# 第十三章 UAB驱动

通用串行总线（USB）是主机计算机与多个外设设备之间的连接。它最初是为了用一种所有设备都可以连接的单一总线类型来替换一系列慢速且不同的总线——并行、串行和键盘连接。USB已经超越了这些慢速连接，现在支持几乎所有可以连接到PC的设备类型。USB规范的最新修订增加了高速连接，理论速度限制为480 MBps。

在拓扑结构上，USB子系统并不是作为总线布局的；它更像是由几个点对点链接构建的树。这些链接是四线电缆（地线、电源和两个信号线），它们连接设备和集线器，就像双绞线以太网一样。USB主控制器负责询问每个USB设备是否有任何数据要发送。由于这种拓扑结构，USB设备在没有首先被主控制器询问的情况下，永远不能开始发送数据。这种配置允许非常容易的即插即用类型的系统，设备可以由主机计算机自动配置。

在技术层面，总线非常简单，因为它是一种单主机实现，其中主机计算机轮询各种外设设备。尽管有这种内在的限制，但总线有一些有趣的特性，例如设备可以请求固定的带宽用于其数据传输，以便可靠地支持视频和音频I/O。USB的另一个重要特性是，它仅作为设备和主机之间的通信通道，无需对其传送的数据具有特定的含义或结构。

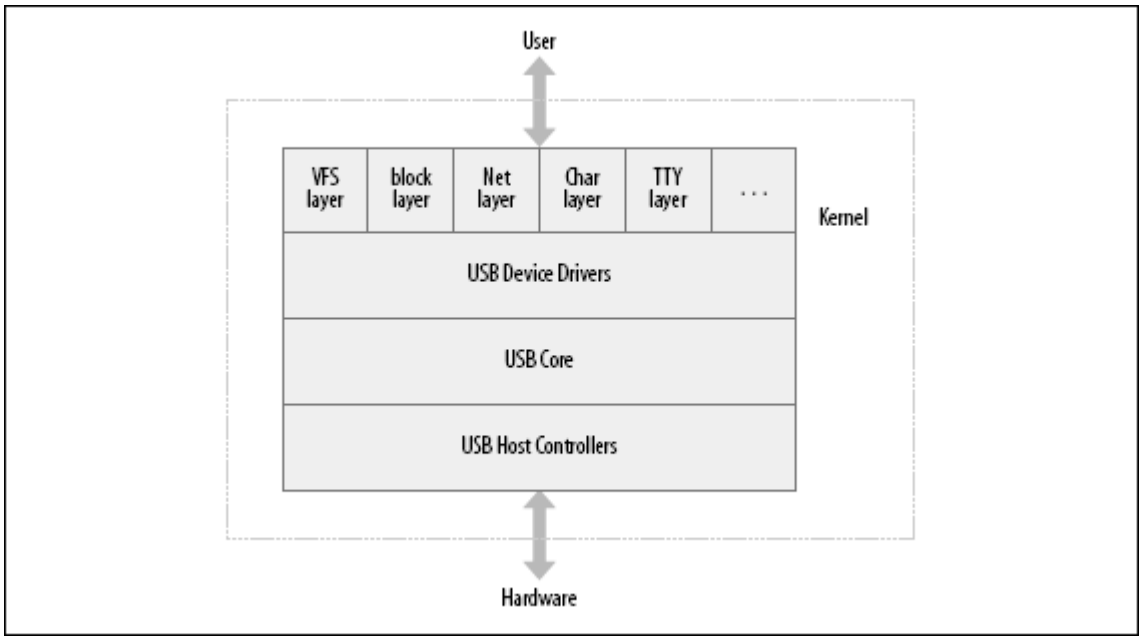
USB协议规范定义了一套任何特定类型的设备都可以遵循的标准。如果设备遵循该标准，那么就不需要该设备的特殊驱动程序。这些不同的类型被称为类别，包括存储设备、键盘、鼠标、操纵杆、网络设备和调制解调器等。

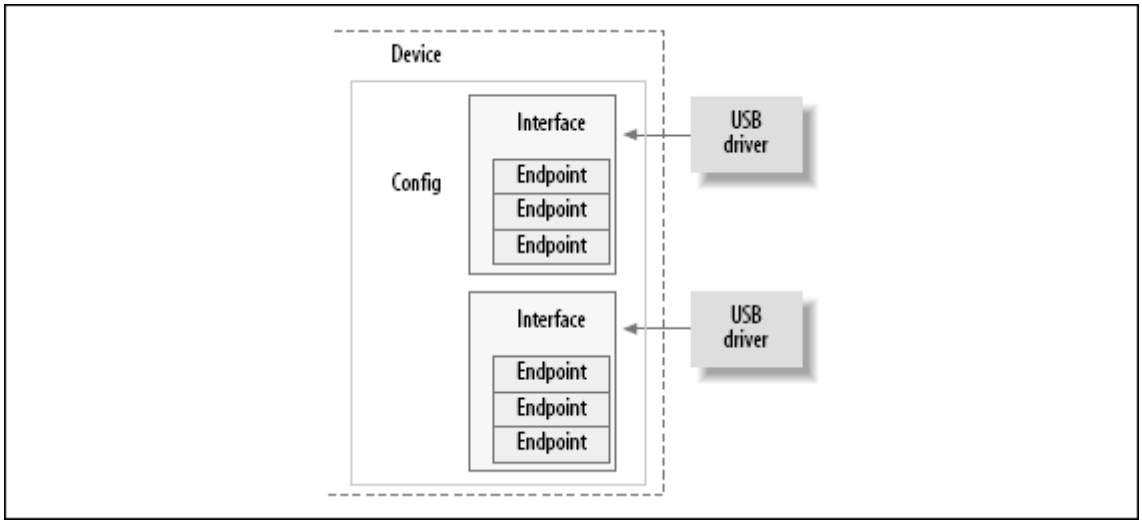
其他不适合这些类别的设备类型需要为该特定设备编写特殊的供应商特定驱动程序。视频设备和USB到串行设备是一个很好的例子，其中没有定义的标准，每个不同的设备都需要一个驱动程序。

这些特性，加上设计的内在热插拔能力，使USB成为一种方便、低成本的机制，可以连接（和断开）多个设备到计算机，无需关闭系统、打开盖子，也无需担心螺丝和线。

Linux内核支持两种主要类型的USB驱动程序：主机系统上的驱动程序和设备上的驱动程序。主机系统的USB驱动程序从主机的角度控制插入其中的USB设备（常见的USB主机是台式计算机）。设备中的USB驱动程序，控制单个设备如何作为USB设备呈现给主机计算机。由于“USB设备驱动程序”这个术语非常混乱，USB开发人员创建了“USB gadget驱动程序”这个术语来描述控制连接到计算机的USB设备的驱动程序（请记住，Linux也运行在那些微型嵌入式设备中）。本章详细介绍了在台式计算机上运行的USB系统的工作方式。在此时，USB gadget驱动程序超出了本书的范围。

如图13-1所示，USB驱动程序位于不同的内核子系统（块、网络、字符等）和USB硬件控制器之间。USB核心为USB驱动程序提供了一个接口，用于访问和控制USB硬件，而无需担心系统上存在的不同类型的USB硬件控制器。




## 13.1. USB 设备基础知识

USB设备是一种非常复杂的东西，如官方USB文档（可在 http://www.usb.org (vscode-file://vscode-app/d:/Microsoft%20VS%20Code/resources/app/out/vs/code/electron-sandbox/workbench/workbench.html) 获取）中所述。幸运的是，Linux内核提供了一个名为USB核心的子系统来处理大部分的复杂性。本章描述了驱动程序与USB核心之间的交互。图13-2显示了USB设备是如何由配置、接口和端点组成的，以及USB驱动程序如何绑定到USB接口，而不是整个USB设备。

### 13.1.1. 端点

USB通信的最基本形式是通过所谓的端点（endpoint）。USB端点只能在一个方向上传输数据，要么从主机计算机到设备（称为OUT端点），要么从设备到主机计算机（称为IN端点）。端点可以被认为是单向管道。

USB端点可以是四种不同类型之一，这些类型描述了数据是如何传输的：

- 控制（CONTROL） 控制端点用于允许访问USB设备的不同部分。它们通常用于配置设备，检索有关设备的信息，向设备发送命令，或检索有关设备的状态报告。这些端点通常较小。每个USB设备都有一个名为“端点0”的控制端点，USB核心在插入时用它来配置设备。USB协议保证这些传输总是有足够的保留带宽来到达设备。

- 中断（INTERRUPT） 中断端点每次USB主机请求设备数据时以固定速率传输少量数据。这些端点是USB键盘和鼠标的主要传输方法。它们也常用于向USB设备发送数据以控制设备，但通常不用于传输大量数据。USB协议保证这些传输总是有足够的保留带宽来通过。

- 批量（BULK） 批量端点传输大量数据。这些端点通常比中断端点大得多（它们一次可以容纳更多的字符）。对于需要传输必须无数据丢失的任何数据的设备，这是常见的。这些传输不保证总是在特定的时间内通过。如果总线上没有足够的空间发送整个BULK包，它会在多个传输中分割到设备或从设备。这些端点在打印机、存储和网络设备上很常见。

- 等时（ISOCHRONOUS） 等时端点也传输大量数据，但数据并不总是保证能通过。这些端点用于可以处理数据丢失的设备，并更依赖于保持数据的持续流动。实时数据收集，如音频和视频设备，几乎总是使用这些端点。

控制和批量端点用于异步数据传输，即驱动程序决定何时使用它们。中断和等时端点是周期性的。这意味着这些端点被设置为在固定时间连续传输数据，这导致它们的带宽被USB核心预留。

USB端点在内核中使用结构体`struct usb_host_endpoint`进行描述。这个结构体包含了另一个叫做`struct usb_endpoint_descriptor`的结构体中的实际端点信息。后者的结构体包含了所有USB特定的数据，格式与设备自身指定的完全一致。驱动程序关心的这个结构体的字段有：

- bEndpointAddress 这是这个特定端点的USB地址。这个8位值中还包含了端点的方向。可以对这个字段使用位掩码`USB_DIR_OUT`和`USB_DIR_IN`来确定这个端点的数据是指向设备还是主机。

- bmAttributes 这是端点的类型。应该对这个值使用位掩码 `USB_ENDPOINT_XFERTYPE_MASK`，以确定端点是类型 `USB_ENDPOINT_XFER_ISOC`，`USB_ENDPOINT_XFER_BULK`，还是类型 `USB_ENDPOINT_XFER_INT`。这些宏分别定义了等时、批量和中断端点。

- wMaxPacketSize 这是这个端点一次可以处理的最大字节大小。注意，驱动程序有可能向一个端点发送大于这个值的数据量，但实际传输到设备时，数据会被分割成 `wMaxPacketSize` 大小的块。对于高速设备，这个字段可以通过在值的上部使用一些额外的位来支持端点的高带宽模式。关于如何做到这一点的更多细节，请参阅USB规范。

- bInterval 如果这个端点是中断类型，这个值是端点的间隔设置，即端点的中断请求之间的时间。该值以毫秒表示。

这个结构体的字段没有“传统”的Linux内核命名方案。这是因为这些字段直接对应USB规范中的字段名。USB内核程序员认为，使用规定的名称以减少阅读规范时的混淆，比让变量名看起来对Linux程序员来说更熟悉更重要。

### 13.1.2. 接口

USB端点被打包成接口（interfaces）。USB接口只处理一种类型的USB逻辑连接，如鼠标、键盘或音频流。一些USB设备有多个接口，如一个USB扬声器可能由两个接口组成：一个USB键盘用于按钮，一个USB音频流。因为USB接口代表基本功能，每个USB驱动程序控制一个接口；所以，对于扬声器的例子，Linux需要为一个硬件设备提供两个不同的驱动程序。

USB接口可能有备选设置（alternate settings），这些是接口参数的不同选择。接口的初始状态在第一个设置中，编号为0。备选设置可以用来以不同的方式控制单个端点，如为设备预留不同数量的USB带宽。每个具有等时端点的设备都使用相同接口的备选设置。

USB接口在内核中使用 `struct usb_interface` 结构体进行描述。这个结构体是USB核心传递给USB驱动程序的，是USB驱动程序然后负责控制的。这个结构体中的重要字段有：

- `struct usb_host_interface *altsetting` 一个接口结构体的数组，包含了可能为此接口选择的所有备选设置。每个 `struct usb_host_interface` 由一组由上述的 `struct usb_host_endpoint` 结构体定义的端点配置组成。注意，这些接口结构体没有特定的顺序。

- `unsigned num_altsetting` 由altsetting指针指向的备选设置的数量。

- `struct usb_host_interface *cur_altsetting` 指向数组altsetting的指针，表示此接口当前活动的设置。

- `int minor` 如果绑定到此接口的USB驱动程序使用USB主要编号，此变量包含USB核心分配给接口的次要编号。只有在成功调用usb_register_dev（在本章后面描述）后，这个才有效。

`struct usb_interface` 结构体中还有其他字段，但USB驱动程序不需要知道它们。

### 13.1.3. 配置

USB接口本身被打包成配置（configurations）。一个USB设备可以有多个配置，并可能在它们之间切换以改变设备的状态。例如，一些允许固件下载到它们的设备包含多个配置来完成这个任务。一次只能启用一个配置。Linux对多配置USB设备的处理并不是很好，但幸运的是，这种设备很少。

Linux使用 `struct usb_host_config` 结构体描述USB配置，并使用 `struct usb_device` 结构体描述整个USB设备。USB设备驱动程序通常不需要读取或写入这些结构体中的任何值，所以这里没有详细定义它们。好奇的读者可以在内核源代码树的文件include/linux/usb.h中找到它们的描述。

USB设备驱动程序通常需要将给定的 `struct usb_interface` 结构体中的数据转换为USB核心需要的 `struct usb_device` 结构体，这对于大范围的函数调用都是必需的。为此，提供了函数 `interface_to_usbdev`。希望在未来，所有当前需要 `struct usb_device` 的USB调用都将被转换为接受 `struct usb_interface` 参数，并且不需要驱动程序进行转换。

所以总结一下，USB设备非常复杂，由许多不同的逻辑单元组成。这些单元之间的关系可以简单地描述如下：

- 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。

## 13.2. USB 和 sysfs

由于单个USB物理设备的复杂性，该设备在sysfs中的表示也相当复杂。物理USB设备（由`struct usb_device`表示）和单个USB接口（由`struct usb_interface`表示）在sysfs中都显示为单个设备。（这是因为这两种结构都包含一个`struct device`结构。）例如，对于只包含一个USB接口的简单USB鼠标，该设备的sysfs目录树如下：

```
/sys/devices/pci0000:00/0000:00:09.0/usb2/2-1
|-- 2-1:1.0
|   |-- bAlternateSetting
|   |-- bInterfaceClass
|   |-- bInterfaceNumber
|   |-- bInterfaceProtocol
|   |-- bInterfaceSubClass
|   |-- bNumEndpoints
|   |-- detach_state
|   |-- iInterface
|   `-- power
|   `-- state
|-- bConfigurationValue
|-- bDeviceClass
|-- bDeviceProtocol
|-- bDeviceSubClass
|-- bMaxPower
|-- bNumConfigurations
|-- bNumInterfaces
|-- bcdDevice
|-- bmAttributes
|-- detach_state
|-- devnum
|-- idProduct
|-- idVendor
|-- maxchild
|-- power
|  `-- state
|-- speed
`-- version
```

结构 usb_device 在树中被表示在:

```
/sys/devices/pci0000:00/0000:00:09.0/usb2/2-1
```

而USB 鼠标设备驱动被绑定到的接口位于目录:

```
/sys/devices/pci0000:00/0000:00:09.0/usb2/2-1/2-1:1.0
```

- 在sysfs中，每个设备都有一组属性，这些属性可以通过读取和（在某些情况下）写入相应的文件来访问。例如，USB设备的`bDeviceClass`、`bDeviceSubClass`和`bDeviceProtocol`字段可以在sysfs中找到。对于USB接口，`bInterfaceClass`、`bInterfaceSubClass`和`bInterfaceProtocol`字段可以在sysfs中找到。

为了帮助读者理解这些长设备路径的含义,我们给出了内核是如何label这些USB设备的表述.

第一个USB设备是一个根集线器（root hub）。这是USB控制器，通常包含在PCI设备中。控制器之所以被这样命名，是因为它控制着连接到它的整个USB总线。控制器是PCI总线和USB总线之间的桥梁，同时也是该总线上的第一个USB设备。

所有的根集线器都被USB核心分配了一个唯一的编号。在我们的例子中，根集线器被称为usb2，因为它是第二个注册到USB核心的根集线器。在任何时候，一个系统中可以包含的根集线器的数量没有限制。

在USB总线上的每个设备都将根集线器的编号作为其名称的第一个数字。然后是一个-字符，然后是设备插入的端口的编号。因为我们的例子中的设备插入到第一个端口，所以在名称中添加了1。所以，主USB鼠标设备的设备名称是2-1。因为这个USB设备包含一个接口，所以在树中添加了另一个设备到sysfs路径。USB接口的命名方案是到此为止的设备名称：在我们的例子中，它是2-1，后面跟着一个冒号和USB配置编号，然后是一个点和接口编号。所以在这个例子中，设备名称是2-1:1.0，因为它是第一个配置并且接口编号为零。

所以总结一下，USB sysfs设备的命名方案是：

```
root_hub-hub_port:config.interface
```

当设备在USB树中进一步下沉，以及使用越来越多的USB集线器时，集线器端口号被添加到字符串中，跟在链中的前一个集线器端口号后面。对于一个两层深的树，设备名称看起来像：

```
root_hub-hub_port-hub_port:config.interface
```

如前面的USB设备和接口的目录列表所示，所有的USB特定信息都可以直接通过sysfs获得（例如，idVendor、idProduct和bMaxPower信息）。其中一个文件，bConfigurationValue，可以写入以改变正在使用的活动USB配置。这对于有多个配置的设备很有用，当内核无法确定选择哪个配置以正确操作设备时。许多USB调制解调器需要将正确的配置值写入这个文件，以便正确的USB驱动程序绑定到设备。

Sysfs并没有暴露USB设备的所有不同部分，因为它在接口级别停止了。设备可能包含的任何备选配置都没有显示，接口关联的端点的详细信息也没有显示。这些信息可以在usbfs文件系统中找到，该文件系统在系统的/proc/bus/usb/目录下挂载。/proc/bus/usb/devices文件确实显示了在sysfs中暴露的所有相同信息，以及系统中所有存在的USB设备的备选配置和端点信息。usbfs还允许用户空间程序直接与USB设备通信，这使得许多内核驱动程序被移动到用户空间，那里更容易维护和调试。USB扫描器驱动程序就是一个很好的例子，因为它现在不再存在于内核中，因为它的功能现在包含在用户空间的SANE库程序中。

## 13.3. USB 的 Urbs

Linux内核中的USB代码使用称为urb（USB请求块）的东西与所有USB设备进行通信。这个请求块用struct urb结构体描述，并可以在include/linux/usb.h文件中找到。

urb用于以异步方式向特定USB设备上的特定USB端点发送或接收数据。它的使用方式类似于文件系统异步I/O代码中使用的kiocb结构体，或者网络代码中使用的struct skbuff。USB设备驱动程序可以为单个端点分配多个urb，也可以根据驱动程序的需要，重用一个urb用于多个不同的端点。设备中的每个端点都可以处理一个urb队列，这样可以在队列为空之前向同一端点发送多个urb。urb的典型生命周期如下：

- 由USB设备驱动程序创建。
- 分配给特定USB设备的特定端点。
- 由USB设备驱动程序提交给USB核心。
- 由USB核心提交给指定设备的特定USB主机控制器驱动程序。
- 由处理USB主机控制器驱动程序进行USB传输到设备。
- 当urb完成时，USB主机控制器驱动程序通知USB设备驱动程序。

urb也可以在任何时候被提交urb的驱动程序取消，或者如果设备从系统中移除，由USB核心取消。urb是动态创建的，并包含一个内部引用计数，使得当urb的最后一个用户释放它时，它可以自动被释放。

本章描述的处理urb的过程是有用的，因为它允许流式传输和其他复杂的、重叠的通信，使驱动程序能够达到最高可能的数据传输速度。但是，如果你只想发送单个的批量或控制消息，并且不关心数据吞吐率，那么也有更简便的程序可用。（参见“无urb的USB传输”部分。）

### 13.3.1. struct urb

对于USB设备驱动程序来说，struct urb结构体的字段包括：

```c
struct usb_device *dev
```

指向发送此urb的struct usb_device的指针。在urb可以发送到USB核心之前，这个变量必须由USB驱动程序初始化。

```c
unsigned int pipe
```

这个urb要发送到的特定struct usb_device的端点信息。在urb可以发送到USB核心之前，这个变量必须由USB驱动程序初始化。

为了设置这个结构体的字段，驱动程序根据流量的方向，使用以下适当的函数。注意，每个端点只能是一种类型。

```c
unsigned int usb_sndctrlpipe(struct usb_device *dev,
unsigned int endpoint)
```

为指定的USB设备和指定的端点号指定一个控制OUT端点。

```c
unsigned int usb_rcvctrlpipe(struct usb_device *dev,
unsigned int endpoint)
```

为指定的USB设备和指定的端点号指定一个控制IN端点。

```c
unsigned int usb_sndbulkpipe(struct usb_device *dev,
unsigned int endpoint)
```

为指定的USB设备和指定的端点号指定一个批量OUT端点。

```c
unsigned int usb_rcvbulkpipe(struct usb_device *dev,
unsigned int endpoint)
```

为指定的USB设备和指定的端点号指定一个批量IN端点。

```c
unsigned int usb_sndintpipe(struct usb_device *dev,
unsigned int endpoint)
```

为指定的USB设备和指定的端点号指定一个中断OUT端点。

```c
unsigned int usb_rcvintpipe(struct usb_device *dev,
unsigned int endpoint)
```

为指定的USB设备和指定的端点号指定一个中断IN端点。

```c
unsigned int usb_sndisocpipe(struct usb_device *dev,
unsigned int endpoint)
```

为指定的USB设备和指定的端点号指定一个等时OUT端点。

```c
unsigned int usb_rcvisocpipe(struct usb_device *dev,
unsigned int endpoint)
```

为指定的USB设备和指定的端点号指定一个等时IN端点。

```c
unsigned int transfer_flags
```

这个变量可以设置为多个不同的位值，取决于USB驱动程序希望urb发生什么。可用的值有：

URB_SHORT_NOT_OK 当设置时，它指定任何可能发生的IN端点上的短读应被USB核心视为错误。这个值只对要从USB设备读取的urb有用，对于写urb无用。

URB_ISO_ASAP 如果urb是等时的，这个位可以设置，如果驱动程序希望urb尽快被调度，只要带宽利用率允许，就设置urb中的start_frame变量。如果这个位没有为等时urb设置，驱动程序必须指定start_frame值，并且如果传输不能在那个时刻开始，必须能够正确恢复。有关等时urb的更多信息，请参见即将到来的部分。

URB_NO_TRANSFER_DMA_MAP 当urb包含要传输的DMA缓冲区时应设置。USB核心使用transfer_dma变量指向的缓冲区，而不是transfer_buffer变量指向的缓冲区。

URB_NO_SETUP_DMA_MAP 像URB_NO_TRANSFER_DMA_MAP位一样，这个位用于已经设置了DMA缓冲区的控制urb。如果设置，USB核心使用setup_dma变量指向的缓冲区，而不是setup_packet变量。

URB_ASYNC_UNLINK 如果设置，对这个urb的usb_unlink_urb调用几乎立即返回，并且urb在后台被取消链接。否则，函数等待直到urb完全取消链接并完成后才返回。小心

使用这个位，因为它可能使同步问题非常难以调试。

URB_NO_FSBR 只被UHCI USB主机控制器驱动程序使用，告诉它不要尝试执行前端总线回收逻辑。这个位通常不应该被设置，因为带有UHCI主机控制器的机器会产生大量的CPU开销，而PCI总线在等待设置了这个位的urb时会饱和。

URB_ZERO_PACKET 如果设置，当数据对齐到端点包边界时，批量out urb通过发送一个不包含数据的短包来结束。一些有问题的USB设备（如许多USB到IR设备）需要这个功能才能正常工作。

URB_NO_INTERRUPT 如果设置，硬件可能在urb完成时不生成中断。这个位应该谨慎使用，只有在将多个urb排队到同一个端点时才使用。USB核心函数使用这个功能来进行DMA缓冲区传输。

```c
void *transfer_buffer
```

指向在向设备发送数据（对于OUT urb）或从设备接收数据（对于IN urb）时使用的缓冲区的指针。为了让主机控制器正确访问这个缓冲区，它必须通过调用kmalloc创建，不能在堆栈或静态上创建。对于控制端点，这个缓冲区用于传输的数据阶段。

```c
dma_addr_t transfer_dma
```

用于使用DMA向USB设备传输数据的缓冲区。

```c
int transfer_buffer_length
```

指向transfer_buffer或transfer_dma变量的缓冲区的长度（因为只有一个可以用于urb）。如果这个值为0，那么USB核心不使用任何传输缓冲区。

对于OUT端点，如果端点的最大大小小于这个变量指定的值，那么向USB设备的传输会被分解成更小的块，以便正确传输数据。这种大的传输发生在连续的USB帧中。在一个

urb中提交一个大的数据块，并让USB主机控制器将其分解成更小的片段，比连续发送更小的缓冲区要快得多。

```c
unsigned char *setup_packet
```

指向控制urb的设置包的指针。它在传输缓冲区中的数据之前被传输。这个变量只对控制urb有效。

```c
dma_addr_t setup_dma
```

控制urb的设置包的DMA缓冲区。它在正常传输缓冲区中的数据之前被传输。这个变量只对控制urb有效。

```c
usb_complete_t complete
```

指向完成处理函数的指针，当urb完全传输或urb发生错误时，由USB核心调用。在这个函数中，USB驱动程序可以检查urb，释放它，或者重新提交它进行另一次传输。（有关完成处理程序的更多详细信息，请参见“完成Urbs：完成回调处理程序”部分。）usb_complete_t typedef定义为：

```c
typedef void (*usb_complete_t)(struct urb *, struct
pt_regs *);
```

`void *context` 指向一个可以由USB驱动程序设置的数据blob的指针。当urb返回给驱动程序时，可以在完成处理程序中使用它。有关这个变量的更多详细信息，请参见下一节。

```c
int actual_length
```

当urb完成时，这个变量被设置为urb发送（对于OUT urbs）或接收（对于IN urbs）的数据的实际长度。对于IN urbs，必须使用这个变量，而不是transfer_buffer_length变量，因为接收的数据可能小于整个缓冲区大小。

```c
int status
```

当urb完成或被USB核心处理时，这个变量被设置为urb的当前状态。USB驱动程序可以安全访问这个变量的唯一时间是在urb完成处理程序函数中（在“完成Urbs：完成回调处理程序”部分中描述）。这个限制是为了防止在USB核心处理urb时发生竞争条件。对于等时urb，这个变量中的成功值（0）仅表示urb是否已被取消链接。要获取等时urb的详细状态，应检查iso_frame_desc变量。
这个变量的有效值包括：

```c
0
```

urb传输成功。

```c
-ENOENT
```

urb被usb_kill_urb调用停止。

```c
-ECONNRESET
```

urb被usb_unlink_urb调用取消链接，并且urb的transfer_flags变量被设置为URB_ASYNC_UNLINK。

```
-EINPROGRESS
```

urb仍在被USB主机控制器处理。如果你的驱动程序看到这个值，那么你的驱动程序中存在一个bug。

```
-EPROTO
```

以下错误之一发生在这个urb： • 传输过程中发生了bitstuff错误。 • 硬件在规定时间内没有收到响应包。

```
-EILSEQ
```

urb传输中存在CRC不匹配。

```
-EPIPE
```

端点现在已经停止。如果涉及的端点不是控制端点，这个错误可以通过调用函数usb_clear_halt清除。

```
-ECOMM
```

在传输过程中，接收到的数据比能写入系统内存的速度快。这个错误值只对IN urb有效。

```
-ENOSR
```

在传输过程中，无法从系统内存中快速获取数据以跟上请求的USB数据速率。这个错误值只对OUT urb有效。

```
-EOVERFLOW
```

urb发生了“babble”错误。当端点接收到的数据超过端点指定的最大包大小时，会发生“babble”错误。

```
-EREMOTEIO
```

只有在urb的transfer_flags变量中设置了URB_SHORT_NOT_OK标志时才会发生，意味着urb请求的全部数据没有被接收。

```
-ENODEV
```

USB设备现在已经从系统中消失。

```
-EXDEV
```

只对等时urb有效，意味着传输只部分完成。为了确定传输了什么，驱动程序必须查看每个帧的状态。

```
-EINVAL
```

urb发生了非常严重的错误。USB内核文档描述了这个值的含义： 如果发生这种情况：注销并回家 如果在urb结构中设置了一个参数错误，或者在usb_submit_urb调用中提交了一个错误的函数参数到USB核心，也可能发生这种情况。

```
-ESHUTDOWN
```

USB主机控制器驱动程序出现了严重错误；它现在已经被禁用，或者设备已经从系统中断开，并且在设备被移除后提交了urb。如果在urb提交给设备时，设备的配置发生了变化，也可能发生这种情况。
一般来说，错误值-EPROTO、-EILSEQ和-EOVERFLOW表示设备、设备固件或连接设备和计算机的线缆存在硬件问题。

```
int start_frame
```

设置或返回等时传输使用的初始帧号。

```
int interval
```

urb被轮询的间隔。这只对中断或等时urb有效。这个值的单位取决于设备的速度。对于低速和全速设备，单位是帧，相当于毫秒。对于设备，单位是微帧，相当于1/8毫秒的单位。这个值必须在urb发送到USB核心之前由USB驱动程序为等时或中断urb设置。

```
int number_of_packets
```

只对等时urb有效，指定由这个urb处理的等时传输缓冲区的数量。这个值必须在urb发送到USB核心之前由USB驱动程序为等时urb设置。

```c
int error_count
```

只有在等时urb完成后，USB核心才为其设置。它指定报告任何类型错误的等时传输的数量。

```c
struct usb_iso_packet_descriptor iso_frame_desc[0]
```

只对等时urb有效。这个变量是一个由struct usb_iso_packet_descriptor结构组成的数组，构成了这个urb。这个结构允许一个单独的urb一次定义多个等时传输。它也用于收集每个单独传输的传输状态。

`struct usb_iso_packet_descriptor` 由以下字段组成：

```c
unsigned int offset
```

传输缓冲区中这个包的数据的位置的偏移量（第一个字节的偏移量为0）。

```c
unsigned int length
```

这个包的传输缓冲区的长度。

```c
unsigned int actual_length
```

接收到这个等时包的传输缓冲区的数据的长度。

```c
unsigned int status
```

这个包的单独等时传输的状态。它可以取与主struct urb结构的status变量相同的返回值。

## 13.3.2. 创建和销毁 urb

struct urb结构体在驱动程序中或在另一个结构体中绝不能静态创建，因为这将破坏USB核心对urb使用的引用计数方案。它必须通过调用usb_alloc_urb函数创建。

这个函数的原型是：

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

第一个参数，iso_packets，是这个urb应该包含的等时包的数量。如果你不想创建一个等时urb，这个变量应该设置为0。第二个参数，mem_flags，是传递给kmalloc函数调用以从内核分配内存的同类型的标志（参见第8章的“The Flags Argument”部分，了解这些标志的详细信息）。如果函数成功分配了足够的空间给urb，一个指向urb的指针会返回给调用者。如果返回值是NULL，那么USB核心内部发生了一些错误，驱动程序需要进行适当的清理。

创建urb后，必须正确初始化它，才能被USB核心使用。请参阅下一节，了解如何初始化不同类型的urb。

为了告诉USB核心驱动程序已经完成了urb的使用，驱动程序必须调用usb_free_urb函数。这个函数只有一个参数：

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

参数是一个指向你想要释放的struct urb的指针。调用这个函数后，urb结构体就消失了，驱动程序不能再访问它。

- URB（USB Request Block）是USB设备驱动程序与USB核心之间交互的基本数据结构。它代表了一个USB事务，可以是数据的发送或接收。URB包含了所有必要的信息，如目标USB设备、端点、数据缓冲区等，以完成一个USB事务。在Linux内核中，URB是通过特定的函数（如usb_submit_urb）提交给USB核心的，然后USB核心负责将这些URB排队并发送到相应的USB设备。

#### 13.3.2.1. 中断 urb

函数usb_fill_int_urb是一个辅助函数，用于正确初始化一个要发送到USB设备的中断端点的urb：

```c
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);
```

这个函数包含很多参数：

```c
struct urb *urb
```

指向要初始化的urb的指针。

```c
struct usb_device *dev
```

这个urb要发送到的USB设备。

```c
unsigned int pipe
```

这个urb要发送到的USB设备的特定端点。这个值是用前面提到的usb_sndintpipe或usb_rcvintpipe函数创建的。

```c
void *transfer_buffer
```

指向从中取出传出数据或接收传入数据的缓冲区的指针。注意，这不能是一个静态缓冲区，必须通过调用kmalloc创建。

```c
int buffer_length
```

由transfer_buffer指针指向的缓冲区的长度。

```c
usb_complete_t complete
```

指向完成处理程序的指针，当这个urb完成时调用。

```c
void *context
```

指向要添加到urb结构体中的blob的指针，以便稍后由完成处理程序函数检索。

```c
int interval
```

这个urb应该被调度的间隔。参见前面对struct urb结构体的描述，找到这个值的正确单位。

### 13.3.2.2. Bulk urbs

批量urb的初始化与中断urb类似。执行此操作的函数是usb_fill_bulk_urb，它的形式如下：

```c
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);
```

函数参数与usb_fill_int_urb函数中的所有参数相同。然而，没有间隔参数，因为批量urb没有间隔值。请注意，unsigned int pipe变量必须通过调用usb_sndbulkpipe或usb_rcvbulkpipe函数进行初始化。

usb_fill_bulk_urb函数不设置urb中的transfer_flags变量，所以对这个字段的任何修改都必须由驱动程序自己完成。

### 13.3.2.3. Control urbs

控制urb的初始化几乎与批量urb相同，通过调用函数usb_fill_control_urb进行：

```c
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);
```

函数参数与usb_fill_bulk_urb函数中的所有参数相同，除了有一个新参数，`unsigned char *setup_packet`，它必须指向要发送到端点的设置包数据。此外，unsigned int pipe变量必须通过调用usb_sndctrlpipe或usb_rcvictrlpipe函数进行初始化。

usb_fill_control_urb函数不设置urb中的transfer_flags变量，所以对这个字段的任何修改都必须由驱动程序自己完成。大多数驱动程序不使用这个函数，因为使用同步API调用（如“无urb的USB传输”部分所述）要简单得多。

### 13.3.2.4. sochronous urbs

不幸的是，等时urb没有像中断、控制和批量urb那样的初始化函数。所以在它们可以提交给USB核心之前，必须在驱动程序中“手动”初始化。以下是如何正确初始化这种类型的urb的示例。它取自主内核源代码树中的drivers/usb/media目录下的konicawc.c内核驱动程序。

```c
urb->dev = dev;

urb->context = uvd;

urb->pipe = usb_rcvisocpipe(dev, uvd->video_endp-1);

urb->interval = 1;

urb->transfer_flags = URB_ISO_ASAP;

urb->transfer_buffer = cam->sts_buf[i];

urb->complete = konicawc_isoc_irq;

urb->number_of_packets = FRAMES_PER_DESC;

urb->transfer_buffer_length = FRAMES_PER_DESC;

for (j=0; j < FRAMES_PER_DESC; j++) {

        urb->iso_frame_desc[j].offset = j;

        urb->iso_frame_desc[j].length = 1;

}
```

- 在这个示例中，首先设置了urb的设备、上下文、管道、间隔、传输标志、传输缓冲区、完成处理程序、包的数量和传输缓冲区的长度。然后，对于每个等时帧描述符，设置了其偏移量和长度。这个示例显示了如何手动初始化等时urb。

### 13.3.3. 提交 urb

一旦urb被USB驱动程序正确创建和初始化，它就准备好被提交到USB核心，以便发送到USB设备。这是通过调用函数usb_submit_urb完成的：

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

urb参数是一个指向要发送到设备的urb的指针。mem_flags参数等同于传递给kmalloc调用的同一参数，用于告诉USB核心如何在此时分配任何内存缓冲区。成功提交urb到USB核心后，它应该永远不试图访问urb结构体的任何字段，直到调用complete函数。

因为函数usb_submit_urb可以在任何时候被调用（包括在中断上下文中），所以mem_flags变量的指定必须是正确的。实际上，只有三个有效的值应该被使用，取决于何时调用usb_submit_urb：

```c
GFP_ATOMIC
```

当以下条件为真时，应使用此值：

- 调用者在urb完成处理程序、中断、底半部、任务或定时器回调中。
- 调用者持有一个自旋锁或读写锁。注意，如果持有一个信号量，这个值是不必要的。
- current→state不是TASK_RUNNING。状态总是TASK_RUNNING，除非驱动程序自己改变了当前状态。

```c
GFP_NOIO
```

如果驱动程序在块I/O路径中，应使用此值。所有存储类型设备的错误处理路径也应使用它。

```c
GFP_KERNEL
```

对于所有其他不属于上述类别的情况，应使用此值。

## 13.3.4. 完成 urb: 完成Callback Handler

如果对usb_submit_urb的调用成功，将urb的控制权转移给USB核心，函数返回0；否则，返回一个负的错误号。如果函数成功，当urb完成时，urb的完成处理程序（由complete函数指针指定）被调用一次。当这个函数被调用时，USB核心已经完成了URB的处理，现在它的控制权返回给设备驱动程序。

只有三种方式可以完成urb并调用complete函数：

- urb成功发送到设备，设备返回适当的确认。对于OUT urb，数据已成功发送，对于IN urb，已成功接收请求的数据。如果发生了这种情况，urb中的status变量被设置为0。
- 在从设备发送或接收数据时发生了某种错误。这通过urb结构体中status变量的错误值来表示。
- urb从USB核心“取消链接”。这发生在驱动程序告诉USB核心通过调用usb_unlink_urb或usb_kill_urb取消提交的urb，或者当设备从系统中移除并且已向其提交了urb时。

在本章后面将展示如何在urb完成调用中测试不同返回值的示例。

### 13.3.5. 取消 urb

要停止已提交给USB核心的urb，应调用函数usb_kill_urb或usb_unlink_urb：

```c
int usb_kill_urb(struct urb *urb);

int usb_unlink_urb(struct urb *urb);
```

这两个函数的urb参数都是指向要取消的urb的指针。

当函数是usb_kill_urb时，urb生命周期被停止。这个函数通常在设备从系统断开连接时，在断开连接回调中使用。

对于一些驱动程序，应使用usb_unlink_urb函数来告诉USB核心停止一个urb。这个函数在返回给调用者之前不等待urb完全停止。这对于在中断处理程序中或者持有自旋锁时停止urb很有用，因为等待urb完全停止需要USB核心有能力让调用进程睡眠。这个函数要求在要求停止的urb中设置URB_ASYNC_UNLINK标志值，以便正确工作。

## 13.4. 编写一个 USB 驱动

编写USB设备驱动程序的方法类似于pci_driver：驱动程序将其驱动对象注册到USB子系统，然后使用供应商和设备标识符来判断其硬件是否已经安装。

### 13.4.1. 驱动支持什么设备

struct usb_device_id结构体提供了此驱动程序支持的不同类型的USB设备的列表。USB核心使用此列表来决定将设备交给哪个驱动程序，热插拔脚本使用此列表来决定当特定设备插入系统时自动加载哪个驱动程序。

struct usb_device_id结构体定义了以下字段：

```c
__u16 match_flags
```

决定设备应与结构体中的哪些字段进行匹配。这是一个由include/linux/mod*devicetable.h文件中指定的不同`USB_DEVICE_ID_MATCH**`值定义的位字段。这个字段通常不直接设置，而是由后面描述的USB_DEVICE类型宏初始化。

```c
__u16 idVendor
```

设备的USB供应商ID。这个数字由USB论坛分配给其成员，不能由其他人制定。

```c
__u16 idProduct
```

设备的USB产品ID。所有被分配了供应商ID的供应商都可以按他们选择的方式管理他们的产品ID。

```c
__u16 bcdDevice_lo

__u16 bcdDevice_hi
```

定义供应商分配的产品版本号范围的低端和高端。bcdDevice_hi值是包含的；它的值是最高编号设备的编号。这两个值都以二进制编码的十进制（BCD）形式表示。这些变量，结合idVendor和idProduct，用于定义设备的特定版本。

```c
__u8 bDeviceClass

__u8 bDeviceSubClass

__u8 bDeviceProtocol
```

分别定义设备的类、子类和协议。这些数字由USB论坛分配，并在USB规范中定义。这些值指定了整个设备的行为，包括此设备上的所有接口。

```c
__u8 bInterfaceClass

__u8 bInterfaceSubClass

__u8 bInterfaceProtocol
```

就像上面的设备特定值一样，这些分别定义了单个接口的类、子类和协议。这些数字由USB论坛分配，并在USB规范中定义。

```c
kernel_ulong_t driver_info
```

这个值不用于匹配，但它保存了驱动程序可以用来在USB驱动程序的probe回调函数中区分不同设备的信息。
与PCI设备一样，有一些宏用于初始化这个结构体：

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

创建一个可以用来只匹配指定的供应商和产品ID值的struct usb_device_id。这对于需要特定驱动程序的USB设备非常常见。

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

创建一个可以用来只匹配指定的供应商和产品ID值在版本范围内的struct usb_device_id。

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

创建一个可以用来匹配特定类别的USB设备的struct usb_device_id。

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

创建一个可以用来匹配特定类别的USB接口的struct usb_device_id。

因此，对于一个简单的USB设备驱动程序，它只控制来自单个供应商的单个USB设备，struct usb_device_id表将被定义为：

```c
/* table of devices that work with this driver */

static struct usb_device_id skel_table [ ] = {

    { USB_DEVICE(USB_SKEL_VENDOR_ID, USB_SKEL_PRODUCT_ID)
},

    { }                     /* Terminating entry */

};

MODULE_DEVICE_TABLE (usb, skel_table);
```

与PCI驱动程序一样，MODULE_DEVICE_TABLE宏是必需的，以便用户空间工具能够找出这个驱动程序可以控制哪些设备。但对于USB驱动程序，字符串usb必须是宏中的第一个值。

### 13.4.2. 注册一个 USB 驱动

所有USB驱动程序都必须创建一个struct usb_driver结构体。这个结构体必须由USB驱动程序填充，它包含了一些函数回调和描述USB驱动程序的变量，供USB核心代码使用：

```c
struct module *owner
```

指向此驱动程序的模块所有者的指针。USB核心使用它来正确地引用计数这个USB驱动程序，以便它不会在不适当的时刻被卸载。这个变量应该设置为THIS_MODULE宏。

```c
const char *name
```

指向驱动程序名称的指针。它必须在内核中的所有USB驱动程序中是唯一的，通常设置为驱动程序的模块名称。当驱动程序在内核中时，它会出现在/sys/bus/usb/drivers/下

的sysfs中。

```c
const struct usb_device_id *id_table
```

指向包含所有这个驱动程序可以接受的不同类型的USB设备的列表的struct usb_device_id表的指针。如果这个变量没有设置，USB驱动程序中的probe函数回调永远不会被调用。如果你希望你的驱动程序总是被每个系统中的USB设备调用，创建一个只设置driver_info字段的条目：

```c
static struct usb_device_id usb_ids[ ] = {
    {.driver_info = 42},
    { }
};
int (*probe) (struct usb_interface *intf, const struct
usb_device_id *id)
```

指向USB驱动程序中的probe函数的指针。当USB核心认为它有一个这个驱动程序可以处理的struct usb_interface时，它会调用这个函数。一个指向USB核心用来做出这个决定的struct usb_device_id的指针也被传递给这个函数。如果USB驱动程序声明了传递给它的struct usb_interface，它应该正确地初始化设备并返回0。如果驱动程序不想声明设备，或者发生错误，它应该返回一个负的错误值。

```c
void (*disconnect) (struct usb_interface *intf)
```

指向USB驱动程序中的disconnect函数的指针。当struct usb_interface从系统中移除或者驱动程序从USB核心卸载时，USB核心会调用这个函数。

因此，要创建一个有效的struct usb_driver结构体，只需要初始化五个字段：

```c
static struct usb_driver skel_driver = {
    .owner = THIS_MODULE,
    .name = "skeleton",
    .id_table = skel_table,
    .probe = skel_probe,
    .disconnect = skel_disconnect,
};
```

struct usb_driver确实包含了一些更多的回调，但是它们通常不常用，并且USB驱动程序正常工作并不需要它们：

```c
int (*ioctl) (struct usb_interface *intf, unsigned int code, void *buf)
```

指向USB驱动程序中的ioctl函数的指针。如果存在，当用户空间程序在与此USB驱动程序关联的USB设备的usbfs文件系统设备条目上进行ioctl调用时，它会被调用。在实践中，只有USB hub驱动程序使用这个ioctl，因为没有任何其他真正需要的USB驱动程序使用它。

```c
int (*suspend) (struct usb_interface *intf, u32 state)
```

指向USB驱动程序中的suspend函数的指针。当设备要被USB核心挂起时，它会被调用。

```c
int (*resume) (struct usb_interface *intf)
```

指向USB驱动程序中的resume函数的指针。当设备被USB核心恢复时，它会被调用。

要将struct usb_driver注册到USB核心，需要调用usb_register_driver，并传递一个指向struct usb_driver的指针。这通常在USB驱动程序的模块初始化代码中完成：

```c
static int __init usb_skel_init(void)

{

    int result;

    /* register this driver with the USB subsystem */

    result = usb_register(&skel_driver);

    if (result)

        err("usb_register failed. Error number %d",
result);

    return result;

}
```

当USB驱动程序要被卸载时，需要从内核中注销struct usb_driver。这是通过调用usb_deregister完成的。当这个调用发生时，任何当前绑定到这个驱动程序的USB接口都会被断开连接，并且会为它们调用disconnect函数。

```c
static void __exit usb_skel_exit(void)

{

    /* deregister this driver with the USB subsystem */

    usb_deregister(&skel_driver);

}
```

### 13.4.2.1. 探测和去连接的细节

在前一节描述的struct usb_driver结构中，驱动程序指定了两个函数，USB核心在适当的时候调用这两个函数。当安装了一个设备，USB核心认为这个驱动程序应该处理时，就会调用probe函数；probe函数应该对传递给它的设备信息进行检查，并决定驱动程序是否真的适合该设备。当驱动程序由于某种原因不再控制设备时，就会调用disconnect函数，这时可以进行清理。

probe和disconnect函数回调都在USB hub内核线程的上下文中被调用，所以在它们内部睡眠是合法的。然而，建议尽可能在设备被用户打开时完成大部分工作，以便将USB探测时间保持在最小。这是因为USB核心在一个单独的线程中处理USB设备的添加和移除，所以任何慢的设备驱动程序都可能导致USB设备检测时间变慢，并被用户注意到。

在probe函数回调中，USB驱动程序应该初始化任何可能用来管理USB设备的本地结构。它还应该将关于设备的任何需要的信息保存到本地结构中，因为通常在这个时候这样做更容易。例如，USB驱动程序通常希望检测设备的端点地址和缓冲区大小，因为它们是与设备通信所必需的。下面是一些示例代码，它检测了BULK类型的IN和OUT端点，并在本地设备结构中保存了一些关于它们的信息:

```c
/* set up the endpoint information */
/* use only the first bulk-in and bulk-out endpoints */
iface_desc = interface->cur_altsetting;
for (i = 0; i < iface_desc->desc.bNumEndpoints; ++i) {
    endpoint = &iface_desc->endpoint[i].desc;
    if (!dev->bulk_in_endpointAddr &&
        (endpoint->bEndpointAddress & USB_DIR_IN) &&
        ((endpoint->bmAttributes &
USB_ENDPOINT_XFERTYPE_MASK)
                == USB_ENDPOINT_XFER_BULK)) {
        /* we found a bulk in endpoint */
        buffer_size = endpoint->wMaxPacketSize;
        dev->bulk_in_size = buffer_size;
        dev->bulk_in_endpointAddr = endpoint-
>bEndpointAddress;
        dev->bulk_in_buffer = kmalloc(buffer_size,
GFP_KERNEL);
        if (!dev->bulk_in_buffer) {
            err("Could not allocate bulk_in_buffer");
            goto error;
        }
    }
    if (!dev->bulk_out_endpointAddr &&
        !(endpoint->bEndpointAddress & USB_DIR_IN) &&
        ((endpoint->bmAttributes &
USB_ENDPOINT_XFERTYPE_MASK)
         == USB_ENDPOINT_XFER_BULK)) {
        /* we found a bulk out endpoint */
        dev->bulk_out_endpointAddr = endpoint-
>bEndpointAddress;
    }
}
if (!(dev->bulk_in_endpointAddr && dev-
>bulk_out_endpointAddr)) {
    err("Could not find both bulk-in and bulk-out
endpoints");
    goto error;
}
```

这段代码首先遍历这个接口中存在的每一个端点，并将一个本地指针赋值给端点结构，以便后续更容易访问：

```c
for (i = 0; i < iface_desc->desc.bNumEndpoints; ++i) {

    endpoint = &iface_desc->endpoint[i].desc;

}
```

然后，当我们有了一个端点，并且还没有找到一个bulk IN类型的端点，我们查看这个端点的方向是否为IN。这可以通过查看bEndpointAddress端点变量是否包含位掩码USB_DIR_IN来测试。如果这是真的，我们通过首先用USB_ENDPOINT_XFERTYPE_MASK位掩码屏蔽bmAttributes变量，然后检查它是否匹配值USB_ENDPOINT_XFER_BULK，来确定端点类型是否为bulk：

```c
if (!dev->bulk_in_endpointAddr &&

    (endpoint->bEndpointAddress & USB_DIR_IN) &&

    ((endpoint->bmAttributes &
USB_ENDPOINT_XFERTYPE_MASK)

            == USB_ENDPOINT_XFER_BULK)) {
```

如果所有这些测试都是真的，驱动程序知道它找到了正确类型的端点，并可以在本地结构中保存关于端点的信息，以便后续在它上面进行通信：

```c
/* we found a bulk in endpoint */

buffer_size = endpoint->wMaxPacketSize;

dev->bulk_in_size = buffer_size;

dev->bulk_in_endpointAddr = endpoint->bEndpointAddress;

dev->bulk_in_buffer = kmalloc(buffer_size, GFP_KERNEL);

if (!dev->bulk_in_buffer) {

    err("Could not allocate bulk_in_buffer");

    goto error;

}
```

因为USB驱动程序需要在设备的生命周期后期检索与这个struct usb_interface关联的本地数据结构，所以可以调用函数usb_set_intfdata：

```c
/* save our data pointer in this interface device */

usb_set_intfdata(interface, dev);
```

这个函数接受一个指向任何数据类型的指针，并将它保存在struct usb_interface结构中以便后续访问。要检索数据，应该调用函数usb_get_intfdata：

```c
struct usb_skel *dev;

struct usb_interface *interface;

int subminor;

int retval = 0;

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);

if (!dev) {

    retval = -ENODEV;

    goto exit;

}
```

usb_get_intfdata通常在USB驱动程序的open函数中被调用，然后在disconnect函数中再次被调用。多亏了这两个函数，USB驱动程序不需要保持一个静态的指针数组，存

储系统中所有当前设备的单独设备结构。设备信息的间接引用允许任何USB驱动程序支持无限数量的设备。

如果USB驱动程序没有与处理设备用户交互的其他类型的子系统（如输入、tty、视频等）关联，驱动程序可以使用USB主要编号，以便使用传统的字符驱动程序接口与用户空间交互。为了做到这一点，USB驱动程序必须在它想要向USB核心注册设备时，在probe函数中调用usb_register_dev函数。确保设备和驱动程序处于适当的状态，以便用户想要访问设备时可以立即调用此函数。

```c
/* we can register the device now, as it is ready */
retval = usb_register_dev(interface, &skel_class);
if (retval) {
    /* something prevented us from registering this driver */
    err("Not able to get a minor for this device.");
    usb_set_intfdata(interface, NULL);
    goto error;
}
```

usb_register_dev函数需要一个指向struct usb_interface的指针和一个指向struct usb_class_driver的指针。这个struct usb_class_driver用于定义USB驱动程序希望USB核心在注册次要编号时知道的许多不同参数。这个结构由以下变量组成：

```c
char *name
```

sysfs用来描述设备的名称。如果存在，前导路径名只在devfs中使用，本书中并未涵盖。如果设备的编号需要在名称中，名称字符串中应该有字符%d。例如，要创建devfs

名称usb/foo1和sysfs类名foo1，名称字符串应设置为usb/foo%d。

```c
struct file_operations *fops;
```

指向这个驱动程序已经定义的struct file_operations的指针，用于注册为字符设备。有关此结构的更多信息，请参见第3章。

```c
mode_t mode;
```

为此驱动程序创建的devfs文件的模式；否则未使用。这个变量的典型设置是值S_IRUSR和值S_IWUSR的组合，这将只提供设备文件所有者的读写访问权限。

```c
int minor_base;
```

这是为这个驱动程序分配的次要范围的开始。所有与这个驱动程序关联的设备都以这个值开始，创建具有唯一的、递增的次要编号。除非已经为内核启用了CONFIG_USB_DYNAMIC_MINORS配置选项，否则一次只允许有16个设备与这个驱动程序关联。如果是这样，这个变量将被忽略，所有设备的次要编号都按照先到先得的方式分配。建议已经启用此选项的系统使用udev之类的程序来管理系统中的设备节点，因为静态的/dev树将无法正常工作。

当USB设备断开连接时，应尽可能清理与设备关联的所有资源。此时，如果在probe函数期间调用了usb_register_dev为这个USB设备分配一个次要编号，那么必须调用函数usb_deregister_dev将次要编号返回给USB核心。在disconnect函数中，从接口中检索之前通过调用usb_set_intfdata设置的任何数据也很重要。然后将struct usb_interface结构中的数据指针设置为NULL，以防止进一步错误地访问数据：

```c
static void skel_disconnect(struct usb_interface
*interface)

{

    struct usb_skel *dev;

    int minor = interface->minor;

    /* prevent skel_open( ) from racing skel_disconnect(
) */

    lock_kernel( );

    dev = usb_get_intfdata(interface);

    usb_set_intfdata(interface, NULL);

    /* give back our minor */

    usb_deregister_dev(interface, &skel_class);

    unlock_kernel( );

     /* decrement our usage count */

    kref_put(&dev->kref, skel_delete);

    info("USB Skeleton #%d now disconnected", minor);

}
```

注意前面的代码片段中对lock_kernel的调用。这会获取大内核锁，以防止disconnect回调与试图获取正确接口数据结构的指针的open调用发生竞态条件。因为open是在获取大内核锁的情况下调用的，如果disconnect也获取了同样的锁，那么驱动程序的只有一部分可以访问并设置接口数据指针。

在为USB设备调用disconnect函数之前，USB核心会取消当前正在为设备传输的所有urb，所以驱动程序不必显式地调用usb_kill_urb为这些urb。如果驱动程序试图在通过调用usb_submit_urb断开连接后向USB设备提交一个urb，那么提交将失败，错误值为-EPIPE。

### 13.4.3. 提交和控制一个 urb

当驱动程序有数据要发送到USB设备时（通常在驱动程序的write函数中发生），必须为将数据传输到设备分配一个urb：

```c
urb = usb_alloc_urb(0, GFP_KERNEL);
if (!urb) {
    retval = -ENOMEM;
    goto error;
}
```

成功分配urb后，还应创建一个DMA缓冲区以最有效的方式将数据发送到设备，并将传递给驱动程序的数据复制到该缓冲区：

```c
buf = usb_buffer_alloc(dev->udev, count, GFP_KERNEL,
&urb->transfer_dma);

if (!buf) {

    retval = -ENOMEM;

    goto error;

}

if (copy_from_user(buf, user_buffer, count)) {

    retval = -EFAULT;

    goto error;

}
```

一旦数据正确地从用户空间复制到本地缓冲区，urb必须在提交给USB核心之前正确初始化：

```c
/* initialize the urb properly */

usb_fill_bulk_urb(urb, dev->udev,

          usb_sndbulkpipe(dev->udev, dev-
>bulk_out_endpointAddr),

          buf, count, skel_write_bulk_callback, dev);

urb->transfer_flags |= URB_NO_TRANSFER_DMA_MAP;
```

现在urb已经正确分配，数据已经正确复制，urb已经正确初始化，它可以提交给USB核心，以便传输到设备：

```c
/* send the data out the bulk port */

retval = usb_submit_urb(urb, GFP_KERNEL);

if (retval) {

    err("%s - failed submitting write urb, error %d",
__FUNCTION__, retval);

    goto error;

}
```

成功将urb传输到USB设备后（或者在传输过程中发生了什么），USB核心会调用urb回调。在我们的例子中，我们初始化urb指向函数skel_write_bulk_callback，这就是被调用的函数：

```c
static void skel_write_bulk_callback(struct urb *urb,
struct pt_regs *regs)

{

    /* sync/async unlink faults aren't errors */

    if (urb->status &&

        !(urb->status = = -ENOENT ||

          urb->status = = -ECONNRESET ||

          urb->status = = -ESHUTDOWN)) {

        dbg("%s - nonzero write bulk status received:
%d",

            __FUNCTION__, urb->status);

    }

    /* free up our allocated buffer */

    usb_buffer_free(urb->dev, urb-
>transfer_buffer_length,

            urb->transfer_buffer, urb->transfer_dma);

}
```

回调函数首先检查urb的状态，以确定这个urb是否成功完成。错误值-ENOENT、-ECONNRESET和-ESHUTDOWN并不是真正的传输错误，只是关于成功传输伴随的条件的报告。（参见“struct urb”部分详细介绍的urb可能的错误列表。）然后，回调释放了分配给这个urb进行传输的缓冲区。

在urb回调函数运行时向设备提交另一个urb是很常见的。这在向设备流式传输数据时很有用。请记住，urb回调在中断上下文中运行，所以它不应该进行任何内存分配，持有任何信号量，或者做任何可能导致进程睡眠的事情。在回调中提交urb时，使用GFP_ATOMIC标志告诉USB核心，如果在提交过程中需要分配新的内存块，不要睡眠。

## 13.5. 无 urb 的 USB 传送

有时候，USB驱动程序不想为了发送或接收一些简单的USB数据，而去经历创建struct urb，初始化它，然后等待urb完成函数运行的所有麻烦。有两个函数可以提供一个更简单的接口。

这两个函数是 `usb_bulk_msg` 和 `usb_control_msg`。这些函数允许驱动程序在不创建和初始化urb的情况下，发送或接收USB数据。这些函数在内部处理urb的创建、初始化和提交，以及等待urb完成。这些函数对于发送或接收简单的USB数据非常有用，但是它们不适合用于流式传输数据，因为它们在urb完成之前会阻塞。

以下是这两个函数的原型：

```c
int usb_bulk_msg(struct usb_device *usb_dev, unsigned int
pipe,
    void *data, int len, int *actual_length,
    int timeout);
int usb_control_msg(struct usb_device *usb_dev, unsigned
int pipe,
    __u8 request, __u8 requesttype, __u16 value, __u16
index,
    void *data, __u16 size, int timeout);
```

这两个函数都返回一个错误代码（如果有错误），或者返回0表示成功。

### 13.5.1. usb_bulk_msg 接口

usb_bulk_msg创建一个USB批量urb并将其发送到指定的设备，然后等待它完成后再返回给调用者。它的定义如下：

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

此函数的参数为：

`struct usb_device *usb_dev` 指向要发送批量消息的USB设备的指针。

unsigned int pipe 要发送此批量消息的USB设备的特定端点。此值是通过调用usb_sndbulkpipe或usb_rcvbulkpipe创建的。

`void *data` 如果这是一个OUT端点，这是指向要发送到设备的数据的指针。如果这是一个IN端点，这是一个指向数据应放置的位置的指针，数据从设备读取后放置在此处。

`int len` 由数据参数指向的缓冲区的长度。

`int *actual_length` 函数将实际传输到设备或从设备接收的字节数放置的位置的指针，具体取决于端点的方向。

`int timeout` 应等待的时间（以jiffies为单位）在超时之前。如果此值为0，函数将永远等待消息完成。

如果函数成功，返回值为0；否则，返回一个负的错误号。此错误号与前面在“struct urb”部分为urbs描述的错误号相匹配。如果成功，actual_length参数包含从此消息传输或接收的字节数。

以下是使用此函数调用的示例：

```c
/* do a blocking bulk read to get data from the device */

retval = usb_bulk_msg(dev->udev,

              usb_rcvbulkpipe(dev->udev, dev-
>bulk_in_endpointAddr),

              dev->bulk_in_buffer,

              min(dev->bulk_in_size, count),

              &count, HZ*10);

/* if the read was successful, copy the data to user
space */

if (!retval) {

    if (copy_to_user(buffer, dev->bulk_in_buffer, count))

        retval = -EFAULT;

    else

        retval = count;

}
```

此示例显示了从IN端点进行的简单批量读取。如果读取成功，数据将被复制到用户空间。这通常在USB驱动程序的read函数中完成。

`usb_bulk_msg` 函数不能在中断上下文中调用，也不能在持有自旋锁的情况下调用。此外，此函数不能被任何其他函数取消，所以在使用它时要小心；确保你的驱动程序的disconnect知道在允许自己从内存中卸载之前，等待调用完成。

## 13.5.2. usb_control_msg 接口

usb_control_msg函数的工作方式与usb_bulk_msg函数类似，只是它允许驱动程序发送和接收USB控制消息：

```c
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);
```

此函数的参数几乎与usb_bulk_msg相同，但有几个重要的区别：

`struct usb_device *dev` 指向要发送控制消息的USB设备的指针。

`unsigned int pipe` 要发送此控制消息的USB设备的特定端点。此值是通过调用usb_sndctrlpipe或usb_rcvctrlpipe创建的。

`__u8 request` 控制消息的USB请求值。

`__u8 requesttype` 控制消息的USB请求类型值。

`__u16 value` 控制消息的USB消息值。

`__u16 index` 控制消息的USB消息索引值。

`void *data` 如果这是一个OUT端点，这是指向要发送到设备的数据的指针。如果这是一个IN端点，这是一个指向数据应放置的位置的指针，数据从设备读取后放置在此处。

`__u16 size` 由数据参数指向的缓冲区的大小。

`int timeout` 应等待的时间（以jiffies为单位）在超时之前。如果此值为0，函数将永远等待消息完成。

如果函数成功，它返回传输到设备或从设备传输的字节数。如果不成功，它返回一个负的错误号。

参数`request`、`requesttype`、`value`和`index`都直接映射到USB规范中定义USB控制消息的方式。有关这些参数的有效值以及它们如何使用的更多信息，请参阅USB规范的第9章。

像`usb_bulk_msg`函数一样，`usb_control_msg`函数不能在中断上下文中调用，也不能在持有自旋锁的情况下调用。此外，此函数不能被任何其他函数取消，所以在使用它时要小心；确保你的驱动程序的`disconnect`函数知道在允许自己从内存中卸载之前，等待调用完成。

### 13.5.3. 其他使用 USB 数据函数

USB核心中有一些辅助函数可以用来从所有USB设备中检索标准信息。这些函数不能在中断上下文中调用，也不能在持有自旋锁的情况下调用。

函数usb_get_descriptor从指定设备检索指定的USB描述符。函数定义如下：

```c
int usb_get_descriptor(struct usb_device *dev, unsigned
char type,
                       unsigned char index, void *buf,
int size);
```

此函数可以由USB驱动程序用来从struct usb_device结构中检索任何设备描述符，这些描述符在现有的struct usb_device和struct usb_interface结构中尚不存在，例如音频描述符或其他类特定信息。函数的参数为：

`struct usb_device *usb_dev` 指向应从中检索描述符的USB设备的指针。

`unsigned char type` 描述符类型。此类型在USB规范中有描述，可以是以下类型之一：

```c
USB_DT_DEVICE USB_DT_CONFIG USB_DT_STRING
USB_DT_INTERFACE USB_DT_ENDPOINT USB_DT_DEVICE_QUALIFIER
USB_DT_OTHER_SPEED_CONFIG USB_DT_INTERFACE_POWER
USB_DT_OTG USB_DT_DEBUG USB_DT_INTERFACE_ASSOCIATION
USB_DT_CS_DEVICE USB_DT_CS_CONFIG USB_DT_CS_STRING
USB_DT_CS_INTERFACE USB_DT_CS_ENDPOINT
```

`unsigned char index` 应从设备检索的描述符的编号。

`void *buf` 指向将描述符复制到的缓冲区的指针。

`int size` 由buf变量指向的内存的大小。

如果此函数成功，它返回从设备读取的字节数。否则，它返回此函数进行的底层调用`usb_control_msg`返回的负错误号。

`usb_get_descriptor`调用的更常见用途之一是从USB设备检索字符串。因为这很常见，所以有一个名为`usb_get_string`的辅助函数：

```c
int usb_get_string(struct usb_device *dev, unsigned short
langid,

                   unsigned char index, void *buf, int
size);
```

如果成功，此函数返回设备接收的字符串的字节数。否则，它返回此函数进行的底层调用usb_control_msg返回的负错误号。

如果此函数成功，它在由buf参数指向的缓冲区中返回以UTF-16LE格式（Unicode，每个字符16位，以小端字节顺序）编码的字符串。由于这种格式通常不是很有用，所以有另一个函数，叫做`usb_string`，它返回从USB设备读取的字符串，并已经转换成ISO 8859-1格式的字符串。这种字符集是Unicode的8位子集，是英语和其他西欧语言的字符串中最常见的格式。由于这通常是USB设备的字符串的格式，所以建议使用`usb_string`函数而不是`usb_get_string`函数。

## 13.6. 快速参考

本节总结本章介绍的符号:

```
#include <linux/usb.h>
```

所有和 USB 相关的头文件. 它必须被所有的 USB 设备驱动包含.

```
struct usb_driver;
```

描述 USB 驱动的结构.

```
struct usb_device_id;
```

描述这个驱动支持的 USB 设备的结构.

```
int usb_register(struct usb_driver *d);
```

用来从USB核心注册和注销一个 USB 驱动的函数.

```
struct usb_device *interface_to_usbdev(struct
usb_interface *intf);
```

从 struct usb_interface 获取控制 struct usb_device *.

```
struct usb_device;
```

控制完整 USB 设备的结构.

```
struct usb_interface;
```

主 USB 设备结构, 所有的 USB 驱动用来和 USB 核心通讯的.

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

设置和获取在 struct usb_interface 中的私有数据指针部分的函数.

```
struct usb_class_driver;
```

描述 USB 驱动的一个结构, 这个驱动要使用 USB 主编号来和用户空间程序通讯.

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

用来注册和注销一个特定 struct usb_interface * 结构到 struct usb_class_driver 结构的函数.

```
struct urb;
```

描述一个 USB 数据传输的结构.

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

用来创建和销毁一个 struct usb urb*的函数.

```
int usb_submit_urb(struct urb *urb, int mem_flags);
int usb_kill_urb(struct urb *urb);
int usb_unlink_urb(struct urb *urb);
```

用来启动和停止一个 USB 数据传输的函数.

```
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);
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);
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);
```

用来在被提交给 USB 核心之前初始化一个 struct urb 的函数.

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

用来发送和接受 USB 数据的函数, 不必使用一个 struct urb.