iTOP-4412-驱动-USB 请求块 urb

在前一篇文档中，我们介绍了 USB 的描述符。对于具体设备驱动的 USB 描述符的配置信息，是存储在 USB 的设备中的，它相当于“固件”，当 USB 主控制器检测到有 USB 设备接入的时候，主控制器会将“固件”的信息读出来。

这个“固件”，相当于字符设备中的“设备注册”部分，在字符设备中，我们需要在平台文件中注册设备，而在 USB 设备驱动中，我们可以这样理解“USB **设备驱动**的**设备注册**是在**USB 设备**中注册的”，我们不需要在平台文件中做任何注册设备的操作。

在介绍 USB 请求块之前，我们回忆下 i2c 和 spi 驱动。在 i2c 驱动中，我们在使用 i2c_transfer 函数通信前，定义结构体 i2c_msg 变量；在 spi 驱动中，我们在使用 spi_async 函数通信前,定义结构体 spi_message 变量。那么在 USB 通信中，要使用 USB 总线通信前，需要申请一个结构体变量，这个结构体叫 urb（USB 请求块），这个结构体是 USB 通信前需要申请和初始化配置的结构体变量。

本文档介绍的 USB 请求块 urb，它是 USB 主控制器和设备通信的数据结构，主机和设备之间需要通过 urb 进行数据传输。当主控制器需要和设备通信的时候，只需要申请 urb 结构，完成 urb 的配置，最后将 urb 提交给 USB 核心层，由核心层对其进行处理。设备驱动将 urb 提交给 USB 核心层之后，urb 的通信是不受 USB 驱动控制的，只是在 urb 通信完成之后，通知 USB 的设备驱动。

1 urb 结构体分析

请注意，这个 urb 结构体非常重要，是理解 USB 驱动的关键点，也是难点，我们会在后面分析具体驱动代码的时候，和这部分对应着分析。

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。这个请求块用 struct urb 结构描述并且可在“include/linux/usb.h”中找到

```
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 };
```

下面详细介绍这个结构体变量中最重要的一些成员。

13 行，struct usb_device *dev

指向这个 urb 要发送的 struct usb_device 的指针，在 urb 发送到 usb 核心之前，这个变量必须被 usb 驱动初始化。

14 行，unsigned int pipe

可以称之为管道，也可以称为端点消息，是给这个 urb 发送到的特定 struct usb_device，同理，在 urb 发送到 usb 核心之前，这个变量必须被 usb 驱动初始化。下面是

8 种 urb，我们后面介绍的 USB 鼠标驱动，是属于第 6 种 usb_rcvctrlpipe，一个中断 IN 的端点，对于鼠标来说，是输入，所以是 IN 端点，鼠标点击的时候会产生中断，检测到中断再获取数据，所以是中断，综合起来就是中断 IN。

```
unsigned int usb_sndctrlpipe(struct usb_device *dev, unsigned int endpoint)//指定一个控制 OUT 端点给特定的带有特定端点号的 USB 设备.
unsigned int usb_rcvctrlpipe(struct usb_device *dev, unsigned int endpoint)//指定一个控制 IN 端点给带有特定端点号的特定 USB 设备.
unsigned int usb_sndbulkpipe(struct usb_device *dev, unsigned int endpoint)//指定一个块 OUT 端点给带有特定端点号的特定 USB 设备
unsigned int usb_rcvbulkpipe(struct usb_device *dev, unsigned int endpoint)//指定一个块 IN 端点给带有特定端点号的特定 USB 设备
unsigned int usb_sndintpipe(struct usb_device *dev, unsigned int endpoint)//指定一个中断 OUT 端点给带有特定端点号的特定 USB 设备
unsigned int usb_rcvintpipe(struct usb_device *dev, unsigned int endpoint)//指定一个中断 IN 端点给带有特定端点号的特定 USB 设备
unsigned int usb_sndisocpipe(struct usb_device *dev, unsigned int endpoint)//指定一个同步 OUT 端点给带有特定端点号的特定 USB 设备
unsigned int usb_rcvisocpipe(struct usb_device *dev, unsigned int endpoint)//指定一个同步 IN 端点给带有特定端点号的特定 USB 设备
```

15 行，int status

当这个 urb 被结束,或者开始由 USB 核心处理,这个变量被设置为 urb 的当前状态。一个 USB 驱动可安全存取这个变量的唯一时间是在 urb 完成处理者函数中。这个限制是阻止竞争情况,发生在这个 urb 被 USB 核心处理当中。对于同步 urb,在这个变量中的一个成功的值(0)只指示是否这个 urb 已被去链，为获得在同步 urb 上的详细状态,应当检查 iso_frame_desc 变量。

这个变量的有效值如下所示。

```
0//这个 urb 传送是成功的.
-ENOENT//这个 urb 被对 usb_kill_urb 的调用停止。
-ECONNRESET//urb 被对 usb_unlink_urb 的调用去链,并且 transfer_flags 变量被设置为 urb_ASYNC_UNLINK
-EINPROGRESS
-EPROTO
-EILSEQ//在这个 urb 传送中有一个 CRC 不匹配
-EPIPE
-ECOMM//在传送中数据接收快于能被写入系统内存,这个错误值只对 IN urb。
```

```
-ENOSR
-EOVERFLOW
-EREMOTEIO
-ENODEV//这个 USB 设备现在从系统中消失
-EXDEV
-EINVAL
-ESHUTDOWN
```

16 行，transfer_flags

这个变量可被设置为不同的位值，根据这个位，usb 驱动可以设置 urb 传输的状态。

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。

17 行，void *transfer_buffer

指向缓冲区的指针，这个指针可以是 OUT urb 或者是 IN urb。主机控制器为了正确使用这个缓冲区，必须使用 kmalloc 调用来创建它，而不是堆栈或者静态数据区。对于控制端点，这个缓冲是给发送的数据。

18 行，dma_addr_t transfer_dma

用来使用 DMA 传送数据到 usb 设备的缓冲。

19 行，u32 transfer_buffer_length

缓冲区的长度，由于 urb 只会使用 buffer 或者 dma 其中一种来传输数据，所以这个长度可以被它们共用。如果这是 0，表示没有传送缓冲被 usb 核心所使用。

21 行，u32 actual_length

当 urb 完成处理后，这个变量被设置为数据的真实长度，或者由这个 urb（OUT）发送，或者由这个 urb（IN）接受。对于 IN urb，这个值必须被用来替代 taansfer_buffer_length，因为接收的数据可能比整个缓冲区的大小小。

22 行，unsigned char *setup_packet

指向一个控制 urb 的 setup 报文的指针，在位于正常传送缓冲的数据之前被传送，这个变量只对控制 urb 有效。

23 行，dma_addr_t setup_dma

给控制 urb 的 setupt 报文的 DMA 缓冲.，在位于正常传送缓冲的数据之前被传送，这个变量只对控制 urb 有效。

24 行，int start_frame

设置或者返回同步传送要使用的初始帧号。

26 行，int interval

只对同步 urb 有效,并且指定这个 urb 要处理的同步传送缓冲的编号。在这个 urb 发送给 USB 核心之前,这个值必须被 USB 驱动设置给同步 urb。

27 行，int error_count

被 USB 核心设置,只给同步 urb 在它们完成之后。它指定报告任何类型错误的同步传送的号码。

28 行，void *context

指向数据点的指针,它可被 USB 驱动设置。它可在完成处理者中使用当 urb 被返回到驱动，可能是在回调函数使用的数据。

29 行，usb_complete_t complete

指向完成处理者函数的指针,它被 USB 核心调用当这个 urb 被完全传送或者当 urb 发生一个错误。在这个函数中,USB 驱动可检查这个 urb,释放它,或者重新提交它给另一次传送。

30 行，struct usb_iso_packet_descriptor iso_frame_desc[0];

只对同步 urb 有效。这个变量是组成这个 urb 的一个 struct usb_iso_packet_descriptor 结构数组。这个结构允许单个 urb 来一次定义多个同步传送。它也用来收集每个单独传送的传送状态。

2 urb 处理流程

在 USB 驱动中，进行初始化部分是围绕 usb 描述符，usb 通信的核心代码都是围绕着 urb。USB 通信的整个通信流程，在初始化进入 probe 之后，都是以 urb 为核心，urb 的处理流程为：创建 urb、初始化 urb（中断 urb，块 urb，控制 urb，同步 urb）、提交 urb、完成 urb、销毁 urb。另外，在 urb 初始化的时候还是会用到 usb 描述符，usb 描述符中有一些配置信息，需要初始化到 urb 结构体中。

2.1 urb 的创建

在介绍 urb 的创建前，回忆一下 i2c 和 spi 的驱动，在 i2c 驱动中，我们在使用 i2c_transfer 函数通信前，定义结构体 i2c_msg 变量；在 spi 驱动中，我们在使用 spi_async 函数通信前,定义结构体 spi_message 变量。例如,i2c 驱动中,要写数据前，我们会做如下处理。

```
struct i2c_msg msgs[1];
...
msgs[0].addr = 0x38;
msgs[0].flags = 0;
msgs[0].len        = 2;
msgs[0].buf        = buf1;
..
ret = i2c_transfer(this_client->adapter, msgs,1);
```

在上面的代码中，直接定义 i2c_msg 结构体变，但是在 urb 的创建中，是绝对不允许出现类似下面的代码：

struct urb my_urb;

my_urb.xx = xx;

urb 的创建必须使用函数 usb_alloc_urb 来创建，因为这样可能会破坏 urb 使用的引用计数方法。

所以创建 urb 的代码应该类似下面的代码。

struct urb my_urb;

my_urb = usb_alloc_urb(xxx)；

说白了，就是在创建 urb 的时候，必须使用内核提供的函数，不允许用其它的方式来创建，另外需要注意的是，当使用 usb_alloc_urb 创建 urb 之后，是可以具体的进行初始化的。

函数 usb_alloc_urb 的原型为：

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

参数 iso_packet，是这个 urb 应当包含的同步报文的数目。如果你不想创建一个同步 urb，这个变量应当被设置为 0。

参数 mem_flags，是和传递给 kmalloc 函数调用来从内核分配内存的相同的标志类型,在驱动中一般使用 GFP_KERNEL。

返回值,如果这个函数在分配足够内存给这个 urb 成功,一个指向 urb 的指针被返回给调用者。如果返回值是 NULL，某个错误在 USB 核心中发生了,并且驱动需要正确地清理。

我们都知道，常见的 USB 设备有鼠标、键盘、U 盘、网卡、摄像头等等，在 U 盘和鼠标中要传输的数据量是不可同日而语的，说以在 urb 被提交给 usb 核心前，需要初始化为不同的类型，下一小节我们介绍初始化 urb。

2.2 初始化 urb

协助 urb 初始化的函数有三个，如下所示。

中断 urb　usb_fill_int_urb；

块 urb　　usb_fill_bulk_urb；

控制 urb　usb_fill_control_urb。

中断 urb 初始化一个 urb，发送给 usb 设备的一个中断端点。

```
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)
```

函数中的 8 个参数如下。

struct urb *urb,指向要被初始化的 urb 的指针；

struct usb_device *dev,初始化的 urb 要发送到的 USB 设备；

unsigned int pipe,这个 urb 要被发送到的 USB 设备的特定端点。这个值被创建,使用前面提过的 usb_sndintpipe、usb_rcvintpipe 等函数。

void *transfer_buffer,指向缓冲的指针,从那里外出的数据被获取或者进入数据被接受。注意这不能是一个静态的缓冲并且必须使用 kmalloc 调用来创建.

int buffer_length,缓冲的长度,被 transfer_buffer 指针指向。

usb_complete_t complete_fn,指针,指向当这个 urb 完成时被调用的完成处理者。

void *context,指向数据块的指针,它被添加到这个 urb 结构为以后被完成处理者函数获取。

int interval，这个 urb 应当被调度的间隔。

usb_fill_control_urb 的函数原型为：

```
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)
```

函数参数和 usb_fill_bulk_urb 函数都相同,除了有个新参数,unsigned char *setup_packet，它必须指向要发送给端点的 setup 报文数据。还有，unsigned int pipe 变量必须被初始化，使用对 usb_sndctrlpipe 或者 usb_rcvictrlpipe 函数的调用。

usb_fill_control_urb 函数不设置 transfer_flags 变量在 urb 中，因此任何对这个成员的修改必须游驱动自己完成。大部分驱动不使用这个函数,因为使用在"USB 传送不用 urb"一节中介绍的同步 API 调用更简单。

usb_fill_bulk_urb 的函数原型为：

```
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)
```

这个函数参数和 usb_fill_int_urb 函数的都相同。但是,没有 interval 参数因为 bulk urb 没有间隔值。请注意这个 unsiged int pipe 变量必须被初始化用对 usb_sndbulkpipe 或者 usb_rcvbulkpipe 函数的调用。usb_fill_int_urb 函数不设置 urb 中的 transfer_flags 变量,因此任何对这个成员的修改不得不由这个驱动自己完成。

另外还有一个同步 urb，同步 urb 没有一个象中断，控制，和块 urb 的初始化函数。因此它们必须在驱动中"手动"初始化,在它们可被提交给 USB 核心之前。

2.3 提交 urb

urb 被正确地创建,并且被 USB 驱动初始化,它已准备好被提交给 USB 核心来发送出到 USB 设备。这通过调用函数 usb_submit_urb 实现。

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

参数*urb，是一个指向 urb 的指针,它要被发送到设备。

参数 mem_flags，等同于传递给 kmalloc 调用的同样的参数，并且用来告诉 USB 核心如何及时分配任何内存缓冲在这个时间。

在 urb 被成功提交给 USB 核心之后,应当从不试图存取 urb 结构的任何成员直到完成函数被调用。因为函数 usb_submit_urb 可被在任何时候被调用,mem_flags 变量的指定必须正确，根据 usb_submit_urb 被调用的时间,只有 3 个有效值可用。

GFP_ATOMIC

只要满足以下条件,就应当使用此值。

1 调用者处于一个 urb 结束处理例程,中断处理例程,底半部 tasklet 或者一个定时器回调函数；

2 调用者持有自旋锁或者读写锁。注意如果正持有一个信号量,这个值不必要；

3 current->state 不是 TASK_RUNNING。除非驱动已自己改变 current 状态,否则状态应该一直是 TASK_RUNNING。

GFP_NOIO

驱动处于块 I/O 处理过程中，它还应当用在所有的存储类型的错误处理过程中。

GFP_KERNEL

所有不属于之前提到的其他情况。

2.4 完成 urb

如果对 usb_submit_urb 的调用成功,驱动将传递对 urb 的控制给 USB 核心,这个函数返回 0;否则,一个负错误值被返回。如果函数调用成功,urb 的完成处理者(就是 complete 指定的的函数)被确切地调用一次,当 urb 被完成。当这个函数被调用,USB 核心完成这个 urb,并且对它的控制现在返回给设备驱动。

有三种情况，一个 urb 可被结束并且使完成函数被调用。

1 urb 被成功发送给设备,并且设备返回正确的确认。对于一个 OUT urb,数据被成功发送,对于一个 IN urb,请求的数据被成功收到。如果发生这个,urb 中的状态变量被设置为 0。

2 一些错误连续发生，当发送或者接受数据从设备中。被 urb 结构中的 status 变量中的错误值所记录。

3 这个 urb 被从 USB 核心去链。这发生在要么当驱动告知 USB 核心取消一个已提交的 urb 通过调用 usb_unlink_urb 或者 usb_kill_urb，要么当设备从系统中去除，以及一个 urb 已经被提交给它。

2.5 urb 的销毁

销毁 urb 的函数是 void usb_free_urb(struct urb *urb)。

参数是一个指向你要释放的 struct urb 的指针，在这个函数被调用之后,urb 结构销毁，驱动不能再使用它。

小结

USB 驱动整个流程简单说来就是：

1 USB 驱动注册（这部分很简单，在介绍具体代码的时候再介绍）；

2 主控器器从 USB 设备中读取 USB 描述符“设备注册”的信息；

3 设备和驱动匹配完成之后进入 probe；

4 申请 USB 请求块 urb，初始化设备，初始化 urb，提交 urb，完成 urb；

5 注销驱动，注销 urb，释放 urb。

联系方式

北京迅为电子有限公司致力于嵌入式软硬件设计，是高端开发平台以及移动设备方案提供商；基于多年的技术积累，在工控、仪表、教育、医疗、车载等领域通过 OEM/ODM 方式为客户创造价值。

iTOP-4412 开发板是迅为电子基于三星最新四核处理器 Exynos4412 研制的一款实验开发平台，可以通过该产品评估 Exynos 4412 处理器相关性能，并以此为基础开发出用户需要的特定产品。

本手册主要介绍 iTOP-4412 开发板的使用方法，旨在帮助用户快速掌握该产品的应用特点，通过对开发板进行后续软硬件开发，衍生出符合特定需求的应用系统。

如需平板电脑案支持，请访问迅为平板方案网“http://www.topeet.com”，我司将有能力为您提供全方位的技术服务，保证您产品设计无忧！

本手册将持续更新，并通过多种方式发布给新老用户，希望迅为电子的努力能给您的学习和开发带来帮助。

迅为电子

2018 年 1 月