文档编号：AN_2011

上海东软载波微电子有限公司

# 用户手册

## ES32 USB 协议栈用户指南

# 修订历史

| 版本 | 修订日期 | 修改概要 |
|---|---|---|
| V1.0 | 2019-5-21 | 初版 |
| | | |
| | | |
| | | |

地　址：中国上海市龙漕路 299 号天华信息科技园 2A 楼 5 层
邮　编：200235
E-mail：support@essemi.com
电　话：+86-21-60910333
传　真：+86-21-60914991
网　址：http://www.essemi.com/

# 目 录

## 内容目录

# 第1章 简介

## 1.1 概述

ES32 USB 协议栈提供了一套供上层应用使用的接口，包括 USB 主机以及设备功能接口。协议栈涵盖了目前市场上大部分常用的 USB 主机与设备的驱动，并同时提供给用户用于拓展协议栈驱动的方法，以及复合类设备开发的方法。

本套协议栈属于 ES32_SDK 的一部分，作为中间层驱动代码，相关驱动代码可以在 ES32_SDK\firmware\ES32_SDK\Middlewares\EastSoft\usblib 目录下找到，相关应用例程可以在 ES32_SDK\firmware\ES32_SDK\Projects\[相应芯片型号]\Applications\USB 目录下找到。USB 协议栈与 MD 以及 ALD 库对接，在 MD/ALD 的基础上拓展出主机/设备核心驱动层、主机/设备类层驱动、主机/设备层驱动。

USB 协议栈核心层驱动提供了包括枚举、中断管理、上层驱动管理、传输管理、描述符管理、标准请求管理等功能接口，是整个 USB 协议栈工作的核心。各主机/设备类层的驱动提供了 USB 各设备类的描述符解析与创建、设备类请求管理、设备类层数据传输管理、事件管理等功能，可以有效地处理 USB 核心层传来的数据。主机/设备层驱动提供了 USB 具体某个设备的驱动。图 1-1、1-2 展示了 USB 协议栈主机以及从机部分的结构：

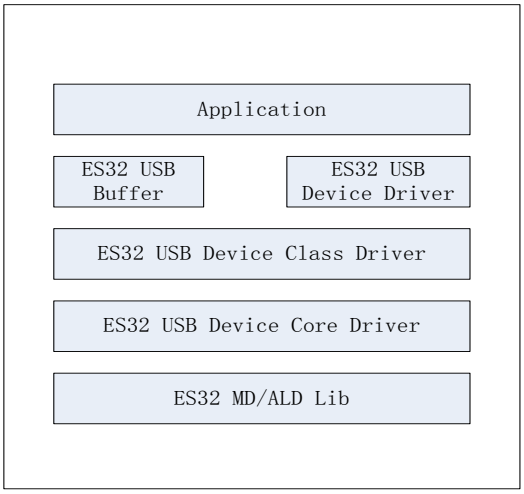


图 1-1 USB 协议栈设备驱动结构

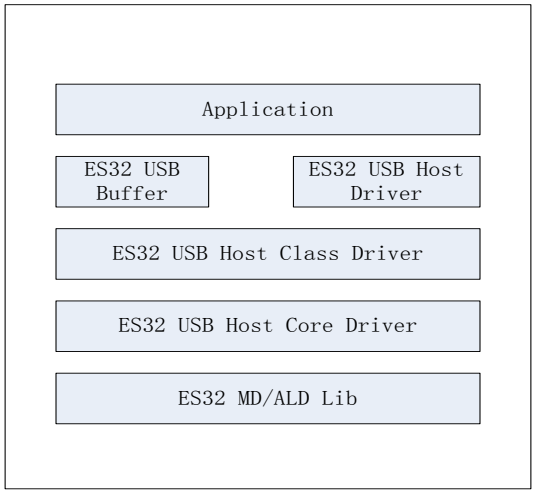


图 1-2 USB 协议栈主机驱动结构

## 1.2 操作模式

ES32 USB 协议栈中定义了以下用户操作模式，分别为：

- **eUSBModeDevice**
- **eUSBModeHost**
- **eUSBModeForceHost**
- **eUSBModeForceDevice**

用户可以使用 usb_stack_mode_set()函数配置。用户配置为 **eUSBModeDevice** 模式时，USB 协议栈工作在设备模式，但是协议栈同时会检测 ID 以及 VBUS 引脚；配置为 **eUSBModeHost** 模式时，USB 协议栈工作在主机模式，但是协议栈同时会检测 ID 以及 VBUS 引脚；配置为 **eUSBModeForceHost** 模式时，USB 协议栈工作在主机模式，协议栈不会检测 ID 以及 VBUS 引脚；配置为 **eUSBModeForceDevice** 模式时，USB 协议栈工作在设备模式，协议栈不会检测 ID 以及 VBUS 引脚。如果应用明确在设备或者主机模式时，建议将协议栈工作模式设为强制主机/设备模式。

## 1.3 文档结构

协议栈文件夹结构如下：

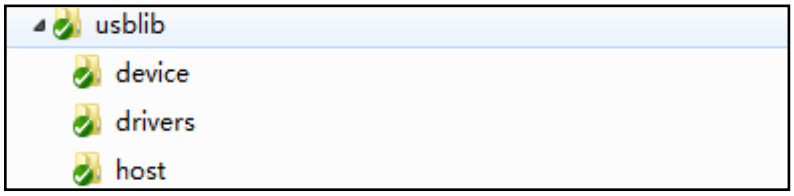


图 1-3 USB 协议栈文件夹结构

usblib/device USB 协议栈设备相关驱动文件夹，放置 USB 设备类、USB 设备的驱动源代码以及头文件，例如" usbdhidmouse.c"文件即为鼠标设备的驱动源代码；

usblib/host USB 协议栈主机相关驱动文件夹，放置 USB 主机驱动的源代码以及头文件；

usblib/drivers 本文件夹放置用来连接 ES32 系列单片机 MD/ALD 库的头文件以及 USB 协议栈用到的宏定义。

# 第2章 设备部分

## 2.1 简介

本章节将介绍 ES32 USB 协议栈设备部分相关驱动，设备部分的驱动涉及从底层 MD/ALD 库到高层驱动部分包括设备类驱动、设备驱动，用户可以从适当的层次去开发。

### 2.1.1 源码文件介绍

USB 协议栈的设备部分相关驱动的源码以及头文件放在 usblib/device 文件夹下。

| | |
|---|---|
| usbdaudio.c | AUDIO 类驱动的源文件，本文件提供 USB AUDIO 设备类相关的 API。 |
| usbdbulk.c | BULK 类驱动的源文件，本文件提供 USB BULK 设备类相关的 API。 |
| usbdcdc.c | CDC 类驱动源文件，本文件提供 USB CDC 设备类相关的 API。 |
| usbdcdesc.c | USB 描述符解析驱动文件，本文件提供 USB 描述符解析相关的 API。 |
| usbdcdesc.c | USB 描述符解析驱动文件，本文件提供 USB 描述符解析相关的 API。 |
| usbdcomp.c | USB 复合设备驱动文件，本文件提供复合设备建立的相关 API 以及相应的描述符。 |
| usbdconfig.c | USB 协议栈设备部分配置文件，本文件包含上层应用所需的 USB 协议栈的相关配置 API。 |
| usbdcore.c | USB 协议栈设备部分的核心层驱动源码。 |
| usbddfu-rt.c | DFU 设备“run time”阶段驱动源文件，本文件提供 DFU “run time”阶段相关的驱动 API。 |
| usbdhandler.c | USB 协议栈设备中断处理文件，本文件包含设备中断的处理函数。 |
| usbdhid.c | HID 类驱动源文件，本文件提供 HID 类驱动的相关 API。 |
| usbdhidgamepad.c | HID 游戏手柄驱动源文件，本文件提供 HID 游戏手柄驱动的相关 API。 |
| usbdhidkeyb.c | HID 键盘驱动源文件，本文件提供 HID 键盘驱动的相关 API。 |
| usbdhidmouse.c | HID 鼠标驱动源文件，本文件提供 HID 鼠标驱动的相关 API。 |
| usbdmsc.c | MSC 类驱动源文件，本文件提供 MSC 类驱动的相关 API。 |

### 2.1.2 设备部分驱动结构

USB 协议栈设备部分的驱动是建立在 MD/ALD 库基础上的一套三层结构的函数库，用户可以在此三层结构上开发设备。协议栈分为核心层、设备类层、设备层的驱动。其中

核心层负责处理 USB 的中断以及将相应的事件传输给设备类层驱动、USB 数据的初步处理，当有需要传输到设备类层驱动的数据，核心层会以事件回调的方式将数据传达设备类层驱动；设备类层驱动负责 USB 各设备类协议的处理，核心层将相关的事件数据传到设备类层时，设备类会将数据进一步处理，如果设备类下有多个子类设备的话，需要子类用到的数据也会以事件回调的方式传输到设备层；USB 某些设备类下有多个子类，例如 HID 设备类下有 HID 键盘、HID 鼠标等设备，那么这些子类设备的驱动放在设备层中，设备层的功能和相关的子设备相关，如 HID 鼠标设备驱动中就提供了 HID 鼠标的描述符以及数据发送的 API。

此外，USB 某些设备类的驱动在使用时需要传输较大的数据量，为了能够更好地管理这些数据的传输，USB 协议栈提供了一个 Buffer 模块用来管理大的数据的传输，例如 CDC 类的驱动就可以将相关的数据传输 API 对接上 Buffer 模块，从而实现仅对 Buffer 的操作就能够实现数据的传输。

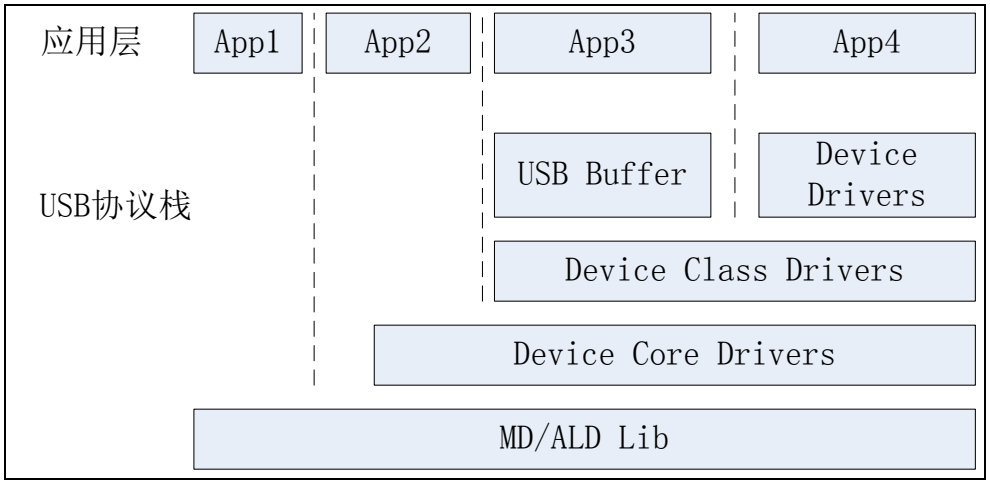


图 2-1 USB 协议栈设备驱动结构

### 2.1.3 底层驱动

USB 协议栈底层的驱动为 MD 库驱动，文件为 Drivers\MD\[device]\Source\md_usb.c。图 2-1 中的 App1 的应用场景便是基于 MD/ALD 库开发的。由于 USB 底层的驱动只是对 USB 外设的寄存器做直接的操作，没有上层协议的数据传输的支持，所以本层不适合应用程序的开发，但是本层支持第三方 USB 协议栈的开发。

### 2.1.4 核心层驱动

USB 协议栈核心层驱动提供了一套独立于 USB 设备类驱动的 API，支持设备枚举，支持 USB 标准请求的处理以及支持其他请求例如类请求的信息传递到设备类层。使用本层接口的应用程序需要提供给本层在初始化期间发布给主机的描述符，这些描述符提供了 USB 核心层驱动 API 配置硬件所需要的信息。与 USB 设备相关的异步事件通过在初始化时提供给 USB 核心层驱动 API 的一组回调函数通知到应用程序。

本层驱动程序用于 USB 设备类层驱动程序的开发，也可以直接用于希望提供现有类驱动不支持的 USB 功能的应用程序。USB 核心层驱动可以看作是 USB 底层驱动的一组高级设备扩展，在开发 USB 核心层驱动的时候也需要调用 USB 底层驱动的接口。

### 2.1.5 设备类层驱动

设备类驱动程序为希望提供特定 USB 功能的应用程序提供高级 USB 功能，而不需要处理大多数 USB 事务处理和连接管理。这些驱动程序为几个常用的 USB 设备类提供了高级 API，具有以下特性：

- 便于使用，设备设置包括一组静态数据结构和调用一个初始化 API。
- 可配置的 VID/PID，电源参数和字符串描述符表，一些应用不需要修改库代码即可轻松使用。
- 一致的接口，所有设备类层驱动都使用类似的 API，因此掌握一个设备类 API 之后再掌握其它设备类驱动 API 是很轻松的事。
- 应用程序开销小，绝大多数 USB 部分的处理都是在类驱动程序以及较低层的驱动中执行，因此应用程序只需要能够处理读和写数据即可。
- 提供 buffer 对象，设备类层驱动可与可选的 USB buffer 对象一起使用，进一步简化数据的传输与接收。USB buffer 与设备类驱动的交互可以使数据在传输时仅仅使用 buffer 对象的读写 API，不需要状态机来确保数据在正确的事件传输或接收。
- 设备类驱动程序完全封装了 USB 底层驱动以及 USB 核心层驱动，因此应用程序可以仅仅使用本层驱动来开发应用。

为了避免多层次之间开发造成的干扰，开发人员在使用设备类层驱动开发时需要注意以下几点：

- 在使用设备类层驱动 API 时，谨慎对待对其它 USB 层 API 的调用。
- 所提供的设备类驱动不支持备用配置。

### 2.1.6 设备层驱动

某些情况下，一个标准的设备类可能提供使用同一个类创建大量不同设备的可能性，在这些情况下，还可以提供一个额外的 API 层来进一步处理特定的设备并简化应用程序的接口。HID 类就是这样一个类，HID 类可以扩展为各种人机接口设备，如键盘、鼠标、手柄。为了简化接口的使用，USB 协议栈提供了特定的 API 来支持键盘、鼠标等操作。例如使用鼠标设备驱动而不是使用 HID 类驱动的 API 就可以让鼠标的应用很便捷，类似的键盘设备包含了键盘设备的初始化以及键盘设备专有的数据传输接口。

## 2.2 CDC设备类驱动

USB CDC 类驱动程序支持 CDC 抽象模型，CDC 类驱动将设备配置为可以在主机显示为虚拟串口的设备。驱动程序提供两个通道，一个传输通道和一个接收通道。这些通道可以和 USB buffer 模块一起使用，从而降低数据读写的复杂度。其它 API 和事件也支持串行链路操作，例如通知 UART 错误、发送 break 信号以及设置通信线路参数。

CDC 类设备的数据传输和 BULK 类相类似，但是由于 CDC 类是一个标准的设备类，主机操作系统有可能不需要像 BULK 设备那样需要特定的驱动程序就可以访问该设备。例如在 windows 上，只需要一个简单的 INF 文件就可以使 USB 设备显示为 COM 端口，任何串行终端应用程序都可以访问该端口。

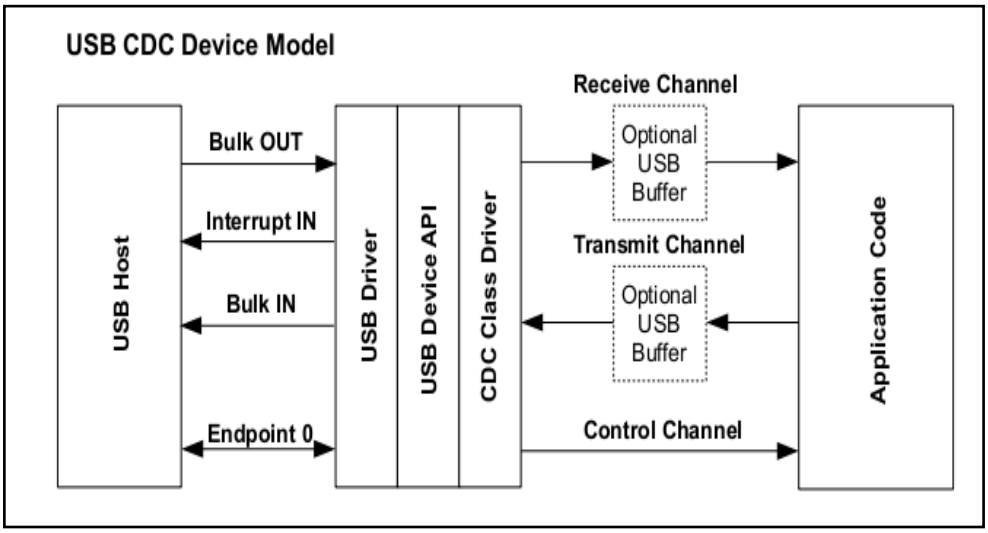


图 2-2 USB CDC 类设备驱动模型

除了端点 0 之外，CDC 设备类还使用了三个端点，两个块传输端点负责在主从机传输数据，中断端点用于控制 CDC 数据传输过程中的一些错误信息以及控制信息（如 break、帧错误、奇偶检验）。端点 0 携带了 USB 请求和 CDC 特定请求，通过回调传递到上层应用。

### 2.2.1 CDC 设备类事件

CDC 设备类驱动提供了以下回调事件：

#### 2.2.1.1 接收通道相关事件

- **USB_EVENT_RX_AVAILABLE**
- **USB_EVENT_DATA_REMAINING**
- **USB_EVENT_ERROR**

#### 2.2.1.2 发送通道相关事件

- **USB_EVENT_TX_COMPLETE**

#### 2.2.1.3 控制通道相关事件

- **USB_EVENT_CONNECTED**
- **USB_EVENT_DISCONNECTED**
- **USB_EVENT_SUSPEND**
- **USB_EVENT_RESUME**
- **USBD_CDC_EVENT_SEND_BREAK**
- **USBD_CDC_EVENT_CLEAR_BREAK**
- **USBD_CDC_EVENT_SET_LINE_CODING**
- **USBD_CDC_EVENT_GET_LINE_CODING**
- **USBD_CDC_EVENT_SET_CONTROL_LINE_STATE**

注意，**USB_EVENT_DISCONNECTED** 事件仅在某些器件内支持，具体请查阅相关器件手册，确定设备模式下是否有断开连接检测功能。。

### 2. 2. 2 CDC设备类驱动使用教程

CDC 设备类驱动使用可以遵循以下几步：

- 将以下头文件加入到需要用到此驱动的源文件中：

```
#include "usblib/usblib.h"
#include "usblib/device/usbdevice.h"
#include "usblib/device/usbdcdc.h"
#include "usblib/drivers/usb_lowlayer_api.h"
```

- 定义 6 个用来描述设备特征的 USB 字符串描述符，修改字符串内容来描述用户自己的设备，注意当字符串改变后，每个描述符数组的长度（每个数组的第一个元素即为数组长度）也需要相应地进行修改。

```
/**
  * The languages supported by this device.
  */
const uint8_t g_pui8LangDescriptor[] =
{
  4,
  USB_DTYPE_STRING,
  USBShort(USB_LANG_EN_US)
};

/**
  * The manufacturer string.
  */
const uint8_t g_pui8ManufacturerString[] =
{
  (8 + 1) * 2,
  USB_DTYPE_STRING,
  'E', 0, 'a', 0, 's', 0, 't', 0, 'S', 0, 'o', 0, 'f', 0, 't', 0,
};

/**
  *  The product string.
  */
const uint8_t g_pui8ProductString[] =
{
  (12 + 1) * 2,
  USB_DTYPE_STRING,
  'V', 0, 'i', 0, 'r', 0, 't', 0, 'u', 0, 'a', 0, 'l', 0, ' ', 0,
  'P', 0, 'o', 0, 'r', 0, 't', 0
};
```

```
/**
  *  The serial number string.
  */
const uint8_t g_pui8SeriailNumberString[] =
{
  (8 + 1) * 2,
  USB_DTYPE_STRING,
  '1', 0, '2', 0, '3', 0, '4', 0, '5', 0, '6', 0, '7', 0, '8', 0
};

/**
  *  The interface description string.
  */
const uint8_t g_pui8InterfaceString[] =
{
  (18 + 1) * 2,
  USB_DTYPE_STRING,
  'C', 0, 'D', 0, 'C', 0, ' ', 0, 'D', 0, 'a', 0, 't', 0, 'a', 0,
  ' ', 0, 'I', 0, 'n', 0, 't', 0, 'e', 0, 'r', 0, 'f', 0, 'a', 0,
  'c', 0, 'e', 0
};

/**
  *  The configuration description string.
  */
const uint8_t g_pui8ConfigString[] =
{
  (23 + 1) * 2,
  USB_DTYPE_STRING,
  'C', 0, 'D', 0, 'C', 0, ' ', 0, 'D', 0, 'a', 0, 't', 0, 'a', 0,
  ' ', 0, ' ', 0, 'C', 0, 'o', 0, 'n', 0, 'f', 0, 'i', 0, 'g', 0,
  'u', 0, 'r', 0, 'a', 0, 't', 0, 'i', 0, 'o', 0, 'n', 0
};

/**
  *  The descriptor string table.
  */
const uint8_t * const g_ppui8StringDescriptors[] =
{
  g_pui8LangDescriptor,
  g_pui8ManufacturerString,
  g_pui8ProductString,
  g_pui8SeriailNumberString,
```

```
    g_pui8InterfaceString,
    g_pui8ConfigString
};
/**
 *  The number of string descriptors.
 */
#define NUM_STRING_DESCRIPTORS (sizeof(g_ppui8StringDescriptors) / sizeof(uint8_t *))
```

- 定义一个 **tUSBDCDCDevice** 类型的结构体用来作为 CDC 类初始化的参数。该结构体包括设备的 PID、VID、供电参数以及 CDC 设备类驱动回调函数入口、回调函数参数等信息。以下是一个简单的 CDC 类初始化结构体例子：

```
// Define CDC device.
const tUSBDCDCDevice g_sCDCDevice =
{
    // The Vendor ID you have been assigned by USB-IF.
    USB_VID_YOUR_VENDOR_ID,
    // The product ID you have assigned for this device.
    USB_PID_YOUR_PRODUCT_ID,
    // The power consumption of your device in milliamps.
    POWER_CONSUMPTION_MA,
    // The value to be passed to the host in the USB configuration descriptor's
    // bmAttributes field.
    USB_CONF_ATTR_SELF_PWR,
    // A pointer to your control callback event handler.
    YourUSBControlEventCallback,
    // A value that you want passed to the control callback alongside every
    // event.
    (void *)&g_sYourInstanceData,
    // A pointer to your receive callback event handler.
    YourUSBReceiveEventCallback,
    // A value that you want passed to the receive callback alongside every
    // event.
    (void * )&g_sYourInstanceData,
    // A pointer to your transmit callback event handler.
    YourUSBTransmitEventCallback,
    // A value that you want passed to the transmit callback alongside every
    // event.
    (void*)&g_sYourInstanceData,
    // A pointer to your string table.
    g_ppui8StringDescriptors,
    // The number of entries in your string table.
    NUM_STRING_DESCRIPTORS
};
```

- 添加接收事件处理函数，如例子中 YourUSBReceiveEventCallback()函数。就

CDC 设备而言，**USB_EVENT_RX_AVAILABLE** 和 **USB_EVENT_DATA_REMAINING** 事件必须在接收事件处理函数中处理。在 **USB_EVENT_RX_AVAILABLE** 事件中，应用程序可以调用 usbdcdc_rx_packet_available()函数来读取目前接收到的有效数据数量，调用 usbdcdc_packet_read()函数来读取数据。等到设备将接收到的数据全部读取完毕，主机便可以发送下一包数据到设备,。如果当前数据设备无法立即读取，设备需要在回调函数中返回 0，这样回调函数就会在若干毫秒之后再次被调用。在 **USB_EVENT_DATA_REMAINING** 事件中，应用程序可以返回当前缓存的数据量。这个事件可以用来控制某些传入请求的计时，例如 break 信号的计时。尽管没有其它事件被强制要求用于 CDC 设备类驱动，**USB_EVENT_CONNECTED** 和 **USB_EVENT_DISCONNECTED** 事件通常都会被用来指示设备的连接与断开。

- 添加发送事件处理函数，如例子中 YourUSBTransmitEventCallback()函数.就 CDC 设备而言，发送事件处理函数中并没有强制要求处理的事件，但是通常发送完成事件 **USB_EVENT_TX_COMPLETE** 会被用来指示数据发送完毕。
- 添加控制事件处理函数，如例子中的 YourUSBControlEventCallback()函数。用户可以使用相关控制事件控制 CDC 数据传输，例如使用 **USBD_CDC_EVENT_GET_LINE_CODING** 事件来向主机返回串口传输的数据特征（波特率、数据位数等信息）。
- 在主函数中调用 CDC 类初始化函数 usbdcdc_init()来配置 USB 控制器。usbdcdc_init(0, &g_psCDCDevice);
- 在 USB 中断函数中调用 USB 协议栈设备中断处理函数 usb_device_int_handler();

### 2. 2. 3 CDC复合设备类驱动使用教程

当 CDC 设备驱动用于创建一个复合设备时，配置过程大致和单个 CDC 设备创建过程相似。需要注意的是，此时调用的 CDC 初始化函数已经不是 usbdcdc_init()函数，而是使用 usbdcdc_composite_init()函数，这个函数为复合设备创建了一个用于枚举的 CDC 串口实体，此函数将初始化 **tCompositeEntry** 类型的指针来作为复合设备参数表，具体的使用方法见下面的例子：

```
tCompositeEntry g_psCompEntries[NUM_COMP_DEVICES];

//*****************************************************************************
// Allocate the Device Data for the top level composite device class.
//*****************************************************************************
tUSBDCompositeDevice g_sCompDevice =
{
    .ui16VID                = USB_VID_EASTSOFT_30CC,
    .ui16PID                = USB_PID_COMP_HID_SER,
    .ui16MaxPowermA         = 250,
    .ui8PwrAttributes       = USB_CONF_ATTR_BUS_PWR,
    .pfnCallback            = 0,
```

```
    .ppui8StringDescriptors        = g_ppui8StringDescriptors,
    .ui32NumStringDescriptors    = NUM_STRING_DESCRIPTORS,
    .ui32NumDevices                 = NUM_COMP_DEVICES,
    .psDevices                        = g_psCompEntries
};

    //
    // Initialize the serial port instances that is part of this
    // composite device.
    //
    g_sCompDevice.psDevices[0].pvInstance =
        usbdcdc_composite_init(0, &g_psCDCDevice, &g_psCompEntries[0]);

    //
    // Pass the device information to the USB library and place the device
    // on the bus.
    //
    usbd_composite_init(0, &g_sCompDevice, DESCRIPTOR_DATA_SIZE,
                                g_pui8DescriptorData);
```

### 2. 2. 4 CDC类驱动接口定义

**数据结构：**

- tLineCoding
- tUSBDCDCDevice

宏：

- COMPOSITE_DCDC_SIZE
- USBD_CDC_EVENT_CLEAR_BREAK
- USBD_CDC_EVENT_SEND_BREAK
- USBD_CDC_EVENT_GET_LINE_CODING
- USBD_CDC_EVENT_SET_LINE_CODING
- USBD_CDC_EVENT_SET_CONTROL_LINE_STATE

**函数接口：**

- void *usbdcdc_composite_init(uint32_t ui32Index,
  tUSBDCDCDevice* psCDCDevice, tCompositeEntry * psCompEntry);
- void *usbdcdc_init(uint32_t ui32Index, tUSBDCDCDevice *psCDCDevice);
- void usbdcdc_term(void *pvCDCDevice);
- void *usbdcdc_set_control_cb_data(void *pvCDCDevice, void *pvCBData);

- void *usbdcdc_set_rx_cb_data(void *pvCDCDevice, void *pvCBData);
- void *usbdcdc_set_tx_cb_data(void *pvCDCDevice, void *pvCBData);
- uint32_t usbdcdc_packet_write(void *pvCDCDevice, uint8_t *pi8Data, uint32_t ui32Length, bool bLast);
- extern uint32_t usbdcdc_packet_read(void *pvCDCDevice, uint8_t *pi8Data, uint32_t ui32Length, bool bLast);
- uint32_t usbdcdc_tx_packet_available(void *pvCDCDevice);
- uint32_t usbdcdc_rx_packet_available(void *pvCDCDevice);
- void usbdcdc_serial_state_change(void *pvCDCDevice, uint16_t ui16State);
- bool usbdcdc_remote_wakeup_request(void *pvCDCDevice);

#### 2. 2. 4. 1 tLineCoding

定义：

```
typedef struct
{
    uint32_t   ui32Rate;
    uint8_t    ui8Stop;
    uint8_t    ui8Parity;
    uint8_t    ui8Databits;
} tLineCoding;
```

参数定义：

**ui32Rate**：数据终端速率，比如控制的设备为 UART 时即为波特率。

**ui8Stop**：停止位位数，值可以是 **USB_CDC_STOP_BITS_1**、**USB_CDC_STOP_BITS_1_5**、**USB_CDC_STOP_BITS_2**。

**ui8Parity**：奇偶校验设置，值可以是 **USB_CDC_PARITY_NONE**、**USB_CDC_PARITY_ODD** 、**USB_CDC_PARITY_EVEN**、**USB_CDC_PARITY_MARK**、**USB_CDC_PARITY_SPACE**。

**ui8Databits**：数据位数，值可以是 5、6、7、8。

#### 2. 2. 4. 2 tUSBDCDCDevice

定义：

```
typedef struct
{
    const uint16_t ui16VID;
    const uint16_t ui16PID;
    const uint16_t ui16MaxPowermA;
    const uint8_t ui8PwrAttributes;
    const tUSBCallback pfnControlCallback;
```

```
    void *pvControlCBData;
    const tUSBCallback pfnRxCallback;
    void *pvRxCBData;
    const tUSBCallback pfnTxCallback;
    void *pvTxCBData;
    const uint8_t * const *ppui8StringDescriptors;
    const uint32_t ui32NumStringDescriptors;
    tCDCSerInstance sPrivateData;
} tUSBDCDCDevice;
```

**参数定义：**

**ui16VID**：厂商 ID，由 USB-IF 统一分配。

**ui16PID**：产品 ID，由制造商分配。

**ui16MaxPowermA**：最大供电电流（mA）。

**ui8PwrAttributes**：设备供电特性，值可以是 **USB_CONF_ATTR_SELF_PWR**、**USB_CONF_ATTR_BUS_PWR**，另外可选的配置为 **USB_CONF_ATTR_RWAKE**（如果设备支持此功能）。

**pfnControlCallback**：指向控制回调函数的指针，该函数将被 USB 协议栈调用来通知上层应用所有与该设备操作相关的异步控制事件。

**pvControlCBData**：应用程序分配的指针，它将作为控制回调函数的第一个参数。

**pfnRxCallback**：指向接收回调的指针，该函数将被 USB 协议栈调用来通知上层应用程序与该设备接收通道相关的事件。

**pvRxCBData**：应用程序分配的指针，它将作为接收回调函数的第一个参数。

**pfnTxCallback**：指向发送回调的指针，该函数将被 USB 协议栈调用来通知上层应用程序与该设备发送通道相关的事件。

**pvTxCBData**：应用程序分配的指针，它将作为发送回调函数的第一个参数。

**ppui8StringDescriptors**：指向字符串描述符表的指针。

**ui32NumStringDescriptors**：设备字符串描述符表的大小。

**sPrivateData**：CDC 驱动私有参数，应用程序不能对其进行操作，在应用程序中需要为此参数分配一定的存储空间。

#### 2. 2. 4. 3 COMPOSITE_DCDC_SIZE

**描述**：为单个 CDC 设备创建一个配置描述符应该分配的内存大小。

#### 2. 2. 4. 4 USBD_CDC_EVENT_CLEAR_BREAK

**描述**：清除“break”信号事件。

#### 2. 2. 4. 5 USBD_CDC_EVENT_SEND_BREAK

**描述**：发送“break”信号事件。

#### 2. 2. 4. 6 USBD_CDC_EVENT_GET_LINE_CODING

**描述**：主机正在查询当前 RS232 通信参数时，回调函数 pvMsgData 参数指向一个 **tLineCoding** 类型结构体，应用程序必须在返回回调之前填写当前设置。

#### 2. 2. 4. 7 USBD_CDC_EVENT_SET_LINE_CODING

**描述**：主机请求配置当前 RS232 通信参数时，回调函数 pvMsgData 参数指向一个 **tLineCoding** 类型结构体，定义所需的参数。

#### 2. 2. 4. 8 USBD_CDC_EVENT_SET_CONTROL_LINE_STATE

**描述**：主机请求将当前 RS232 信号线设置为特定状态，回调函数 ui32MsgValue 参数包含了 RTS 和 DTR 控制状态，根据 USB 协议，相关配置可以为：

(RTS) U**SB_CDC_DEACTIVATE_CARRIER** or **USB_CDC_ACTIVATE_CARRIER**

(DTR) **USB_CDC_DTE_NOT_PRESENT** or **USB_CDC_DTE_PRESENT**。

#### 2. 2. 4. 9 usbdcdc_composite_init()

当需要创建复合设备时用于 CDC 设备的初始化。

**参数定义：**

**ui32Index**：USB 控制器索引。

**psCDCDevice**：指向 CDC 初始化结构体 **tUSBDCDCDevice** 的指针。

**psCompEntry**：指向复合设备初始化入口结构体 **tCompositeEntry** 的指针，当需要创建复合设备时作为复合设备的初始化入口参数。

**简介：**

本函数和 CDC 初始化函数类似，当需要创建复合函数时，这个函数会初始化指向 **tCompositeEntry** 结构体的入口参数。

**返回值：**

失败返回 0，成功则返回指向 **tUSBDCDCDevice** 的指针，用于接下来 CDC 类驱动 API 的参数。

#### 2. 2. 4. 10 usbdcdc_init()

初始化 CDC 设备。

**参数定义：**

**ui32Index**：USB 控制器索引。

**psCDCDevice**：指向 CDC 初始化结构体 **tUSBDCDCDevice** 的指针。

**简介：**

应用程序希望创建一个 CDC 设备并在主机端显示为虚拟串口时，可以调用这个函数来初始化 USB 控制器，此函数执行所有的 CDC 设备的初始化工作。

此函数调用成功后会返回传递给它的 **tUSBDCDCDevice** 指针，这个指针必须作为以后所有的 CDC 类驱动 API 的参数。

USB CDC 类驱动的数据收发是基于 USB 数据包的（每包的传输大小和具体的传输方式有关，例如全速设备的批量传输，一包数据最大为 64 bytes），如果应用程序希望基于块的收发，可以使用 USB 协议栈提供的 buffer 模块。

当需要发送操作时，调用 usbdcdc_packet_write()函数可以一次性发送不超过 64 bytes 的数据。当一包数据传输完成后，USB 协议栈会发送 **USB_EVENT_TX_COMPLETE** 事件到应用程序的回调函数中去，以通知应用程序可以进行接下来的操作。当需要接收操作时，USB 协议栈会向应用程序回调函数发送 **USB_EVENT_RX_AVAILABLE** 事件，应用程序可以调用 usbdcdc_packet_read()函数来读取接收到的数据，接收到的数据量大小可以使用 usbdcdc_rx_packet_available()函数来读取。

**返回值：**

失败返回 0，成功则返回指向 **tUSBDCDCDevice** 的指针，用于接下来 CDC 类驱动 API 的参数。

#### 2. 2. 4. 11 usbdcdc_packet_read()

从 CDC 数据接口读取一包数据。

**参数定义：**

**pvCDCDevice**：指向由 usbdcdc_init()函数返回的 CDC 初始化结构体 **tUSBDCDCDevice** 的指针。

**pi8Data**：指向数据接收的存储空间指针。

**ui32Length**：pi8Data 所指向的存储空间大小。

**bLast**：指示从机是否需要进一步调用从包中读取数据，本位供 USB 协议栈 buffer 模块调度使用，当应用程序直接使用此函数读取数据时，将本位置 1 即可。

**简介：**

此函数将读取最多 ui32Length 指示的数据量的数据。当从机可以确定读取数据量的时候，bLast 参数可以忽略。

**返回值：**

返回读取到的数据数量。

#### 2. 2. 4. 12 usbdcdc_packet_write()

向主机发送一包数据。

**参数定义：**

**pvCDCDevice**：指向由 usbdcdc_init()函数初始化返回的 CDC 初始化结构体 **tUSBDCDCDevice** 的指针。

**pi8Data**：指向数据发送的存储空间指针。

**ui32Length**：pi8Data 所指向的存储空间大小

**bLast**：指示从机是否需要进一步调用以发送数据，如果为 true，USB 协议栈将会进一步调用此函数，如果为 false，在调度完一个短包传输后则不会再进行任何调用。本位供 USB 协议栈 buffer 模块调度使用，当应用程序直接使用此函数发送数据时，将本位置 1 即可。

**简介：**

此函数将提供数据调度功能，将一个 USB 数据包传输到 USB 主机端。如果当前没有传输正在进行，数据将立即复制到相关的 USB 端点 FIFO。如果 bLast 参数为 true，那么新写的数据包就会被安排传输。当 USB 主机确认接收到数据包后，协议栈会向应用程序发送 **USB_EVENT_TX_COMPLETE** 事件，指示可以发送接下来的数据包。

调用此函数一次最大发送的数据长度为 64 bytes。

**返回值：**

返回成功发送的数据数量。

#### 2. 2. 4. 13 usbdcdc_term()

停用 CDC 驱动。

**参数定义：**

**pvCDCDevice**：指向由 usbdcdc_init()函数初始化返回的 CDC 初始化结构体 **tUSBDCDCDevice** 的指针。

**简介：**

此函数将终止所有 CDC 类驱动操作，并从总线删除设备，如果 CDC 设备是复合设备的一部分，那么不应该调用这个函数，而应该为整个复合设备调用 usbd_composite_term()函数。在调用此函数之后，**tUSBDCDCDevice** 指针实例不应该在任何其它 API 中使用。

**返回值：**

无。

#### 2. 2. 4. 14 usbdcdc_tx_packet_available()

返回可以发送的数据数量。

**参数定义：**

**pvCDCDevice**：指向由 usbdcdc_init()函数初始化返回的 CDC 初始化结构体 **tUSBDCDCDevice** 的指针。

**简介：**

此函数返回 usbdcdc_packet_write()函数可以发送的最大数据量。

**返回值：**

返回可以传输的数据数量。

#### 2. 2. 4. 15 usbdcdc_rx_packet_available()

返回可以读取的数据数量。

**参数定义：**

**pvCDCDevice**：指向由 usbdcdc_init()函数初始化返回的 CDC 初始化结构体 **tUSBDCDCDevice** 的指针。

**简介：**

此函数返回 usbdcdc_packet_read()函数可以读取的最大数据量。

**返回值：**

返回可以读取的数据数量。

#### 2. 2. 4. 16 usbdcdc_serial_state_change()

通知 CDC 模块串口控制参数有改变或者有错误发生。

**参数定义：**

**pvCDCDevice**：指向由 usbdcdc_init()函数初始化返回的 CDC 初始化结构体 **tUSBDCDCDevice** 的指针。

**ui16State**：指示检测到的控制线的状态和任何接收错误。位定义和 USB CDC 串口状态异步通知一样，在头文件“usbcdc.h”中定义。

**简介：**

应用程序可以调用此函数来通知主机串口总线状态的变化，ui16State 参数可以试一下的值：

- **USB_CDC_SERIAL_STATE_OVERRUN**
- **USB_CDC_SERIAL_STATE_PARITY**
- **USB_CDC_SERIAL_STATE_FRAMING**
- **USB_CDC_SERIAL_STATE_RING_SIGNAL**
- **USB_CDC_SERIAL_STATE_BREAK**
- **USB_CDC_SERIAL_STATE_TXCARRIER**
- **USB_CDC_SERIAL_STATE_RXCARRIER**

**返回值：**

无。

#### 2. 2. 4. 17 usbdcdc_set_control_cb_data()

修改 CDC 驱动的控制回调函数。

**参数定义：**

**pvCDCDevice**：指向由 usbdcdc_init()函数初始化返回的 CDC 初始化结构体 **tUSBDCDCDevice** 的指针。

**pvCBData**：指向控制回调函数的指针。

**简介：**

应用程序使用此函数来修改由 usbdcdc_init()函数配置好的回调函数指针。如果应用程序需要在运行时修改控制回调函数的指针值，需要保证 pvCDCDevice 存储在 SRAM 中。

**返回值：**

返回先前使用的回调函数指针值。

#### 2. 2. 4. 18 usbdcdc_set_rx_cb_data()

修改 CDC 驱动的接收回调函数。

**参数定义：**

**pvCDCDevice**：指向由 usbdcdc_init()函数初始化返回的 CDC 初始化结构体 **tUSBDCDCDevice** 的指针。

**pvCBData**：指向接收回调函数的指针。

**简介：**

应用程序使用此函数来修改由 usbdcdc_init()函数配置好的回调函数指针。如果应用程序需要在运行时修改接收回调函数的指针值，需要保证 pvCDCDevice 存储在 SRAM 中。

**返回值：**

返回先前使用的回调函数指针值。

#### 2. 2. 4. 19 usbdcdc_set_tx_cb_data()

修改 CDC 驱动的发送回调函数。

**参数定义：**

**pvCDCDevice**：指向由 usbdcdc_init()函数初始化返回的 CDC 初始化结构体 **tUSBDCDCDevice** 的指针。

**pvCBData**：指向发送回调函数的指针。

**简介：**

应用程序使用此函数来修改由 usbdcdc_init()函数配置好的回调函数指针。如果应用

程序需要在运行时修改发送回调函数的指针值，需要保证 pvCDCDevice 存储在 SRAM 中。

**返回值：**

返回先前使用的回调函数指针值。

#### 2. 2. 4. 20 usbdcdc_power_status_set()

修改设备的供电特性。

**参数定义：**

**pvCDCDevice**：指向由 usbdcdc_init()函数初始化返回的 CDC 初始化结构体 **tUSBDCDCDevice** 的指针。

**ui8Power**：供电特性。

**简介：**

USB 协议栈支持设备供电特性的修改，修改的值可以是 **USB_STATUS_SELF_PWR** 或者 **USB_STATUS_BUS_PWR**。在 USB 主机设备再次请求后，USB 协议栈会将修改后的供电特性报告给主机。

**返回值：**

无。

#### 2. 2. 4. 21 usbdcdc_remote_wakeup_request()

当 USB 控制器处于挂起状态时，发送“wakeup”信号到主机。

**参数定义：**

**pvCDCDevice**：指向由 usbdcdc_init()函数初始化返回的 CDC 初始化结构体 **tUSBDCDCDevice** 的指针。

**简介：**

当处于挂起状态时，如果从机支持远程唤醒（通过初始化结构体设置）功能的话，可以调用此函数来唤醒主机。

**返回值：**

返回 true 表示远程唤醒功能没有被禁止且被发送，返回 false 表示远程唤醒被禁止。

## 2. 3 BULK设备类驱动

虽然 USB BULK 类不是标准的 USB 设备类，但是 BULK 驱动提供给用户使用批量传输的高效传输数据的方法。BULK 类驱动创建了一个批量传输的接收通道以及发送通道，当接收或者发送通道映射到 buffer 模块后，可以使用 buffer 模块提供的读写接口对批量传输的收发通道进行读写操作。本驱动适用于只是传输大量数据，数据协议由用户自定义的应用程序。

由于 BULK 类的驱动不是标准的设备类，所以在使用 PC 与之通信时，需要 PC 端自定义 USB 驱动。

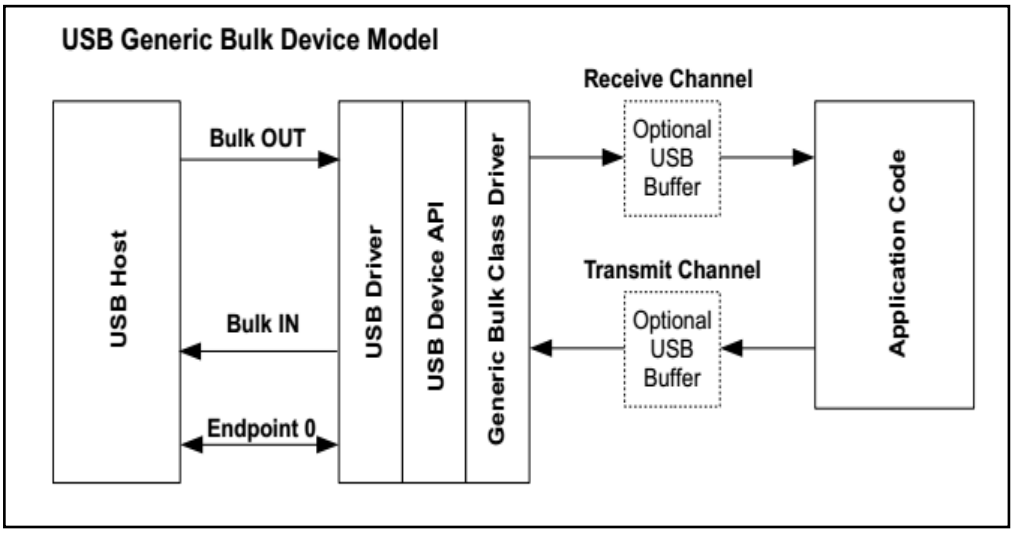


图 2-3 USB BULK 类设备驱动模型

### 2.3.1 BULK设备类事件

BULK 设备类驱动提供了以下回调事件：

#### 2.3.1.1 接收通道相关事件

- **USB_EVENT_RX_AVAILABLE**
- **USB_EVENT_ERROR**
- **USB_EVENT_CONNECTED**
- **USB_EVENT_DISCONNECTED**
- **USB_EVENT_SUSPEND**
- **USB_EVENT_RESUME**

注意，某些器件不支持断开连接检测，即不支持 USB_EVENT_DISCONNECTED 事件，具体请查阅相关数据手册。

#### 2.3.1.2 发送通道相关事件

- **USB_EVENT_TX_COMPLETE**

### 2.3.2 BULK 设备类驱动使用教程

BULK 设备类驱动使用可以遵循以下几步：

- 将以下头文件加入到需要用到此驱动的源文件中：

```
#include "usblib/usblib.h"
#include "usblib/device/usbdevice.h"
#include "usblib/device/usbdbulk.h"
#include "usblib/drivers/usb_lowlayer_api.h"
```

- 定义 6 个用来描述设备特征的 USB 字符串描述符，修改字符串内容来描述用户自己的设备，注意当字符串改变后，每个描述符数组的长度（每个数组的第一

个元素即为数组长度）也需要相应地进行修改。

```
/**
  * The languages supported by this device.
  */
const uint8_t g_pui8LangDescriptor[] =
{
  4,
  USB_DTYPE_STRING,
  USBShort(USB_LANG_EN_US)
};

/**
  * The manufacturer string.
  */
const uint8_t g_pui8ManufacturerString[] =
{
  (8 + 1) * 2,
  USB_DTYPE_STRING,
  'E', 0, 'a', 0, 's', 0, 't', 0, 'S', 0, 'o', 0, 'f', 0, 't', 0,
};

/**
  *  The product string.
  */
const uint8_t g_pui8ProductString[] =
{
  (19 + 1) * 2,
  USB_DTYPE_STRING,
  'G', 0, 'e', 0, 'n', 0, 'e', 0, 'r', 0, 'i', 0, 'c', 0, ' ', 0, 'B', 0,
  'u', 0, 'l', 0, 'k', 0, ' ', 0, 'D', 0, 'e', 0, 'v', 0, 'i', 0, 'c', 0,
  'e', 0
};

/**
  *  The serial number string.
  */
const uint8_t g_pui8SeriailNumberString[] =
{
  (8 + 1) * 2,
  USB_DTYPE_STRING,
  '1', 0, '2', 0, '3', 0, '4', 0, '5', 0, '6', 0, '7', 0, '8', 0
};

/**
```

```
    *    The interface description string.
    */
const uint8_t g_pui8InterfaceString[] =
{
    (19 + 1) * 2,
    USB_DTYPE_STRING,
    'B', 0, 'u', 0, 'l', 0, 'k', 0, ' ', 0, 'D', 0, 'a', 0, 't', 0,
    'a', 0, ' ', 0, 'I', 0, 'n', 0, 't', 0, 'e', 0, 'r', 0, 'f', 0,
    'a', 0, 'c', 0, 'e', 0
};

/**
    *    The configuration description string.
    */
const uint8_t g_pui8ConfigString[] =
{
    (23 + 1) * 2,
    USB_DTYPE_STRING,
    'B', 0, 'u', 0, 'l', 0, 'k', 0, ' ', 0, 'D', 0, 'a', 0, 't', 0,
    'a', 0, ' ', 0, 'C', 0, 'o', 0, 'n', 0, 'f', 0, 'i', 0, 'g', 0,
    'u', 0, 'r', 0, 'a', 0, 't', 0, 'i', 0, 'o', 0, 'n', 0};

/**
    *    The descriptor string table.
    */
const uint8_t * const g_ppui8StringDescriptors[] =
{
    g_pui8LangDescriptor,
    g_pui8ManufacturerString,
    g_pui8ProductString,
    g_pui8SeriailNumberString,
    g_pui8InterfaceString,
    g_pui8ConfigString
};

/**
    *    The number of string descriptors.
    */
#define NUM_STRING_DESCRIPTORS (sizeof(g_ppui8StringDescriptors) / sizeof(uint8_t *))
```

- 定义一个 **tUSBDBulkDevice** 类型的结构体用来作为 BULK 类初始化的参数。该结构体包括设备的 PID、VID、供电参数以及 BULK 设备类驱动回调函数入口、回调函数参数等信息。以下是一个简单的 BULK 类初始化结构体例子：

```
// Define CDC device.
const tUSBDCDCDevice g_sCDCDevice =
{
    // The Vendor ID you have been assigned by USB-IF.
    USB_VID_YOUR_VENDOR_ID,
    // The product ID you have assigned for this device.
    USB_PID_YOUR_PRODUCT_ID,
    // The power consumption of your device in milliamps.
    POWER_CONSUMPTION_MA,
    // The value to be passed to the host in the USB configuration descriptor's
    // bmAttributes field.
    USB_CONF_ATTR_SELF_PWR,
    // A pointer to your receive callback event handler.
    YourUSBReceiveEventCallback,
    // A value that you want passed to the receive callback alongside every
    // event.
    (void *)&g_sYourInstanceData,
    // A pointer to your transmit callback event handler.
    YourUSBTransmitEventCallback,
    // A value that you want passed to the transmit callback alongside every
    // event.
    (void *)&g_sYourInstanceData,
    // A pointer to your string table.
    g_ppui8StringDescriptors,
    // The number of entries in your string table.
    NUM_STRING_DESCRIPTORS
};
```

- 添加接收事件处理函数，如例子中 YourUSBReceiveEventCallback()函数。就 BULK 设备而言，**USB_EVENT_RX_AVAILABLE** 和 **USB_EVENT_DATA_REMAINING** 事件必须在接收事件处理函数中处理。在 **USB_EVENT_RX_AVAILABLE** 事件中，应用程序可以调用 usbd_bulk_rx_packet_available ()函数来读取目前接收到的有效数据数量，调用 usbd_bulk_packet_read ()函数来读取数据。等到设备将接收到的数据全部读取完毕，主机便可以发送下一包数据到设备,。如果当前数据设备无法立即读取，设备需要在回调函数中返回 0，这样回调函数就会在若干毫秒之后再次被调用。尽管没有其它事件被强制要求用于 BULK 设备类驱动，**USB_EVENT_CONNECTED** 和 **USB_EVENT_DISCONNECTED** 事件通常都会被用来指示设备的连接与断开。

- 添加发送事件处理函数，如例子中 YourUSBTransmitEventCallback()函数.就 BULK 设备而言，发送事件处理函数中并没有强制要求处理的事件，但是通常发送完成事件 **USB_EVENT_TX_COMPLETE** 会被用来指示 USB 端点一包数据发送完毕。

- 在主函数中调用 BULK 类初始化函数 usbd_bulk_init()来配置 USB 控制器。

usbd_bulk_init(0, & g_sBulkDevice);

- 在 USB 中断函数中调用 USB 协议栈设备中断处理函数 usb_device_int_handler();

### 2. 3. 3 BULK复合设备类驱动使用教程

当 BULK 设备驱动用于创建一个复合设备时，配置过程大致和单个 BULK 设备创建过程相似。需要注意的是，此时调用的 BULK 初始化函数已经不是 usbd_bulk_init()函数，而是使用 usbd_bulk_composite_init()函数，usbd_bulk_composite_init()函数为复合设备创建了一个用于枚举的 BULK 设备实体，此函数将初始化 **tCompositeEntry** 类型的指针来作为复合设备参数表，具体的使用方法见下面的例子：

```
// These should be initialized with valid values for each class.
extern tUSBDBulkDevice g_sBulkDevice;
void *pvBulkDevice;

tCompositeEntry g_psCompEntries[NUM_COMP_DEVICES];

// Allocate the device data for the top level composite device class.
tUSBDCompositeDevice g_sCompDevice =
{
    // VID.
    USB_VID_EASTSOFT_30CC,
    // PID.
    USB_PID_YOUR_COMPOSITE_PID,
    // This is in 2mA increments so 500mA.
    250,
    // Bus powered device.
    USB_CONF_ATTR_BUS_PWR,
    // Composite event handler.
    EventHandler,
    // The string table.
    g_pui8StringDescriptors,
    NUM_STRING_DESCRIPTORS,
    // The Composite device array.
    2,
    g_psCompEntries
};

    // Initialize the bulk instances that is part of this
    // composite device.
    g_sCompDevice.psDevices[0].pvInstance =
        usbd_bulk_composite_init(0, &g_psBULKDevice, &g_psCompEntries[0]);

    // Pass the device information to the USB library and place the device
```

```
// on the bus.
usbd_composite_init(0, &g_sCompDevice, DESCRIPTOR_DATA_SIZE,
                    g_pui8DescriptorData);
```

### 2. 3. 4 BULK类驱动接口定义

**数据结构：**

- tUSBDBulkDevice

宏：

- COMPOSITE_DBULK_SIZE

**函数接口：**

- void *usbd_bulk_composite_init(uint32_t ui32Index, tUSBDBulkDevice *psBulkDevice, tCompositeEntry * psCompEntry);
- void *usbd_bulk_init(uint32_t ui32Index, tUSBDBulkDevice *psBulkDevice);
- void usbd_bulk_term(void *pvBulkDevice);
- void *usbd_bulk_set_rx_cb_data(void *pvBulkDevice, void *pvCBData);
- void *usbd_bulk_set_tx_cb_data(void *pvBulkDevice, void *pvCBData);
- uint32_t usbd_bulk_packet_write(void *pvBulkDevice, uint8_t *pi8Data, uint32_t ui32Length, bool bLast);
- extern uint32_t usbd_bulk_packet_read(void *pvBulkDevice, uint8_t *pi8Data, uint32_t ui32Length, bool bLast);
- uint32_t usbd_bulk_tx_packet_available(void *pvBulkDevice);
- uint32_t usbd_bulk_rx_packet_available(void *pvBulkDevice);
- bool usbd_bulk_remote_wakeup_request(void *pvBulkDevice);

#### 2. 3. 4. 1 tUSBDBulkDevice

**定义：**

```
typedef struct
{
  const uint16_t ui16VID;
  const uint16_t ui16PID;
  const uint16_t ui16MaxPowermA;
  const uint8_t ui8PwrAttributes;
  const tUSBCallback pfnRxCallback;
  void *pvRxCBData;
  const tUSBCallback pfnTxCallback;
  void *pvTxCBData;
  const uint8_t *const *ppui8StringDescriptors;
```

```
    const uint32_t ui32NumStringDescriptors;
    tBulkInstance sPrivateData;
} tUSBDBulkDevice;
```

**参数定义：**

**ui16VID**：厂商 ID，由 USB-IF 统一分配。

**ui16PID**：产品 ID，由制造商分配。

**ui16MaxPowermA**：最大供电电流（mA）。

**ui8PwrAttributes**：设备供电特性，值可以是 **USB_CONF_ATTR_SELF_PWR**、**USB_CONF_ATTR_BUS_PWR**，另外可选的配置为 **USB_CONF_ATTR_RWAKE**（如果设备支持此功能）。

**pfnRxCallback**：指向接收回调的指针，该函数将被 USB 协议栈调用来通知上层应用程序与该设备接收通道相关的事件。

**pvRxCBData**：应用程序分配的指针，它将作为接收回调函数的第一个参数。

**pfnTxCallback**：指向发送回调的指针，该函数将被 USB 协议栈调用来通知上层应用程序与该设备发送通道相关的事件。

**pvTxCBData**：应用程序分配的指针，它将作为发送回调函数的第一个参数。

**ppui8StringDescriptors**：指向字符串描述符表的指针。

**ui32NumStringDescriptors**：设备字符串描述符表的大小。

**sPrivateData**：BULK 类驱动私有参数，应用程序不能对其进行操作，在应用程序中需要为此参数分配一定的存储空间。

#### 2. 3. 4. 2 COMPOSITE_DBULK_SIZE

**描述**：为单个 BULK 设备实体创建一个配置描述符应该分配的内存大小，并不包含复合设备类自动忽略的配置描述符。

#### 2. 3. 4. 3 usbd_bulk_composite_init()

当需要创建复合设备时用于 BULK 设备的初始化。

**参数定义：**

**ui32Index**：USB 控制器索引。

**psBulkDevice**：指向 BULK 初始化结构体 **tUSBDBulkDevice** 的指针。

**psCompEntry**：指向复合设备初始化入口结构体 **tCompositeEntry** 的指针，当需要创建复合设备时作为复合设备的初始化入口参数。

**简介：**

本函数和 BULK 初始化函数类似，当需要创建复合函数时，这个函数会初始化指向 **tCompositeEntry** 结构体的入口参数。

**返回值：**

失败返回 0，成功则返回指向 **tUSBDBulkDevice** 的指针，用于接下来 BULK 类驱动 API 的参数。

#### 2. 3. 4. 4 usbd_bulk_init()

初始化 BULK 设备。

**参数定义：**

**ui32Index**：USB 控制器索引。

**psBulkDevice**：指向 BULK 初始化结构体 **tUSBDBulkDevice** 的指针。

**简介：**

应用程序希望创建一个 BULK 设备与 PC 端传输数据，可以调用这个函数来初始化 USB 控制器，此函数执行所有的 BULK 设备的初始化工作。

此函数调用成功后会返回传递给它的 **tUSBDBulkDevice** 指针，这个指针必须作为以后所有的 BULK 类驱动 API 的参数。

USB BULK 类驱动的数据收发是基于 USB 数据包的（每包的传输大小和具体的传输方式有关，例如全速设备的批量传输，一包数据最大为 64 bytes），如果应用程序希望基于块的收发，可以使用 USB 协议栈提供的 buffer 模块。

当需要发送操作时，可以调用 usbd_bulk_packet_write()函数，全速设备一包数据不超过 64 bytes，高速设备一包数据不超过 512 bytes。当一包数据传输完成后，USB 协议栈会发送 **USB_EVENT_TX_COMPLETE** 事件到应用程序的回调函数中去，以通知应用程序可以进行接下来的操作。当需要接收操作时，USB 协议栈会向应用程序回调函数发送 **USB_EVENT_RX_AVAILABLE** 事件，应用程序可以调用 usbd_bulk_packet_read()函数来读取接收到的数据，接收到的数据量大小可以使用 usbd_bulk_rx_packet_available()函数来读取。

**返回值：**

失败返回 0，成功则返回指向 **tUSBDBulkDevice** 的指针，用于接下来 BULK 类驱动 API 的参数。

#### 2. 3. 4. 5 usbd_bulk_packet_read()

从 BULK 数据接口读取一包数据。

**参数定义：**

**pvBulkDevice**：指向由 usbd_bulk_init()函数返回的 BULK 初始化结构体 **tUSBDBulkDevice** 的指针。

**pi8Data**：指向数据接收的存储空间指针。

**ui32Length**：pi8Data 所指向的存储空间大小

**bLast**：指示从机是否需要进一步调用从包中读取数据，本位供 USB 协议栈 buffer

模块调度使用，当应用程序直接使用此函数读取数据时，将本位置 1 即可。

**简介：**

此函数将读取最多 ui32Length 指示的数据量的数据。当从机可以确定读取数据量的时候，bLast 参数可以忽略。

**返回值：**

返回读取到的数据数量。

#### 2. 3. 4. 6 usbd_bulk_packet_write()

向主机发送一包数据。

**参数定义：**

**pvBulkDevice**：指向由 usbd_bulk_init()函数初始化返回的 BULK 初始化结构体 **tUSBDBulkDevice** 的指针。

**pi8Data**：指向数据发送的存储空间指针。

**ui32Length**：pi8Data 所指向的存储空间大小

**bLast**：指示从机是否需要进一步调用以发送数据，如果为 true，USB 协议栈将会进一步调用此函数，如果为 false，在调度完一个短包传输后则不会再进行任何调用。本位供 USB 协议栈 buffer 模块调度使用，当应用程序直接使用此函数发送数据时，将本位置 1 即可。

**简介：**

此函数将提供数据调度功能，将一个 USB 数据包传输到 USB 主机端。如果当前没有传输正在进行，数据将立即复制到相关的 USB 端点 FIFO。如果 bLast 参数为 true，那么新写的数据包就会被安排传输。当 USB 主机确认接收到数据包后，协议栈会向应用程序发送 **USB_EVENT_TX_COMPLETE** 事件，指示可以发送接下来的数据包。

调用此函数一次最大发送的数据长度为 64 bytes。

**返回值：**

成功返回发送的数据数量，失败返回 0，表明目前不能发送数据。

#### 2. 3. 4. 7 usbd_bulk_term()

停用 BULK 驱动。

**参数定义：**

**pvBulkDevice**：指向由 usbd_bulk_init()函数初始化返回的 BULK 初始化结构体 **tUSBDBulkDevice** 的指针。

**简介：**

此函数将终止所有 BULK 类驱动操作，并从总线删除设备，如果 BULK 设备是复合设备的一部分，那么不应该调用这个函数，而应该为整个复合设备调用

usbd_composite_term()函数。在调用此函数之后，**tUSBDBulkDevice** 指针实例不应该在任何其它 API 中使用。

**返回值：**

无。

#### 2. 3. 4. 8 usbd_bulk_tx_packet_available()

返回可以发送的数据数量。

**参数定义：**

**pvBulkDevice**：指向由 usbd_bulk_init()函数初始化返回的 BULK 初始化结构体 **tUSBDBulkDevice** 的指针。

**简介：**

此函数返回 usbd_bulk_packet_write()函数可以发送的最大数据量，返回 64 的话说明目前没有数据正在传输，返回 0 说明传输正在进行。

**返回值：**

返回可以传输的数据数量。

#### 2. 3. 4. 9 usbd_bulk_rx_packet_available()

返回可以读取的数据数量。

**参数定义：**

**pvBulkDevice**：指向由 usbd_bulk_init()函数初始化返回的 BULK 初始化结构体 **tUSBDBulkDevice** 的指针。

**简介：**

此函数返回 usbd_bulk_packet_read()函数可以读取的最大数据量。

**返回值：**

返回可以读取的数据数量。

#### 2. 3. 4. 10 usbd_bulk_set_rx_cb_data()

修改 BULK 驱动的接收回调函数。

**参数定义：**

**pvBulkDevice**：指向由 usbd_bulk_init()函数初始化返回的 BULK 初始化结构体 **tUSBDBulkDevice** 的指针。

**pvCBData**：指向接收回调函数的指针。

**简介：**

应用程序使用此函数来修改由 usbd_bulk_init()函数配置好的回调函数指针。如果应用程序需要在运行时修改接收回调函数的指针值，需要保证 pvBulkDevice 存储在 SRAM

中。

**返回值：**

返回先前使用的回调函数指针值。

#### 2. 3. 4. 11 usbd_bulk_set_tx_cb_data()

修改 BULK 驱动的发送回调函数。

**参数定义：**

**pvBulkDevice**：指向由 usbd_bulk_init()函数初始化返回的 BULK 初始化结构体 **tUSBDBulkDevice** 的指针。

**pvCBData**：指向发送回调函数的指针。

**简介：**

应用程序使用此函数来修改由 usbd_bulk_init()函数配置好的回调函数指针。如果应用程序需要在运行时修改发送回调函数的指针值，需要保证 pvBulkDevice 存储在 SRAM 中。

**返回值：**

返回先前使用的回调函数指针值。

#### 2. 3. 4. 12 usbd_bulk_remote_wakeup_request()

当 USB 控制器处于挂起状态时，发送“wakeup”信号到主机。

**参数定义：**

**pvBulkDevice**：指向由 usbd_bulk_init()函数初始化返回的 BULK 初始化结构体 **tUSBDBulkDevice** 的指针。

**简介：**

当处于挂起状态时，如果从机支持远程唤醒（通过初始化结构体设置）功能的话，可以调用此函数来唤醒主机。

**返回值：**

返回 true 表示远程唤醒功能没有被禁止且被发送，返回 false 表示远程唤醒被禁止。

## 2. 4 AUDIO设备类驱动

USB AUDIO 类设备驱动创建了一个通用的音频设备驱动，帮助应用程序创建音频设备。大多数操作系统为通用的音频类设备提供本地支持，这意味着不再需要用户开发 PC 端驱动。使用 USB AUDIO 类驱动可以创建一个 16 bits 立体声、48KHz 采样率并同时支持音量和静音控制的设备。

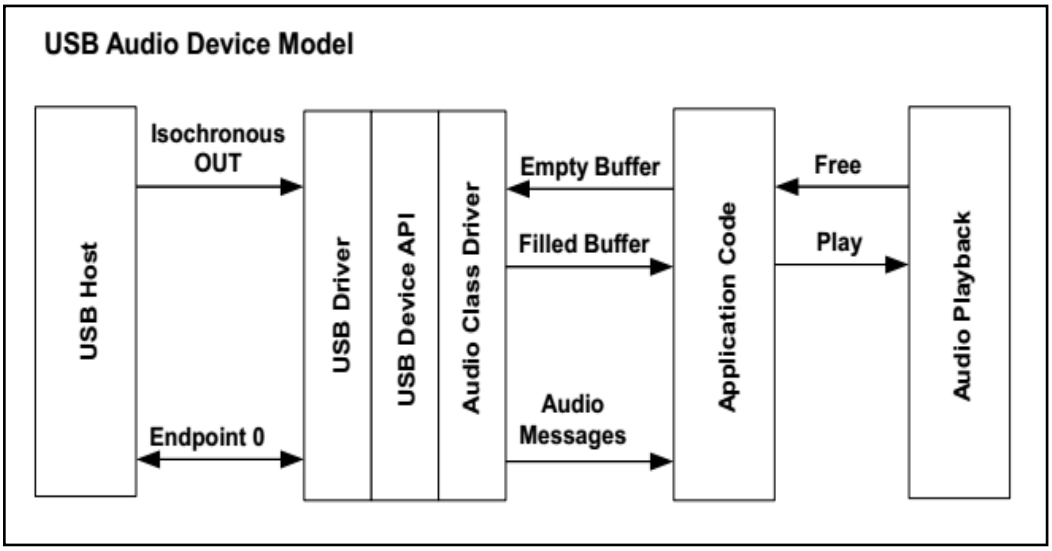


图 2-4 USB AUDIO 类设备驱动模型

### 2.4.1 AUDIO播放处理

AUDIO 驱动中提供给用户读取 PC 端传输来的音频数据的接口，应用程序可以设置音频数据输出地缓冲区。USB AUDIO 类驱动支持一个缓冲区，并在它填充满的时候返回到应用程序。由于只有一个缓冲区，所以应用程序需要谨慎处理缓冲区数据，防止缓冲区数据未读取完毕，下一帧数据将缓冲区已有的数据覆盖。由于大多数音频应用都会有两个缓冲区，一个用于正在播放的缓冲，一个用于填充缓冲。处理以上数据覆盖问题可以在应用程序中创建两块缓冲区，在每次 **USBD_AUDIO_EVENT_DATAOUT** 事件到来时，将已经传给 USB 协议栈的 AUDIO 缓冲区中的数据依次读出到应用程序中的两个缓冲区，实现音频数据的双缓冲。USB AUDIO 驱动中提供了 usb_audio_buffer_out()函数来将应用程序创建的用于音频数据缓冲的缓冲区地址以及缓冲区数据处理应用函数传递给 USB AUDIO 类驱动，协议栈通过音频数据处理回调函数将缓冲区数据返回到应用程序。如果 USB AUDIO 驱动先前已经有了一个缓冲区，则再次调用这个函数将返回失败的值。USB AUDIO 类驱动没有提供应用程序停止播放的方法，因为当 PC 端停止向设备传输音频数据后，播放便停止，而不是 **USBD_AUDIO_EVENT_DATAOUT** 事件到来时意味着播放停止。当然应用程序忽略传输来的音频数据即可停止播放。这里还有一点需要注意，**USBD_AUDIO_EVENT_DATAOUT** 事件仅仅用在 usb_audio_buffer_out()函数分配的音频数据处理回调函数，而不是初始化结构体中所设置的信息控制回调函数。

### 2.4.2 AUDIO其它信息处理

USB AUDIO 驱动还为应用程序提供了其它事件以通知应用程序，如 **USBD_AUDIO_EVENT_VOLUME** 和 **USBD_AUDIO_EVENT_MUTE** 事件。**USBD_AUDIO_EVENT_VOLUME** 事件向应用程序返回了一个范围从 0-100 的音量数值，**USBD_AUDIO_EVENT_MUTE** 事件用于控制静音，当协议栈向应用程序返回 1 时，表示静音，返回 0 表示非静音。应用程序应当妥善处理这些事件，防止程序阻塞造成音频缓冲区被下一帧数据覆盖。此外 **USBD_AUDIO_EVENT_ACTIVE** 事件还用来通知应用程序主机端将 AUDIO 设备激活；**USBD_AUDIO_EVENT_IDLE** 事件用来指示设备空闲。

### 2. 4. 3 AUDIO设备类驱动使用教程

AUDIO 设备类驱动使用可以遵循以下几步：

- 将以下头文件加入到需要用到此驱动的源文件中：

```
#include "usblib/drivers/usb_lowlayer_api.h"
#include "usblib/usblib.h"
#include "usblib/device/usbdevice.h"
#include "usblib/device/usbdaudio.h"
```

- 定义 6 个用来描述设备特征的 USB 字符串描述符，修改字符串内容来描述用户自己的设备，注意当字符串改变后，每个描述符数组的长度（每个数组的第一个元素即为数组长度）也需要相应地进行修改。

```
/**
  * The languages supported by this device.
  */
const uint8_t g_pui8LangDescriptor[] =
{
  4,
  USB_DTYPE_STRING,
  USBShort(USB_LANG_EN_US)
};

/**
  * The manufacturer string.
  */
const uint8_t g_pui8ManufacturerString[] =
{
  (8 + 1) * 2,
  USB_DTYPE_STRING,
  'E', 0, 'a', 0, 's', 0, 't', 0, 'S', 0, 'o', 0, 'f', 0, 't', 0,
};

/**
  *   The product string.
  */
const uint8_t g_pui8ProductString[] =
{
  (20 + 1) * 2,
  USB_DTYPE_STRING,
  'G', 0, 'e', 0, 'n', 0, 'e', 0, 'r', 0, 'i', 0, 'c', 0, ' ', 0, 'A', 0,
  'u', 0, 'd', 0, 'i', 0, 'o', 0, ' ', 0, 'D', 0, 'e', 0, 'v', 0, 'i', 0,
  'c', 0, 'e', 0
};
```

```
/**
  *  The serial number string.
  */
const uint8_t g_pui8SeriailNumberString[] =
{
  (8 + 1) * 2,
  USB_DTYPE_STRING,
  '1', 0, '2', 0, '3', 0, '4', 0, '5', 0, '6', 0, '7', 0, '8', 0
};

/**
  *  The interface description string.
  */
const uint8_t g_pui8InterfaceString[] =
{
  (20 + 1) * 2,
  USB_DTYPE_STRING,
  'A', 0, 'u', 0, 'd', 0, 'i', 0, 'o', 0, ' ', 0, 'D', 0, 'a', 0,
  't', 0, 'a', 0, ' ', 0, 'I', 0, 'n', 0, 't', 0, 'e', 0, 'r', 0,
  'f', 0, 'a', 0, 'c', 0, 'e', 0
};

/**
  *  The configuration description string.
  */
const uint8_t g_pui8ConfigString[] =
{
  (24 + 1) * 2,
  USB_DTYPE_STRING,
  'A', 0, 'u', 0, 'd', 0, 'i', 0, 'o', 0, ' ', 0, 'D', 0, 'a', 0,
  't', 0, 'a', 0, ' ', 0, 'C', 0, 'o', 0, 'n', 0, 'f', 0, 'i', 0,
  'g', 0, 'u', 0, 'r', 0, 'a', 0, 't', 0, 'i', 0, 'o', 0, 'n', 0
};

/**
  *  The descriptor string table.
  */
const uint8_t * const g_ppui8StringDescriptors[] =
{
  g_pui8LangDescriptor,
  g_pui8ManufacturerString,
  g_pui8ProductString,
  g_pui8SeriailNumberString,
  g_pui8InterfaceString,
```

```
    g_pui8ConfigString
};

/**
 * The number of string descriptors.
 */
#define NUM_STRING_DESCRIPTORS (sizeof(g_ppui8StringDescriptors) / sizeof(uint8_t *))
```

- 定义一个 **tUSBDAudioDevice** 类型的结构体用来作为 AUDIO 类初始化的参数。该结构体包括设备的 PID、VID、供电参数以及 AUDIO 设备类驱动回调函数入口、回调函数参数等信息。以下是一个简单的 AUDIO 类初始化结构体例子：

```
const tUSBDAudioDevice g_sAudioDevice =
{
    // The Vendor ID you have been assigned by USB-IF.
    USB_VID_YOUR_VENDOR_ID,
    // The product ID you have assigned for this device.
    USB_PID_YOUR_PRODUCT_ID,
    // The vendor string for your device (8 chars).
    USB_YOUR_VENDOR_STRING,
    // The product string for your device (16 chars).
    USB_YOUR_PRODUCT_STRING,
    // The revision string for your device (4 chars BCD).
    USB_YOUR_REVISION_STRING,
    // The power consumption of your device in milliamps.
    POWER_CONSUMPTION_MA,
    // The value to be passed to the host in the USB configuration descriptor's
    // bmAttributes field.
    USB_CONF_ATTR_SELF_PWR,
    // A pointer to your control callback event handler.
    YourUSBAudioMessageHandler,
    // A pointer to your string table.
    g_ppui8StringDescriptors,
    // The number of entries in your string table.
    NUM_STRING_DESCRIPTORS,
    // Maximum volume setting expressed as and 8.8 signed fixed point number.
    VOLUME_MAX,
    // Minimum volume setting expressed as and 8.8 signed fixed point number.
    VOLUME_MIN,
    // Minimum volume step expressed as and 8.8 signed fixed point number.
    VOLUME_STEP
};
```

如例子中 YourUSBAudioMessageHandler()函数。尽管没有其它事件被强制要求用于 AUDIO 设备类驱动，**USB_EVENT_CONNECTED** 和 **USB_EVENT_DISCONNECTED** 事件通常都会被用来指示设备的连接与断开。

- 在主函数中调用 AUDIO 类初始化函数 usbd_audio_init()来配置 USB 控制器。usbd_audio_init(0, & g_sAudioDevice);
- 在 USB 中断函数中调用 USB 协议栈设备中断处理函数 usb_device_int_handler()；
- 一旦主机端激活 AUDIO 设备，协议栈将向应用程序通知 **USBD_AUDIO_EVENT_ACTIVE** 事件。

### 2. 4. 4 AUDIO复合设备类驱动使用教程

当 AUDIO 设备驱动用于创建一个复合设备时，配置过程大致和单个 AUDIO 设备创建过程相似。需要注意的是，此时调用的 AUDIO 初始化函数已经不是 usbd_audio_init()函数，而是使用 usbd_audio_composite_init()函数，usbd_audio_composite_init()函数为复合设备创建了一个用于枚举的 AUDIO 设备实体，此函数将初始化 **tCompositeEntry** 类型的指针来作为复合设备参数表，具体的使用方法见下面的例子：

```
// These should be initialized with valid values for each class.
extern tUSBDAudioDevice g_sAudioDevice;
void *pvAudioDevice;

tCompositeEntry g_psCompEntries[NUM_COMP_DEVICES];

// Allocate the device data for the top level composite device class.
tUSBDCompositeDevice g_sCompDevice =
{
    // VID.
    USB_VID_EASTSOFT_30CC,
    // PID.
    USB_PID_YOUR_COMPOSITE_PID,
    // This is in 2mA increments so 500mA.
    250,
    // Bus powered device.
    USB_CONF_ATTR_BUS_PWR,
    // Composite event handler.
    EventHandler,
    // The string table.
    g_pui8StringDescriptors,
    NUM_STRING_DESCRIPTORS,
    // The Composite device array.
    2,
    g_psCompEntries
};

    // Initialize the audio instances that is part of this
    // composite device.
    g_sCompDevice.psDevices[0].pvInstance =
```

```
    usbd_audio_composite_init(0, & g_sAudioDevice, &g_psCompEntries[0]);

// Pass the device information to the USB library and place the device
// on the bus.
usbd_composite_init(0, &g_sCompDevice, DESCRIPTOR_DATA_SIZE,
                    g_pui8DescriptorData);
```

### 2. 4. 5 AUDIO类驱动接口定义

**数据结构：**

- tUSBDAudioDevice

宏：

- COMPOSITE_DAUDIO_SIZE

**函数接口：**

- void * usbd_audio_composite_init(uint32_t ui32Index, tUSBDAudioDevice * psAudioDevice, tCompositeEntry * psCompEntry);
- void * usbd_audio_init(uint32_t ui32Index, tUSBDAudioDevice*psAudioDevice);
- void usbd_audio_term(void * psAudioDevice);
- int32_t usb_audio_buffer_out(void *pvAudioDevice, void *pvBuffer, uint32_t ui32Size, tUSBAudioBufferCallback pfnCallback);

#### 2. 4. 5. 1 tUSBDAudioDevice

定义：

```
typedef struct
{
    const uint16_t ui16VID;
    const uint16_t ui16PID;
    const char pcVendor[8];
    const char pcProduct[16];
    const char pcVersion[4];
    const uint16_t ui16MaxPowermA;
    const uint8_t ui8PwrAttributes;
    const tUSBCallback pfnCallback;
    const uint8_t *const *ppui8StringDescriptors;
    const uint32_t ui32NumStringDescriptors;
    const int16_t i16VolumeMax;
    const int16_t i16VolumeMin;
    const int16_t i16VolumeStep;
    tAudioInstance sPrivateData;
} tUSBDAudioDevice;
```

**参数定义：**

**ui16VID**：厂商 ID，由 USB-IF 统一分配。

**ui16PID**：产品 ID，由制造商分配。

**pcVendor**：8 Bytes 制造商字符串（创建初始化结构体时可以不赋值）。

**pcProduct**：16 Bytes 产品字符串（创建初始化结构体时可以不赋值）。

**pcVersion**：4 Bytes 版本字符串（创建初始化结构体时可以不赋值）。

**ui16MaxPowermA**：最大供电电流（mA）。

**ui8PwrAttributes**：设备供电特性，值可以是 **USB_CONF_ATTR_SELF_PWR**、**USB_CONF_ATTR_BUS_PWR**，另外可选的配置为 **USB_CONF_ATTR_RWAKE**（如果设备支持此功能）。

**pfnCallback**：指向接收回调函数的指针，该函数将被 USB 协议栈调用来通知上层应用程序与 AUDIO 设备相关的事件。

**ppui8StringDescriptors**：指向字符串描述符表的指针。

**ui32NumStringDescriptors**：设备字符串描述符表的大小。

**i16VolumeMax**：设备支持的最大音量。

**i16VolumeMin**：设备支持的最小音量。

**i16VolumeStep**：音量最小步阶。

**sPrivateData**：AUDIO 类驱动私有参数，应用程序不能对其进行操作，在应用程序中需要为此参数分配一定的存储空间。

#### 2. 4. 5. 2 COMPOSITE_DAUDIO_SIZE

**描述**：为单个 AUDIO 设备实体创建一个配置描述符应该分配的内存大小，并不包含复合设备类自动忽略的配置描述符。

#### 2. 4. 5. 3 usbd_audio_composite_init()

当需要创建复合设备时用于 AUDIO 设备的初始化。

**参数定义：**

**ui32Index**：USB 控制器索引。

**psAudioDevice**：指向 AUDIO 初始化结构体 **tUSBDAudioDevice** 的指针。

**psCompEntry**：指向复合设备初始化入口结构体 **tCompositeEntry** 的指针，当需要创建复合设备时作为复合设备的初始化入口参数。

**简介：**

本函数和 AUDIO 初始化函数类似，当需要创建复合函数时，这个函数会初始化指向 **tCompositeEntry** 结构体的入口参数，用户程序需要提供给此函数一个合法的指向

**tUSBDAudioDevice** 的结构体。

**返回值：**

失败返回 0，成功则返回指向 **tUSBDAudioDevice** 的指针（但是此时类型是 void *），用于接下来 AUDIO 类驱动 API 的参数。

#### 2. 4. 5. 4 usbd_audio_init()

初始化 AUDIO 设备。

**参数定义：**

**ui32Index**：USB 控制器索引。

**psAudioDevice**：指向 AUDIO 初始化结构体 **tUSBDAudioDevice** 的指针。

**简介：**

应用程序希望创建一个 AUDIO 设备与 PC 端传输数据，可以调用这个函数来初始化 USB 控制器，此函数执行所有的 AUDIO 设备的初始化工作。

此函数调用成功后会返回传递给它的 **tUSBDBulkDevice** 指针（实际返回类型为 void *），这个指针必须作为以后所有的 AUDIO 类驱动 API 的参数。

**返回值：**

失败返回 0，成功则返回指向 **tUSBDAudioDevice** 的指针，用于接下来 AUDIO 类驱动 API 的参数。

#### 2. 4. 5. 5 usbd_audio_term()

停用 AUDIO 驱动。

**参数定义：**

**psAudioDevice**：指向由 usbd_audio_init()函数初始化返回的 AUDIO 初始化结构体 **tUSBDAudioDevice** 的指针。

**简介：**

此函数将终止所有 AUDIO 类驱动操作，并从总线删除设备，如果 AUDIO 设备是复合设备的一部分，那么不应该调用这个函数，而应该为整个复合设备调用 usbd_composite_term()函数。在调用此函数之后，**tUSBDAudioDevice** 指针实例不应该在任何其它 API 中使用。

**返回值：**

无。

#### 2. 4. 5. 6 usb_audio_buffer_out()

应用程序使用此函数向 USB 协议栈设置音频数据缓冲区地址以及音频数据处理回调函数。

**参数定义：**

**psAudioDevice**：指向 AUDIO 初始化结构体 **tUSBDAudioDevice** 的指针。

**pvBuffer**：指向音频数据缓冲区。

**ui32Size**：音频数据缓冲区大小。

**pfnCallback**：音频数据处理回调函数。

**简介：**

本函数由应用程序调用，向 USB 协议栈传入用于缓冲音频数据的相关信息，调用此函数后缓冲区将指向 pvBuffer 所指向的区域，最大缓冲数据量为 ui32Size 指示的大小。ui32Size 的值最大不能超过一个 USB 同步传输的端点最大 FIFO 存储大小，即不能超过 **ISOC_OUT_EP_MAX_SIZE** 字节。由于音频数据可能没有刚好填充完缓冲区，pfnCallback()函数将提供实际的有效数据数量，此时音频数据处理函数的 ui32Param 参数即代表所读取到的音频数据数量。

**返回值：**

成功返回 0，其它值代表所提供的缓冲区不能被填充。

## 2.5 MSC设备类驱动

USB MSC 设备类允许应用程序充当一个物理存储设备，用于另一个 USB 应用程序或主机操作系统。由于每个应用程序的存储类型会有所不同，所以 MSC 类将存储驱动接口是基于块的抽象 API。这些 API 允许 USB MSC 类调用实际上在物理存储媒体上执行操作的外部函数集。所有与 MSC 类的交互都是通过向 USB 协议栈的 MSC 类接口提供的回调函数发生的，当设备正在读取、写入或空闲时，用于通知应用程序回调函数。USB 协议栈的 MSC 类也提供了一个接口，当媒体状态发生变化时，将会通知应用程序。

### 2.5.1 MSC设备类驱动使用教程

MSC 设备类驱动使用可以遵循以下几步：

- 将以下头文件加入到需要用到此驱动的源文件中：

```
#include "usblib/usblib.h"
#include "usblib/device/usbdevice.h"
#include "usblib/device/ usbdmsc.h"
#include "usblib/drivers/usb_lowlayer_api.h"
```

- 定义 6 个用来描述设备特征的 USB 字符串描述符，修改字符串内容来描述用户自己的设备，注意当字符串改变后，每个描述符数组的长度（每个数组的第一个元素即为数组长度）也需要相应地进行修改。

```
/**
  * The languages supported by this device.
  */
const uint8_t g_pui8LangDescriptor[] =
{
  4,
  USB_DTYPE_STRING,
```

```
    USBShort(USB_LANG_EN_US)
};

/**
  * The manufacturer string.
  */
const uint8_t g_pui8ManufacturerString[] =
{
  (8 + 1) * 2,
  USB_DTYPE_STRING,
  'E', 0, 'a', 0, 's', 0, 't', 0, 'S', 0, 'o', 0, 'f', 0, 't', 0,
};

/**
  *  The product string.
  */
const uint8_t g_pui8ProductString[] =
{
  (18 + 1) * 2,
  USB_DTYPE_STRING,
  'G', 0, 'e', 0, 'n', 0, 'e', 0, 'r', 0, 'i', 0, 'c', 0, ' ', 0, 'M', 0,
  'S', 0, 'C', 0, ' ', 0, 'D', 0, 'e', 0, 'v', 0, 'i', 0, 'c', 0,'e', 0
};

/**
  *  The serial number string.
  */
const uint8_t g_pui8SeriailNumberString[] =
{
  (8 + 1) * 2,
  USB_DTYPE_STRING,
  '1', 0, '2', 0, '3', 0, '4', 0, '5', 0, '6', 0, '7', 0, '8', 0
};

/**
  *  The interface description string.
  */
const uint8_t g_pui8InterfaceString[] =
{
  (18 + 1) * 2,
  USB_DTYPE_STRING,
  'M', 0, 'S', 0, 'C', 0, ' ', 0, 'D', 0, 'a', 0, 't', 0,
  'a', 0, ' ', 0, 'I', 0, 'n', 0, 't', 0, 'e', 0, 'r', 0, 'f', 0,
  'a', 0, 'c', 0, 'e', 0
```

```
};

/**
  *  The configuration description string.
  */
const uint8_t g_pui8ConfigString[] =
{
  (22 + 1) * 2,
  USB_DTYPE_STRING,
  'M', 0, 'S', 0, 'C', 0, ' ', 0, 'D', 0, 'a', 0, 't', 0,
  'a', 0, ' ', 0, 'C', 0, 'o', 0, 'n', 0, 'f', 0, 'i', 0, 'g', 0,
  'u', 0, 'r', 0, 'a', 0, 't', 0, 'i', 0, 'o', 0, 'n', 0};

/**
  *  The descriptor string table.
  */
const uint8_t * const g_ppui8StringDescriptors[] =
{
  g_pui8LangDescriptor,
  g_pui8ManufacturerString,
  g_pui8ProductString,
  g_pui8SeriailNumberString,
  g_pui8InterfaceString,
  g_pui8ConfigString
};

/**
  *  The number of string descriptors.
  */
#define NUM_STRING_DESCRIPTORS (sizeof(g_ppui8StringDescriptors) / sizeof(uint8_t *))
```

- 定义一个 **tUSBDMSCDevice** 类型的结构体用来作为 MSC 类初始化的参数。该结构体包括设备的 PID、VID、供电参数以及 MSC 设备类驱动回调函数入口、回调函数参数等信息。以下是一个简单的 MSC 类初始化结构体例子：

```
const tUSBDMSCDevice g_sMSCDevice =
{
    // Vendor ID.
    USB_VID_EASTSOFT_30CC,
    // Product ID.
    USB_PID_MSC,
    // Vendor Information.
    "EastSoft",
    // Product Identification.
    "Mass Storage ",
    // Revision.
```

```
    "1.00",
    // 200mA.
    200,
    // Bus Powered.
    USB_CONF_ATTR_BUS_PWR,
    // A list of string descriptors and the number of descriptors.
    g_pStringDescriptors,
    NUM_STRING_DESCRIPTORS,
    // The media access functions.
    {
        USBDMSCStorageOpen,
        USBDMSCStorageClose,
        USBDMSCStorageRead,
        USBDMSCStorageWrite,
        USBDMSCStorageNumBlocks
    },
    // The event notification call back function.
    USBMSCDEventCallback,
};
```

■ 添加事件处理函数来处理以及通知应用程序 MSC 类的状态变化。

```
uint32_t
USBDMSCEventCallback(void *pvCBData, uint32_t ui32Event,
uint32_t ui32MsgParam, void *pvMsgData)
{
    switch(ui32Event)
    {
        // Writing to the device.
        case USBD_MSC_EVENT_WRITING:
        {
        // Handle write case.
        ...
        break;
        }
        // Reading from the device.
        case USBD_MSC_EVENT_READING:
        {
        // Handle read case.
        ...
        break;
        }
        case USBD_MSC_EVENT_IDLE:
        {
        // Handle idle case.
        ...
```

```
            }
            default:
            {
            break;
            }
        }
        return(0);
}
```

- 在主函数中调用 MSC 类初始化函数 usbdmsc_init()来配置 USB 控制器。usbdmsc_init(0, & g_sMSCDevice);
- 在 USB 中断函数中调用 USB 协议栈设备中断处理函数 usb_device_int_handler()；

### 2. 5. 2 MSC复合设备类驱动使用教程

当 MSC 设备驱动用于创建一个复合设备时，配置过程大致和单个 MSC 设备创建过程相似。需要注意的是，此时调用的 MSC 初始化函数已经不是 usbdmsc_init()函数，而是使用 usbdmsc_composite_init()函数，usbdmsc_composite_init()函数为复合设备创建了一个用于枚举的 MSC 设备实体，此函数将初始化 **tCompositeEntry** 类型的指针来作为复合设备参数表，具体的使用方法见下面的例子：

```
// These should be initialized with valid values for each class.
extern tUSBDMSCDevice g_sMSCDevice;
// The array of composite device entries.
tCompositeEntry psCompEntries[2];
// Allocate the device data for the top level composite device class.
tUSBDCompositeDevice g_sCompDevice =
{
    // VID.
    USB_VID_EASTSOFT_30CC,
    // PID for composite serial device.
    USB_PID_YOUR_COMPOSITE_PID,
    // This is in 2mA increments so 500mA.
    250,
    // Bus powered device.
    USB_CONF_ATTR_BUS_PWR,
    // Composite event handler.
    EventHandler,
    // The string table.
    g_pui8StringDescriptors,
    NUM_STRING_DESCRIPTORS,
    // The Composite device array.
    2,
    g_psCompEntries
};
```

```
// The OTHER_SIZES here are the sizes of the descriptor data for other classes
// that are part of the composite device.
#define DESCRIPTOR_DATA_SIZE (COMPOSITE_DMSC_SIZE + OTHER_SIZES)
uint8_t g_pui8DescriptorData[DESCRIPTOR_DATA_SIZE];
// Save the instance data for this mass storage device.
sMSCDevice = usbdmsc_composite_init(0, &g_sMSCDevice, &psCompEntries[0]);
...
// Initialize the USB controller as a composite device.
usbd_composite_init(0, &g_sCompDevice, DESCRIPTOR_DATA_SIZE,
g_pui8DescriptorData);
```

### 2. 5. 3 MSC类驱动接口定义

**数据结构：**

- tMSCDMedia
- tUSBDMSCDevice

宏：

- COMPOSITE_DMSC_SIZE
- USBD_MSC_EVENT_IDLE
- USBD_MSC_EVENT_READING
- USBD_MSC_EVENT_WRITING

**函数接口：**

- void *usbdmsc_init(uint32_t ui32Index, tUSBDMSCDevice *psMSCDevice);
- void *usbdmsc_composite_init(uint32_t ui32Index, tUSBDMSCDevice *psMSCDevice, tCompositeEntry *psCompEntry);
- void usbdmsc_term(void *pvInstance);
- void usbdmsc_media_change(void *pvInstance, tUSBDMSCMediaStatus eMediaStatus);

#### 2. 5. 3. 1 tMSCDMedia

定义：

```
typedef struct
{
    void *(*pfnOpen)(uint32_t ui32Drive);
    void (*pfnClose)(void *pvDrive);
    uint32_t (*pfnBlockRead)(void *pvDrive, uint8_t *pui8Data,
                             uint32_t ui32Sector, uint32_t ui32NumBlocks);
    uint32_t (*pfnBlockWrite)(void *pvDrive, uint8_t *pui8Data,
                              uint32_t ui32Sector, uint32_t ui32NumBlocks);
```

```
    uint32_t (*pfnNumBlocks)(void *pvDrive);
    uint32_t (*pfnBlockSize)(void *pvDrive);

}
tMSCDMedia;
```

**参数定义：**

**pfnOpen**：该函数用于初始化和打开与参数 ui32Drive 相关的物理驱动器号。如果驱动器不能以某种原因打开，函数将返回零。在使用可移动设备的情况下如 SD 卡，如果 SD 卡不存在，这个函数必须返回零。该函数返回一个指向数据的传递给其他 API 的指针，如果没有发现驱动器，则应该返回 0。

**pfnClose**：这个函数关闭了 MSC 类设备使用的驱动器号。pvDrive 是从调用 pfnOpen 返回的指针。该函数用于关闭与参数 pvDrive 相关的物理驱动器号。如果驱动器成功关闭此函数将返回 0，任何其他值表示失败。

**pfnBlockRead**：该函数读取由 pfnOpen 调用打开的设备上的数据块。pvDrive 参数是从原始调用 pfnOpen 返回的指针。pui8Data 参数是将数据写入的缓冲区，由 pui8Data 指向的数据区域必须至少是 ui32NumBlocks *块大小的字节。ui32Sector 是读取的块地址，ui32numblock 是读取的块数。这个函数返回放置到 pui8Data 缓冲区中读取的字节数。

**pfnBlockWrite**：这个函数用于将 pui8Data 指向的缓冲区块写入到由 pvDrive 指向的物理设备。pvDrive 参数是从原始调用 pfnOpen 返回的指针。ui32NumBlocks 是写入的块数。ui32Sector 参数是用来写块的扇区号。这个函数返回被写入设备的字节数。

**pfnNumBlocks**：这个函数根据 pvDrive 参数返回物理设备上块的总数。pvDrive 参数是从原始调用 pfnOpen 返回的指针。

**pfnBlockSize**：这个函数根据 pvDrive 参数返回一个物理设备的块大小。pvDrive 参数是从原始调用 pfnOpen 返回的指针。

#### 2. 5. 3. 2 tUSBDMSCDevice

**定义：**

```
typedef struct
{
    const uint16_t ui16VID;
    const uint16_t ui16PID;
    const uint8_t pui8Vendor[8];
    const uint8_t pui8Product[16];
    const uint8_t pui8Version[4];
    const uint16_t ui16MaxPowermA;
    const uint8_t ui8PwrAttributes;
    const uint8_t *const *ppui8StringDescriptors;
    const uint32_t ui32NumStringDescriptors;
    const tMSCDMedia sMediaFunctions;
    const tUSBCallback pfnEventCallback;
```

```
    tMSCInstance sPrivateData;
}
tUSBDMSCDevice;
```

**参数定义：**

**ui16VID**：厂商 ID，由 USB-IF 统一分配。

**ui16PID**：产品 ID，由制造商分配。

**pcVendor**:8 Bytes 制造商字符串（创建初始化结构体时可以不赋值）。

**pcProduct**：16 Bytes 产品字符串（创建初始化结构体时可以不赋值）。

**pcVersion**：4 Bytes 版本字符串（创建初始化结构体时可以不赋值）。

**ui16MaxPowermA**：最大供电电流（mA）。

**ui8PwrAttributes**：设备供电特性，值可以是 **USB_CONF_ATTR_SELF_PWR**、**USB_CONF_ATTR_BUS_PWR**，另外可选的配置为 **USB_CONF_ATTR_RWAKE**（如果设备支持此功能）。

**ppui8StringDescriptors**：指向字符串描述符表的指针。

**ui32NumStringDescriptors**：设备字符串描述符表的大小。

**sMediaFunctions**：这个结构为 MSC 类设备的实例使用的媒体的访问函数。在这个结构体中所有的函数都需要用有效的函数来赋值。

**pfnEventCallback**：指向接收回调函数的指针，该函数将被 USB 协议栈调用来通知上层应用程序与 MSC 设备相关的事件。

**sPrivateData**：MSC 类驱动私有参数，应用程序不能对其进行操作，在应用程序中需要为此参数分配一定的存储空间。

#### 2. 5. 3. 3 COMPOSITE_DMSC_SIZE

**描述**：为单个 MSC 设备实体创建一个配置描述符应该分配的内存大小，并不包含复合设备类自动忽略的配置描述符。

#### 2. 5. 3. 4 USBD_MSC_EVENT_IDLE

**描述**：这个事件表明主机已经完成了其他操作，并不再访问设备。

#### 2. 5. 3. 5 USBD_MSC_EVENT_READING

**描述**：指示主机正在读取存储器。

#### 2. 5. 3. 6 USBD_MSC_EVENT_WRITING

**描述**：指示主机正在写存储器。

#### 2. 5. 3. 7 usbdmsc_composite_init()

当需要创建复合设备时用于 MSC 类设备的初始化。

**参数定义：**

**ui32Index**：USB 控制器索引。

**psMSCDevice**：指向 MSC 初始化结构体 **tUSBDMSCDevice** 的指针。

**psCompEntry**：指向复合设备初始化入口结构体 **tCompositeEntry** 的指针，当需要创建复合设备时作为复合设备的初始化入口参数。

**简介：**

本函数和 MSC 类初始化函数类似，当需要创建复合函数时，这个函数会初始化指向 **tCompositeEntry** 结构体的入口参数。

**返回值：**

失败返回 0，成功则返回指向 **tUSBDMSCDevice** 的指针，用于接下来 MSC 类驱动 API 的参数。

#### 2. 5. 3. 8 usbdmsc_init()

初始化 MSC 设备。

**参数定义：**

**ui32Index**：USB 控制器索引。

**psMSCDevice**：指向 MSC 初始化结构体 **tUSBDMSCDevice** 的指针。

**简介：**

应用程序希望创建一个 MSC 类设备与 PC 端传输数据，可以调用这个函数来初始化 USB 控制器，此函数执行所有的 MSC 类设备的初始化工作。

此函数调用成功后会返回传递给它的 **tUSBDMSCDevice** 指针，这个指针必须作为以后所有的 MSC 类驱动 API 的参数。

**返回值：**

失败返回 0，成功则返回指向 **tUSBDMSCDevice** 的指针，用于接下来 MSC 类驱动 API 的参数。

#### 2. 5. 3. 9 usbdmsc_media_change()

用于删除和插入 USB 存储设备。

**参数定义：**

**pvInstance**：MSC 类设备实例，指向 **tUSBDMSCDevice** 的指针。

**eMediaStatus**：更新的状态。

**简介：**

当检测到媒体在 USB MSC 类使用的状态的变化时，该函数应该被应用程序调用。iMediaStatus 参数可用的值为：

- **eUSBDMSCMediaPresent** 指示设备存在。

- **eUSBDMSCMediaNotPresent** 指示设备不存在。
- **eUSBDMSCMediaUnknown** 指示设备无法识别。

**返回值：**

无。

#### 2. 5. 3. 10 usbdmsc_term()

停用 MSC 驱动。

**参数定义：**

**pvInstance**：指向由 usbdmsc_init()函数初始化返回的 MSC 初始化结构体 **tUSBDMSCDevice** 的指针。

**简介：**

此函数将终止所有 MSC 类驱动操作，并从总线删除设备，如果 MSC 设备是复合设备的一部分，那么不应该调用这个函数，而应该为整个复合设备调用 usbd_composite_term()函数。在调用此函数之后，**tUSBDMSCDevice** 指针实例不应该在任何其它 API 中使用。

**返回值：**

无。

## 2. 6 HID设备类驱动

USB HID 接口是一种多功能的架构，用于支持各种各样的输入输出设备。虽然通常认为键盘、鼠标、和操纵杆等为人机控制的设备，但是 HID 设备类规范几乎可以覆盖任何的控制或者数据传输的设备。

HID 设备和主机之间的通信是通过 HID 报表来完成的，HID 报表由 HID 报告描述符所定义。HID 报表支持设备传输给主机数据，同时也支持主机向设备输出数据。

除了 HID 的基本架构灵活外，HID 设备的开发及使用是不需要用户开发 PC 端驱动的，基本上现有的操作系统都会支持 HID 设备，即使某些厂商定义的特殊 HID 设备也不需要特别复杂的 PC 端驱动开发。

尽管优势比较明显，但是 HID 类设备还是存在一些自身的限制，HID 每个时间间隙所能传输的数据量是有限的，因此 HID 设备不适合高带宽的数据传输设备或者大数据量的传输设备。HID 全速设备每毫秒的数据量仅为 64 Bytes（中断传输的限制），如果需要传输比较大的数据量的话建议应用程序考虑 CDC 或者 BULK 类驱动。

HID 类设备使用一个或多个除端点 0 以外的端点，主机到设备的特性和报告通常是通过端点 0 来进行的，需要高速率的主机到设备的数据时可以选择一个独立的中断端点来进行传输。端点 0 携带标准的 USB 请求和特定的描述符请求。

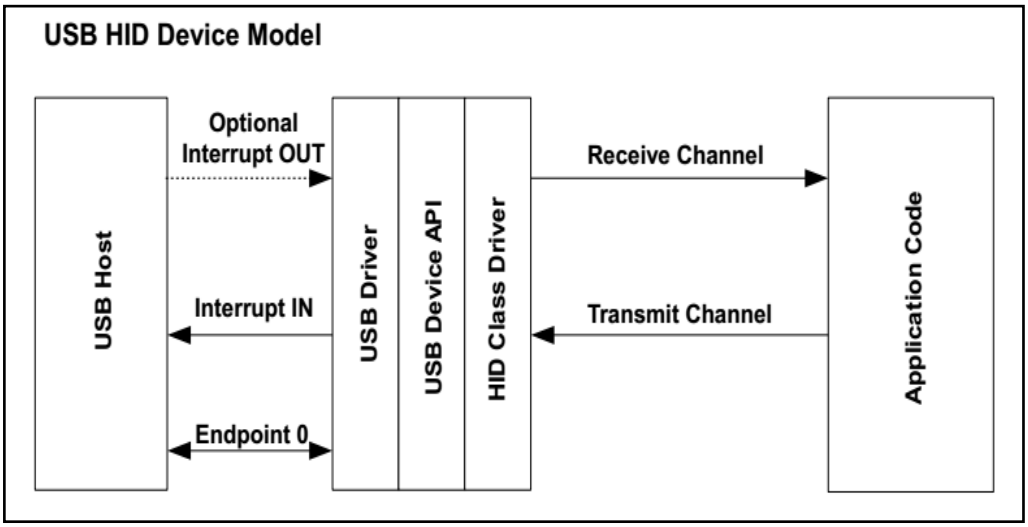


图 2-5 USB HID 类设备驱动模型

### 2.6.1 HID设备类事件

HID 设备类驱动提供了以下回调事件：

#### 2.6.1.1 接收通道相关事件

- **USB_EVENT_RX_AVAILABLE**
- **USB_EVENT_ERROR**
- **USB_EVENT_CONNECTED**
- **USB_EVENT_DISCONNECTED**
- **USB_EVENT_SUSPEND**
- **USB_EVENT_RESUME**
- **USBD_HID_EVENT_IDLE_TIMEOUT**
- **USBD_HID_EVENT_GET_REPORT_BUFFER**
- **USBD_HID_EVENT_GET_REPORT**
- **USBD_HID_EVENT_SET_PROTOCOL**
- **USBD_HID_EVENT_GET_PROTOCOL**
- **USBD_HID_EVENT_SET_REPORT**
- **USBD_HID_EVENT_REPORT_SENT**

注：

1. **USB_EVENT_DISCONNECTED** 事件仅在某些器件内支持，具体请查阅相关器件手册，确定设备模式下是否有断开连接检测功能。

#### 2.6.1.2 发送通道相关事件

- **USB_EVENT_TX_COMPLETE**

### 2. 6. 2 HID设备类驱动使用教程

HID 设备类驱动使用可以遵循以下几步：

- 将以下头文件加入到需要用到此驱动的源文件中：

```
#include "usblib/drivers/usb_lowlayer_api.h"
#include "usblib/usblib.h"
#include "usblib/usbhid.h"
#include "usblib/device/usbdevice.h"
#include "usblib/device/usbdhid.h"
```

- 定义 6 个用来描述设备特征的 USB 字符串描述符，修改字符串内容来描述用户自己的设备，注意当字符串改变后，每个描述符数组的长度（每个数组的第一个元素即为数组长度）也需要相应地进行修改。这里以鼠标的例程的字符串描述符为例：

```
/**
  * The languages supported by this device.
  */
const uint8_t g_pui8LangDescriptor[] =
{
  4,
  USB_DTYPE_STRING,
  USBShort(USB_LANG_EN_US)
};

/**
  * The manufacturer string.
  */
const uint8_t g_pui8ManufacturerString[] =
{
  (8 + 1) * 2,
  USB_DTYPE_STRING,
  'E', 0, 'a', 0, 's', 0, 't', 0, 'S', 0, 'o', 0, 'f', 0, 't', 0,
};

/**
  *  The product string.
  */
const uint8_t g_pui8ProductString[] =
{
  (13 + 1) * 2,
  USB_DTYPE_STRING,
  'M', 0, 'o', 0, 'u', 0, 's', 0, 'e', 0, ' ', 0, 'E', 0, 'x', 0, 'a', 0,
  'm', 0, 'p', 0, 'l', 0, 'e', 0
};
```

```
/**
  *  The serial number string.
  */
const uint8_t g_pui8SeriailNumberString[] =
{
  (8 + 1) * 2,
  USB_DTYPE_STRING,
  '1', 0, '2', 0, '3', 0, '4', 0, '5', 0, '6', 0, '7', 0, '8', 0
};

/**
  *  The interface description string.
  */
const uint8_t g_pui8InterfaceString[] =
{
  (19 + 1) * 2,
  USB_DTYPE_STRING,
  'H', 0, 'I', 0, 'D', 0, ' ', 0, 'M', 0, 'o', 0, 'u', 0, 's', 0,
  'e', 0, ' ', 0, 'I', 0, 'n', 0, 't', 0, 'e', 0, 'r', 0, 'f', 0,
  'a', 0, 'c', 0, 'e', 0
};

/**
  *  The configuration description string.
  */
const uint8_t g_pui8ConfigString[] =
{
  (23 + 1) * 2,
  USB_DTYPE_STRING,
  'H', 0, 'I', 0, 'D', 0, ' ', 0, 'M', 0, 'o', 0, 'u', 0, 's', 0,
  'e', 0, ' ', 0, 'C', 0, 'o', 0, 'n', 0, 'f', 0, 'i', 0, 'g', 0,
  'u', 0, 'r', 0, 'a', 0, 't', 0, 'i', 0, 'o', 0, 'n', 0
};

/**
  *  The descriptor string table.
  */
const uint8_t * const g_ppui8StringDescriptors[] =
{
  g_pui8LangDescriptor,
  g_pui8ManufacturerString,
  g_pui8ProductString,
  g_pui8SeriailNumberString,
```

```
    g_pui8InterfaceString,
    g_pui8ConfigString
};

/**
 *  The number of string descriptors.
 */
#define NUM_STRING_DESCRIPTORS (sizeof(g_ppui8StringDescriptors) / sizeof(uint8_t *))
```

- 修改 HID 报告描述符，HID 不同项的构建已经作为宏定义提供在 usbdhid.h 文件中，例如“UsagePage”、“ Collection”。关于如何构建 HID 报告描述符可以参考“USB Device Class Definition for Human Interface Devices, version 1.11”文件，在 http://www.usb.org/developers/devclass_docs/HID1_11.pdf 中可以下载。以下为一鼠标报告描述符示例：

```
//*****************************************************************************
//
// The report descriptor for the mouse class device.
//
//*****************************************************************************
static const uint8_t g_pui8MouseReportDescriptor[] =
{
    UsagePage(USB_HID_GENERIC_DESKTOP),
    Usage(USB_HID_MOUSE),
    Collection(USB_HID_APPLICATION),
    Usage(USB_HID_POINTER),
    Collection(USB_HID_PHYSICAL),
    // The buttons.
    UsagePage(USB_HID_BUTTONS),
    UsageMinimum(1),
    UsageMaximum(3),
    LogicalMinimum(0),
    LogicalMaximum(1),
    // 3 - 1 bit values for the buttons.
    ReportSize(1),
    ReportCount(3),
    Input(USB_HID_INPUT_DATA | USB_HID_INPUT_VARIABLE |
    USB_HID_INPUT_ABS),
    // 1 - 5 bit unused constant value to fill the 8 bits.
    ReportSize(5),
    ReportCount(1),
    Input(USB_HID_INPUT_CONSTANT | USB_HID_INPUT_ARRAY |
    USB_HID_INPUT_ABS),
    // The X and Y axis.
    UsagePage(USB_HID_GENERIC_DESKTOP),
    Usage(USB_HID_X),
```

```
    Usage(USB_HID_Y),
    Usage(USB_HID_WHEEL),
    LogicalMinimum(-127),
    LogicalMaximum(127),
    // 2 - 8 bit Values for x and y.
    ReportSize(8),
    ReportCount(3),
    Input(USB_HID_INPUT_DATA | USB_HID_INPUT_VARIABLE |
    USB_HID_INPUT_RELATIVE),
    EndCollection,
    EndCollection,
};

//*****************************************************************************
//
// The HID descriptor for the mouse device.
//
//*****************************************************************************
static const tHIDDescriptor g_sMouseHIDDescriptor =
{
    9,                                  // bLength
    USB_HID_DTYPE_HID,                      // bDescriptorType
    0x111,                              // bcdHID (version 1.11 compliant)
    0,                                  // bCountryCode (not localized)
    1,                                  // bNumDescriptors
    {
      {
        USB_HID_DTYPE_REPORT,     // Report descriptor
        sizeof(g_pui8MouseReportDescriptor)
        // Size of report descriptor
      }
    }
};
```

- 定义 **tHIDReportIdle** 类型的结构体用来定义 HID 不同 ID 的 HID 报表报告时间间隔（此项属于可选的配置步骤，用户可以像 SDK 所提供的例程配置 HID 设备），每个 **tHIDReportIdle** 结构体中都有 ui8Duration4mS 和 ui8ReportID 用来记录HID报表的ID，需要注意的是ui8Duration4mS定义的时间以4ms为单位，用在 USB HID “Set_Idle” 和“Get_Idle”的请求中。在这些结构体中定义的时间被用来确定给定的输入报告在没有任何设备状态变化时对主机的影响。例如一个设备支持两个输入报告，报告 1 和报告 2 可以初始化为：

```
tHIDReportIdle g_psReportIdle[2] =
{
{ 125, 1, 0, 0 }, // Report 1 polled every 500mS (4 * 125).
{ 0, 2, 0, 0} // Report 2 is not polled (0mS timeout)
```

```
};
```

- 定义一个 **tUSBDHIDDevice** 类型的结构体用来作为 HID 类初始化的参数，并对初始化结构体所有项进行赋值。该结构体包括设备的 PID、VID、供电参数以及 HID 设备类驱动回调函数入口、回调函数参数等信息。以下是一个简单的 HID 类初始化结构体例子：

```
const tUSBDHIDDevice g_sHIDMouseDevice =
{
  // The Vendor ID you have been assigned by USB-IF.
  USB_VID_YOUR_VENDOR_ID,
  // The product ID you have assigned for this device.
  USB_PID_YOUR_PRODUCT_ID,
  // The power consumption of your device in milliamps.
  POWER_CONSUMPTION_MA,
  // The value to be passed to the host in the USB configuration descriptor's
  // bmAttributes field.
  USB_CONF_ATTR_BUS_PWR,
  // This mouse supports the boot subclass.
  USB_HID_SCLASS_BOOT,
  // This device supports the BIOS mouse report protocol.
  USB_HID_PROTOCOL_MOUSE,
  // The device has a single input report.
  1,
  // A pointer to our array of tHIDReportIdle structures. For this device,
  // the array must have 1 element (matching the value of the previous field).
  g_psMouseReportIdle,
  // A pointer to your receive callback event handler.
  YourUSBReceiveEventCallback,
  // A value that you want passed to the receive callback alongside every
  // event.
  (void *)&g_sYourInstanceData,
  // A pointer to your transmit callback event handler.
  YourUSBTransmitEventCallback,
  // A value that you want passed to the transmit callback alongside every
  // event.
  (void *)&g_sYourInstanceData,
  // This device does not want to use a dedicated interrupt OUT endpoint
  // since there are no output or feature reports required.
  false,
  // A pointer to the HID descriptor for the device.
  &g_sMouseHIDDescriptor,
  // A pointer to the array of HID class descriptor pointers for this device.
  // The number of elements in this array and their order must match the
  // information in the HID descriptor provided above.
  g_pMouseClassDescriptors,
```

```
    // A pointer to your string table.
    g_pStringDescriptors,
    // The number of entries in your string table. This must equal
    // (1 + (5 + (num HID strings)) * (num languages)).
    NUM_STRING_DESCRIPTORS
};
```

- 添加接收事件处理函数，如例子中 YourUSBReceiveEventCallback()函数。就 HID 类设备而言，以下回调事件必须由应用程序处理：
  - **USB_EVENT_RX_AVAILABLE**
  - **USBD_HID_EVENT_IDLE_TIMEOUT**
  - **USBD_HID_EVENT_GET_REPORT_BUFFER**
  - **USBD_HID_EVENT_GET_REPORT**
  - **USBD_HID_EVENT_SET_PROTOCOL (for BIOS protocol devices)**
  - **USBD_HID_EVENT_GET_PROTOCOL (for BIOS protocol devices)**
  - **USBD_HID_EVENT_SET_REPORT**

  虽然其它事件没有强制应用处理，但是通常应用程序会处理 **USB_EVENT_CONNECTED** 以及 **USB_EVENT_DISCONNECTED** 事件。
- 添加发送事件处理函数，如例子中 YourUSBTransmitEventCallback()函数.就 HID 设备而言，发送完成事件 **USB_EVENT_TX_COMPLETE** 会被用来指示 USB 端点一包数据发送完毕。当一报告数据正在传输时，使用 usbdhid_report_write()函数发送数会被忽略或产生错误。
- 在主函数中调用具体的 HID 类设备初始化函数来配置 USB 控制器，例如鼠标例程中就是用 HID 鼠标初始化函数 usbdhid_mouse_init(0, &g_sMouseDevice);
- 在 USB 中断函数中调用 USB 协议栈设备中断处理函数 usb_device_int_handler();

### 2. 6. 3 HID复合设备类驱动使用教程

当 HID 类设备驱动用于创建一个复合设备时，配置过程大致和单个 BULK 设备创建过程相似。就 HID 鼠标举例而言，需要注意的是，此时调用的 HID 初始化函数已经不是 usbdhid_mouse_init() 函数，而是使用 usbdhid_mouse_composite_init() 函数，usbdhid_mouse_composite_init()函数为复合设备创建了一个用于枚举的 HID 类设备实体，此函数将初始化 **tCompositeEntry** 类型的指针来作为复合设备参数表，具体的使用方法见下面的例子：

```
tCompositeEntry g_psCompEntries[NUM_COMP_DEVICES];

//*****************************************************************************
  Allocate the Device Data for the top level composite device class.
```

```
*************************************************************************/
tUSBDCompositeDevice g_sCompDevice =
{
    .ui16VID                = USB_VID_EASTSOFT_30CC,
    .ui16PID                = USB_PID_COMP_HID_SER,
    .ui16MaxPowermA         = 250,
    .ui8PwrAttributes       = USB_CONF_ATTR_BUS_PWR,
    .pfnCallback            = 0,
    .ppui8StringDescriptors = g_ppui8StringDescriptors,
    .ui32NumStringDescriptors = NUM_STRING_DESCRIPTORS,
    .ui32NumDevices         = NUM_COMP_DEVICES,
    .psDevices              = g_psCompEntries
};

    // Initialize the mouse instances that is part of this
    // composite device.
    g_sCompDevice.psDevices[0].pvInstance =
        usbdhid_mouse_composite_init(0, &g_sMouseDevice, &g_psCompEntries[0]);

    // Pass the device information to the USB library and place the device
    // on the bus.
    usbd_composite_init(0, &g_sCompDevice, DESCRIPTOR_DATA_SIZE,
                        g_pui8DescriptorData);
```

### 2. 6. 4 HID报告处理

HID 设备之间通信都是通过 HID 报告来完成的，HID 报告是遵循某种格式的一段数据。

输入报告从设备发往主机，用以报告设备状态的改变，主机可以配置查询间隔时间。在状态改变时，设备通过中断端点将相关输入报告发送给主机。发送报告通过 usbdhid_report_write()函数来完成，在每次调用时，所发的数据不能超过 USB 一个中断端点的数据包长度，一旦主机成功收取 HID 报告，应用程序将接收到 **USB_EVENT_TX_COMPLETE** 事件。主机也可以主动读取 HID 报告，当主机通过端点 0 发送读取报告请求时，**USBD_HID_EVENT_GET_REPORT** 事件会被发向应用程序，应用程序需要将最新的 HID 报表数据地址以及长度再传入协议栈，然后此 HID 报表数据将会通过端点 0 传输到主机。主机成功读取后，协议栈会向应用程序发送 **USBD_HID_EVENT_REPORT_SENT** 事件。

另一种情况也会导致发送输入报告，在设备初始化阶段已经设置好主机查询的事件间隔，当这个时间间隔到期后，不管设备状态是否发生变化，HID 报告都需要返回到主机。

HID 设备类驱动会跟踪每个报告的主机查询时间间隔，当事件到期后，表明该报告必须被发送到主机，此时协议栈会向应用程序发送 **USBD_HID_EVENT_IDLE_TIMEOUT** 事件，应用程序此时必须返回适当字节数的 HID 报告。返回的数据通过中断端点发送到主机，当主机成功接收后，协议栈会向应用程序发送 **USB_EVENT_TX_COMPLETE** 事件。

输出和特性报告是由主机发往设备，以指示从机设置各种参数和选项。设备可以选择所有的主机对设备报告通信是否通过端点 0 进行，或者使用专用的中断端点。一般来说，

对于低带宽的数据传输通过端点 0 通信就可以了，如果需要传输比较大的数据，可以将 **tUSBDHIDDevice** 结构体中的 bUseOutEndpoint 位设为 true。

如果使用一个专用端点用于输出和特性报告，那么应用程序接收回调函数就会被使用，每当数据包接收到时，**USB_EVENT_RX_AVAILABLE** 事件会被协议栈发向应用程序，应用程序可以使用 usbdhid_packet_read()函数读取接收到的报告数据。

在最典型的案例中，端点 0 用于传输输出和特性报告，应用程序可以处理一下相关的事件：

- **USBD_HID_EVENT_GET_REPORT_BUFFER** 指示从主机接收到 Set_Report 请求，应用程序必须返回缓冲区的指针，它至少和存储的报告所需的大小一样大。
- **USBD_HID_EVENT_SET_REPORT** 指示报告数据已经从端点 0 读取到前面 **USBD_HID_EVENT_GET_REPORT_BUFFER** 回调所提供的缓冲区，在发送此事件后，USB 协议栈不能访问报告缓冲区，应用程序可以在此之后处理内存。

### 2. 6. 5 HID类设备驱动接口定义

**数据结构：**

- tHIDClassDescriptorInfo
- tHIDDescriptor
- tHIDReportIdle
- tUSBDHIDDevice

宏：

- COMPOSITE_DHID_SIZE
- USBD_HID_EVENT_GET_PROTOCOL
- USBD_HID_EVENT_GET_REPORT
- USBD_HID_EVENT_GET_REPORT_BUFFER
- USBD_HID_EVENT_IDLE_TIMEOUT
- USBD_HID_EVENT_REPORT_SENT
- USBD_HID_EVENT_SET_PROTOCOL
- USBD_HID_EVENT_SET_REPORT

**函数接口：**

- void *usbdhid_composite_init(uint32_t ui32Index, tUSBDHIDDevice *psHIDDevice, tCompositeEntry * psCompEntry);
- void *usbdhid_init(uint32_t ui32Index, tUSBDHIDDevice *psHIDDevice);
- void usbdhid_term(void *pvHIDInstance);

- void *usbdhid_set_rx_cb_data(void *pvHIDInstance, void *pvCBData);
- void *usbdhid_set_tx_cb_data(void *pvHIDInstance, void *pvCBData);
- uint32_t usbdhid_report_write(void *pvHIDInstance, uint8_t *pi8Data, uint32_t ui32Length, bool bLast);
- uint32_t usbdhid_packet_read(void *pvHIDInstance, uint8_t *pi8Data, uint32_t ui32Length, bool bLast);
- uint32_t usbdhid_tx_packet_available(void *pvHIDInstance);
- uint32_t usbdhid_rx_packet_available(void *pvHIDInstance);
- void usbdhid_power_status_set(void *pvHIDInstance, uint8_t ui8Power);
- bool usbdhid_remote_wakeup_request(void *pvHIDInstance);

#### 2. 6. 5. 1 tHIDClassDescriptorInfo

定义：

```
typedef struct
{
  uint8_t bDescriptorType;
  uint16_t wDescriptorLength;
} tHIDClassDescriptorInfo;
```

参数定义：

**bDescriptorType**：HID 设备类描述符类型，值可以是 **USB_HID_DTYPE_REPORT** 或者 **USB_HID_DTYPE_PHYSICAL**。

**wDescriptorLength**：HID 描述符总长度。

描述：

此结构体用于声明 HID 特定的描述符。

#### 2. 6. 5. 2 tHIDDescriptor

定义：

```
typedef struct
{
  uint8_t bLength;
  uint8_t bDescriptorType;
  uint16_t bcdHID;
  uint8_t bCountryCode;
  uint8_t bNumDescriptors;
  tHIDClassDescriptorInfo sClassDescriptor[1];
} tHIDDescriptor;
```

参数定义：

**bLength**：描述符长度。

**bDescriptorType**：描述符种类，就 HID 描述符而言，此项值为 **USB_HID_DTYPE_HID**。

**bcdHID**：支持的 HID 类规范版本号。

**bCountryCode**：国籍码或者语言支持码。

**bNumDescriptors**：类描述符数量。

**sClassDescriptor**：类描述符列表。

#### 2. 6. 5. 3 tHIDReportIdle

**定义：**

```
typedef struct
{
  uint8_t ui8Duration4mS;
  uint8_t ui8ReportID;
  uint16_t ui16TimeTillNextmS;
  uint32_t ui32TimeSinceReportmS;
} tHIDReportIdle;
```

**参数定义：**

**ui8Duration4mS**：定义 HID 报告的空闲时间，0 表示无限制除非状态更改，否则不要发送报告。

**ui8ReportID**：HID 报表的 ID。

**ui16TimeTillNextmS**：发送报告的时间间隔。

**ui32TimeSinceReportmS**：距离上次报告的时间间隔，一旦 Set_Idle 请求产生，HID 设备类驱动就需要追踪这个值。

#### 2. 6. 5. 4 tUSBDHIDDevice

**定义：**

```
typedef struct
{
  uint16_t ui16VID;
  uint16_t ui16PID;
  uint16_t ui16MaxPowermA;
  uint8_t ui8PwrAttributes;
  uint8_t ui8Subclass;
  uint8_t ui8Protocol;
  uint8_t ui8NumInputReports;
  tHIDReportIdle *psReportIdle;
  tUSBCallback pfnRxCallback;
  void *pvRxCBData;
```

```
    tUSBCallback pfnTxCallback;
    void *pvTxCBData;
    bool bUseOutEndpoint;
    const tHIDDescriptor *psHIDDescriptor;
    const uint8_t *const *ppui8ClassDescriptors;
    const uint8_t *const *ppui8StringDescriptors;
    uint32_t ui32NumStringDescriptors;
    const tConfigHeader *const *ppsConfigDescriptor;
    tHIDInstance sPrivateData;
} tUSBDHIDDevice;
```

**参数定义：**

**ui16VID**：厂商 ID，由 USB-IF 统一分配。

**ui16PID**：产品 ID，由制造商分配。

**ui16MaxPowermA**：最大供电电流（mA）。

**ui8PwrAttributes**：设备供电特性，值可以是 **USB_CONF_ATTR_SELF_PWR**、**USB_CONF_ATTR_BUS_PWR**，另外可选的配置为 **USB_CONF_ATTR_RWAKE**（如果设备支持此功能）。

**ui8Subclass**：HID 子类设备。

**ui8Protocol**：HID 接口协议。

**ui8NumInputReports**：输入报告的数量。这个值必须和此设备的 HID 描述符中的 HID 描述符数量相等。

**psReportIdle**：指向 **tHIDReportIdle** 类型的结构体。

**pfnRxCallback**：指向接收回调函数的指针。

**pvRxCBData**：接收回调函数参数指针。

**pfnTxCallback**：指向发送回调函数的指针。

**pvTxCBData**：发送回调函数指针。

**bUseOutEndpoint**：是否使用额外的 OUT 端点，如果使用则接下来的数据接收将通过 **USB_EVENT_RX_AVAILABLE** 事件发往应用程序，如果不使用协议栈将使用端点 0 并向应用程序发送 **USBD_HID_EVENT_REPORT_SENT** 事件。

**psHIDDescriptor**：指向 **tHIDDescriptor** 结构的 HID 描述符结构体指针。

**ppui8ClassDescriptors**：指向 HID 子类的描述符指针。

**ppui8StringDescriptors**：指向字符串描述符表的指针。

**ui32NumStringDescriptors**：设备字符串描述符表的大小。

**ppsConfigDescriptor**：HID 设备配置描述符。

**sPrivateData**：HID 类驱动私有参数，此内存空间必须可以访问但是不可以被 HID 类

驱动以外的程序代码修改。

#### 2. 6. 5. 5 COMPOSITE_DHID_SIZE

**描述**：为单个 HID 类设备实体创建一个配置描述符应该分配的内存大小，并不包含复合设备类自动忽略的配置描述符。

#### 2. 6. 5. 6 USBD_HID_EVENT_GET_PROTOCOL

**描述**：此事件在响应主机 Get_Protocol 请求时发向应用程序，回调函数应该提供当前协议的种类，值可以是 **USB_HID_PROTOCOL_BOOT** 和 **USB_HID_PROTOCOL_REPORT**。

#### 2. 6. 5. 7 USBD_HID_EVENT_GET_REPORT

**描述**：这个事件表明主机正在请求一个通过端点 0 返回的的报告，回调函数中 ui32MsgValue 参数包含了报告的种类（高字节）和报告的 ID（低字节），格式和 USB 请求中的 wValue 项一样。回调函数中的 pvMsgData 参数包含和报告种类相关的数据。回调函数必须包含*pvMsgData 所包含的数据的长度。应用程序响应这个事件后，响应的数据必须保持不变直到 **USBD_HID_EVENT_REPORT_SENT** 事件到来。

#### 2. 6. 5. 8 USBD_HID_EVENT_GET_REPORT_BUFFER

**描述**：这个事件表明主机已经发送了一个 Set_Report 的请求，并请求该设备提供一个缓冲报告的缓冲区。ui32MsgValue 参数包含高字节的接收报告类型以及低字节的报告 ID（USB 请求的 wValue 项传递）。pvMsgData 参数包含请求的缓冲区长度。回调函数必须返回一个合适的缓冲区地址。

#### 2. 6. 5. 9 USBD_HID_EVENT_IDLE_TIMEOUT

**描述**：此事件指示应用程序已经发生了一个报告空闲超时，并请求一个指向报告的指针。ui32MsgData 的值包含请求报告的 ID，pvMsgData 包含发送报告数据的指针，回调函数应该返回报告的字节数。

#### 2. 6. 5. 10 USBD_HID_EVENT_REPORT_SENT

**描述**：此事件表明先前主机请求的 **USBD_HID_EVENT_GET_REPORT** 事件数据已经成功被主机接收，应用程序可以重新使用这块占有的内存。这个事件看似是一个发送类型的事件，但是这是为了确保与请求相关的以及通过端点 0 传输报告的相关事件都可以在一个函数中处理。

#### 2. 6. 5. 11 USBD_HID_EVENT_SET_PROTOCOL

**描述**：此事件用于回复 Set_Protocol 请求，ui32MsgData 值可以是 **USB_HID_PROTOCOL_BOOT** 和 **USB_HID_PROTOCOL_REPORT**。

#### 2. 6. 5. 12 USBD_HID_EVENT_SET_REPORT

**描述**：此事件表明主机通过端点 0 发送了一个报告，ui32MsgValue 代表报告的大小，pvMsgData 指向报告的数据。报告的内存即先前 **USBD_HID_EVENT_GET_REPORT_BUFFER** 事件所返回的内存空间。HID 驱动程序不能在此回调后再对此内存操作，但是应用程序可以。

#### 2. 6. 5. 13 usbdhid_composite_init()

当需要创建复合设备时用于 HID 设备的初始化。

**参数定义：**

**ui32Index**：USB 控制器索引。

**psDevice**：指向 HID 初始化结构体 **tUSBDHIDDevice** 的指针。

**psCompEntry**：指向复合设备初始化入口结构体 **tCompositeEntry** 的指针，当需要创建复合设备时作为复合设备的初始化入口参数。

**简介：**

本函数和 HID 类初始化函数类似，当需要创建复合函数时，这个函数会初始化指向 **tCompositeEntry** 结构体的入口参数。

**返回值：**

失败返回 0，成功则返回指向 **tUSBDHIDDevice** 的指针，用于接下来 HID 类驱动 API 的参数。

#### 2. 6. 5. 14 usbdhid_init()

初始化 HID 类设备。

**参数定义：**

**ui32Index**：USB 控制器索引。

**psHIDDevice**：指向 HID 初始化结构体 **tUSBDHIDDevice** 的指针。

**简介：**

应用程序希望创建一个 HID 类设备与 PC 端传输数据，可以调用这个函数来初始化 USB 控制器，此函数执行所有的 HID 类设备的初始化工作。

此函数调用成功后会返回传递给它的 **tUSBDHIDDevic** 指针，这个指针必须作为以后所有的 HID 类驱动 API 的参数。

HID 设备类驱动为应用程序提供了以报告为基础的传输接口。OUTPUT 报告可以通过控制端点接收，也可以通过专用的中断端点接收。如果使用专用的端点，报告的数据将直接传递到应用程序层。如果应用程序报告超出了 **USBDHID_MAX_PACKET** 字节长度的数据就需要在此通道上使用 USB buffer 模块。

发送操作：

调用 usbdhid_report_write()函数完成报告数据的发送，一旦整个 INPUT 报告被 USB 主机接收，**USB_EVENT_TX_COMPLETE** 事件将会发向应用程序。

接收操作（使用专用的输出端点）：

当 USB 协议栈接收到一包数据后，将向应用程序发送 **USB_EVENT_RX_AVAILABLE** 事件。应用程序可以调用 usbdhid_packet_read()函

数来读取数据，报告数据数量可以调用 usbdhid_rx_packet_available()函数来确定。

接收操作（不使用输出端点）：

如果没有使用专用的输出端点，OUTPUT 以及 FEATURE 报告都将通过端点 0 来传输。当这些报告接收到时，USB 协议栈将会向应用程序发送 **USBD_HID_EVENT_GET_REPORT_BUFFER** 事件来索取一个大的缓冲区来缓冲报告，然后协议栈将报告数据复制到这块缓冲区，最后发送 **USBD_HID_EVENT_SET_REPORT** 事件通知应用程序缓冲区数据有效。

**返回值：**

失败返回 NULL，成功则返回指向 **psHIDDevice** 的指针。

#### 2. 6. 5. 15 usbdhid_packet_read()

从 HID 输出端点读取一包数据。

**参数定义：**

**pvHIDInstance**：指向由 usbdhid_init()函数返回的 HID 设备类初始化结构体 **psHIDDevice** 的指针。

**pi8Data**：指向数据接收的存储空间指针。

**ui32Length**：pi8Data 所指向的存储空间大小

**bLast**：指示从机是否需要进一步调用从包中读取数据，本位供 USB 协议栈 buffer 模块调度使用，当应用程序直接使用此函数读取数据时，将本位置 1 即可。

**简介：**

此函数将读取最多 ui32Length 指示的数据量的数据。当从机可以确定读取数据量的时候，bLast 参数可以忽略。

**返回值：**

返回读取到的数据数量。

#### 2. 6. 5. 16 usbdhid_report_write()

向主机发送一包数据。

**参数定义：**

**pvHIDInstance**：指向由 usbdhid_init()函数初始化返回的 HID 设备类初始化结构体 **psHIDDevice** 的指针。

**pi8Data**：指向数据发送的存储空间指针。

**ui32Length**：pi8Data 所指向的存储空间大小

**bLast**：指示从机是否需要进一步调用以发送数据，如果为 true，USB 协议栈将会进一步调用此函数，如果为 false，在调度完一个短包传输后则不会再进行任何调用。本位供 USB 协议栈 buffer 模块调度使用，当应用程序直接使用此函数发送数据时，将

本位置 1 即可。

**简介：**

此函数将提供数据调度功能，将一个 USB 数据包传输到 USB 主机端。如果当前没有传输正在进行，数据将立即复制到相关的 USB 端点 FIFO。如果 bLast 参数为 true，那么新写的数据包就会被安排传输。当 USB 主机确认接收到数据包后，协议栈会向应用程序发送 **USB_EVENT_TX_COMPLETE** 事件，指示可以发送接下来的数据包。

调用此函数一次最大发送的数据长度为 64 bytes。

**返回值：**

成功返回发送的数据数量，失败返回 0，表明目前不能发送数据。

#### 2. 6. 5. 17 usbdhid_term()

停用 HID 设备类驱动。

**参数定义：**

**pvHIDInstance**：指向由 usbdhid_init()函数初始化返回的 HID 设备类初始化结构体 **psHIDDevice** 的指针。

**简介：**

此函数将终止所有 HID 类驱动操作，并从总线删除设备，如果 HID 设备是复合设备的一部分，那么不应该调用这个函数，而应该为整个复合设备调用 usbd_composite_term()函数。在调用此函数之后，**psHIDDevice** 指针实例不应该在任何其它 API 中使用。

**返回值：**

无。

#### 2. 6. 5. 18 usbdhid_tx_packet_available()

返回可以发送的数据数量。

**参数定义：**

**pvHIDInstance**：指向由 usbdhid_init()函数初始化返回的 HID 设备类初始化结构体 **psHIDDevice** 的指针。

**简介：**

此函数返回 usbdhid_report_write()函数可以发送的最大数据量，返回 64 的话说明目前没有数据正在传输，返回 0 说明传输正在进行。

**返回值：**

返回可以传输的数据数量。

#### 2. 6. 5. 19 usbdhid_rx_packet_available()

返回可以读取的数据数量。

**参数定义：**

**pvHIDInstance**：指向由 usbdhid_init()函数初始化返回的 HID 设备类初始化结构体 **psHIDDevice** 的指针。

**简介：**

此函数返回 usbdhid_packet_read()函数可以读取的最大数据量。

**返回值：**

返回可以读取的数据数量。

#### 2. 6. 5. 20 usbdhid_set_rx_cb_data()

修改 HID 类驱动的接收回调函数。

**参数定义：**

**pvHIDInstance**：指向由 usbdhid_init()函数初始化返回的 HID 设备类初始化结构体 **psHIDDevice** 的指针。

**pvCBData**：指向接收回调函数的指针。

**简介：**

应用程序使用此函数来修改由 usbdhid_init()函数配置好的回调函数指针。如果应用程序需要在运行时修改接收回调函数的指针值，需要保证 pvHIDInstance 存储在 SRAM 中。

**返回值：**

返回先前使用的回调函数指针值。

#### 2. 6. 5. 21 usbdhid_set_tx_cb_data()

修改 HID 类驱动的发送回调函数。

**参数定义：**

**pvHIDInstance**：指向由 usbdhid_init()函数初始化返回的 HID 设备类初始化结构体 **psHIDDevice** 的指针。

**pvCBData**：指向发送回调函数的指针。

**简介：**

应用程序使用此函数来修改由 usbdhid_init()函数配置好的回调函数指针。如果应用程序需要在运行时修改接收回调函数的指针值，需要保证 pvHIDInstance 存储在 SRAM 中。

**返回值：**

返回先前使用的回调函数指针值。

#### 2. 6. 5. 22 usbdhid_power_status_set()

修改设备的供电特性。

**参数定义：**

**pvHIDInstance**：指向由 usbdhid_init()函数初始化返回的 HID 设备类初始化结构体 **psHIDDevice** 的指针。

**ui8Power**：供电特性。

**简介：**

USB 协议栈支持设备供电特性的修改，修改的值可以是 **USB_STATUS_SELF_PWR** 或者 **USB_STATUS_BUS_PWR**。在 USB 主机设备再次请求后，USB 协议栈会将修改后的供电特性报告给主机。

**返回值：**

无。

#### 2. 6. 5. 23 usbdhid_remote_wakeup_request()

当 USB 控制器处于挂起状态时，发送“wakeup”信号到主机。

**参数定义：**

**pvHIDInstance**：指向由 usbdhid_init()函数初始化返回的 HID 设备类初始化结构体 **psHIDDevice** 的指针。

**简介：**

当处于挂起状态时，如果从机支持远程唤醒（通过初始化结构体设置）功能的话，可以调用此函数来唤醒主机。

**返回值：**

返回 true 表示远程唤醒功能没有被禁止且被发送，返回 false 表示远程唤醒被禁止。

## 2. 7 HID鼠标设备驱动

USB HID 类设备使用非常广泛。HID 鼠标驱动位于 HID 设备类驱动层次之上，对于想要呈现鼠标功能的应用程序可以直接使用本层驱动，而不需要在 HID 设备类驱动上再进行开发。HID 鼠标驱动帮助应用程序创建一个鼠标设备，并提供了一个用于报告鼠标状态的接口，允许鼠标设备报告其与鼠标相关的一些按键、位置状态。

本驱动使用的是 BIOS 鼠标子类协议，因此在绝大多数的操作系统中不需要额外的驱动软件就可以识别出来。鼠标提供两个位置轴（x、y）、三个按键（左右中）以及一个滑轮滑动的数据报告接口。

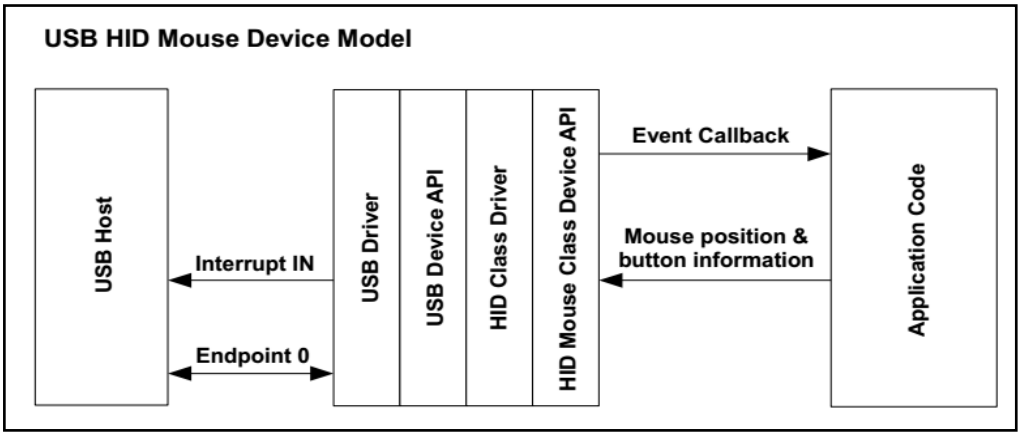


图 2-6 USB HID 鼠标驱动模型

### 2.7.1 HID鼠标驱动事件

HID 鼠标驱动提供了以下回调事件：

- **USB_EVENT_CONNECTED**
- **USB_EVENT_DISCONNECTED**
- **USB_EVENT_TX_COMPLETE**
- **USB_EVENT_ERROR**
- **USB_EVENT_SUSPEND**
- **USB_EVENT_RESUME**

注：
1. **USB_EVENT_DISCONNECTED** 事件仅在某些器件内支持，具体请查阅相关器件手册，确定设备模式下是否有断开连接检测功能。

### 2.7.2 HID鼠标驱动使用教程

HID 鼠标驱动使用可以遵循以下几步：

- 将以下头文件加入到需要用到此驱动的源文件中：

```
#include "usblib/drivers/usb_lowlayer_api.h"
#include "usblib/usblib.h"
#include "usblib/device/usbdhidmouse.h"
```

- 定义 6 个用来描述设备特征的 USB 字符串描述符，修改字符串内容来描述用户自己的设备，注意当字符串改变后，每个描述符数组的长度（每个数组的第一个元素即为数组长度）也需要相应地进行修改。这里以鼠标的例程的字符串描述符为例：

```
/**
  * The languages supported by this device.
  */
const uint8_t g_pui8LangDescriptor[] =
```

```
{
    4,
    USB_DTYPE_STRING,
    USBShort(USB_LANG_EN_US)
};

/**
 * The manufacturer string.
 */
const uint8_t g_pui8ManufacturerString[] =
{
    (8 + 1) * 2,
    USB_DTYPE_STRING,
    'E', 0, 'a', 0, 's', 0, 't', 0, 'S', 0, 'o', 0, 'f', 0, 't', 0,
};

/**
 *  The product string.
 */
const uint8_t g_pui8ProductString[] =
{
    (13 + 1) * 2,
    USB_DTYPE_STRING,
    'M', 0, 'o', 0, 'u', 0, 's', 0, 'e', 0, ' ', 0, 'E', 0, 'x', 0, 'a', 0,
    'm', 0, 'p', 0, 'l', 0, 'e', 0
};

/**
 *  The serial number string.
 */
const uint8_t g_pui8SeriailNumberString[] =
{
    (8 + 1) * 2,
    USB_DTYPE_STRING,
    '1', 0, '2', 0, '3', 0, '4', 0, '5', 0, '6', 0, '7', 0, '8', 0
};

/**
 *  The interface description string.
 */
const uint8_t g_pui8InterfaceString[] =
{
    (19 + 1) * 2,
    USB_DTYPE_STRING,
```

```
    'H', 0, 'I', 0, 'D', 0, ' ', 0, 'M', 0, 'o', 0, 'u', 0, 's', 0,
    'e', 0, ' ', 0, 'I', 0, 'n', 0, 't', 0, 'e', 0, 'r', 0, 'f', 0,
    'a', 0, 'c', 0, 'e', 0
};

/**
 *  The configuration description string.
 */
const uint8_t g_pui8ConfigString[] =
{
    (23 + 1) * 2,
    USB_DTYPE_STRING,
    'H', 0, 'I', 0, 'D', 0, ' ', 0, 'M', 0, 'o', 0, 'u', 0, 's', 0,
    'e', 0, ' ', 0, 'C', 0, 'o', 0, 'n', 0, 'f', 0, 'i', 0, 'g', 0,
    'u', 0, 'r', 0, 'a', 0, 't', 0, 'i', 0, 'o', 0, 'n', 0
};

/**
 *  The descriptor string table.
 */
const uint8_t * const g_ppui8StringDescriptors[] =
{
    g_pui8LangDescriptor,
    g_pui8ManufacturerString,
    g_pui8ProductString,
    g_pui8SeriailNumberString,
    g_pui8InterfaceString,
    g_pui8ConfigString
};

/**
 *  The number of string descriptors.
 */
#define NUM_STRING_DESCRIPTORS (sizeof(g_ppui8StringDescriptors) / sizeof(uint8_t *))
```

■ 定义鼠标初始化结构体。

```
const tUSBDHIDMouseDevice g_sMouseDevice =
{
    // The Vendor ID you have been assigned by USB-IF.
    USB_VID_YOUR_VENDOR_ID,
    // The product ID you have assigned for this device.
    USB_PID_YOUR_PRODUCT_ID,
    // The power consumption of your device in milliamps.
    POWER_CONSUMPTION_MA,
```

```
    // The value to be passed to the host in the USB configuration descriptor's
    // bmAttributes field.
    USB_CONF_ATTR_SELF_PWR,
    // A pointer to your mouse callback event handler.
    YourMouseHandler,
    // A value that you want passed to the callback alongside every event.
    (void *)&g_sYourInstanceData,
    // A pointer to your string table.
    g_pStringDescriptors,
    // The number of entries in your string table. This must equal
    // (1 + (5 * (num languages))).
    NUM_STRING_DESCRIPTORS
};
```

- 添加鼠标事件处理函数，YourMouseHandler()函数。**USB_EVENT_TX_COMPLETE** 事件可以用来确认鼠标报告数据是否成功传输到主机端。如果在上一报告数据还没有成功被主机接收，下一包数据发送的话会产生 **MOUSE_ERR_TX_ERROR** 事件。
- 在主函数中调用具体的 HID 鼠标设备初始化函数来配置 USB 控制器， HID 鼠标初始化函数 usbdhid_mouse_init(0, & g_sMouseDevice);
- 在 USB 中断函数中调用 USB 协议栈设备中断处理函数 usb_device_int_handler()；

### 2.7.3 HID鼠标复合设备类驱动使用教程

当 HID 鼠标驱动用于创建一个复合设备时，配置过程大致和单个鼠标设备创建过程相似。就 HID 鼠标举例而言，需要注意的是，此时调用的 HID 初始化函数已经不是 usbdhid_mouse_init() 函数，而是使用 usbdhid_mouse_composite_init() 函数，usbdhid_mouse_composite_init()函数为复合设备创建了一个用于枚举的 HID 类设备实体，此函数将初始化 **tCompositeEntry** 类型的指针来作为复合设备参数表，具体的使用方法见下面的例子：

```
tCompositeEntry g_psCompEntries[NUM_COMP_DEVICES];

//*****************************************************************************
// Allocate the Device Data for the top level composite device class.
//*****************************************************************************
tUSBDCompositeDevice g_sCompDevice =
{
    .ui16VID                = USB_VID_EASTSOFT_30CC,
    .ui16PID                = USB_PID_COMP_HID_SER,
    .ui16MaxPowermA         = 250,
    .ui8PwrAttributes       = USB_CONF_ATTR_BUS_PWR,
    .pfnCallback            = 0,
    .ppui8StringDescriptors = g_ppui8StringDescriptors,
    .ui32NumStringDescriptors = NUM_STRING_DESCRIPTORS,
```

```
    .ui32NumDevices         = NUM_COMP_DEVICES,
    .psDevices              = g_psCompEntries
};

    // Initialize the mouse instances that is part of this
    // composite device.
    g_sCompDevice.psDevices[0].pvInstance =
        usbdhid_mouse_composite_init(0, &g_sMouseDevice, &g_psCompEntries[0]);

    // Pass the device information to the USB library and place the device
    // on the bus.
    usbd_composite_init(0, &g_sCompDevice, DESCRIPTOR_DATA_SIZE,
                        g_pui8DescriptorData);
```

### 2. 7. 4 HID鼠标设备驱动接口定义

**数据结构：**

- tUSBDHIDMouseDevice

宏：

- MOUSE_ERR_NOT_CONFIGURED
- MOUSE_ERR_TX_ERROR
- MOUSE_REPORT_BUTTON_1
- MOUSE_REPORT_BUTTON_2
- MOUSE_REPORT_BUTTON_3
- MOUSE_SUCCESS

**函数接口：**

- void * usbdhid_mouse_composite_init(uint32_t ui32Index, tUSBDHIDMouseDevice *psMouseDevice, tCompositeEntry * psCompEntry);
- void * usbdhid_mouse_init(uint32_t ui32Index, tUSBDHIDMouseDevice *psMouseDevice);
- void usbdhid_mouse_term(void * pvMouseDevice);
- void *usbdhid_mouse_set_cb_data(void *pvMouseDevice, void *pvCBData);
- void usbdhid_mouse_power_status_set(void * pvMouseDevice, uint8_t ui8Power);
- bool usbdhid_mouse_remote_wakeup_request(void *pvHIDInstance);
- uint32_t usbdhid_mouse_state_change(void *pvMouseDevice, int8_t i8DeltaX, int8_t i8DeltaY, uint8_t ui8Buttons, int8_t i8Wheel);

#### 2. 7. 4. 1 tUSBDHIDMouseDevice

定义：

```
typedef struct
{
    const uint16_t ui16VID;
    const uint16_t ui16PID;
    const uint16_t ui16MaxPowermA;
    const uint8_t ui8PwrAttributes;
    const tUSBCallback pfnCallback;
    void *pvCBData;
    const uint8_t *const *ppui8StringDescriptors;
    const uint32_t ui32NumStringDescriptors;
    tHIDMouseInstance sPrivateData;
} tUSBDHIDMouseDevice;
```

参数定义：

**ui16VID**：厂商 ID，由 USB-IF 统一分配。

**ui16PID**：产品 ID，由制造商分配。

**ui16MaxPowermA**：最大供电电流（mA）。

**ui8PwrAttributes**：设备供电特性，值可以是 **USB_CONF_ATTR_SELF_PWR**、**USB_CONF_ATTR_BUS_PWR**，另外可选的配置为 **USB_CONF_ATTR_RWAKE**（如果设备支持此功能）。

**pfnCallback**：鼠标事件处理回调函数。

**pvCBData**：回调函数参数指针。

**ppui8StringDescriptors**：指向字符串描述符表的指针。

**ui32NumStringDescriptors**：设备字符串描述符表的大小。

**sPrivateData**：HID 类驱动私有参数，此内存空间必须可以访问但是不可以被 HID 类驱动以外的程序代码修改。

#### 2. 7. 4. 2 MOUSE_ERR_NOT_CONFIGURED

**描述**：在设备连接和配置之前，如果调用 usbdhid_mouse_state_change()函数将返回此值。

#### 2. 7. 4. 3 MOUSE_ERR_TX_ERROR

**描述**：调用 usbdhid_mouse_state_change()函数返回此值时表示报告有错误产生。

#### 2. 7. 4. 4 MOUSE_REPORT_BUTTON_1

**描述**：在 usbdhid_mouse_state_change()函数 ui8Buttons 参数中设置此位表示按键 1 被按下。

#### 2. 7. 4. 5 MOUSE_REPORT_BUTTON_2

**描述**：在 usbdhid_mouse_state_change()函数 ui8Buttons 参数中设置此位表示按键 2 被按下。

#### 2. 7. 4. 6 MOUSE_REPORT_BUTTON_3

**描述**：在 usbdhid_mouse_state_change()函数 ui8Buttons 参数中设置此位表示按键 3 被按下。

#### 2. 7. 4. 7 MOUSE_SUCCESS

**描述**：usbdhid_mouse_state_change()函数返回此值表明调用成功。

#### 2. 7. 4. 8 usbdhid_mouse_composite_init()

当需要创建复合设备时用于 HID 鼠标的初始化。

**参数定义**：

**ui32Index**：USB 控制器索引。

**psMouseDevice**：指向 HID 鼠标初始化结构体 **tUSBDHIDMouseDevice** 的指针。

**psCompEntry**：指向复合设备初始化入口结构体 **tCompositeEntry** 的指针，当需要创建复合设备时作为复合设备的初始化入口参数。

**简介**：

本函数和 HID 鼠标初始化函数类似，当需要创建复合函数时，这个函数会初始化指向 **tCompositeEntry** 结构体的入口参数。

**返回值**：

失败返回 0，成功则返回指向 **tUSBDHIDMouseDevice** 的指针，用于接下来 HID 鼠标驱动 API 的参数。

#### 2. 7. 4. 9 usbdhid_mouse_init()

初始化 HID 鼠标设备。

**参数定义**：

**ui32Index**：USB 控制器索引。

**psMouseDevice**：指向 HID 鼠标初始化结构体 **tUSBDHIDMouseDevice** 的指针。

**简介**：

应用程序希望创建一个 HID 鼠标设备与 PC 端传输数据，可以调用这个函数来初始化 USB 控制器，此函数执行所有的 HID 鼠标设备的初始化工作。

此函数调用成功后会返回传递给它的 **tUSBDHIDMouseDevice** 指针，这个指针必须作为以后所有的 HID 鼠标驱动 API 的参数。

**返回值**：

失败返回 NULL，成功则返回指向 **tUSBDHIDMouseDevice** 的指针。

#### 2. 7. 4. 10 usbdhid_mouse_power_status_set()

修改设备的供电特性。

**参数定义：**

**pvMouseDevice**：指向由 usbdhid_mouse_init()函数初始化返回的 HID 鼠标初始化结构体 **tUSBDHIDMouseDevice** 的指针。

**ui8Power**：供电特性。

**简介：**

USB 协议栈支持设备供电特性的修改，修改的值可以是 **USB_STATUS_SELF_PWR** 或者 **USB_STATUS_BUS_PWR**。在 USB 主机设备再次请求后，USB 协议栈会将修改后的供电特性报告给主机。

**返回值：**

无。

#### 2. 7. 4. 11 usbdhid_mouse_state_change()

向主机发送鼠标报表数据。

**参数定义：**

**pvMouseDevice**：指向由 usbdhid_mouse_init()函数初始化返回的 HID 鼠标初始化结构体 **tUSBDHIDMouseDevice** 的指针。

**i8DeltaX**：x 轴移动数据，取值范围为[-127,127]。

**i8DeltaY**：y 轴移动数据，取值范围为[-127,127]。

**ui8Buttons**：按键数据，按键掩码可以是 **MOUSE_REPORT_BUTTON_1**、**MOUSE_REPORT_BUTTON_2**、**MOUSE_REPORT_BUTTON_3**。

**i8Wheel**：滑轮值，取值范围为[-127,127]。

**简介：**

此函数用来报告鼠标报表数据到主机。

**返回值：**

返回 **MOUSE_SUCCESS** 表明将成功发送，返回 **MOUSE_ERR_TX_ERROR** 表明发送时出错（最典型的原因是当前包在发送时上一包还没有发送出去）。返回 **MOUSE_ERR_NOT_CONFIGURED** 表明在调用此函数时还没有成功连接主机或被主机配置。

#### 2. 7. 4. 12 usbdhid_mouse_term()

停用 HID 鼠标驱动。

**参数定义：**

**pvMouseDevice**：指向由 usbdhid_mouse_init()函数初始化返回的 HID 鼠标初始化结构体 **tUSBDHIDMouseDevice** 的指针。

**简介：**

此函数将终止所有 HID 鼠标驱动操作，并从总线删除设备，如果 HID 鼠标设备是复合设备的一部分，那么不应该调用这个函数，而应该为整个复合设备调用 usbd_composite_term()函数。在调用此函数之后，**tUSBDHIDMouseDevice** 指针实例不应该在任何其它 API 中使用。

**返回值：**

无。

#### 2. 7. 4. 13 usbdhid_mouse_set_cb_data()

修改 HID 鼠标驱动的回调函数。

**参数定义：**

**pvMouseDevice**：指向由 usbdhid_mouse_init()函数初始化返回的 HID 鼠标初始化结构体 **tUSBDHIDMouseDevice** 的指针。

**pvCBData**：指向鼠标回调函数的指针。

**简介：**

应用程序使用此函数来修改由 usbdhid_mouse_init()函数配置好的回调函数指针。如果应用程序需要在运行时修改接收回调函数的指针值，需要保证 **tUSBDHIDMouseDevice** 存储在 SRAM 中。

**返回值：**

返回先前使用的回调函数指针值。

#### 2. 7. 4. 14 usbdhid_mouse_remote_wakeup_request()

当 USB 控制器处于挂起状态时，发送“wakeup”信号到主机。

**参数定义：**

**pvMouseDevice**：指向由 usbdhid_mouse_init()函数初始化返回的 HID 鼠标初始化结构体 **tUSBDHIDMouseDevice** 的指针。

**简介：**

当处于挂起状态时，如果从机支持远程唤醒（通过初始化结构体设置）功能的话，可以调用此函数来唤醒主机。

**返回值：**

返回 true 表示远程唤醒功能没有被禁止且被发送，返回 false 表示远程唤醒被禁止。

## 2.8 HID键盘设备驱动

和 HID 鼠标驱动类似，HID 键盘驱动也是位于 HID 类驱动层的上一层。HID 键盘驱动对于上层的应用程序来说更加简单易用，该驱动将创建一个 BIOS 兼容键盘，键盘支持同时 6 个键同时按下，以及支持 5 个状态灯的控制。

按键的按下与释放将和修饰键一同报告给主机，协议栈同时提供回调函数来通知 LED 状态的变化。

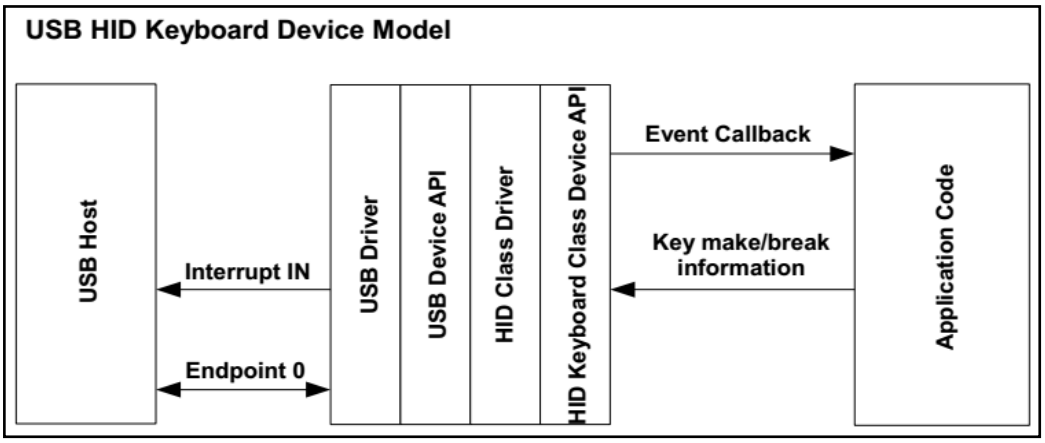


图 2-2 USB HID 键盘驱动模型

### 2.8.1 HID键盘驱动事件

HID 键盘驱动提供了以下回调事件：

- **USB_EVENT_CONNECTED**
- **USB_EVENT_DISCONNECTED**
- **USB_EVENT_TX_COMPLETE**
- **USB_EVENT_ERROR**
- **USB_EVENT_SUSPEND**
- **USB_EVENT_RESUME**
- **USBD_HID_KEYB_EVENT_SET_LEDS**

注：
1. **USB_EVENT_DISCONNECTED** 事件仅在某些器件内支持，具体请查阅相关器件手册，确定设备模式下是否有断开连接检测功能。

### 2.8.2 HID键盘驱动使用教程

HID 键盘驱动使用可以遵循以下几步：

- 将以下头文件加入到需要用到此驱动的源文件中：

```
#include "usblib/drivers/usb_lowlayer_api.h"
#include "usblib/usblib.h"
```

```
#include "usblib/device/usbdhidkeyb.h"
```

- 定义 6 个用来描述设备特征的 USB 字符串描述符，修改字符串内容来描述用户自己的设备，注意当字符串改变后，每个描述符数组的长度（每个数组的第一个元素即为数组长度）也需要相应地进行修改。这里以键盘的例程的字符串描述符为例：

```
/**
  * The languages supported by this device.
  */
const uint8_t g_pui8LangDescriptor[] =
{
  4,
  USB_DTYPE_STRING,
  USBShort(USB_LANG_EN_US)
};

/**
  * The manufacturer string.
  */
const uint8_t g_pui8ManufacturerString[] =
{
  (8 + 1) * 2,
  USB_DTYPE_STRING,
  'E', 0, 'a', 0, 's', 0, 't', 0, 'S', 0, 'o', 0, 'f', 0, 't', 0,
};

/**
  *  The product string.
  */
const uint8_t g_pui8ProductString[] =
{
  (16 + 1) * 2,
  USB_DTYPE_STRING,
  'K', 0, 'e', 0, 'y', 0, 'b', 0, 'o', 0, 'a', 0, 'r', 0, 'd', 0, ' ', 0,
  'E', 0, 'x', 0, 'a', 0, 'm', 0, 'p', 0, 'l', 0, 'e', 0
};

/**
  *  The serial number string.
  */
const uint8_t g_pui8SeriailNumberString[] =
{
  (8 + 1) * 2,
  USB_DTYPE_STRING,
  '1', 0, '2', 0, '3', 0, '4', 0, '5', 0, '6', 0, '7', 0, '8', 0
```

```
};

/**
  *  The interface description string.
  */
const uint8_t g_pui8InterfaceString[] =
{
  (22 + 1) * 2,
  USB_DTYPE_STRING,
  'H', 0, 'I', 0, 'D', 0, ' ', 0, 'K', 0, 'e', 0, 'y', 0, 'b', 0,
  'o', 0, 'a', 0, 'r', 0, 'd', 0, ' ', 0, 'I', 0, 'n', 0, 't', 0,
  'e', 0, 'r', 0, 'f', 0, 'a', 0, 'c', 0, 'e', 0
};

/**
  *  The configuration description string.
  */
const uint8_t g_pui8ConfigString[] =
{
  (26 + 1) * 2,
  USB_DTYPE_STRING,
  'H', 0, 'I', 0, 'D', 0, ' ', 0, 'K', 0, 'e', 0, 'y', 0, 'b', 0,
  'o', 0, 'a', 0, 'r', 0, 'd', 0, ' ', 0, 'C', 0, 'o', 0, 'n', 0,
  'f', 0, 'i', 0, 'g', 0, 'u', 0, 'r', 0, 'a', 0, 't', 0, 'i', 0,
  'o', 0, 'n', 0
};

/**
  *  The descriptor string table.
  */
const uint8_t * const g_ppui8StringDescriptors[] =
{
  g_pui8LangDescriptor,
  g_pui8ManufacturerString,
  g_pui8ProductString,
  g_pui8SeriailNumberString,
  g_pui8InterfaceString,
  g_pui8ConfigString
};

/**
  *  The number of string descriptors.
  */
#define NUM_STRING_DESCRIPTORS (sizeof(g_ppui8StringDescriptors) / sizeof(uint8_t *))
```

- 定义键盘初始化结构体 **tUSBDHIDKeyboardDevice**。

```
const tUSBDHIDKeyboardDevice g_sKeyboardDevice =
{
    // The Vendor ID you have been assigned by USB-IF.
    USB_VID_YOUR_VENDOR_ID,
    // The product ID you have assigned for this device.
    USB_PID_YOUR_PRODUCT_ID,
    // The power consumption of your device in milliamps.
    POWER_CONSUMPTION_MA,
    // The value to be passed to the host in the USB configuration descriptor's
    // bmAttributes field.
    USB_CONF_ATTR_SELF_PWR,
    // A pointer to your keyboard callback event handler.
    YourKeyboardHandler,
    // A value that you want passed to the callback alongside every event.
    (void *)&g_sYourInstanceData,
    // A pointer to your string table.
    g_pStringDescriptors,
    // The number of entries in your string table. This must equal
    // (1 + (5 * (num languages))).
    NUM_STRING_DESCRIPTORS
};
```

- 添加键盘事件处理函数，YourKeyboardHandler() 函数。**USB_EVENT_TX_COMPLETE** 事件可以用来确认键盘报告数据是否成功传输到主机端。
- 在主函数中调用具体的 HID 键盘设备初始化函数来配置 USB 控制器， HID 键盘初始化函数 usbdhid_keyboard_init(0, & g_sMouseDevice);
- 在 USB 中断函数中调用 USB 协议栈设备中断处理函数 usb_device_int_handler();

### 2. 8. 3 HID键盘设备驱动接口定义

**数据结构：**

- tUSBDHIDKeyboardDevice

宏：

- KEYB_ERR_NOT_CONFIGURED
- KEYB_ERR_NOT_FOUND
- KEYB_ERR_TX_ERROR
- KEYB_ERR_TOO_MANY_KEYS
- KEYB_SUCCESS

- USBD_HID_KEYB_EVENT_SET_LEDS

**函数接口：**

- void * usbdhid_keyboard_composite_init(uint32_t ui32Index, tUSBDHIDMouseDevice *psHIDKbDevice, tCompositeEntry * psCompEntry);
- void * usbdhid_keyboard_init(uint32_t ui32Index, tUSBDHIDKeyboardDevice *psHIDKbDevice);
- void usbdhid_keyboard_term(void * pvMouseDevice);
- void *usbdhid_keyboard_set_cb_data(void *pvKeyboardInstance, void *pvCBData);
- void usbdhid_keyboard_power_status_set(void * pvKeyboardInstance, uint8_t ui8Power);
- bool usbdhid_keyboard_remote_wakeup_request(void *pvHIDInstance);
- uint32_t usbdhid_keyboard_key_state_change(void *pvKeyboardInstance, uint8_t ui8Modifiers, uint8_t ui8UsageCode, bool bPressed);

#### 2. 8. 3. 1 tUSBDHIDKeyboardDevice

**定义：**

```
typedef struct
{
  const uint16_t ui16VID;
  const uint16_t ui16PID;
  const uint16_t ui16MaxPowermA;
  const uint8_t ui8PwrAttributes;
  const tUSBCallback pfnCallback;
  void *pvCBData;
  const uint8_t *const *ppui8StringDescriptors;
  const uint32_t ui32NumStringDescriptors;
  tHIDKeyboardInstance sPrivateData;
} tUSBDHIDKeyboardDevice;
```

**参数定义：**

**ui16VID**：厂商 ID，由 USB-IF 统一分配。

**ui16PID**：产品 ID，由制造商分配。

**ui16MaxPowermA**：最大供电电流（mA）。

**ui8PwrAttributes**：设备供电特性，值可以是 **USB_CONF_ATTR_SELF_PWR**、**USB_CONF_ATTR_BUS_PWR**，另外可选的配置为 **USB_CONF_ATTR_RWAKE**（如果设备支持此功能）。

**pfnCallback**：键盘事件处理回调函数。

**pvCBData**：回调函数参数指针。

**ppui8StringDescriptors**：指向字符串描述符表的指针。

**ui32NumStringDescriptors**：设备字符串描述符表的大小。

**sPrivateData**：HID 类驱动私有参数，此内存空间必须可以访问但是不可以被 HID 类驱动以外的程序代码修改。

#### 2. 8. 3. 2 KEYB_ERR_NOT_CONFIGURED

**描述**：在设备连接和配置之前，如果调用 usbdhid_keyboard_state_change()函数将返回此值。

#### 2. 8. 3. 3 KEYB_ERR_NOT_FOUND

**描述**：当 bPress 参数设为 false 但是 ui8UsageCode 参数指定的按键最近没有被按下的记录时， usbdhid_keyboard_state_change()函数会返回此值。这种现象通常发生在先前报告超过 6 个按键，先前的按键记录被删除。这种情况不应该视为断开或者错误。

#### 2. 8. 3. 4 KEYB_ERR_TOO_MANY_KEYS

**描述**：当一次报告超过 6 个按键时， usbdhid_keyboard_state_change()函数会返回此值。USB HID BIOS 键盘协议允许最多不超过 6 个按键同时被按下，当超过 6 个按键被按下时，再次报告其它按键被按下就需要等待先前 6 个被按下的按键释放。

#### 2. 8. 3. 5 KEYB_ERR_TX_ERROR

**描述**：调用 usbdhid_keyboard_state_change()函数返回此值时表示报告有错误产生。

#### 2. 8. 3. 6 KEYB_SUCCESS

**描述**：usbdhid_keyboard_state_change()函数返回此值表明调用成功。

#### 2. 8. 3. 7 USBD_HID_KEYB_EVENT_SET_LEDS

**描述**：本事件表明键盘的 LED 灯状态发生改变，回调函数 ui32MsgValue 参数包含了每个 LED 的状态位，1 代表 LED 打开，0 代表 LED 关闭。每个 LED 的位使用宏定义 **HID_KEYB_NUM_LOCK** 、 **HID_KEYB_CAPS_LOCK** 、 **HID_KEYB_SCROLL_LOCK**、**HID_KEYB_COMPOSE** 和 **HID_KEYB_KANA** 来定义

#### 2. 8. 3. 8 usbdhid_keyboard_composite_init()

当需要创建复合设备时用于 HID 键盘的初始化。

**参数定义**：

**ui32Index**：USB 控制器索引。

**psHIDKbDevice**：指向 HID 键盘初始化结构体 **tUSBDHIDKeyboardDevice** 的指针。

**psCompEntry**：指向复合设备初始化入口结构体 **tCompositeEntry** 的指针，当需要创建复合设备时作为复合设备的初始化入口参数。

**简介：**

本函数和 HID 键盘初始化函数类似，当需要创建复合函数时，这个函数会初始化指向 **tCompositeEntry** 结构体的入口参数。

**返回值：**

失败返回 0，成功则返回指向 **tUSBDHIDKeyboardDevice** 的指针，用于接下来 HID 键盘驱动 API 的参数。

#### 2. 8. 3. 9 usbdhid_keyboard_init()

初始化 HID 键盘设备。

**参数定义：**

**ui32Index**：USB 控制器索引。

**psHIDKbDevice**：指向 HID 键盘初始化结构体 **tUSBDHIDKeyboardDevice** 的指针。

**简介：**

应用程序希望创建一个 HID 键盘设备与 PC 端传输数据，可以调用这个函数来初始化 USB 控制器，此函数执行所有的 HID 键盘设备的初始化工作。

此函数调用成功后会返回传递给它的 **tUSBDHIDKeyboardDevice** 指针，这个指针必须作为以后所有的 HID 键盘驱动 API 的参数。

**返回值：**

失败返回 NULL，成功则返回指向 **tUSBDHIDKeyboardDevice** 的指针。

#### 2. 8. 3. 10 usbdhid_keyboard_power_status_set()

修改设备的供电特性。

**参数定义：**

**pvKeyboardInstance**：指向由 usbdhid_keyboard_init()函数初始化返回的 HID 键盘初始化结构体 **tUSBDHIDKeyboardDevice** 的指针。

**ui8Power**：供电特性。

**简介：**

USB 协议栈支持设备供电特性的修改，修改的值可以是 **USB_STATUS_SELF_PWR** 或者 **USB_STATUS_BUS_PWR**。在 USB 主机设备再次请求后，USB 协议栈会将修改后的供电特性报告给主机。

**返回值：**

无。

#### 2. 8. 3. 11 usbdhid_keyboard_key_state_change()

向主机发送键盘报表数据。

**参数定义：**

**pvKeyboardInstance**：指向由 usbdhid_keyboard_init()函数初始化返回的 HID 键盘初始化结构体 **tUSBDHIDKeyboardDevice** 的指针。

**ui8Modifiers**：包含每个修饰键的状态（左右 shift、ctrl、alt、GUI）。有效值可以是 **HID_KEYB_LEFT_CTRL**、**HID_KEYB_LEFT_SHIFT**、**HID_KEYB_LEFT_ALT**、**HID_KEYB_LEFT_GUI**、**HID_KEYB_RIGHT_CTRL**、**HID_KEYB_RIGHT_SHIFT**、**HID_KEYB_RIGHT_ALT** 和 **HID_KEYB_RIGHT_GUI**。

**ui8UsageCode**：按键所对应的键页值。

**bPressed**：true 代表被按下，false 代表释放。

**简介：**

此函数用来向按键列表（将会发送到主机，每次报告最多 6 个按键状态）中添加或者删除一个按键。

**返回值：**

返回 **KEYB_SUCCESS** 表明将成功发送，返回 **KEYB_ERR_TX_ERROR** 表明发送时出错（最典型的原因是当前包在发送时上一包还没有发送出去）。返回 **KEYB_ERR_NOT_CONFIGURED** 表明在调用此函数时还没有成功连接主机或被主机配置，删除按键返回 **KEYB_ERR_NOT_FOUND** 表明按键列表中没有此键。

#### 2. 8. 3. 12 usbdhid_keyboard_term()

停用 HID 键盘驱动。

**参数定义：**

**pvKeyboardInstance**：指向由 usbdhid_keyboard_init()函数初始化返回的 HID 键盘初始化结构体 **tUSBDHIDKeyboardDevice** 的指针。

**简介：**

此函数将终止所有 HID 键盘驱动操作，并从总线删除设备，如果 HID 键盘设备是复合设备的一部分，那么不应该调用这个函数，而应该为整个复合设备调用 usbd_composite_term()函数。在调用此函数之后，**tUSBDHIDKeyboardDevice** 指针实例不应该在任何其它 API 中使用。

**返回值：**

无。

#### 2. 8. 3. 13 usbdhid_keyboard_set_cb_data()

修改 HID 键盘驱动的回调函数。

**参数定义：**

**pvKeyboardInstance**：指向由 usbdhid_keyboard_init()函数初始化返回的 HID 键盘初始化结构体 **tUSBDHIDKeyboardDevice** 的指针。

**pvCBData**：指向键盘回调函数的指针。

**简介：**

应用程序使用此函数来修改由 usbdhid_keyboard_init()函数配置好的回调函数指针。如果应用程序需要在运行时修改接收回调函数的指针值，需要保证 **tUSBDHIDKeyboardDevice** 存储在 SRAM 中。

**返回值：**

返回先前使用的回调函数指针值。

#### 2. 8. 3. 14 usbdhid_keyboard_remote_wakeup_request()

当 USB 控制器处于挂起状态时，发送“wakeup”信号到主机。

**参数定义：**

**pvKeyboardInstance**：指向由 usbdhid_keyboard_init()函数初始化返回的 HID 键盘初始化结构体 **tUSBDHIDKeyboardDevice** 的指针。

**简介：**

当处于挂起状态时，如果从机支持远程唤醒（通过初始化结构体设置）功能的话，可以调用此函数来唤醒主机。

**返回值：**

返回 true 表示远程唤醒功能没有被禁止且被发送，返回 false 表示远程唤醒被禁止。

## 2. 9 HID游戏手柄设备驱动

HID 游戏手柄驱动提供给应用程序手柄相关的 API，应用程序可以使用此接口很便捷地创建一个 HID 游戏手柄设备。手柄默认支持 3 个 8 位轴值（X、Y、Z）以及 8 个按钮，手柄默认每 1 毫秒通过调用 usbdhid_gamepad_send_report()函数向主机报告一次数据。

### 2. 9. 1 HID手柄驱动使用教程

HID 手柄驱动使用可以遵循以下几步：

- 将以下头文件加入到需要用到此驱动的源文件中：

```
#include "usblib/drivers/usb_lowlayer_api.h"
#include "usblib/usbhid.h"
#include "usblib/device/usbdevice.h"
#include "usblib/device/usbdhid.h"
#include "usblib/device/usbdhidgamepad.h"
```

- 定义 6 个用来描述设备特征的 USB 字符串描述符，定义方法可以参考 HID 设备类字符串描述符介绍。
- 应用程序必须提供手柄的事件回调函数，USB 协议栈中的事件格式为 **USB_EVEVT_**，HID 手柄应用程序可用的事件有：

- **USB_EVENT_CONNECTED**
- **USB_EVENT_DISCONNECTED**
- **USB_EVENT_TX_COMPLETE**
- **USB_EVENT_ERROR**
- **USB_EVENT_SUSPEND**
- **USB_EVENT_RESUME**
- **USB_EVENT_LPM_SLEEP**
- **USB_EVENT_LPM_RESUME**
- **USB_EVENT_LPM_ERROR**

注：
1. **USB_EVENT_DISCONNECTED** 事件仅在某些器件内支持，具体请查阅相关器件手册，确定设备模式下是否有断开连接检测功能。
2. 不是所有 ES32 系列控制器 USB 模块都支持 LMP，**USB_EVENT_LPM_**类型的事件需要相应的设备支持才可以用。

- 定义手柄初始化结构体 **tUSBDHIDGamepadDevice**。

```
const tUSBDHIDGamepadDevice g_sGamepadDevice =
{
    USB_VID_YOUR_VENDOR_ID,
    USB_PID_YOUR_PRODUCT_ID,
    POWER_CONSUMPTION_MA,
    USB_CONF_ATTR_SELF_PWR,
    YourGamepadHandler,
    (void *)&g_sYourInstanceData,
    g_pStringDescriptors,
    NUM_STRING_DESCRIPTORS,
    g_pui8GameReportDescriptor,
    sizeof(g_pui8GameReportDescriptor)
};
```

- 在主函数中调用具体的 HID 手柄设备初始化函数来配置 USB 控制器， HID 手柄初始化函数 usbdhid_gamepad_init(0, & g_sMouseDevice);
- 在 USB 中断函数中调用 USB 协议栈设备中断处理函数 usb_device_int_handler()；
- 一旦手柄连接上主机，手柄便可以通过 usbdhid_gamepad_send_repor()函数发送报告数据，成功发送数据后 **USB_EVENT_TX_COMPLETE** 事件会被发往应用程序。

示例：

```
tGamepadReport sReport;
```

```
//
// Send a gamepad HID report.
//
usbdhid_gamepad_send_report(&g_sGamepadDevice, & sReport, sizeof(sReport));
```

- 如果用户有需要重新定义游戏手柄的报告，那么需要修改现有的协议栈程序中的 HID 游戏手柄的 HID 报告描述符。USB 协议栈目前参考 USB HID 1.11 版本的规范文件，用户如果需要自定义 HID 报告描述符需要参阅此文件。以下示例中首先定义了 12 its 的 4 轴控制器，每个 12 bits 的数值代表每个轴的数据，每个轴的报告数据后填充 4 bits 的无意义数据，最后定义了 16 个按钮，每个按钮值宽度为 1 bit。

```
//*****************************************************************************
static const uint8_t g_pui8GameReportDescriptor[] =
{
    UsagePage(USB_HID_GENERIC_DESKTOP),
    Usage(USB_HID_JOYSTICK),
    Collection(USB_HID_APPLICATION),
    // The axis for the controller.
    UsagePage(USB_HID_GENERIC_DESKTOP),
    Usage (USB_HID_POINTER),
    Collection (USB_HID_PHYSICAL),
    // The X, Y, RX, and RY values, which are specified as 8-bit
    // absolute position values.
    Usage (USB_HID_X),
    // 12-bit absolute X value.
    ReportSize(12),
    ReportCount(1),
    Input(USB_HID_INPUT_DATA | USB_HID_INPUT_VARIABLE |
    USB_HID_INPUT_ABS),
    // 4-bit padding to 16 bits.
    ReportCount(1),
    ReportSize(4),
    Input(USB_HID_INPUT_CONSTANT),
    // 12-bit absolute Y value.
    Usage (USB_HID_Y),
    ReportSize(12),
    ReportCount(1),
    Input(USB_HID_INPUT_DATA | USB_HID_INPUT_VARIABLE |
    USB_HID_INPUT_ABS),
    // 4-bit padding to 16 bits.
    ReportCount(1),
    ReportSize(4),
    Input(USB_HID_INPUT_CONSTANT),
    // 12-bit absolute RX value.
```

```
    Usage (USB_HID_RX),
    ReportSize(12),
    ReportCount(1),
    Input(USB_HID_INPUT_DATA | USB_HID_INPUT_VARIABLE |
    USB_HID_INPUT_ABS),
    // 4-bit padding to 16 bits.
    ReportCount(1),
    ReportSize(4),
    Input(USB_HID_INPUT_CONSTANT),
    // 12-bit absolute RY value.
    Usage (USB_HID_RY),
    ReportSize(12),
    ReportCount(1),
    Input(USB_HID_INPUT_DATA | USB_HID_INPUT_VARIABLE |
    USB_HID_INPUT_ABS),
    // 4-bit padding to 16 bits.
    ReportCount(1),
    ReportSize(4),
    Input(USB_HID_INPUT_CONSTANT),
    // The 16 buttons.
    UsagePage(USB_HID_BUTTONS),
    UsageMinimum(1),
    UsageMaximum(16),
    LogicalMinimum(0),
    LogicalMaximum(1),
    PhysicalMinimum(0),
    PhysicalMaximum(1),
    // 16 1-bit values for the buttons.
    ReportSize(1),
    ReportCount(16),
    Input(USB_HID_INPUT_DATA | USB_HID_INPUT_VARIABLE |
    USB_HID_INPUT_ABS),
    EndCollection,
    EndCollection
};
//*****************************************************************************
//
// The packed custom report structure that is sent to the host to match the
// report descriptor defined above.
//
//*****************************************************************************
typedef struct
{
    uint16_t i16XPos;
```

```
    uint16_t i16YPos;
    uint16_t i16RXPos;
    nt16_t i16RYPos;
    nt16_t ui16Buttons;
} CustomReport;
```

### 2. 9. 2 HID手柄驱动接口定义

**数据结构：**

- tGamepadReport
- tUSBDHIDGamepadDevice

宏：

- USBDGAMEPAD_NOT_CONFIGURED
- USBDGAMEPAD_SUCCESS
- USBDGAMEPAD_TX_ERROR

**函数接口：**

- tUSBDHIDGamepadDevice *usbdhid_gamepad_init(uint32_t ui32Index, tUSBDHIDGamepadDevice *psHIDGamepad);
- tUSBDHIDGamepadDevice *usbdhid_gamepad_composite_init( uint32_t ui32Index, tUSBDHIDGamepadDevice *psHIDGamepad, tCompositeEntry *psCompEntry);
- void usbdhid_gamepad_term(tUSBDHIDGamepadDevice *psCompEntry);
- uint32_t usbdhid_gamepad_send_report( tUSBDHIDGamepadDevice *psHIDGamepad, void *pvReport, uint32_t ui32Size);

#### 2. 9. 2. 1 tUSBDHIDGamepadDevice

**定义：**

```
typedef struct
{
    const uint16_t ui16VID;
    const uint16_t ui16PID;
    const uint16_t ui16MaxPowermA;
    const uint8_t ui8PwrAttributes;
    const tUSBCallback pfnCallback;
    void *pvCBData;
    const uint8_t *const *ppui8StringDescriptors;
    const uint32_t ui32NumStringDescriptors;
    const uint8_t *pui8ReportDescriptor;
    const uint32_t ui32ReportSize;
    tUSBDGamepadInstance sPrivateData;
```

```
} tUSBDHIDGamepadDevice;
```

**参数定义：**

**ui16VID**：厂商 ID，由 USB-IF 统一分配。

**ui16PID**：产品 ID，由制造商分配。

**ui16MaxPowermA**：最大供电电流（mA）。

**ui8PwrAttributes**：设备供电特性，值可以是 **USB_CONF_ATTR_SELF_PWR**、**USB_CONF_ATTR_BUS_PWR**，另外可选的配置为 **USB_CONF_ATTR_RWAKE**（如果设备支持此功能）。

**pfnCallback**：手柄事件处理回调函数。

**pvCBData**：回调函数参数指针。

**ppui8StringDescriptors**：指向字符串描述符表的指针。

**ui32NumStringDescriptors**：设备字符串描述符表的大小。

**pui8ReportDescriptor**:可选的 HID 报告描述符，如果用户需要重新定义 HID 报告描述符，那么就通过此项传入协议栈。

**ui32ReportSize**：可选的报告描述符大小

**sPrivateData**：HID 类驱动私有参数，此内存空间必须可以访问但是不可以被 HID 类驱动以外的程序代码修改。

#### 2. 9. 2. 2 tGamepadReport

**定义：**

```
typedef struct
{
  int8_t i8XPos;
  int8_t i8YPos;
  int8_t i8ZPos;
  uint8_t ui8Buttons;
}tGamepadReport;
```

**描述：**

本结构体定义了协议栈默认的 HID 游戏手柄的报表结构，该结构对应于协议栈默认的 HID 游戏手柄报告描述符。如果用户定义了自定义手柄报告描述符，那么这个结构体将不能再使用。

#### 2. 9. 2. 3 USBDGAMEPAD_NOT_CONFIGURED

**描述**：指示设备还没有被配置，不可以进行任何操作。

#### 2. 9. 2. 4 USBDGAMEPAD_SUCCESS

**描述**：成功调用 usbdhid_gamepad_send_report()函数并开始调度数据发送。

#### 2. 9. 2. 5 USBDGAMEPAD_TX_ERROR

**描述**：调用 usbdhid_gamepad_send_report()函数时产生错误，数据不能被发送。

#### 2. 9. 2. 6 usbdhid_gamepad_composite_init()

当需要创建复合设备时用于 HID 手柄的初始化。

**参数定义**：

**ui32Index**：USB 控制器索引。

**psHIDGamepad**：指向 HID 手柄初始化结构体 **tUSBDHIDGamepadDevice** 的指针。

**psCompEntry**：指向复合设备初始化入口结构体 **tCompositeEntry** 的指针，当需要创建复合设备时作为复合设备的初始化入口参数。

**简介**：

本函数和 HID 手柄初始化函数类似，当需要创建复合函数时，这个函数会初始化指向 **tCompositeEntry** 结构体的入口参数。

**返回值**：

失败返回 0，成功则返回指向 **tUSBDHIDGamepadDevice** 的指针，用于接下来 HID 手柄驱动 API 的参数。

#### 2. 9. 2. 7 usbdhid_gamepad_init()

初始化 HID 手柄设备。

**参数定义**：

**ui32Index**：USB 控制器索引。

**psHIDGamepad**：指向 HID 手柄初始化结构体 **tUSBDHIDGamepadDevice** 的指针。

**简介**：

应用程序希望创建一个 HID 手柄设备与 PC 端传输数据，可以调用这个函数来初始化 USB 控制器，此函数执行所有的 HID 手柄设备的初始化工作。

此函数调用成功后会返回传递给它的 **tUSBDHIDGamepadDevice** 指针，这个指针必须作为以后所有的 HID 手柄驱动 API 的参数。

**返回值**：

失败返回 NULL，成功则返回指向 **tUSBDHIDGamepadDevice** 的指针。

#### 2. 9. 2. 8 usbdhid_gamepad_send_report()

修改设备的供电特性。

**参数定义**：

**psHIDGamepad**：指向由 usbdhid_gamepad_init()函数初始化返回的 HID 手柄初始化结构体 **tUSBDHIDGamepadDevice** 的指针。

**pvReport**：手柄报告数据指针。

**ui32Size**：报告数据大小。

**简介：**

调用此函数后发送下一次报告数据时，必须等待先前一次发送数据的 **USB_EVENT_TX_COMPLETE** 事件到来。返回 **USBDGAMEPAD_SUCCESS** 表示报告数据成功被协议栈接收并开始调度发送；返回 **USBDGAMEPAD_TX_ERROR** 表示数据发送出错，不可以发送；返回 **USBDGAMEPAD_NOT_CONFIGURED** 表示手柄还没有被配置，不能使用。

**返回值：**

返回 **USBDGAMEPAD_**类型的值。

#### 2. 9. 2. 9 usbdhid_gamepad_term()

停用 HID 手柄驱动。

**参数定义：**

**psHIDGamepad**：指向由 usbdhid_gamepad_init()函数初始化返回的 HID 手柄初始化结构体 **tUSBDHIDGamepadDevice** 的指针。

**简介：**

此函数将终止所有 HID 手柄驱动操作，并从总线删除设备，如果 HID 手柄设备是复合设备的一部分，那么不应该调用这个函数，而应该为整个复合设备调用 usbd_composite_term()函数。在调用此函数之后，**tUSBDHIDGamepadDevice** 指针实例不应该在任何其它 API 中使用。

**返回值：**

无。

# 第3章 主机部分

## 3.1 简介

本章节将介绍 ES32 USB 协议栈主机部分相关驱动，主机部分的驱动涉及从底层 MD/ALD 库到高层驱动部分包括主机类驱动、主机驱动，用户可以从适当的层次去开发。为了简化应用程序以及添加新的主机或者主机类驱动，USB 协议栈在 USB 核心驱动上产生了许多支持不同主机类的驱动，这一层为主机类驱动层。在主机类层驱动上产生了具体的主机驱动，主要用于处理具体主机的数据传输操作。主机库同样会和设备库一样提供丰富的回调来通知应用程序相关的操作。

### 3.1.1 源码文件介绍

USB 协议栈的主机部分相关驱动的源码以及头文件放在 usblib/host 文件夹下。

| | |
|---|---|
| usbhcore.c | USB 协议栈核心驱动文件，处理主机枚举以及数据初始传输的一些操作。 |
| usbhaudio.c | AUDIO 类驱动的源文件，本文件提供 USB AUDIO 设备类主机驱动相关的 API。 |
| usbhhandler.c | USB 协议栈设备中断处理文件，本文件包含主机中断的处理函数。 |
| usbhhid.c | HID 类主机驱动源文件，本文件提供 HID 类驱动的相关 API。 |
| usbhhidkeyboard.c | HID 键盘主机驱动源文件，本文件提供 HID 键盘驱动的相关 API。 |
| usbhhidmouse.c | HID 鼠标主机驱动源文件，本文件提供 HID 鼠标驱动的相关 API。 |
| usbhmsc.c | MSC 类主机驱动源文件，本文件提供 MSC 类驱动的相关 API。 |
| usbhhub.c | HUB 类主机驱动源文件，本文件提供 HUB 类驱动的相关 API。 |
| usbhmsc.c | MSC 类主机驱动源文件，本文件提供 MSC 类驱动的相关 API。 |

### 3.1.2 主机部分驱动结构

USB 协议栈主机部分的驱动是建立在 MD/ALD 库基础上的一套三层结构的函数库，用户可以在此三层结构上开发设备。协议栈分为核心层、主机类层、主机层的驱动。其中核心层负责处理 USB 的中断以及将相应的事件传输给主机类层驱动、USB 数据的初步处理，当有需要传输到主机类层驱动的数据，核心层会以事件回调的方式将数据传达主机类层驱动；主机类层驱动负责 USB 各主机类协议的处理，核心层将相关的事件数据传到主机类层时，主机类会将数据进一步处理，如果主机类下有多个子类主机的话，需要子类用到的数据也会以事件回调的方式传输到主机层；USB 某些主机类下有多个子类，例如 HID 主机类下有 HID 键盘、HID 鼠标等主机，那么这些子类主机的驱动放在主机层中，主机层的功能和相关的子主机相关，如 HID 鼠标主机驱动中就提供了 HID 鼠标的描述符以及数据发送的 API。

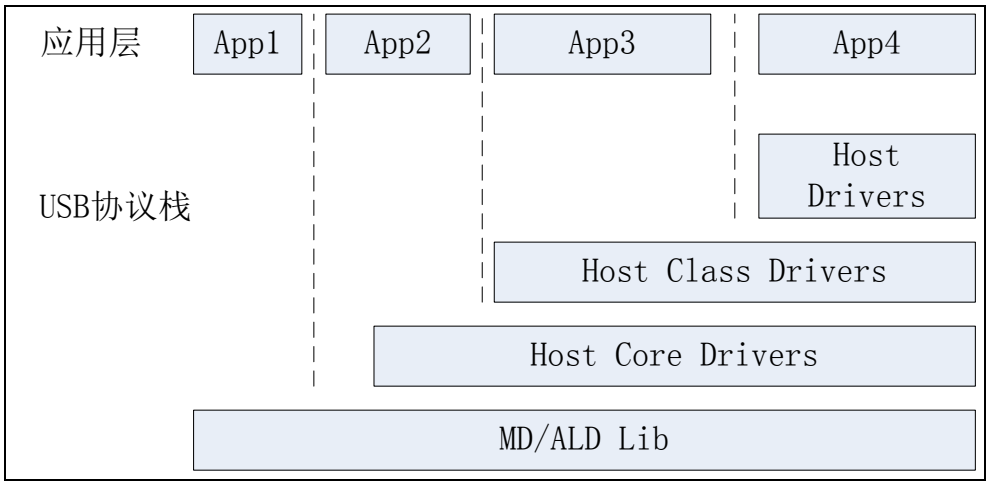


图 3-1 USB 协议栈主机驱动结构

## 3.2 核心层驱动

### 3.2.1 简介

和设备部分驱动相似，USB 协议栈底层的驱动为 MD 库驱动，文件为 Drivers\MD\[device]\Source\md_usb.c。

USB 协议栈核心层驱动提供了枚举阶段所需要的功能，枚举操作执行不考虑设备类型的操作并且允许上层驱动程序做具体的操作。为了上层驱动能够只处理本类设备的数据，核心层驱动只将与上层驱动相关的数据发向对应的上层驱动，例如 HID 类的数据，核心层驱动只会将和 HID 类相关的数据发送到 HID 类驱动层，所以协议栈允许应用程序处理多个类的设备。

#### 3.2.1.1 枚举

USB 核心层驱动处理所有需要发现和枚举任何 USB 设备所需的细节。USB 主机核心层驱动执行依赖主机类驱动程序的枚举，和 USB 设备进行任何其他通信，以及对该设备端点的分配。用于枚举的大多数代码都在中断上下文中执行，并包含在枚举处理程序中。为了完成枚举过程，主机核心层驱动要求应用程序定期调用 usbhcd_main()函数，当主机需要访问设备端点 0 的时候，可以使用 usbhcd_control_transfer()函数将数据发送到设备或者接收设备的数据。枚举过程中主机核心层驱动会搜索应用程序在调用 usbhcd_register_drivers()函数时注册的主机类驱动，如果主机核心层驱动找到与该枚举设备类匹配的应用程序，那么将打开该驱动，如果主机核心层驱动没有发现对应的驱动，那么主机将忽略这个设备。主机核心驱动以及主机类驱动都通过 USB 协议栈提供回调，以通知应用程序相关的事件。主机类驱动负责配置基于被发现的设备类的 USB 管道。主机协议栈将通过两种方式通知应用程序：一种来自于连接设备的主机类驱动，另一个来自于非设备特定的事件驱动程序。类特定的事件来自于主机类驱动程序，而通用的如连接等和其它设备类无关的事件来自于主机核心层的事件驱动程序。

#### 3.2.1.2 管道

USB 核心层驱动使用称为管道的接口作为主要的和 USB 设备通信的接口。这些管道可以在 USB 枚举期间由 USB 驱动程序动态分配或者静态分配。USB 管道通常只在

USB 核心驱动以及主机类驱动中使用，一般不由应用程序直接访问。USB 管道通过调用 usbhcd_pipe_alloc()函数和 usbhcd_pipe_free()函数来创建以及消除，通过调用 usbhcd_pipe_config()函数来配置。usbhcd_pipe_alloc()和 usbhcd_pipe_config()函数在 USB 设备枚举中使用，将 USB 管道分配到 USB 设备特定的端点。在断开连接后将通过 usbhcd_pipe_free()函数回收。在使用管道期间，协议栈会通过 usbhcd_pipe_write()、usbhcd_pipe_read()函数以及主机类驱动提供的回调函数访问管道。由于端点 0 为所有设备公用，所以端点 0 的数据传输将通过 usbhcd_control_transfer()函数来完成。

#### 3. 2. 1. 3 控制传输

所有的 USB 控制传输事务都通过 usbhcd_control_transfer()函数来处理。这个函数主要用于枚举期间核心层驱动与设备的数据传输，但是有些设备在枚举过后也是通过端点 0 来传输的，HID 类驱动程序就是一个例子。因为 usbhcd_control_transfer()函数是一个依赖于中断的阻塞函数，所以不应该在中断函数中调用该函数。由于大多数回调是在中断状态中发生的，所以 usbhcd_control_transfer()函数的调用不应该放在应用程序回调函数中，而是应该推迟到回调函数以外执行。

#### 3. 2. 1. 4 中断处理

所有的中断处理都由 USB 协议栈提供的函数来处理，大多数回调是在中断上下文中完成的，所以中断处理程序应该推迟任何在中断上下文之外发生的事件的实际处理，实际应用中应用程序应该定期调用 usbhcd_main()函数来处理 USB 各种调度。

### 3. 2. 2 核心层接口定义

**数据结构：**

- tUSBHostClassDriver

**函数接口：**

- void usb_host_int_handler(void);
- uint32_t usbhcd_control_transfer(uint32_t ui32Index, tUSBRequest *psSetupPacket, tUSBHostDevice *psDevice, uint8_t *pui8Data, uint32_t ui32Size, uint32_t ui32MaxPacketSize);
- uint8_t usbhcd_dev_class(uint32_t ui32Instance, uint32_t ui32Interface);
- uint8_t usbhcd_dev_hub_port(uint32_t ui32Instance);
- uint8_t usbhcd_dev_protocol(uint32_t ui32Instance, uint32_t ui32Interface);
- uint8_t usbhcd_dev_sub_class(uint32_t ui32Instance, uint32_t ui32Interface);
- int32_t usbhcd_event_enable(uint32_t ui32Index, void *pvEventDriver, uint32_t ui32Event);
- int32_t usbhcd_event_disable(uint32_t ui32Index, void *pvEventDriver, uint32_t ui32Event);
- bool usbhcd_feature_set(uint32_t ui32Index, uint32_t ui32Feature, void

*pvFeature);

- void usbhcd_init(uint32_t ui32Index, void *pvData, uint32_t ui32Size);
- void usbhcdlpm_resume(uint32_t ui32Index);
- uint32_t usbhcdlpm_sleep(tUSBHostDevice *psDevice);
- void usbhcd_main(void);
- uint32_t usbhcd_pipe_alloc(uint32_t ui32Index, uint32_t ui32EndpointType, tUSBHostDevice *psDevice, tHCDPipeCallback pfnCallback);
- uint32_t usbhcd_pipe_alloc_size(uint32_t ui32Index, uint32_t ui32EndpointType, tUSBHostDevice *psDevice, uint32_t ui32FIFOSize, tHCDPipeCallback pfnCallback);
- uint32_t usbhcd_pipe_config(uint32_t ui32Pipe, uint32_t ui32MaxPayload, uint32_t ui32Interval,uint32_t ui32TargetEndpoint);
- void usbhcd_pipe_data_ack(uint32_t ui32Pipe);
- void usbhcd_pipe_free(uint32_t ui32Pipe);
- uint32_t usbhcd_pipe_read(uint32_t ui32Pipe, uint8_t *pui8Data, uint32_t ui32Size);
- uint32_t usbhcd_pipe_read_non_blocking(uint32_t ui32Pipe, uint8_t *pui8Data, uint32_t ui32Size);
- uint32_t usbhcd_pipe_schedule(uint32_t ui32Pipe, uint8_t *pui8Data, uint32_t ui32Size);
- uint32_t usbhcd_pipe_status(uint32_t ui32Pipe);
- uint32_t usbhcd_pipe_transfer_size_get(uint32_t ui32Pipe);
- uint32_t usbhcd_pipe_write(uint32_t ui32Pipe, uint8_t *pui8Data, uint32_t ui32Size);
- uint32_t usbhcd_power_automatic(uint32_t ui32Index);
- uint32_t usbhcd_power_config_get(uint32_t ui32Index);
- void usbhcd_power_config_init(uint32_t ui32Index, uint32_t ui32Flags);
- uint32_t usbhcd_power_config_set(uint32_t ui32Index, uint32_t ui32Config);
- void usbhcd_register_drivers(uint32_t ui32Index, const tUSBHostClassDriver *const *ppsHClassDrvrs, uint32_t ui32NumDrivers);
- void usbhcd_reset(uint32_t ui32Index);
- void usbhcd_resume(uint32_t ui32Index);
- void usbhcd_suspend(uint32_t ui32Index);

- void usbhcd_term(uint32_t ui32Index);
- void usbhcd_set_interface(uint32_t ui32Index, uint32_t ui32Device, uint32_t ui32Interface, uint32_t ui32AltSetting);
- uint32_t usbhcd_string_descriptor_get(tUSBHostDevice *psDevice, uint8_t *pui8Buffer, uint32_t ui32Size, uint32_t ui32LangID, uint32_t ui32StringIndex);

#### 3. 2. 2. 1 tUSBHostClassDriver

定义：

```
typedef struct
{
  uint32_t ui32InterfaceClass;
  void *(*pfnOpen)(tUSBHostDevice *psDevice);
  void (*pfnClose)(void *pvInstance);
  void (*pfnIntHandler)(void *pvInstance);
} tUSBHostClassDriver;
```

**参数定义：**

**ui32InterfaceClass**：设备驱动的接口类型。

**pfnOpen**：主机驱动接口打开函数，当设备检测到时 USB 协议栈会打开此函数。

**pfnClose**：主机驱动接口关闭函数，当设备断开后，先前已经有调用过打开函数则调用此函数关闭驱动。

**pfnIntHandler**：可选的中断处理函数，当和此设备端点相关的中断产生时会被调用。

#### 3. 2. 2. 2 usb_host_int_handler()

USB 协议栈主机处理函数，在 USB 中断服务函数中调用。

**简介：**

USB 中断的入口点，这个函数将根据 USB 控制器当前的中断状态来调度到不同的处理程序中。

**返回值：**

无。

#### 3. 2. 2. 3 usbhcd_control_transfer()

此函数用于完成一次与设备的控制传输。

**参数定义：**

**ui32Index**：USB 控制器索引。

**psSetupPacket**：“setup”阶段主机请求数据。

**psDevice**：设备类指针，指向一个设备实体。

**pui8Data**：OUT 或者 IN 请求的数据指针。

**ui32Size**：pui8Data 数据的大小。

**ui32MaxPacketSize**：本次传输的最大包大小。

**简介：**

本函数通过控制事务处理必要的状态改变。清注意这个函数不要在中断或者 USB 回调中调用，因为此函数是一个阻塞函数。

**返回值：**

本次请求读取或者发送的数据大小。

#### 3. 2. 2. 4 usbhcd_dev_class()

本函数根据请求的 USB 实体返回 USB 设备类。

**参数定义：**

**ui32Instance**：查询设备的唯一值。

**ui32Interface**：连接 USB 类的接口号。

**简介：**

本函数返回 ui32Instance 参数指向的设备的 USB 类。ui32Instance 的值在核心层的事件驱动中在 **USB_EVENT_CONNECTED** 事件到来时通过 pvData:: ui32Instance 参数来给到应用程序。如果 ui32Interface 被设置成了 0xFFFFFFFF，那么协议栈将设备描述符中第一个实体作为参数。

**返回值：**

对应于请求接口的 USB 类。

#### 3. 2. 2. 5 usbhcd_dev_hub_port()

此函数返回对应于请求设备实体的 USB HUB 口号。

**参数定义：**

**ui32Instance**：查询设备的唯一值。

**简介：**

本函数返回 ui32Instance 参数指向的设备的 USB HUB 口。ui32Instance 的值在核心层的事件驱动中在 **USB_EVENT_CONNECTED** 事件到来时通过 pvData:: ui32Instance 参数来给到应用程序。

**返回值：**

对应于请求接口的 USB HUB 口。

#### 3. 2. 2. 6 usbhcd_dev_protocol()

返回对应于设备实体的 USB 协议。

**参数定义：**

**ui32Instance**：查询设备的唯一值。

**ui32Interface**：连接 USB 类的接口号。

**简介：**

本函数返回 ui32Instance 参数指向的设备的 USB 协议。ui32Instance 的值在核心层的事件驱动中在**USB_EVENT_CONNECTED** 事件到来时通过 pvData:: ui32Instance 参数来给到应用程序。如果 ui32Interface 被设置成了 0xFFFFFFFF，那么协议栈将设备描述符中第一个实体作为参数。

**返回值：**

请求接口的 USB 协议。

#### 3. 2. 2. 7 usbhcd_dev_sub_class()

此函数返回对应于请求设备实体的子类。

**参数定义：**

**ui32Instance**：查询设备的唯一值。

**ui32Interface**：连接 USB 类的接口号。

**简介：**

本函数返回 ui32Instance 参数指向的设备的 USB 子类。ui32Instance 的值在核心层的事件驱动中在**USB_EVENT_CONNECTED** 事件到来时通过 pvData:: ui32Instance 参数来给到应用程序。如果 ui32Interface 被设置成了 0xFFFFFFFF，那么协议栈将设备描述符中第一个实体作为参数。

**返回值：**

对应于请求接口的 USB 子类。

#### 3. 2. 2. 8 usbhcd_event_disable()

此函数将关闭特定的 USB 事件通知。

**参数定义：**

**ui32Index**：USB 控制器索引。

**pvEventDriver**：指向通过 usbhcd_register_drivers()函数注册的事件驱动的结构体。

**ui32NumDrivers**：将被关闭通知的事件。

**简介：**

此函数用来关闭特定的事件通知，调用此函数后不会在应用程序中接收到此事件。

**返回值：**

成功关闭相关事件通知返回一个非 0 的值，失败则返回 0。

#### 3. 2. 2. 9 usbhcd_event_enable()

此函数将打开特定的 USB 事件通知。

**参数定义：**

**ui32Index**：USB 控制器索引。

**pvEventDriver**：指向通过 usbhcd_register_drivers()函数注册的事件驱动的结构体。

**ui32NumDrivers**：将被打开通知的事件。

**简介：**

此函数用来打开特定的事件通知，调用此函数后会在应用程序中接收到此事件。注意不是所有的事件通知都可以被打开。

**返回值：**

成功打开相关事件通知返回一个非 0 的值，失败则返回 0。

#### 3. 2. 2. 10 usbhcd_feature_set()

此函数返回对应于请求设备实体的子类。

**参数定义：**

**ui32Index**：USB 控制器索引。

**ui32Feature**：以 **USBLIB_FEATURE_**格式的的特点宏定义。

**pvFeature**：特点值。

**简介：**

注意不是所以的特点在所有的 ES32 系列中都支持，调用此函数时需要确定本器件是否支持此特点。

**返回值：**

返回 true 表示特点被成功设置。

#### 3. 2. 2. 11 usbhcd_init()

USB 主机控制器初始化。

**参数定义：**

**ui32Index**：USB 控制器索引。

**pvPool**：USB 主机协议栈内存池指针。

**ui32PoolSize**：USB 主机协议栈内存池大小。

**简介：**

这个函数将执行所有必要的操作,以允许 USB 主机控制器开始枚举和与设备通信。这个函数通常应该在应用程序开始时调用,所有设备和类驱动程序都准备好进行正常操

作。这个调用将启动 USB 主机控制器。

USB 协议栈的工作模式可以通过 usb_stack_mode_set()函数来设置。

传递给这个函数的内存池必须至少与支持的设备的典型配置描述符一样大。这个值是应用程序依赖的,但它永远不应该小于 32 字节,在大多数情况下,应该至少 64 个字节。如果没有足够的内存从设备中加载配置描述符,那么该设备将不会被 USB 库的主机控制器驱动程序识别。

**返回值：**

无。

#### 3. 2. 2. 12 usbhcdlpm_resume()

产生 LPM 请求。

**参数定义：**

**ui32Index**：USB 控制器索引。

**简介：**

这个函数将在 USB 总线上产生 LPM 恢复信号。

**返回值：**

无。

#### 3. 2. 2. 13 usbhcdlpm_sleep()

产生 LPM 请求使设备进入睡眠状态。

**参数定义：**

**psDevice**：USB 设备查询码。

**简介：**

这个函数将发送 LPM 信号使设备进入低功耗模式。

**返回值：**

**USBHCD_LPM_AVAIL**-请求被成功发送。

#### 3. 2. 2. 14 usbhcd_main()

USB 协议栈主机事务处理函数。

**简介：**

此函数必须定期地在 USB 协议栈回调函数以外调用，协议栈中所有的非阻塞操作是在中断中处理，所有的阻塞操作便在此函数中处理。

**返回值：**

**无。**

#### 3. 2. 2. 15 usbhcd_pipe_alloc()

分配一个 USB 主机控制器管道，管道的缓存固定为 64 bytes。

**参数定义：**

**ui32Index**：USB 控制器索引。

**ui32EndpointType**：端点的类型。

**psDevice**：设备的实体。

**pfnCallback**：回调函数，当此管道上有事件发生时此函数将被调用。

**简介：**

由于在主机控制器中可以使用有限数量的 USB HCD 管道，所以该函数用于临时或永久地获得其中一个端点。它还提供了一个方法来注册这个端点的状态更改的回调。如果不需要回调,pfnCallback 函数应该设置为 0。当使用 usbhcd_pipe_schedule()函数时,应该使用回调,以便在操作完成时通知调用者。

**返回值：**

这个函数返回一个值指示哪个管道被分配。如果值为 0，则目前没有可用的管道。返回值应该作为所有管道 API 的输入参数。

#### 3. 2. 2. 16 usbhcd_pipe_alloc_size()

分配一个 USB 主机控制器管道，管道的大小可设置。

**参数定义：**

**ui32Index**：USB 控制器索引。

**ui32EndpointType**：端点的类型。

**psDevice**：设备的实体。

**ui32FIFOSize**：管道缓存大小。

**pfnCallback**：回调函数，当此管道上有事件发生时此函数将被调用。

**简介：**

由于在主机控制器中可以使用有限数量的 USB HCD 管道，所以该函数用于临时或永久地获得其中一个端点。与 usbhcd_pipe_alloc()函数不同的是，这个函数允许调用方指定在 ui32 大小参数中分配给这个端点的 FIFO 的大小。这个函数还提供了一个方法来注册这个端点的状态更改的回调。如果不需要回调，pfnCallback 函数应该设置为 0。当使用 usbhcd_pipe_schedule()函数时，应该使用回调，以便在操作完成时通知调用者。

**返回值：**

这个函数返回一个值指示哪个管道被分配。如果值为 0,则目前没有可用的管道。返回值应该作为所有管道 API 的输入参数。

#### 3. 2. 2. 17 usbhcd_pipe_config()

配置指定的 USB 主机控制器的管道。

**参数定义：**

**ui32Pipe**：需要配置的管道号。

**ui32MaxPayload**：管道传输的最大数据量。

**ui32Interval**：每帧数据传输的轮询间隔。

**ui32TargetEndpoint**：传输的目标端点。

**简介：**

在调用 usbhcd_pipe_alloc()配置 USB 管道后，此函数应该被调用。它用于设置与端点相关的配置，比如目标端点的 ui32MaxPayload。ui32maxload 参数通常从设备端点描述符直接读取，并以字节表示。

设置 ui32Interval 取决于配置的端点类型，对于不需要使用 ui32Interval 的端点应该设置为 0。对于 BULK 端点，ui32Interval 是一个从 2-16 的值， NAK 超时值为 2 ^(ui32Interval - 1)帧。对于中断端点 ui32Interval 是一个值从 1-255 的值，用于端点轮询的帧数最大计数。对于同步端点 ui32Interval 范围为 1-16，代表 2 ^(ui32 interval1)帧的轮询间隔。

**返回值：**

成功返回 0，失败返回其它值。

#### 3. 2. 2. 18 usbhcd_pipe_data_ack()

在 IN 管道中向设备指示数据被接收。

**参数定义：**

**ui32Pipe**：IN 管道所对应的管道号。

**简介：**

该函数用于 IN 管道中断传输的数据接收。中断 IN 端点的传输是通过调用 usbhcd_pipe_schedule()函数来实现的，并通过 USB_EVENT_RX_AVAILABLE 事件在获取数据时通知应用程序，在此事件的处理程序中，应用程序必须调用 usbhcd_pipe_data_ack()函数，使 USB 控制器从设备中提取数据并完成事务。

**返回值：**

无。

#### 3. 2. 2. 19 usbhcd_pipe_free()

释放分配的管道。

**参数定义：**

**ui32Pipe**：需要释放的管道所对应的管道号。

**简介：**

这个函数用于释放一个先前调用 usbhcd_pipe_alloc()函数分配的 USB 管道，以用于系统中其他设备端点的使用。释放一个未分配的或无效的管道不会产生错误，而是简单地返回。

**返回值：**

无。

#### 3. 2. 2. 20 usbhcd_pipe_read()

从 USB 管道中读取数据。

**参数定义：**

**ui32Pipe**：管道号。

**pui8Data**：保存读取数据的指针。

**ui32Size**：pui8Data 指示的缓存区大小。

**简介：**

当此函数从 USB 管道中读取的数据时这个函数将阻塞。调用者必须用 usbhcd_pipe_alloc()函数调用来注册一个回调，以便在收到数据时被告知。如果调用者提供了非零的 pui8Data 指针，那么数据将在回调发生之前复制到缓冲区中。如果调用者将 pui8Data 参数设置为 0，那么调用者将负责在 **USB_EVENT_RX_AVAILABLE** 的回调事件发生时从 FIFO 中读取数据。如果 USB 管道的数据比被请求的要少，这个函数返回的值可能小于请求的 ui32Size 大小。

**返回值：**

返回 pui8Data 指示的内存中得到的数据大小。

#### 3. 2. 2. 21 usbhcd_pipe_read_non_blocking()

从 USB 管道中读取数据。

**参数定义：**

**ui32Pipe**：管道号。

**pui8Data**：保存读取数据的指针。

**ui32Size**：pui8Data 指示的缓存区大小。

**简介：**

当此函数从 USB 管道中读取的数据时这个函数将阻塞。调用者必须用 usbhcd_pipe_alloc()函数调用来注册一个回调，以便在收到数据时被告知。如果 USB 管道的数据比被请求的要少，这个函数返回的值可能小于请求的 ui32Size 大小。

**返回值：**

返回 pui8Data 指示的内存中得到的数据大小。

#### 3. 2. 2. 22 usbhcd_pipe_schedule()

此函数用于调度管道的传输。

**参数定义：**

**ui32Pipe**：管道号。

**pui8Data**：保存读取数据的指针。

**ui32Size**：pui8Data 指示的缓存区大小。

**简介：**

这个函数不会因为通过的管道类型而阻塞,要么将数据发送到设备，要么将数据从设备中读取。在这两种情况下，数据的数量将被限制在给定端点的 FIFO 中。

**返回值：**

发送数据时返回发送的数据大小，接收数据时返回 0。

#### 3. 2. 2. 23 usbhcd_pipe_status()

此函数用于查询管道的状态。

**参数定义：**

**ui32Pipe**：管道号。

**简介：**

这个函数将返回给定 USB 管道的当前状态。如果没有报告这个调用的状态,则只需返回 **USBHCD_PIPE_NO_CHANGE**。

**返回值：**

返回 **USBHCD_PIPE_**格式的状态值。

#### 3. 2. 2. 24 usbhcd_pipe_transfer_size_get()

此函数用于查询管道的当前数据量。

**参数定义：**

**ui32Pipe**：管道号。

**简介：**

这个调用返回当前或最后一个字节计数，使用 ui32Pipe 参数指定的管道进行传输。这通常用于确定 **USB_EVENT_RX_AVAILABLE** 发生时所收到的实际字节数。

**返回值：**

成功则返回管道传输的数据大小。

#### 3. 2. 2. 25 usbhcd_pipe_write()

此函数用于写数据到 USB 控制器管道。

**参数定义：**

**ui32Pipe**：管道号。

**pui8Data**：发送数据的指针。

**ui32Size**：pui8Data 指示的缓存区大小。

**简介：**

这个函数将阻塞直到它发送尽可能多的数据。调用者应该通过 usbhcd_pipe_alloc()调用来注册一个回调，以便在数据传输时被告知。如果 USB 管道的空间比请求要少，这个函数返回的值可以小于 ui32Size。

**返回值：**

发送数据时返回发送的数据大小。

#### 3. 2. 2. 26 usbhcd_power_automatic()

此函数返回当前电源设置是否配置为自动处理使能和除能 VBUS。

**参数定义：**

**ui32Index**：USB 控制器索引。

**简介：**

是否当前的电源控制引脚配置会自动应用。

**返回值：**

非零值代表电源自动适应。

#### 3. 2. 2. 27 usbhcd_power_config_get()

获取 VBUS 配置。

**参数定义：**

**ui32Index**：USB 控制器索引。

**简介：**

获取 VBUS 的通过 usbhcd_power_config_init()函数的配置。

**返回值：**

VBUS 相关配置。

#### 3. 2. 2. 28 usbhcd_power_config_init()

配置 VBUS。

**参数定义：**

**ui32Index**：USB 控制器索引。

**ui32PwrConfig**：配置选项。

**简介：**

此函数必须在调用 usbhcd_init()之前被调用。相关的配置选项可以是：

- **USBHCD_FAULT_LOW**
- **USBHCD_FAULT_HIGH**
- **USBHCD_FAULT_VBUS_NONE**
- **USBHCD_FAULT_VBUS_TRI**
- **USBHCD_FAULT_VBUS_DIS**
- **USBHCD_VBUS_MANUAL**
- **USBHCD_VBUS_AUTO_LOW**
- **USBHCD_VBUS_AUTO_HIGH**

**返回值：**

无。

#### 3. 2. 2. 29 usbhcd_power_config_set()

重新配置 VBUS 引脚。

**参数定义：**

**ui32Index**：USB 控制器索引。

**ui32PwrConfig**：配置选项。

**简介：**

此函数用于重新配置 VBUS 引脚，相关配置项和 usbhcd_power_config_init()函数中配置项一致。

**返回值：**

返回 0 代表电源配置已经生效。

#### 3. 2. 2. 30 usbhcd_register_drivers()

初始化 USB 主机控制器类驱动列表，注册类驱动。

**参数定义：**

**ui32Index**：USB 控制器索引。

**ppsHClassDrvrs**：指向主机类驱动列表。

**ui32NumDrivers**：主机驱动包含的驱动个数。

**简介：**

这个函数将设置由 ui32Index 参数指定的主机控制器支持的主机类。在启用 usbhcd_init()函数之前，应该调用该函数。

**返回值：**

无。

#### 3. 2. 2. 31 usbhcd_reset()

产生 USB 复位信号。

**参数定义：**

**ui32Index**：USB 控制器索引。

**简介：**

这个函数处理在 USB 总线上发送重置信号。从这个函数返回后，USB 总线上的任何设备都应该返回到它的复位状态。

**返回值：**

无。

#### 3. 2. 2. 32 usbhcd_resume()

产生 USB 恢复信号。

**参数定义：**

**ui32Index**：USB 控制器索引。

**简介：**

这个函数处理在 USB 总线上发送恢复信号。此函数仅仅用于先前调用了 usbhcd_suspend()函数处于挂起状态的主机控制器。

**返回值：**

无。

#### 3. 2. 2. 33 usbhcd_suspend()

产生 USB 挂起信号。

**参数定义：**

**ui32Index**：USB 控制器索引。

**简介：**

这个函数处理在 USB 总线上发送挂起信号。

**返回值：**

无。

#### 3. 2. 2. 34 usbhcd_term()

复位 USB 主机控制器。

**参数定义：**

**ui32Index**：USB 控制器索引。

**简介：**

该函数用于关闭或切换到 USB 设备模式，将清理 USB 主机控制器并禁用它。调用此函数后可以调用 usbhcd_init()来重新初始化控制器。

**返回值：**

无。

#### 3. 2. 2. 35 usbhcd_set_interface()

配置 USB 主机控制器。

**参数定义：**

**ui32Index**：USB 控制器索引。

**ui32Device**：指定的 USB 设备。

**ui32Interface**：接口号。

**ui32AltSetting**：替换 ui32Interface 的接口号。

**简介：**

该函数用于更改 USB 设备上有效接口的另一个设置。ui32Device 指定在设备连接时返回的设备实例。这个调用将根据 ui32Interface 和 ui32AltSetting 设置 USB 设备的接口。

**返回值：**

无。

#### 3. 2. 2. 36 usbhcd_string_descriptor_get()

配置 USB 主机控制器。

**参数定义：**

**psDevice**：所要读取字符串描述符的设备。

**pui8Buffer**：存储描述符的内存。

**ui32Size**：pui8Buffer 指定的内存大小。

**ui32LangID**：语言描述符 ID。

**ui32StringIndex**：字符串描述符索引。

**简介：**

这个函数将从 ui32StringIndex 参数指定的类型的设备请求一个字符串描述符。pui8Buffer 指针是存储请求结果的位置。应该通过 ui32Size 来表示 pui8Buffer 缓冲区的大小。ui32DevAddress 参数用于指定与 USB 总线通信的设备地址。

**返回值：**

返回 pui8Buffer 中返回的数据大小。

## 3.3 主机类层驱动

主机类驱动程序提供使用通用USB类的设备的访问接口。USB库目前支持以下两个USB类驱动程序:大容量存储类(MSC)和人类接口设备(HID)。为了能够使用这些类驱动程序，应用程序必须调用 usbhcd_register_drivers() 函数来注册主机类驱动程序的列表。**g_USBHIDClassDriver** 结构定义了主机 HID 类驱动程序的接口，**g_USBHostMSCClassDriver** 结构定义了 Host MSC 类驱动程序的接口。

主机类驱动程序在它的底层提供接口给到 USB 主机控制器核心驱动程序，而主机设备层的接口在它的顶层。在所有 USB 主机类驱动程序中，USB 主机控制器核心接口的下层接口与所有 USB 主机类驱动程序相同，而在给定类的所有 USB 主机接口中，顶部的设备接口层是常见的。因此 MSC 类驱动程序的顶层不需要与 HID 类驱动程序的顶层相匹配，但是对于底层两者都必须是相同的。除了枚举之外，所有与主机类驱动程序的通信都通过端点管道。主机类驱动程序解析并通过调用 usbhcd_pipe_alloc()和 usbhcd_pipe_config()函数来分配它所需要的端点管道。这些 USB 管道提供了从 USB 主机控制器核心驱动层读取/写入和获取回调通知的方法。

### 3.3.1 USB事件驱动

USB 主机库通过使用 USB 事件驱动程序在应用程序中接收非设备类特定事件。事件驱动程序允许应用程序接收到以下事件：

- **USB_EVENT_CONNECTED**（指示一个支持的设备连接）
- **USB_EVENT_UNKNOWN_CONNECTED**（指示一个不支持的设备连接到主机）
- **USB_EVENT_DISCONNECTED**（指示设备断开连接）
- **USB_EVENT_POWER_FAULT**（指示供电问题）
- **USB_EVENT_POWER_ENABLE**（指示供电应该被使能）
- **USB_EVENT_POWER_DISABLE**（指示供电应该被关闭）
- **USB_EVENT_SOF**（指示 SOF 事件发生）

USB 主机库提供了通过调用 usbhcd_event_enable ()或 usbhcd_event_disable()函数来启用或禁用这些事件的能力。所有的事件除了 **USB_EVENT_SOF** 都是在 USB 主机库初始化时默认启用的。默认情况下 **USB_EVENT_SOF** 被禁用，以避免过度开销，因为这个事件每毫秒发生一次。

下面的代码示例显示了一个 USB 库事件驱动程序回调函数的基本实现。

```
//
```

```
// Declare the driver.
//
DECLARE_EVENT_DRIVER(g_sEventDriver, 0, 0, USBHCDEvents)
void
USBHCDEvents(void *pvData)
{
  tEventInfo *psEventInfo;
  //
  // Cast this pointer to its actual type.
  //
  psEventInfo = (tEventInfo *)pvData;
  switch(psEventInfo->ui32Event)
  {
    //
    // Unknown device connected.
    //
    case USB_EVENT_CONNECTED:
    {
      ...
      break;
    }
    //
    // Unknown device disconnected.
    //
    case USB_EVENT_DISCONNECTED:
    {
      ...
      break;
    }
    //
    // Power Fault detected.
    //
    case USB_EVENT_POWER_FAULT:
    {
      ...
      break;
    }
    default:
    {
      break;
    }
  }
}
```

### 3. 3. 2 HID类主机驱动

HID 类驱动程序通过将 HID 设备的细节处理放在 HID 类驱动器之上的层中，可以对任何类型的 HID 类进行访问。HID 类驱动程序的顶层提供了打开或关闭设备实例的公共功能，读取设备的报告描述符，以便它可以由 HID 设备层代码解析，并可以在 HID 设备层上设置报告。在 **g_USBHIDClassDriver** 结构中指定连接到主机控制器核心驱动程序的底层接口。该结构用于注册 HID 类驱动程序与主机类驱动程序，以便在连接和枚举设备时调用它。**g_USBHIDClassDriver** 结构中的函数不应该被应用程序调用。

在下面的例子中，通用的 HID 类驱动程序在 USB 主机控制器核心驱动程序中注册，然后调用一个鼠标类设备的实例。通常 usbhhid_open()是由设备层接口调用，而 usbhcd_register_drivers()是由应用程序调用的。例如 usbhhid_open()用于 USB 库提供的鼠标设备主机驱动，是在 usbh_mouse_open()函数中被调用的，它是 USB 鼠标接口的一部分。

在 HID 类驱动程序的顶层，驱动程序有一个设备驱动接口，用于各种具体 HID 设备使用。为了让 HID 类驱动程序识别一个设备，设备层负责调用 usbhhid_open()，此调用指定设备类型和该设备的回调，以便与该设备类型相关的任何事件都可以返回给设备驱动程序。定义的类是在 tHIDSubClassProtocol 中定义的值，并通过 eDeviceType 参数传入 usbhhid_open()调用。为了释放一个 HID 类驱动程序的实例，HID 设备层或应用程序必须调用 usbhhid_close()函数，以允许一个新的或不同类型的设备连接。在 USB 库中提供的示例中，报告描述符将被检索，但不会被使用，因为实例依赖于 USB 键盘和鼠标的“boot”模式来解析报告描述符的格式。这是通过使用 usbhhid_set_report()接口来完成的，以将该设备引入其引导协议模式。由于在其他类型的应用程序或设备中可以限制或不能使用，usbhhid_get_report_descriptor()提供了一个通用的 HID 设备的能力来查询该设备的报告描述符(s)。还有两个接口 usbhhid_set_report()和 usbhhid_get_report()提供对 HID 报告的访问。

```
const tUSBHostClassDriver * const g_ppsUSBHostClassDrivers[] =
{
   &g_USBHIDClassDriver
};
//
// Register the host class drivers.
//
usbhcd_register_drivers(0, g_ppsUSBHostClassDrivers, 1);
...
//
// Open an instance of a HID mouse class driver.
//
psMouseInstance = usbhhid_open(USBH_HID_DEV_MOUSE,
USBHMouseCallback,
(void *)&g_sUSBHMouse);
```

一旦设备被打开，它接收的第一个回调是 **USB_EVENT_CONNECTED** 事件，它表明传入 usbhhid_open()的一个 HID 设备已经连接，USB 协议栈核心驱动程序已经完成了该设备的枚举。当设备被删除时，就会发生一个 **USB_EVENT_DISCONNECTED** 事件。当

关闭或打开设备时，应用程序应该调用 usbhhid_close()来禁用回调。这实际上并没有控制设备，但它阻止驱动程序调用应用程序。在正常运行期间，主机类驱动程序接收 **USB_EVENT_SCHEDULER** 和 **USB_EVENT_RX_AVAILABLE** 事件。**USB_EVENT_SCHEDULER** 表明如果设备已经准备好了，HID 类驱动程序应该调度一个新的请求。通过调用 usbhcd_pipe_schedule()来在给定的 IN 中断管道时产生一个新的 IN 请求。当 **USB_EVENT_RX_AVAILABLE** 发生表明中断管道产生新的数据可读。**USB_EVENT_RX_AVAILABLE** 在设备层接口上调用，此事件允许通过调用 usbhhid_get_report()请求数据并由设备层驱动程序来解析返回的报表结构中的数据。在某些情况下，比如键盘设备也可能需要调用主机类驱动程序来发布一个集合报告，将数据发送到设备。这是通过调用主机类驱动程序的 usbhhid_set_report()接口完成的。

### 3. 3. 3 MSC类主机驱动

大容量存储主机类驱动程序提供支持大容量存储类协议的设备。最常见的是 USB 闪存驱动器。这个主机类驱动程序提供了一个简单的基于块的接口，它可以与应用程序的文件系统相匹配。一个大容量存储设备的 USB 主机类驱动被包括在 USB 库中，它提供了一个简单的基于块的接口，它可以与应用程序的文件系统一起使用，因为它为基于逻辑块地址的大容量存储设备提供直接的块接口。

MSC 主机类驱动程序为访问 USB 闪存驱动器提供了一个应用程序 API。所提供的 API 需要与基于块的读取/写入的文件系统相匹配。usbhmsc_block_read() 和 usbhmsc_block_write()函数提供块读取和块写设备访问。这些函数根据 flash 驱动器指定的大小执行块操作。由于一些 flash 驱动器在准备好访问驱动器之前需要一些设置时间，所以 MSC 类驱动程序提供了 usbhmsc_drive_ready()函数来检查驱动器是否已经为正常操作做好了准备。

MSC 主机类驱动程序还为 USB 协议栈核心驱动程序提供一个接口，以完成 MSC 类设备的枚举。在全局结构 g_USBMSCClassDriver 中保存了 MSC 类驱动程序信息。这个结构只应该由应用程序使用，而这个结构中的函数指针通常不应该被除了主机核心层程序之外的任何程序直接调用。usbhmsc_open()和 usbhmsc_close()为主机控制器的枚举代码提供接口，当检测或删除 MSC 类设备时调用。MSC 主机类驱动程序提供一个回调到文件系统或应用程序来通知驱动器被删除或添加。为了使 MSC 类设备对主机核心层可见，应用程序必须在 usbhcd_register_drivers()函数调用中提供的驱动程序列表中添加它。类枚举常数设置为 **USB_CLASS_MASS_STORAGE**，所以任何枚举值为此值的设备都加载这个类驱动程序。

接下来将介绍应用程序或文件系统如何与 USB 协议栈提供的主机 MSC 类驱动程序交互。应用程序或文件系统必须注册 MSC 类驱动程序，通过向 usbhcd_register_driver()函数注册 g_USBHostMSCClassDriver 驱动。一旦注册了 MSC 类驱动程序，应用程序就必须调用 usbhmsc_open()，以便在一个新的 MSC 设备连接或断开或任何其他 MSC 类事件发生时驱动被调用。

```
const tUSBHostClassDriver * const g_ppsUSBHostClassDrivers[] =
{
  &g_sUSBHostMSCClassDriver
};
//
```

```
// Register the host class drivers.
//
usbhcd_register_drivers(0, g_ppsUSBHostClassDrivers, 1);
//
// Initialize the mass storage class driver on controller 0 with the
// MSCCallback() function as the callback for events.
//
usbhmsc_drive_open(0, MSCCallback);
```

第一个回调是一个 **USB_EVENT_CONNECTED** 事件，表明插入了 MSC 类闪存驱动器，USB 主机已经完成了该设备的枚举。但这并没有表明 flash 驱动器已经准备好读取/写操作，flash 驱动器是否准备好是需要检测出来的，应用程序应该调用 usbhmsc_drive_ready()函数来确定 flash 驱动器是否准备好读取/写入操作。当设备被删除时，就会发生一个 **USB_EVENT_DISCONNECTED** 事件，应用程序应该调用 usbhmsc_drive_close()来禁用回调。

一旦 usbhmsc_drive_ready()调用指示 flash 驱动器已经准备好，应用程序可以使用 usbhmsc_block_read()和 usbhmsc_block_write()函数来访问该设备，这些都是基于逻辑块的操作。值得注意的是，传递到这些函数的数据量是根据存储设备的块大小分配的，最常见的块大小是512字节。这些调用总是读取或写入一个完整块，因此必须适当分配空间。下面的示例显示了从 MSC 类设备中读取和写入块的调用。

```
//
// Read 1 block starting at logical block 0.
//
usbhmsc_block_read(ui32MSCDevice, 0, pui8Buffer, 1);
//
// Write 2 blocks starting at logical block 500.
//
usbhmsc_block_write(ui32MSCDevice, 500, pui8Buffer, 2);
```

由于大多数 MSC 类设备都遵循基于 block 的调用的 SCSI 协议，USB 协议栈为 MSC 类驱动程序提供了 SCSI 函数，以与 flash 驱动器进行通信。命令和数据通过主机控制器驱动程序提供的 USB 管道来传输。支持的唯一类型的 MSC 类设备是使用 SCSI 协议的设备。由于 flash 驱动器只支持 SCSI 协议的有限子集，所以只有 MSC 类所需的 SCSI 函数才能实现闪存驱动器。usbhscsi_read10()和 usbhscsi_write10()函数是用于读取和写入质量存储类设备的两个函数。剩余的 SCSI 函数被用来获取关于存储设备的信息，比如设备上的块的大小和块的数量。如果设备已经准备好了新的命令，则使用其他的接口用于错误处理或测试。

### 3. 3. 4 AUDIO类主机驱动

USB 音频主机类驱动程序提供支持 USB 音频类协议的设备的接口。应用程序通过调用 usb_host_audio_open()打开一个音频设备的实例，并提供一个回调函数来接收事件通知。当一个音频设备被枚举且为正常操作，USBH_AUDIO_EVENT_OPEN 事件会发向应用程序，当一个活跃的音频设备被断开，USBH_AUDIO_EVENT_CLOSE 事件会发向应用程序。当 usb_host_audio_open() 函数被调用后，应用程序在 USBH_AUDIO_EVENT_OPEN 事件到来之前或 USBH_AUDIO_EVENT_CLOSE 事件到

来后不可调用其它 AUDIO 的主机类驱动。当应用程序不再需要音频接口时，可以调用 usb_host_audio_close()来停止音频设备，不再接收音频设备的更改通知。通过调用 usb_host_audio_play()向主机音频驱动程序提供缓冲区，并为缓冲区提供回调函数，从而处理音频输出。缓冲区将在 usb_host_audio_play()中提供的回调函数中作为参数返回到应用程序。这允许应用程序在音频输出数据被调度时获得控制。AUDIO 的输入是通过调用 usb_host_audio_record()函数提供缓冲区以及回调函数并在回调函数中缓冲区作为参数来实现的。通过调用 usb_host_audio_record()函数向主机音频驱动程序提供缓冲区，并在回调中传递，从而处理音频输入。缓冲区将被 USB 控制器核心驱动层填充，并通过在 usb_host_audio_record()函数调用中提供的回调函数返回到应用程序。接下来将为每个应用程序级别 API 提供更多的细节和示例。

USB 主机音频应用接口提供了一种控制音频输出、输入和一些音量控制的接口。由于 USB 主机音频驱动内部只提供了少量的缓冲区，所以应用程序可以提供足够的缓冲区以保持音频流传输流畅。

USB 主机音频驱动程序需要一些初始配置。应用程序必须通过调用 usbhcd_register_drivers()函数来注册 USB 主机音频驱动程序。应用程序必须通过调用 usb_host_audio_open()函数来创建 USB 主机音频设备的实例，并为基本的 USB 音频事件提供回调。

```
//
// The instance data for the USB host audio driver.
//
uint32_t g_psAudioInstance = 0;

//
// The global that holds all of the host drivers in use in the application.
// In this case, only the host audio class is loaded.
//
static tUSBHostClassDriver const * const g_ppsHostClassDrivers[] =
{
  &g_sUSBHostAudioClassDriver,
  &g_sUSBEventDriver
};
...

//
// Register the host class drivers.
//
usbhcd_register_drivers(0, g_ppsHostClassDrivers, g_ui32NumHostClassDrivers);
//
// Open an instance of the mass storage class driver.
//
g_psAudioInstance = usb_host_audio_open(0, AudioCallback);
...
```

通过设置音频流的格式来处理音频输出，然后调用 usb_host_audio_play()函数为音频

设备提供新的缓冲区。为了启动音频输出，应用程序必须首先调用usb_host_audio_format_set()设置音频格式。如果该格式不受音频设备的支持，那么这个函数返回一个非零值，此时 usb_host_audio_play()函数在选择有效的格式之前不应该被调用。应用程序一旦设置了有效的格式，该应用程序应该通过调用 usb_host_audio_play()函数向主机音频驱动程序提供音频数据，然后一直等待回调指示缓冲区已经释放。在前面的缓冲区被释放之前调用 usb_host_audio_play()函数可以导致之前的传输数据被中断或取消。由于 USB 主机音频驱动程序提供了有限的缓冲，应用程序需要为音频数据提供更大的缓冲区。应用程序可以直接从回调函数中调用 usb_host_audio_play()函数为 USB 音频设备提供一个新的缓冲区。

```
void AudioOutCallback(void *pvBuffer, uint32_t ui32Param, uint32_t ui32Event)
{
  //
  // Check if this was a buffer free event and provide a new buffer to the
  // host audio driver.
  //
  if(ui32Event == USB_EVENT_TX_COMPLETE)
  {
    usb_host_audio_play(psAudioInstance, pNewBuffer, ui32Size,
    AudioOutCallback));
  }
}

void AudioPlay(void)
{
  //
  // Wait for USBH_AUDIO_EVENT_OPEN event.
  //
  ...
  //
  // Set the audio format to 48KHz 16 bit stereo output.
  //
  usb_host_audio_format_set(psAudioInstance, 48000, 16, 2,
  USBH_AUDIO_FORMAT_OUT);
  ...
  //
  // Start the output of the first buffer and let the callback start the
  // remaining buffers.
  //
  usb_host_audio_play(psAudioInstance, pBuffer, ui32Size, AudioOutCallback));
  //
  // Handle filling returned buffers.
  //
  ...
}
```

音频输入首先需要设置音频流的格式，然后通过调用 usb_host_audio_record()函数来提供一个音频数据接收缓冲区。当音频驱动程序有新数据时，协议栈通过回调函数向应用程序返回缓冲区。为了启动音频输入，应用程序必须首先设置音频输入格式，并调用 usb_host_audio_format_set()来设置数据格式。如果该格式不受音频设备的支持，那么这个函数返回一个非零值。usb_host_audio_record()不应该在选择有效的格式之前调用。应用程序一旦设置了有效的格式，应该通过调用 usb_host_audio_record()向主机音频驱动器提供一个音频缓冲区，并等待回调指示缓冲区已经填充。在前面的缓冲区被填充之前调用 usb_host_audio_record()函数可以导致之前的输入数据传输被中断或丢失。由于 USB 主机音频驱动程序提供了有限的缓冲区，应用程序需要提供合适的缓冲区。应用程序可以从回调函数中安全地调用 usb_host_audio_record()函数，以为 USB 音频设备提供一个新的缓冲区。

```
void AudioInCallback(tUSBHostAudioInstance *psAudioInstance, uint32_t ui32Event,
uint32_t ui32Param, void *pvMsgData)
{
  //
  // Check if this was a buffer full event and provide a new buffer to the
  // host audio driver.
  //
  if(ui32Event == USB_EVENT_RX_AVAILABLE)
  {
    usb_host_audio_record(psAudioInstance, pNewBuffer, ui32Size,
    AudioInCallback));
  }
}

void AudioRecord(void)
{
  //
  // Wait for USBH_AUDIO_EVENT_OPEN event.
  //
  ...
  //
  // Set the audio format to 48KHz 16 bit stereo output.
  //
  usb_host_audio_format_set(psAudioInstance, 48000, 16, 2,
  USBH_AUDIO_FORMAT_IN);
  ...
  //
  // Start the input of the first buffer and let the callback start the
  // remaining buffers.
  //
  usb_host_audio_record(psAudioInstance, pBuffer, ui32Size, AudioInCallback));
  //
  // Handle filling returned buffers.
```

```
    //
    ...
}
```

### 3. 3. 5 用户自定义主机驱动

本节将介绍如何实现自定义主机类驱动程序，以及主机控制器核心驱动程序如何找到驱动程序。所有主机类驱动程序必须为主机控制器核心驱动程序提供自己的驱动接口。与已有的 USB 协议栈主机类驱动程序一样，这意味着要向核心层驱动注册 tUSBClassDriver 的驱动程序接口。在下面的示例中，当主机核心层程序枚举一个与“USB_CLASS_SOMECLASS”接口类匹配的设备时 USBGenericOpen()函数被调用。当调用 USBGenericClose()函数时，这个类的设备被删除。下面的示例显示了自定义主机类驱动程序的定义。

```
tUSBClassDriver   sUSBGenericClassDriver =
{
    USB_CLASS_SOMECLASS,
    USBGenericOpen,
    USBGenericClose,
    USBGenericIntHandler
};
```

tUSBClassDriver 结构的 ulInterfaceClass 项是在枚举中从设备的接口描述符中读取的，此参数用于作为主机类驱动程序的主要搜索值。如果一个设备与这个结构成员匹配，那么主机类驱动程序就会加载。当检测到能够匹配接口类的设备时，协议栈会调用 tUSBClassDriver 结构的 pfnOpen 函数。这个函数应该做任何需要处理设备检测和初始配置设备的操作，这包括分配 USB 管道。此函数将解析设备端点的端点描述符，然后根据发现的端点类型和数量分配 USB 管道。这个调用不是中断级别的，因此可以被其他 USB 事件打断。任何必须在与其他该设备通信之前完成的东西都应该在 pfnOpen 函数中完成。pfnOpen 函数应该返回一个句柄，可以作为参数传递到函数 pfnClose 和 pfnIntHandler。这个句柄应该使主机类驱动程序区分相同类型的设备之间的不同实例。当使用 pfnOpen 调用创建的设备需要从系统中被删除时，可以将调用 pfnClose 结构成员。所有驱动程序都应该在 pfnClose 调用中被清理，因为不再向主机类驱动程序进行调用。如果主机类驱动程序需要对 USB 中断进行响应，则需要提供一个可选的 pfnIntHandler 函数指针。该函数在 USB 中断到来时运行，当然此函数不是强制要求的。

### 3. 3. 6 主机类驱动接口

**宏定义：**

- USBH_AUDIO_EVENT_CLOSE
- USBH_AUDIO_EVENT_OPEN
- USBH_EVENT_HID_KB_MOD
- USBH_EVENT_HID_KB_PRESS
- USBH_EVENT_HID_KB_REL
- USBH_EVENT_HID_MS_PRESS

- USBH_EVENT_HID_MS_REL
- USBH_EVENT_HID_MS_X
- USBH_EVENT_HID_MS_Y

**枚举：**

- tHIDSubClassProtocol

**函数：**

- void usbhhid_close(tHIDInstance *psHIDInstance)
- uint32_t usbhhid_get_report(tHIDInstance *psHIDInstance, uint32_t ui32Interface, uint8_t *pui8Data, uint32_t ui32Size)
- uint32_t usbhhid_get_report_descriptor(tHIDInstance *psHIDInstance, uint8_t *pui8Buffer, uint32_t ui32Size)
- tHIDInstance * usbhhid_open(tHIDSubClassProtocol iDeviceType, tUSBCallback pfnCallback, void *pvCBData)
- uint32_t usbhhid_set_idle(tHIDInstance *psHIDInstance, uint8_t ui8Duration, uint8_t ui8ReportID)
- uint32_t usbhhid_set_protocol(tHIDInstance *psHIDInstance, uint32_t ui32BootProtocol)
- uint32_t usbhhid_set_report(tHIDInstance *psHIDInstance, uint32_t ui32Interface, uint8_t *pui8Data, uint32_t ui32Size)
- int32_t usbhmsc_block_read(tUSBHMSCInstance *psMSCInstance, uint32_t ui32LBA, uint8_t *pui8Data, uint32_t ui32NumBlocks)
- int32_t usbhmsc_block_write(tUSBHMSCInstance *psMSCInstance, uint32_t ui32LBA, uint8_t *pui8Data, uint32_t ui32NumBlocks)
- void usbhmsc_drive_close(tUSBHMSCInstance *psMSCInstance)
- tUSBHMSCInstance * usbhmsc_drive_open(uint32_t ui32Drive, tUSBHMSCCallback pfnCallback)
- int32_t usbhmsc_drive_ready(tUSBHMSCInstance *psMSCInstance)
- void usb_host_audio_close(tUSBHostAudioInstance *psAudioInstance)
- uint32_t usb_host_audio_format_get(tUSBHostAudioInstance *psAudioInstance, uint32_t ui32SampleRate, uint32_t ui32Bits, uint32_t ui32Channels, uint32_t ui32Flags)
- uint32_t usb_host_audio_format_set(tUSBHostAudioInstance *psAudioInstance, uint32_t ui32SampleRate, uint32_t ui32Bits, uint32_t ui32Channels, uint32_t ui32Flags)

- tUSBHostAudioInstance * usb_host_audio_open(uint32_t ui32Index, tUSBHostAudioCallback pfnCallback)
- int32_t usb_host_audio_play(tUSBHostAudioInstance *psAudioInstance, void *pvBuffer, uint32_t ui32Size, tUSBHostAudioCallback pfnCallback)
- int32_t usb_host_audio_record(tUSBHostAudioInstance *psAudioInstance, void *pvBuffer, uint32_t ui32Size, tUSBHostAudioCallback pfnCallback)
- uint32_t usb_host_audio_volume_get(tUSBHostAudioInstance *psAudioInstance, uint32_t ui32Interface, uint32_t ui32Channel)
- uint32_t usb_host_audio_volume_max_get(tUSBHostAudioInstance *psAudioInstance, uint32_t ui32Interface, uint32_t ui32Channel)
- uint32_t usb_host_audio_volume_min_get(tUSBHostAudioInstance *psAudioInstance, uint32_t ui32Interface, uint32_t ui32Channel)
- uint32_t usb_host_audio_volume_res_get(tUSBHostAudioInstance *psAudioInstance, uint32_t ui32Interface, uint32_t ui32Channel)
- void usb_host_audio_volume_set(tUSBHostAudioInstance *psAudioInstance, uint32_t ui32Interface, uint32_t ui32Channel, uint32_t ui32Value)
- uint32_t usbhscsi_inquiry(uint32_t ui32InPipe, uint32_t ui32OutPipe, uint8_t *pui8Data, uint32_t *pui32Size)
- uint32_t usbhscsi_mode_sense6(uint32_t ui32InPipe, uint32_t ui32OutPipe, uint32_t ui32Flags, uint8_t *pui8Data, uint32_t *pui32Size)
- uint32_t usbhscsi_read10(uint32_t ui32InPipe, uint32_t ui32OutPipe, uint32_t ui32LBA, uint8_t *pui8Data, uint32_t *pui32Size, uint32_t ui32NumBlocks)
- uint32_t usbhscsi_read_capacities(uint32_t ui32InPipe, uint32_t ui32OutPipe, uint8_t *pui8Data, uint32_t *pui32Size)
- uint32_t usbhscsi_read_capacity(uint32_t ui32InPipe, uint32_t ui32OutPipe, uint8_t *pui8Data, uint32_t *pui32Size)
- uint32_t usbhscsi_request_sense(uint32_t ui32InPipe, uint32_t ui32OutPipe, uint8_t *pui8Data, uint32_t *pui32Size)
- uint32_t usbhscsi_test_unit_ready(uint32_t ui32InPipe, uint32_t ui32OutPipe)
- uint32_t usbhscsi_write10(uint32_t ui32InPipe, uint32_t ui32OutPipe, uint32_t ui32LBA,uint8_t *pui8Data, uint32_t *pui32Size, uint32_t ui32NumBlocks)

#### 3. 3. 6. 1 USBH_AUDIO_EVENT_CLOSE

USB 主机 AUDIO 事件，指示先前音频设备处于不激活状态。

#### 3. 3. 6. 2 USBH_AUDIO_EVENT_OPEN

USB 主机 AUDIO 事件，指示先前音频设备处于激活状态。

#### 3. 3. 6. 3 USBH_EVENT_HID_KB_MOD

USB 主机 HID 键盘事件，指示键盘辅助键被按下。

#### 3. 3. 6. 4 USBH_EVENT_HID_KB_PRESS

USB 主机 HID 键盘事件，指示键盘按键被按下。

#### 3. 3. 6. 5 USBH_EVENT_HID_KB_REL

USB 主机 HID 键盘事件，指示键盘按键被释放。

#### 3. 3. 6. 6 USBH_EVENT_HID_MS_PRESS

USB 主机 HID 鼠标事件，指示鼠标按键被按下。

#### 3. 3. 6. 7 USBH_EVENT_HID_MS_REL

USB 主机 HID 鼠标事件，指示鼠标按键被释放。

#### 3. 3. 6. 8 USBH_EVENT_HID_MS_X

USB 主机 HID 鼠标事件，指示鼠标 x 轴数据更新。

#### 3. 3. 6. 9 USBH_EVENT_HID_MS_Y

USB 主机 HID 鼠标事件，指示鼠标 y 轴数据更新。

#### 3. 3. 6. 10 tHIDSubClassProtocol

**参数定义：**

以下值将用来注册 USB HID 主机子类的回调：

- **eUSBHHIDClassNone**（无设备驱动使用）
- **eUSBHHIDClassKeyboard**（键盘设备）
- **eUSBHHIDClassMouse**（鼠标设备）
- **eUSBHHIDClassVendor**（用户自定义）

#### 3. 3. 6. 11 usbhhid_close()

释放一个 HID 设备实例。

**参数定义：**

**psHIDInstance**：指向 HID 实例。

**简介：**

此函数将释放由 usbhhid_open()函数创建的 HID 设备实例。

**返回值：**

无。

#### 3. 3. 6. 12 usbhhid_get_report()

从 HID 设备报告中提取一个报告。

**参数定义：**

**psHIDInstance**：调用 usbhhid_open()函数返回的设备实例。

**ui32Interface**：需要提取报告的接口。

**pui8Data**：存储报告的缓存区。

**ui32Size**：缓存区大小。

**简介：**

此函数将从管道中提取 USB 报告。当报告可以被提取时将产生 **USB_EVENT_RX_AVAILABLE** 事件。

**返回值：**

读取的报告的字节数。

#### 3. 3. 6. 13 usbhhid_get_report_descriptor()

提取 HID 报告描述符。

**参数定义：**

**psHIDInstance**：调用 usbhhid_open()函数返回的设备实例。

**pui8Buffer**：存储报告描述符的缓存区。

**ui32Size**：缓存区大小。

**简介：**

此函数用于提取 HID 设备的报告描述符。这个函数是阻塞类型的，将会返回读取到的数据的字节数。

**返回值：**

读取的报告描述符的字节数。

#### 3. 3. 6. 14 usbhhid_open()

打开一个 HID 实例。

**参数定义：**

**iDeviceType**：设备的类型。

**pfnCallback**：回调函数指针，当此设备状态有所改变时将会调用此回调。

**pvCBData**：回调函数参数。

**简介：**

此函数将创建一个特定类型的 HID 设备实例。iDeviceType 参数是 USB 协议栈中子类的值。

**返回值：**

该函数返回设备的实例值，该值应该作为接下来相关 API 的参数。如果返回 0 值，则表示无法创建设备实例。

#### 3. 3. 6. 15 usbhhid_set_idle()

设置 HID 设备的空闲时间。

**参数定义：**

**psHIDInstance**：调用 usbhhid_open()函数返回的设备实例。

**ui8Duration**：超时时间，以毫秒为单位。

**ui8ReportID**：报告的 ID。

**简介：**

该函数将 USB 设置空闲命令发送到 HID 设备，以设置给定报告的空闲超时。超时的长度是由 ui8Duration 参数指定的，报告的超时的 ID 是在 ui8ReportID 的值中。

**返回值：**

0。

#### 3. 3. 6. 16 usbhhid_set_protocol()

设置 HID 设备的协议。

**参数定义：**

**psHIDInstance**：调用 usbhhid_open()函数返回的设备实例。

**ui32BootProtocol**：指示设备的协议。

**简介：**

USB 主机设备可以使用这个函数来设置连接的 HID 设备的协议。这通常用于将键盘和鼠标设置为简化的“boot”协议模式，以修复报告结构到一个已知的状态。

**返回值：**

0。

#### 3. 3. 6. 17 usbhhid_set_report()

发送报告。

**参数定义：**

**psHIDInstance**：调用 usbhhid_open()函数返回的设备实例。

**ui32Interface**：发送报告的接口。

**pui8Data**：存储报告的缓存区指针。

**ui32Size**：缓存区大小。

**简介：**

该函数用于将报告发送到 USB HID 设备。它只能在回调以外调用。

**返回值：**

返回发送的数据字节数。

#### 3. 3. 6. 18 usbhmsc_block_read()

读取 MSC 设备块数据。

**参数定义：**

**psMSCInstance**：调用 usbhmsc_drive_open()函数返回的设备实例。

**ui32LBA**：逻辑块地址。

**pui8Data**：数据读取缓存区。

**ui32NumBlocks**：读取的块数量。

**简介：**

该函数将从与 psMSCInstance 参数相关的设备执行块大小的读取。ui32LBA 参数指定在设备上读取的逻辑块地址。这个函数只执行 ui32NumBlocks 块大小的读取。在大多数情况下，这是对 512 字节数据的读取。使用的 pui8Data 缓冲区至少应该是 ui32NumBlocks*512 字节的大小。

**返回值：**

返回 0 指示成功。

#### 3. 3. 6. 19 usbhmsc_block_write()

发送 MSC 设备块数据。

**参数定义：**

**psMSCInstance**：调用 usbhmsc_drive_open()函数返回的设备实例。

**ui32LBA**：逻辑块地址。

**pui8Data**：数据发送缓存区。

**ui32NumBlocks**：发送的块数量。

**简介：**

该函数将从与 psMSCInstance 参数相关的设备执行块大小的发送。ui32LBA 参数指定在设备上发送的逻辑块地址。这个函数只执行 ui32NumBlocks 块大小的发送。在大多数情况下，这是对 512 字节数据的发送。使用的 pui8Data 缓冲区至少应该是 ui32NumBlocks*512 字节的大小。

**返回值：**

返回 0 指示成功。

#### 3. 3. 6. 20 usbhmsc_drive_close()

释放一个 MSC 设备实例。

**参数定义：**

**psMSCInstance**：调用 usbhmsc_drive_open()函数返回的设备实例。

**返回值：**

无。

#### 3. 3. 6. 21 usbhmsc_drive_open()

打开一个 MSC 设备实例。

**参数定义：**

**ui32Drive**：驱动号。

**pfnCallback**：MSC 设备回调函数。

**简介：**

这个函数被调用来打开一个 MSC 设备的实例。在连接任何设备之前应该调用它，以允许其对驱动连接和断开连接进行适当的通知。ui32Drive 参数是系统中驱动器提供的索引。应用程序还应该提供 pfnCallback，以通知诸如设备枚举和设备删除之类的 MSC 相关事件。

**返回值：**

返回供后续 MSC 设备相关 API 作为参数的 MSC 实例。

#### 3. 3. 6. 22 usbhmsc_drive_ready()

检测 MSC 设备是否正常工作。

**参数定义：**

**psMSCInstance**：usbhmsc_drive_open()函数返回的设备实例。

**简介：**

这个函数将检查当前 MSC 设备是否可用。

**返回值：**

返回 0 表示设备准备好，其它值代表有错误发生。

#### 3. 3. 6. 23 usbhscsi_inquiry()

向设备发送 SCSI 查询指令。

**参数定义：**

**ui32InPipe**：使用此命令的 IN 管道。

**ui32OutPipe**：使用此命令的 OUT 管道。

**pui8Buffer**：返回结果的数据缓存。

**pui32Size**：缓存大小。

**简介：**

该函数应用于对一个 MSC 设备发出 SCSI 查询命令。为了允许多个设备，ui32InPipe 和 ui32OutPipe 参数指示该调用使用的 USB 管道。

**返回值：**

返回 SCSI 状态，可以是 **SCSI_CMD_STATUS_PASS** 或者 **SCSI_CMD_STATUS_FAIL**。

#### 3. 3. 6. 24 usbhscsi_mode_sense6()

向设备发送 SCSI“Mode Sense(6)”指令。

**参数定义：**

**ui32InPipe**：使用此命令的 IN 管道。

**ui32OutPipe**：使用此命令的 OUT 管道。

**ui32Flags**：定义要做的确切查询的标志的组合。

**pui8Buffer**：返回结果的数据缓存。

**pui32Size**：缓存大小。

**简介：**

该函数应用于向 MSC 设备发出 SCSI“Mode Sense(6)”命令。为了兼容多个设备，ui32InPipe 和 ui32OutPipe 参数指示使用该调用的 USB 管道。调用将最多返回在 pui32Size 参数中指示的字节数。

ui32Flags 参数可以是以下参数：

以下值之一需要被指定：

- **SCSI_MS_PC_CURRENT** 请求当前设置
- **SCSI_MS_PC_CHANGEABLE** 请求可变设置
- **SCSI_MS_PC_DEFAULT** 请求默认设置
- **SCSI_MS_PC_SAVED** 请求保存值

以下值之一被用来指定请求的页码：

- **SCSI_MS_PC_VENDOR** 制造商指定页码值
- **SCSI_MS_PC_DISCO** 连接或重连接页码值

- **SCSI_MS_PC_CONTROL** 控制页码值
- **SCSI_MS_PC_LUN LUN** 页码值
- **SCSI_MS_PC_PORT** 协议指定页码值
- **SCSI_MS_PC_POWER** 供电条件页码值
- **SCSI_MS_PC_INFORM** 信息异常页码值
- **SCSI_MS_PC_ALL** 请求该设备支持的所有页码值

最后一个可选的值：

- **SCSI_MS_DBD** 关闭返回块描述

**返回值：**

返回 SCSI 状态，可以是 **SCSI_CMD_STATUS_PASS** 或者 **SCSI_CMD_STATUS_FAIL**。

#### 3. 3. 6. 25 usbhscsi_read10()

向设备发送 SCSI“Read(10)”指令。

**参数定义：**

**ui32InPipe**：使用此命令的 IN 管道。

**ui32OutPipe**：使用此命令的 OUT 管道。

**ui32LBA**：逻辑块地址。

**pui8Data**：返回结果的数据缓存。

**pui32Size**：缓存大小。

**ui32NumBlocks**：数据块数量。

**简介：**

该函数应用于向 MSC 设备发出 SCSI“Read(10)”命令。

**返回值：**

返回 SCSI 状态，可以是 **SCSI_CMD_STATUS_PASS** 或者 **SCSI_CMD_STATUS_FAIL**。

#### 3. 3. 6. 26 usbhscsi_read_capacity()

向设备发送 SCSI 读取能力指令。

**参数定义：**

**ui32InPipe**：使用此命令的 IN 管道。

**ui32OutPipe**：使用此命令的 OUT 管道。

**pui8Data**：返回结果的数据缓存。

**pui32Size**：缓存大小。

**简介：**

该函数应用于向 MSC 设备发出 SCSI 读取能力命令。

**返回值：**

返回 SCSI 状态，可以是 **SCSI_CMD_STATUS_PASS** 或者 **SCSI_CMD_STATUS_FAIL**。

#### 3. 3. 6. 27 usbhscsi_read_capacities()

向设备发送 SCSI 读取能力指令。

**参数定义：**

**ui32InPipe**：使用此命令的 IN 管道。

**ui32OutPipe**：使用此命令的 OUT 管道。

**pui8Data**：返回结果的数据缓存。

**pui32Size**：缓存大小。

**简介：**

该函数应用于向 MSC 设备发出 SCSI 读取能力命令。pui8Data 至少需要 **SCSI_READ_CAPACITY_SZ** 指定的大小。

**返回值：**

返回 SCSI 状态，可以是 **SCSI_CMD_STATUS_PASS** 或者 **SCSI_CMD_STATUS_FAIL**。

#### 3. 3. 6. 28 usbhscsi_request_sense()

向设备发送 SCSI“Request Sense”指令。

**参数定义：**

**ui32InPipe**：使用此命令的 IN 管道。

**ui32OutPipe**：使用此命令的 OUT 管道。

**pui8Data**：返回结果的数据缓存。

**pui32Size**：缓存大小。

**简介：**

该函数应用于向 MSC 设备发出 SCSI“Request Sense”命令。

**返回值：**

返回 SCSI 状态，可以是 **SCSI_CMD_STATUS_PASS** 或者

**SCSI_CMD_STATUS_FAIL**。

#### 3. 3. 6. 29 usbhscsi_test_unit_ready()

向设备发送 SCSI“Test Unit Ready”指令。

**参数定义：**

**ui32InPipe**：使用此命令的 IN 管道。

**ui32OutPipe**：使用此命令的 OUT 管道。

**简介：**

该函数应用于向 MSC 设备发出 SCSI“Test Unit Ready”命令。

**返回值：**

返回 SCSI 状态，可以是 **SCSI_CMD_STATUS_PASS** 或者 **SCSI_CMD_STATUS_FAIL**。

#### 3. 3. 6. 30 usbhscsi_write10()

向设备发送 SCSI“Write(10)”指令。

**参数定义：**

**ui32InPipe**：使用此命令的 IN 管道。

**ui32OutPipe**：使用此命令的 OUT 管道。

**ui32LBA**：逻辑块地址。

**pui8Data**：数据缓存区。

**pui32Size**：缓存区大小。

**ui32NumBlocks**：块数量。

**简介：**

该函数应用于向 MSC 设备发出 SCSI“Write(10)”命令。

**返回值：**

返回 SCSI 状态，可以是 **SCSI_CMD_STATUS_PASS** 或者 **SCSI_CMD_STATUS_FAIL**。

#### 3. 3. 6. 31 usb_host_audio_close()

释放 AUDIO 设备实例。

**参数定义：**

**psAudioInstance**：AUDIO 设备实例。

**简介：**

该函数应将释放调用 usb_host_audio_open()函数产生的 AUDIO 设备实例。

**返回值：**

无。

#### 3. 3. 6. 32 usb_host_audio_format_get()

此函数检测音频的格式是否被当前设备支持。

**参数定义：**

**psAudioInstance**：AUDIO 设备实例。

**ui32SampleRate**：采样率。

**ui32Bits**：样本位数。

**ui32Channels**：通道数。

**ui32Flags**：指示哪种接口。

**简介：**

该函数用来确认音频的数据格式是否被音频设备支持。ui32Flags 参数可以是 **USBH_AUDIO_FORMAT_IN** 或者 **USBH_AUDIO_FORMAT_OUT**。

**返回值：**

返回 0 表示格式被支持，其它值则表示不支持。

#### 3. 3. 6. 33 usb_host_audio_format_set()

此函数设置音频数据格式。

**参数定义：**

**psAudioInstance**：AUDIO 设备实例。

**ui32SampleRate**：采样率。

**ui32Bits**：样本位数。

**ui32Channels**：通道数。

**ui32Flags**：指示哪种接口。

**简介：**

该函数用来设置输入或者输出地音频数据的格式。ui32Flags 参数可以是 **USBH_AUDIO_FORMAT_IN** 或者 **USBH_AUDIO_FORMAT_OUT**。此函数必须在 usb_host_audio_record()以及 usb_host_audio_play()函数调用前被调用。

**返回值：**

返回 0 表示格式被支持，其它值则表示不支持。

#### 3. 3. 6. 34 usb_host_audio_open()

打开 AUDIO 设备驱动。

**参数定义：**

**ui32Index**：AUDIO 设备索引，当前只有 0 号设备被支持。

**pfnCallback**：AUDIO 设备回调。

**简介：**

这个函数被调用打开一个主机音频设备的实例，应该为 pfnCallback 参数中的主机音频事件提供一个有效的回调函数。

**返回值：**

返回 AUDIO 设备实例。

#### 3. 3. 6. 35 usb_host_audio_play()

向 AUDIO 设备发送音频数据。

**参数定义：**

**psAudioInstance**：AUDIO 设备实例。

**pvBuffer**：音频数据缓冲区。

**ui32Size**：缓冲区大小。

**pfnCallback**：当缓冲区重新可用时，协议栈将调用该回调。

**简介：**

当应用程序需要输出数据到 USB 音频设备时，需要调用此函数。由于此调用安排了传输并立即返回，应用程序应该提供一个 pfnCallback 函数，当应用程序可以再次使用缓冲区时，该函数将被调用。ui32Event 参数是 **USB_EVENT_TX_COMPLETE**。

**返回值：**

返回被调度发送的数据量。

#### 3. 3. 6. 36 usb_host_audio_record()

向 AUDIO 设备提供输入数据缓冲区。

**参数定义：**

**psAudioInstance**：AUDIO 设备实例。

**pvBuffer**：音频数据缓冲区。

**ui32Size**：缓冲区大小。

**pfnCallback**：当缓冲区满时，协议栈将调用该回调。

**简介：**

此函数用于输入音频数据。由于此调用安排了传输并立即返回，应用程序应该提供一个 pfnCallback 函数，当应用程序可以缓冲区满时，该函数将被调用。ui32Event 参数是 **USB_EVENT_RX_AVAILABLE**。

**返回值：**

返回被调度接收的数据量。

#### 3. 3. 6. 37 usb_host_audio_volume_get()

该函数用于获得给定音频设备的当前音量设置。

**参数定义：**

**psAudioInstance**：AUDIO 设备实例。

**ui32Interface**：接口号。

**ui32Channel**：声道。

**简介：**

该函数用于检索 ui32Channel 指定的声道上的音频设备的当前音量设置。ui32Interface 被忽略，应该设置为 0，以访问默认的音频控制接口。ui32Channel 从 0 开始，这是主音频音量控制接口。其余的 ui32Channel 值可以访问其他各种音频通道，其中 1 和 2 是左和右音频通道。

**返回值：**

返回请求接口的音量设置。

#### 3. 3. 6. 38 usb_host_audio_volume_max_get()

该函数用于获得给定音频设备的最大音量设置。

**参数定义：**

**psAudioInstance**：AUDIO 设备实例。

**ui32Interface**：接口号。

**ui32Channel**：声道。

**简介：**

该函数用于检索 ui32Channel 指定的声道上的音频设备的最大音量设置。ui32Interface 被忽略，应该设置为 0，以访问默认的音频控制接口。ui32Channel 从 0 开始，这是主音频音量控制接口。其余的 ui32Channel 值可以访问其他各种音频通道，其中 1 和 2 是左和右音频通道。

**返回值：**

返回请求接口的最大音量设置。

#### 3. 3. 6. 39 usb_host_audio_volume_min_get()

该函数用于获得给定音频设备的最小音量设置。

**参数定义：**

**psAudioInstance**：AUDIO 设备实例。

**ui32Interface**：接口号。

**ui32Channel**：声道。

**简介：**

该函数用于检索 ui32Channel 指定的声道上的音频设备的最小音量设置。ui32Interface 被忽略，应该设置为 0，以访问默认的音频控制接口。ui32Channel 从 0 开始，这是主音频音量控制接口。其余的 ui32Channel 值可以访问其他各种音频通道，其中 1 和 2 是左和右音频通道。

**返回值：**

返回请求接口的最小音量设置。

#### 3. 3. 6. 40 usb_host_audio_volume_res_get()

该函数用于获得给定音频设备的音阶设置。

**参数定义：**

**psAudioInstance**：AUDIO 设备实例。

**ui32Interface**：接口号。

**ui32Channel**：声道。

**简介：**

该函数用于检索 ui32Channel 指定的声道上的音频设备的音阶设置。ui32Interface 被忽略，应该设置为 0，以访问默认的音频控制接口。ui32Channel 从 0 开始，这是主音频音阶控制接口。其余的 ui32Channel 值可以访问其他各种音频通道，其中 1 和 2 是左和右音频通道。

**返回值：**

返回请求接口的音阶设置。

#### 3. 3. 6. 41 usb_host_audio_volume_set()

该函数用于设置给定音频设备的音量。

**参数定义：**

**psAudioInstance**：AUDIO 设备实例。

**ui32Interface**：接口号。

**ui32Channel**：声道。

**简介：**

该函数用于检索 ui32Channel 指定的声道上的音频设备的音量大小。ui32Interface 被忽略，应该设置为 0，以访问默认的音频控制接口。ui32Channel 从 0 开始，这是主音频音量控制接口。其余的 ui32Channel 值可以访问其他各种音频通道，其中 1 和 2 是左和右音频通道。

**返回值：**

无。

## 3.4 主机层驱动

主机层接口提供了一个 HID 鼠标、HID 键盘、MSC 设备的实例。

### 3.4.1 鼠标设备

HID 鼠标设备接口主要通过一个回调函数控制，该函数作为打开鼠标设备接口调用的一部分。为了打开鼠标设备的一个实例，应用程序调用 usbh_mouse_open()，并传递回调函数，以及鼠标设备使用的一些缓冲数据。所提供的缓冲区由鼠标设备内部使用，不应被应用程序使用。一旦设备被打开，应用程序应该等待一个 **USB_EVENT_CONNECTED** 事件，以指示已经成功地检测到鼠标并被枚举。此时应用程序应该调用 usbh_mouse_init()函数来初始化连接的实际设备。在此之后，应用程序可以通过 usbh_mouse_open()调用中提供的回调启动接收以下事件：

- **USBH_EVENT_HID_MS_PRESS**
- **USBH_EVENT_HID_MS_REL**
- **USBH_EVENT_HID_MS_X**
- **USBH_EVENT_HID_MS_Y**

**USBH_EVENT_HID_MS_PRESS** 以及 **USBH_EVENT_HID_MS_REL** 事件中，ui32MsgParam 参数将携带 **HID_MOUSE_BUTTON_1**、**HID_MOUSE_BUTTON_2**、**HID_MOUSE_BUTTON_3** 信息。**USBH_EVENT_HID_MS_X** 事件中 ui32MsgParam 参数将携带 x 轴的位移数据。**USBH_EVENT_HID_MS_Y** 事件中 ui32MsgParam 参数将携带 y 轴的位移数据。

### 3.4.2 键盘设备

和鼠标一样，键盘设备接口主要是通过一个回调函数来控制的，这是一个用来打开键盘设备接口的调用的一部分。为了打开键盘设备的一个实例，应用程序调用 usbh_keyboard_open()，并传递回调函数，以及键盘设备使用的一些缓冲数据。提供的缓冲区由键盘设备内部使用，不应被应用程序使用。一旦设备打开，应用程序应该等待一个 **USB_EVENT_CONNECTED** 事件来指示一个键盘设备被成功连接并枚举。此时应用程序应该调用 usbh_keyboard_init()来初始化键盘设备。此后 usbh_keyboard_open()注册的回调函数可以接收到以下事件：

- **USBH_EVENT_HID_KB_PRESS**
- **USBH_EVENT_HID_KB_REL**

- **USBH_EVENT_HID_KB_MOD**

**USBH_EVENT_HID_KB_PRESS** 事件中，ui32MsgParam 参数为所要求的键提供了 USB 使用标识符。应用程序将此使用标识符映射到一个实际可打印的字符，使用 usbh_keyboard_usage_to_char()函数，或者它可以简单地响应按键，而输出键值。应该注意的是，像“Cap_Lock”键这样的特殊键，需要通知实际的键盘设备。**USBH_EVENT_HID_KB_REL** 事件中，ui32MsgParam 参数为所要求的键提供了 USB 使用标识符。**USBH_EVENT_HID_KB_MOD** 事件中，有以下值可能被置位：

- **HID_KEYB_LEFT_CTRL**
- **HID_KEYB_LEFT_SHIFT**
- **HID_KEYB_LEFT_ALT**
- **HID_KEYB_LEFT_GUI**
- **HID_KEYB_RIGHT_CTRL**
- **HID_KEYB_RIGHT_SHIFT**
- **HID_KEYB_RIGHT_ALT**
- **HID_KEYB_RIGHT_GUI**

### 3. 4. 3 主机驱动接口

**函数：**

- uint32_t usbh_keyboard_close(tUSBHKeyboard *psKbInstance)
- uint32_t usbh_keyboard_init(tUSBHKeyboard *psKbInstance)
- uint32_t usbh_keyboard_modifier_set(tUSBHKeyboard *psKbInstance, uint32_t ui32Modifiers)
- tUSBHKeyboard * usbh_keyboard_open(tUSBHIDKeyboardCallback pfnCallback, uint8_t *pui8Buffer, uint32_t ui32Size)
- uint32_t usbh_keyboard_poll_rate_set(tUSBHKeyboard *psKbInstance, uint32_t ui32PollRate)
- uint32_t usbh_keyboard_usage_to_char(tUSBHKeyboard *psKbInstance, const tHIDKeyboardUsageTable *psTable, uint8_t ui8UsageID)
- uint32_t usbh_mouse_close(tUSBHMouse *psMsInstance)
- uint32_t usbh_mouse_init(tUSBHMouse *psMsInstance)
- tUSBHMouse *usbh_mouse_open(tUSBHIDMouseCallback pfnCallback, uint8_t *pui8Buffer, uint32_t ui32Size)

#### 3. 4. 3. 1 usbh_keyboard_close()

释放一个 HID 键盘设备实例。

**参数定义：**

**psHIDInstance**：指向 HID 键盘设备实例。

**简介：**

此函数将释放由 usbh_keyboard_open()函数创建的 HID 键盘设备实例。

**返回值：**

返回 0 表示成功。

#### 3. 4. 3. 2 usbh_keyboard_init()

在键盘设备检测到之后初始化键盘设备。

**参数定义：**

**psHIDInstance**：指向 HID 键盘设备实例。

**简介：**

此函数需要在 usbh_keyboard_open()函数提供的回调函数中接收到 **USB_EVENT_CONNECTED** 事件之后被调用以初始化键盘设备。

**返回值：**

返回 0 表示成功。

#### 3. 4. 3. 3 usbh_keyboard_modifier_set()

设置一个辅助按键。

**参数定义：**

**psHIDInstance**：指向 HID 键盘设备实例。

**ui32Modifiers**：辅助按键掩码。

**简介：**

此函数用于向键盘设备发送辅助按键状态。ui32Modifiers 参数可以是以下掩码：

- **HID_KEYB_NUM_LOCK**
- **HID_KEYB_CAPS_LOCK**
- **HID_KEYB_SCROLL_LOCK**
- **HID_KEYB_COMPOSE**
- **HID_KEYB_KANA**

但并不是所有的键盘都能在所有键盘上得到支持。psKbInstance 值是调用 usbh_keyboard_open()时返回的值。如果使用 **HID_KEYB_CAPS_LOCK**，它将修改来自 usbh_keyboard_usage_to_char()函数返回的值。

**返回值：**

返回 0 表示成功。

#### 3. 4. 3. 4 usbh_keyboard_open()

打开一个键盘实例。

**参数定义：**

**pfnCallback**：回调函数，当和此设备有关事件发生时，此函数将被调用。

**pui8Buffer**：用于键盘设备使用的内存空间。

**ui32BufferSize**：pui8Buffer 内存空间大小。

**简介：**

这个函数用于打开键盘的一个实例。从这个函数返回的值应该作为其他键盘调用的参数。协议栈使用 pui8Buffer 内存缓冲区访问键盘，所需的缓冲区大小至少足以为该设备保存一个正常的报告描述符。如果没有足够的空间，只有一个部分报表描述符将被读出。

**返回值：**

返回键盘设备实例值。

#### 3. 4. 3. 5 usbh_keyboard_poll_rate_set()

设置键盘自动轮询的时间。

**参数定义：**

**psKbInstance**：键盘实例。

**ui32PollRate**：键盘状态主动更新率，以毫秒为单位。

**简介：**

这个函数将允许应用程序告诉键盘，不管键盘状态有任何变化，它应该经常向 USB 主机发送更新。psKbInstance 值是调用 usbh_keyboard_open()时返回的值。ui32PollRate 以毫秒为单位。这个值最初设置为 0，这表明键盘应该只在键盘状态变化时更新。

**返回值：**

返回 0 表示成功。

#### 3. 4. 3. 6 usbh_keyboard_usage_to_char()

将 USB 的键值转化成可打印的字符。

**参数定义：**

**psKbInstance**：键盘实例。

**psTable**：键盘页符对照表。

**ui8UsageID**：键盘按键的页码。

**简介：**

该函数用于将 USB 使用按键页码映射到一个字符。所提供的 psTable 用于执行映射，并由 thidkeyboarddusagetable 类型定义的结构描述。请参阅 thidkeyboarddusagetable 结构上的文档，以了解该结构内部的更多细节。这个函数使用 shift 键的当前状态和大写锁键来修改这个函数返回的数据。psTable 结构具有值指示哪些键被大写的大写和交替值修改为移位的情况。

**返回值：**

返回对应的字符值。

#### 3. 4. 3. 7 usbh_mouse_close()

释放一个 HID 鼠标设备实例。

**参数定义：**

**psMsInstance**：指向 HID 鼠标设备实例。

**简介：**

此函数将释放由 usbh_mouse_open()函数创建的 HID 鼠标设备实例。

**返回值：**

返回 0 表示成功。

#### 3. 4. 3. 8 usbh_mouse_init()

在鼠标设备检测到之后初始化鼠标设备。

**参数定义：**

**psMsInstance**：指向 HID 鼠标设备实例。

**简介：**

此函数需要在 usbh_mouse_open() 函数提供的回调函数中接收到 **USB_EVENT_CONNECTED** 事件之后被调用以初始化鼠标设备。

**返回值：**

返回 0 表示成功。

#### 3. 4. 3. 9 usbh_mouse_open()

打开一个鼠标实例。

**参数定义：**

**pfnCallback**：回调函数，当和此设备有关事件发生时，此函数将被调用。

**pui8Buffer**：用于鼠标设备使用的内存空间。

**ui32BufferSize**：pui8Buffer 内存空间大小。

**简介：**

这个函数用于打开鼠标的一个实例。从这个函数返回的值应该作为其他鼠标调用的参数。协议栈使用 pui8Buffer 内存缓冲区访问鼠标，所需的缓冲区大小至少足以为该设备保存一个正常的报告描述符。如果没有足够的空间，只有一个部分报表描述符将被读出。

**返回值：**

返回鼠标设备实例值。

# 第4章 通用部分

## 4. 1 USB Buffer

在低层级的 USB 协议栈传输中，USB 通信是基于包来传输的，每个数据包的大小依赖于 USB 端点的配置。当一个数据包在传输时，需要等到此包数据传输完成才能进行下一包数据的传输。

USB Buffer 模块适用于那些一个简单的读取/写入 API 就能实现任意大小的数据块接收或传输的应用程序。USB Buffer API 允许应用程序在与特定主机或设备类驱动程序连接时选择这种操作类型。

USB Buffer 为单个端点提供了一个单向缓冲区，并可以将操作配置为接收 Buffer(从 USB 协议栈接受数据，并将其传递给应用程序)或发送 Buffer (从应用程序接收数据，并将其传递给传输的 USB 协议栈)。在每一种情况下，Buffer 处理数据的所有打包和分包工作，并允许应用程序在适合的时候读取或写入任意大小的数据块(当然受 Buffer 内的空间限制)。

每个 USB Buffer 使用一个循环 Buffer 对象来存储缓冲数据。如果试图缓冲 USB 数据流，则应该使用 USB Buffer API，因为它代表应用程序处理 USB 驱动端交互。应用程序不能调用同一对象的两个 API 混合在一起，如果使用 USB 缓冲区，只需要使用表单 usb_buffer_xx() 的 API 来访问该缓冲区。

USB Buffer 对象的设计是为了允许在 USB 设备类驱动程序和设备应用程序之间或在 USB 主机核心驱动程序之间或在应用程序和类独立的主机类驱动程序之间方便数据的传输。在操作过程中，Buffer 下面的层的事件在 Buffer 的事件处理函数中被检查，如果这些事件没有被识别或对数据的传输没有影响，它们就会被传递到更高的层中，如果它们与数据传输有关，那么 Buffer 就会拦截它们，并在将适当的事件传递到上面的应用程序之前执行传输或接收数据的必要操作。

要在传输或接收通道或管道中插入 Buffer，就必须初始化一个 tUSBBuffer 对象。

**bTransmitBuffer** 如果 Buffer 将应用程序数据从应用程序传递到 USB 协议栈，此字段应该设为或 true，如果从 USB 协议栈传递到应用程序，这个字段必须被设置为 false。

**pfnCallback** 这个字段应该指向应用程序代码中的事件处理程序回调函数，与 Buffer 相关的异步事件通知将由调用该函数。

**pvCBData** 回调数据指针，将作为应用程序事件处理程序的所有未来调用的第一个参数(设置为 pfnCallback)。通常一个应用程序将这个指针设置为允许它容易访问与通道相关的数据的值，例如指向内部实例数据结构的指针。实际内容是应用程序指定的，USB Buffer 仅仅存储数据，并在需要时将其传递回调用方。

**pfnTransfer** 当数据在缓冲区和底层之间传输时，该字段通知该函数的 USB Buffer，如果这是一个发送 Buffer (**bTransmitBuffer** 设置为 true)，则用于发送数据或者接收一个接收缓冲区(**bTransmitBuffer** 设置为 false)，则用于发送数据包。以一个 BULK 类传输 Buffer 为例，用于将数据传输到 USB 通用的 BULK 设备类驱动程序，这将被设

置为 usbd_bulk_packet_write()。

**pfnAvailable** 对于发送 Buffer，该函数指针必须设置为指向较低层的函数，可以调用相关的 USB 端点或管道 API 判断是否准备接受一个新的传输包。对于接收缓冲区，该字段指向来确定读取新接收数据包所需的缓冲区的大小的函数。比如在 USB 通用的 BULK 设备类驱动程序上面的一个传输 Buffer，将会有这个字段以指向 usbd_bulk_tx_packet_available()，相反若为接收 Buffer，在相同的驱动程序将设置字段指向 usbd_bulk_rx_packet_available()。

**pvHandle** 该字段必须设置为具体 USB 驱动对象的句柄，是 pfnTransfer 和 pfnAvailable 所提供的函数的第一个参数。通常是对底层对象使用的实例结构的指针。在 USB 通用的 BULK 设备类的情况下，BULK 设备指针是最初传递给(并返回) usbd_bulk_init()的指针。

**pcBuffer** 这个字段必须被初始化，以指向将用于该通道上的 Buffer 数据的 RAM 块，Buffer 将被管理为循环 Buffer。

**ulBufferSize** 指示 **pcBuffer** 的字节数。

**pvWorkspace** USB Buffer 需要一个 RAM 块，它可以存储状态变量。这个字段指向应用程序提供的 RAM，可以供 Buffer 对象使用作为工作区。此 RAM 不能被应用程序访问，只要循环 Buffer 存在就必须在 USB Buffer 中访问。标签 **USB_BUFFER_WORKSPACE_SIZE** 定义了所需要的工作空间的字节数。

一旦一个传输 Buffer 被初始化，应用程序可以使用函数 usb_buffer_write()写入数据，当空间可用时，USB Buffer 驱动程序将处理数据包传输到底层。类似的 usb_buffer_read()可以在任何时候从接收 Buffer 读取接收到的数据。在这两种情况下，USB Buffer 都使用相同的事件协议，当更多的数据可以接收或发送数据时，较低的层使用事件指示应用程序。当将 USB 协议栈的数据传递到一个接收 Buffer 时，**USB_EVENT_RX_AVAILABLE** 事件被传递给应用程序，当数据在被发送到更低的层后从发送 Buffer 中删除数据时，**USB_EVENT_TX_COMPLETE** 事件被发送到应用程序。

### 4.1.1 Buffer驱动接口

**函数：**

- void * usb_buffer_callback_data_set(tUSBBuffer *psBuffer, void *pvCBData)
- uint32_t usb_buffer_data_available(const tUSBBuffer *psBuffer)
- uint32_t usb_buffer_event_callback(void *pvCBData, uint32_t ui32Event, uint32_t ui32MsgValue, void *pvMsgData)
- void usb_buffer_flush(const tUSBBuffer *psBuffer)
- const tUSBBuffer * usb_buffer_init(tUSBBuffer *psBuffer)
- uint32_t usb_buffer_read(const tUSBBuffer *psBuffer, uint8_t *pui8Data, uint32_t ui32Length)

- uint32_t usb_buffer_space_available(const tUSBBuffer *psBuffer)
- uint32_t usb_buffer_write(const tUSBBuffer *psBuffer, const uint8_t *pui8Data, uint32_t ui32Length)
- void usb_buffer_zero_length_packet_insert(const tUSBBuffer *psBuffer, bool bSendZLP)

#### 4. 1. 1. 1 usb_buffer_callback_data_set()

释放一个 HID 键盘设备实例。

**参数定义：**

**psBuffer**：指向 Buffer 对象实例。

**pvCBData**：设置的回调函数参数。

**简介：**

如果要使用此函数，应用程序必须确保用于描述此缓冲区的 tUSBBuffer 结构在 RAM 中，而不是 flash。通过了 pvCBData 值直接写入这个结构。

**返回值：**

返回 0 表示成功。

#### 4. 1. 1. 2 usb_buffer_data_available()

返回 Buffer 中的数据字节数。

**参数定义：**

**psBuffer**：指向 Buffer 对象实例。

**简介：**

这个函数可以用来确定 Buffer 中数据的字节数。对于接收缓冲区，这表示客户端可以使用 usb_buffer_read()读取缓冲区的字节数。对于传输缓冲区，这表明可以发送到 USB 协议栈的数据量。

**返回值：**

返回 Buffer 中的数据字节数。

#### 4. 1. 1. 3 usb_buffer_event_callback()

被 USB Buffer 对象调用用来通知客户端异步事件。

**参数定义：**

**pvCBData**：客户端指定的回调函数参数。

**ui32Event**：发送的事件。

**ui32MsgValue**：事件所指定的参数值。

**pvMsgData**：事件所指定的参数数据。

**简介：**

这个函数是 USB Buffer 相关事件处理程序，应用程序如果使用 Buffer 驱动则应该用 USB 设备类驱动程序注册，作为通道的回调。

注意，此函数不应该由应用程序调用。

**返回值：**

返回值视事件而定。

#### 4. 1. 1. 4 usb_buffer_flush()

清空 Buffer 里的数据。

**参数定义：**

**psBuffer**：指向 Buffer 对象实例。

**简介：**

此函数清除 Buffer 中所有数据。

**返回值：**

无。

#### 4. 1. 1. 5 usb_buffer_init()

初始化 Buffer 模块。

**参数定义：**

**psBuffer**：指向 Buffer 对象实例。

**简介：**

这个函数用于初始化一个 USB Buffer 对象并将其插入到潜在驱动程序和应用程序之间的函数和回调接口中。调用方在 RAM 上提供信息，用于记录创建的 Buffer 类型(发送或接收)和在底层调用的函数，以将数据传输到 USB 协议栈或从 USB 协议栈读出。

**返回值：**

返回 tUSBBuffer 的 Buffer 对象实例指针。

#### 4. 1. 1. 6 usb_buffer_read()

从 Buffer 模块读取数据。

**参数定义：**

**psBuffer**：指向 Buffer 对象实例。

**pui8Data**：指向用来缓存从 Buffer 中读出数据的缓存区。

**ui32Length**：缓存区大小。

**简介：**

该函数读取到从 USB 主机接收到应用程序的 Buffer 数据。如果接收缓冲区包含小于 ui32Length 字节的数据，则将复制目前的数据，返回值将显示复制到 pui8Data 的实际字节数。

**返回值：**

返回复制到 pui8Data 的实际字节数。

#### 4. 1. 1. 7 usb_buffer_space_available()

返回 Buffer 空闲空间。

**参数定义：**

**psBuffer**：指向 Buffer 对象实例。

**简介：**

这个函数返回 Buffer 中的空闲字节数。对于传输 Buffer，这表示可以在调用 usb_buffer_data_written()接受发送的最大字节数。对于一个接收缓冲区，它指示在 Buffer 将被填充之前可以从 USB 协议栈读取的字节数。

**返回值：**

返回缓冲区中的空闲字节数。

#### 4. 1. 1. 8 usb_buffer_write()

向 Buffer 模块写入数据。

**参数定义：**

**psBuffer**：指向 Buffer 对象实例。

**pui8Data**：指向用来写入到 Buffer 中的数据的缓存区。

**ui32Length**：缓存区大小。

**简介：**

这个函数将提供的数据复制到发送 Buffer 中。发送 Buffer 数据将根据下层的约束，尽快发送到 USB 协议栈。一旦一个数据包发送并被确认，一个 **USB_EVENT_TX_COMPLETE** 事件将被发送到应用程序回调，指示从缓冲区发送的字节数。

**返回值：**

返回实际被写入的数据字节数。

#### 4. 1. 1. 9 usb_buffer_zero_length_packet_insert()

发送一个 0 长度的数据包。

**参数定义：**

**psBuffer**：指向 Buffer 对象实例。

**bSendZLP**：true 则发送 0 长度数据包，false 则组织发送。

**简介：**

这个函数允许应用程序使用零长度的数据包。在 USB 缓冲区发送一个完整(64 字节)包的情况下，然后发现传输缓冲区是空的，默认的行为是什么都不做。一些协议要求插入一个零长度的包来标志数据的结束，当使用这种协议时，该函数应该将 bSendZLP 设为 true 以实现预期的行为。

**返回值：**

无。