即将要看到的 uhci_reserve_bandwidth()函数之后我总算是对带宽有一点了解了。

带宽这个词用江湖上的话来说，就是单位时间内传输的数据量，即单位时间内最大可能提供多少个二进制位传输，按江湖规矩，单位时间指的就是每秒。既然扯到时间，自然就应该计算时间。从软件的角度来说，每次建立一个管道我们都需要计算它所消耗的总线时间，或者说带宽，如果带宽不够了当然就不能建立了。

事实上以上这一堆的计算都是依据 USB spec 2.0 中 5.11.3 节里提供的公式，我们这里列举出全速的情况，spec 中的公式如图 3.15.1：

Full-speed (输入)

非等时传输 (包含 握手通信)
= 9107 + (83.54 + Floor (3.167 + BitStuffTime (Data_ba))) + Host_Delay

等时传输 (不包含握手通信)
= 7268 + (83.54 + Floor (3.167 + BitStuffTime (Data_ba))) + Host_Delay

Full-speed (输出)

非等时传输 (包含 握手通信)
= 9107 + (83.54 + Floor (3.167 + BitStuffTime (Data_ba))) + Host_Delay

等时传输 (不包含握手通信)
= 6265 + (83.54 + Floor (3.167 + BitStuffTime (Data_ba))) + Host_Delay

图 3.15.1　全速设备的计算公式

这一切的单位都是纳秒。

其中 BW_HOST_DELAY 是定义于 drivers/usb/core/hcd.h 的宏：

```
318 #define BW_HOST_DELAY   1000L         /* nanoseconds */
```

它被定义为 1000L。BW 就是 BandWidth。这个宏对应于 spec 中的 Host_Delay。而 BitTime 对应于 spec 中的 BitStuffTime，仔细对比这个函数和 spec 中的这一堆公式，你会发现，这个函数真是一点创意也没有，完全是按照 spec 来办事。所以写代码的这些人如果来参加大学生挑战杯，那么等待他们的只能是早早被淘汰，连上 PK 台的机会都甭想有。

总之，这个函数返回的就是传输这些字节将会占有多少时间，单位是纳秒。在我们的故事中，usb_calc_bus_time 这个计算时间的函数只会出现在这一个地方，即每次咱们调用 uhci_alloc_qh 申请一个正常的 QH 时会被调用。（最开始建立框架时当然不会被调用。）而且只有针对中断传输和等时传输才需要申请带宽。这里我们把返回值赋给了 qh->load，赋值之前我们除了 1000，即把单位转换成为了微秒。

于是又结束 uhci_alloc_qh，回到了 uhci_urb_enqueue。当然我还想友情提醒一下，uhci_alloc_qh 中，第 267 行，对 qh->type 赋了值，这个值来自 struct usb_host_endpoint 结构体指针 hep，确切的说就是来自于端点描述符中的 bmAttributes 这一项。USB spec 2.0 中规定好了，这个属性的 Bit1 和 Bit0 两位表征了端点的传输类型。00 为控制，01 为等时，10 为 Bulk，11 为 Interrupt，而咱们这里的 USB_ENDPOINT_XFERTYPE_MASK 就是为了提取出这两个 bit 来。

于是回到 uhci_urb_enqueue 中以后，对 qh->type 进行判断，如果是中断传输类型，则 uhci_submit_interrupt 会被调用，依然来自 drivesr/usb/host/uhci-q.c：

```
1060 static int uhci_submit_interrupt(struct uhci_hcd *uhci, struct urb *urb,
1061              struct uhci_qh *qh)
1062 {
1063     int ret;
1064
1065     /* USB 1.1 interrupt transfers only involve one packet per interval.
1066      * Drivers can submit URBs of any length, but longer ones will need
1067      * multiple intervals to complete.
1068      */
1069
1070     if (!qh->bandwidth_reserved) {
1071         int exponent;
1072
1073         /* Figure out which power-of-two queue to use */
1074         for (exponent = 7; exponent >= 0; --exponent) {
1075             if ((1 << exponent) <= urb->interval)
1076                 break;
1077         }
1078         if (exponent < 0)
1079             return -EINVAL;
1080         qh->period = 1 << exponent;
1081         qh->skel = SKEL_INDEX(exponent);
1082
1083         /* For now, interrupt phase is fixed by the layout
1084          * of the QH lists. */
1085         qh->phase = (qh->period / 2) & (MAX_PHASE - 1);
1086         ret = uhci_check_bandwidth(uhci, qh);
1087         if (ret)
1088         return ret;
1089     } else if (qh->period > urb->interval)
1090         return -EINVAL;        /* Can't decrease the period */
1091
1092     ret = uhci_submit_common(uhci, urb, qh);
1093     if (ret == 0) {
1094         urb->interval = qh->period;
1095         if (!qh->bandwidth_reserved)
1096             uhci_reserve_bandwidth(uhci, qh);
1097     }
1098     return ret;
1099 }
```

首先 struct uhci_qh 中有一个成员 unsigned int bandwidth_reserved，顾名思义，用来表征是否申请了带宽的，对于等时传输和中断传输，是需要为之分配带宽的，带宽就是占用总线的时间，UHCI 的世界里，等时传输和中断传输这两者在每一个 Frame 内加起来是不可以超过该 Frame 的 90%的。

设置 bandwidth_reserved 为 1 只有一个地方，那就是 uhci_reserve_bandwidth 函数。而与之相反的一个函数 uhci_release_bandwidth 会把这个变量设置为 0。而调用 uhci_reserve_bandwidth 的又是谁呢？只有两个地方，一个恰恰就是这里的 uhci_submit_interrupt，另一个则是等时传输中要用到的 uhci_submit_isochronous。而释放这个带宽的函数 uhci_release_bandwidth 则是在 uhci_giveback_urb 中被调用。

1074 行，临时变量 exponent 从 7 开始，最多循环 8 次，把 1 左移 exponent 位就是进行指数运算，比如 exponent 为 1，左移以后就是 2 的 1 次方，exponent 为 7，则左移以后就是 2 的 7 次方。把这个数和

urb->interval 相比较，如果小于等于 urb->interval，就算找到了。这是什么意思呢？我们知道，UHCI 是 USB spec 1.1 的产物，那时候只有全速和低速设备，而 USB spec 中规定，对于全速设备来说，其 interval 必须在 1 毫秒到 255 毫秒之间，对于低速设备来说，其 interval 必须在 10 毫秒到 255 毫秒之间，所以这里 exponent 最多取 7 即可，2 的 7 次方就是 128。如果 interval 比 128 还大，那么就是处于 128 至 255 之间。而 interval 最小也不能小于 1，小于 1 也就出错了。那么从 1074 行到 1080 行这一段的目的是什么呢？就是根据 interval 确定最终的周期，就是说甭管您 interval 具体是多少，最终设定的周期都是 2 的整数次方，只要周期小于等于 interval，设备驱动就不会有意见。

SKEL_INDEX 这个宏我们贴出来过，struct uhci_qh 有一个成员 int skel，qh->skel 将被赋值为 9-exponent，即比如 exponent 为 1，qh->skel 就是 8。但同时我们知道，比如 exponent 为 3，那么说明 urb->interval 是介于 8 毫秒和 16 毫秒之间。而 qh->skel 为 8 意味着 qh 最终将挂在 skelqh[]数组的 skel int8 QH 后面。

此外，struct uhci_qh 另有两个元素：unsigned int period 和 short phase，刚才说了 period 就是周期，这里看到它被赋值为 2 的 exponent 次方，即比如 exponent 为 3，那么 period 就是 8。我们知道，标准情况下一个 Frame 是 1 毫秒，所以对于中断传输来说，这里的意思就是每 8 个 Frame 主机关心一次设备。MAX_PHASE 被定义为 32，此时我们还看不出来 phase 这个变量有什么用，到时候再看。

前面我们计算的是总线时间，现在还得转换成带宽的概念。uhci_check_bandwidth 函数就是用来检查带宽的，它来自 drivers/usb/host/uhci-q.c：

```
627 static int uhci_check_bandwidth(struct uhci_hcd *uhci, struct uhci_qh *qh)
628 {
629     int minimax_load;
630
631     /* Find the optimal phase (unless it is already set) and get
632      * its load value. */
633     if (qh->phase >= 0)
634         minimax_load = uhci_highest_load(uhci, qh->phase, qh->period);
635     else {
636         int phase, load;
637         int max_phase = min_t(int, MAX_PHASE, qh->period);
638
639         qh->phase = 0;
640     minimax_load = uhci_highest_load(uhci, qh->phase, qh->period);
641         for (phase = 1; phase < max_phase; ++phase) {
642             load = uhci_highest_load(uhci, phase, qh->period);
643             if (load < minimax_load) {
644                 minimax_load = load;
645                 qh->phase = phase;
646             }
647         }
648     }
649
650     /* Maximum allowable periodic bandwidth is 90%, or 900 us per frame*/
651     if (minimax_load + qh->load > 900) {
652         dev_dbg(uhci_dev(uhci), "bandwidth allocation failed: "
653                     "period %d, phase %d, %d + %d us\n",
654                     qh->period, qh->phase, minimax_load, qh->load);
655         return -ENOSPC;
656     }
657       return 0;
658 }
```

在提交中断类型的 urb 或者是等时类型的 urb 时，需要检查带宽，看带宽够不够了。这种情况下这个函数就会被调用。这个函数正常的话就将返回 0，负责就返回错误码-ENOSPC。不过你别小看这个函数，做程序员的最高境界就是像和尚研究佛法一样研究算法！所以写代码的人在这里用代码体现了他的境界。我们来仔细分析一下这个函数。

633 行判断 qh->phase 是否小于零，在 uhci_submit_interrupt 中设置了 qh->phase，显然从这个上下文来看 qh->phase 一定是大于等于 0 的，不过您别忘了，检测带宽这件事情在提交等时类型的 urb 时也会被调用，到时候你会发现，我们会把 qh->phase 设置为-1。所以再回过头来看这个函数，而现在，635 到 648 这一段先略过，因为现在不会被执行。现在只要关注 634 行就够了。uhci_highest_load 这个函数来自 drivers/usb/host/uhci-q.c：

```
614 static int uhci_highest_load(struct uhci_hcd *uhci, int phase, int period)
615 {
616     int highest_load = uhci->load[phase];
617
618     for (phase += period; phase < MAX_PHASE; phase += period)
619     highest_load = max_t(int, highest_load, uhci->load[phase]);
620     return highest_load;
621 }
```

代码本身超级简单，难的是这代码背后的哲学。struct uhci_hcd 有一个成员，short load[MAX_PHASE]，前面说过，MAX_PHASE 就是 32。所以这里就是为每一个 UHCI 主机控制器准备这么一个数组，来记录它的负载。这个数组 32 个元素，每一个元素就代表一个 Frame，所以这个数组实际上就是记录了一个主机控制器的 32 个 Frame 内的负载。我们知道一个 UHCI 主机控制器对应 1024 个 Frame 组成的 Frame List。但是软件角度来说，本着建设节约型社会的原则，没有必要申请一个 1024 的元素的数组，所以就申请 32 个元素。这个数组被称为“periodic load table”。于是这个函数所做的就是以 period 为步长，找到这个数组中最大的元素，即该 Frame 的负载最重。

得到这个最大的负载所对应的 Frame 之后，在 651 行计算这个负载加上刚才计算总线时间得到的那个 qh->load，这两个值不能超过 900，单位是微秒，因为一个 Frame 是一个毫秒，而 USB spec 规定了，等时传输和中断传输所占的带宽不能超过一个 Frame 的 90%。道理很简单，资源都被它们俩占了，别人就没法混了。无论如何也要为批量传输和控制传输着想一下。

于是 uhci_check_bandwidth 结束了，这里会调用 uhci_submit_common，这个函数在批量传输中已经讲过了，这是它们之间的公共函数，其执行过程也和批量传输一样，无非是通过 urb 得到 TD，依次调用 uhci_alloc_td，uhci_add_td_to_urbp 和 uhci_fill_td，完了之后设置最后一个 TD 的中断标志 TD_CTRL_IOC。

然后，uhci_submit_common 结束之后回到 uhci_submit_interrupt，剩下的代码也不多了，正常时返回 0，于是设置 urb->interval 为 qh->period，没有保留带宽就执行 uhci_reserve_bandwidth 去保留带宽。仍然来自 drivers/usb/host/uhci-q.c：

```
663 static void uhci_reserve_bandwidth(struct uhci_hcd *uhci, struct uhci_qh *qh)
664 {
665     int i;
666     int load = qh->load;
667     char *p = "??";
668
669     for (i = qh->phase; i < MAX_PHASE; i += qh->period) {
670         uhci->load[i] += load;
671         uhci->total_load += load;
672     }
673     uhci_to_hcd(uhci)->self.bandwidth_allocated = uhci->total_load / MAX_PHASE;
675     switch (qh->type) {
676     case USB_ENDPOINT_XFER_INT:
677         ++uhci_to_hcd(uhci)->self.bandwidth_int_reqs;
678         p = "INT";
679         break;
680     case USB_ENDPOINT_XFER_ISOC:
681         ++uhci_to_hcd(uhci)->self.bandwidth_isoc_reqs;
682         p = "ISO";
683         break;
684     }
685     qh->bandwidth_reserved = 1;
686     dev_dbg(uhci_dev(uhci),
687                     "%s dev %d ep%02x-%s, period %d, phase %d, %d us\n",
688                     "reserve", qh->udev->devnum,
689                     qh->hep->desc.bEndpointAddress, p,
690                     qh->period, qh->phase, load);
691 }
```

其实这个函数也挺简单的。uhci->load 数组就是在这个函数这里被赋值的。当然它的“情侣”函数 uhci_release_bandwidth 里面也会改变这个数组。而 uhci->total_load 则是把所有的负担（load）全都加到一起去。而 bandwidth_allocated 则是 total_load 除以 32，即一个平均值。

然后根据 QH 是中断类型还是等时类型，分别增加 bandwidth_int_reqs 和 bandwidth_isoc_reqs。这两个都是 struct usb_bus 的 int 类型成员，前者记录中断请求的数量，后者记录等时请求的数量。

最后设置 qh->bandwidth_reserved 为 1。这个函数就结束了。这样，uhci_submit_interrupt 这个函数也结束了。终于回到了 uhci_urb_enqueue。

1426 行，把 qh 赋给 urbp 的 qh。

把 urbp 给链入到 qh 的队列中来。qh 里面专门有一个队列记录它所领导的各个 urbp。因为一个端点对应一个 QH，而该端点可以有多个 urb，所以就把它们都排成一个队列。

1433 行，如果这个队列的下一个节点就是现在这个 urbp，并且 qh 没有停止，则调用 uhci_activate_qh()和 uhci_urbp_wants_fsbr()。这两个函数当初在控制传输中就已经讲过了，不过对于 uhci_activate_qh()现在进去看会有所不同。

在 514 行开始的这一小段判断中，看到的是对 qh->skel 进行的判断，这是一个 int 型的变量，当初在 uhci_submit_interrupt 中对这个变量进行了赋值，赋的值是 SKEL_INDEX(exponent)。很显然它小于 SKEL_ASYNC，所以这里 link_interrupt 会被执行。这个函数来自 drivers/usb/host/uhci-q.c：

```
439 static void link_interrupt(struct uhci_hcd *uhci, struct uhci_qh *qh)
440 {
441     struct uhci_qh *pqh;
442
443     list_add_tail(&qh->node, &uhci->skelqh[qh->skel]->node);
444
445     pqh = list_entry(qh->node.prev, struct uhci_qh, node);
446     qh->link = pqh->link;
447     wmb();
448     pqh->link = LINK_TO_QH(qh);
449 }
```

把 qh 的 node 给链入到 uhci->skelqh[qh->skel]的 node 链表中去。然后让这个 qh 的 link 指向前一个 qh 的 link，并且把前一个 qh 的 link 指针指向这个 qh。这就是典型的队列插入的操作。很明显这里又是物理地址的链接。

uhci_activate_qh 就算执行完了。剩下的代码就和控制传输/批量传输一样了。uhci_urb_enqueue 也就这样结束了，usb_hcd_submit_urb 啊，usb_submit_urb 啊，也纷纷跟着结束了。似乎调用 usb_submit_urb 提交了一个中断请求的 urb 之后整个世界没有发生任何变化，完全没有看出咱们这个函数对这个世界的影响，俨然这个函数的调用没有任何意义，但我要告诉你，其实不是的，这次函数调用就像流星，短暂的划过却能照亮整个天空。此刻，让我们利用 debugfs 来看个究竟，当我们没有提交任何 urb 时，/sys/kernel/debug/uhci 目录下面的文件是这个样子的：

```
localhost:~ # cat /sys/kernel/debug/uhci/0000\:00\:1d.1
Root-hub state: suspended   FSBR: 0
HC status
  usbcmd    =    0048   Maxp32 CF EGSM
  usbstat   =    0020   HCHalted
  usbint    =    0002
  usbfrnum  =   (1)168
  flbaseadd = 194a9168
  sof       =      40
  stat1     =    0080
  stat2     =    0080
Most recent frame: 45a (90)   Last ISO frame: 45a (90)
Periodic load table
       0      0      0      0      0      0      0      0
       0      0      0      0      0      0      0      0
       0      0      0      0      0      0      0      0
       0      0      0      0      0      0      0      0
Total: 0, #INT: 0, #ISO: 0
Frame List
Skeleton QHs
- skel_unlink_qh
    [d91a1000] Skel QH link (00000001) element (00000001)
      queue is empty
- skel_iso_qh
    [d91a1060] Skel QH link (00000001) element (00000001)
      queue is empty
- skel_int128_qh
    [d91a10c0] Skel QH link (191a1362) element (00000001)
```

```
      queue is empty
  - skel_int64_qh
     [d91a1120] Skel QH link (191a1362) element (00000001)
      queue is empty
  - skel_int32_qh
     [d91a1180] Skel QH link (191a1362) element (00000001)
      queue is empty
  - skel_int16_qh
     [d91a11e0] Skel QH link (191a1362) element (00000001)
      queue is empty
  - skel_int8_qh
     [d91a1240] Skel QH link (191a1362) element (00000001)
      queue is empty
  - skel_int4_qh
     [d91a12a0] Skel QH link (191a1362) element (00000001)
      queue is empty
  - skel_int2_qh
     [d91a1300] Skel QH link (191a1362) element (00000001)
      queue is empty
  - skel_async_qh
     [d91a1360] Skel QH link (00000001) element (197bd000)
      queue is empty
  [d97bd000] link (00000001) e0 Length=0 MaxLen=7ff DT0 EndPt=0 Dev=7f, PID=69(IN) (buf=00000000)
  - skel_term_qh
     [d91a13c0] Skel QH link (191a13c2) element (197bd000)
      queue is empty
```

可以看到，那 11 个 skel QH 都被打印了出来，link 后面的括号里面的东西是 link 的地址，element 后面的括号里面的东西是 element 的地址。这个时候整个调度框架中没有任何有实质意义的 QH 或者 TD。

Periodic load table 后面打印出来的是 uhci->load[]数组的 32 个元素，看到这时候这 32 个元素全是 0，因为目前没有任何中断调度或者等时调度。下面我们做一个实验，往 USB 端口里插入一个 USB 键盘，然后加载其驱动程序，比如 usbhid 模块，再来看同一个文件：

```
  localhost:~ # cat /sys/kernel/debug/uhci/0000\:00\:1d.1
  Root-hub state: running   FSBR: 0
  HC status
    usbcmd    =    00c1   Maxp64 CF RS
    usbstat   =    0000
    usbint    =    000f
    usbfrnum  =   (1)a70
    flbaseadd = 194a9a70
    sof       =      40
    stat1     =    0080
    stat2     =    01a5   LowSpeed Enabled Connected
  Most recent frame: 8ae66 (614)   Last ISO frame: 8ae66 (614)
  Periodic load table
         0      0      0      0      0      0      0      0
         118    0      0      0      0      0      0      0
         0      0      0      0      0      0      0      0
         118    0      0      0      0      0      0      0
  Total: 236, #INT: 1, #ISO: 0
  Frame List
  Skeleton QHs
  - skel_unlink_qh
     [d91a1000] Skel QH link (00000001) element (00000001)
      queue is empty
  - skel_iso_qh
     [d91a1060] Skel QH link (00000001) element (00000001)
      queue is empty
  - skel_int128_qh
     [d91a10c0] Skel QH link (191a1362) element (00000001)
      queue is empty
  - skel_int64_qh
     [d91a1120] Skel QH link (191a1362) element (00000001)
      queue is empty
  - skel_int32_qh
     [d91a1180] Skel QH link (191a1362) element (00000001)
      queue is empty
  - skel_int16_qh
     [d91a11e0] Skel QH link (191a1482) element (00000001)
      queue is empty
     [d91a1480] INT QH link (191a1362) element (197bd0c0)
        period 16 phase 8 load 118 us
```

```
    urb_priv [d4b31720] urb [d9d84440] qh [d91a1480] Dev=2 EP=1(IN) INT Actlen=0
          1: [d97bd0c0] link (197bd030) e3 LS IOC Active NAK Length=7ff MaxLen=7 DT0 EndPt=1 Dev=2,
PID=69(IN) (buf=18c69000)
        Dummy TD
    [d97bd030] link (197bd060) e0 Length=0 MaxLen=7ff DT0 EndPt=0 Dev=0, PID=e1(OUT) (buf=00000000)
    - skel_int8_qh
       [d91a1240] Skel QH link (191a1362) element (00000001)
         queue is empty
    - skel_int4_qh
       [d91a12a0] Skel QH link (191a1362) element (00000001)
         queue is empty
    - skel_int2_qh
       [d91a1300] Skel QH link (191a1362) element (00000001)
         queue is empty
    - skel_async_qh
       [d91a1360] Skel QH link (00000001) element (197bd000)
         queue is empty
    [d97bd000] link (00000001) e0 Length=0 MaxLen=7ff DT0 EndPt=0 Dev=7f, PID=69(IN) (buf=00000000)
    - skel_term_qh
       [d91a13c0] Skel QH link (191a13c2) element (197bd000)
         queue is empty
```

最显著的两个变化是：第一，Periodic load table 这张表不再全是 0 了；第二，在 skel_int16_qh 下面不再是空空如也了，有一个 int QH 了，有一个 urb_priv 了，这个 int QH 的周期是 16，phase 是 8，load 是 118μs。对照 Periodic load table，再结合这三个数字，你是不是能明白 phase 的含义了。没错，load 这个数组一共 32 个元素，编号从 0 开始到 31 结束，周期是 16 就意味着每隔 16 ms 这个中断传输会被调度一次，phase 是 8 就意味着它的起始点位于编号为 8 的位置，即从 8 开始，8，24，40，56，……每隔 16 ms 就安置一个中断传输的调度。而 118μs 则是它在每个 Frame 中占据总线的时间。

至此，既了解了中断传输的处理，也了解了 debugfs 在 uhci-hcd 模块中的应用。文件 drivers/usb/host/uhci-debug.c 一共 592 行就是努力让我们能够在/sys/kernel/debug/目录下面看到刚才这些信息。实际上通过以上 sysfs 提供的信息，对于整个 uhci-hcd 的结构也有了很好的了解，之前的任何一个数据结构，比如 skel_term_qh，Dummy_TD，整个 skelqh 数组，link， element，periodic load table 这一切的一切，都通过这些信息展现得淋漓尽致。也正是通过这些信息，我们才真正体会到了 skelqh 这个数组的意义和价值，没有它们构建的基础框架，毫无疑问，在 uhci-hcd 中使用 skelqh 这个数组是一个无比英明的决定。尽管有人觉得 skelqh 的存在浪费了内存，而且搞得代码看上去复杂了许多，但它确实非常实用。

17、等时传输

由于等时传输的特殊性，很多地方它都被特别地对待了。从 usb_submit_urb 开始就显示出了它的与众不同了。该函数中 268 行，判断 temp 是不是 PIPE_ISOCHRONOUS，即是不是等时传输，如果是，就执行下面那段代码。

278 行，int number_of_packets 是 struct urb 的一个成员，它用来指定该 urb 所处理的等时传输缓冲区的数量，或者说这个等时传输要传输多少个 packet，每一个 packet 用一个 struct usb_iso_packet_descriptor 结构体变量来描述。对于每一个 packet，需要建立一个 td。

同时，还注意到 struct urb 有另外一个成员，struct usb_iso_packet_descriptor iso_frame_desc[0]，又是一个零长度数组，这个数组用来帮助这个 urb 定义多个等时传输，而这个数组的实际长度恰恰就是我们前面提到的那个 number_of_packets。设备驱动程序在提交等时传输 urb 时，必须设置好 urb 的 iso_frame_desc 数组。有人提问，为何 iso_frame_desc 数组的长度恰好是 number_of_packets？从哪里看出来的？还记得很久很久以前，曾经讲过一个叫做 usb_alloc_urb()的函数么？不管是在 usb-storage 中还是在 Hub 驱动中，都见过这个函数，它的作用就是申请 urb，你或许忘记了这个函数的参数，在 include/linux/usb.h 中找到它的原型：

```
    1266 extern struct urb *usb_alloc_urb(int iso_packets,  gfp_t mem_flags);
```

这其中第一个参数，iso_packets，其实就是咱们这里的 number_of_packets。所以，设备驱动在申请等时 urb 时，必须指定需要传输多少个 packets。usb_alloc_urb()， 来自 drivers/usb/core/urb.c：

```
56 struct urb *usb_alloc_urb(int iso_packets, gfp_t mem_flags)
57 {
58      struct urb *urb;
59
60      urb = kmalloc(sizeof(struct urb) +
61              iso_packets * sizeof(struct usb_iso_packet_descriptor),
62              mem_flags);
63      if (!urb) {
64          err("alloc_urb: kmalloc failed");
65          return NULL;
66      }
67      usb_init_urb(urb);
68      return urb;
69 }
```

再一次看到了零长度数组的应用，或者叫做变长度数组的应用。struct usb_iso_packet_descriptor 的定义来自 include/linux/usb.h：

```
952 struct usb_iso_packet_descriptor {
953     unsigned int offset;
954     unsigned int length;          /* expected length */
955     unsigned int actual_length;
956     int status;
957 };
```

这个结构体的意思很简洁明了。这个结构体描述的就是一个 iso 包。而 urb 的 iso_frame_desc 数组的元素都是在设备驱动提交 urb 之前就设置好了。其中 length 就如注释里说的一样，是期待长度。而 actual_length 是实际长度，这里我们先把它设置为 0。

至于 348 行，对于高速设备，如果 urb->interval 大于 1024*8，则设置为 1024*8，注意这里单位是微帧，即 125μs，以及 360 行，对于全速设备的 ISO 传输，如果 urb->interval 大于 1024，则设置为 1024，注意这里单位是帧，即 1 ms。关于这两条，Alan Stern 的解释是，由于主机控制器驱动中并不支持超过 1024 个 ms 的间隔，（想想也很简单，比如 UHCI 吧，总共 Frame List 才 1024 个元素，你这个间隔期总不能超过它吧，要不还不乱了去）。

进入 usb_hcd_submit_urb，因为 Root Hub 是不会有等时传输的，所以针对非 Root Hub，调用 uhci_urb_enqueue。1419 行，调用 uhci_submit_isochronous()。这个函数来自 drivers/usb/host/uhci-q.c：

```
1231 static int uhci_submit_isochronous(struct uhci_hcd *uhci, struct urb *urb,
1232              struct uhci_qh *qh)
1233 {
1234     struct uhci_td *td = NULL;     /* Since urb->number_of_packets > 0 */
1235     int i, frame;
1236     unsigned long destination, status;
1237     struct urb_priv *urbp = (struct urb_priv *) urb->hcpriv;
1238
1239     /* Values must not be too big (could overflow below) */
1240     if (urb->interval >= UHCI_NUMFRAMES ||
1241                     urb->number_of_packets >= UHCI_NUMFRAMES)
1242         return -EFBIG;
1243
1244     /* Check the period and figure out the starting frame number */
1245     if (!qh->bandwidth_reserved) {
1246         qh->period = urb->interval;
1247         if (urb->transfer_flags & URB_ISO_ASAP) {
1248             qh->phase = -1;        /* Find the best phase */
1249              i = uhci_check_bandwidth(uhci, qh);
1250             if (i)
1251                 return i;
1252
1253             /* Allow a little time to allocate the TDs */
1254             uhci_get_current_frame_number(uhci);
1255             frame = uhci->frame_number + 10;
1256
1257             /* Move forward to the first frame having the
1258              * correct phase */
1259             urb->start_frame = frame + ((qh->phase - frame) &
1260                                 (qh->period - 1));
1261         } else {
1262             i = urb->start_frame - uhci->last_iso_frame;
1263             if (i <= 0 || i >= UHCI_NUMFRAMES)
```

```
1264                return -EINVAL;
1265            qh->phase = urb->start_frame & (qh->period - 1);
1266            i = uhci_check_bandwidth(uhci, qh);
1267            if (i)
1268                return i;
1269        }
1270
1271    } else if (qh->period != urb->interval) {
1272        return -EINVAL;       /* Can't change the period */
1273
1274    } else {       /* Pick up where the last URB leaves off */
1275        if (list_empty(&qh->queue)) {
1276            frame = qh->iso_frame;
1277        } else {
1278            struct urb *lurb;
1279
1280            lurb = list_entry(qh->queue.prev,
1281                              struct urb_priv, node)->urb;
1282            frame = lurb->start_frame +
1283                              lurb->number_of_packets *
1284                              lurb->interval;
1285        }
1286        if (urb->transfer_flags & URB_ISO_ASAP)
1287            urb->start_frame = frame;
1288        else if (urb->start_frame != frame)
1289            return -EINVAL;
1290    }
1291
1292    /* Make sure we won't have to go too far into the future */
1293    if (uhci_frame_before_eq(uhci->last_iso_frame + UHCI_NUMFRAMES,
1294                    urb->start_frame + urb->number_of_packets *
1295                    urb->interval))
1296        return -EFBIG;
1297
1298    status = TD_CTRL_ACTIVE | TD_CTRL_IOS;
1299   destination=(urb->pipe & PIPE_DEVEP_MASK)|usb_packetid(urb->pipe);
1300
1301    for (i = 0; i < urb->number_of_packets; i++) {
1302        td = uhci_alloc_td(uhci);
1303        if (!td)
1304            return -ENOMEM;
1305
1306        uhci_add_td_to_urbp(td, urbp);
1307        uhci_fill_td(td, status, destination |
1308                    uhci_explen(urb->iso_frame_desc[i].length),
1309                    urb->transfer_dma +
1310                    urb->iso_frame_desc[i].offset);
1311    }
1312
1313    /* Set the interrupt-on-completion flag on the last packet. */
1314    td->status |= __constant_cpu_to_le32(TD_CTRL_IOC);
1315
1316    /* Add the TDs to the frame list */
1317    frame = urb->start_frame;
1318    list_for_each_entry(td, &urbp->td_list, list) {
1319        uhci_insert_td_in_frame_list(uhci, td, frame);
1320        frame += qh->period;
1321    }
1322
1323    if (list_empty(&qh->queue)) {
1324        qh->iso_packet_desc = &urb->iso_frame_desc[0];
1325        qh->iso_frame = urb->start_frame;
1326        qh->iso_status = 0;
1327    }
1328
1329    qh->skel = SKEL_ISO;
1330    if (!qh->bandwidth_reserved)
1331        uhci_reserve_bandwidth(uhci, qh);
1332    return 0;
1333 }
```

1240 行，UHCI_NUMFRAMES 是 1024，同样，urb 的 interval 显然不能比这个还大，它的 number_of_packets 也不能比这个大，要不然肯定就溢出了。就像伤痛，当眼泪掉下来，一定是伤痛已经超载。

接下来看，URB_ISO_ASAP 这个 flag 是专门给等时传输用的，它的意思就是告诉驱动程序，只要带宽允许，那么就从此点开始设置这个 urb 的 start_frame 变量。通常为了尽可能快地得到图像数据，应当在 urb 中指定这个 flag，因为它意味着尽可能快发出本 urb。比如说，之前有一个 urb，是针对 iso 端点的，假设它有两个 packets，被安排在 Frame 号 108 和 109，即假设其 interval 是 1。现在再假设新的一个 urb 是在 Frame 111 被提交的，如果设置了 URB_ISO_ASAP 这个 flag，那么这个 urb 的第一个 packet 就会在下一个可以接受的 Frame 中被执行，比如 Frame 112。但是如果没有设置这个 URB_ISO_ASAP 的 flag 呢？这个 packet 就会被安排在上一个 urb 结束之后的下一个 Frame，即 110。尽管 Frame 110 已经过去了，但是这种调度仍然有意义，因为它可以保证一定接下来的 packets 处于特定的阶段，因为有时，驱动程序并不在乎丢掉一些包，尤其是等时传输。

我们看到这里 qh 的 phase 被设置为了-1。所以在 uhci_check_bandwidth 函数中面有一个判断条件是 qh 的 phase 是否大于等于 0。如果调用 uhci_check_bandwidth 之前设置了 phase 大于等于 0，则表明咱们手工设置了 phase，否则可通过一种算法来选择出一个合适的 phase。这个函数正常应该返回 0。

接下来是 uhci_get_current_frame_number()：

```
441 static void uhci_get_current_frame_number(struct uhci_hcd *uhci)
442 {
443     if (!uhci->is_stopped) {
444         unsigned delta;
445
446         delta = (inw(uhci->io_addr + USBFRNUM) - uhci->frame_number) &
447                         (UHCI_NUMFRAMES - 1);
448         uhci->frame_number += delta;
449     }
450 }
```

UHCI 主机控制器有一个 Frame 计数器，Frame 从 0 到 1023，然后又从 0 开始，那么这个数到底是多少呢？这个函数就是用来获得这个值的，读了端口和 USBFRNUM 寄存器。uhci->frame_number 用来记录 Frame Number，所以这里的做法就是把当前的 Frame Number 减去上次保存在 uhci->frame_number 中的值，然后转换成二进制，得到一个差值，再更新 UHCI 的 frame_number。

而 start_frame 就是这个传输开始的 Frame。这里让 Frame 等于当前的 Frame 加上 10，就是给个延时，如注释所说的那样，给内存申请一点点时间。再让 start_frame 等于 Frame 加上（qh->phase-frame）和（qh->period-1）相与。熟悉二进制运算的同志们应该不难知道这样做最终得到的 start_frame 是什么，很显然，它会满足 phase 的要求。

1261 行，else，就是驱动程序指定了 start_frame，这种情况下就是直接设置 phase，last_iso_frame 就对应于刚才这个例子中的 frame 109。

1293 行，uhci_frame_before_eq 就是一个普通的宏，来自 drivers/usb/host/uhci-hcd.h：

```
441 /* Utility macro for comparing frame numbers */
442 #define uhci_frame_before_eq(f1, f2)    (0 <= (int) ((f2) - (f1)))
```

其实就是比较两个 Frame Number，如果第二个比第一个大的话，就返回真，反之就返回假。而咱们这里代码的意思是，如果第二个比第一个大，那么说明出错了。last_iso_frame 是记录着上一次扫描时的 Frame 号，在 uhci_scan_schedule 中会设置，UHCI_NUMFRAMES 我们知道是 1024。urb 的 number_of_packets 与 interval 的乘积就表明将要花掉多少时间，它们加上 urb 的 start_frame 就等于这些包传输完之后的时间，或者说 Frame Number。这里的意思就是希望一次传输的东西别太大了，不能越界。-EFBIG 这个错误码的含义本身就是文件太大（File too large）。

1298 行，TD_CTRL_IOS，对应于 TD 的 bit25，IOS 的意思是 Isochronous Select，这一位为 1 表示这 TD 是一个等时传输描述符，即 Isochronous Transfer Descriptor。如果为 0 则表示这是一个非等时传输描述符。等时传输的 TD 在执行完之后会被主机控制器设置为 inactive，不管执行的结果是什么。下面还设置了 TD_CTRL_IOC，告诉主机控制器在 TD 执行的 Frame 结束时发送一个中断。

然后根据 packets 的数量申请 TD，再把本 urb 的各个 TD 加入到 frame list 中去。uhci_insert_td_in_frame_list 是来自 drivers/usb/host/uhci-q.c：

```
159 static inline void uhci_insert_td_in_frame_list(struct uhci_hcd *uhci,
160               struct uhci_td *td, unsigned framenum)
161 {
162     framenum &= (UHCI_NUMFRAMES - 1);
163
164     td->frame = framenum;
165
166     /* Is there a TD already mapped there? */
167     if (uhci->frame_cpu[framenum]) {
168         struct uhci_td *ftd, *ltd;
169
170         ftd = uhci->frame_cpu[framenum];
171         ltd = list_entry(ftd->fl_list.prev, struct uhci_td, fl_list);
172
173         list_add_tail(&td->fl_list, &ftd->fl_list);
174
175         td->link = ltd->link;
176         wmb();
177         ltd->link = LINK_TO_TD(td);
178     } else {
179         td->link = uhci->frame[framenum];
180         wmb();
181         uhci->frame[framenum] = LINK_TO_TD(td);
182         uhci->frame_cpu[framenum] = td;
183     }
184 }
```

只有等时传输才需要使用这个函数。先看 else 这一段，让 td 在物理上指向 uhci 的 frame 数组中对应元素，framenum 是传递进来的参数，其实就是 urb 的 start_frame。而 frame 数组里又设置为 td 的物理地址。要知道之前曾经在 configure_hc 中把 frame 和实际的硬件的 frame list 给联系了起来，因此只要把 td 和 frame 联系起来就等于和硬件联系了起来，另一方面这里又把 frame_cpu 和 td 联系起来，所以以后只要直接通过 frame_cpu 来操作队列即可。正如下面在 if 段所看到的那样。

来看 if 这一段，struct uhci_td 有一个成员 struct list_head fl_list，struct uhci_hcd 中有一个成员 void **frame_cpu，当初在 uhci_start 函数中为 uhci->frame_cpu 申请好了内存，而刚才在 else 里面看到每次会把 frame_cpu 数组的元素赋值为 td，所以这里就是把 td 通过 fl_list 链入到 ftd 的 fl_list 队列里去。而物理上，也把 td 给插入到这个队列中来。

如果 qh 的 queue 为空，即没有任何 urb，就设置 qh 的几个成员，iso_packet_desc 是下一个 urb 的 iso_frame_desc，iso_frame 则是该 iso_packet_desc 的 frame 号，iso_status 则是该 iso urb 的状态。

最后，令 qh->skel 等于 SKEL_ISO，然后调用 uhci_reserve_bandwidth 保留带宽。

至此，uhci_submit_isochronous 就结束了。回到 uhci_urb_enqueue，下一步执行，uhci_activate_qh，而在这个函数中，我们将调用 link_iso。

而 link_iso 同样来自 drivers/usb/host/uhci-q.c：

```
428 static inline void link_iso(struct uhci_hcd *uhci, struct uhci_qh *qh)
429 {
430     list_add_tail(&qh->node, &uhci->skel_iso_qh->node);
431
432     /* Isochronous QHs aren't linked by the hardware */
433 }
```

这就简单多了，直接加入到 skel_iso_qh 中去就可以了。

终于，四大传输也就这样结束了。而我们的故事也即将 ALT+F4 了。我只是说也许。

如果失败的人生可以 F5，如果莫名的悲伤可以 DEL；

如果逝去的岁月可以 CTRL＋C，如果甜蜜的往事可以 CTRL＋V；

如果一切都可以 CTRL＋ALT＋DEL，那么我们所有的故事是不是永远都不会 ALT＋F4？

18、“脱”就一个字

我们知道，整个故事里一直围绕着 QH 的队列在说来说去，不停地进行着队列操作，有时候把 QH 连接（link）起来成一个个的队列，而有时候又把 QH 从队列里给“unlink”，unlink 翻译成中文就是解开，拆开，松开。Okay，简洁一点说，一个字，脱！

还记得 skel_unlink_qh 么？skelqh[]数组里边 11 个元素，另外那 10 个都知道怎么回事了，但是第一个元素，或者说这 0 号元素，一直就不太明白。现在就来仔细解读一下。

事实上，uhci_unlink_qh 函数有这么一句话：list_move_tail(&qh->node，&uhci->skel_unlink_qh->node)，换言之，凡是调用过 uhci_unlink_qh 的 QH，最终都被加入到了由 skel_unlink_qh 领衔的队列。但问题是加入这个队列之后呢？是不是就算隐退江湖了？其实不然，生活哪有那么简单啊？不是想退出江湖就能退出江湖的。咱们回过头来看这个函数，uhci_scan_schedule：

```
1708 static void uhci_scan_schedule(struct uhci_hcd *uhci)
1709 {
1710     int i;
1711     struct uhci_qh *qh;
1712
1713     /* Don't allow re-entrant calls */
1714     if (uhci->scan_in_progress) {
1715         uhci->need_rescan = 1;
1716         return;
1717     }
1718     uhci->scan_in_progress = 1;
1719 rescan:
1720     uhci->need_rescan = 0;
1721     uhci->fsbr_is_wanted = 0;
1722
1723     uhci_clear_next_interrupt(uhci);
1724     uhci_get_current_frame_number(uhci);
1725     uhci->cur_iso_frame = uhci->frame_number;
1726
1727     /* Go through all the QH queues and process the URBs in each one */
1728     for (i = 0; i < UHCI_NUM_SKELQH - 1; ++i) {
1729         uhci->next_qh = list_entry(uhci->skelqh[i]->node.next,
1730                         struct uhci_qh, node);
1731         while ((qh = uhci->next_qh) != uhci->skelqh[i]) {
1732             uhci->next_qh = list_entry(qh->node.next,
1733                             struct uhci_qh, node);
1734
1735             if (uhci_advance_check(uhci, qh)) {
1736                 uhci_scan_qh(uhci, qh);
1737                 if (qh->state == QH_STATE_ACTIVE) {
1738                     uhci_urbp_wants_fsbr(uhci,
1739                     list_entry(qh->queue.next, struct urb_priv, node));
1740                 }
1741             }
1742         }
1743     }
1745     uhci->last_iso_frame = uhci->cur_iso_frame;
1746     if (uhci->need_rescan)
1747         goto rescan;
1748     uhci->scan_in_progress = 0;
1749
1750     if (uhci->fsbr_is_on && !uhci->fsbr_is_wanted &&
1751                     !uhci->fsbr_expiring) {
1752         uhci->fsbr_expiring = 1;
1753         mod_timer(&uhci->fsbr_timer, jiffies + FSBR_OFF_DELAY);
1754     }
1756     if (list_empty(&uhci->skel_unlink_qh->node))
1757         uhci_clear_next_interrupt(uhci);
1758     else
1759         uhci_set_next_interrupt(uhci);
1760 }
```

1728 行的 for 循环，对 skelqh[]数组从 0 开始循环，直到 9。1 到 9 就不说了，而顺着 0 往下看，针对 skel_unlink_qh 队伍里的每一个 qh 进行循环，每一个 qh 执行一次 uhci_advance_check()，而 skel_unlink_qh 这个队伍里的 qh 有一部分是上一次刚刚在 uhci_advance_check 中设置了 wait_expired 为 1 的，另一部分可能没有设置过，因为调用 uhic_unlink_qh()的并非只有 uhci_advance_check()，还有别的地方。而别的地方调用它的话就和超时不超时没有关系了。

于是现在分两种情况来看待这个 uhci_advance_check。第一种，qh 是因为超时被拉进 skel_unlink_qh 的，那么 1673 行 if 条件是满足的，这种情况下 uhci_advance_check 就直接返回了，但是返回的肯定是 1。返回了之后来到 uhci_scan_schedule，1736 行，uhci_scan_qh 就会被执行，进入到 uhci_scan_qh 中，1602 行，由于 qh 中还有 urb，1617 行的 uhci_activate_qh 就会被执行，因此 qh 将重新激活，qh->state 会被设置为 QH_STATE_ACTIVE，它会再次被拉入它自己的归属。于是又幸运地获得了重生。

而对于第二种情况，也是通过 uhci_unlink_qh()给拉入 skel_unlink_qh 了。但是人家起码没超时，所以这次再看 uhci_advance_check，1673 行这个 if 条件就不一定满足了。然后如果真没超时，那么 1697 行会被执行，而 1697 这个 if 条件是否满足得看 1654 行这个 if 是否满足了，如果 1654 行满足，换言之，qh->state 不是 QH_STATE_ACTIVE，则设置 urbp 为空，而我们知道 uhci_unlink_qh 会把 qh->state 设置为 QH_STATE_UNLINKING，所以，1654 行肯定满足。而 urbp 设置为了 NULL，因此 1697 行这个 if 不会满足。因此，对于 unlink 的 qh，uhci_advance_check 这次除了返回 1 其他什么也不做，但是回到了 uhci_scan_schedule 之后，uhci_scan_qh 会执行，1622 行。

```
1532 #define QH_FINISHED_UNLINKING(qh)                       \
1533         (qh->state == QH_STATE_UNLINKING &&    \
1534             uhci->frame_number + uhci->is_stopped != qh->unlink_frame)
```

首先 qh->unlink_frame 当初是在 uhci_unlink_qh()中设置的，设置的就是当时的 Frame 号。而 uhci->frame_number 是当前的 Frame 号。但对于眼前这个宏的含义，我曾经一度困惑过。我猜测这个宏判断的就是一个 QH 是否已经彻底失去利用价值，但我并不清楚为什么这个宏被这样定义。后来，Alan Stern 语重心长地说：

“When a QH is unlinked, the controller is allowed to continue using it until the end of the frame. So the unlink isn't finished until the frame is over and a new frame has begun. qh->unlink_frame is the frame number when the QH was unlinked. uhci->frame_number is the current frame number. If the two are unequal then the unlink is finished.”

没错，当一个 QH 在某个 Frame 被“unlink”了之后，在这个 Frame 结束之后主机控制器就不会再使用它了。也就是说，到下一个 Frame 开始，这个 QH 就算是真正彻底地完成了它的“脱”。

这里尤其需要注意的是 uhci->is_stopped，顾名思义，当 UHCI 正常工作时这个值应该为 0，而只有 UHCI 停了下来时，这个值才会是非 0。但我们知道，如果主机控制器自己都停止了下来，那么显然这个 QH 就算是彻底脱了，因为主机控制器不可能再使用它了，或者主机控制器不可能再访问它了，停下来了就意味着 is_stopped 不为 0，显然只要 is_stopped 不为 0，则 uhci->frame_number+uhci->is_stopped 是不可能等于 qh->unlink_frame 的。（uhci->frame_number>=qh->unlink_frame 恒成立，而 is_stopped 永远大于等于 0。）

所以，1622 行这个宏这么一判断，发现 QH 确实已经没有利用价值了，就调用 uhci_make_qh_idle 从而把 qh->state 设置为 QH_STATE_IDLE，并且把本 QH 拉入 uhci->idle_qh_list，一旦加入这个 list，这个 QH 将从此永不见天日，没有人会再去理睬它了。

不过，最后，关于 uhci_scan_qh，有三点需要强调一下。

第一点，1569 行调用了 uhci_giveback_urb()，为何在 1594 行也调用了 uhci_giveback_urb。但事实上你会发现任何一个 urb 都不可能在这两处先后被调用，要么在前者被调用，要么在后者被调用。道理很简单，一旦调用了 uhci_giveback_urb()，那么其 urbp 就会脱离 QH 的队列。这是 uhci_giveback_urb()中 1507 行 list_del_init 干的。甚至 urbp 的内存也会被释放掉，这是 uhci_giveback_urb()中 1514 行 uhci_free_urb_priv()函数干的。

所以，事实上，对于大多数正常工作正常结束的 urb，在 uhci_scan_qh 中，1569 行这个 uhci_giveback_urb 会被调用，而一旦调用了，这个 urbp 就不复存在了，因此之后的代码就跟它毫无关系了。

那么另一方面，1551 行和 1571 行这两个 break 语句的存在使得 while 循环有可能提前结束，这就意味着，while 循环结束时，qh->queue 里面的 urbp 并不一定全部被遍历到了，因此，也就是说有些 urb 可能并没有执行 1569 行的 uhci_giveback_urb()，因此它们就有可能在 1594 行被传递给 uhci_giveback_urb。

第二点，1595 行 goto restart 是什么意思？乍一看，我愣是以为这个 goto restart 会导致这段代码成为死循环，可后来我算是琢磨出来了，list_for_each_entry 不是想遍历 qh->queue 这个 urbp 构成的队列么？可是每次如果它走到 1594 行这个 uhci_giveback_urb 的话，该 urbp 会被删掉，于是队列就改变了，好家伙，你在调用 list_for_each_entry 遍历队列时改变队列那还能不出事？所以也就甭犹豫了，重新调用 list_for_each_entry，重新遍历不就成了么？

第三点，虽然 uhci_scan_qh 函数看上去挺复杂，但是正如 1574 行这个注释所说的那样，事实上对于大多数情况来说，这个函数执行到 1579 行就会返回。只有两种情况下才会执行 1579 之后的代码，第一个就是 1576 行 QH_FINISHED_UNLINKING 条件满足，即这个 QH 是刚刚被“unlink”，刚刚完成“脱”的，这种情况下要继续往下走，第二个就是虽然不是完成了“脱”的，但 is_stopped 不为 0。

但是虽然说这两种情况是小概率事件，但毕竟这节讨论的就是 QH_FINISHED_UNLINKING，所以这种情况究竟怎么处理还是要关注的。而这其中除了 1623 行 uhci_make_qh_idle 是最后一步要做的事情之外，1590 行，uhci_cleanup_queue 也没有仔细看过。既然整个故事已经进入到了 cleanup 的阶段，那么就以这个 cleanup 函数作为结束吧。它来自 drivers/usb/host/uhci-q.c：

```
319 static int uhci_cleanup_queue(struct uhci_hcd *uhci, struct uhci_qh *qh,
320             struct urb *urb)
321 {
322     struct urb_priv *urbp = urb->hcpriv;
323     struct uhci_td *td;
324     int ret = 1;
325
326     /* Isochronous pipes don't use toggles and their TD link pointers
327      * get adjusted during uhci_urb_dequeue().  But since their queues
328      * cannot truly be stopped, we have to watch out for dequeues
329      * occurring after the nominal unlink frame. */
330     if (qh->type == USB_ENDPOINT_XFER_ISOC) {
331         ret = (uhci->frame_number + uhci->is_stopped !=
332                         qh->unlink_frame);
333     goto done;
334     }
335
336     /* If the URB isn't first on its queue, adjust the link pointer
337      * of the last TD in the previous URB.  The toggle doesn't need
338      * to be saved since this URB can't be executing yet. */
339     if (qh->queue.next != &urbp->node) {
340         struct urb_priv *purbp;
341         struct uhci_td *ptd;
342
343         purbp = list_entry(urbp->node.prev, struct urb_priv, node);
344         WARN_ON(list_empty(&purbp->td_list));
345         ptd = list_entry(purbp->td_list.prev, struct uhci_td,
346                         list);
347         td = list_entry(urbp->td_list.prev, struct uhci_td,
348                         list);
349         ptd->link = td->link;
350         goto done;
351     }
352
353     /* If the QH element pointer is UHCI_PTR_TERM then then currently
354      * executing URB has already been unlinked, so this one isn't it. */
355     if (qh_element(qh) == UHCI_PTR_TERM)
356         goto done;
357     qh->element = UHCI_PTR_TERM;
358
359     /* Control pipes don't have to worry about toggles */
360     if (qh->type == USB_ENDPOINT_XFER_CONTROL)
361         goto done;
362
363     /* Save the next toggle value */
364     WARN_ON(list_empty(&urbp->td_list));
365     td = list_entry(urbp->td_list.next, struct uhci_td, list);
366     qh->needs_fixup = 1;
367     qh->initial_toggle = uhci_toggle(td_token(td));
368
369 done:
370     return ret;
371 }
```

首先注释里说得也很清楚。

330 行，对于 ISO 类型，它并不使用 toggle bits，所以这里就是判断是否彻底“脱”了，是就返回 1。

339 行，如果当前讨论的这个 urb 不是 qh->queue 队列里的第一个 urb，那么就进入 if 里面的语句，purbp 将是 urbp 的前一个节点，即前一个 urbp，ptd 则是 purbp 的 TD 队列中最后一个 TD，而 td 又是 urbp 的 TD 队列中最后一个 TD。让 ptd 的 link 指向 td 的 link，即让前一个 urbp 的最后一个 TD 指向原来又本 urbp 的最后一个 TD 所指向的位置。有了接班人之后，当前这个 urb 或者说这个 urbp 就可以淡出“历史舞台”了。

357 行，让 qh->element 等于 UHCI_PTR_TERM，等于宣布本 QH 正式“退休”。

365，366，367 行的目的也很明确，保存好下一个 TD 的 toggle 位，以待时机进行修复。至于如何修复，在讲 uhci_giveback_urb 和 uhci_scan_qh 中都已经看到了，会通过判断 needs_fixup 来执行相应的代码。此处不再赘述。

关于“脱”，就讲到这里吧。

联系方式