

# 第 16 章

## USB 主机、设备与 Gadget 驱动

### 本章导读

在 Linux 系统中，提供了主机侧和设备侧视角的 USB 驱动框架，本章主要讲解从主机侧看到的 USB 主机控制器驱动和设备驱动，以及从设备侧看到的设备控制器和 Gadget 驱动。

16.1 节给出了 Linux 系统中 USB 驱动的整体视图，讲解了 Linux 中从主机侧和设备侧看到的 USB 驱动层次。

从主机侧的角度来看，需要编写的 USB 驱动程序包括主机控制器驱动和设备驱动两类，USB 主机控制器驱动程序控制插入其中的 USB 设备，而 USB 设备驱动程序控制该设备如何作为从设备与主机通信。16.2 节分析了 USB 主机控制器驱动的结构并给出 Chipidea USB 主机驱动实例，16.3 节讲解了 USB 设备驱动的结构及其设备请求块处理过程，并给出了 USB 键盘驱动实例。

从设备侧的角度来看，包含编写 USB 设备控制器（UDC）驱动和 Gadget Function 驱动两类，16.4 节对 UDC 和 Gadget 驱动进行了讲解，并给出了 Chipidea USB UDC 和 Loopback Function 作为实例。

16.5 节简单地介绍了一下 USB OTG 驱动。

16.1 节与 16.2 ~ 16.5 节是整体与部分的关系。

### 16.1 Linux USB 驱动层次

#### 16.1.1 主机侧与设备侧 USB 驱动

USB 采用树形拓扑结构，主机侧和设备侧的 USB 控制器分别称为主机控制器（Host Controller）和 USB 设备控制器（UDC），每条总线上只有一个主机控制器，负责协调主机和设备间的通信，而设备不能主动向主机发送任何消息。如图 16.1 所示，在 Linux 系统中，USB 驱动可以从两个角度去观察，一个角度是主机侧，一个角度是设备侧。

如图 16.1 的左侧所示，从主机侧去看，在 Linux 驱动中，处于 USB 驱动最底层的是 USB 主机控制器硬件，在其上运行的是 USB 主机控制器驱动，在主机控制器上的为 USB 核

心层，再上层为USB设备驱动层（插入主机上的U盘、鼠标、USB转串口等设备驱动）。因此，在主机侧的层次结构中，要实现的USB驱动包括两类：USB主机控制器驱动和USB设备驱动，前者控制插入其中的USB设备，后者控制USB设备如何与主机通信。Linux内核中的USB核心负责USB驱动管理和协议处理的主要工作。主机控制器驱动和设备驱动之间的USB核心非常重要，其功能包括：通过定义一些数据结构、宏和功能函数，向上为设备驱动提供编程接口，向下为USB主机控制器驱动提供编程接口；维护整个系统的USB设备信息；完成设备热插拔控制、总线数据传输控制等。

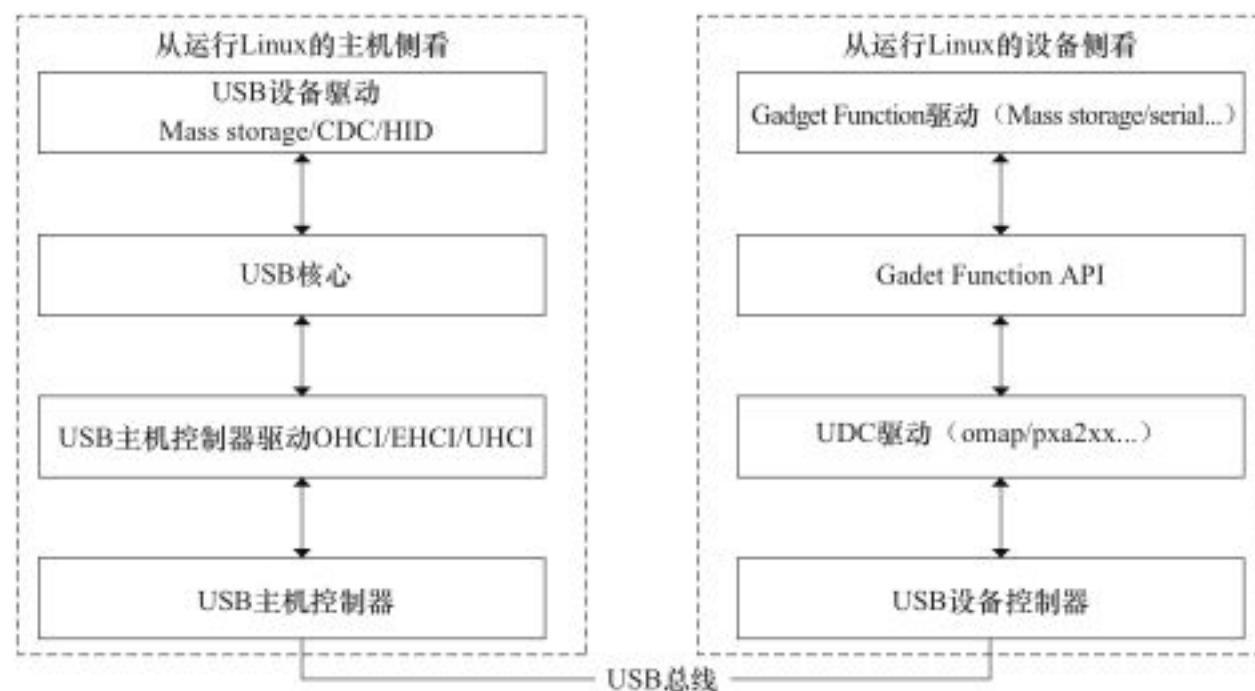


图 16.1 Linux USB 驱动总体结构

如图16.1的右侧所示，Linux内核中USB设备侧驱动程序分为3个层次：UDC驱动程序、Gadget Function API和Gadget Function驱动程序。UDC驱动程序直接访问硬件，控制USB设备和主机间的底层通信，向上层提供与硬件相关操作的回调函数。当前Gadget Function API是UDC驱动程序回调函数的简单包装。Gadget Function驱动程序具体控制USB设备功能的实现，使设备表现出“网络连接”、“打印机”或“USB Mass Storage”等特性，它使用Gadget Function API控制UDC实现上述功能。Gadget Function API把下层的UDC驱动程序和上层的Gadget Function驱动程序隔离开，使得在Linux系统中编写USB设备侧驱动程序时能够把功能的实现和底层通信分离。

### 16.1.2 设备、配置、接口、端点

在USB设备的逻辑组织中，包含设备、配置、接口和端点4个层次。

每个USB设备都提供不同级别的配置信息，可以包含一个或多个配置，不同的配置使设备表现出不同的功能组合（在探测/连接期间需从其中选定一个），配置由多个接口组成。

在 USB 协议中，接口由多个端点组成，代表一个基本的功能，是 USB 设备驱动程序控制的对象，一个功能复杂的 USB 设备可以具有多个接口。每个配置中可以有多个接口，而设备接口是端点的汇集（Collection）。例如，USB 扬声器可以包含一个音频接口以及对旋钮和按钮的接口。一个配置中的所有接口可以同时有效，并可被不同的驱动程序连接。每个接口可以有备用接口，以提供不同质量的服务参数。

端点是 USB 通信的最基本形式，每一个 USB 设备接口在主机看来就是一个端点的集合。主机只能通过端点与设备进行通信，以使用设备的功能。在 USB 系统中每一个端点都有唯一的地址，这是由设备地址和端点号给出的。每个端点都有一定的属性，其中包括传输方式、总线访问频率、带宽、端点号和数据包的最大容量等。一个 USB 端点只能在一个方向上承载数据，从主机到设备（称为输出端点）或者从设备到主机（称为输入端点），因此端点可看作是一个单向的管道。端点 0 通常为控制端点，用于设备初始化参数等。只要设备连接到 USB 上并且上电，端点 0 就可以被访问。端点 1、2 等一般用作数据端点，存放主机与设备间往来的数据。

总体而言，USB 设备非常复杂，由许多不同的逻辑单元组成，如图 16.2 所示，这些单元之间的关系如下：

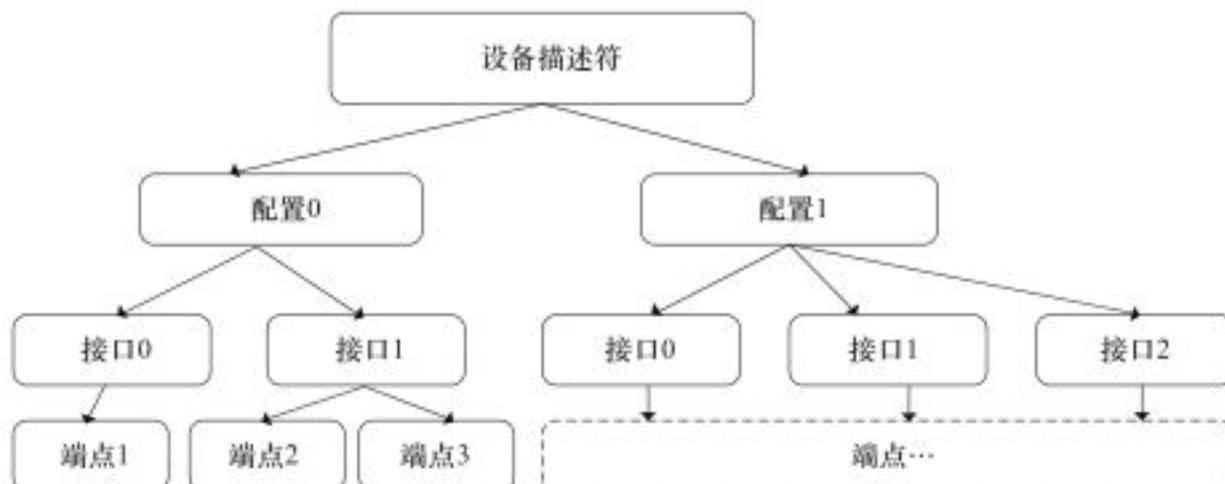


图 16.2 USB 设备、配置、接口和端点

- 设备通常有一个或多个配置；
- 配置通常有一个或多个接口；
- 接口通常有一个或多个设置；
- 接口有零个或多个端点。

这种层次化配置信息在设备中通过一组标准的描述符来描述，如下所示。

- 设备描述符：关于设备的通用信息，如供应商 ID、产品 ID 和修订 ID，支持的设备类、子类和适用的协议以及默认端点的最大包大小等。在 Linux 内核中，USB 设备用 `usb_device` 结构体来描述，USB 设备描述符定义为 `usb_device_descriptor` 结构体，位于 `include/uapi/linux/usb/ch9.h` 文件中，如代码清单 16.1 所示。

代码清单 16.1 usb\_device\_descriptor 结构体

---

```

1 struct usb_device_descriptor {
2     __u8 bLength;                      /* 描述符长度 */
3     __u8 bDescriptorType;               /* 描述符类型编号 */
4
5     __le16 bcdUSB;                    /* USB 版本号 */
6     __u8 bDeviceClass;                /* USB 分配的设备类 code */
7     __u8 bDeviceSubClass;              /* USB 分配的子类 code */
8     __u8 bDeviceProtocol;              /* USB 分配的协议 code */
9     __u8 bMaxPacketSize0;              /* endpoint0 最大包大小 */
10    __le16 idVendor;                 /* 厂商编号 */
11    __le16 idProduct;                /* 产品编号 */
12    __le16 bcdDevice;                 /* 设备出厂编号 */
13    __u8 iManufacturer;              /* 描述厂商字符串的索引 */
14    __u8 iProduct;                   /* 描述产品字符串的索引 */
15    __u8 iSerialNumber;              /* 描述设备序列号字符串的索引 */
16    __u8 bNumConfigurations;         /* 可能的配置数量 */
17 } __attribute__ ((packed));

```

---

- 配置描述符：此配置中的接口数、支持的挂起和恢复能力以及功率要求。USB 配置在内核中使用 `usb_host_config` 结构体描述，而 USB 配置描述符定义为结构体 `usb_config_descriptor`，如代码清单 16.2 所示。

代码清单 16.2 usb\_config\_descriptor 结构体

---

```

1 struct usb_config_descriptor {
3     __u8 bLength;                      /* 描述符长度 */
4     __u8 bDescriptorType;               /* 描述符类型编号 */
5
6     __le16 wTotalLength;                /* 配置所返回的所有数据的大小 */
7     __u8 bNumInterfaces;                /* 配置所支持的接口数 */
8     __u8 bConfigurationValue;           /* Set_Configuration 命令需要的参数值 */
9     __u8 iConfiguration;                /* 描述该配置的字符串的索引值 */
10    __u8 bmAttributes;                  /* 供电模式的选择 */
11    __u8 bMaxPower;                    /* 设备从总线提取的最大电流 */
12 } __attribute__ ((packed));

```

---

- 接口描述符：接口类、子类和适用的协议，接口备用配置的数目和端点数目。USB 接口在内核中使用 `usb_interface` 结构体描述，而 USB 接口描述符定义为结构体 `usb_interface_descriptor`，如代码清单 16.3 所示。

代码清单 16.3 usb\_interface\_descriptor 结构体

---

```

1 struct usb_interface_descriptor {
3     __u8 bLength;                      /* 描述符长度 */
4     __u8 bDescriptorType;               /* 描述符类型 */
5
6     __u8 bInterfaceNumber;              /* 接口的编号 */

```

---

---

```

7   __u8 bAlternateSetting;      /* 备用的接口描述符编号 */
8   __u8 bNumEndpoints;        /* 该接口使用的端点数，不包括端点0 */
9   __u8 bInterfaceClass;      /* 接口类型 */
10  __u8 bInterfaceSubClass;    /* 接口子类型 */
11  __u8 bInterfaceProtocol;   /* 接口所遵循的协议 */
12  __u8 iInterface;          /* 描述该接口的字符串索引值 */
13 } __attribute__ ((packed));

```

---

- 端点描述符：端点地址、方向和类型，支持的最大包大小，如果是中断类型的端点则还包括轮询频率。在Linux内核中，USB端点使用usb\_host\_endpoint结构体来描述，而USB端点描述符定义为usb\_endpoint\_descriptor结构体，如代码清单16.4所示。

代码清单16.4 usb\_endpoint\_descriptor结构体

---

```

1 struct usb_endpoint_descriptor {
3   __u8 bLength;                /* 描述符长度 */
4   __u8 bDescriptorType;         /* 描述符类型 */
5   __u8 bEndpointAddress;       /* 端点地址：0~3位是端点号，第7位是方向(0为输出,1为输入) */
6   __u8 bmAttributes;           /* 端点属性: bit[0:1] 的值为00表示控制, 为01表示同步,
                                /* 为02表示批量, 为03表示中断 */
7   __le16 wMaxPacketSize;        /* 本端点接收或发送的最大信息包的大小 */
8   __u8 bInterval;              /* 轮询数据传送端点的时间间隔 */
9                                /* 对于批量传递的端点以及控制传递的端点, 此域忽略 */
10                                /* 对于同步传递的端点, 此域必须为1 */
11                                /* 对于中断传递的端点, 此域值的范围为1~255 */
12   __u8 bRefresh;
13   __u8 bSynchAddress;
14 } __attribute__ ((packed));

```

---

- 字符串描述符：在其他描述符中会为某些字段提供字符串索引，它们可被用来检索描述性字符串，可以以多种语言形式提供。字符串描述符是可选的，有的设备有，有的设备没有，字符串描述符对应于usb\_string\_descriptor结构体，如代码清单16.5所示。

代码清单16.5 usb\_string\_descriptor结构体

---

```

1 struct usb_string_descriptor {
3   __u8 bLength;                /* 描述符长度 */
4   __u8 bDescriptorType;         /* 描述符类型 */
5
6   __le16 wData[1];              /* 以UTF-16LE编码 */
7 } __attribute__ ((packed));

```

---

例如，笔者在PC上插入一个SanDisk U盘后，通过lsusb命令得到与这个U盘相关的描述符，从中可以显示这个U盘包含了一个设备描述符、一个配置描述符、一个接口描述符、批量输入和批量输出两个端点描述符。呈现出来的信息内容直接对应于usb\_device\_descriptor、usb\_config\_descriptor、usb\_interface\_descriptor、usb\_endpoint\_descriptor、usb\_string\_descriptor结构体，如下所示：

```
Bus 001 Device 004: ID 0781:5151 SanDisk Corp.
Device Descriptor:
bLength          18
bDescriptorType   1
bcdUSB           2.00
bDeviceClass      0 Interface
bDeviceSubClass    0
bDeviceProtocol    0
bMaxPacketSize0   64
idVendor          0x0781 SanDisk Corp.
idProduct         0x5151
bcdDevice         0.10
iManufacturer     1 SanDisk Corporation
iProduct          2 Cruzer Micro
iSerial           3 20060877500A1BE1FDE1
bNumConfigurations 1
Configuration Descriptor:
bLength          9
bDescriptorType   2
wTotalLength      32
bNumInterfaces    1
bConfigurationValue 1
iConfiguration     0
bmAttributes       0x80
MaxPower          200mA
Interface Descriptor:
bLength          9
bDescriptorType   4
bInterfaceNumber  0
bAlternateSetting 0
bNumEndpoints     2
bInterfaceClass    8 Mass Storage
bInterfaceSubClass 6 SCSI
bInterfaceProtocol 80 Bulk (Zip)
iInterface         0
Endpoint Descriptor:
bLength          7
bDescriptorType   5
bEndpointAddress  0x81 EP 1 IN
bmAttributes       2
Transfer Type      Bulk
Synch Type        none
wMaxPacketSize    512
bInterval          0
Endpoint Descriptor:
bLength          7
bDescriptorType   5
bEndpointAddress  0x01 EP 1 OUT
bmAttributes       2
Transfer Type      Bulk
```

```

Synch Type          none
wMaxPacketSize     512
bInterval          1
Language IDs: (length=4)
0409 English (US)

```

## 16.2 USB 主机控制器驱动

### 16.2.1 USB 主机控制器驱动的整体结构

USB 主机控制器有这些规格：OHCI (Open Host Controller Interface)、UHCI (Universal Host Controller Interface)、EHCI (Enhanced Host Controller Interface) 和 xHCI (eXtensible Host Controller Interface)。OHCI 驱动程序用来为非 PC 系统上以及带有 SiS 和 ALi 芯片组的 PC 主板上的 USB 芯片提供支持。UHCI 驱动程序多用来为大多数其他 PC 主板（包括 Intel 和 Via）上的 USB 芯片提供支持。EHCI 由 USB 2.0 规范所提出，它兼容于 OHCI 和 UHCI。由于 UHCI 的硬件线路比 OHCI 简单，所以成本较低，但需要较复杂的驱动程序，CPU 负荷稍重。xHCI，即可扩展的主机控制器接口是 Intel 公司开发的一个 USB 主机控制器接口，它目前主要是面向 USB 3.0 的，同时它也支持 USB 2.0 及以下的设备。

#### 1. 主机控制器驱动

在 Linux 内核中，用 `usb_hcd` 结构体描述 USB 主机控制器驱动，它包含 USB 主机控制器的“家务”信息、硬件资源、状态描述和用于操作主机控制器的 `hc_driver` 等，其定义如代码清单 16.6 所示。

代码清单 16.6 `usb_hcd` 结构体

---

```

1 struct usb_hcd {
2     struct usb_bus           self;           /* hcd is-a bus */
3     struct kref              kref;           /* reference counter */
4
5     const char               *product_desc;    /* product/vendor string */
6     int                      speed;           /* Speed for this roothub,
7                                         * May be different from
8                                         * hcd->driver->flags & HCD_MASK
9                                         */
10    char                     irq_descr[24];    /* driver + bus # */
11
12    struct timer_list         rh_timer;        /* drives root-hub polling */
13    struct urb                *status_urb;      /* the current status urb */
14 #ifdef CONFIG_PM
15    struct work_struct       wakeup_work;     /* for remote wakeup */
16 #endif
17
18    const struct hc_driver   *driver;         /* hw-specific hooks */
19
20    struct usb_phy           *usb_phy;

```

```

21 struct phy      *phy;
22
23 unsigned long     flags;
24
25 ...
26
27 /* The HC driver's private data is stored at the end of
28 * this structure.
29 */
30 unsigned long hcd_priv[0]
31         __attribute__ ((aligned(sizeof(s64))));
```

---

usb\_hcd 结构体中第 18 行的 hc\_driver 成员非常重要，它包含具体的用于操作主机控制器的钩子函数，即“hw-specific hooks”，其定义如代码清单 16.7 所示。

代码清单 16.7 hc\_driver 结构体

```

1 struct hc_driver {
2     const char    *description;           /* "ehci-hcd" etc */
3     const char    *product_desc;          /* product/vendor string */
4     size_t        hcd_priv_size;          /* size of private data */
5
6     /* irq handler */
7     irqreturn_t   (*irq) (struct usb_hcd *hcd);
8
9     int flags;
10
11    /* called to init HCD and root hub */
12    int (*reset) (struct usb_hcd *hcd);
13    int (*start) (struct usb_hcd *hcd);
14    ...
15    /* cleanly make HCD stop writing memory and doing I/O */
16    void (*stop) (struct usb_hcd *hcd);
17
18    /* shutdown HCD */
19    void (*shutdown) (struct usb_hcd *hcd);
20
21    /* return current frame number */
22    int (*get_frame_number) (struct usb_hcd *hcd);
23
24    /* manage i/o requests, device state */
25    int (*urb_enqueue)(struct usb_hcd *hcd,
26                      struct urb *urb, gfp_t mem_flags);
27    int (*urb_dequeue)(struct usb_hcd *hcd,
28                      struct urb *urb, int status);
29    ...
30    /* Allocate endpoint resources and add them to a new schedule */
31    int (*add_endpoint)(struct usb_hcd *, struct usb_device *,
32                        struct usb_host_endpoint *);
```

```

33      /* Drop an endpoint from a new schedule */
34  int (*drop_endpoint)(struct usb_hcd *, struct usb_device *,
35                      struct usb_host_endpoint *);
36
37  int (*check_bandwidth)(struct usb_hcd *, struct usb_device *);
38  void (*reset_bandwidth)(struct usb_hcd *, struct usb_device *);
39      /* Returns the hardware-chosen device address */
40  int (*address_device)(struct usb_hcd *, struct usb_device *udev);
41 ...
42  int (*set_usb2_hw_lpm)(struct usb_hcd *, struct usb_device *, int);
43  /* USB 3.0 Link Power Management */
44      /* Returns the USB3 hub-encoded value for the U1/U2 timeout. */
45  int (*enable_usb3_lpm_timeout)(struct usb_hcd *,
46                                struct usb_device *, enum usb3_link_state state);
47  int (*disable_usb3_lpm_timeout)(struct usb_hcd *,
48                                struct usb_device *, enum usb3_link_state state);
49  int (*find_raw_port_number)(struct usb_hcd *, int);
50  /* Call for power on/off the port if necessary */
51  int (*port_power)(struct usb_hcd *hcd, int portnum, bool enable);
52 };

```

---

在 Linux 内核中，使用如下函数来创建 HCD：

```
struct usb_hcd *usb_create_hcd(const struct hc_driver *driver,
                               struct device *dev, char *bus_name);
```

如下函数被用来增加和移除 HCD：

```
int usb_add_hcd(struct usb_hcd *hcd,
                 unsigned int irqnum, unsigned long irqflags);
void usb_remove_hcd(struct usb_hcd *hcd);
```

第 25 行的 `urb_enqueue()` 函数非常关键，实际上，上层通过 `usb_submit_urb()` 提交 1 个 USB 请求后，该函数调用 `usb_hcd_submit_urb()`，并最终调用至 `usb_hcd` 的 `driver` 成员（`hc_driver` 类型）的 `urb_enqueue()` 函数。

## 2. EHCI 主机控制器驱动

EHCI HCD 驱动属于 HCD 驱动的实例，它定义了一个 `ehci_hcd` 结构体，通常作为代码清单 16.6 定义的 `usb_hcd` 结构体的私有数据（`hcd_priv`），这个结构体的定义位于 `drivers/usb/host/ehci.h` 中，如代码清单 16.8 所示。

代码清单 16.8 `ehci_hcd` 结构体

```

1 struct ehci_hcd {                           /* one per controller */
2     /* timing support */
3     enum ehci_hrtimer_event    next_hrtimer_event;
4     unsigned      enabled_hrtimer_events;
5     ktime_t       hr_timeouts[EHCI_HRTIMER_NUM_EVENTS];
6     struct hrtimer    hrtimer;

```

```

7
8     int          PSS_poll_count;
9     int          ASS_poll_count;
10    int         died_poll_count;
11    ...
12    /* general schedule support */
13    bool         scanning:l;
14    bool         need_rescan:l;
15    bool         intr_unlinking:l;
16    bool         iaa_in_progress:l;
17    bool         async_unlinking:l;
18    bool         shutdown:l;
19    struct ehci_qh      *qh_scan_next;
20
21    /* async schedule support */
22    struct ehci_qh      *async;
23    struct ehci_qh      *dummy;           /* For AMD quirk use */
24    struct list_head  async_unlink;
25    struct list_head  async_idle;
26    unsigned        async_unlink_cycle;
27    unsigned        async_count;        /* async activity count */
28
29    /* periodic schedule support */
30 #define  DEFAULT_I_TDPS      1024       /* some HCs can do less */
31    unsigned        periodic_size;
32    __hc32          *periodic;        /* hw periodic table */
33    dma_addr_t      periodic_dma;
34    struct list_head  intr_qh_list;
35    unsigned        i_thresh;        /* uframes HC might cache */
36    ...
37    /* bandwidth usage */
38 #define EHCI_BANDWIDTH_SIZE 64
39 #define EHCI_BANDWIDTH_FRAMES  (EHCI_BANDWIDTH_SIZE >> 3)
40    u8            bandwidth[EHCI_BANDWIDTH_SIZE];
41                      /* us allocated per uframe */
42    u8            tt_budget[EHCI_BANDWIDTH_SIZE];
43                      /* us budgeted per uframe */
44    struct list_head tt_list;
45
46    /* platform-specific data -- must come last */
47    unsigned long   priv[0] __aligned(sizeof(s64));
48 }

```

使用如下内联函数可实现 `usb_hcd` 和 `ehci_hcd` 的相互转换：

```

struct ehci_hcd *hcd_to_ehci (struct usb_hcd *hcd);
struct usb_hcd *ehci_to_hcd (const struct ohci_hcd *ohci);

```

从 `usb_hcd` 得到 `ehci_hcd` 只是取得“私有”数据，而从 `ehci_hcd` 得到 `usb_hcd` 则是通过 `container_of()` 从结构体成员获得结构体指针。

使用如下函数可初始化 EHCI 主机控制器：

```
static int ehci_init(struct usb_hcd *hcd);
```

如下函数分别用于开启、停止及复位 EHCI 控制器：

```
static int ehci_run (struct usb_hcd *hcd);
static void ehci_stop (struct usb_hcd *hcd);
static int ehci_reset (struct ehci_hcd *ehci);
```

上述函数在 drivers/usb/host/ehci-hcd.c 文件中被填充给了一个 hc\_driver 结构体的 generic 的实例 ehci\_hc\_driver。

```
static const struct hc_driver ehci_hc_driver = {
    ...
    .reset =          ehci_setup,
    .start =         ehci_run,
    .stop =          ehci_stop,
    .shutdown =     ehci_shutdown,
}
```

drivers/usb/host/ehci-hcd.c 实现了绝大多数 EHCI 主机驱动的工作，具体的 EHCI 实例简单地调用

```
void ehci_init_driver(struct hc_driver *drv,
                      const struct ehci_driver_overrides *over);
```

初始化 hc\_driver 即可，这个函数会被 generic 的 ehci\_hc\_driver 实例复制给每个具体底层驱动的实例，当然底层驱动可以通过第 2 个参数，即 ehci\_driver\_overrides 重写中间层的 reset()、port\_power() 这 2 个函数，另外也可以填充一些额外的私有数据，这一点从代码清单 16.9 ehci\_init\_driver() 的实现中可以看出。

代码清单 16.9 ehci\_init\_driver 的实现

---

```
1 void ehci_init_driver(struct hc_driver *drv,
2                       const struct ehci_driver_overrides *over)
3 {
4     /* Copy the generic table to drv and then apply the overrides */
5     *drv = ehci_hc_driver;
6
7     if (over) {
8         drv->hcd_priv_size += over->extra_priv_size;
9         if (over->reset)
10             drv->reset = over->reset;
11         if (over->port_power)
12             drv->port_power = over->port_power;
13     }
14 }
```

---

### 16.2.2 实例：Chipidea USB 主机驱动

Chipidea 的 USB IP 在嵌入式系统中应用比较广泛，它的驱动位于 drivers/usb/chipidea/ 目录下。

当 Chipidea USB 驱动的内核代码 drivers/usb/chipidea/core.c 中的 ci\_hdrc\_probe() 被执行后（即一个 platform\_device 与 ci\_hdrc\_driver 这个 platform\_driver 匹配上了），它会调用 drivers/usb/chipidea/host.c 中的 ci\_hdrc\_host\_init() 函数，该函数完成 hc\_driver 的初始化并赋值一系列与 Chipidea 平台相关的私有数据，如代码清单 16.10 所示。

代码清单 16.10 Chipidea USB host 驱动初始化

---

```

1 int ci_hdrc_host_init(struct ci_hdrc *ci)
2 {
3     struct ci_role_driver *rdrv;
4
5     if (!hw_read(ci, CAP_DCCPARAMS, DCCPARAMS_HC))
6         return -ENXIO;
7
8     rdrv = devm_kzalloc(ci->dev, sizeof(struct ci_role_driver), GFP_KERNEL);
9     if (!rdrv)
10        return -ENOMEM;
11
12     rdrv->start      = host_start;
13     rdrv->stop       = host_stop;
14     rdrv->irq        = host_irq;
15     rdrv->name       = "host";
16     ci->roles[CI_ROLE_HOST] = rdrv;
17
18     ehci_init_driver(&ci_ehci_hc_driver, &ehci_ci_overrides);
19
20     return 0;
21 }
```

---

## 16.3 USB 设备驱动

### 16.3.1 USB 设备驱动的整体结构

这里所说的 USB 设备驱动指的是从主机角度来看，怎样访问被插入的 USB 设备，而不是指 USB 设备本身运行的固件程序。Linux 系统实现了几类通用的 USB 设备驱动（也称客户驱动），划分为如下几个设备类。

- 音频设备类。
- 通信设备类。
- HID（人机接口）设备类。

- 显示设备类。
- 海量存储设备类。
- 电源设备类。
- 打印设备类。
- 集线器设备类。

一般的通用 Linux 设备（如 U 盘、USB 鼠标、USB 键盘等）都不需要工程师再编写驱动，而工程师需要编写的是特定厂商、特定芯片的驱动，而且往往也可以参考已经在内核中提供的驱动模板。

Linux 内核为各类 USB 设备分配了相应的设备号，如 ACM USB 调制解调器的主设备号为 166（默认设备名 /dev/ttyACMn）、USB 打印机的主设备号为 180，次设备号为 0 ~ 15（默认设备名 /dev/lpn）、USB 串口的主设备号为 188（默认设备名 /dev/ttyUSBn）等，详见 <http://www.lanana.org/> 网站的设备列表。

在 debugfs 下，/sys/kernel/debug/usb/devices 包含了 USB 的设备信息，在 Ubuntu 上插入一个 U 盘后，我们在 /sys/kernel/debug/usb/devices 中可看到类似信息。

```
$ sudo cat /sys/kernel/debug/usb/devices

T: Bus=02 Lev=00 Prnt=00 Port=00 Cnt=00 Dev#= 1 Spd=12 MxCh= 8
B: Alloc= 2/900 us ( 0%), #Int= 1, #Iso= 0
D: Ver= 1.10 Cls=09(hub ) Sub=00 Prot=00 MxPS=64 #Cfgs= 1
P: Vendor=1d6b ProdID=0001 Rev= 4.00
S: Manufacturer=Linux 4.0.0-rc1 ohci_hcd
S: Product=OHCI PCI host controller
S: SerialNumber=0000:00:06.0
C:* #Ifs= 1 Cfg#= 1 Atr=e0 MxPwr= 0mA
I: * If#= 0 Alt= 0 #EPs= 1 Cls=09(hub ) Sub=00 Prot=00 Driver=hub
E: Ad=81(I) Atr=03(Int.) MxPS= 2 Ivl=255ms

...
T: Bus=01 Lev=01 Prnt=01 Port=00 Cnt=01 Dev#= 2 Spd=480 MxCh= 0
D: Ver= 2.10 Cls=00(>ifc ) Sub=00 Prot=00 MxPS=64 #Cfgs= 1
P: Vendor=0930 ProdID=6545 Rev= 1.00
S: Manufacturer=Kingston
S: Product=DataTraveler 3.0
S: SerialNumber=60A44C3FAE22EEA0797900F7
C:* #Ifs= 1 Cfg#= 1 Atr=80 MxPwr=498mA
I: * If#= 0 Alt= 0 #EPs= 2 Cls=08(stor.) Sub=06 Prot=50 Driver=usb-storage
E: Ad=81(I) Atr=02(Bulk) MxPS= 512 Ivl=0ms
E: Ad=02(O) Atr=02(Bulk) MxPS= 512 Ivl=0ms
```

通过分析上述记录信息，可以得到系统中 USB 的完整信息。USBView (<http://www.kroah.com/linux-usb/>) 是一个图形化的 GTK 工具，可以显示 USB 信息。

此外，在 sysfs 文件系统中，同样包含了 USB 相关信息的描述，但只限于接口级别。

USB设备和USB接口在sysfs中均表示为单独的USB设备，其目录命名规则如下：

根集线器 - 集线器端口号 (- 集线器端口号 - ...) : 配置 . 接口

下面给出一个 /sys/bus/usb 目录下的树形结构实例，其中的多数文件都是锚定到 /sys/devices 及 /sys/drivers 中相应文件的链接。

```

└── devices
    ├── 1-0:1.0 -> ../../devices/pci0000:00/0000:00:0b.0/usb1/1-0:1.0
    ├── 1-1 -> ../../devices/pci0000:00/0000:00:0b.0/usb1/1-1
    ├── 1-1:1.0 -> ../../devices/pci0000:00/0000:00:0b.0/usb1/1-1/1-1:1.0
    ├── 2-0:1.0 -> ../../devices/pci0000:00/0000:00:06.0/usb2/2-0:1.0
    ├── 2-1 -> ../../devices/pci0000:00/0000:00:06.0/usb2/2-1
    ├── 2-1:1.0 -> ../../devices/pci0000:00/0000:00:06.0/usb2/2-1/2-1:1.0
    ├── usb1 -> ../../devices/pci0000:00/0000:00:0b.0/usb1
    └── usb2 -> ../../devices/pci0000:00/0000:00:06.0/usb2

└── drivers
    ├── hub
        ├── 1-0:1.0 -> ../../../../../../devices/pci0000:00/0000:00:0b.0/usb1/1-0:1.0
        ├── 2-0:1.0 -> ../../../../../../devices/pci0000:00/0000:00:06.0/usb2/2-0:1.0
        ├── bind
        ├── module -> ../../../../../../module/usbcore
        ├── new_id
        ├── remove_id
        ├── uevent
        └── unbind
    └── usb
        ├── 1-1 -> ../../../../../../devices/pci0000:00/0000:00:0b.0/usb1/1-1
        ├── 2-1 -> ../../../../../../devices/pci0000:00/0000:00:06.0/usb2/2-1
        ├── bind
        ├── uevent
        ├── unbind
        ├── usb1 -> ../../../../../../devices/pci0000:00/0000:00:0b.0/usb1
        └── usb2 -> ../../../../../../devices/pci0000:00/0000:00:06.0/usb2

```

正如 tty\_driver、i2c\_driver 等，在Linux内核中，使用 usb\_driver 结构体描述一个USB设备驱动，usb\_driver 结构体的定义如代码清单 16.11 所示。

代码清单 16.11 usb\_driver 结构体

---

```

1 struct usb_driver {
2     const char *name;
3
4     int (*probe) (struct usb_interface * intf,
5                   const struct usb_device_id * id);
6
7     void (*disconnect) (struct usb_interface * intf);
8
9     int (*unlocked_ioctl) (struct usb_interface * intf, unsigned int code,
10                           void *buf);

```

```

11
12     int (*suspend) (struct usb_interface *intf, pm_message_t message);
13     int (*resume) (struct usb_interface *intf);
14     int (*reset_resume) (struct usb_interface *intf);
15
16     int (*pre_reset) (struct usb_interface *intf);
17     int (*post_reset) (struct usb_interface *intf);
18
19     const struct usb_device_id *id_table;
20
21     struct usb_dynids dynids;
22     struct usbdrv_wrap drvwrap;
23     unsigned int no_dynamic_id:l;
24     unsigned int supports_autosuspend:l;
25     unsigned int disable_hub_initiated_lpm:l;
26     unsigned int soft_unbind:l;
27 };

```

---

在编写新的USB设备驱动时，主要应该完成的工作是probe()和disconnect()函数，即探测和断开函数，它们分别在设备被插入和拔出的时候调用，用于初始化和释放软硬件资源。对usb\_driver的注册和注销可通过下面两个函数完成：

```

int usb_register(struct usb_driver *new_driver)
void usb_deregister(struct usb_driver *driver);

```

usb\_driver结构体中的id\_table成员描述了这个USB驱动所支持的USB设备列表，它指向一个usb\_device\_id数组，usb\_device\_id结构体包含有USB设备的制造商ID、产品ID、产品版本、设备类、接口类等信息及其要匹配标志成员match\_flags（标明要与哪些成员匹配，包含DEV\_LO、DEV\_HI、DEV\_CLASS、DEV\_SUBCLASS、DEV\_PROTOCOL、INT\_CLASS、INT\_SUBCLASS、INT\_PROTOCOL）。可以借助下面一组宏来生成usb\_device\_id结构体的实例：

```
USB_DEVICE(vendor, product)
```

该宏根据制造商ID和产品ID生成一个usb\_device\_id结构体的实例，在数组中增加该元素将意味着该驱动可支持与制造商ID、产品ID匹配的设备。

```
USB_DEVICE_VER(vendor, product, lo, hi)
```

该宏根据制造商ID、产品ID、产品版本的最小值和最大值生成一个usb\_device\_id结构体的实例，在数组中增加该元素将意味着该驱动可支持与制造商ID、产品ID匹配和lo~hi范围内版本的设备。

```
USB_DEVICE_INFO(class, subclass, protocol)
```

该宏用于创建一个匹配设备指定类型的usb\_device\_id结构体实例。

```
USB_INTERFACE_INFO(class, subclass, protocol)
```

该宏用于创建一个匹配接口指定类型的 `usb_device_id` 结构体实例。

代码清单 16.12 所示为两个用于描述某 USB 驱动支持的 USB 设备的 `usb_device_id` 结构体数组实例。

代码清单 16.12 `usb_device_id` 结构体数组实例

---

```

1  /* 本驱动支持的 USB 设备列表 */
2
3  /* 实例 1 */
4  static struct usb_device_id id_table [] = {
5      { USB_DEVICE(VENDOR_ID, PRODUCT_ID) },
6      { },
7  };
8  MODULE_DEVICE_TABLE (usb, id_table);
9
10 /* 实例 2 */
11 static struct usb_device_id id_table [] = {
12     { .idVendor = 0x10D2, .match_flags = USB_DEVICE_ID_MATCH_VENDOR, },
13     { },
14 };
15 MODULE_DEVICE_TABLE (usb, id_table);

```

---

当 USB 核心检测到某个设备的属性和某个驱动程序的 `usb_device_id` 结构体所携带的信息一致时，这个驱动程序的 `probe()` 函数就被执行（如果这个 USB 驱动是个模块的话，相关的 `.ko` 还应被 Linux 自动加载）。拔掉设备或者卸掉驱动模块后，USB 核心就执行 `disconnect()` 函数来响应这个动作。

上述 `usb_driver` 结构体中的函数是 USB 设备驱动中与 USB 相关的部分，而 USB 只是一个总线，USB 设备驱动真正的主体工作仍然是 USB 设备本身所属类型的驱动，如字符设备、tty 设备、块设备、输入设备等。因此 USB 设备驱动包含其作为总线上挂接设备的驱动和本身所属设备类型的驱动两部分。

与 `platform_driver`、`i2c_driver` 类似，`usb_driver` 起到了“牵线”的作用，即在 `probe()` 里注册相应的字符、tty 等设备，在 `disconnect()` 注销相应的字符、tty 等设备，而原先对设备的注册和注销一般直接发生在模块加载和卸载函数中。

尽管 USB 本身所属设备驱动的结构与其挂不挂在 USB 总线上没什么关系，但是据此在访问方式上却有很大的变化，例如，对于 USB 接口的字符设备而言，尽管仍然是 `write()`、`read()`、`ioctl()` 这些函数，但是在这些函数中，贯穿始终的是称为 URB 的 USB 请求块。

如图 16.3 所示，在这棵树里，我们把树根比作主机控制器，树叶比作具体的 USB 设备，树干和树枝就是 USB 总线。树叶本身与树枝通过 `usb_driver` 连接，而树叶本身的驱动（读写、控制）则需要通过其树叶设备本身所属类设备驱动来完成。树根和树叶之间的“通信”依靠在树干和树枝里“流淌”的 URB 来完成。



图 16.3 USB 设备驱动结构

由此可见，`usb_driver` 本身只是有找到 USB 设备、管理 USB 设备连接和断开的作用，也就是说，它是公司入口处的“打卡机”，可以获得员工（USB 设备）的上 / 下班情况。树叶和员工一样，可以是研发工程师也可以是销售工程师，而作为 USB 设备的树叶可以是字符树叶、网络树叶或块树叶，因此必须实现相应设备类的驱动。

### 16.3.2 USB 请求块

#### 1. urb 结构体

USB 请求块（USB Request Block, URB）是 USB 设备驱动中用来描述与 USB 设备通信所用的基本载体和核心数据结构，非常类似于网络设备驱动中的 `sk_buff` 结构体。

代码清单 16.13 URB 结构体

---

```

1 struct urb {
2 ...
3     /* public: documented fields in the urb that can be used by drivers */
4     struct list_head urb_list;      /* list head for use by the urb's

```

```

5           * current owner */
6
7           ...
8           struct usb_host_endpoint *ep;      /* (internal) pointer to endpoint */
9           unsigned int pipe;             /* (in) pipe information */
10          unsigned int stream_id;        /* (in) stream ID */
11          int status;                 /* (return) non-ISO status */
12          unsigned int transfer_flags;   /* (in) URB_SHORT_NOT_OK | ... */
13          void *transfer_buffer;       /* (in) associated data buffer */
14          dma_addr_t transfer_dma;    /* (in) dma addr for transfer_buffer */
15          struct scatterlist *sg;      /* (in) scatter gather buffer list */
16          int num_mapped_sgs;         /* (internal) mapped sg entries */
17          int num_sgs;                /* (in) number of entries in the sg list */
18          u32 transfer_buffer_length;  /* (in) data buffer length */
19          u32 actual_length;          /* (return) actual transfer length */
20          unsigned char *setup_packet; /* (in) setup packet (control only) */
21          dma_addr_t setup_dma;       /* (in) dma addr for setup_packet */
22          int start_frame;           /* (modify) start frame (ISO) */
23          int number_of_packets;     /* (in) number of ISO packets */
24          int interval;              /* (modify) transfer interval
25                           * (INT/ISO) */
26          int error_count;            /* (return) number of ISO errors */
27          void *context;              /* (in) context for completion */
28          usb_complete_t complete;    /* (in) completion routine */
29          struct usb_iso_packet_descriptor iso_frame_desc[0];
30      };                         /* (in) ISO ONLY */

```

---

## 2. URB 处理流程

USB 设备中的每个端点都处理一个 URB 队列，在队列被清空之前，一个 URB 的典型生命周期如下。

1) 被一个 USB 设备驱动创建。

创建 URB 结构体的函数为：

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

`iso_packets` 是这个 URB 应当包含的等时数据包的数目，若为 0 表示不创建等时数据包。`mem_flags` 参数是分配内存的标志，和 `kmalloc()` 函数的分配标志参数含义相同。如果分配成功，该函数返回一个 URB 结构体指针，否则返回 0。

URB 结构体在驱动中不宜静态创建，因为这可能破坏 USB 核心给 URB 使用的引用计数方法。

`usb_alloc_urb()` 的“反函数”为：

```
void usb_free_urb(struct urb *urb);
```

该函数用于释放由 `usb_alloc_urb()` 分配的 URB 结构体。

2) 初始化，被安排给一个特定 USB 设备的特定端点。

对于中断 URB，使用 `usb_fill_int_urb()` 函数来初始化 URB，如下所示：

```
void usb_fill_int_urb(struct urb *urb, struct usb_device *dev,
    unsigned int pipe, void *transfer_buffer,
    int buffer_length, usb_complete_t complete,
    void *context, int interval);
```

URB 参数指向要被初始化的 URB 的指针；`dev` 指向这个 URB 要被发送到的 USB 设备；`pipe` 是这个 URB 要被发送到的 USB 设备的特定端点；`transfer_buffer` 是指向发送数据或接收数据的缓冲区的指针，和 URB 一样，它也不能是静态缓冲区，必须使用 `kmalloc()` 来分配；`buffer_length` 是 `transfer_buffer` 指针所指向缓冲区的大小；`complete` 指针指向当这个 URB 完成时被调用的完成处理函数；`context` 是完成处理函数的“上下文”；`interval` 是这个 URB 应当被调度的间隔。

上述函数参数中的 `pipe` 使用 `usb_sndintpipe()` 或 `usb_rcvintpipe()` 创建。

对于批量 URB，使用 `usb_fill_bulk_urb()` 函数来初始化，如下所示：

```
void usb_fill_bulk_urb(struct urb *urb, struct usb_device *dev,
    unsigned int pipe, void *transfer_buffer,
    int buffer_length, usb_complete_t complete,
    void *context);
```

除了没有对应于调度间隔的 `interval` 参数以外，该函数的参数和 `usb_fill_int_urb()` 函数的参数含义相同。

上述函数参数中的 `pipe` 使用 `usb_sndbulkpipe()` 或者 `usb_rcvbulkpipe()` 函数来创建。

对于控制 URB，使用 `usb_fill_control_urb()` 函数来初始化，如下所示：

```
void usb_fill_control_urb(struct urb *urb, struct usb_device *dev,
    unsigned int pipe, unsigned char *setup_packet,
    void *transfer_buffer, int buffer_length,
    usb_complete_t complete, void *context);
```

除了增加了新的 `setup_packet` 参数以外，该函数的参数和 `usb_fill_bulk_urb()` 函数的参数含义相同。`setup_packet` 参数指向即将被发送到端点的设置数据包。

上述函数参数中的 `pipe` 使用 `usb_sndctrlpipe()` 或 `usb_rcvctrlpipe()` 函数来创建。

等时 URB 没有像中断、控制和批量 URB 的初始化函数 `usb_fill_iso_urb()`，我们只能手动对它初始化，而后才能提交给 USB 核心。代码清单 16.14 给出了初始化等时 URB 的例子，它来自 `drivers/media/usb/uvc/uvc_video.c` 文件。

代码清单 16.14 初始化等时 URB

---

```
1      for (i = 0; i < UVC_URBS; ++i) {
2          urb = usb_alloc_urb(npackets, gfp_flags);
3          if (urb == NULL) {
4              uvc_uninit_video(stream, 1);
5              return -ENOMEM;
```

```

6
7
8     urb->dev = stream->dev->udev;
9     urb->context = stream;
10    urb->pipe = usb_rcvisocpipe(stream->dev->udev,
11                                ep->desc.bEndpointAddress);
12 #ifndef CONFIG_DMA_NONCOHERENT
13     urb->transfer_flags = URB_ISO_ASAP | URB_NO_TRANSFER_DMA_MAP;
14     urb->transfer_dma = stream->urb_dma[i];
15 #else
16     urb->transfer_flags = URB_ISO_ASAP;
17 #endif
18     urb->interval = ep->desc.bInterval;
19     urb->transfer_buffer = stream->urb_buffer[i];
20     urb->complete = uvc_video_complete;
21     urb->number_of_packets = npackets;
22     urb->transfer_buffer_length = size;
23
24     for (j = 0; j < npackets; ++j) {
25         urb->iso_frame_desc[j].offset = j * psize;
26         urb->iso_frame_desc[j].length = psize;
27     }
28
29     stream->urb[i] = urb;
30 }

```

---

### 3 ) 被 USB 设备驱动提交给 USB 核心。

在完成第 1)、2) 步的创建和初始化 URB 后，URB 便可以提交给 USB 核心了，可通过 `usb_submit_urb()` 函数来完成，如下所示：

```
int usb_submit_urb(struct urb *urb, gfp_t mem_flags);
```

URB 参数是指向 URB 的指针，`mem_flags` 参数与传递给 `kmalloc()` 函数参数的意义相同，它用于告知 USB 核心如何在此时分配内存缓冲区。

在提交 URB 到 USB 核心后，直到完成函数被调用之前，不要访问 URB 中的任何成员。

`usb_submit_urb()` 在原子上下文和进程上下文中都可以被调用，`mem_flags` 变量需根据调用环境进行相应的设置，如下所示。

- `GFP_ATOMIC`：在中断处理函数、底半部、tasklet、定时器处理函数以及 URB 完成函数中，在调用者持有自旋锁或者读写锁时以及当驱动将 `current->state` 修改为非 `TASK_RUNNING` 时，应使用此标志。
- `GFP_NOIO`：在存储设备的块 I/O 和错误处理路径中，应使用此标志；
- `GFP_KERNEL`：如果没有任何理由使用 `GFP_ATOMIC` 和 `GFP_NOIO`，就使用 `GFP_KERNEL`。

如果 `usb_submit_urb()` 调用成功，即 URB 的控制权被移交给 USB 核心，该函数返回 0；否则，返回错误号。

4) 提交由 USB 核心指定的 USB 主机控制器驱动。

5) 被 USB 主机控制器处理，进行一次到 USB 设备的传送。

第 4) ~ 5) 步由 USB 核心和主机控制器完成，不受 USB 设备驱动的控制。

6) 当 URB 完成，USB 主机控制器驱动通知 USB 设备驱动。

在如下 3 种情况下，URB 将结束，URB 完成回调函数将被调用（完成回调是通过 `usb_fill_xxx_urb` 的参数传入的）。在完成回调中，我们通常要进行 `urb->status` 的判断。

- URB 被成功发送给设备，并且设备返回正确的确认。如果 `urb->status` 为 0，意味着对于一个输出 URB，数据被成功发送；对于一个输入 URB，请求的数据被成功收到。
- 如果发送数据到设备或从设备接收数据时发生了错误，`urb->status` 将记录错误值。
- URB 被从 USB 核心“去除连接”，这发生在驱动通过 `usb_unlink_urb()` 或 `usb_kill_urb()` 函数取消或 URB 虽已提交而 USB 设备被拔出的情况下。

`usb_unlink_urb()` 和 `usb_kill_urb()` 这两个函数用于取消已提交的 URB，其参数为要被取消的 URB 指针。`usb_unlink_urb()` 是异步的，搞定后对应的完成回调会被调用；而 `usb_kill_urb()` 会彻底终止 URB 的生命周期并等待这一行为，它通常在设备的 `disconnect()` 函数中被调用。

当 URB 生命结束时（处理完成或被解除链接），在 URB 的完成回调中通过 URB 结构体的 `status` 成员可以获知其原因，如 0 表示传输成功，`-ENOENT` 表示被 `usb_kill_urb()` 杀死，`-ECONNRESET` 表示被 `usb_unlink_urb()` 杀死，`-EPROTO` 表示传输中发生了 bitstuff 错误或者硬件未能及时收到响应数据包，`-ENODEV` 表示 USB 设备已被移除，`-EXDEV` 表示等时传输仅完成了一部分等。

对以上 URB 的处理步骤进行一个总结，图 16.4 给出了一个 URB 的完整处理流程，虚线框的 `usb_unlink_urb()` 和 `usb_kill_urb()` 并不一定会发生，它们只是在 URB 正在被 USB 核心和主机控制器处理时又被驱动程序取消的情况下才发生。

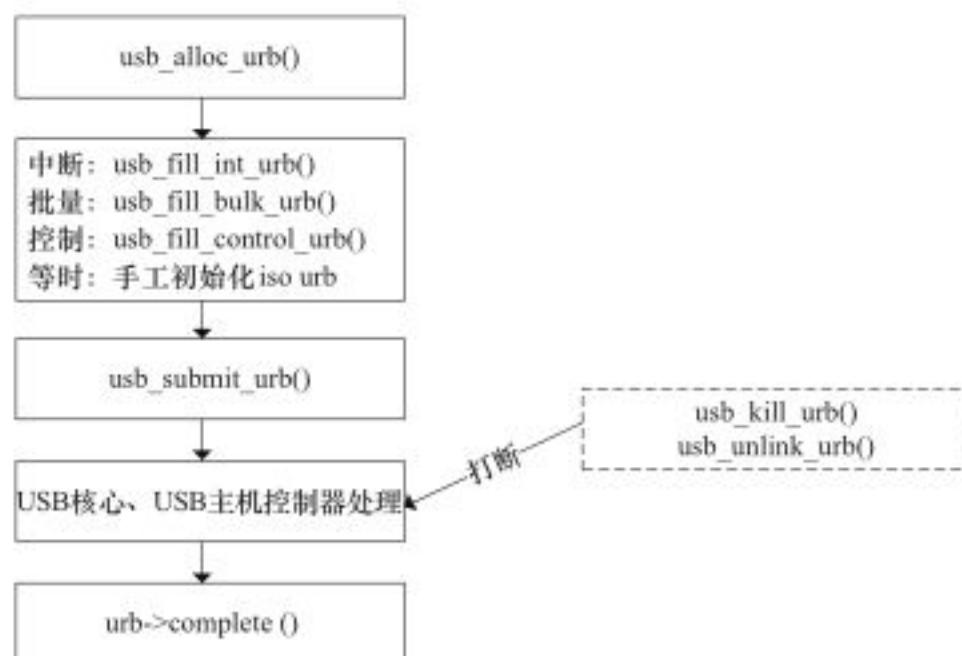


图 16.4 URB 处理流程

### 3. 简单的批量与控制 URB

有时 USB 驱动程序只是从 USB 设备上接收或向 USB 设备发送一些简单的数据，这时候，没有必要将 URB 创建、初始化、提交、完成处理的整个流程走一遍，而可以使用两个更简单的函数，如下所示。

#### (1) `usb_bulk_msg()`

`usb_bulk_msg()` 函数创建一个 USB 批量 URB 并将它发送到特定设备，这个函数是同步的，它一直等待 URB 完成后才返回。`usb_bulk_msg()` 函数的原型为：

```
int usb_bulk_msg(struct usb_device *usb_dev, unsigned int pipe,
                  void *data, int len, int *actual_length,
                  int timeout);
```

`usb_dev` 参数为批量消息要发送的 USB 设备的指针，`pipe` 为批量消息要发送到的 USB 设备的端点，`data` 参数为指向要发送或接收的数据缓冲区的指针，`len` 参数为 `data` 参数所指向的缓冲区的长度，`actual_length` 用于返回实际发送或接收的字节数，`timeout` 是发送超时，以 jiffies 为单位，0 意味着永远等待。

如果函数调用成功，返回 0；否则，返回 1 个负的错误值。

#### (2) `usb_control_msg()` 函数

`usb_control_msg()` 函数与 `usb_bulk_msg()` 函数类似，不过它提供给驱动发送和结束 USB 控制信息而不是批量信息的能力，该函数的原型为：

```
int usb_control_msg(struct usb_device *dev, unsigned int pipe, __u8 request,
                     __u8 requesttype, __u16 value, __u16 index, void *data,
                     __u16 size, int timeout);
```

`dev` 指向控制消息发往的 USB 设备，`pipe` 是控制消息要发往的 USB 设备的端点，`request` 是这个控制消息的 USB 请求值，`requesttype` 是这个控制消息的 USB 请求类型，`value` 是这个控制消息的 USB 消息值，`index` 是这个控制消息的 USB 消息索引值，`data` 指向要发送或接收的数据缓冲区，`size` 是 `data` 参数所指向的缓冲区的大小，`timeout` 是发送超时，以毫秒为单位，0 意味着永远等待。

参数 `request`、`requesttype`、`value` 和 `index` 与 USB 规范中定义的 USB 控制消息直接对应。

如果函数调用成功，该函数返回发送到设备或从设备接收到的字节数；否则，返回一个负的错误值。

对 `usb_bulk_msg()` 和 `usb_control_msg()` 函数的使用要特别慎重，由于它们是同步的，因此不能在中断上下文和持有自旋锁的情况下使用。而且，该函数也不能被任何其他函数取消，因此，务必要使得驱动程序的 `disconnect()` 函数掌握足够的信息，以判断和等待该调用的结束。

### 16.3.3 探测和断开函数

在 USB 设备驱动 `usb_driver` 结构体的 `probe()` 函数中，应该完成如下工作。

- 探测设备的端点地址、缓冲区大小，初始化任何可能用于控制 USB 设备的数据结构。
- 把已初始化的数据结构的指针保存到接口设备中。

`usb_set_intfdata()` 函数可以设置 `usb_interface` 的私有数据，这个函数的原型为：

```
void usb_set_intfdata (struct usb_interface *intf, void *data);
```

这个函数的“反函数”用于得到 `usb_interface` 的私有数据，其原型为：

```
void *usb_get_intfdata (struct usb_interface *intf);
```

- 注册 USB 设备。

如果是简单的字符设备，则可调用 `usb_register_dev()`，这个函数的原型为：

```
int usb_register_dev(struct usb_interface *intf,
                     struct usb_class_driver *class_driver);
```

上述函数中的第二个参数为 `usb_class_driver` 结构体，这个结构体的定义如代码清单 16.15 所示。

代码清单 16.15 `usb_class_driver` 结构体

---

```
1 struct usb_class_driver {
2     char *name;
3     char *(*devnode) (struct device *dev, umode_t *mode);
4     const struct file_operations *fops;
5     int minor_base;
6 };
```

---

对于字符设备而言，`usb_class_driver` 结构体的 `fops` 成员中的 `written()`、`read()`、`ioctl()` 等函数的地位完全等同于本书第 6 章中的 `file_operations` 成员函数。

如果是其他类型的设备，如 `tty` 设备，则调用对应设备的注册函数。

在 USB 设备驱动 `usb_driver` 结构体的 `probe()` 函数中，应该完成如下工作。

- 释放所有为设备分配的资源。
- 设置接口设备的数据指针为 `NULL`。
- 注销 USB 设备。

对于字符设备，可以直接调用 `usb_register_dev()` 函数的“反函数”，如下所示：

```
void usb_deregister_dev(struct usb_interface *intf,
                        struct usb_class_driver *class_driver);
```

对于其他类型的设备，如 `tty` 设备，则调用对应设备的注销函数。

### 16.3.4 USB 骨架程序

Linux 内核源代码中的 `driver/usb/usb-skeleton.c` 文件为我们提供了一个最基础的 USB 驱动程序，即 USB 骨架程序，它可被看作一个最简单的 USB 设备驱动实例。尽管具体的 USB

设备驱动千差万别，但其骨架则万变不离其宗。

首先看看USB骨架程序的usb\_driver结构体定义，如代码清单16.16所示。

代码清单16.16 USB骨架程序的usb\_driver结构体

---

```

1 static struct usb_driver skel_driver = {
2     .name =           "skeleton",
3     .probe =          skel_probe,
4     .disconnect =    skel_disconnect,
5     .suspend =        skel_suspend,
6     .resume =         skel_resume,
7     .pre_reset =      skel_pre_reset,
8     .post_reset =     skel_post_reset,
9     .id_table =       skel_table,
10    .supports_autosuspend = 1,
11 };

```

---

从上述代码第9行可以看出，它所支持的USB设备的列表数组为skel\_table[]，其定义如代码清单16.17所示。

代码清单16.17 USB骨架程序的id\_table

---

```

1 static struct usb_device_id skel_table [] = {
2     { USB_DEVICE(USB_SKEL_VENDOR_ID, USB_SKEL_PRODUCT_ID) },
3     {}                      /* Terminating entry */
4 };
5 MODULE_DEVICE_TABLE(usb, skel_table);

```

---

对上述usb\_driver的注册和注销发生在USB骨架程序的模块加载与卸载函数内，其分别调用了usb\_register()和usb\_deregister()，不过这个注册和注销的代码却不用写出来，直接用一个快捷宏module\_usb\_driver即可，如代码清单16.18所示。

代码清单16.18 USB骨架程序的模块加载

---

```

1 static struct usb_driver skel_driver = {
2     .name =           "skeleton",
3     .probe =          skel_probe,
4     .disconnect =    skel_disconnect,
5     .suspend =        skel_suspend,
6     .resume =         skel_resume,
7     .pre_reset =      skel_pre_reset,
8     .post_reset =     skel_post_reset,
9     .id_table =       skel_table,
10    .supports_autosuspend = 1,
11 };
12
13 module_usb_driver(skel_driver);

```

---

在usb\_driver的probe()成员函数中，会根据usb\_interface的成员寻找第一个批量输入和

输出端点，将端点地址、缓冲区等信息存入为USB骨架程序定义的usb\_skel结构体中，并将usb\_skel实例的指针传入usb\_set\_intfdata()中以作为USB接口的私有数据，最后，它会注册USB设备，如代码清单16.19所示。

代码清单16.19 USB骨架程序的probe()函数

```

1 static int skel_probe(struct usb_interface *interface,
2                      const struct usb_device_id *id)
3 {
4     struct usb_skel *dev;
5     struct usb_host_interface *iface_desc;
6     struct usb_endpoint_descriptor *endpoint;
7     size_t buffer_size;
8     int i;
9     int retval = -ENOMEM;
10
11    /* allocate memory for our device state and initialize it */
12    dev = kzalloc(sizeof(*dev), GFP_KERNEL);
13    ...
14    kref_init(&dev->kref);
15    sema_init(&dev->limit_sem, WRITES_IN_FLIGHT);
16    mutex_init(&dev->io_mutex);
17    spin_lock_init(&dev->err_lock);
18    init_usb_anchor(&dev->submitted);
19    init_waitqueue_head(&dev->bulk_in_wait);
20
21    dev->udev = usb_get_dev(interface_to_usbdev(interface));
22    dev->interface = interface;
23
24    /* set up the endpoint information */
25    /* use only the first bulk-in and bulk-out endpoints */
26    iface_desc = interface->cur_altsetting;
27    for (i = 0; i < iface_desc->desc.bNumEndpoints; ++i) {
28        endpoint = &iface_desc->endpoint[i].desc;
29
30        if (!dev->bulk_in_endpointAddr &&
31            usb_endpoint_is_bulk_in(endpoint)) {
32            /* we found a bulk in endpoint */
33            buffer_size = usb_endpoint_maxp(endpoint);
34            dev->bulk_in_size = buffer_size;
35            dev->bulk_in_endpointAddr = endpoint->bEndpointAddress;
36            dev->bulk_in_buffer = kmalloc(buffer_size, GFP_KERNEL);
37            ...
38            dev->bulk_in_urb = usb_alloc_urb(0, GFP_KERNEL);
39            ...
40        }
41
42        if (!dev->bulk_out_endpointAddr &&
43            usb_endpoint_is_bulk_out(endpoint)) {
44            /* we found a bulk out endpoint */

```

```

45         dev->bulk_out_endpointAddr = endpoint->bEndpointAddress;
46     }
47 }
48 ...
49
50 /* save our data pointer in this interface device */
51 usb_set_intfdata(interface, dev);
52
53 /* we can register the device now, as it is ready */
54 retval = usb_register_dev(interface, &skel_class);
55 ...
56 return 0;
57 ...
58 }

```

`usb_skel` 结构体可以被看作是一个私有数据结构体，其定义如代码清单 16.20 所示，应该根据具体的设备量身定制。

代码清单 16.20 USB 骨架程序的自定义数据结构 `usb_skel`

```

1 struct usb_skel {
2     struct usb_device      *udev;          /* the usb device for this device */
3     struct usb_interface   *interface;    /* the interface for this device */
4     struct semaphore limit_sem; /* limiting the number of writes in progress */
5     struct usb_anchor submitted; /* in case we need to retract our submissions */
6     struct urb   *bulk_in_urb;        /* the urb to read data with */
7     unsigned char  *bulk_in_buffer;    /* the buffer to receive data */
8     size_t       bulk_in_size;        /* the size of the receive buffer */
9     size_t       bulk_in_filled;      /* number of bytes in the buffer */
10    size_t      bulk_in_copied;      /* already copied to user space */
11    __u8        bulk_in_endpointAddr; /* the address of the bulk in endpoint */
12    __u8        bulk_out_endpointAddr; /* the address of the bulk out endpoint */
13    int         errors;           /* the last request tanked */
14    bool        ongoing_read; /* a read is going on */
15    spinlock_t   err_lock;        /* lock for errors */
16    struct kref   kref;
17    struct mutex   io_mutex;        /* synchronize I/O with disconnect */
18    wait_queue_head_t bulk_in_wait; /* to wait for an ongoing read */
19 };

```

USB 骨架程序的断开函数会完成与 `probe()` 函数相反的工作，即设置接口数据为 NULL，注销 USB 设备，如代码清单 16.21 所示。

代码清单 16.21 USB 骨架程序的断开函数

```

1 static void skel_disconnect(struct usb_interface *interface)
2 {
3     struct usb_skel *dev;
4     int minor = interface->minor;
5

```

```

6     dev = usb_get_intfdata(interface);
7     usb_set_intfdata(interface, NULL);
8
9     /* give back our minor */
10    usb_deregister_dev(interface, &skel_class);
11
12    /* prevent more I/O from starting */
13    mutex_lock(&dev->io_mutex);
14    dev->interface = NULL;
15    mutex_unlock(&dev->io_mutex);
16
17    usb_kill_anchored_urbs(&dev->submitted);
18
19    /* decrement our usage count */
20    kref_put(&dev->kref, skel_delete);
21
22    dev_info(&interface->dev, "USB Skeleton #%d now disconnected", minor);
23 }

```

代码清单 16.19 第 54 行 `usb_register_dev(interface, &skel_class)` 中的第二个参数包含了字符设备的 `file_operations` 结构体指针，而这个结构体中的成员实现也是 USB 字符设备的另一个组成成分。代码清单 16.22 给出了 USB 骨架程序的字符设备文件操作 `file_operations` 结构体的定义。

代码清单 16.22 USB 骨架程序的字符设备文件操作结构体

```

1 static const struct file_operations skel_fops = {
2     .owner =      THIS_MODULE,
3     .read =       skel_read,
4     .write =      skel_write,
5     .open =       skel_open,
6     .release =    skel_release,
7     .flush =      skel_flush,
8     .llseek =     noop_llseek,
9 };

```

由于只是一个象征性的骨架程序，`open()` 成员函数的实现非常简单，它根据 `usb_driver` 和次设备号通过 `usb_find_interface()` 获得 USB 接口，之后通过 `usb_get_intfdata()` 获得接口的私有数据并赋予 `file->private_data`，如代码清单 16.23 所示。

代码清单 16.23 USB 骨架程序的字符设备 `open()` 函数

```

1 static int skel_open(struct inode *inode, struct file *file)
2 {
3     struct usb_skel *dev;
4     struct usb_interface *interface;
5     int subminor;
6     int retval = 0;

```

```

7      subminor = iminor(inode);
8
9
10     interface = usb_find_interface(&skel_driver, subminor);
11     ...
12     dev = usb_get_intfdata(interface);
13     ...
14
15     retval = usb_autopm_get_interface(interface);
16     if (retval)
17         goto exit;
18
19     /* increment our usage count for the device */
20     kref_get(&dev->kref);
21
22     /* save our object in the file's private structure */
23     file->private_data = dev;
24
25 exit:
26     return retval;
27 }

```

---

由于在 open() 函数中并没有申请任何软件和硬件资源，所以与 open() 函数对应的 release() 函数不用进行资源的释放，而只需进行减少在 open() 中增加的引用计数等工作。

接下来要分析的是读写函数，前面已经提到，在访问 USB 设备的时候，贯穿其中的“中枢神经”是 URB 结构体。

在 skel\_write() 函数中进行的关于 URB 的操作与 16.3.2 小节的描述完全对应，即进行了 URB 的分配（调用 usb\_alloc\_urb()）、初始化（调用 usb\_fill\_bulk\_urb()）和提交（调用 usb\_submit\_urb()）的操作，如代码清单 16.24 所示。

代码清单 16.24 USB 骨架程序的字符设备写函数

```

1 static ssize_t skel_write(struct file *file, const char *user_buffer,
2                           size_t count, loff_t *ppos)
3 {
4     struct usb_skel *dev;
5     int retval = 0;
6     struct urb *urb = NULL;
7     char *buf = NULL;
8     size_t writesize = min(count, (size_t)MAX_TRANSFER);
9
10    dev = file->private_data;
11    ...
12    spin_lock_irq(&dev->err_lock);
13    retval = dev->errors;
14    ...
15    spin_unlock_irq(&dev->err_lock);
16    ...

```

```

17
18     /* create a urb, and a buffer for it, and copy the data to the urb */
19     urb = usb_alloc_urb(0, GFP_KERNEL);
20     ...
21
22     buf = usb_alloc_coherent(dev->udev, writesize, GFP_KERNEL,
23                               &urb->transfer_dma);
24     ...
25
26     if (copy_from_user(buf, user_buffer, writesize)) {
27         retval = -EFAULT;
28         goto error;
29     }
30
31     /* this lock makes sure we don't submit URBs to gone devices */
32     mutex_lock(&dev->io_mutex);
33     ...
34
35     /* initialize the urb properly */
36     usb_fill_bulk_urb(urb, dev->udev,
37                       usb_sndbulkpipe(dev->udev, dev->bulk_out_endpointAddr),
38                       buf, writesize, skel_write_bulk_callback, dev);
39     urb->transfer_flags |= URB_NO_TRANSFER_DMA_MAP;
40     usb_anchor_urb(urb, &dev->submitted);
41
42     /* send the data out the bulk port */
43     retval = usb_submit_urb(urb, GFP_KERNEL);
44     mutex_unlock(&dev->io_mutex);
45     ...
46     usb_free_urb(urb);
47
48     return writesize;
49     ...
50 }

```

在写函数中发起的URB结束后，第38行填入的完成函数skel\_write\_bulk\_callback()将被调用，它会进行urb->status的判断，如代码清单16.25所示。

代码清单 16.25 USB 骨架程序的字符设备写操作完成函数

---

```

1 static void skel_write_bulk_callback(struct urb *urb)
2 {
3     struct usb_skel *dev;
4
5     dev = urb->context;
6
7     /* sync/async unlink faults aren't errors */
8     if (urb->status) {
9         if (!(urb->status == -ENOENT ||

```

```

10         urb->status == -ECONNRESET ||
11         urb->status == -ESHUTDOWN))
12         dev_err(&dev->interface->dev,
13                 "%s - nonzero write bulk status received: %d\n",
14                 __func__, urb->status);
15
16     spin_lock(&dev->err_lock);
17     dev->errors = urb->status;
18     spin_unlock(&dev->err_lock);
19 }
20
21 /* free up our allocated buffer */
22 usb_free_coherent(urb->dev, urb->transfer_buffer_length,
23                     urb->transfer_buffer, urb->transfer_dma);
24 up(&dev->limit_sem);
25 }

```

---

### 16.3.5 实例：USB 键盘驱动

在 Linux 系统中，键盘被认定为标准输入设备，对于一个 USB 键盘而言，其驱动主要由两部分组成：usb\_driver 的成员函数以及输入设备驱动的 input\_event 获取和报告。

在 USB 键盘设备驱动的模块加载和卸载函数中，将分别注册和注销对应于 USB 键盘的 usb\_driver 结构体 usb\_kbd\_driver，代码清单 16.26 所示为模块加载与卸载函数以及 usb\_driver 结构体的定义。

代码清单 16.26 USB 键盘设备驱动的模块加载与卸载函数以及 usb\_driver 结构体

```

1 static struct usb_device_id usb_kbd_id_table [] = {
2     { USB_INTERFACE_INFO(USB_INTERFACE_CLASS_HID, USB_INTERFACE_SUBCLASS_BOOT,
3             USB_INTERFACE_PROTOCOL_KEYBOARD) },
4     {}                                     /* Terminating entry */
5 };
6
7 MODULE_DEVICE_TABLE (usb, usb_kbd_id_table);
8
9 static struct usb_driver usb_kbd_driver = {
10     .name =        "usbkbd",
11     .probe =       usb_kbd_probe,
12     .disconnect =  usb_kbd_disconnect,
13     .id_table =    usb_kbd_id_table,
14 };
15
16 module_usb_driver(usb_kbd_driver);

```

---

在 usb\_driver 的 probe() 函数中，将进行输入设备的初始化和注册，USB 键盘要使用的中断 URB 和控制 URB 的初始化，并设置接口的私有数据，如代码清单 16.27 所示。

代码清单 16.27 USB 键盘设备驱动的 probe() 函数

```

1 static int usb_kbd_probe(struct usb_interface *iface,
2                           const struct usb_device_id *id)
3 {
4     struct usb_device *dev = interface_to_usbdev(iface);
5     struct usb_host_interface *interface;
6     struct usb_endpoint_descriptor *endpoint;
7     struct usb_kbd *kbd;
8     struct input_dev *input_dev;
9     ...
10    interface = iface->cur_altsetting;
11
12    endpoint = &interface->endpoint[0].desc;
13
14    pipe = usb_rcvintpipe(dev, endpoint->bEndpointAddress);
15    maxp = usb_maxpacket(dev, pipe, usb_pipeout(pipe));
16
17    kbd = kzalloc(sizeof(struct usb_kbd), GFP_KERNEL);
18    input_dev = input_allocate_device();
19
20    if (usb_kbd_alloc_mem(dev, kbd))
21        goto fail2;
22
23    kbd->usbdev = dev;
24    kbd->dev = input_dev;
25    ...
26    usb_make_path(dev, kbd->phys, sizeof(kbd->phys));
27    strlcat(kbd->phys, "/input0", sizeof(kbd->phys));
28
29    input_dev->name = kbd->name;
30    input_dev->phys = kbd->phys;
31    usb_to_input_id(dev, &input_dev->id);
32    input_dev->dev.parent = &iface->dev;
33
34    input_set_drvdata(input_dev, kbd);
35
36    input_dev->evbit[0] = BIT_MASK(EV_KEY) | BIT_MASK(EV_LED) |
37        BIT_MASK(EV_REP);
38    ...
39    input_dev->event = usb_kbd_event;
40    input_dev->open = usb_kbd_open;
41    input_dev->close = usb_kbd_close;
42
43    usb_fill_int_urb(kbd->irq, dev, pipe,
44                      kbd->new, (maxp > 8 ? 8 : maxp),
45                      usb_kbd_irq, kbd, endpoint->bInterval);
46    kbd->irq->transfer_dma = kbd->new_dma;
47    kbd->irq->transfer_flags |= URB_NO_TRANSFER_DMA_MAP;
48    ...
49    usb_fill_control_urb(kbd->led, dev, usb_sndctrlpipe(dev, 0),

```

```

50         (void *) kbd->cr, kbd->leds, 1,
51             usb_kbd_led, kbd);
52     kbd->led->transfer_dma = kbd->leds_dma;
53     kbd->led->transfer_flags |= URB_NO_TRANSFER_DMA_MAP;
54
55     error = input_register_device(kbd->dev);
56     if (error)
57         goto fail2;
58
59     usb_set_intfdata(iface, kbd);
60     device_set_wakeup_enable(&dev->dev, 1);
61     return 0;
62     ...
63 }

```

在 `usb_driver` 的断开函数中，将设置接口私有数据为 `NULL`、终止已提交的 URB 并注销输入设备，如代码清单 16.28 所示。

代码清单 16.28 USB 键盘设备驱动的断开函数

```

1 static void usb_kbd_disconnect(struct usb_interface *intf)
2 {
3     struct usb_kbd *kbd = usb_get_intfdata(intf);
4
5     usb_set_intfdata(intf, NULL);
6     if (kbd) {
7         usb_kill_urb(kbd->irq);
8         input_unregister_device(kbd->dev);
9         usb_kill_urb(kbd->led);
10        usb_kbd_free_mem(interface_to_usbdev(intf), kbd);
11        kfree(kbd);
12    }
13 }

```

键盘主要依赖于中断传输模式，在键盘中断 URB 的完成函数 `usb_kbd_irq()` 中（通过代码清单 16.27 的第 45 行可以看出），将会通过 `input_report_key()` 报告按键事件，通过 `input_sync()` 报告同步事件，如代码清单 16.29 所示。

代码清单 16.29 USB 键盘设备驱动的中断 URB 完成函数

```

1 static void usb_kbd_irq(struct urb *urb)
2 {
3     ...
4     for (i = 0; i < 8; i++)
5         input_report_key(kbd->dev, usb_kbd_keycode[i + 224], (kbd->new[0] >> i) & 1);
6
7     for (i = 2; i < 8; i++) {
8
9         if (kbd->old[i]>3 && memscan(kbd->new + 2, kbd->old[i], 6)==kbd->new + 8) {

```

```

10         if (usb_kbd_keycode[kbd->old[i]])
11             input_report_key(kbd->dev, usb_kbd_keycode[kbd->old[i]], 0);
12         else
13             hid_info(urb->dev,
14                     "Unknown key (scancode %#x) released.\n",
15                     kbd->old[i]);
16     }
17
18     if (kbd->new[i] > 3&&memscan(kbd->old + 2, kbd->new[i], 6)==kbd->old + 8) {
19         if (usb_kbd_keycode[kbd->new[i]])
20             input_report_key(kbd->dev, usb_kbd_keycode[kbd->new[i]], 1);
21         else
22             hid_info(urb->dev,
23                     "Unknown key (scancode %#x) pressed.\n",
24                     kbd->new[i]);
25     }
26 }
27
28 input_sync(kbd->dev);
29
30 ...
31 }

```

从 USB 键盘驱动的例子中，我们进一步看到了 `usb_driver` 本身只是起一个挂接总线的作用，而具体设备类型的驱动仍然是工作的主体，例如键盘就是 `input`、USB 串口就是 `tty`，只是在这些设备底层进行硬件访问的时候，调用的都是与 URB 相关的接口，这套 USB 核心层 API——URB 的存在使我们无须关心底层 USB 主机控制器的具体细节，因此，USB 设备驱动也变得与平台无关，同样的驱动可应用于不同的 SoC。

## 16.4 USB UDC 与 Gadget 驱动

### 16.4.1 UDC 和 Gadget 驱动的关键数据结构与 API

这里的 USB 设备控制器（UDC）驱动指的是作为其他 USB 主机控制器外设的 USB 硬件设备上底层硬件控制器的驱动，该硬件和驱动负责将一个 USB 设备依附于一个 USB 主机控制器上。例如，当某运行 Linux 系统的手机作为 PC 的 U 盘时，手机中的底层 USB 控制器行使 USB 设备控制器的功能，这时候运行在底层的是 UDC 驱动，而手机要成为 U 盘，在 UDC 驱动之上仍然需要另外一个驱动，对于 USB 大容量存储器而言，这个驱动为 File Storage 驱动，称为 Function 驱动。从图 16.1 左边可以看出，USB 设备驱动调用 USB 核心的 API，因此具体驱动与 SoC 无关；同样，从图 16.1 右边可以看出，Function 驱动调用通用的 Gadget Function API，因此具体 Function 驱动也变得与 SoC 无关。软件分层设计的好处再一次得到了深刻的体现。

UDC 驱动和 Function 驱动都位于内核的 drivers/usb/gadget 目录中，如 drivers/usb/gadget/udc 下面的 fsl\_mxc\_udc.c、omap\_udc.c、s3c2410\_udc.c 等是对应 SoC 平台上的 UDC 驱动，而 drivers/usb/gadget/function 子目录的 f\_serial.c、f\_mass\_storage.c、f\_rndis.c 等文件实现了一些 Gadget 功能，重要的 Function 驱动如下所示。

**Ethernet over USB**：该驱动模拟以太网网口，它支持多种运行方式——CDC Ethernet（实现标准的 Communications Device Class "Ethernet Model" 协议）、CDC Subset 以及 RNDIS（微软公司对 CDC Ethernet 的变种实现）。

**File-Backed Storage Gadget**：最常见的 U 盘功能实现。

**Serial Gadget**：包括 Generic Serial 实现（只需要 Bulk-in/Bulk-out 端点 +ep0）和 CDC ACM 规范实现。内核源代码中的 Documentation/usb/gadget\_serial.txt 文档讲解了如何将 Serial Gadget 与 Windows 和 Linux 主机连接。

**Gadget MIDI**：暴露 ALSA MIDI 接口。

**USB Video Class Gadget 驱动**：让 Linux 系统成为另外一个系统的 USB 视频采集源。

另外，drivers/usb/gadget 源代码还实现了一个 Gadget 文件系统（GadgetFS），可以将 Gadget API 接口暴露给应用层，以便在应用层实现用户空间的驱动。

在 USB 设备控制器驱动中，我们主要关心几个核心的数据结构，这些数据结构包括描述一个 USB 设备控制器的 usb\_gadget、UDC 操作 usb\_gadget\_ops、描述一个端点的 usb\_ep 以及描述端点操作的 usb\_ep\_ops 结构体。UDC 驱动围绕这些数据结构及其成员函数而展开，代码清单 16.30 列出了这些关键的数据结构，它们都定义于 include/linux/usb/gadget.h 文件。

代码清单 16.30 UDC 驱动的关键数据结构

---

```

1 struct usb_gadget {
2     struct work_struct           work;
3     /* readonly to gadget driver */
4     const struct usb_gadget_ops *ops;
5     struct usb_ep                *ep0;
6     struct list_head              ep_list;      /* of usb_ep */
7     enum usb_device_speed        speed;
8     enum usb_device_speed        max_speed;
9     enum usb_device_state        state;
10    const char                  *name;
11    struct device                dev;
12    unsigned                     out_epnum;
13    unsigned                     in_epnum;
14
15    unsigned                     sg_supported:1;
16    unsigned                     is_otg:1;
17    unsigned                     is_a_peripheral:1;
18    unsigned                     b_hnp_enable:1;
19    unsigned                     a_hnp_support:1;
20    unsigned                     a_alt_hnp_support:1;
21    unsigned                     quirk_ep_out_aligned_size:1;

```

```

22     unsigned           is_selfpowered;
23 };
24
25 struct usb_ep {
26     void             *driver_data;
27
28     const char       *name;
29     const struct usb_ep_ops  *ops;
30     struct list_head    ep_list;
31     unsigned           maxpacket:16;
32     unsigned           maxpacket_limit:16;
33     unsigned           max_streams:16;
34     unsigned           mult:2;
35     unsigned           maxburst:5;
36     u8                address;
37     const struct usb_endpoint_descriptor  *desc;
38     const struct usb_ss_ep_comp_descriptor *comp_desc;
39 };
40
41 struct usb_gadget_ops {
42     int    (*get_frame)(struct usb_gadget *);
43     int    (*wakeup)(struct usb_gadget *);
44     int    (*set_selfpowered) (struct usb_gadget *, int is_selfpowered);
45     int    (*vbus_session) (struct usb_gadget *, int is_active);
46     int    (*vbus_draw) (struct usb_gadget *, unsigned mA);
47     int    (*pullup) (struct usb_gadget *, int is_on);
48     int    (*ioctl)(struct usb_gadget *,
49                     unsigned code, unsigned long param);
50     void   (*get_config_params)(struct usb_dcd_config_params *);
51     int    (*udc_start)(struct usb_gadget *,
52                         struct usb_gadget_driver *);
53     int    (*udc_stop)(struct usb_gadget *);
54 };
55 struct usb_ep_ops {
56     int (*enable) (struct usb_ep *ep,
57                    const struct usb_endpoint_descriptor *desc);
58     int (*disable) (struct usb_ep *ep);
59
60     struct usb_request *(*alloc_request) (struct usb_ep *ep,
61                                            gfp_t gfp_flags);
62     void (*free_request) (struct usb_ep *ep, struct usb_request *req);
63
64     int (*queue) (struct usb_ep *ep, struct usb_request *req,
65                   gfp_t gfp_flags);
66     int (*dequeue) (struct usb_ep *ep, struct usb_request *req);
67
68     int (*set_halt) (struct usb_ep *ep, int value);
69     int (*set_wedge) (struct usb_ep *ep);
70
71     int (*fifo_status) (struct usb_ep *ep);
72     void (*fifo_flush) (struct usb_ep *ep);
73 };

```



---

```

36  /* private: */
37  /* internals */
38  struct list_head      list;
39  DECLARE_BITMAP(endpoints, 32);
40  const struct usb_function_instance *fi;
41  };

```

---

第4行的 `fs_descriptors` 是全速和低速的描述符表；第5行的 `hs_descriptors` 是高速描述符表；`ss_descriptors` 是超高速描述符。`bind()` 完成在 Gadget 注册时获取 I/O 缓冲、端点等资源。

在 `usb_function` 的成员函数以及各种描述符准备好后，在内核通过 `usb_function_register()` API 来完成 Gadget Function 的注册，该 API 的原型为：

```
int usb_function_register(struct usb_function_driver *newf);
```

在 Gadget 驱动中，用 `usb_request` 结构体来描述一次传输请求，这个结构体的地位类似于 USB 主机侧的 URB。`usb_request` 结构体的定义如代码清单 16.32 所示。

代码清单 16.32 `usb_request` 结构体

---

```

1  struct usb_request {
2      void                  *buf; /* Buffer used for data */
3      unsigned             length;
4      dma_addr_t           dma; /* DMA address corresponding to 'buf' */
5
6      struct scatterlist     *sg; /* a scatterlist for SG-capable controllers */
7      unsigned             num_sgs;
8      unsigned             num_mapped_sgs;
9
10     unsigned            stream_id:16;
11     unsigned            no_interrupt:1;
12     unsigned            zero:1;
13     unsigned            short_not_ok:1;
14
15     void                (*complete)(struct usb_ep *ep,
16                                struct usb_request *req); /* Function called when request completes */
17     void                *context;
18     struct list_head      list;
19
20     int                 status;
21     unsigned            actual;
22  };

```

---

在 `include/linux/usb/gadget.h` 文件中，还封装了一些常用的 API，以供 Gadget Function 驱动调用，从而便于它们操作端点，如下所示。

### (1) 使能和禁止端点

```
static inline int usb_ep_enable(struct usb_ep *ep);
static inline int usb_ep_disable(struct usb_ep *ep);
```

它们分别调用了 “`ep->ops->enable(ep, desc);`” 和 “`ep->ops->disable(ep);`”。

### (2) 分配和释放 usb\_request

```
struct usb_request *alloc_ep_req(struct usb_ep *ep, int len, int default_len);
static inline struct usb_request *usb_ep_alloc_request(struct usb_ep *ep,
                                                       gfp_t gfp_flags);
static inline void usb_ep_free_request(struct usb_ep *ep,
                                       struct usb_request *req);
```

usb\_ep\_alloc\_request() 和 usb\_ep\_free\_request() 分别调用了 “ep->ops->alloc\_request(ep, gfp\_flags);” 和 “ep->ops->free\_request(ep, req);”，以用于分配和释放一个依附于某端点的 usb\_request，而 alloc\_ep\_req() 则是内嵌了对 usb\_ep\_alloc\_request(ep, GFP\_ATOMIC) 的调用，同时自动申请了 usb\_request 的缓冲器的内存。

### (3) 提交和取消 usb\_request

```
static inline int usb_ep_queue(struct usb_ep *ep,
                               struct usb_request *req, gfp_t gfp_flags);
static inline int usb_ep_dequeue(struct usb_ep *ep, struct usb_request *req);
```

它们分别调用 “ep->ops->queue(ep, req, gfp\_flags);” 和 “ep->ops->dequeue(ep, req);”。usb\_ep\_queue 函数告诉 UDC 完成 usb\_request (读写缓冲)，当请求被完成后，与该请求对应的 completion() 函数会被调用。

### (4) 端点 FIFO 管理

```
static inline int usb_ep_fifo_status(struct usb_ep *ep);
static inline void usb_ep_fifo_flush(struct usb_ep *ep);
```

前者调用 “ep->ops->fifo\_status(ep)” 返回目前 FIFO 中的字节数，后者调用 “ep->ops->fifo\_flush(ep)” 以冲刷掉 FIFO 中的数据。

### (5) 端点自动配置

```
struct usb_ep *usb_ep_autoconfig(
    struct usb_gadget *gadget,
    struct usb_endpoint_descriptor *desc);
```

根据端点描述符及控制器端点情况，分配一个合适的端点。

## 16.4.2 实例：Chipidea USB UDC 驱动

drivers/usb/chipidea/udc.c 是 Chipidea USB UDC 驱动的主体代码，代码清单 16.33 列出了它的初始化流程部分。它定义了 usb\_ep\_ops、usb\_gadget\_ops，在最终进行 usb\_add\_gadget\_udc() 之前填充好了 UDC 的端点列表。

代码清单 16.33 Chipidea USB UDC 驱动实例

---

```
1 static const struct usb_ep_ops usb_ep_ops = {
2     .enable        = ep_enable,
3     .disable       = ep_disable,
```

```

4     .alloc_request  = ep_alloc_request,
5     .free_request   = ep_free_request,
6     .queue          = ep_queue,
7     .dequeue         = ep_dequeue,
8     .set_halt        = ep_set_halt,
9     .set_wedge       = ep_set_wedge,
10    .fifo_flush     = ep_fifo_flush,
11  };
12
13 static const struct usb_gadget_ops usb_gadget_ops = {
14     .vbus_session= ci_udc_vbus_session,
15     .wakeup      = ci_udc_wakeup,
16     .set_selfpowered = ci_udc_selfpowered,
17     .pullup       = ci_udc_pullup,
18     .vbus_draw    = ci_udc_vbus_draw,
19     .udc_start    = ci_udc_start,
20     .udc_stop     = ci_udc_stop,
21  };
22
23 static int init_eps(struct ci_hdrc *ci)
24 {
25     int retval = 0, i, j;
26
27     for (i = 0; i < ci->hw_ep_max/2; i++)
28         for (j = RX; j <= TX; j++) {
29             int k = i + j * ci->hw_ep_max/2;
30             struct ci_hw_ep *hwep = &ci->ci_hw_ep[k];
31
32             ...
33
34             hwep->ep.name      = hwep->name;
35             hwep->ep.ops        = &usb_ep_ops;
36
37             usb_ep_set_maxpacket_limit(&hwep->ep, (unsigned short)~0);
38
39             ...
40
41             /*
42              * set up shorthands for ep0 out and in endpoints,
43              * don't add to gadget's ep_list
44              */
45             if (i == 0) {
46                 if (j == RX)
47                     ci->ep0out = hwep;
48                 else
49                     ci->ep0in = hwep;
50
51             usb_ep_set_maxpacket_limit(&hwep->ep, CTRL_PAYLOAD_MAX);
52             continue;
53         }
54
55     list_add_tail(&hwep->ep.ep_list, &ci->gadget.ep_list);

```

```

56      }
57
58  return retval;
59 }
60
61 static int udc_start(struct ci_hdrc *ci)
62 {
63     ...
64     ci->gadget.ops     = &usb_gadget_ops;
65     ci->gadget.speed    = USB_SPEED_UNKNOWN;
66     ci->gadget.max_speed = USB_SPEED_HIGH;
67     ci->gadget.is_otg   = ci->is_otg ? 1 : 0;
68     ci->gadget.name     = ci->platdata->name;
69
70     INIT_LIST_HEAD(&ci->gadget.ep_list);
71
72     ...
73
74     retval = init_eps(ci);
75     if (retval)
76         goto free_pools;
77
78     ci->gadget.ep0 = &ci->ep0in->ep;
79
80     retval = usb_add_gadget_udc(dev, &ci->gadget);
81     ...
82 }

```

---

### 16.4.3 实例：Loopback Function 驱动

drivers/usb/gadget/function/f\_loopback.c 实现了一个最简单的 Loopback 驱动，它完成的主要工作如下。

- 1) 实现 `usb_function` 实例及其中的成员函数 `bind()`、`set_alt()`、`disable()`、`free_func()` 等成员函数。
- 2) 准备 USB 外设的配置描述符接口描述符 `usb_interface_descriptor`、端点描述符 `usb_endpoint_descriptor` 等。
- 3) 发起 `usb_request` 处理 `usb_request` 的完成并回环。

代码清单 16.34 是抽取了 drivers/usb/gadget/function/f\_loopback.c 文件中能反映一个 Function 驱动主体结构的少量代码。

代码清单 16.34 Loopback USB Gadget Function 驱动实例

```

1 static struct usb_interface_descriptor loopback_intf = (
2     .bLength           = sizeof(loopback_intf),
3     .bDescriptorType   = USB_DT_INTERFACE,
4

```

```

5     .bNumEndpoints =      2,
6     .bInterfaceClass =    USB_CLASS_VENDOR_SPEC,
7     /* .iInterface = DYNAMIC */
8 };
9
10 static struct usb_endpoint_descriptor fs_loop_source_desc = {
11     .bLength =          USB_DT_ENDPOINT_SIZE,
12     .bDescriptorType =   USB_DT_ENDPOINT,
13
14     .bEndpointAddress =  USB_DIR_IN,
15     .bmAttributes =      USB_ENDPOINT_XFER_BULK,
16 };
17 static struct usb_descriptor_header *fs_loopback_descs[] = {
18     (struct usb_descriptor_header *) &loopback_intf,
19     (struct usb_descriptor_header *) &fs_loop_sink_desc,
20     (struct usb_descriptor_header *) &fs_loop_source_desc,
21     NULL,
22 };
23 static struct usb_string strings_loopback[] = {
24     [0].s = "loop input to output",
25     ( )                                /* end of list */
26 };
27
28 static struct usb_gadget_strings stringtab_loop = {
29     .language      = 0x0409,           /* en-us */
30     .strings       = strings_loopback,
31 };
32
33 static struct usb_gadget_strings *loopback_strings[] = {
34     &stringtab_loop,
35     NULL,
36 };
37
38 static int loopback_bind(struct usb_configuration *c, struct usb_function *f)
39 {
40 ...
41     loop->in_ep = usb_ep_autoconfig(cdev->gadget, &fs_loop_source_desc);
42 ...
43     loop->out_ep = usb_ep_autoconfig(cdev->gadget, &fs_loop_sink_desc);
44     if (!loop->out_ep)
45         goto autoconf_fail;
46     loop->out_ep->driver_data = cdev;           /* claim */
47
48     /* support high speed hardware */
49     hs_loop_source_desc.bEndpointAddress =
50         fs_loop_source_desc.bEndpointAddress;
51     hs_loop_sink_desc.bEndpointAddress = fs_loop_sink_desc.bEndpointAddress;
52
53     /* support super speed hardware */
54     ss_loop_source_desc.bEndpointAddress =

```

```
55     fs_loop_source_desc.bEndpointAddress;
56 ss_loop_sink_desc.bEndpointAddress = fs_loop_sink_desc.bEndpointAddress;
57
58 ret = usb_assign_descriptors(f, fs_loopback_descs, hs_loopback_descs,
59                             ss_loopback_descs);
60 ...
61 return 0;
62 }
63
64 static void lb_free_func(struct usb_function *f)
65 {
66 ...
67 usb_free_all_descriptors(f);
68 kfree(func_to_loop(f));
69 }
70
71 static struct usb_function *loopback_alloc(struct usb_function_instance *fi)
72 {
73 ...
74     loop->function.name = "loopback";
75     loop->function.bind = loopback_bind;
76     loop->function.set_alt = loopback_set_alt;
77     loop->function.disable = loopback_disable;
78     loop->function.strings = loopback_strings;
79
80     loop->function.free_func = lb_free_func;
81
82     return &loop->function;
83 }
84
85 static void loopback_complete(struct usb_ep *ep, struct usb_request *req)
86 {
87 ...
88 }
89
90 static int enable_endpoint(struct usb_composite_dev *cdev, struct f_loopback *loop,
91                           struct usb_ep *ep)
92 {
93     struct usb_request           *req;
94 ...
95     result = config_ep_by_speed(cdev->gadget, &(loop->function), ep);
96
97     result = usb_ep_enable(ep);
98
99     ep->driver_data = loop;
100
101    for (i = 0; i < glen && result == 0; i++) {
102        req = lb_alloc_ep_req(ep, 0);
103        if (!req)
104            goto fail1;
```

```

105
106     req->complete = loopback_complete;
107     result = usb_ep_queue(ep, req, GFP_ATOMIC);
108     if (result) {
109         ERROR(cdev, "%s queue req --> %d\n",
110               ep->name, result);
111         goto fail1;
112     }
113 }
114 ...
115 }
116 }

```

---

## 16.5 USB OTG 驱动

USB OTG 标准在完全兼容 USB 2.0 标准的基础上，它允许设备既可作为主机，也可作为外设操作，OTG 新增了主机通令协议（HNP）和对话请求协议（SRP）。

在 OTG 中，初始主机设备称为 A 设备，外设称为 B 设备。可用电缆的连接方式来决定初始角色。两用设备使用新型 Mini-AB 插座，从而使 Mini-A 插头、Mini-B 插头和 Mini-AB 插座增添了第 5 个引脚（ID），以用于识别不同的电缆端点。Mini-A 插头中的 ID 引脚接地，Mini-B 插头中的 ID 引脚浮空。当 OTG 设备检测到接地的 ID 引脚时，表示默认的是 A 设备（主机），而检测到 ID 引脚浮空的设备则认为是 B 设备（外设）。系统一旦连接后，OTG 的角色还可以更换，以采用新的 HNP 协议。而 SRP 允许 B 设备请求 A 设备打开 VBUS 电源并启动一次对话。一次 OTG 对话可通过 A 设备提供 VBUS 电源的时间来确定。

自 Linux 2.6.9 开始，OTG 相关源代码已经被包含在内核中了，新增的主要内容包括：

### (1) UDC 驱动端添加的 OTG 相关属性和函数

```

struct usb_gadget {
    ...
    unsigned          is_otg:1;
    unsigned          is_a_peripheral:1;
    unsigned          b_hnp_enable:1;
    unsigned          a_hnp_support:1;
    unsigned          a_alt_hnp_support:1;
    ...
};

int usb_gadget_vbus_connect(struct usb_gadget *gadget);
int usb_gadget_vbus_disconnect(struct usb_gadget *gadget);

int usb_gadget_vbus_draw(struct usb_gadget *gadget, unsigned mA);

/* 控制 USB D+ 的上拉 */

```

```

int usb_gadget_connect(struct usb_gadget *gadget);
int usb_gadget_disconnect(struct usb_gadget *gadget);

/* 尝试唤醒USB host, 它也会尝试SRP会话 */
int usb_gadget_wakeup(struct usb_gadget *gadget);

```

### (2) Gadget 驱动端添加的 OTG 相关属性和函数

如果 `gadget->is_otg` 字段为真，则增加一个 OTG 描述符；通过 `printf()`、LED 等方式报告 HNP 可用；当挂起开始时，通过用户界面报告 HNP 切换开始（B-Peripheral 到 B-Host 或 A-Peripheral 到 A-Host）。

### (3) 主机侧添加的 OTG 相关属性和函数

在 USB 核心中新增了关于 OTG 设备枚举的信息：

```

struct usb_bus {
    ...
    u8 otg_port;           /* 0, or index of OTG/HNP port */
    unsigned is_b_host:1;   /* true during some HNP roleswitches */
    unsigned b_hnp_enable:1; /* OTG: did A-Host enable HNP? */
    ...
};

```

为了实现 HNP 需要的挂起 / 恢复，新增如下通用接口：

```

int usb_suspend_device(struct usb_device *dev, u32 state);
int usb_resume_device(struct usb_device *dev);

```

### (4) 新增 OTG 功能切换和协议的描述结构体 `usb_otg`

```

struct usb_otg {
    u8             default_a;

    struct phy          *phy;
    /* old usb_phy interface */
    struct usb_phy      *usb_phy;
    struct usb_bus      *host;
    struct usb_gadget   *gadget;

    enum usb_otg_state state;

    /* bind/unbind the host controller */
    int   (*set_host)(struct usb_otg *otg, struct usb_bus *host);

    /* bind/unbind the peripheral controller */
    int   (*set_peripheral)(struct usb_otg *otg,
                           struct usb_gadget *gadget);

    /* effective for A-peripheral, ignored for B devices */
    int   (*set_vbus)(struct usb_otg *otg, bool enabled);

    /* for B devices only: start session with A-Host */
};

```

```

    int (*start_srp)(struct usb_otg *otg);

    /* start or continue HNP role switch */
    int (*start_hnp)(struct usb_otg *otg);

};

usb_otg 的代码一般在 USB 的 phy 端实现，目前，从内核的 drivers/usb/musb/musb_gadget.c、drivers/usb/phy/phy-tw16030-usb.c、drivers/usb/phy/phy-isp1301-omap.c、drivers/usb/phy/phy-fsl-usb.c 和 drivers/usb/musb/musb_gadget.c 等驱动中可以找到类似的例子。

```

## 16.6 总结

USB 驱动分为 USB 主机驱动和 USB 设备驱动，如果系统的 USB 主机控制器符合 OHCI 等标准，那主机驱动的绝大部分工作都可以沿用通用的代码。

对于一个 USB 设备而言，它至少具备两重身份：首先它是“USB”的，其次它是“自己”的。USB 设备是“USB”的，指它挂接在 USB 总线上，其必须完成 `usb_driver` 的初始化和注册；USB 设备是“自己”的，意味着本身可能是一个字符设备、tty 设备、网络设备等，因此，在 USB 设备驱动中也必须实现符合相应框架的代码。

USB 设备驱动的自身设备驱动部分的读写等操作流程有其特殊性，即以 URB 来贯穿始终，一个 URB 的生命周期通常包含创建、初始化、提交和被 USB 核心及 USB 主机传递、完成后回调函数被调用的过程，当然，在 URB 被驱动提交后，也可以被取消。

在 UDC 和 Gadget Function 侧，UDC 关心底层的硬件操作，而 Function 驱动则只是利用通用的 API，并通过 `usb_request` 与底层 UDC 驱动交互。