CherryUSB 设备协议栈 API 的使用和注意事项

初始化类

```
int usbd_initialize(uint8_t busid, uint32_t reg_base, void (*event_handler)
(uint8_t busid, uint8_t event));

int usbd_deinitialize(uint8_t busid);
```

```
static void usbd_event_handler(uint8_t busid, uint8_t event)
{
    switch (event) {
        case USBD_EVENT_RESET:
            break;
        case USBD_EVENT_CONNECTED: 不支持
            break;
        case USBD_EVENT_DISCONNECTED: 不支持
            break;
        case USBD_EVENT_RESUME: 不支持
            break;
        case USBD_EVENT_SUSPEND: 不支持
            break;
        case USBD_EVENT_CONFIGURED:
            break;
        case USBD_EVENT_SET_REMOTE_WAKEUP:
            break;
        case USBD_EVENT_CLR_REMOTE_WAKEUP:
            break;

        default:
            break;
    }
}
```

需要注意，event handler 函数处于中断上下文

usbd_initialize 如果使用了 os 需要在 task中初始化

注册类

- 描述符注册类

```
void usbd_desc_register(uint8_t busid, const uint8_t *desc);
void usbd_msosv1_desc_register(uint8_t busid, struct usb_msosv1_descriptor
*desc);
void usbd_msosv2_desc_register(uint8_t busid, struct usb_msosv2_descriptor
*desc);
void usbd_bos_desc_register(uint8_t busid, struct usb_bos_descriptor *desc);
```

desc： 设备描述符 + 配置描述符 + class 描述符 + 字符串描述符 + 设备限定描述符，最后需要以 \0 结尾

描述符的格式参考usb官方 usb2.0 文档

- 接口驱动注册类（用于驱动注册）

```
void usbd_add_interface(uint8_t busid, struct usbd_interface *intf);
```

什么是接口?

简单来说，一个接口对应一个 class，对应一个class驱动（class 相关的请求）

有些 class 驱动可能有两个接口：一个命令一个数据

接口描述符：

偏移量	域	大小	值	说明
0	bLength	1	数字	此表的字节数
1	bDescriptorType	1	常量	接口描述表类（此处应为0x04）
2	bInterfaceNumber	1	数字	接口号，当前配置支持的接口数组索引（从零开始）。
3	bAlternateSetting	1	数字	可选设置的索引值。
4	bNumEndpoints	1	数字	此接口用的端点数量，如果是零则说明此接口只用缺省控制管道。
5	bInterfaceClass	1	类	接口所属的类值： 零值为将来的标准保留。 如果此域的值设为FFH，则此接口类由厂商说明。 所有其它的值由USB 说明保留。
6	bInterfaceSubClass	1	子类	子类码 这些值的定义视bInterfaceClass域而定。 如果bInterfaceClass域的值为零则此域的值必须为零。 bInterfaceClass域不为FFH则所有值由USB 所保留。
7	bInterfaceProtocol	1	协议	协议码：bInterfaceClass 和bInterfaceSubClass 域的值而定 如果一个接口支持设备类相关的 请求此域的值指出了设备类说明中所定义的协议.
8	iInterface	1	索引	描述此接口的字串描述表的索引值。

- 端点注册类（用于端点传输完成中断）

```
void usbd_add_endpoint(uint8_t busid, struct usbd_endpoint *ep);
```

什么是端点?

简单来说就是一个传输通道，定义了传输方向，访问地址，每个包的最大包长，传输间隔等等====》 可以简单理解为一个dma通道，实现发送或者接收功能

端点描述符：

Table 9-13. Standard Endpoint Descriptor

Offset	Field	Size	Value	Description
0	*bLength*	1	Number	Size of this descriptor in bytes
1	*bDescriptorType*	1	Constant	ENDPOINT Descriptor Type
2	*bEndpointAddress*	1	Endpoint	The address of the endpoint on the USB device described by this descriptor. The address is encoded as follows: Bit 3...0: The endpoint number Bit 6...4: Reserved, reset to zero Bit 7: Direction, ignored for control endpoints 0 = OUT endpoint 1 = IN endpoint

Offset	Field	Size	Value	Description
3	*bmAttributes*	1	Bitmap	This field describes the endpoint's attributes when it is configured using the *bConfigurationValue*. Bits 1..0: Transfer Type 00 = Control 01 = Isochronous 10 = Bulk 11 = Interrupt If not an isochronous endpoint, bits 5..2 are reserved and must be set to zero. If isochronous, they are defined as follows: Bits 3..2: Synchronization Type 00 = No Synchronization 01 = Asynchronous 10 = Adaptive 11 = Synchronous Bits 5..4: Usage Type 00 = Data endpoint 01 = Feedback endpoint 10 = Implicit feedback Data endpoint 11 = Reserved Refer to Chapter 5 for more information. All other bits are reserved and must be reset to zero. Reserved bits must be ignored by the host.

Offset	Field	Size	Value	Description
4	*wMaxPacketSize*	2	Number	Maximum packet size this endpoint is capable of sending or receiving when this configuration is selected. For isochronous endpoints, this value is used to reserve the bus time in the schedule, required for the per-(micro)frame data payloads. The pipe may, on an ongoing basis, actually use less bandwidth than that reserved. The device reports, if necessary, the actual bandwidth used via its normal, non-USB defined mechanisms. For all endpoints, bits 10..0 specify the maximum packet size (in bytes). For high-speed isochronous and interrupt endpoints: Bits 12..11 specify the number of additional transaction opportunities per microframe: 00 = None (1 transaction per microframe) 01 = 1 additional (2 per microframe) 10 = 2 additional (3 per microframe) 11 = Reserved Bits 15..13 are reserved and must be set to zero. Refer to Chapter 5 for more information.
6	*bInterval*	1	Number	Interval for polling endpoint for data transfers. Expressed in frames or microframes depending on the device operating speed (i.e., either 1 millisecond or 125 µs units). For full-/high-speed isochronous endpoints, this value must be in the range from 1 to 16. The *bInterval* value is used as the exponent for a $2^{\text{bInterval}-1}$ value; e.g., a *bInterval* of 4 means a period of 8 (2^{4-1}). For full-/low-speed interrupt endpoints, the value of this field may be from 1 to 255. For high-speed interrupt endpoints, the *bInterval* value is used as the exponent for a $2^{\text{bInterval}-1}$ value; e.g., a *bInterval* of 4 means a period of 8 (2^{4-1}). This value must be from 1 to 16. For high-speed bulk/control OUT endpoints, the *bInterval* must specify the maximum NAK rate of the endpoint. A value of 0 indicates the endpoint never NAKs. Other values indicate at most 1 NAK each *bInterval* number of microframes. This value must be in the range from 0 to 255. See Chapter 5 description of periods for more detail.

数据传输类

异步 API，尽量 <= 16K

```
int usbd_ep_start_write(uint8_t busid, const uint8_t ep, const uint8_t *data,
uint32_t data_len);

int usbd_ep_start_read(uint8_t busid, const uint8_t ep, uint8_t *data, uint32_t
data_len);
```

什么是分包?

由于 USB 端点包长的限制，当发送数据或者接收的数据大于端点包长时，需要进行分段发送

分包一般分为两种：软件分包(一般硬件设计为FIFO)，硬件自动分包（一般硬件设计为 DMA）

不正确的分包如何影响速度?

- 存在多次copy
- 有dma模式不当dma用（明明可以发很长数据，非要软件再分包），
- 分包之间过慢（不在中断中分包）

端点何时触发完成中断?

分包的最后一个包小于包长，或者长度等于设置的长度

Y = EP_MPS *N + X(X < EP_MPS)

接收举例：端点包长为 512

usbd_ep_start_read(uint8_t busid, const uint8_t ep, uint8_t *data, 2048);

主机发送1字节，接收完成，接收长度1

主机发送 2048字节，接收完成，接收长度2048

主机发送 512*n < 2048，接收不完成

主机发送2048 + 1字节，接收完成，接收长度2048，1字节处于 NAK 状态，下一次启动接收中接收

发送举例：端点包长为 512

- 第一种

usbd_ep_start_write(uint8_t busid, const uint8_t ep, const uint8_t *data, 1025)

512 + 512 + 1，发送完成

- 第二种

usbd_ep_start_write(uint8_t busid, const uint8_t ep, const uint8_t *data, 1024)

512 + 512，发送不一定完成

在 cdc 模式下，这种数据是不能表示完成的，需要再格外发一个0长数据的包，才能表示完成。（这是因为主机接收的原因，我们后面讲解）

usbd_ep_start_write(uint8_t busid, const uint8_t ep, const uint8_t *data, 0)

对于主机设置读取长度的协议中，则不需要发送0数据包，比如MSC,主机会先发一个 cbw（里面包含发送长度）

常规使用流程

USBD_EVENT_RESET 或者 USBD_EVENT_CONFIGURED event 中复位相关标志

等待枚举完成（异步等待）

发送

- 调用 usbd_ep_start_write，并等待完成标志（如果是同步方式，必须设置超时），发送完成中复位标志
- 异步调用 usbd_ep_start_write，不用标志

接收

- USBD_EVENT_CONFIGURED 调用 usbd_ep_start_read，启动第一次接收
- 接收完成中断中如果可以处理数据，设置标志，在其他地方处理（如果是用ringbuffer，可以在当前函数内启动下一次接收）
- 其他地方处理完成后，复位标志，并再次调用usbd_ep_start_read，启动下一次接收