LINUX

业界权威机构和专家强力推荐
多年培训、研发经验的总结

设备驱动开发详解

华清远见嵌入式培训中心 宋宝华 编著

Linux

附源代码光盘

人民邮电出版社
POSTS & TELECOM PRESS

第 20 章　USB 主机与设备驱动

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角度看到的 USB 主机控制器驱动和设备驱动。

20.1 节给出了 Linux 系统中 USB 驱动的整体视图，讲解了 Linux 中主机侧和设备侧角度的 USB 驱动层次。

从主机侧的角度而言，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。本章 20.2 节分析了 USB 主机控制器驱动的结构并给出实例，20.3 节讲解了 USB 设备驱动的结构及其设备请求块处理过程，并分析了 USB 设备驱动的骨架程序，20.4 节则给出了 Linux 设备驱动的实例。

20.1 节与 20.2～20.4 节是整体与部分的关系，20.2 节与 20.3～20.4 节是并列关系。

20.1 Linux USB 驱动层次

20.1.1 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 20.1 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。

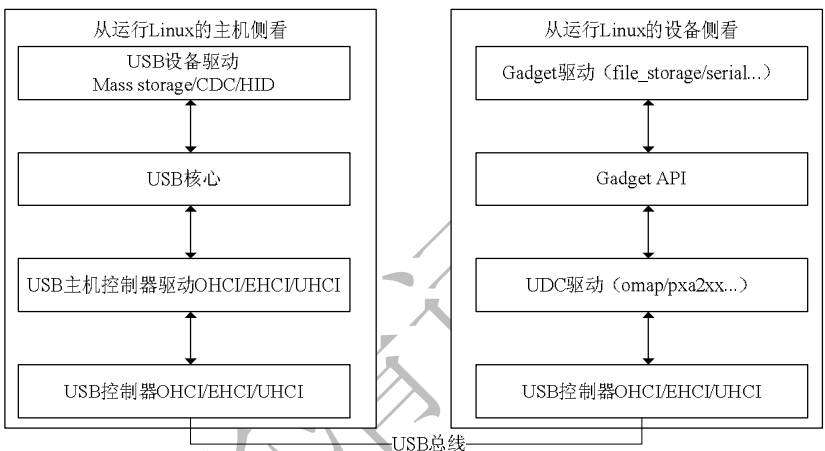

图 20.1 Linux USB 驱动总体结构

如图 20.1 的左侧所示，从主机侧的观念去看，在 Linux 驱动中，USB 驱动处于最底层的是 USB 主机控制器硬件，在其之上运行的是 USB 主机控制器驱动，主机控制器之上为 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 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；通过全局变量维护整个系统的 USB 设备信息；完成设备热插拔控制、总线数据传输控制等。

如图 20.1 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 API 和 Gadget 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 API 是 UDC 驱动程序回调函数的简单包装。Gadget 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 API 控制 UDC 实现上述功能。Gadget API 把下层的 UDC 驱动程序和上层的 Gadget 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。

本章将重点讲解从主机侧角度看到的 USB 主机控制器驱动与 USB 设备驱动，关于设备侧的 Linux 驱动，将不会详细讲解。

20.1.2 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 设备都提供了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惟一的地址，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承载数据，或者从主机到设备（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 20.2 所示，这些单元之间的关系如下：

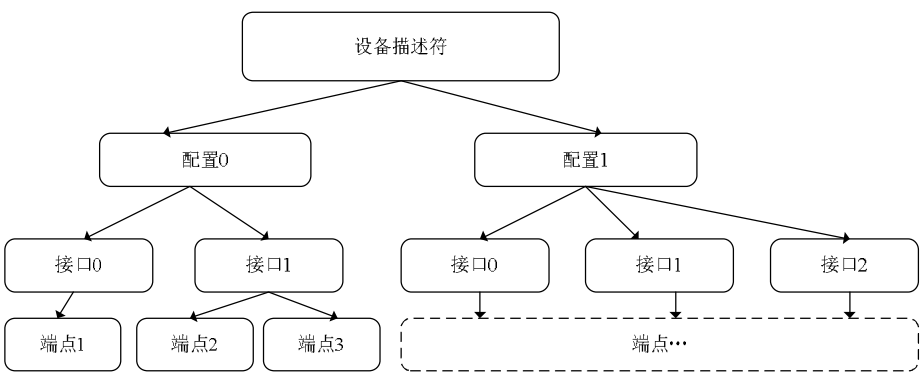

图 20.2 USB 设备、配置、接口和端点

- 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，如代码清单 20.1 所示。

代码清单 20.1　usb_device_descriptor 结构体

```
1  struct usb_device_descriptor
2  {
3   __u8  bLength; //描述符长度
4   __u8  bDescriptorType; //描述符类型编号
5
6   __le16 bcdUSB; //USB 版本号
7   __u8  bDeviceClass; //USB 分配的设备类 code
8   __u8  bDeviceSubClass;// USB 分配的子类 code
9   __u8  bDeviceProtocol; //USB 分配的协议 code
10  __u8  bMaxPacketSize0; //endpoint0 最大包大小
11  __le16 idVendor; //厂商编号
12  __le16 idProduct; //产品编号
13  __le16 bcdDevice; //设备出厂编号
14  __u8  iManufacturer; //描述厂商字符串的索引
15  __u8  iProduct; //描述产品字符串的索引
16  __u8  iSerialNumber; //描述设备序列号字符串的索引
17  __u8  bNumConfigurations; //可能的配置数量
18 } __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，USB 配置描述符定义为结构体 usb_config_descriptor，如代码清单 20.2 所示。

代码清单 20.2　usb_config_descriptor 结构体

```
1  struct usb_config_descriptor
2  {
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
```

```
    __u8  iConfiguration; //描述该配置的字符串的索引值
    __u8  bmAttributes; //供电模式的选择
    __u8  bMaxPower; //设备从总线提取的最大电流
} __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，USB 接口描述符定义为结构体 usb_interface_descriptor，如代码清单 20.3 所示。

代码清单 20.3　usb_interface_descriptor 结构体

```
struct usb_interface_descriptor
{
    __u8  bLength;          //描述符长度
    __u8  bDescriptorType; //描述符类型

    __u8  bInterfaceNumber;   // 接口的编号
    __u8  bAlternateSetting; //备用的接口描述符编号
    __u8  bNumEndpoints;      //该接口使用的端点数，不包括端点 0
    __u8  bInterfaceClass;    //接口类型
    __u8  bInterfaceSubClass; //接口子类型
    __u8  bInterfaceProtocol; //接口所遵循的协议
    __u8  iInterface; //描述该接口的字符串索引值
} __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，USB 端点描述符定义为 usb_endpoint_descriptor 结构体，如代码清单 20.4 所示。

代码清单 20.4　usb_endpoint_descriptor 结构体

```
struct usb_endpoint_descriptor
{
    __u8  bLength; //描述符长度
    __u8  bDescriptorType; //描述符类型
    __u8  bEndpointAddress; //端点地址: 0~3 位是端点号，第 7 位是方向(0-OUT,1-IN)
    __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
    __le16 wMaxPacketSize; //// 本端点接收或发送的最大信息包的大小
    __u8  bInterval;    //轮询数据传送端点的时间间隔
                        //对于批量传送的端点以及控制传送的端点，此域忽略
                        //对于同步传送的端点，此域必须为 1
                        //对于中断传送的端点，此域值的范围为 1~255
    __u8  bRefresh;
    __u8  bSynchAddress;
} __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 20.5 所示。

代码清单 20.5　usb_string_descriptor 结构体

```
1 struct usb_string_descriptor
2 {
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，笔者在运行 Linux 2.6.15.5 的系统上插入一个 SanDisk U 盘后，通过 lsusb 命令得到这个 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、一个接口描述符以及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)
```

20.2 USB 主机驱动

20.2.1 USB 主机驱动的整体结构

USB 主机控制器有 3 种规格：OHCI (Open Host Controller Interface)、UHCI (Universal Host Controller Interface) 和 EHCI (Enhanced 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。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。本节将重点介绍嵌入式系统中常用的 OHCI 主机控制器驱动。

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 20.6 所示。

代码清单 20.6 usb_hcd 结构体

```
1  struct usb_hcd
2  {
3    /* 管理“家务” */
4    struct usb_bus self;
5    const char *product_desc; /* 产品/厂商字符串 */
```

```
    6   char irq_descr[24]; /* 驱动 + 总线 # */
    7
    8   struct timer_list rh_timer; /* 根 Hub 轮询 */
    9   struct urb *status_urb; /* 目前的状态 urb */
   10
   11   /* 硬件信息/状态 */
   12   const struct hc_driver *driver; /* 硬件特定的钩子函数 */
   13
   14   /* 需要维护的标志 */
   15   unsigned long flags;
   16   #define HCD_FLAG_HW_ACCESSIBLE  0x00000001
   17   #define HCD_FLAG_SAW_IRQ        0x00000002
   18
   19   unsigned rh_registered: 1; /* 根 Hub 注册? */
   20
   21   /* 下一个标志的采用只是“权益之计”，当所有 HCDs 支持新的根 Hub 轮询机制后将
移除 */
   22   unsigned uses_new_polling: 1;
   23   unsigned poll_rh: 1; /* 轮询根 Hub 状态? */
   24   unsigned poll_pending: 1; /* 状态已经改变? */
   25
   26   int irq; /* 被分配的 irq */
   27   void __iomem *regs; /* 设备内存和 I/O */
   28   u64 rsrc_start; /* 内存和 I/O 资源开始位置 */
   29   u64 rsrc_len; /* 内存和 I/O 资源长度 */
   30   unsigned power_budget; /* mA, 0 = 无限制 */
   31
   32   #define HCD_BUFFER_POOLS        4
   33   struct dma_pool *pool[HCD_BUFFER_POOLS];
   34
   35   int state;
   36   #define __ACTIVE              0x01
   37   #define __SUSPEND             0x04
   38   #define __TRANSIENT           0x80
   39
   40   #define HC_STATE_HALT         0
   41   #define HC_STATE_RUNNING      (__ACTIVE)
   42   #define HC_STATE_QUIESCING   (__SUSPEND|__TRANSIENT|__ACTIVE)
   43   #define HC_STATE_RESUMING     (__SUSPEND|__TRANSIENT)
   44   #define HC_STATE_SUSPENDED    (__SUSPEND)
   45
   46   #define HC_IS_RUNNING(state) ((state) & __ACTIVE)
   47   #define HC_IS_SUSPENDED(state) ((state) & __SUSPEND)
   48   /* 主机控制器驱动的私有数据 */
```

```
    49   unsigned long hcd_priv[0]__attribute__((aligned(sizeof(unsigned
long))));
    50 };
```

usb_hcd 中的 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，其定义如代码清单 20.7 所示。

代码清单 20.7　hc_driver 结构体

```
    1  struct hc_driver
    2  {
    3    const char *description; /* "ehci-hcd" 等 */
    4    const char *product_desc; /* 产品/厂商字符串 */
    5    size_t hcd_priv_size; /* 私有数据的大小 */
    6
    7    /* 中断处理函数 */
    8    irqreturn_t(*irq)(struct usb_hcd *hcd, struct pt_regs *regs);
    9
    10   int flags;
    11   #define HCD_MEMORY     0x0001     /* HC 寄存器使用的内存和 I/O */
    12   #define HCD_USB11      0x0010        /* USB 1.1 */
    13   #define HCD_USB2       0x0020        /* USB 2.0 */
    14
    15   /* 被调用以初始化 HCD 和根 Hub */
    16   int(*reset)(struct usb_hcd *hcd);
    17   int(*start)(struct usb_hcd *hcd);
    18
    19   /* 挂起 Hub 后，进入 D3(etc)前被调用 */
    20   int(*suspend)(struct usb_hcd *hcd, pm_message_t message);
    21
    22   /* 在进入 D0(etc)后，恢复 Hub 前调用 */
    23   int(*resume)(struct usb_hcd *hcd);
    24
    25   /* 使 HCD 停止写内存和进行 I/O 操作 */
    26   void(*stop)(struct usb_hcd *hcd);
    27
    28   /* 返回目前的帧数 */
    29   int(*get_frame_number)(struct usb_hcd *hcd);
    30
    31   /* 管理 I/O 请求和设备状态 */
    32   int(*urb_enqueue)(struct usb_hcd *hcd, struct usb_host_endpoint
*ep, struct
    33     urb *urb, gfp_t mem_flags);
    34   int(*urb_dequeue)(struct usb_hcd *hcd, struct urb *urb);
    35
    36   /* 释放 endpoint 资源 */
    37         void(*endpoint_disable)(struct   usb_hcd   *hcd,   struct
usb_host_endpoint *ep);
    38
    39   /* 根 Hub 支持 */
    40   int(*hub_status_data)(struct usb_hcd *hcd, char *buf);
    41   int(*hub_control)(struct usb_hcd *hcd, u16 typeReq, u16 wValue, u16
wIndex,
    42     char *buf, u16 wLength);
    43   int(*bus_suspend)(struct usb_hcd*);
    44   int(*bus_resume)(struct usb_hcd*);
    45   int(*start_port_reset)(struct usb_hcd *, unsigned port_num);
```

```
46   void(*hub_irq_enable)(struct usb_hcd*);
47 };
```

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);
```

2．OHCI 主机控制器驱动

OHCI HCD 驱动属于 HCD 驱动的实例，它定义了一个 ohci_hcd 结构体，作为代码清单 20.6 给出的 usb_hcd 结构体的私有数据，这个结构体的定义如代码清单 20.8 所示。

代码清单 20.8　ohci_hcd 结构体

```
1  struct ohci_hcd
2  {
3    spinlock_t lock;
4
5    /* 与主机控制器通信的 I/O 内存(DMA 一致) */
6    struct ohci_regs __iomem *regs;
7
8    /* 与主机控制器通信的主存(DMA 一致) */
9    struct ohci_hcca *hcca;
10   dma_addr_t hcca_dma;
11
12   struct ed *ed_rm_list; /* 将被移除 */
13   struct ed *ed_bulktail; /* 批量队列尾 */
14   struct ed *ed_controltail; /* 控制队列尾 */
15   struct ed *periodic[NUM_INTS]; /* int_table “影子” */
16
17   /* OTG 控制器和收发器需要软件交互，其他的外部收发器应该是软件透明的 */
18   struct otg_transceiver *transceiver;
19
20   /* 队列数据的内存管理 */
21   struct dma_pool *td_cache;
22   struct dma_pool *ed_cache;
23   struct td *td_hash[TD_HASH_SIZE];
24   struct list_head pending;
25
26   /* driver 状态 */
27   int num_ports;
28   int load[NUM_INTS];
29   u32 hc_control; /* 主机控制器控制寄存器的复制 */
30   unsigned long next_statechange; /* 挂起/恢复 */
31   u32 fminterval; /* 被保存的寄存器 */
32
33   struct notifier_block reboot_notifier;
34   unsigned long flags;
```

```
35 };
```

使用如下内联函数可实现 usb_hcd 和 ohci_hcd 的相互转换：

```
struct ohci_hcd *hcd_to_ohci (struct usb_hcd *hcd);
struct usb_hcd *ohci_to_hcd (const struct ohci_hcd *ohci);
```

从 usb_hcd 得到 ohci_hcd 只是取得“私有”数据，而从 ohci_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 OHCI 主机控制器：

```
int ohci_init (struct ohci_hcd *ohci);
```

如下函数分别用于开启、停止及复位 OHCI 控制器：

```
int ohci_run (struct ohci_hcd *ohci);
void ohci_stop (struct usb_hcd *hcd);
void ohci_usb_reset (struct ohci_hcd *ohci);
```

OHCI 主机控制器驱动的主机工作仍然是实现代码清单 20.7 给出的 hc_driver 结构体中的成员函数。

20.2.2 实例：S3C2410 USB 主机驱动

S3C2410 内部集成了一个 USB 主机控制器，完全兼容 OCHI 1.0、USB 1.1 标准，支持低速和全速 USB 设备，从基地址 0x49000000 开始分别提供了 OHCI 的 HcRevision、HcControl、HcCommonStatus、HcInterruptStatus、HcInterruptEnable、HcInterruptDisable 、 HcHCCA 、 HcPeriodCuttentED 、 HcControlHeadED 、 HcControlCurrentED 、 HcBulkHeadED 、 HcBulk CurrentED 、 HcDoneHead 、 HcRmInterval、HcFmRemaining、HcFmNumber、HcPeriodicStart、HcLSThreshold、HcRhDescriptorA 、 HcRhDescriptorB 、 HcRhStatus 、 HcRhPortStatus1 、 HcRhPortStatus2 寄存 器。

S3C2410 主机控制器驱动 hc_driver 结构体中的大多数成员函数都是通用的 ohci_xxx()函数，而 start()、hub_status_data()、hub_control()函数则针对 S3C2410 而编写的，如代码清单 20.9 所示。

代码清单 20.9 S3C2410 主机控制器驱动的 hc_driver 结构体

```
1  static const struct hc_driver ohci_s3c2410_hc_driver =
2  {
3    .description =      hcd_name,
4    .product_desc =     "S3C24XX OHCI",
5    .hcd_priv_size =    sizeof(struct ohci_hcd),
6
7    /* 通用硬件联接    */
8    .irq =  ohci_irq,
9    .flags =    HCD_USB11 | HCD_MEMORY, //USB 1.1标准，hc寄存器位于内存
10
11   /* 基本的生命周期操作 */
12   .start =        ohci_s3c2410_start,
13   .stop =     ohci_stop,
14
15   /* 管理I/O请求和相关的设备资源 */
16   .urb_enqueue =      ohci_urb_enqueue,
17   .urb_dequeue =      ohci_urb_dequeue,
18   .endpoint_disable = ohci_endpoint_disable,
19
20   /* 调度支持 */
21   .get_frame_number = ohci_get_frame,
```

```
22
23  /* 根 Hub 支持 */
24  .hub_status_data = ohci_s3c2410_hub_status_data,
25  .hub_control =     ohci_s3c2410_hub_control,
26 #ifdef   CONFIG_PM
27  .bus_suspend =     ohci_bus_suspend,
28  .bus_resume =      ohci_bus_resume,
29 #endif
30  .start_port_reset = ohci_start_port_reset,
31 };
```

hc_driver 的 start()成员函数用于初始化 OHCI 并启动主机控制器，如代码清单 20.10 所示。

代码清单 20.10　S3C2410 主机控制器驱动的 start()函数

```
1  static int ohci_s3c2410_start (struct usb_hcd *hcd)
2  {
3   struct ohci_hcd *ohci = hcd_to_ohci (hcd);
4   int ret;
5
6   if ((ret = ohci_init(ohci)) < 0) //初始化 ohci_hcd
7       return ret;
8
9   if ((ret = ohci_run (ohci)) < 0) { //启动 ohci_hcd
10      err ("can't start %s", hcd->self.bus_name);
11      ohci_stop (hcd);
12      return ret;
13  }
14
15  return 0;
16 }
```

hc_driver 的 hub_control()成员函数 ohci_s3c2410_hub_control()中的主体是调用通用的 ohci_ hub_control()函数，hub_status_data()成员函数 ohci_s3c2410_hub_status_data()的主体是调用通用的 ohci_hub_status_data()函数。

20.3 USB 设备驱动

20.3.1　USB 设备驱动整体结构

Linux 系统实现了几类通用的 USB 设备驱动，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。

一般的通用的 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，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内核中已提供的驱动的模板。

这里所说的 USB 设备驱动指的是从主机角度观察，怎样访问被插入的 USB 设备，而不是指 USB 设备内部本身运行的固件程序。USB 设备内的固件称为“设备用户固件”，“设备用户固件”完成设备内部的控制任务，并在 USB 传输中对接收到的设备请求做出解释并予以正确响应。

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。

内核中提供了 USB 设备文件系统（usbdevfs，Linux 2.6 改为 usbfs，即 USB 文件系统），它和/proc 类似，都是动态产生的。通过在/etc/fstab 文件中添加如下一行：

```
none /proc/bus/usb usbfs defaults
```

或者输入命令：

```
mount -t usbfs none /proc/bus/usb
```

可以实现 USB 设备文件系统的挂载。

一个典型的/proc/bus/usb/devices 文件的结构如下（笔者在 VmWare 上运行的 Linux 2.6.15.5 内核上的机器上插入了一个 SanDisk U 盘）：

```
T:  Bus=01 Lev=00 Prnt=00 Port=00 Cnt=00 Dev#=  1 Spd=12  MxCh= 2
B:  Alloc=  0/900 us ( 0%), #Int=  0, #Iso=  0
D:  Ver= 1.10 Cls=09(hub  ) Sub=00 Prot=00 MxPS=64 #Cfgs=  1
P:  Vendor=0000 ProdID=0000 Rev= 2.06
S:  Manufacturer=Linux 2.6.15.5 uhci_hcd
S:  Product=UHCI Host Controller
S:  SerialNumber=0000:00:07.2
C:* #Ifs= 1 Cfg#= 1 Atr=c0 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=12  MxCh= 0
D:  Ver= 2.00 Cls=00(>ifc ) Sub=00 Prot=00 MxPS=64 #Cfgs=  1
P:  Vendor=0781 ProdID=5151 Rev= 0.10
S:  Manufacturer=SanDisk Corporation
S:  Product=Cruzer Micro
S:  SerialNumber=20060877500A1BE1FDE1
C:* #Ifs= 1 Cfg#= 1 Atr=80 MxPwr=200mA
I:  If#= 0 Alt= 0 #EPs= 2 Cls=08(stor.) Sub=06 Prot=50 Driver=(none)
E:  Ad=81(I) Atr=02(Bulk) MxPS= 512 Ivl=0ms
E:  Ad=01(O) Atr=02(Bulk) MxPS= 512 Ivl=0ms
```

通过分析 usbfs 中记录的信息，可以得到系统中 USB 完整的信息，例如，usbview 可以以图形化的方式显示系统中的 USB 设备。

当然，在编译 Linux 内核时，应该包括“USB device filesystem”，如图 20.3 所示。usbfs 动态跟踪总线上插入和移除的设备，通过它可以查看系统中 USB 设备的信息，包括拓扑、带宽、设备描述符信息、产品 ID、字符串描述符、配置

描述符、接口描述符、端点描述　　符等。

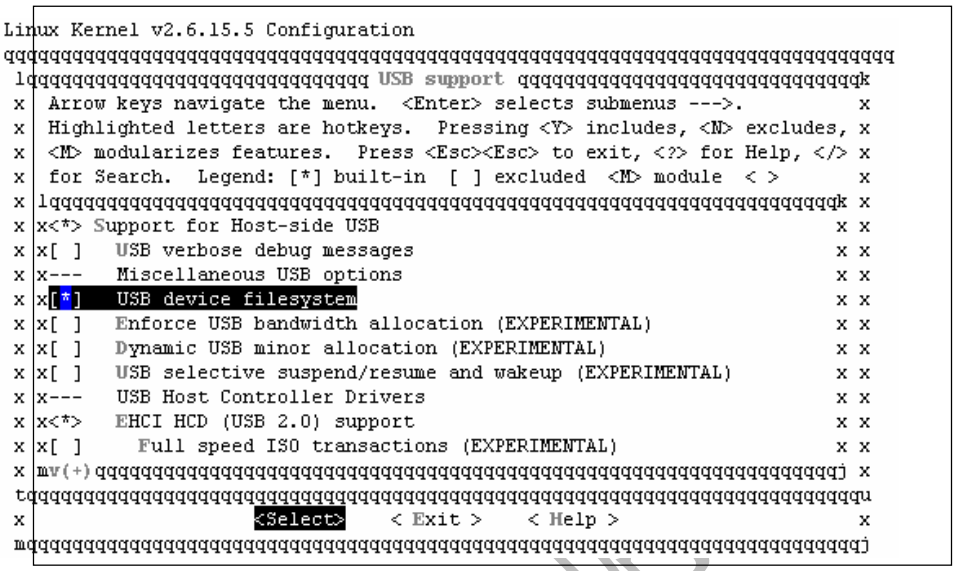

图 20.3　USB 设备文件系统编译

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。

下面讲解/sys/bus/usb 目录的树形结构实例，其中的多数文件都是到/sys/devices 及/sys/drivers 中相应文件的链接。

```
    usb
    |-- devices
    |                                          |--              1-0:1.0
-> ../../../devices/pci0000:00/0000:00:07.2/usb1/1-0:1.0
    |   |-- 1-1 -> ../../../devices/pci0000:00/0000:00:07.2/usb1/1-1
    |                                          |--              1-1:1.0
-> ../../../devices/pci0000:00/0000:00:07.2/usb1/1-1/1-1:1.0
    |   '-- usb1 -> ../../../devices/pci0000:00/0000:00:07.2/usb1
    '-- drivers
        |-- hub
        |                                           |--              1-0:1.0
-> ../../../../devices/pci0000:00/0000:00:07.2/usb1/1-0:1.0
        |   |-- bind
        |   |-- module -> ../../../../module/usbcore
        |   '-- unbind
        |-- usb
        |                                              |--              1-1
-> ../../../../devices/pci0000:00/0000:00:07.2/usb1/1-1
        |   |-- bind
        |   |-- module -> ../../../../module/usbcore
        |   |-- unbind
```

```
       |   '-- usb1 -> ../../../../devices/pci0000:00/0000:00:07.2/usb1
       '-- usbfs
          |-- bind
          |-- module -> ../../../../module/usbcore
          '-- unbind
```

正如 tty_driver、pci_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 20.11 所示。

代码清单 20.11　usb_driver 结构体

```
1  struct usb_driver {
2   const char *name; /* 驱动名称 */
4   int (*probe) (struct usb_interface *intf,
5                     const struct usb_device_id *id); /*探测函数*/
7   void (*disconnect) (struct usb_interface *intf); /*断开函数*/
9   int (*ioctl) (struct usb_interface *intf, unsigned int code,
10                      void *buf);    /* I/O控制函数 */
12    int  (*suspend)  (struct  usb_interface  *intf,  pm_message_t
message);/*挂起函数*/
13  int (*resume) (struct usb_interface *intf); /* 恢复函数 */
15  void (*pre_reset) (struct usb_interface *intf);
16  void (*post_reset) (struct usb_interface *intf);
18  const struct usb_device_id *id_table;/* usb_device_id表指针 */
20  struct usb_dynids dynids;
21  struct usbdrv_wrap drvwrap;
22  unsigned int no_dynamic_id:1;
23  unsigned int supports_autosuspend:1;
24 };
```

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被调用，用于初始化和释放软硬件资源。对 usb_driver 的注册和注销通过这两个函数完成：

```
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 结构体用于包含 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-Flash（标明要与哪些成员匹配）。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匹配制造商 ID、产品 ID 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匹配制造商 ID、产品 ID 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 20.12 所示为两个用于描述某 USB 驱动所支持的 USB 设备的 usb_device_id 结构体数组实例。

代码清单 20.12 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);
```

上述 usb_driver 结构体中的函数是 USB 设备驱动中 USB 相关的部分，而 USB 只是一个总线，真正的 USB 设备驱动的主体工作仍然是 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在设备的驱动和本身所属设备类型的驱动两部分。

与 platform_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不挂在 USB 总线上时完全相同，但是在访问方式上却发生了很大的变化，例如，对于字符设备而言，尽管仍然是 write()、read()、ioctl()这些函数，但是在这些函数中，与 USB 设备通信时不再是 I/O 内存和 I/O 端口的访问，而贯穿始终的是称为 URB 的 USB 请求块。

如图 20.4 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。

由此可见，usb_driver 本身只是起到了找到 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。

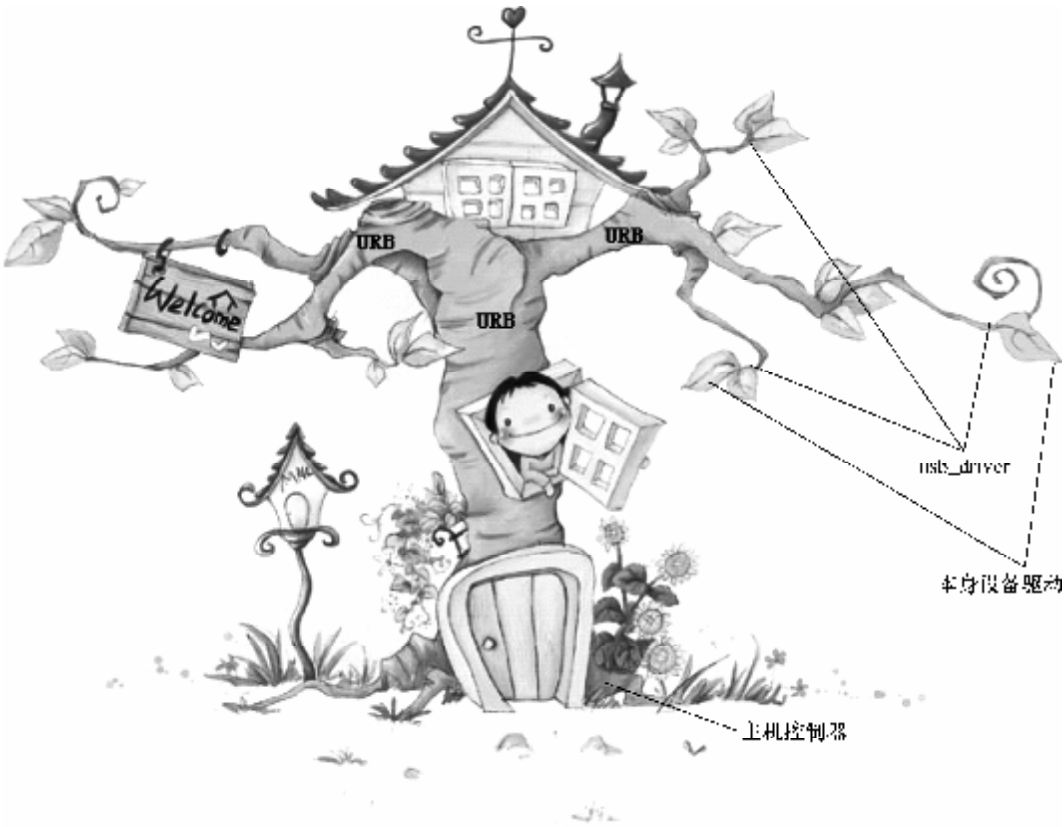

图 20.4　USB 设备驱动结构

20.3.2　USB 请求块（URB）

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，是 USB 主机与设备通信的“电波”。

代码清单 20.13　urb 结构体

```
1  struct urb
2  {
3    /* 私有的: 只能由 USB 核心和主机控制器访问的字段 */
4    struct kref kref; /*urb 引用计数 */
5    spinlock_t lock; /* urb 锁 */
6    void *hcpriv; /* 主机控制器私有数据 */
7    int bandwidth; /* INT/ISO 请求的带宽 */
8    atomic_t use_count; /* 并发传输计数 */
9    u8 reject; /* 传输将失败*/
10
11   /* 公共的:  可以被驱动使用的字段 */
12   struct list_head urb_list; /* 链表头*/
13   struct usb_device *dev; /* 关联的 USB 设备 */
14   unsigned int pipe; /* 管道信息 */
15   int status; /* URB 的当前状态 */
```

```
    16   unsigned int transfer_flags; /* URB_SHORT_NOT_OK | ...*/
    17   void *transfer_buffer; /* 发送数据到设备或从设备接收数据的缓冲区 */
    18   dma_addr_t transfer_dma; /*用来以 DMA 方式向设备传输数据的缓冲区 */
    19   int transfer_buffer_length;/*transfer_buffer 或 transfer_dma 指向
缓冲区的大小 */
    20
    21   int actual_length; /* URB 结束后，发送或接收数据的实际长度 */
    22   unsigned char *setup_packet; /* 指向控制 URB 的设置数据包的指针*/
    23   dma_addr_t setup_dma; /*控制 URB 的设置数据包的 DMA 缓冲区*/
    24   int start_frame; /*等时传输中用于设置或返回初始帧*/
    25   int number_of_packets; /*等时传输中等时缓冲区数据 */
    26   int interval; /* URB 被轮询到的时间间隔（对中断和等时 urb 有效） */
    27   int error_count;  /* 等时传输错误数量 */
    28   void *context; /* completion 函数上下文 */
    29   usb_complete_t complete; /* 当 URB 被完全传输或发生错误时，被调用 */
    30   struct usb_iso_packet_descriptor iso_frame_desc[0];
    31   /*单个 URB 一次可定义多个等时传输时，描述各个等时传输 */
    32 };
```

当 transfer_flags 标志中的 URB_NO_TRANSFER_DMA_MAP 被置位时，USB 核心将使用 transfer_dma 指向的缓冲区而非 transfer_buffer 指向的缓冲区，意味着即将传输 DMA 缓冲区。

当 transfer_flags 标志中的 URB_NO_SETUP_DMA_MAP 被置位时，对于有 DMA 缓冲区的控制 urb 而言，USB 核心将使用 setup_dma 指向的缓冲区而非 setup_packet 指向的缓冲区。

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, int 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 结构体在驱动中不能静态创建，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 urb，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 urb，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_rcvictrlpipe()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，我们只能手动地初始化 urb，而后才能提交给 USB 核心。代码清单 20.14 给出了初始化等时 urb 的例子，它来自 drivers/usb/media/usbvideo.c 文件。

代码清单 20.14　初始化等时 urb

```
1  for (i = 0; i < USBVIDEO_NUMSBUF; i++)
2  {
3    int j, k;
4    struct urb *urb = uvd->sbuf[i].urb;
5    urb->dev = dev;
6    urb->context = uvd;
7    urb->pipe = usb_rcvisocpipe(dev, uvd->video_endp);/*端口*/
8    urb->interval = 1;
9    urb->transfer_flags = URB_ISO_ASAP; /*urb 被调度*/
10   urb->transfer_buffer = uvd->sbuf[i].data;/*传输 buffer*/
11   urb->complete = usbvideo_IsocIrq; /* 完成函数 */
12   urb->number_of_packets = FRAMES_PER_DESC; /*urb 中的等时传输数量*/
13            urb->transfer_buffer_length    =    uvd->iso_packet_len
*FRAMES_PER_DESC;
14   for (j = k = 0; j < FRAMES_PER_DESC; j++, k += uvd->iso_packet_len)
15   {
16     urb->iso_frame_desc[j].offset = k;
17     urb->iso_frame_desc[j].length = uvd->iso_packet_len;
```

```
18   }
19 }
```

（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，通过 usb_submit_urb()函数来完成，如下所示：

```
int usb_submit_urb(struct urb *urb, int 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 完成函数将被调用。

- 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 urb，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。对 usb_unlink_urb()而言，如果 urb 结构体中的 URB_ASYNC_UNLINK（即异步 unlink）的标志被置位，则对该 urb 的 usb_unlink_urb()调用将立即返回，具体的 unlink 动作将在后台进行。否则，此函数一直等到 urb 被解开链接或结束时才返回。usb_kill_urb()会彻底终止 urb 的生命周期，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），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 20.5 给出了一个 urb 的整个处理流程，虚线框的 usb_unlink_urb()和 usb_kill_urb()并非一定会发生，它只是在 urb 正在被 USB 核心和主机控制器处理时，被驱动程序取消的情况下才发生。

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);
```

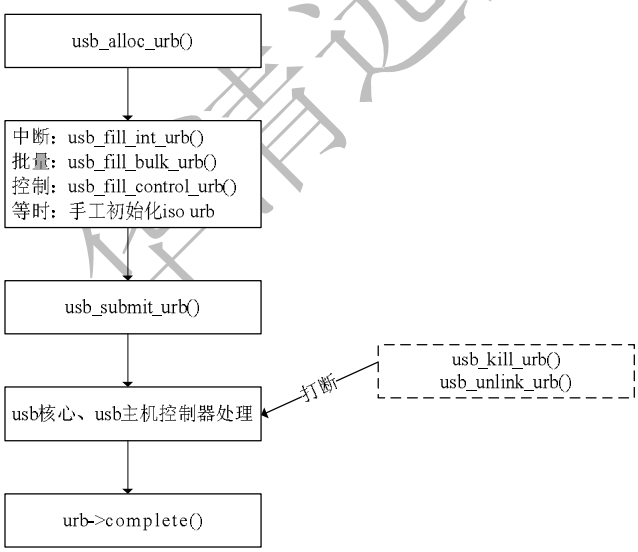

图 20.5　urb 处理流程

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，不过它提供驱动发送和结束 USB 控制信息而非批量信息的能力，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 jiffies 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。

20.3.3 探测和断开函数

在 USB 设备驱动 usb_driver 结构体的探测函数中，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，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 20.15 所示。

代码清单 20.15 usb_class_driver 结构体

```
1 struct usb_class_driver
2 {
3   char *name;  /*sysfs 中用来描述设备名*/
4   struct file_operations *fops;/*文件操作结构体指针*/
5   int minor_base; /*开始次设备号*/
6 };
```

对于字符设备而言，usb_class_driver 结构体的 fops 成员中的 write()、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 结构体的探测函数中，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。

对探测函数的调用发生在 USB 设备被安装且 USB 核心认为该驱动程序与安装的 USB 设备对应时（usb_driver 的 id_table 成员在此时发挥作用），而对断开函数的调用则发生在驱动因为种种原因不再控制该设备的时候。对这两个函数的调用都是在内核线程中进行的，因此，可以在其中进行任何可能引起睡眠的操作。但是，由于 USB 核心对所有探测和断开函数的调用都发生在单一线程内，因此，太慢的 probe()或 disconnect()将拖延其他 USB 设备的探测时间。

20.3.4 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，可被看做一个最简单的 USB 设备驱动实例。尽管具体 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 20.16 所示。

代码清单 20.16 USB 骨架程序的 usb_driver 结构体

```
1 static struct usb_driver skel_driver =
2 {
3   .name =     "skeleton",
4   .probe =    skel_probe,
5   .disconnect =   skel_disconnect,
6   .id_table = skel_table,
7 };
```

从上述代码第 6 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 20.17 所示。

代码清单 20.17 USB 骨架程序的 id_table

```
1  /* 本驱动支持的 USB 设备列表 */
2  static struct usb_device_id skel_table [] = {
3     { USB_DEVICE(USB_SKEL_VENDOR_ID,
4       USB_SKEL_PRODUCT_ID) },
5     { }
6  };
7  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，如代码清单 20.18 所示，其分别调用了 usb_register()和 usb_deregister()。

代码清单 20.18 USB 骨架程序的模块加载与卸载函数

```
1  static int __init usb_skel_init(void)
2  {
3   int result;
4
5   /* 注册 USB 驱动 */
6   result = usb_register(&skel_driver);
7   if (result)
8       err("usb_register failed. Error number %d", result);
9
10  return result;
11 }
```

```
12 static void __exit usb_skel_exit(void)
13 {
14   /* 注销 USB 驱动 */
15   usb_deregister(&skel_driver);
16 }
```

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，并将 usb_skel 实例的指针传入 usb_set_intfdata()作为 USB 接口的私有数据，最后，它会注册 USB 设备，如代码清单 20.19 所示。

代码清单 20.19　USB 骨架程序的探测函数

```
1  static int skel_probe(struct usb_interface *interface, const struct
usb_device_id *id)
2  {
3   struct usb_skel *dev = NULL;
4   struct usb_host_interface *iface_desc;
5   struct usb_endpoint_descriptor *endpoint;
6   size_t buffer_size;
7   int i;
8   int retval = -ENOMEM;
9
10  /* 分配设备状态的内存并初始化 */
11  dev = kzalloc(sizeof(*dev), GFP_KERNEL);
12  if (dev == NULL) {
13      err("Out of memory");
14      goto error;
15  }
16  kref_init(&dev->kref);
17  sema_init(&dev->limit_sem, WRITES_IN_FLIGHT);
18
19  dev->udev = usb_get_dev(interface_to_usbdev(interface));
20  dev->interface = interface;
21
22  /* 设置端点信息 */
23  /* 仅使用第一个 bulk-in 和 bulk-out */
24  iface_desc = interface->cur_altsetting;
25  for (i = 0; i < iface_desc->desc.bNumEndpoints; ++i) {
26      endpoint = &iface_desc->endpoint[i].desc;
27
28      if (!dev->bulk_in_endpointAddr &&
29          ((endpoint->bEndpointAddress & USB_ENDPOINT_DIR_MASK)
```

```
30                  == USB_DIR_IN) &&
31          ((endpoint->bmAttributes & USB_ENDPOINT_XFERTYPE_MASK)
32                  == USB_ENDPOINT_XFER_BULK)) {
33          /* 找到了一个批量 IN 端点 */
34          buffer_size = le16_to_cpu(endpoint->wMaxPacketSize);
35          dev->bulk_in_size = buffer_size;
36          dev->bulk_in_endpointAddr = endpoint->bEndpointAddress;
37          dev->bulk_in_buffer = kmalloc(buffer_size, GFP_KERNEL);
38          if (!dev->bulk_in_buffer) {
39              err("Could not allocate bulk_in_buffer");
40              goto error;
41          }
42      }
43
44      if (!dev->bulk_out_endpointAddr &&
45          ((endpoint->bEndpointAddress & USB_ENDPOINT_DIR_MASK)
46                  == USB_DIR_OUT) &&
47          ((endpoint->bmAttributes & USB_ENDPOINT_XFERTYPE_MASK)
48                  == USB_ENDPOINT_XFER_BULK)) {
49          /* 找到了一个批量 OUT 端点 */
50          dev->bulk_out_endpointAddr = endpoint->bEndpointAddress;
51      }
52  }
53  if (!(dev->bulk_in_endpointAddr && dev->bulk_out_endpointAddr)) {
54      err("Could not find both bulk-in and bulk-out endpoints");
55      goto error;
56  }
57
58  /* 在设备结构中保存数据指针 */
59  usb_set_intfdata(interface, dev);
60
61  /* 注册 USB 设备 */
62  retval = usb_register_dev(interface, &skel_class);
63  if (retval) {
64      /* something prevented us from registering this driver */
65      err("Not able to get a minor for this device.");
66      usb_set_intfdata(interface, NULL);
67      goto error;
```

```
    68  }
    69
    70  /* 告知用户设备依附于什么 node */
    71  info("USB  Skeleton  device  now  attached  to  USBSkel-%d",
interface->minor);
    72  return 0;
    73
    74 error:
    75  if (dev)
    76      kref_put(&dev->kref, skel_delete);
    77  return retval;
    78 }
```

usb_skel 结构体可以被看作一个私有数据结构体，其定义如代码清单 20.20 所示，应该根据具体的设备量身定制。

代码清单 20.20　USB 骨架程序的自定义数据结构 usb_skel

```
    1  struct usb_skel
    2  {
    3   struct usb_device * udev;              /* 该设备的 usb_device 指针
*/
    4   struct usb_interface *  interface;     /* 该设备的 usb_interface 指
针 */
    5   struct semaphore    limit_sem;             /* 限制进程写的数量 */
    6   unsigned char *     bulk_in_buffer;    /* 接收数据的缓冲区 */
    7   size_t          bulk_in_size;              /* 接收缓冲区大小 */
    8   __u8        bulk_in_endpointAddr;          /*批量 IN 端点的地址 */
    9   __u8        bulk_out_endpointAddr;     /* 批量 OUT 端点的地址 */
    10  struct kref     kref;
    11 };
```

USB 骨架程序的断开函数会完成探测函数相反的工作，即设置接口数据为 NULL，注销 USB 设备，如代码清单 20.21 所示。

代码清单 20.21　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   /* 阻止 skel_open()与 skel_disconnect()的竞争 */
    7   lock_kernel();
    8
```

```
9   dev = usb_get_intfdata(interface);
10  usb_set_intfdata(interface, NULL);
11
12  /* 注销 usb 设备，释放次设备号 */
13  usb_deregister_dev(interface, &skel_class);
14
15  unlock_kernel();
16
17  /* 减少引用计数 */
18  kref_put(&dev->kref, skel_delete);
19
20  info("USB Skeleton #%d now disconnected", minor);
21 }
```

代码清单 20.19 第 62 行的 usb_register_dev(interface, &skel_class)中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 20.22 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 20.22　USB 骨架程序的字符设备文件操作结构体

```
1 static struct file_operations skel_fops =
2 {
3   .owner =    THIS_MODULE,
4   .read =     skel_read,  //读函数
5   .write =    skel_write,     //写函数
6   .open =     skel_open,      //打开函数
7   .release =  skel_release, //释放函数
8 };
```

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 20.23 所示。

代码清单 20.23　USB 骨架程序的字符设备打开函数

```
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);

    interface = usb_find_interface(&skel_driver, subminor);
    if (!interface) {
        err ("%s - error, can't find device for minor %d",
             __FUNCTION__, subminor);
        retval = -ENODEV;
        goto exit;
    }

    dev = usb_get_intfdata(interface);/*获得接口数据*/
    if (!dev) {
        retval = -ENODEV;
        goto exit;
    }

    /* 增加设备的引用计数 */
    kref_get(&dev->kref);

    /* 将接口数据保存在 file->private_data 中 */
    file->private_data = dev;

exit:
    return retval;
}
```

由于在 open()函数中并没有申请任何软件和硬件资源，因此与 open()函数对应的 release()函数不用进行资源的释放，它会减少在 open()中增加的引用计数，如代码清单 20.24 所示。

代码清单 20.24　USB 骨架程序的字符设备释放函数

```
static int skel_release(struct inode *inode, struct file *file)
{
    struct usb_skel *dev;

    dev = (struct usb_skel *)file->private_data;
    if (dev == NULL)
        return -ENODEV;

    /* 减少设备的引用计数 */
    kref_put(&dev->kref, skel_delete);
    return 0;
}
```

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，贯穿于其中的“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 20.3.2 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 20.25 所示。

代码清单 20.25　USB 骨架程序的字符设备写函数

```
static ssize_t skel_write(struct file *file, const char *user_buffer, size_t
count, loff_t *ppos)
{
    struct usb_skel *dev;
    int retval = 0;
```

```
    5   struct urb *urb = NULL;
    6   char *buf = NULL;
    7   size_t writesize = min(count, (size_t)MAX_TRANSFER);
    8
    9   dev = (struct usb_skel *)file->private_data;
    10
    11  /* 验证实际上有数据要写入 USB 设备 */
    12  if (count == 0)
    13      goto exit;
    14
    15  /* 限制 urb 的数量，防止一个用户用完所有的 RAM */
    16  if (down_interruptible(&dev->limit_sem)) {
    17      retval = -ERESTARTSYS;
    18      goto exit;
    19  }
    20
    21  /* 创建 urb、urb 的缓冲区，将数据复制给 urb */
    22  urb = usb_alloc_urb(0, GFP_KERNEL);
    23  if (!urb) {
    24      retval = -ENOMEM;
    25      goto error;
    26  }
    27
    28  buf   =   usb_buffer_alloc(dev->udev,   writesize,   GFP_KERNEL,
&urb->transfer_dma);
    29  if (!buf) {
    30      retval = -ENOMEM;
    31      goto error;
    32  }
    33
    34  if (copy_from_user(buf, user_buffer, writesize)) {
    35      retval = -EFAULT;
    36      goto error;
    37  }
    38
    39  /* 恰当地初始化 urb */
    40  usb_fill_bulk_urb(urb, dev->udev,
    41            usb_sndbulkpipe(dev->udev, dev->bulk_out_endpointAddr),
```

```
    42           buf, writesize, skel_write_bulk_callback, dev);
    43  urb->transfer_flags |= URB_NO_TRANSFER_DMA_MAP;
    44
    45  /* 将数据发送到批量端口 */
    46  retval = usb_submit_urb(urb, GFP_KERNEL);
    47  if (retval) {
    48      err("%s - failed submitting write urb, error %d", __FUNCTION__, retval);
    49      goto error;
    50  }
    51
    52  /* 释放对 urb 的引用 */
    53  usb_free_urb(urb);
    54
    55 exit:
    56  return writesize;
    57
    58 error:
    59  usb_buffer_free(dev->udev, writesize, buf, urb->transfer_dma);
    60  usb_free_urb(urb);
    61  up(&dev->limit_sem);
    62  return retval;
    63 }
```

写函数中发起的 urb 结束后，其完成函数 skel_write_bulk_callback()将被调用，它会进行 urb->status 的判断，如代码清单 20.26 所示。

代码清单 20.26　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, struct pt_regs *regs)
    2  {
    3   struct usb_skel *dev;
    4
    5   dev = (struct usb_skel *)urb->context;
    6
    7   /* sync/async 去除链路故障，不是错误 */
    8   if (urb->status &&
    9      !(urb->status == -ENOENT ||
    10       urb->status == -ECONNRESET ||
    11       urb->status == -ESHUTDOWN)) {
    12      dbg("%s - nonzero write bulk status received: %d",
    13          __FUNCTION__, urb->status);
    14  }
    15
    16  /* 释放被分配的内存 */
```

```
    usb_buffer_free(urb->dev, urb->transfer_buffer_length,
            urb->transfer_buffer, urb->transfer_dma);
    up(&dev->limit_sem);
}
```

USB 骨架程序的字符设备读函数并没有进行类似写函数的一系列针对 urb 的操作，而是简单地调用 usb_bulk_msg()发起一次同步 urb 传输操作，如代码清单 20.27 所示。

代码清单 20.27　USB 骨架程序的字符设备读函数

```
static ssize_t skel_read(struct file *file, char *buffer, size_t
count, loff_t *ppos)
{
    struct usb_skel *dev;
    int retval = 0;
    int bytes_read;

    dev = (struct usb_skel *)file->private_data;

    /* 从设备进行一次阻塞的批量读 */
    retval = usb_bulk_msg(dev->udev,
                usb_rcvbulkpipe(dev->udev,
dev->bulk_in_endpointAddr),
                dev->bulk_in_buffer,
                min(dev->bulk_in_size, count),
                &bytes_read, 10000);

    /* 如果读成功，将数据复制到用户空间 */
    if (!retval) {
        if (copy_to_user(buffer, dev->bulk_in_buffer, bytes_read))
            retval = -EFAULT;
        else
            retval = bytes_read;
    }

    return retval;
}
```

20.4 USB 设备驱动实例

20.4.1　USB 串口驱动

在 Linux 内核中，串口属于 tty 设备，对于一个 USB 串口设备而言，其驱动主要由两部分组成：usb_driver 的成员函数和 tty 设备的 tty_operations 结构体成员函数。

在 USB 串口设备驱动的模块加载函数中，将注册对应于 USB 串口的 usb_driver，

并初始化和注册 tty 驱动，如代码清单 20.28 所示。

代码清单 20.28　USB 串口设备驱动的模块加载函数

```
1  static int __init usb_serial_init(void)
2  {
3    int i;
4    int result;
5
6    /* 分配 tty_driver */
7    usb_serial_tty_driver = alloc_tty_driver(SERIAL_TTY_MINORS);
8    if (!usb_serial_tty_driver)
9      return  - ENOMEM;
10
11   /* 初始化全局数据 */
12   for (i = 0; i < SERIAL_TTY_MINORS; ++i)
13   {
14     serial_table[i] = NULL;
15   }
16
17   /* 注册总线 */
18   result = bus_register(&usb_serial_bus_type);
19   if (result)
20   {
21     err("%s - registering bus driver failed", __FUNCTION__);
22     goto exit_bus;
23   }
24
25   /* 初始化 tty_driver */
26   usb_serial_tty_driver->owner = THIS_MODULE;
27   usb_serial_tty_driver->driver_name = "usbserial";
28   usb_serial_tty_driver->devfs_name = "usb/tts/";
29   usb_serial_tty_driver->name = "ttyUSB";
30   usb_serial_tty_driver->major = SERIAL_TTY_MAJOR;
31   usb_serial_tty_driver->minor_start = 0;
32   usb_serial_tty_driver->type = TTY_DRIVER_TYPE_SERIAL;
33   usb_serial_tty_driver->subtype = SERIAL_TYPE_NORMAL;
34        usb_serial_tty_driver->flags   =   TTY_DRIVER_REAL_RAW   |
TTY_DRIVER_NO_DEVFS;
35   usb_serial_tty_driver->init_termios = tty_std_termios;
36   usb_serial_tty_driver->init_termios.c_cflag = B9600 | CS8 | CREAD
| HUPCL |
```

```
    CLOCAL;
  tty_set_operations(usb_serial_tty_driver, &serial_ops);
  /* 注册 tty_driver */
  result = tty_register_driver(usb_serial_tty_driver);
  if (result)
  {
    err("%s - tty_register_driver failed", __FUNCTION__);
    goto exit_reg_driver;
  }

  /* 注册 USB 驱动 */
  result = usb_register(&usb_serial_driver);
  if (result < 0)
  {
    err("%s - usb_register failed", __FUNCTION__);
    goto exit_tty;
  }

  result = usb_serial_generic_register(debug);
  if (result < 0)
  {
    err("%s - registering generic driver failed", __FUNCTION__);
    goto exit_generic;
  }

  info(DRIVER_DESC);

  return result;

  exit_generic: usb_deregister(&usb_serial_driver);
  exit_tty: tty_unregister_driver(usb_serial_tty_driver);
  exit_reg_driver: bus_unregister(&usb_serial_bus_type);
   exit_bus: err("%s - returning with error %d", __FUNCTION__,
result);
  put_tty_driver(usb_serial_tty_driver);
  return result;
}
```

在 USB 串口设备驱动的模块卸载函数中，将注销对应于 USB 串口的 usb_driver，并注销 tty 驱动，如代码清单 20.29 所示。

代码清单 20.29　USB 串口设备驱动的模块卸载函数

```
static void __exit usb_serial_exit(void)
{
  usb_serial_console_exit();
  usb_serial_generic_deregister();
  usb_deregister(&usb_serial_driver);//注销 usb_driver
  tty_unregister_driver(usb_serial_tty_driver);//注销 tty_driver
  put_tty_driver(usb_serial_tty_driver);//减少引用计数
```

```
8   bus_unregister(&usb_serial_bus_type);//注销 bus
9 }
```

在 usb_driver 的探测成员函数 usb_serial_probe()中，将初始化 USB 端点等信息，并通过 usb_set_intfdata()设置接口私有数据，它也将初始化 urb。相反地，在断开成员函数 usb_serial_disconnect()中将设置接口私有数据为 NULL，并释放引用计数。

USB 串口驱动的 tty_operations 结构体实例 serial_ops 定义如代码清单 20.30 所示，它封装了 USB 串口设备驱动中的串口驱动成分。

代码清单 20.30　USB 串口驱动的 tty_operations 结构体

```
1  static struct tty_operations serial_ops =
2  {
3    .open =         serial_open,
4    .close =        serial_close,
5    .write =        serial_write,
6    .write_room =       serial_write_room,
7    .ioctl =        serial_ioctl,
8    .set_termios =      serial_set_termios,
9    .throttle =     serial_throttle,
10   .unthrottle =       serial_unthrottle,
11   .break_ctl =        serial_break,
12   .chars_in_buffer =  serial_chars_in_buffer,
13   .read_proc =        serial_read_proc,
14   .tiocmget =     serial_tiocmget,
15   .tiocmset =     serial_tiocmset,
16 };
```

在 tty_operations 的各 write()、read()等成员函数中，将调用 usb_serial_driver 结构体中的相应函数，usb_serial_driver 结构体中封装了串口的各函数（读写、读写中断端点完成函数、读写批量端点完成函数等），其定义如代码清单 20.31 所示。

代码清单 20.31　usb_serial_driver 结构体

```
1  struct usb_serial_driver
2  {
3    const char *description; //用于描述该驱动的字符串
4    const struct usb_device_id *id_table; //usb_device_id 数组
5    char num_interrupt_in; //中断输入端点数量
6    char num_interrupt_out; //中断输出端点数量
7    char num_bulk_in; //批量输入端点数量
8    char num_bulk_out; //批量输出端点数量
9    char num_ports; //设备包含的端口数量
10
11   struct list_head driver_list;
```

```
    12   struct device_driver driver;
    13
    14   int(*probe)(struct usb_serial *serial, const struct usb_device_id
*id);
    15   int(*attach)(struct usb_serial *serial);
    16   int(*calc_num_ports)(struct usb_serial *serial);
    17
    18   void(*shutdown)(struct usb_serial *serial);
    19
    20   int(*port_probe)(struct usb_serial_port *port);
    21   int(*port_remove)(struct usb_serial_port *port);
    22
    23   /* 串口函数 */
    24   int(*open)(struct usb_serial_port *port, struct file *filp);
    25   void(*close)(struct usb_serial_port *port, struct file *filp);
    26   int(*write)(struct usb_serial_port *port, const unsigned char *buf,
int count)
    27    ;
    28   int(*write_room)(struct usb_serial_port *port);
    29    int(*ioctl)(struct usb_serial_port *port, struct file *file,
unsigned int cmd,
    30    unsigned long arg);
    31   void(*set_termios)(struct usb_serial_port *port, struct termios
*old);
    32   void(*break_ctl)(struct usb_serial_port *port, int break_state);
    33   int(*chars_in_buffer)(struct usb_serial_port *port);
    34   void(*throttle)(struct usb_serial_port *port);
    35   void(*unthrottle)(struct usb_serial_port *port);
    36   int(*tiocmget)(struct usb_serial_port *port, struct file *file);
    37   int(*tiocmset)(struct usb_serial_port *port, struct file *file,
unsigned int
    38    set, unsigned int clear);
    39   /* urb 完成回调函数 */
    40   void(*read_int_callback)(struct urb *urb, struct pt_regs *regs);
    41   void(*write_int_callback)(struct urb *urb, struct pt_regs *regs);
    42   void(*read_bulk_callback)(struct urb *urb, struct pt_regs *regs);
    43   void(*write_bulk_callback)(struct urb *urb, struct pt_regs *regs);
    44 };
```

文件 drivers/usb/serial/Generic.c 文件中提供了 USB 串口驱动的通用打开/关闭、写函数、批量 urb 完成函数，如 usb_serial_generic_write()、usb_serial_generic_write_bulk_callback()、usb_serial_generic_read_bulk_callback()等。代码清单 20.32 列举了通用写函数及输出批量 urb 完成回调函数。

代码清单 20.32 USB 串口通用写函数及批量写 urb 完成函数

```
1  int usb_serial_generic_write(struct usb_serial_port *port, const unsigned char
2    *buf, int count)
3  {
4    struct usb_serial *serial = port->serial;
5    int result;
6    unsigned char *data;
7
8    if (count == 0)
9    {
10     dbg("%s - write request of 0 bytes", __FUNCTION__);
11     return (0);
12   }
13
14   /* 如果有批量输出端点 */
15   if (serial->num_bulk_out)
16   {
17     spin_lock(&port->lock);
18     if (port->write_urb_busy)
19     {
20       spin_unlock(&port->lock);
21       dbg("%s - already writing", __FUNCTION__);
22       return 0;
23     }
24     port->write_urb_busy = 1;
25     spin_unlock(&port->lock);
26
27      count = (count > port->bulk_out_size) ? port->bulk_out_size: count;
28
29     memcpy(port->write_urb->transfer_buffer, buf, count);
30     data = port->write_urb->transfer_buffer;
31     usb_serial_debug_data(debug, &port->dev, __FUNCTION__, count, data);
32
33     /* 设置urb */
34                 usb_fill_bulk_urb(port->write_urb,   serial->dev, usb_sndbulkpipe(serial->dev,
35                                   port->bulk_out_endpointAddress), port->write_urb->transfer_buffer, count,
36                         ((serial->type->write_bulk_callback)   ?
```

```
serial->type->write_bulk_callback:
    37      usb_serial_generic_write_bulk_callback), port);
    38
    39    /* 将数据送出给批量端口 */
    40    port->write_urb_busy = 1;
    41    result = usb_submit_urb(port->write_urb, GFP_ATOMIC);
    42    if (result)
    43    {
    44      port->write_urb_busy = 0;
    45    }
    46    else
    47      result = count;
    48
    49    return result;
    50  }
    51
    52  return 0;
    53 }
    54
    55 void usb_serial_generic_write_bulk_callback(struct urb *urb, struct
pt_regs
    56  *regs)
    57 {
    58              struct     usb_serial_port     *port     =     (struct
usb_serial_port*)urb->context;
    59
    60  port->write_urb_busy = 0;
    61  if (urb->status)  //不成功
    62  {
    63    dbg("%s - nonzero write bulk status received: %d", __FUNCTION_
_, urb
    64      ->status);
    65    return ;
    66  }
    67
    68  usb_serial_port_softint((void*)port);//tty_wakeup
    69  schedule_work(&port->work);//调度底半部
    70 }
```

20.4.2 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 的成员函数和输入设备的打开、关闭、中断处理等函数。

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 20.33 所示为模块加载与卸载函数以及 usb_kbd_driver 结构体的定义。

代码清单 20.33　USB 键盘设备驱动的模块加载与卸载函数及 usb_driver 结构体

```
1  static int __init usb_kbd_init(void)
2  {
3    int result = usb_register(&usb_kbd_driver); //注册 USB 设备驱动
4    if (result == 0)
5      info(DRIVER_VERSION ":" DRIVER_DESC);
6    return result;
7  }
8
9  static void __exit usb_kbd_exit(void)
10 {
11   usb_deregister(&usb_kbd_driver); //注销 USB 设备驱动
12 }
13
14 //usb_driver 结构体
15 static struct usb_driver usb_kbd_driver =
16 {
17   .name = "usbkbd",
18   .probe = usb_kbd_probe,
19   .disconnect = usb_kbd_disconnect,
20   .id_table = usb_kbd_id_table,
21 };
22
23 //支持的设备列表
24 static struct usb_device_id usb_kbd_id_table [] = {
25  { USB_INTERFACE_INFO(3, 1, 1) },
26  { }
27 };
28 MODULE_DEVICE_TABLE (usb, usb_kbd_id_table);
```

在 usb_driver 的探测函数中，将进行 input 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 20.34 所示。

代码清单 20.34　USB 键盘设备驱动的探测函数

```
1  static int usb_kbd_probe(struct usb_interface *iface, const struct
2    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    int i, pipe, maxp;
```

```
10
11  interface = iface->cur_altsetting;
12  /* 设备是否适合本驱动? */
13  if (interface->desc.bNumEndpoints != 1)
14    return  - ENODEV;
15
16  endpoint = &interface->endpoint[0].desc;
17  if (!(endpoint->bEndpointAddress &USB_DIR_IN))
18    return  - ENODEV;
19  if ((endpoint->bmAttributes &USB_ENDPOINT_XFERTYPE_MASK) !=
20    USB_ENDPOINT_XFER_INT)
21    return  - ENODEV;
22
23  pipe = usb_rcvintpipe(dev, endpoint->bEndpointAddress);//创建端点
的管道
24  maxp = usb_maxpacket(dev, pipe, usb_pipeout(pipe));
25
26  kbd = kzalloc(sizeof(struct usb_kbd), GFP_KERNEL);
27  input_dev = input_allocate_device();//分配 input_dev 结构体
28  if (!kbd || !input_dev)             //分配
29    goto fail1;
30
31  if (usb_kbd_alloc_mem(dev, kbd))
32    goto fail2;
33
34  kbd->usbdev = dev;
35  kbd->dev = input_dev;
36
37  if (dev->manufacturer)//制造商名非空
38    strlcpy(kbd->name, dev->manufacturer, sizeof(kbd->name));
39
40  if (dev->product)//产品名非空
41  {
42    if (dev->manufacturer)
43      strlcat(kbd->name, " ", sizeof(kbd->name));
44    strlcat(kbd->name, dev->product, sizeof(kbd->name));
45  }
46
47  if (!strlen(kbd->name))
48    snprintf(kbd->name, sizeof(kbd->name), "USB HIDBP Keyboard
49                %04x:%04x",le16_to_cpu(dev->descriptor.idVendor),
le16_to_cpu(dev
50      ->descriptor.idProduct));
51
52  usb_make_path(dev, kbd->phys, sizeof(kbd->phys));
53  strlcpy(kbd->phys, "/input0", sizeof(kbd->phys));
54  /* 输入设备初始化 */
55  input_dev->name = kbd->name;
```

```
    input_dev->phys = kbd->phys;
    usb_to_input_id(dev, &input_dev->id);
    input_dev->cdev.dev = &iface->dev;
     input_dev->private = kbd;

    input_dev->evbit[0] = BIT(EV_KEY) | BIT(EV_LED) | BIT(EV_REP);
    input_dev->ledbit[0] = BIT(LED_NUML) | BIT(LED_CAPSL) |
     BIT(LED_SCROLLL) |BIT(LED_COMPOSE) | BIT(LED_KANA);

    for (i = 0; i < 255; i++)
     set_bit(usb_kbd_keycode[i], input_dev->keybit);
    clear_bit(0, input_dev->keybit);

    input_dev->event = usb_kbd_event;
    input_dev->open = usb_kbd_open;
    input_dev->close = usb_kbd_close;
    /* 初始化中断 urb */
    usb_fill_int_urb(kbd->irq, dev, pipe, kbd->new, (maxp > 8 ? 8 :
maxp),
     usb_kbd_irq, kbd, endpoint->bInterval);
    kbd->irq->transfer_dma = kbd->new_dma;
    kbd->irq->transfer_flags |= URB_NO_TRANSFER_DMA_MAP;

    kbd->cr->bRequestType = USB_TYPE_CLASS | USB_RECIP_INTERFACE;
    kbd->cr->bRequest = 0x09;
    kbd->cr->wValue = cpu_to_le16(0x200);
    kbd->cr->wIndex = cpu_to_le16(interface->desc.bInterfaceNumber);
    kbd->cr->wLength = cpu_to_le16(1);
    /* 初始化控制 urb */
     usb_fill_control_urb(kbd->led, dev, usb_sndctrlpipe(dev, 0),
(void*)kbd->cr,
     kbd->leds, 1, usb_kbd_led, kbd);
    kbd->led->setup_dma = kbd->cr_dma;
    kbd->led->transfer_dma = kbd->leds_dma;
    kbd->led->transfer_flags |= (URB_NO_TRANSFER_DMA_MAP |
     URB_NO_SETUP_DMA_MAP);
    input_register_device(kbd->dev); //注册输入设备

    usb_set_intfdata(iface, kbd);  //设置接口私有数据
    return 0;

    fail2: usb_kbd_free_mem(dev, kbd);
    fail1: input_free_device(input_dev);
    kfree(kbd);
    return  - ENOMEM;
 }
```

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 20.35 所示。

代码清单 20.35　USB 键盘设备驱动的断开函数

```
 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);//设置接口私有数据为 NULL
6   if (kbd)
7   {
8     usb_kill_urb(kbd->irq);//终止 urb
9     input_unregister_device(kbd->dev); //注销输入设备
10    usb_kbd_free_mem(interface_to_usbdev(intf), kbd);
11    kfree(kbd);
12  }
13 }
```

在键盘中断处理函数，也就是中断 urb 的完成函数中，将会通过 input_report_key()报告按键事件，通过 input_sync()报告同步事件，并发起一次新的控制 urb 传输，如代码清单 20.36 所示。

代码清单 20.36　USB 键盘设备驱动的中断 urb 完成函数

```
1  static void usb_kbd_irq(struct urb *urb, struct pt_regs *regs)
2  {
3    struct usb_kbd *kbd = urb->context;
4    int i;
5
6    switch (urb->status)
7    {
8      case 0:  /* 成功 */
9        break;
10       case  - ECONNRESET:  /* unlink */
11       case  - ENOENT:
12       case  - ESHUTDOWN:
13       return ;
14     default: /* 错误 */
15       goto resubmit;
16   }
17 /*获得键盘扫描码并报告按键事件*/
18   input_regs(kbd->dev, regs);
19
20   for (i = 0; i < 8; i++)
21           input_report_key(kbd->dev,  usb_kbd_keycode[i  +  224],
(kbd->new[0] >> i) &1);
22
23   for (i = 2; i < 8; i++)
24   {
25     if (kbd->old[i] > 3 && memscan(kbd->new + 2, kbd->old[i], 6) ==
26       kbd->new +8)
27     {
```

```
    28      if (usb_kbd_keycode[kbd->old[i]])
    29          input_report_key(kbd->dev, usb_kbd_keycode[kbd->old[i]],
0);
    30      else
    31        info("Unknown key (scancode %#x) released.", kbd->old[i]);
    32    }
    33
    34    if (kbd->new[i] > 3 && memscan(kbd->old + 2, kbd->new[i], 6) ==
    35      kbd->old +8)
    36    {
    37      if (usb_kbd_keycode[kbd->new[i]])
    38          input_report_key(kbd->dev, usb_kbd_keycode[kbd->new[i]],
1);
    39      else
    40        info("Unknown key (scancode %#x) pressed.", kbd->new[i]);
    41    }
    42  }
    43
    44  input_sync(kbd->dev);报告同步事件
    45
    46  memcpy(kbd->old, kbd->new, 8);
    47
    48  resubmit: i = usb_submit_urb(urb, SLAB_ATOMIC);
    49  if (i)
    50          err("can't  resubmit  intr,  %s-%s/input0,  status  %d",
kbd->usbdev->bus
    51      ->bus_name, kbd->usbdev->devpath, i);
    52 }
```

20.5 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，这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，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 主机传递及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。此外，简单的控制及批量消息传递可以用同步的 usb_bulk_msg()、usb_control_msg()函数完成。