

Ucos 系统 usb 驱动分析

何 聪

2012-8-2

目录

一、ehci 初始化	2
1. usb_init.....	2
2. usb_lowlevel_init-1.....	2
3. ehci_hcd_init.....	3
4. lowlevel_init-2	3
二、枚举设备	10
1. usb_scan_devices.....	10
2. usb_new_device.....	11
3. usb_hub_probe	13
4. usb_hub_configure	14
5. usb_hub_port_connect_change.....	15
三、热插拔的实现	16
1. usb_hot_plug.....	17
三、参考文献	18

看龙芯 1b 数据手册,可知开发板上有一个 ehci 控制器和一个 ohci 控制器,分别兼容 ehci rc1.0 和 ohci rcv1.0 协议,本文只涉及 ehci 部分。

一、ehci 初始化

ehci 协议中,专门讲了主机控制器的初始化,具体请参考规范 63 页。Ehci 规范上描述的初始化操作如下:

- Program the CTRLDSSEGMENT register with 4-Gigabyte segment where all of the interface data
- 根据系统需要,写合适的值到 USBINTR 寄存器,来使能合适的中断
- 写 Periodic Frame List (周期帧链表) 基地址到 PERIODICLIST BASE 寄存器,如果没有周期传输的条目,那么设置该寄存器的 T 标志位为 1
- 初始化 USBCMD 寄存器,设置希望的中断入口,帧列表大小,开关主机控制器等等
- 设置 CONFIGFLAG 寄存器为 1,路由所有的端口到 ehci 控制器

详细情况,请听我后面慢慢道来。

1. usb_init

看 ucos 的 usb 源码,入口点在 usb.c 的 usb_init 函数:

```
int result;
running = 0;
dev_index = 0;
asynch_allowed = 1;
usb_hub_reset();
/* init low_level USB */
printf("\nUSB:  ");
result = usb_lowlevel_init();
/* if lowlevel init is OK, scan the bus for devices
 * i.e. search HUBs and configure them */
if (result == 0) {
    printf("scanning bus for devices... ");
    running = 1;
    usb_scan_devices(hc_private);
    usb_started = 1;
    return 0;
} else {
    printf("Error, couldn't init Lowlevel part\n");
    usb_started = 0;
    return -1;
}
```

主要调用了 usb_lowlevel_init 来初始化 ehci, usb_scan_devices 来枚举新设备。

2. usb_lowlevel_init-1

- 先来看 usb_lowlevel_init (以下并不是所有代码,请参考源码)

```
if (ehci_hcd_init() != 0)
    return -1;
```

```
if (ehci_halt() != 0)
    return -1;
/* EHCI spec section 4.1 */
if (ehci_reset() != 0)
    return -1;
```

3. ehci_hcd_init

➤ 进入 ehci_hcd_init

```
*(volatile unsigned int *)0xbfd00424 &= ~(0x80000000);
*(volatile unsigned int *)0xbfd00424;
```

```
delay_ms(1);
```

```
/*ls1b usb reset stop*/
*(volatile unsigned int *)0xbfd00424 |= 0x80000000;
```

```
hccr = (struct ehci_hccr *)((uint32_t)SYS_FSL_USB_ADDR); //lxy
hcor = (struct ehci_hcor *)((uint32_t) hccr + HC_LENGTH(ehci_readl(&hccr->cr_capbase)));
```

```
*(volatile unsigned int *)0xbfd00424 &= ~(0x80000000);
*(volatile unsigned int *)0xbfd00424;
```

这个函数主要做了两件事：

- 置位 usb reset 位（1b 数据手册 203 页，MUX 寄存器的 31 位）
- 初始化 hccr, hcor 两个结构体指针，这两个结构体对应了 ehci 控制器三大寄存器空间中的两个：Host Controller Capability Registers 和 Host Controller Operational Registers（具体请参考 ehci rcv1.0）。具体的地址，就在 1b 的数据手册上了。

➤ ehci_halt 和 ehci_reset

从字面上看，就很明显，停止、重启。

ehci_halt:

关闭所有 ehci 中断，停止 ehci 控制器。通过写 Host Controller Operational Registers 空间中的相应寄存器来实现（详细请看 ehci 规范，第二章 register interface, 中的 USBINTR）

ehci_reset:

和 ehci_halt 类似，不赘述了。

4. lowlevel_init-2

下面逐行分析 lowlevel_init 函数剩下的代码：

```
/* Set head of reclaim list */
```

```

if ((u32)&qh_list & 0x1f)
    printf ("qh_list is not aligned !\n");//lxy

qh_p = CACHED_TO_UNCACHED(&qh_list);
memset(&qh_list, 0, sizeof(qh_list));
// ehci_flush_dcache(&qh_list); //lxy
flush_invalidate(&qh_list, sizeof(qh_list), 1); //lxy

qh_p->qh_link = CACHED_TO_PHYS(cpu_to_hc32((uint32_t)&qh_list | QH_LINK_TYPE_QH));
qh_p->qh_endpt1 = cpu_to_hc32((1 << 15) | (USB_SPEED_HIGH << 12));
qh_p->qh_curtd = cpu_to_hc32(QT_NEXT_TERMINATE);
qh_p->qh_overlay.qt_next = cpu_to_hc32(QT_NEXT_TERMINATE);
qh_p->qh_overlay.qt_altnext = cpu_to_hc32(QT_NEXT_TERMINATE);
qh_p->qh_overlay.qt_token = cpu_to_hc32(0x40);

/* Set async. queue head pointer. */
ehci_writel(&hcor->or_asynclistaddr, ((uint32_t)&qh_list & 0xfffffff));

```

上面几行初始化了一个 struct QH 对象，并且把该对象地址写入 hcor->or_asynclistaddr。要讲清楚为什么要这些操作，得费一番功夫了。

先来一段规范中的话（P₇₁）：

The host controller executes transactions for devices using a simple, shared-memory schedule. The schedule is comprised of a few data structures, organized into two distinct lists. The data structures are designed to provide the maximum flexibility required by USB, minimize memory traffic and hardware / software complexity.

大概意思就是：主机控制器用一个简单的、共享内存得出调度来执行设备事务。调度由含有很少数据的结构体组成的两个列表组成。这些数据结构使 usb 设计达到最大的灵活性，最小内存的传输，以及减少硬件和软件的复杂性。系统软件维持这两个调度传输，也就是我们驱动做的事情，一个是周期调度，另一个是异步调度。

USB 协议中把 USB 的传输类型分为控制传输，批量传输，中断传输，等时传输。这几种传输类型的定义其实是逻辑上的。我们知道，USB 的物理数据通道就一条（D+/D-），怎样才能达到 USB 协议中这几种传输类型的要求呢，这就要看主控制器是如何调度的了。

在 EHCI 中，把等时传输和中断传输都用周期调度来控制。请看下图：

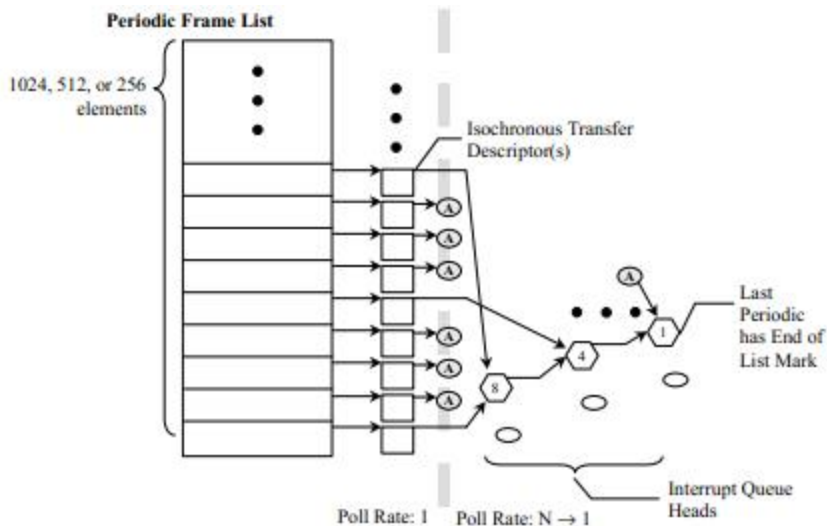


Figure 4-8. Example Periodic Schedule

所谓的分时调度，就是把每秒的时间分为若干片（一般是 1024/256 等），每一个时间片（Frame）处理一组（一般是 ISO 数据）数据。CPU 会把 ISO 数据和 INT 数据建立一张表放在内核中，而 ECHI 的寄存器 FRINDEX 则会跟踪这个表，每一个时间片加一，FRINDEX 所指之处，控制器就会把这处指针所指向的数据结构中的数据送到总线上去。整个过程看起来有点像是 CPU 的调度。有了时间片的调度，其实控制器就可以完成所有的功能，但为了方便用户的使用，控制器还提拱了另一种调度来处理实时性要求不是很强的数据。

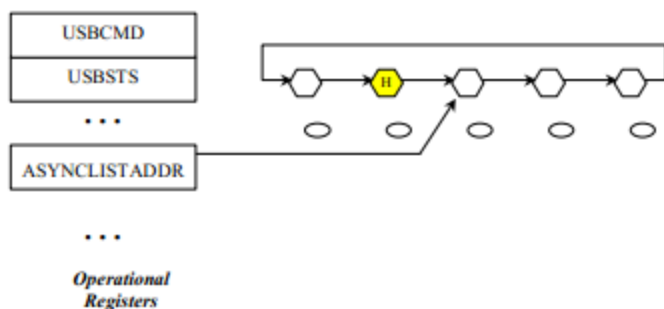


Figure 4-4. General Format of Asynchronous Schedule List

异步调度负责所有的控制传输和块传输，使用 queue head 数据结构来进行传输管理，驱动建立 queue head 链表，把基地址写入 ASYNCLISTADDR 寄存器，ehci 控制器能够读这个寄存器，并执行相应的事务。queue head 结构内容如下图：

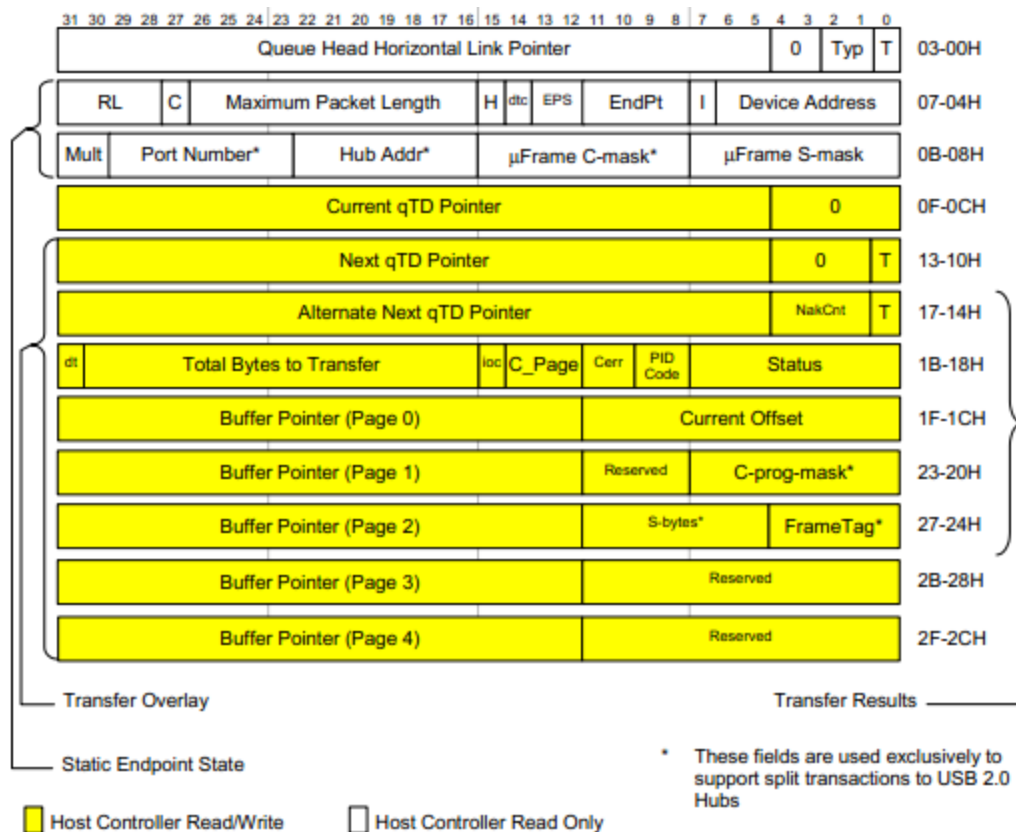


Figure 3-7. Queue Head Structure Layout

详细每个字段的意思，请参考 ehci 规范 P₅₆。

现在上面一段代码就很清楚了，初始化 queue head（就是代码的 struct QH），然后把这个链表的基地址写入 ASYNCLISTADDR 寄存器。继续下一段代码。

```
reg = ehci_readl(&hccr->cr_hcsparams);
descriptor.hub.bNbrPorts = HCS_N_PORTS(reg);
// printf("Register %x NbrPorts %d\n", reg, descriptor.hub.bNbrPorts);
/* Port Indicators */
if (HCS_INDICATOR(reg))
    descriptor.hub.wHubCharacteristics |= 0x80;
/* Port Power Control */
if (HCS_PPC(reg))
    descriptor.hub.wHubCharacteristics |= 0x01;
```

老规矩，看规范，找到相应的寄存器 hcsparams

16	Port Indicators (P_INDICATOR). This bit indicates whether the ports support port indicator control. When this bit is a one, the port status and control registers include a read/writeable field for controlling the state of the port indicator. See Section 2.3.9 for definition of the port indicator control field.
----	--

7	Port Routing Rules. This field indicates the method used by this implementation for how all ports are mapped to companion controllers. The value of this field has the following interpretation: <table> <tr> <th>Value</th><th>Meaning</th></tr> <tr> <td>0</td><td>The first <i>N_PCC</i> ports are routed to the lowest numbered function companion host controller, the next <i>N_PCC</i> port are routed to the next lowest function companion controller, and so on.</td></tr> <tr> <td>1</td><td>The port routing is explicitly enumerated by the first <i>N_PORTS</i> elements of the <i>HCSP-PORTROUTE</i> array.</td></tr> </table>	Value	Meaning	0	The first <i>N_PCC</i> ports are routed to the lowest numbered function companion host controller, the next <i>N_PCC</i> port are routed to the next lowest function companion controller, and so on.	1	The port routing is explicitly enumerated by the first <i>N_PORTS</i> elements of the <i>HCSP-PORTROUTE</i> array.
Value	Meaning						
0	The first <i>N_PCC</i> ports are routed to the lowest numbered function companion host controller, the next <i>N_PCC</i> port are routed to the next lowest function companion controller, and so on.						
1	The port routing is explicitly enumerated by the first <i>N_PORTS</i> elements of the <i>HCSP-PORTROUTE</i> array.						
6:5	Reserved. These bits are reserved and should be set to zero.						
4	Port Power Control (PPC). This field indicates whether the host controller implementation includes port power control. A one in this bit indicates the ports have port power switches. A zero in this bit indicates the port do not have port power switches. The value of this field affects the functionality of the <i>Port Power</i> field in each port status and control register (see Section 2.3.8).						
3:0	N_PORTS. This field specifies the number of physical downstream ports implemented on this host controller. The value of this field determines how many port registers are addressable in the Operational Register Space (see Table 2-8). Valid values are in the range of 1H to FH. A zero in this field is undefined.						

读 hcspparams, 然后把 N_PORTS, Port_Indicators, Port Power Control 保存, 供后面的代码使用。
具体各字段意思, 规范已经写的很清楚, 不罗嗦。

```
/* Start the host controller. */
cmd = ehci_readl(&hcor->or_usbcmd);
/*
 * Philips, Intel, and maybe others need CMD_RUN before the
 * root hub will detect new devices (why?); NEC doesn't
 */
cmd &= ~(CMD_LRESET|CMD_IAAD|CMD_PSE|CMD_ASE|CMD_RESET);
cmd |= CMD_RUN;
ehci_writel(&hcor->or_usbcmd, cmd);
```

很容易知道, 这段代码功能: 读 usbcmd 寄存器, 然后拿一堆值跟它与, 最后再写回去。
我们具体分析下 usbcmd 寄存器对应的位。先上图:

Bit	Description
7	Light Host Controller Reset (OPTIONAL) — R/W. This control bit is not required. If implemented, it allows the driver to reset the EHCI controller without affecting the state of the ports or the relationship to the companion host controllers. For example, the PORSTC registers should not be reset to their default values and the CF bit setting should not go to zero (retaining port ownership relationships). A host software read of this bit as zero indicates the Light Host Controller Reset has completed and it is safe for host software to re-initialize the host controller. A host software read of this bit as a one indicates the Light Host Controller Reset has not yet completed. If not implemented a read of this field will always return a zero.
6	Interrupt on Async Advance Doorbell — R/W. This bit is used as a doorbell by software to tell the host controller to issue an interrupt the next time it advances asynchronous schedule. Software must write a 1 to this bit to <i>ring</i> the doorbell. When the host controller has evicted all appropriate cached schedule state, it sets the <i>Interrupt on Async Advance</i> status bit in the USBSTS register. If the <i>Interrupt on Async Advance Enable</i> bit in the USBINTR register is a one then the host controller will assert an interrupt at the next interrupt threshold. See Section 4.8.2 for operational details. The host controller sets this bit to a zero after it has set the <i>Interrupt on Async Advance</i> status bit in the USBSTS register to a one. Software should not write a one to this bit when the asynchronous schedule is disabled. Doing so will yield undefined results.
5	Asynchronous Schedule Enable — R/W. Default 0b. This bit controls whether the host controller skips processing the Asynchronous Schedule. Values mean: 0b Do not process the Asynchronous Schedule 1b Use the ASYNCLISTADDR register to access the Asynchronous Schedule.
4	Periodic Schedule Enable — R/W. Default 0b. This bit controls whether the host controller skips processing the Periodic Schedule. Values mean: 0b Do not process the Periodic Schedule 1b Use the PERIODICLISTBASE register to access the Periodic Schedule.
3:2	Frame List Size — (R/W or RO). Default 00b. This field is R/W only if <i>Programmable Frame List Flag</i> in the HCCPARAMS registers is set to a one. This field specifies the size of the frame list. The size the frame list controls which bits in the Frame Index Register should be used for the Frame List Current index. Values mean: 00b 1024 elements (4096 bytes) Default value 01b 512 elements (2048 bytes) 10b 256 elements (1024 bytes) – for resource-constrained environments 11b Reserved

Bit	Description
1	Host Controller Reset (HCRESET) —R/W. This control bit is used by software to reset the host controller. The effects of this on Root Hub registers are similar to a Chip Hardware Reset. When software writes a one to this bit, the Host Controller resets its internal pipelines, timers, counters, state machines, etc. to their initial value. Any transaction currently in progress on USB is immediately terminated. A USB reset is not driven on downstream ports. PCI Configuration registers are not affected by this reset. All operational registers, including port registers and port state machines are set to their initial values. Port ownership reverts to the companion host controller(s), with the side effects described in Section 4.2. Software must reinitialize the host controller as described in Section 4.1 in order to return the host controller to an operational state. This bit is set to zero by the Host Controller when the reset process is complete. Software cannot terminate the reset process early by writing a zero to this register. Software should not set this bit to a one when the <i>HCHalted</i> bit in the USBSTS register is a zero. Attempting to reset an actively running host controller will result in undefined behavior.
0	Run/Stop (RS) —R/W. Default 0b. 1=Run. 0=Stop. When set to a 1, the Host Controller proceeds with execution of the schedule. The Host Controller continues execution as long as this bit is set to a 1. When this bit is set to 0, the Host Controller completes the current and any actively pipelined transactions on the USB and then halts. The Host Controller must halt within 16 micro-frames after software clears the Run bit. The HC Halted bit in the status register indicates when the Host Controller has finished its pending pipelined transactions and has entered the stopped state. Software must not write a one to this field unless the host controller is in the Halted state (i.e. <i>HCHalted</i> in the USBSTS register is a one). Doing so will yield undefined results.

用到的宏

```
#define CMD_ASE      (1 << 5)      /* async schedule enable */
#define CMD_LRESET   (1 << 7)      /* partial reset */
#define CMD_IAAD     (1 << 6)      /* "doorbell" interrupt */
#define CMD_PSE      (1 << 4)      /* periodic schedule enable */
#define CMD_RESET     (1 << 1)      /* reset HC not bus */
#define CMD_RUN       (1 << 0)      /* start/stop HC */
```

简单讲下这些字段，详细自己看规范。ASE 和 PSE 就是两个调度的使能开关，你在看 ehci 规范第四章，async schedule 和 periodic schedule 小节时，里面自然会提到让你记得这两个开关位。在我们的 usb 驱动中，中断、等时传输没用到，所以 disable ASE 很明显，至于现在 disable PSE，我们前面初始化的 QH 还只是个空的链表，只有一个头，没有对应的链表项，所以先关掉，后面在传输的时候还会打开的。最后 RUN 位置为 1，这样 ehci 主机控制器启动起来。

```
/* take control over the ports */
cmd = ehci_readl(&hcor->or_configflag);
cmd |= FLAG_CF;
ehci_writel(&hcor->or_configflag, cmd);
```

Bit	Description
31:1	Reserved. These bits are reserved and should be set to zero.
0	Configure Flag (CF) —R/W. Default 0b. Host software sets this bit as the last action in its process of configuring the Host Controller (see Section 4.1). This bit controls the default port-routing control logic. Bit values and side-effects are listed below. See Section 4.2 For operational details. 0b Port routing control logic default-routes each port to an implementation dependent classic host controller. 1b Port routing control logic default-routes all ports to this host controller.

这段代码涉及端口路由的问题，configureflag 寄存器来控制，

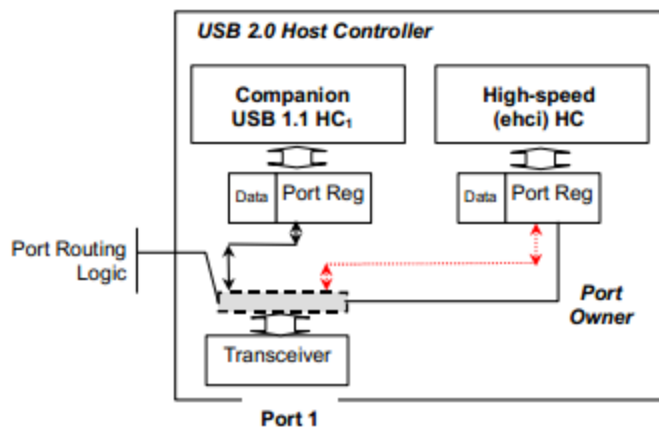


Figure 4-1. Example USB 2.0 Host Controller Port Routing Block Diagram

每次控制器上电和重启后，默认都是路由所有端口到副控制器（Companion HC），1b 上有个副控制器 ohci，这样端口只能工作在全速或低速状态。写 1 到该寄存器，就让所有端口都路由到 ehci 控制器，只有 ehci 自动释放端口，才能路由到 ohci。我们的 ohci 还没驱动，所以果断置 1，端口全路由到 ehci 来控制。端口路由就是指端口变化时，报告给谁，这里端口路由讲的比较简单，详细请自己去看规范 4.1。

```
/* unblock posted write */
cmd = ehci_readl(&hcor->or_usbcmd);
/* Turn On interrupt */
/* ehci_writel(&hcor->or_usbintr, INTR_MASK); */
```

```
wait_ms(5);
reg = HC_VERSION(ehci_readl(&hccr->cr_capbase));
printf("USB EHCI %x.%02x\n", reg >> 8, reg & 0xff);
```

读取当前 ehci 支持的 ehci 规范版本

```
rootdev = 0;
return 0;
```

到此为止，ehci 主机控制器初始化完成。

二、枚举设备

1. usb_scan_devices

枚举从 usb_scan_devices 函数开始：

```
void usb_scan_devices(void * hc_private)
{
    int i;
    struct usb_device *dev;
    /* first make all devices unknown */
```

```
for (i = 0; i < USB_MAX_DEVICE; i++) {
    memset(&usb_dev[i], 0, sizeof(struct usb_device));
    usb_dev[i].devnum = -1;
}
```

在 `usb.c` 开始就定义了 `usb_device` 结构体:

```
static struct usb_device usb_dev[USB_MAX_DEVICE];
```

`usb.h` 中有:

```
#define USB_MAX_DEVICE 32
```

很明显, 我们知道这个驱动, 最多支持 32 个 `usb` 设备

```
dev_index = 0;
```

这是针对前面的定义的结构体数组的索引, 来指示当前可以分配的数组编号, 在后面我们会多次操作这个变量。可能有人会提出疑问, 为什么不像 `linux` 中那样, 采用动态分配结构体空间。我的理解是:

- 1、动态分配, 需要在适当的时机进行内存释放, 不然会造成内存泄露
- 2、更重要的是, `ucos` 是以实时性见长的操作系统, 采用动态分配, 会增加一定时间, 并且造成时间的不可确定行, `malloc` (我们自己在 `ucos` 中实现的) 的一大块内存也是一个全局的数组, 可以去看 `malloc` 的实现
- 3、至于说内存的浪费, 因为 `ucos` 代码较少, 不像 `linux` 那么庞大, 我们可以在系统中根据需要来设置, 工作量也不会太大, 另外存在一些浪费也是在所难免, 时间和空间两者不可兼得啊。

```
/* device 0 is always present (root hub, so let it analyze) */
```

```
// dev = usb_alloc_new_device();
```

```
dev=usb_alloc_new_device(hc_private);
```

分配一个 `usb` 设备空间, 具体分配函数, 我们到后面再讲解。

```
if (usb_new_device(dev))
```

```
    printf("No USB Device found\n");
```

```
else
```

```
    printf("%d USB Device(s) found\n", dev_index);
```

2. `usb_new_device`

好了, 终于到了我们的主角上场了: `usb_new_device`, 该函数比较多, 我们只截取部分重要的来进行讲解, 具体请参照源码。

还是先来看 `usb2.0` 的规范, 打开对驱动开发最重要的一章: 第九章、`USB Device Framework`,

再看 9.1.2 `Bus Enumeration`, 没错就是传说中的总线枚举。到底什么是枚举, 规范上说了: 当一个 `usb` 设备从 `usb` 上连接或者移除时, 用来识别和管理必要的设备状态改变的过程就称作枚举。简单来说, `USB` 的枚举, 对应的就是 `USB` 的 `Host` 和 `Device` 之间的对话, 即 `Host` 根据 `Device` 所报告上来的参数, 得知 `USB` 的 `device` 是啥类型的, 具有啥功能, 然后初始化相关参数。接下来, 就 `USB Device` 就可以正常工作了。所以, 可以简单的理解为, `USB` 枚举, 就是 `USB` 设备的初始化。

当一个 `usb` 设备连接时, 大概执行下面的一些操作:

1、系统拥有了新设备。用户将设备连接到 `usb` 端口, 或者系统在连接有设备的情况下开机。端口可能处于主机内的根 `hub` 上, 或者处于连接主机和下行设备的其他 `hub` 上。`hub` 提供端口电源, 设备处于开机状态 (`Powered state`)。设备可从总线获取 `100mA` 的电流。

2、主机控制器获悉新设备。主机询问 hub 来获知基本的改变。

3、现在主机控制器已经知道连接了新设备的端口。然后，主机控制器发送端口使能和端口复位命令给该端口。

4、读设备描述符，以了解缺省管道的最大信息包尺寸（maximum packet），通过 usb 默认管道。（关于端口、管道、描述符、分配设备地址等基本概念不了解的，请先查看<<usb 开发大全（第四版）>>）

5、复位完后，主机控制器将通过发送 Set Address 请求，为设备指定唯一的地址。

6、请求设备的配置信息，基于配置信息，决定设备如何被使用。

详细的请参考 usb 规范，或者<<usb 开发大全（第四版）>>。

```
err = usb_get_descriptor(dev, USB_DT_DEVICE, 0, desc, 64);
if (err < 0) {
    USB_PRINTF("usb_new_device: usb_get_descriptor() failed\n");
    return 1;
}
```

这段代码就是获取缺省管道最大信息包尺寸，也就是上面第 4 点的步骤

```
/* find the port number we're at */
if (parent) {
    int j;
    for (j = 0; j < parent->maxchild; j++) {
        if (parent->children[j] == dev) {
            port = j;
            break;
        }
    }
    if (port < 0) {
        printf("usb_new_device:cannot locate device's port.\n");
        return 1;
    }
    /* reset the port for the second time */
    err = hub_port_reset(dev->parent, port, &portstatus);
    if (err < 0) {
        printf("\n    Couldn't reset port %i\n", port);
        return 1;
    }
}
```

这段代码现在是没有执行的，因为 parent 是 NULL，这个在 usb_alloc_new_device 函数中式被初始化位 NULL 的。先不管它。

```
err = usb_set_address(dev); /* set address */
```

设置地址，步骤 5。

```
err = usb_get_descriptor(dev, USB_DT_DEVICE, 0,
    &dev->descriptor, sizeof(dev->descriptor));
```

获取设备描述符，设备描述符含有端点 0 的最大信息包尺寸、设备支持的配置数量，以及其他的设备基本信息。

```
/* only support for one config for now */
```

```
usb_get_configuration_no(dev, &tmpbuf[0], 0);
```

获取 usb 的配置描述符，在获取配置描述符时，会同时获取接口和端点描述符。

```
usb_parse_config(dev, &tmpbuf[0], 0);
```

解析这些描述符中有用的信息，配置描述符设备的能力和特征；接口描述符提供了关于设备所实现的功能和特性的信息，包含有类、子类和协议的信息，以及接口所使用的端点数目；端点描述符中指明了端点的数目和方向等等。提取出这些有用的信息，以供后面使用。

3. usb_hub_probe

```
usb_hub_probe(dev, 0);
```

又是一个重要的函数，传进两个参数，一个是 usb 设备对象，一个是 0，代表接口 0，我们默认使用接口 0。

```
struct usb_interface *iface;
```

```
struct usb_endpoint_descriptor *ep;
```

```
int ret;
```

接口和端点描述符，请参考规范 9.6 或者 USB 开发大全，把它们弄清楚。

```
iface = &dev->config.if_desc[ifnum];
```

```
/* Is it a hub? */
```

```
if (iface->desc.bInterfaceClass != USB_CLASS_HUB)
```

```
    return 0;
```

```
/* Some hubs have a subclass of 1, which AFAICT according to the */
```

```
/* specs is not defined, but it works */
```

```
if ((iface->desc.bInterfaceSubClass != 0) &&
```

```
    (iface->desc.bInterfaceSubClass != 1))
```

```
    return 0;
```

```
/* Multiple endpoints? What kind of mutant ninja-hub is this? */
```

判断是不是 hub，因为我们当前检测的就是主机控制器里带的根 hub，而不是插入 usb 的其他设备。

```
if (iface->desc.bNumEndpoints != 1)
```

```
    return 0;
```

看规范就知道，bNumEndpoints 字段是指除端点 0 外所支持的端点数目，对于 hub，只有两个端点，除了端点 0 外，还有个中断性的端口状态改变端点。

```
ep = &iface->ep_desc[0];
```

```
/* Output endpoint? Curiousier and curiousier.. */
```

```
if (!(ep->bEndpointAddress & USB_DIR_IN))
```

```
    return 0;
```

端点描述符中字段 bEndpointAddress 指明了端点数目和方向，因为 hub 的端口改变端点是 IN 方向的，所以如果不是 IN 方向，就不是 hub。

```
/* If it's not an interrupt endpoint, we'd better punt! */
```

```
if ((ep->bmAttributes & 3) != 3)
```

```
    return 0;
```

bmAttributes 指明了端点的类型，看 hub 是不是中断类型。

```
/* We found a hub */
```

```
USB_HUB_PRINTF("USB hub found\n");
```

```
ret = usb_hub_configure(dev);
```

条件都满足，说明该设备是 hub，告诉世界我们发现了一个 hub，然后进入 usb_hub_configure.

4. usb_hub_configure

```
/* "allocate" Hub device */
```

```
hub = usb_hub_allocate();
```

分配 hub 空间，和 usb_devices 分配原理类似，先声明一个全局的结构体数组，然后用一个全局变量来索引。

```
if (usb_get_hub_descriptor(dev, buffer, 4) < 0) {
```

```
    USB_HUB_PRINTF("usb_hub_configure: failed to get hub " \
```

```
                  "descriptor, giving up %IX\n", dev->status);
```

```
    return -1;
```

```
}
```

```
descriptor = (struct usb_hub_descriptor *)buffer;
```

```
/* silence compiler warning if USB_BUFSIZ is > 256 [= sizeof(char)] */
```

```
i = descriptor->bLength;
```

```
if (i > USB_BUFSIZ) {
```

```
    USB_HUB_PRINTF("usb_hub_configure: failed to get hub " \
```

```
                  "descriptor - too long: %d\n",
```

```
                  descriptor->bLength);
```

```
    return -1;
```

```
}
```

先获取 hub 描述符的长度，请看 hub 描述符字段 bLength (11.23.2.1)。

```
if (usb_get_hub_descriptor(dev, buffer, descriptor->bLength) < 0) {
```

```
    USB_HUB_PRINTF("usb_hub_configure: failed to get hub " \
```

```
                  "descriptor 2nd giving up %IX\n", dev->status);
```

```
    return -1;
```

```
}
```

```
memcpy((unsigned char *)&hub->desc, buffer, descriptor->bLength);
```

```
/* adjust 16bit values */
```

真正请求 hub 描述符，并把它复制到函数初试分配的 hub 结构体中。

```
hub->desc.wHubCharacteristics =
```

```
    le16_to_cpu(descriptor->wHubCharacteristics);
```

```
/* set the bitmap */
```

```
bitmap = (unsigned char *)&hub->desc.DeviceRemovable[0];
```

```
/* devices not removable by default */
```

```
memset(bitmap, 0xff, (USB_MAXCHILDREN+1+7)/8);
```

hub 描述符的 DeviceRemovable 字段表明 hub 上的端口是否可移除，默认都设为不可移除，现在我们设为可移除，0 代表可移除，1 代表不可移除，这里给了 0xff，理应说是变成不可移除，但是 DeviceRemovable 是 unsigned char，所以 0xff 等于 0。？

```
bitmap = (unsigned char *)&hub->desc.PortPowerCtrlMask[0];
```

```
memset(bitmap, 0xff, (USB_MAXCHILDREN+1+7)/8); /* PowerMask = 1B */
```

和上面一样，进行设置。

```
for (i = 0; i < dev->maxchild; i++) {  
    struct usb_port_status portsts;  
    unsigned short portstatus, portchange;  
  
    if (usb_get_port_status(dev, i + 1, &portsts) < 0) {  
        USB_HUB_PRINTF("get_port_status failed\n");  
        continue;  
    }  
  
    portstatus = le16_to_cpu(portsts.wPortStatus);  
    portchange = le16_to_cpu(portsts.wPortChange);  
    USB_HUB_PRINTF("Port %d Status %X Change %X\n",  
        i + 1, portstatus, portchange);  
  
    if (portchange & USB_PORT_STAT_C_CONNECTION) {  
        USB_HUB_PRINTF("port %d connection change\n", i + 1);  
        usb_hub_port_connect_change(dev, i);  
    }  
    .....  
}
```

这段代码后面都类似，没完全贴出来了。

获取 port 状态，规范 11.24.2.7。包含 Port Status Bits，Port Status Change Bits，对应了代码如下结构体：

```
struct usb_port_status {  
    unsigned short wPortStatus;  
    unsigned short wPortChange;  
} __attribute__((packed));
```

USB_PORT_STAT_C_CONNECTION 用来检查 Connect Status Change 位，有插拔发生时，状态就会改变，后面的一些 if 原理一样，可具体看每个位代表的含义，就能很清楚的理解。

如果有状态改变，进入 usb_hub_port_connect_change。

5. usb_hub_port_connect_change

```
/* Clear the connection change status */  
usb_clear_port_feature(dev, port + 1, USB_PORT_FEAT_C_CONNECTION);  
Port status change bits can be cleared with a ClearPortFeature() request or by a hub  
reset.port status change bits 能被清除，用 ClearPortFeature().  
/* Disconnect any existing devices under this port */  
if (((!(portstatus & USB_PORT_STAT_CONNECTION)) &&  
    (!(portstatus & USB_PORT_STAT_ENABLE))) || (dev->children[port])) {  
    USB_HUB_PRINTF("usb_disconnect(&hub->children[port]);\n");  
    if (!(portstatus & USB_PORT_STAT_CONNECTION) && portchange) {  
        .....  
        if (dev_index > 2) {
```

```

        dev_index--;
        printf("\n----->usb popup success\n");
    }
    dev->children[port] = NULL;
}
/* Return now if nothing is connected */
return;
}

```

如果是移除设备，那就释放分配的数据内存，只要把 `dev_index-1` 就行。我们前面已经说过了，不清楚的也可以去看下 `usb_devices` 如何分配的，就能够明白此时的释放过程。然后，就可以直接返回。

如果有设备连接，那么接下来就是一个枚举设备的过程。前面讲解了规范中的枚举步骤，所以可以知道接下就是：

`hub_port_reset(dev, port, &portstatus)` 复位端口

`usb_new_device(usb)` 该函数，前面也分析了，做了大部分枚举的工作，当设备不是 hub 时，是不会进入 hub 相关的函数，因为它根本不可能通过 `hub_probe` 那些对 hub 进行的“认证”。

层层调用结束后，回到起始函数 `usb_scan_devices`，这个函数也同样结束了。

三、热插拔的实现

先简单介绍下 USB HUB controller，详细在 2.0 规范 11.12。

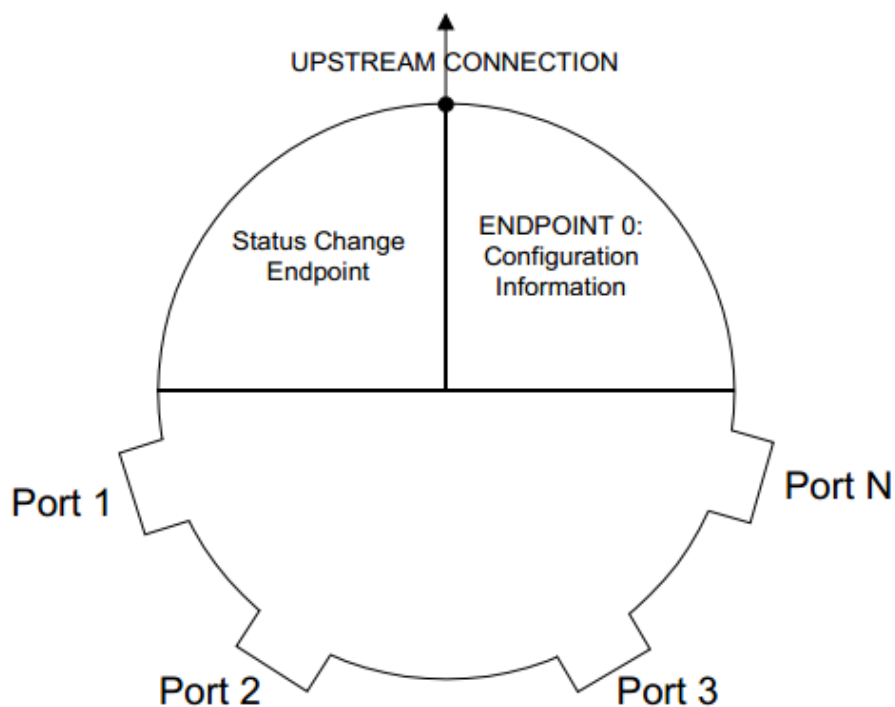


Figure 11-19. Example Hub Controller Organization

HUB 除了默认端点 0 外, 还有一个 status Change Endpoint, 这是所有的 hub 都必须有的。图中左上方是一个中断型的状态改变端点, HOST 通过以固定的时间间隔向该端点发起 IN 的中断传输查询该 HUB 自身和各下行 PORT 的状态是否改变。当没有状态改变时, 该端点返回 NAK 的握手包; 当有状态改变时返回一个反应了 HUB 自身和各下行 PORT 状态改变信息的 Bitmap。每个 PORT 映射到该 Bitmap 中的一位, 当该 PORT 的状态有改变时, 对应的位置 1, 这是由 USB HUB 硬件完成的。status Change 端点是一个中断类型的端点, 主机通过这个端点来获知端口和 hub 状态改变的信息。

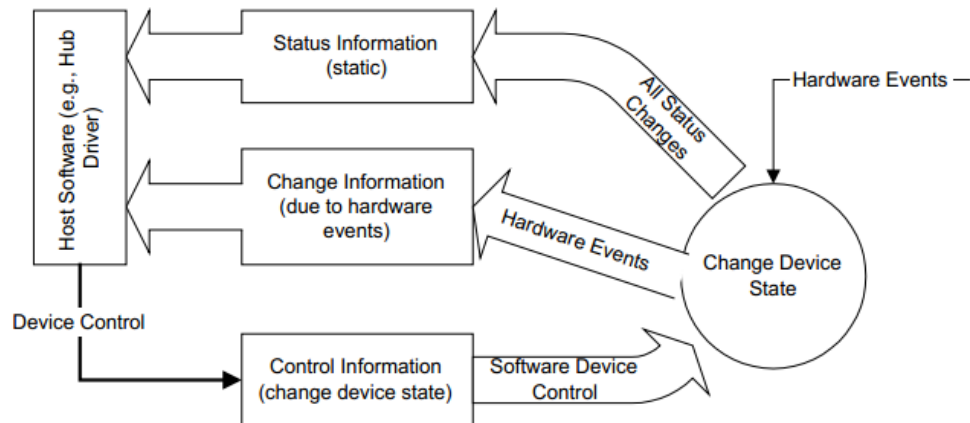
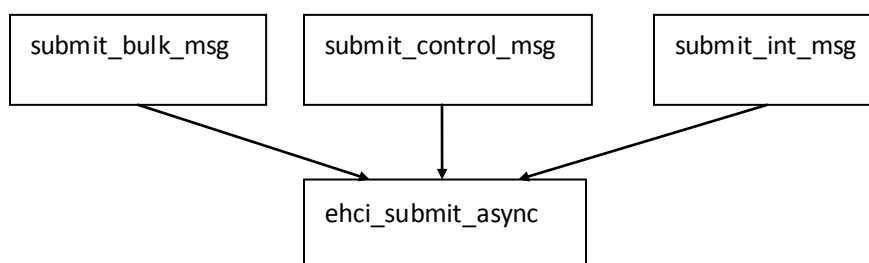


Figure 11-20. Relationship of Status, Status Change, and Control Information to Device States

本来应该采用中断传输, 中断传输可以保证最大等待时间, 但是在原有基础上添加中断传输, 需要增加比较多的代码, 我对这块也不是很熟悉, 所以采用了一种很拙劣的实现方式。跟控制传输、bulk 传输, 共用底层包发送函数。



三个传输类型都调用底层异步调度。ehci_submit_async 实现也比较简单, 没有 linux 那么复杂, 组装一个发送包, 然后就发送一个, 这就造成了该函数是不可重入的。所以必须保证在该函数发生批量传输时, 不应该被中断传输或者控制传输打断。

热插拔, 通过创建一个进程, 定期掉用 usb_hot_plug 轮询 hub 中断端口。

1. usb_hot_plug

```

if(usb_submit_int_msg(dev, pipe, &bitmap, length, ep->blInterval) != -1)
    hub_events();

```

通过中断传输查询 hub 状态改变端口, 然后调用 hub_events 去处理返回的数据 Bitmap。

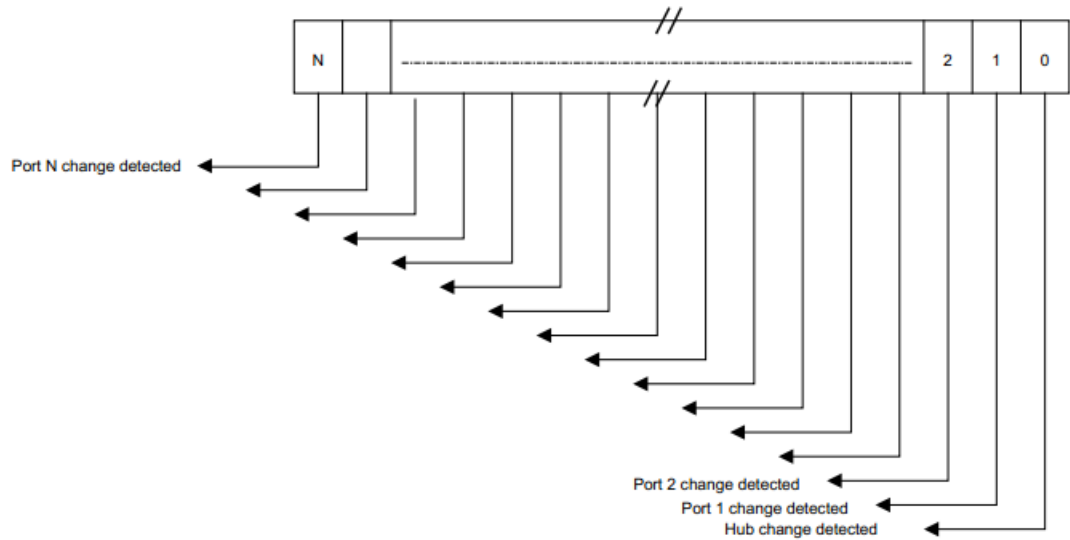


Figure 11-22. Hub and Port Status Change Bitmap

```

/* check bitmap */
for (i = 1; i <= hub->desc.bNbrPorts; i++) {
    if (!test_bit(i, bitmap)){
        USB_HUB_PRINTF ("i=%d, bitmap=%d\n", i, bitmap);
        continue;
    }
    usb_hub_port_connect_change(hdev, i-1);
} /* end for i */

```

测试所有端口对应的位,如果是 1,说明有改变,调用 `usb_hub_port_connect_change`,这个函数前面已经分析过。最终,就能发现新插入的设备。

三、参考文献

1. 《usb 开发大全》第四版
2. usb 2.0 规范
3. ehci 1.0 规范