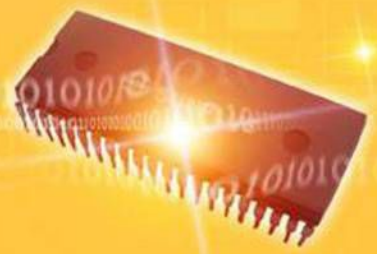
嵌入式系统工程师



USB驱动开发

大纲

- USB概况
- USB拓扑结构
- USB电气特性
- USB数据传输过程
- USB设备描述符及枚举过程
- Linux USB驱动开发

大纲

- USB概况
- USB拓扑结构
- USB电气特性
- USB数据传输过程
- USB设备描述符及枚举过程
- Linux USB驱动开发

USB概况

- USB(Universal Serial BUS)是“通用串行总线”英文的缩写。
- USB是在1994年底由英特尔、康柏、IBM、Microsoft等多家公司联合提出的。
- USB是一个使计算机周边设备连接标准化、单一化的接口 。
- 广泛应用于日益增多的外围设备，如键盘、鼠标、调制解调器、打印机、网卡、显示器以及各种PDA等手持设备。

USB概况

- USB的出现：
  - 1994.11.11—USB v0.7：该版本推出响应不大
  - 1996年—USB(LowSpeed) v1.0：
    1.5Mbps
  - 1998年—USB(FullSpeed) v1.1：
    12Mbps
  - 2004.04—USB(HighSpeed) v2.0：
    480Mbps I<500mA
  - 2008年上半年—USB(SuperSpeed) v3.0：
    5Gbps I<900mA
- 没有USB，就不会有移动数据业务如此的迅猛发展。

大纲

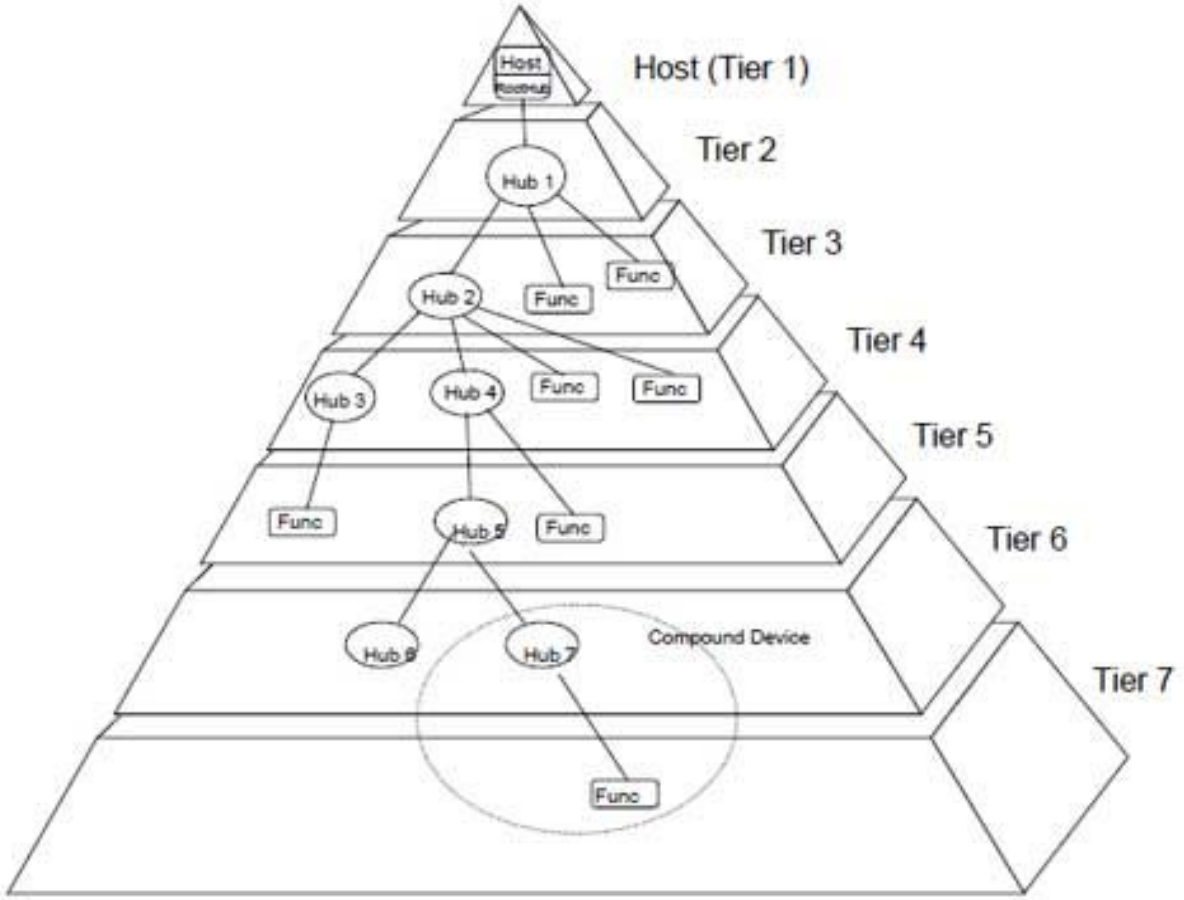
- USB概况
- USB拓扑结构
- USB电气特性
- USB数据传输过程
- USB设备描述符及枚举过程
- Linux USB驱动开发

USB拓扑结构

- USB是一种主从结构的系统。
  - USB主机由USB主控制器(Host Controller)和根集线器(Root Hub)构成
    - USB主控制器：主要负责数据处理。
    - 根集线器：提供一个连接主控制器与设备之间的接口和通路。
  - USB从机可以是各种USB设备也可以是集线器
- USB集线器(USB Hub)：
  - 对原有的USB接口数量进行扩展，以获得更多的USB设备接口，但不能扩展出更多的带宽。

USB拓扑结构

- USB的金字塔拓扑结构图：



USB拓扑结构

- 集线器(USB Hub)本身就是一个USB设备。
- USB协议对集线器的层数限制：
  - USB1.1规定最多为4层(不包括Host)
  - USB2.0规定最多为6层(不包括Host)
- USB设备由7bit二进制表示，所以一个USB主控制器理论上可以接127个设备，但通常0地址被保留给未初始化的设备使用。

USB拓扑结构

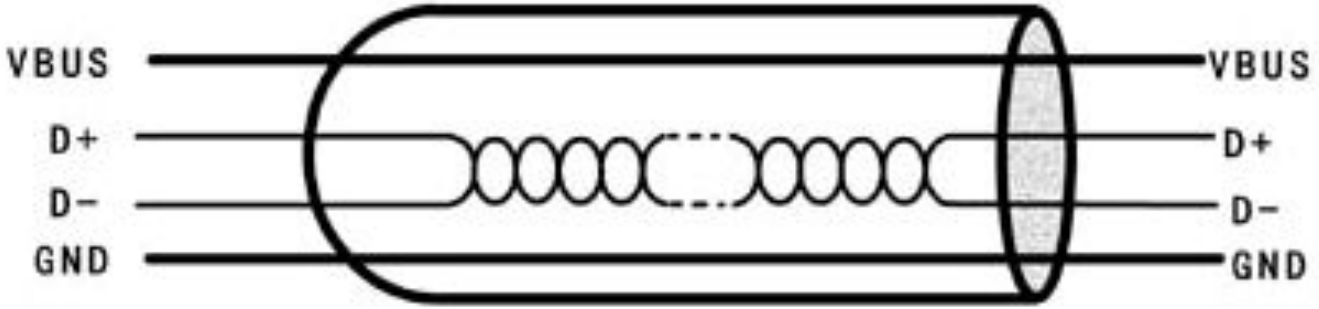
- USB数据交换过程：
  - 数据交换不能发生在主机与主机之间或设备与设备之间，只能发生在主机与设备之间。
- USB On-The-Go
  - USB On-The-Go 是USB协议的补充版本，是USB主机嵌入式化的一种实现，可翻译为“便携式USB”或“移动USB”简记成USB OTG
  - USB是主从模式，设备与设备之间、主机与主机之间不能互连，为了解决这个问题，扩大USB的使用范围，就出现了USB OTG，这样，同一个USB设备在不同的场合下可以在主机和设备（从机）之间进行任意切换。

大纲

- USB概况
- USB拓扑结构
- USB电气特性
- USB数据传输过程
- USB设备描述符及枚举过程
- Linux USB驱动开发

USB电气特性

- USB的连线
  - 标准USB连线使用4芯电缆：5V电源线(V_{BUS})、差分数据线负(D-)、差分数据线正(D+)及地线(GND)。



USB电气特性

- USB的数据传输
  - USB2.0支持3种传输速度：低速(1.5Mbps)、全速(12Mbps)、高速(480Mbps)三种模式。
  - USB低速和全速模式中，采用的是电压传输模式；而在高速模式下，则是电流传输模式。
  - USB实际传输速率比理论值要低，因为有很多协议开销，例如同步、令牌、校验、位填充和包间隙等。
  - USB使用的是NRZI(非归零反相编码)编码方式。具体的数据传输任务由USB控制器实现，用户不必关心

USB电气特性

- USB的插拔检测机制
  - 在USB集线器的每个下游端口D+和D-上，都分别接有一个15K左右的下拉电阻。
  - 对于USB设备来说，正好接有一个1.5K左右的上接电阻：
    - 低速设备，接在端口的D-上。
    - 全速和高速设备，接在端口的D+上
  - 当设备与集线器端口连接时，集线器端口的下拉低电平会由于上拉分压，而变成高电平，从而识别出USB的插拔动作。

大纲

- USB概况
- USB拓扑结构
- USB电气特性
- USB数据传输过程
- USB设备描述符及枚举过程
- Linux USB驱动开发

USB包结构及传输过程

- USB的四种传输类型
  - USB协议规定了4种传输类型：批量传输、同步传输(或等时传输)、中断传输、控制传输。
  - 其中批量、同步、中断三种传输中，每完整传输一次数据都称做一个事务。
  - 控制传输包括三个过程：建立过程和状态过程分别是一个事务，数据过程则可能包含多个事务。

USB包结构及传输过程

- 批量传输：
  - 一次批量事务有三个阶段：

上图：一次正确的批量输入事务

上图：一次正确的批量输出事务

USB包结构及传输过程

- 中断传输：
  - 这里所说的中断，不同于硬件中断，它不是由设备主动地发出一个中断请，而是由主机保证在不大于某个时间间隔内安排一次传输。
  - 一般用于对时间要求较严格的设备中，如大多数HID设备，也可以作为批量传输的状态检测。
  - 中断传输与批量传输的结构基本一样。

USB包结构及传输过程

- 同步传输：
  - 用于数据量大，且时实性要求高，能容忍少量的数据错误的场合，如音视频设备等，并有可能因此占用批量传输等的带宽，而优先满足同步传输的需要。
  - 由于不保证数据100%正确，所以数据错误时，并不进行重传，也没有应答包，如果CRC等错误，由软件来决定怎样处理。

USB包结构及传输过程

- 控制传输：
  - 建立过程：使用一个建立事务，SETUP+DATA0+ACK，建立过程实际就是输出数据。
  - 数据过程：可以没有数据过程，也可以有多笔数据事务，由建立过程中的设置命令决定的

大纲

- USB概况
- USB拓扑结构
- USB电气特性
- USB数据传输过程
- USB设备描述符及枚举过程
- Linux USB驱动开发

USB设备描述符

- 描述符概念
  - 前面所描述的USB总线结构，只是一个数据通路，我们还需要一些描述设备特性的东西来代表和区分不同的USB设备，这就是描述符。
  - 描述符：描述了设备的各种行为和具体参数类型等，让主机明确应该加载什么样的驱动程序与设备之间进行怎样的操作。
- USB2.0协议定义的描述符
  - 设备描述符(Device Descriptor)
  - 配置描述符(Configuration Descriptor)

USB设备描述符

- 接口描述符(Interface Descriptor)
- 端点描述符(Endpoint Descriptor)
- 字符串描述符(String Descriptor)
- Qualifier Descriptor
- Other Speed Configuration Descriptor
- interface_power
- 类特殊描述符：
  - HID描述符
  - 音频接口描述符
  - 厂商自定义描述符

USB设备描述符

- USB描述符介绍
  - 设备描述符：
    - 设备所使用的USB协议版本号、设备类型、端点0的最大包大小、厂商ID(VID)和产品ID(PID)、设备版本号、厂商字符串索引、产品字符串索引、设备序列号索引、可能的配置数量等。
  - 配置描述符：
    - 反映设备对主机的配置需求，包含配置的编号、供电方式、接口数、是否支持远程唤醒、电流需求量等。

USB设备描述符

- 接口描述符：
  - 接口的编号、接口的端点数、接口所使用的类、子类、协议等。
- 端点描述符：
  - 端点号及方向、端点的传输类型(控制、同步、批量、中断传输)、最大包长度、查寻时间间隔等。
- 字符串描述符：
  - 不是必需的，采用unicode编码，主要是提供一些方便人们阅读的厂商或设备名称等文字信息。

USB设备描述符

- USB描述符的分析及模型
  - 配置和接口是为了更方便地管理端点而抽象出来的概念：
    - 不同的配置可以使设备发挥不同的功能。
    - 不同的接口可以实现功能的复用。
    - 非0端点只有在配置之后才能使用。
    - 端点0作为默认控制管道的端点，在设备连接、上电和收到总线复位信号时就可以访问0端点。

USB设备的枚举

- USB描述符介绍
- 枚举就是通过控制传输从设备读取各种描述符信息的过程。之后主机会据此加载合适的驱动程序。
- USB设备的枚举过程
  - USB主机检测到USB设备的插入(主机通过查询集线器端口确定变化的类型)
  - 主机等待100ms让设备的电源变得稳定,然后对设备进行复位使设备拥用0地址和0端点，使设备处于默认状态

USB设备的枚举

- 获取设备描述符，记录端点0的最大包大小
- 再次对设备进行复位，通过默认通道发出输出令牌配置设备地址
- 主机发出输入令牌，设备成功确认回复则启用新的地址
- 再次通过新地址，发送多次输入令牌，获取完整的设备描述符
- 获取配置描述符，以及其它描述符集合
- 单独获取字符串、报告等描述符
- 最后主机给设备分配一个配置，设备处于配置状态

大纲

- USB概况
- USB拓扑结构
- USB电气特性
- USB数据传输过程
- USB设备描述符及枚举过程
- Linux USB驱动开发
  - Linux USB驱动概述
  - 驱动框架
  - USB请求块——URB

USB驱动概述

- Linux中的USB驱动分两种类型：
  - USB设备驱动程序(USB device drivers)：控制器端驱动，控制插入其中的USB设备
  - USB器件驱动程序(USB gadget drivers)：设备端驱动，控制该设备如何作为一个USB设备和主机通信
- 对于USB设备端驱动一般都固化在USB设备中，由制造USB设备的厂家提供。我们这里主要探讨USB主控制器端的驱动。

USB驱动概述

- USB主控制器功能及分类
  - 主机控制器负责处理主机与设备之间电气和协议层的互连，根集线器提供USB设备连接点。
  - USB系统使用USB主控制器来管理主机和USB设备之间的数据传输，并且管理着USB相关资源，如带宽等。
  - 为了实现USB主机功能统一，提高系统的可靠性与可移植性，上游芯片厂商确定了相应的主机规范，用的比较广泛的有如下四种：

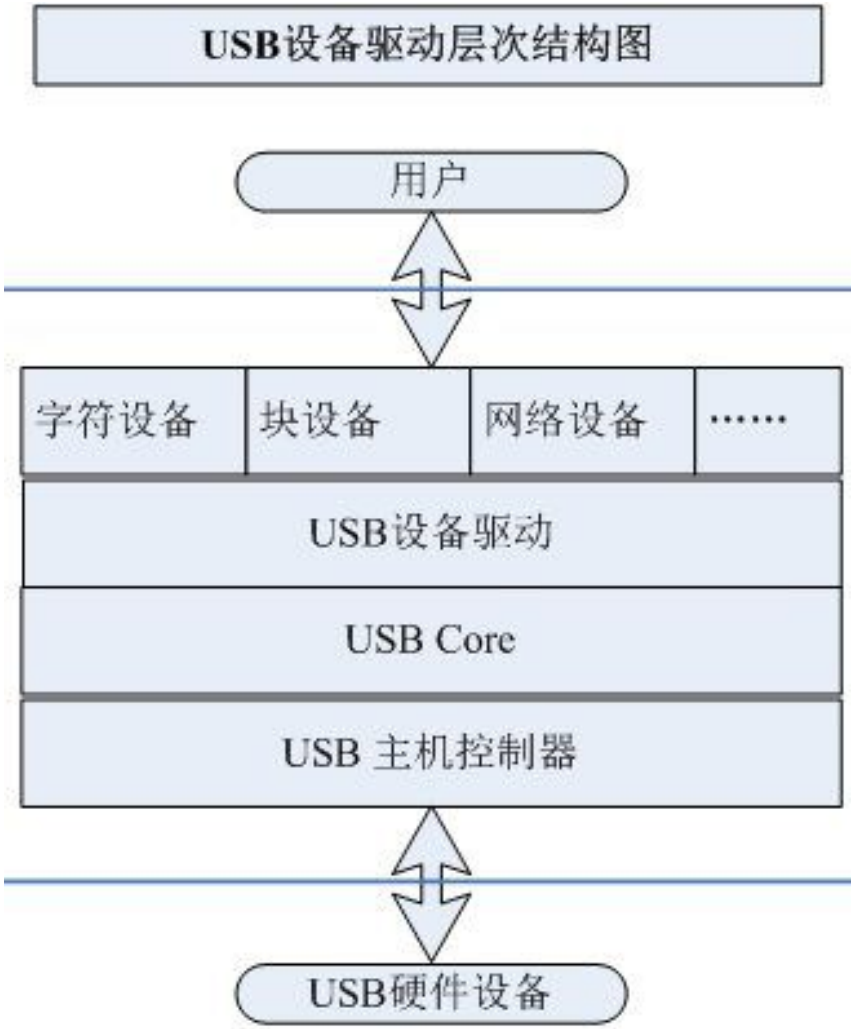
USB驱动概述

- UHCI(Universal host control interface 通用主机控制器接口)：intel推出的，用于全速与低速USB系统中，常用于PC机。
- OHCI（open host controller interface开放主机控制器接口）：由Compaq、Microsoft等公司推出，硬件比UHCI智能，所以HCD（主机控制器驱动）更简单
- EHCI(Enhanced host control interface 增强主机控制接口)：支持高速的USB2.0设备，为支持低速USB设备通常包含UHCI和OHCI

USB驱动概述

- USB OTG控制器：在嵌入式领域越来越受欢迎，设备可以根据功能需要在主机模式和设备模式之间任意切换
- linux系统实现了几类通用的USB设备驱动，可划分为如下几类：
  - 音频设备类
  - 通信设备类
  - HID（人机接口）设备类
  - mass storage（人机接口）设备类
  - 显示设备类

  ....


USB设备驱动层次结构图
用户
字符设备
块设备
网络设备
……
USB设备驱动
USB Core
USB 主机控制器
USB硬件设备

大纲

- USB概况
- USB拓扑结构
- USB电气特性
- USB数据传输过程
- USB设备描述符及枚举过程
- Linux USB驱动开发
  - Linux USB驱动概述
  - 驱动框架
  - USB请求块——URB

USB设备驱动架构

- 在linux中，每个USB驱动程序也同样以模块的形式进行组织，这样做的好处是方便我们调试模块，同时增加驱动程序独立性
- 每个模块同样都是以：
  - module_init()注册驱动
  - module_exit()注销驱动
- 我们以usb_skel驱动为例进行USB设备驱动框架的讲解（usb_skel为内核提供的usb设备驱动的参考模板）：

USB设备驱动架构

```
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;
}

static void __exit usb_skel_exit(void)
{
        /* deregister this driver with the USB subsystem */
        usb_deregister(&skel_driver);
}
```

USB设备驱动架构

```
static struct usb_driver skel_driver = {
        .name =         "skeleton",
        .probe =  skel_probe,
        .disconnect =  skel_disconnect,
        .suspend =     skel_suspend,
        .resume =      skel_resume,
        .pre_reset =   skel_pre_reset,
        .post_reset =  skel_post_reset,
        .id_table =    skel_table,
        .supports_autosuspend = 1,
};
```

USB设备驱动架构

```
/* Define these values to match your devices */
#define USB_SKEL_VENDOR_ID 0xfff0
#define USB_SKEL_PRODUCT_ID     0xfff0

/* table of devices that work with this driver */
static const struct usb_device_id skel_table[] = {
     { USB_DEVICE(USB_SKEL_VENDOR_ID, USB_SKEL_PRODUCT_ID) },
     { }                         /* Terminating entry */
};
MODULE_DEVICE_TABLE(usb, skel_table);
```

- 如果插入的USB设备能与skel_table表中的ID号匹配上，则skel_probe函数被回调。

USB设备驱动架构

- USB设备驱动usb_driver结构体的探测函数中,应该完成如下工作:
  1. 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。
  2. 把已初始化数据结构的指针保存到接口设备中,如接口描述信息:
     - usb_set_intfdata()函数可以添加usb_interface的私有数据到device结构中,这个函数的原型为:
     - void usb_set_intfdata (struct usb_interface *intf, void *data);

USB设备驱动架构

- 这个函数的“反函数”还可以从device中获取usb_interface的私有数据,其原型为:
- void *usb_get_intfdata (struct usb_interface *intf);

3. 注册USB设备

- 如果是字符设备,调用usb_register_dev这个函数进行注册。
- 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file_operations成员函数。
- 如果是其他类型的设备,如输入子系统(input_register_device)、Block设备、网络设备等,则调用对应设备的注册函数。

USB设备驱动架构

- USB设备驱动usb_driver结构体的断开函数中,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。
- 对探测函数probe()的调用发生在USB设备被USB核心检测到该驱动程序与安装的USB设备对应的id_table成员匹配时。
- 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。

USB设备驱动架构

- 一般USB驱动程序包括三个部分：
  - 以usb-skeleton为例分别是:USB驱动程序skel_driver、设备类驱动程序skel_class和设备操作函数skel_fops
  - skel_driver描述了主机控制器如何探测设备、断开设备及得到设备信息
  - skel_class包括有设备的操作函数集和设备访问权限

USB设备驱动架构

- skel_fops描述了对设备文件的读写操作方法(取决于对设备的封装)
- 整个驱动程序就是实现这几个结构中函数。所有这些函数的实现都通过调用USB驱动核心层中的接口完成

大纲

- USB概况
- USB拓扑结构
- USB电气特性
- USB数据传输过程
- USB设备描述符及枚举过程
- Linux USB驱动开发
  - Linux USB驱动概述
  - 驱动框架
  - USB请求块——URB

USB请求块—URB

- 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sk_buff结构体。
- URB数据结构定义如下：

USB请求块—URB

```
struct urb {
    /* private: usb core and host controller only fields in the urb */
    struct kref kref;           /* reference count of the URB */
    void *hcpriv;               /* private data for host controller */
    atomic_t use_count;             /* concurrent submissions counter */
    atomic_t reject;            /* submissions will fail */
    int unlinked;               /* unlink error code */

    /* public: documented fields in the urb that can be used by drivers */
    struct list_head urb_list;      /* list head for use by the urb's
                                 * current owner */
    struct list_head anchor_list;       /* the URB may be anchored */
    struct usb_anchor *anchor;
    struct usb_device *dev;             /* (in) pointer to associated device */
    struct usb_host_endpoint *ep;       /* (internal) pointer to endpoint */
    unsigned int pipe;          /* (in) pipe information */
    unsigned int stream_id;             /* (in) stream ID */
    int status;                 /* (return) non-ISO status */
    unsigned int transfer_flags;        /* (in) URB_SHORT_NOT_OK | ...*/
    void *transfer_buffer;          /* (in) associated data buffer */
    dma_addr_t transfer_dma;    /* (in) dma addr for transfer_buffer */
    struct scatterlist *sg;             /* (in) scatter gather buffer list */
```

USB请求块—URB

```
    int num_sgs;            /* (in) number of entries in the sg list */
    u32 transfer_buffer_length;     /* (in) data buffer length */
    u32 actual_length;          /* (return) actual transfer length */
    unsigned char *setup_packet;    /* (in) setup packet (control only) */
    dma_addr_t setup_dma;           /* (in) dma addr for setup_packet */
    int start_frame;        /* (modify) start frame (ISO) */
    int number_of_packets;          /* (in) number of ISO packets */
    int interval;           /* (modify) transfer interval
                     * (INT/ISO) */
    int error_count;        /* (return) number of ISO errors */
    void *context;              /* (in) context for completion */
    usb_complete_t complete;    /* (in) completion routine */
    struct usb_iso_packet_descriptor iso_frame_desc[0];
                    /* (in) ISO ONLY */
} ? end urb ? ;
```

USB请求块—URB

- 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处理流程：
  - 创建URB
  - 初始化URB
  - 提交URB到USB核心
  - URB处理完成
- URB通常被一个USB设备驱动创建，创建URB结构体的函数原形：

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

USB请求块—URB

- iso_packets:是当前URB应当包含的等时数据包数目，用于同步传输，为0表示不创建等时数据包。
- mem_flag：分配内存的标志，和kmalloc()函数的分配标志参数含义相同，失败返回0。
- usb_alloc_urb()反函数为：

```
void usb_free_urb(struct urb *urb)
{
    if (urb)
        kref_put(&urb->kref, urb_destroy);
}
```

USB请求块—URB

- URB初始化，目的是将URB安排给一个特定的USB设备端点，根据端点类型的不同又分成4种不同的初始化方法
- 中断URB初始化函数原形：

```
static inline 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_fn,
                                    void *context,
                                    int interval)
```

USB请求块—URB

- urb参数指向被初始化的URB指针
- dev参数指向这个URB要被发送到的USB设备
- pipe参数是这个URB将要被发送到的USB设备端点
- transfer_buffer参数指向收发数据绥冲区
- buffer_length参数表示绥冲区大小
- complete_fn参数指向URB处理完成函数
- context参数是处理“上下文”私有数据结构
- interval参数表示URB被轮询调度的间隔
- pipe采用usb_sndintpipe()或usb_rcvintpipe()函数从设备获取。

USB请求块—URB

- 批量URB初始化函数原形：

```
static inline 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_fn,
                                     void *context)
```

- 控制URB初始化函数原形：

```
static inline 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_fn,
                                        void *context)
```

USB请求块—URB

- 同步URB初始化，不同于前面三种URB有专门的初始化函数，而是需要手动初始化URB，初始化的例子可以参考：

  drivers\media\video\usbvideo\usbvideo.c

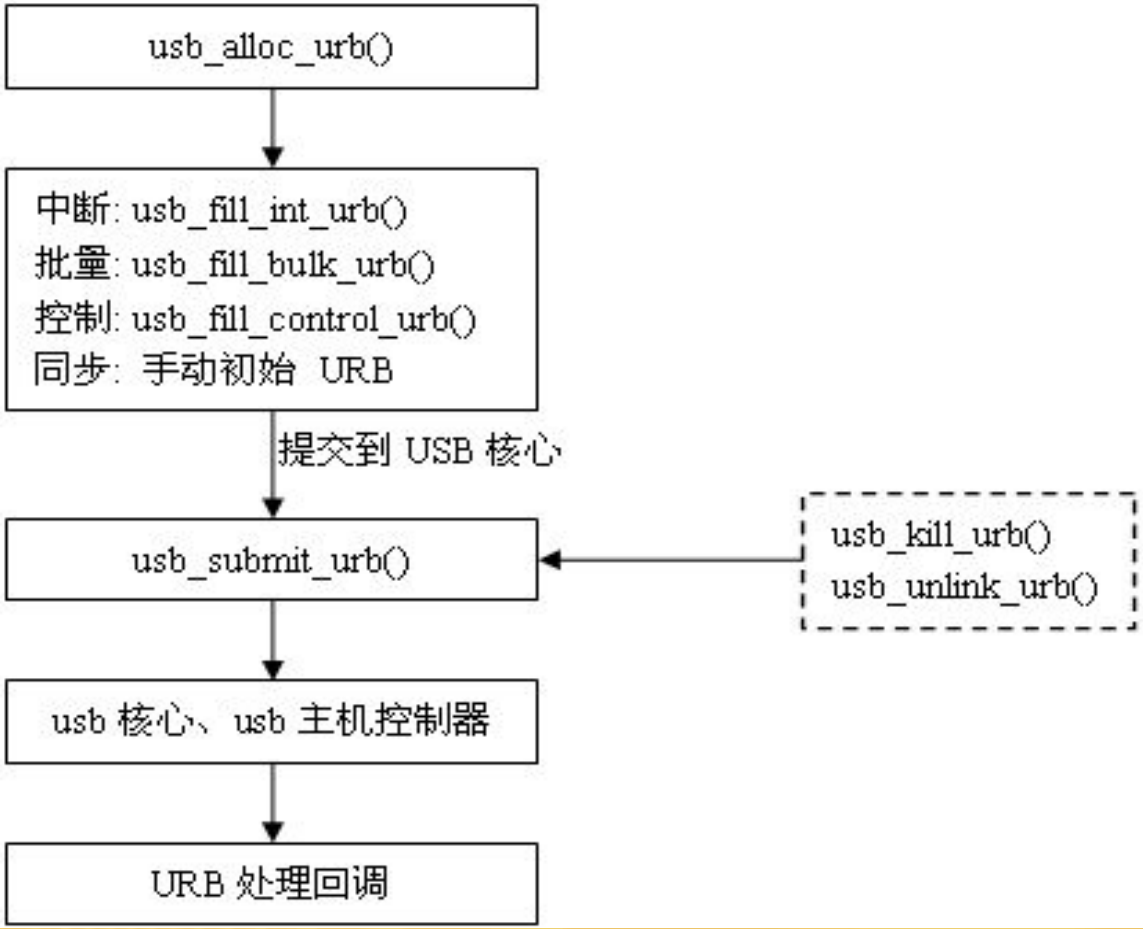
- 提交USB设备驱动到USB核心：
  - 提交过程通过usb_submit_urb()完成：

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

移交成功则返回0，否则返回错误号

USB键盘驱动实例

➢ URB处理流程总结：



USB键盘驱动实例

- 下面我们来一起分析一下linux内核中通用的USB键盘驱动实例
- 驱动文件在2.6.35.7内核中的路径为
  /drivers/hid/usbhid/usbkbd.c
  /drivers/hid/usbhid/usbmouse.c

凌阳教育官方微信：Sunplusedu

Tel: 400-705-9680 , BBS: www.51develop.net , QQ群：241275518