小川工作室编写，本书为 LM3S 的 USB 芯片编写，上传的均为草稿，还有没修改，可能还有很多地方不足，希望各位网友原谅！
QQ:2609828265
TEL:15882446438
E-mail:paulhyde@126.com

# 第 1 章 USB 基础

## 1.1 USB 介绍

USB，Universal Serial BUS（通用串行总线）的缩写，而其中文简称为“通用串型总线”，是一个外部总线标准，用于规范电脑与外部设备的连接和通讯。是应用在 PC 领域的接口技术。USB 接口支持设备的即插即用和热插拔功能。在 1994 年底由英特尔、康柏、IBM、Microsoft 等多家公司联合提出的，目前使用最多的是 USB1.1 和 USB2.0，几乎每一台电脑上都有 4-8 个 USB 接头。USB3.0 已经“崛起”，并向下兼容，USB3.0 芯片已经大量生产。USB 具有传输速度快（USB1.1 是 12Mbps，USB2.0 是 480Mbps，USB3.0是 5 Gbps），使用方便，支持热插拔，连接灵活，独立供电等优点，可以连接鼠标、键盘、打印机、扫描仪、摄像头、闪存盘、MP3 机、手机、数码相机、移动硬盘、外置光软驱、USB 网卡、ADSL Modem、Cable Modem 等，几乎所有的外部设备。所以，掌握一种 USB 开发是很重要的。

## 1.2 USB 常用术语

USB 主机：在任何 USB 系统中，只有一个主机。USB 和主机系统的接口称作主机控制器，主机控制器可由硬件、固件和软件综合实现。

USB 设备：主机的“下行”设备，为系统提供具体功能，并受主机控制的外部 USB 设备。也称作 USB 外设，使用 USB B 型连接器连接。

集线器：集线器扩展了 USB 主机所能连接设备的数量。

SIE：串行接口引擎，USB 控制器内部的“核心”，将低级信号转换成字节，以供控制器使用，并负责处理底层协议，如填充位、CRC 生成和校验，并可发出错误报告。

端点：位于 USB 外设内部，所有通信数据的来源或目的都基于这些端点，是一个可寻址的 FIFO。似于高速公路收费口的入口或出口。一个端点地址对应一个方向。所以，端点 2-IN 与端点 2-OUT 完全不同。端点 0 默认双向控制传输，共享一个 FIFO。

设备接口：非物理接口，是一种与主机通信的信道管理。设备接口中包含多个端点，与主机进行通信。

描述符：是一个数据结构，使主机了解设备的格式化信息。每一个描述符可能包含整个设备的信息，或是设备中的一个组件。标准 USB 设备有 5 种描述符：设备描述符、配置描述符、字符串描述符、接口描述符、端点描述符。

枚举：USB 主机通过一系列命令要求设备发送描述符信息。从而知道设备具有什么功能、属于哪一类设备、要占用多少带宽、使用哪类传输方式及数据量的大小，只有主机确定了这些信息之后，设备才能真正开始工作。

三种传输速率：低速模式传输速率为 1.5Mbps，多用于键盘和鼠标。全速模式传输速率为 12Mbps。高速模式传输速率为 480Mbps

输入：相对主机而言，如果设备输入端点发送的数据为设备发送数据到主机。

输出：相对主机而言，如果设备输出端点接收的数据为主机发送到设备的数据。

## 1.3 USB 设备枚举

对于 USB 设备来说，最重要的是枚举，让主机知道设备相关信息。枚举不成功，设备无法识别、使用。本节主要讲枚举过程，有关设备其它信息，请参阅 USB 官方协议手册。

### 1.3.1 USB 设备请求

USB 设备与主机连接时会发出 USB 请求命令。每个请求命令数据包由 8 个字节（5 个字段）组成，具有相同的数据结构。

| 偏移量 | 域 | 大小 | 值 | 描述 |
|---|---|---|---|---|
| 0 | bmRequestType | 1 | 位 | 请求特征 |
| 1 | bRequest | 1 | 值 | 请求命令 |
| 2 | wValue | 2 | 值 | 请求不同，含义不同 |
| 4 | wIndex | 2 | 索引 | 请求不同，含义不同 |
| 6 | wLength | 2 | 值 | 数据传输阶段，为数据字节数 |

表 1. 数据包的格式

C 语言数据结构为：

```
typedef struct
{
    //定义传输方向、接受者、接受者。
    unsigned char bmRequestType;
    //请求类型。
    unsigned char bRequest;
    //根据请求而定实际含义。
    unsigned short wValue;
    //根据请求而定，提供索引
    unsigned short wIndex;
    //在数据传输阶段时，指示传输数据的字节数
    unsigned short wLength;
}
tUSBRequest;
```

① bmRequestType 参数：

```
//位 7，说明请求的传输方向.
#define USB_RTYPE_DIR_IN        0x80
#define USB_RTYPE_DIR_OUT       0x00
//位 6:5，定义请求的类型
#define USB_RTYPE_TYPE_M        0x60
#define USB_RTYPE_VENDOR        0x40
#define USB_RTYPE_CLASS         0x20
#define USB_RTYPE_STANDARD      0x00
// 位 4:0，定义接收者
#define USB_RTYPE_RECIPIENT_M   0x1f
#define USB_RTYPE_OTHER         0x03
#define USB_RTYPE_ENDPOINT      0x02
#define USB_RTYPE_INTERFACE     0x01
#define USB_RTYPE_DEVICE        0x00
```

② bRequest 参数：

```
//标准请求的请求类型
#define USBREQ_GET_STATUS       0x00
#define USBREQ_CLEAR_FEATURE    0x01
#define USBREQ_SET_FEATURE      0x03
```

```
#define USBREQ_SET_ADDRESS      0x05
#define USBREQ_GET_DESCRIPTOR   0x06
#define USBREQ_SET_DESCRIPTOR   0x07
#define USBREQ_GET_CONFIG       0x08
#define USBREQ_SET_CONFIG       0x09
#define USBREQ_GET_INTERFACE    0x0a
#define USBREQ_SET_INTERFACE    0x0b
#define USBREQ_SYNC_FRAME       0x0c
```

③wValue 参数：

ⓐ USBREQ_CLEAR_FEATURE 和 USBREQ_SET_FEATURE 请求命令时：

```
#define USB_FEATURE_EP_HALT     0x0000     // Endpoint halt feature.
#define USB_FEATURE_REMOTE_WAKE 0x0001   // Remote wake feature, device only.
#define USB_FEATURE_TEST_MODE   0x0002    // Test mode
```

ⓑ USBREQ_GET_DESCRIPTOR 请求命令时：

```
#define USB_DTYPE_DEVICE        1
#define USB_DTYPE_CONFIGURATION 2
#define USB_DTYPE_STRING        3
#define USB_DTYPE_INTERFACE     4
#define USB_DTYPE_ENDPOINT      5
#define USB_DTYPE_DEVICE_QUAL   6
#define USB_DTYPE_OSPEED_CONF   7
#define USB_DTYPE_INTERFACE_PWR 8
#define USB_DTYPE_OTG            9
#define USB_DTYPE_INTERFACE_ASC 11
#define USB_DTYPE_CS_INTERFACE  36
```

请求命令数据包由主机通过端点 0 发送，当设备端点为接收到请求命令数据包时，会发出端点 0 中断，控制器可以读取端点 0 中的数据，此数据格式为 tUSBRequest 所定义。根据 bmRequestType 判断是不是标准请求，bRequest 获得具体请求类型，wValue 指定更具体的对像，wIndex 指出索引和偏移。在数据传输阶段时，wLength 为传输数据的字节数。整个控制传输都依靠这 11 个标准请求命令，非标准请求通过 callback 函数返回给用户处理。

### 1.3.2 设备描述符

每一个设备都有自己的一套完整的描述符，包括设备描述符、配置描述符、接口描述符、端点描述符和字符串描述符。这些描述符反映设备特性，由特定格式排列的一组数据结构组。在 USB 设备枚举过程中，主机通过标准命令 USBREQ_GET_DESCRIPTOR 获取描述符，从而得知设备的各种特性。

① 设备描述符：

设备描述符给出了 USB 设备的一般信息。一个 USB 设备只能有一个设备描述符。主机发出 USBREQ_GET_DESCRIPTOR 请求命令，且 wValue 值为 USB_DTYPE_DEVICE 时，获取设备描述符，标准设备描述符如表 2：

| 偏移量 | 域 | 大小 | 值 | 描述 |
|---|---|---|---|---|
| 0 | bLength | 1 | 数字 | 此描述表的字节数 |
| 1 | bDecriptorType | 1 | 常量 | 描述表种类为设备 |
| 2 | bcdUSB | 2 | BCD 码 | USB 设备版本号（BCD 码） |
| 4 | bDeviceClass | 1 | 类 | 设备类码 |
| 5 | bDeviceSubClass | 1 | 子类 | 子类码 |
| 6 | bDevicePortocol | 1 | 协议 | 协议码 |
| 7 | bMaxPacketSize0 | 1 | 数字 | 端点 0 最大包大小（8,16,32,64） |
| 8 | idVendor | 2 | ID | 厂商标志（VID） |

| | | | | |
|---|---|---|---|---|
| 10 | idProduct | 2 | ID | 产品标志（PID） |
| 12 | bcdDevice | 2 | BCD 码 | 设备发行号（BCD 码） |
| 14 | iManufacturer | 1 | 索引 | 厂商信息字串索引 |
| 15 | iProduct | 1 | 索引 | 产品信息字串索引 |
| 16 | iSerialNumber | 1 | 索引 | 设备序列号信息字串索引 |
| 17 | bNumConfigurations | 1 | 数字 | 配置描述符数目 |

表 2. 标准设备描述符

C 语言设备描述符结构体为：

```
typedef struct
{
    //本描述符字节数，设备描述符为 18 字节。
    unsigned char bLength;
    //本描述符类型，设备描述符为 1，USB_DTYPE_DEVICE。
    unsigned char bDescriptorType;
    //USB 版本号
    unsigned short bcdUSB;
    //设备类代码
    unsigned char bDeviceClass;
    //子类代码
    unsigned char bDeviceSubClass;
    //协议代码
    unsigned char bDeviceProtocol;
    //端点 0 最大包长
    unsigned char bMaxPacketSize0;
     //VIID
    unsigned short idVendor;
     //PID
    unsigned short idProduct;
    //设备发行号
    unsigned short bcdDevice;
    //厂商信息字串索引
    unsigned char iManufacturer;
    //产品信息字串索引
    unsigned char iProduct;
    //设备序列号信息字串索引
    unsigned char iSerialNumber;
    //配置描述符数目
    unsigned char bNumConfigurations;
}
tDeviceDescriptor;
```

ⓐ bDecriptorType 表示本描述符类型，不同描述符，取值不一样：

```
#define USB_DTYPE_DEVICE        1
#define USB_DTYPE_CONFIGURATION 2
#define USB_DTYPE_STRING        3
#define USB_DTYPE_INTERFACE     4
#define USB_DTYPE_ENDPOINT      5
#define USB_DTYPE_DEVICE_QUAL   6
```

ⓑ bDeviceClass 本设备使用类的代码。

```
#define USB_CLASS_DEVICE        0x00
#define USB_CLASS_AUDIO         0x01
#define USB_CLASS_CDC           0x02
#define USB_CLASS_HID           0x03
#define USB_CLASS_PHYSICAL      0x05
#define USB_CLASS_IMAGE         0x06
#define USB_CLASS_PRINTER       0x07
#define USB_CLASS_MASS_STORAGE  0x08
#define USB_CLASS_HUB           0x09
#define USB_CLASS_CDC_DATA      0x0a
```

```
#define USB_CLASS_SMART_CARD    0x0b
#define USB_CLASS_SECURITY      0x0d
#define USB_CLASS_VIDEO         0x0e
#define USB_CLASS_HEALTHCARE    0x0f
#define USB_CLASS_DIAG_DEVICE   0xdc
#define USB_CLASS_WIRELESS      0xe0
#define USB_CLASS_MISC          0xef
#define USB_CLASS_APP_SPECIFIC  0xfe
#define USB_CLASS_VEND_SPECIFIC 0xff
#define USB_CLASS_EVENTS        0xffffffff
```

如果设备为鼠标，则 bDeviceClass 选为 USB_CLASS_HID 人机接口。

ⓒ bDeviceSubClass 子类码、bDevicePortocol 设备协议码，根据不同的设备类 bDeviceClass 来选择。例如，bDeviceClass = USB_CLASS_HID，即为人机接口，则 bDeviceSubClass 有以下参数可选择：

```
#define USB_HID_SCLASS_NONE     0x00
#define USB_HID_SCLASS_BOOT     0x01
```

bDevicePortocol 有以下参数可选择：

```
#define USB_HID_PROTOCOL_NONE   0
#define USB_HID_PROTOCOL_KEYB   1
#define USB_HID_PROTOCOL_MOUSE  2
```

配置一个键盘设备描述符如下：

```
tDeviceDescriptor  *HIDKEYDeviceDescriptor;
unsigned char g_pHIDDeviceDescriptor[] =
{
    18,                         // Size of this structure.
    USB_DTYPE_DEVICE,           // Type of this structure.
    USBShort(0x110),            // USB version 1.1
    USB_CLASS_HID,              // USB Device Class
    USB_HID_SCLASS_BOOT,        // USB Device Sub-class
    USB_HID_PROTOCOL_KEYB,      // USB Device protocol
    64,                         // Maximum packet size for default pipe.
    USBShort(0x1234),           // Vendor ID (VID).
    USBShort(0x5678),           // Product ID (PID).
    USBShort(0x100),            // Device Version BCD.
    1,                          // Manufacturer string identifier.
    2,                          // Product string identifier.
    3,                          // Product serial number.
    1                           // Number of configurations.
};
HIDKEYDeviceDescriptor = (tDeviceDescriptor   *)g_pHIDDeviceDescriptor;
```

② 配置描述符

配置描述符包括描述符的长度、供电方式、最大耗电量等，与设备配置相关的数据组。主机发出 USBREQ_GET_DESCRIPTOR 请求，且 wValue 值为 USB_DTYPE_CONFIGURATION 时，获取设备描述符，那么此配置包含的所有接口描述符与端点描述符都将发送给 USB 主机。USB 标准配置描述符如下表 3：

| 偏移量 | 域 | 大小 | 值 | 描述 |
|---|---|---|---|---|
| 0 | bLength | 1 | 数字 | 字节数。 |
| 1 | bDescriptorType | 1 | 常量 | 配置描述符类型 |
| 2 | wTotalLength | 2 | 数字 | 配置总长（配置、接口、端点、设备类） |
| 4 | bNumInterfaces | 1 | 数字 | 此配置所支持的接口个数 |
| 5 | bCongfigurationValue | 1 | 数字 | USBREQ_SET_CONFIG 请求选定此配置 |
| 6 | iConfiguration | 1 | 索引 | 配置的字串索引 |
| 7 | bmAttributes | 1 | 位图 | 实际电源模式 |
| 8 | MaxPower | 1 | mA | 总线电源耗费量，以 2mA 为一个单位 |

表 3. 标准配置描述符

C 语言配置描述符结构体为：

```
typedef struct
{
    //配置描述符，长度为 9。
    unsigned char bLength;
    //本描述符类型，2，USB_DTYPE_CONFIGURATION
    unsigned char bDescriptorType;
    //配置总长，包括配置、接口、端点、设备类描述符。
    unsigned short wTotalLength;
    //支持接口个数.
    unsigned char bNumInterfaces;

    // USBREQ_SET_CONFIG 请求选定此配置
    unsigned char bConfigurationValue;
    //配置描述符的字串索引
    unsigned char iConfiguration;
    //电源模式
    unsigned char bmAttributes;
    //最大耗电量。2mA 为单位。
    unsigned char bMaxPower;
}
tConfigDescriptor;
```

bmAttributes 电源模式参数：

```
#define USB_CONF_ATTR_PWR_M     0xC0        //方便使用，定义了一个屏蔽参数。
#define USB_CONF_ATTR_SELF_PWR  0xC0        //自身供电
#define USB_CONF_ATTR_BUS_PWR   0x80        //总线取电
#define USB_CONF_ATTR_RWAKE     0xA0        //Remove Wakeup
```

配置一个键盘配置描述符如下：

```
tConfigDescriptor *HIDKEYConfigDescriptor;
unsigned char g_pHIDDescriptor[] =
{
    9,                          // Size of the configuration descriptor.
    USB_DTYPE_CONFIGURATION,    // Type of this descriptor.
    USBShort(34),               // The total size of this full structure.
    1,                          // The number of interfaces in this
                                // configuration.
    1,                          // The unique value for this configuration.
    5,                          // The string identifier that describes this
                                // configuration.
    USB_CONF_ATTR_SELF_PWR,     // Bus Powered, Self Powered, remote wake up.
    250,                        // The maximum power is.500mA.
};
HIDKEYConfigDescriptor = (tConfigDescriptor *)pHIDDescriptor;
```

③ 接口描述符

配置描述符中包含了一个或多个接口描述符，“接口”为“功能”意义，例如一个设备既有鼠标功能又有键盘功能，则这个设备至少就有两个“接口”。

如果配置描述符不止支持一个接口描述符，并且每个接口描述符都有一个或多个端点描述符，那么在响应 USB 主机的配置描述符命令时，USB 设备的端点描述符总是紧跟着相关的接口描述符后面，作为配置描述符的一部分。接口描述符不可用 Set_Descriptor 和 Get_Descriptor 来存取。如果一个接口仅使用端点 0，则接口描述符以后就不再返回端点描述符，并且此接口表现的是一个控制接口的特性，在这种情况下 bNumberEndpoints 域应被设置成 0。接口描述符在说明端点个数并不把端点 0 计算在内。标准接口描述符如下表 4：

| 偏移量 | 域 | 大小 | 值 | 说明 |
|---|---|---|---|---|
| 0 | bLength | 1 | 数字 | 此表的字节数 |
| 1 | bDescriptorType | 1 | 常量 | 接口描述表类 |
| 2 | bInterfaceNumber | 1 | 数字 | 接口号，从零开始 |
| 3 | bAlternateSetting | 1 | 数字 | 索引值 |

| 4 | bNumEndpoints | 1 | 数字 | 接口端点数量 |
|---|---|---|---|---|
| 5 | bInterfaceClass | 1 | 类 | 类值 |
| 6 | bInterfaceSubClass | 1 | 子类 | 子类码 |
| 7 | bInterfaceProtocol | 1 | 协议 | 协议码 |
| 8 | iInterface | 1 | 索引 | 接口字串描述索引 |

表 4. 标准接口描述符

C 语言接口描述符结构体为：

```
typedef struct
{
    //接口描述符长度，9 字节。
    unsigned char bLength;
    //本描述符类型，4，USB_DTYPE_INTERFACE
    unsigned char bDescriptorType;
    //接口号，从 0 开始编排。
    unsigned char bInterfaceNumber;
    //接口索引值
    unsigned char bAlternateSetting;
    //本接口使用除端点 0 外的端点数。
    unsigned char bNumEndpoints;
    //USB 接口类码。
    unsigned char bInterfaceClass;
     //子类码
    unsigned char bInterfaceSubClass;
    //接口协议
    unsigned char bInterfaceProtocol;
    //描述本接口的字符串索引
    unsigned char iInterface;
}
tInterfaceDescriptor;
```

bInterfaceClass 接口使用类的代码：

```
#define USB_CLASS_DEVICE         0x00
#define USB_CLASS_AUDIO          0x01
#define USB_CLASS_CDC            0x02
#define USB_CLASS_HID            0x03
#define USB_CLASS_PHYSICAL       0x05
#define USB_CLASS_IMAGE          0x06
#define USB_CLASS_PRINTER        0x07
#define USB_CLASS_MASS_STORAGE   0x08
#define USB_CLASS_HUB            0x09
#define USB_CLASS_CDC_DATA       0x0a
#define USB_CLASS_SMART_CARD     0x0b
#define USB_CLASS_SECURITY       0x0d
#define USB_CLASS_VIDEO          0x0e
#define USB_CLASS_HEALTHCARE     0x0f
#define USB_CLASS_DIAG_DEVICE    0xdc
#define USB_CLASS_WIRELESS       0xe0
#define USB_CLASS_MISC           0xef
#define USB_CLASS_APP_SPECIFIC   0xfe
#define USB_CLASS_VEND_SPECIFIC 0xff
#define USB_CLASS_EVENTS         0xffffffff
```

例如：定义 USB 键盘接口描述符：

```
tInterfaceDescriptor  *HIDKEYIntDescriptor;
unsigned char g_pHIDInterface[] =
{
    9,                          // Size of the interface descriptor.
    USB_DTYPE_INTERFACE,        // Type of this descriptor.
    0,                          // The index for this interface.
    0,                          // The alternate setting for this interface.
```

```
    2,                          // The number of endpoints used by this
                                // interface.
    USB_CLASS_HID,              // The interface class
    USB_HID_SCLASS_BOOT,        // The interface sub-class.
    USB_HID_PROTOCOL_KEYB,      // The interface protocol for the sub-class
                                // specified above.
    4,                          // The string index for this interface.
};
```

④ 字符串描述符

字串描述符是可有可无的，如果一个设备无字串描述符，所有其它描述符中有关字串描述表的索引都必须为 0。字串描述符使用 UNICODE 编码。标准字符串描述符如表 5 所示：

| 偏移量 | 域 | 大小 | 值 | 描述 |
|---|---|---|---|---|
| 0 | bLength | 1 | 数字 | 此描述表的字节数 |
| 1 | bDescriptorType | 1 | 常量 | 字串描述表类型 |
| 2 | bString | N | 数字 | UNICODE 编码的字串 |

表 5. 标准字符串描述符

C 语言字符串描述符结构体为：

```
typedef struct
{
    //字符串描述总长度
    unsigned char bLength;
    //本描述符类型 USB_DTYPE_STRING (3)
    unsigned char bDescriptorType;
    //unicode 字符串
    unsigned char bString;
}
tStringDescriptor;
```

例如：一设备的所有字符串描述符。

```
//*****************************************************************************
// USB 键盘语言描述
//*****************************************************************************
const unsigned char g_pLangDescriptor[] =
{
    4,
    USB_DTYPE_STRING,
    USBShort(USB_LANG_EN_US)
};
//*****************************************************************************
// 制造商 字符串 描述
//*****************************************************************************
const unsigned char g_pManufacturerString[] =
{
    (11 + 1) * 2,
    USB_DTYPE_STRING,
    'O', 0, 'g', 0, 'a', 0, 'w', 0, 'a', 0, 's', 0, 't', 0, 'u', 0, 'd', 0,
    'i', 0, 'o', 0,
};
//*****************************************************************************
//产品 字符串 描述
//*****************************************************************************
const unsigned char g_pProductString[] =
{
    (17 + 1) * 2,
    USB_DTYPE_STRING,
    'K', 0, 'e', 0, 'y', 0, 'b', 0, 'o', 0, 'a', 0, 'r', 0, 'd', 0, ' ', 0,
    'F', 0, 'o', 0, 'r', 0, ' ', 0, 'p', 0, 'a', 0, 'u', 0, 'l' , 0
};
```

```
//*****************************************************************************
//   产品 序列号 描述
//*****************************************************************************
const unsigned char g_pSerialNumberString[] =
{
    (7 + 1) * 2,
    USB_DTYPE_STRING,
    '6', 0, '6', 0, '7', 0, '2', 0, '1', 0, '1', 0, '5', 0,
};
//*****************************************************************************
// 设备接口字符串描述
//*****************************************************************************
const unsigned char g_pHIDInterfaceString[] =
{
    (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
};
//*****************************************************************************
//   设备配置字符串描述
//*****************************************************************************
const unsigned char g_pConfigString[] =
{
    (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
};
//*****************************************************************************
//字符串描述集合
//*****************************************************************************
const unsigned char * const g_pStringDescriptors[] =
{
    g_pLangDescriptor,
    g_pManufacturerString,
    g_pProductString,
    g_pSerialNumberString,
    g_pHIDInterfaceString,
    g_pConfigString
};
```

也可以使用专用的软件生成 USB 字符串描述符，从而简化字符串描述符的生成，特别是在使用中文字符串描述符时，更简洁、方便。

⑤ 端点描述符

每个接口使用的端点都有自己的描述符，此描述符被主机用来决定每个端点的带宽需求。每个端点的描述符总是作为配置描述符的一部分，端点 0 无描述符。标准端点描述符如下表 6：

| 偏移量 | 域 | 大小 | 值 | 说明 |
|---|---|---|---|---|
| 0 | bLength | 1 | 数字 | 字节数 |
| 1 | bDescriptorType | 1 | 常量 | 端点描述符类型 |
| 2 | bEndpointAddress | 1 | 端点 | 端点的地址及方向 |
| 3 | bmAttributes | 1 | 位图 | 传送类型 |
| 4 | wMaxPacketSize | 2 | 数字 | 端点能够接收或发送的最大数据包的大小 |

| 6 | bInterval | 1 | 数字 | 数据传送端点的时间间隙 |
|---|---|---|---|---|

表 5. 标准端点描述符

C 语言端点描述符结构体为：

```
typedef struct
{
    //本描述符总长度
    unsigned char bLength;
    //描述符类型 USB_DTYPE_ENDPOINT (5).
    unsigned char bDescriptorType;
    //端点地址及方向
    unsigned char bEndpointAddress;
    //传输类型
    unsigned char bmAttributes;
    //传输包最大长度
    unsigned short wMaxPacketSize;
    //时间间隔
    unsigned char bInterval;
}
tEndpointDescriptor;
```

ⓐ bEndpointAddress 定义端点地址及方向，方向参数如下：

```
#define USB_EP_DESC_OUT                 0x00
#define USB_EP_DESC_IN                  0x80
```

ⓑ bmAttributes 定义端点传输类型：

```
#define USB_EP_ATTR_CONTROL             0x00
#define USB_EP_ATTR_ISOC                0x01
#define USB_EP_ATTR_BULK                0x02
#define USB_EP_ATTR_INT                 0x03
#define USB_EP_ATTR_TYPE_M              0x03
#define USB_EP_ATTR_ISOC_M              0x0c
#define USB_EP_ATTR_ISOC_NOSYNC         0x00
#define USB_EP_ATTR_ISOC_ASYNC          0x04
#define USB_EP_ATTR_ISOC_ADAPT          0x08
#define USB_EP_ATTR_ISOC_SYNC           0x0c
#define USB_EP_ATTR_USAGE_M             0x30
#define USB_EP_ATTR_USAGE_DATA          0x00
#define USB_EP_ATTR_USAGE_FEEDBACK      0x10
#define USB_EP_ATTR_USAGE_IMPFEEDBACK   0x20
```

键盘端点描述符：

```
const unsigned char g_pHIDInEndpoint[] =
{
    7,                          // The size of the endpoint descriptor.
    USB_DTYPE_ENDPOINT,         // Descriptor type is an endpoint.
    USB_EP_DESC_IN | 3,         //  Endpoint 3  -->Input
    USB_EP_ATTR_INT,            // Endpoint is an interrupt endpoint.
    USBShort(0x8),          // The maximum packet size.
    16,                         // The polling interval for this endpoint.
};
const unsigned char g_pHIDOutEndpoint[] =
{
    7,                          // The size of the endpoint descriptor.
    USB_DTYPE_ENDPOINT,         // Descriptor type is an endpoint.
    USB_EP_DESC_OUT | 3,        //  Endpoint 3  -->Output
    USB_EP_ATTR_INT,            // Endpoint is an interrupt endpoint.
    USBShort(0x8),          // The maximum packet size.
    16,                         // The polling interval for this endpoint.
};
```

g_pHIDInEndpoint 定义了端点 3 方向为输入，g_pHIDOutEndpoint 定义了端点 3 方向为输出，虽然是同一“端点 3”，但使用的 FIFO 不一样，实际为两个端点。犹如高速路出

口，一进一出，但是两个路口。注意，每个接口中使用的端点，相应端点描述符必须紧跟接口描述符返回给主机。

### 1.3.3 设备枚举过程

枚举是 USB 主机通过一系列命令要求设备发送描述符信息，标准 USB 设备有 5 种描述符：设备描述符、配置描述符、字符串描述符、接口描述符、端点描述符。枚举过程就是对设备识别过程。

① 首先，USB 主机检测到有 USB 设备插入，会对设备复位。USB 设备复位后其地址都为 0，主机就可以和刚刚插入的设备通过 0 地址端点 0 进行通信。

② USB 主机对设备发送获取设备描述符的标准请求，设备收到请求后，将设备描述符发给主机。

③ 主机对总线进行复位，之后发送 USBREQ_SET_ADDRESS 请求，设置设备地址。

④ 主机发送请求到新 USB 地址，并再获取设备描述符的标准请求。

⑤ 主机获取配置描述符，包括配置描述符、接口描述符、设备类描述符（如 HID 设备）、端点描述符等。

⑥ 主机根据已获得的描述符，请求相关字符串描述符。

对于 HID 设备，主机还会请求 HID 报告描述符。此时枚举完成，USB 设备初始化工作完成。可以通过端点与主机通信。

例如：C 语言枚举程序。

定义标准请求调用函数，方便使用：

```
typedef void (* tStdRequest)(void *pvInstance, tUSBRequest *pUSBRequest);
static const tStdRequest g_psUSBDStdRequests[] =
{
    USBDGetStatus,
    USBDClearFeature,
    0,
    USBDSetFeature,
    0,
    USBDSetAddress,
    USBDGetDescriptor,
    USBDSetDescriptor,
    USBDGetConfiguration,
    USBDSetConfiguration,
    USBDGetInterface,
    USBDSetInterface,
    USBDSyncFrame
};
```

枚举处理主函数：

```
void USBDeviceEnumHandler(tDeviceInstance *pDevInstance)
{
    unsigned long ulEPStatus;
    //获取端点 0 的中断情况，并清 0
    ulEPStatus = USBEndpointStatus(USB0_BASE, USB_EP_0);
    //判断端点 0 的当前状态
    switch(pDevInstance->eEP0State)
    {
        //如果处理于等待状态
        case USB_STATE_STATUS:
        {
            //修改端点 0 为空闲状态
            pDevInstance->eEP0State = USB_STATE_IDLE;
            // 检查地址是否改变。
            if(pDevInstance->ulDevAddress & DEV_ADDR_PENDING)
            {
                //设置设备地址
                pDevInstance->ulDevAddress &= ~DEV_ADDR_PENDING;
                USBDevAddrSet(USB0_BASE, pDevInstance->ulDevAddress);
```

```
    }
    //判断端点 0 是否有数据要接收
    if(ulEPStatus & USB_DEV_EP0_OUT_PKTRDY)
    {
        //接收并处理
        USBDReadAndDispatchRequest(0);
    }
    break;
}
//端点 0 空闲状态
case USB_STATE_IDLE:
{
     //判断端点 0 是否有数据要接收
    if(ulEPStatus & USB_DEV_EP0_OUT_PKTRDY)
    {
        //接收并处理
        USBDReadAndDispatchRequest(0);
    }
    break;
}
//端点 0 数据发送阶段
case USB_STATE_TX:
{
    USBDEP0StateTx(0);
    break;
}
// 发送端点 0 配置
case USB_STATE_TX_CONFIG:
{
    USBDEP0StateTxConfig(0);
    break;
}
//接收阶段
case USB_STATE_RX:
{
    unsigned long ulDataSize;
    if(pDevInstance->ulEP0DataRemain > EP0_MAX_PACKET_SIZE)
    {
        ulDataSize = EP0_MAX_PACKET_SIZE;
    }
    else
    {
        ulDataSize = pDevInstance->ulEP0DataRemain;
    }
    USBEndpointDataGet(USB0_BASE, USB_EP_0, pDevInstance->pEP0Data,
                       &ulDataSize);
    if(pDevInstance->ulEP0DataRemain < EP0_MAX_PACKET_SIZE)
    {
        USBDevEndpointDataAck(USB0_BASE, USB_EP_0, true);
        pDevInstance->eEP0State =  USB_STATE_IDLE;
        if((pDevInstance->psInfo->sCallbacks.pfnDataReceived) &&
           (pDevInstance->ulOUTDataSize != 0))
        {
            pDevInstance->psInfo->sCallbacks.pfnDataReceived(
                pDevInstance->pvInstance,
                pDevInstance->ulOUTDataSize);
            pDevInstance->ulOUTDataSize = 0;
        }
    }
    else
```

```
            {
                USBDevEndpointDataAck(USB0_BASE, USB_EP_0, false);
            }
            pDevInstance->pEP0Data += ulDataSize;
            pDevInstance->ulEP0DataRemain -= ulDataSize;

            break;
        }
        // STALL 状态
        case USB_STATE_STALL:
        {
            //发送 STALL 包
            if(ulEPStatus & USB_DEV_EP0_SENT_STALL)
            {
                USBDevEndpointStatusClear(USB0_BASE, USB_EP_0,
                                          USB_DEV_EP0_SENT_STALL);
                pDevInstance->eEP0State = USB_STATE_IDLE;
            }
            break;
        }
        default:
        {
            ASSERT(0);
        }
    }
}
```

枚举主要发生在端点 0 处于 USB_STATE_STATUS 和 USB_STATE_IDLE 阶段，USBDReadAndDispatchRequest(0)获取主机发送的请求，在内部进行主机请求解析并响应。USBDReadAndDispatchRequest()函数如下：

```
USBDReadAndDispatchRequest(unsigned long ulIndex)
{
    unsigned long ulSize;
    tUSBRequest *pRequest;
    pRequest = (tUSBRequest *)g_pucDataBufferIn;
    //端点 0 的最大数据包大小
    ulSize = EP0_MAX_PACKET_SIZE;
    /获取端点 0 中的数据
    USBEndpointDataGet(USB0_BASE,
                       USB_EP_0,
                       g_pucDataBufferIn,
                       &ulSize);
    //判断是否有数据读出
    if(!ulSize)
    {
        return;
    }
    //判断是否是标准请求
    if((pRequest->bmRequestType & USB_RTYPE_TYPE_M) != USB_RTYPE_STANDARD)
    {
        //如果不是标准请求，通过 callback 函数返回给用户处理。
        if(g_psUSBDevice[0].psInfo->sCallbacks.pfnRequestHandler)
        {
            g_psUSBDevice[0].psInfo->sCallbacks.pfnRequestHandler(
                    g_psUSBDevice[0].pvInstance, pRequest);
        }
        else
        {
            USBDCDStallEP0(0);
        }
    }
```

```
        else
        {
            //标准请求，通过 g_psUSBDStdRequests 调用标准请求处理函数
            if((pRequest->bRequest <
               (sizeof(g_psUSBDStdRequests) / sizeof(tStdRequest))) &&
               (g_psUSBDStdRequests[pRequest->bRequest] != 0))
            {
                g_psUSBDStdRequests[pRequest->bRequest](&g_psUSBDevice[0],
                                                        pRequest);
            }
            else
            {
                USBDCDStallEP0(0);
            }
        }
    }
```

标准请求中 USBDGetDescriptor 函数调用最多，用于描述符获取，使主机充分了解设备特性，是枚举的重要组成部分，下面是 USBDGetDescriptor 函数：

```
    static void USBDGetDescriptor(void *pvInstance, tUSBRequest *pUSBRequest)
    {
        tBoolean bConfig;
        tDeviceInstance *psUSBControl;
        tDeviceInfo *psDevice;
        psUSBControl = (tDeviceInstance *)pvInstance;
        psDevice = psUSBControl->psInfo;
        USBDevEndpointDataAck(USB0_BASE, USB_EP_0, false);
        bConfig = false;
        //通过 Value 判断获取什么描述符
        switch(pUSBRequest->wValue >> 8)
        {
            //获取设备描述符
            case USB_DTYPE_DEVICE:
            {
                //把设备描述放入 PE0 中，等待发送。
                psUSBControl->pEP0Data =
                    (unsigned char *)psDevice->pDeviceDescriptor;
                psUSBControl->ulEP0DataRemain = psDevice->pDeviceDescriptor[0];
                break;
            }
            //获取配置描述符
            case USB_DTYPE_CONFIGURATION:
            {
                const tConfigHeader *psConfig;
                const tDeviceDescriptor *psDeviceDesc;
                unsigned char ucIndex;
                ucIndex = (unsigned char)(pUSBRequest->wValue & 0xFF);
                psDeviceDesc =
                    (const tDeviceDescriptor *)psDevice->pDeviceDescriptor;
                if(ucIndex >= psDeviceDesc->bNumConfigurations)
                {
                    USBDCDStallEP0(0);
                    psUSBControl->pEP0Data = 0;
                    psUSBControl->ulEP0DataRemain = 0;
                }
                else
                {
                    psConfig = psDevice->ppConfigDescriptors[ucIndex];
                    psUSBControl->ucConfigSection = 0;
                    psUSBControl->ucSectionOffset = 0;
                    psUSBControl->pEP0Data = (unsigned char *)
```

```
                                        psConfig->psSections[0]->pucData;
                psUSBControl->ulEP0DataRemain =
                                          USBDCDConfigDescGetSize(psConfig);
                psUSBControl->ucConfigIndex = ucIndex;
                bConfig = true;
            }
            break;
        }
        //字符串描述符
        case USB_DTYPE_STRING:
        {
            long lIndex;
            lIndex = USBDStringIndexFromRequest(pUSBRequest->wIndex,
                                                pUSBRequest->wValue & 0xFF);
            if(lIndex == -1)
            {
                USBDCDStallEP0(0);
                break;
            }
            psUSBControl->pEP0Data =
                (unsigned char *)psDevice->ppStringDescriptors[lIndex];
            psUSBControl->ulEP0DataRemain =
                psDevice->ppStringDescriptors[lIndex][0];
            break;
        }
        default:
        {
            if(psDevice->sCallbacks.pfnGetDescriptor)
            {
                psDevice->sCallbacks.pfnGetDescriptor(psUSBControl->pvInstance,
                                                      pUSBRequest);
                return;
            }
            else
            {
                USBDCDStallEP0(0);
            }
            break;
        }
    }
    //判断是否有数据要发送
    if(psUSBControl->pEP0Data)
    {
        if(psUSBControl->ulEP0DataRemain > pUSBRequest->wLength)
        {
            psUSBControl->ulEP0DataRemain = pUSBRequest->wLength;
        }
        if(!bConfig)
        {
            //发送上面的配置信息
            USBDEP0StateTx(0);
        }
        else
        {
            //发送端点 0 的状态信息
            USBDEP0StateTxConfig(0);
        }
    }
}
```

下面是一个枚举过程：

```
Start Demo:
Init Hardware..............
LED && KEY  Ok!..............
USB_STATE_IDLE..............（端点 0 处于 USB_STATE_IDLE 状态）
0x0008  0x0080  0x0006  0x0100  0x0000  0x0040     （主机请求数据：数据长度+数据包内容）
USBDGetDescriptor..............                    （解析为获取描述符）
USBDGetDescriptor USB_DTYPE_DEVICE.............    （获取设备描述符，并发送设备描述符）
USB_STATE_STATUS..............                     （端点 0 处于 USB_STATE_STATUS 状态）
USB_STATE_IDLE..............                      （端点 0 处于 USB_STATE_IDLE 状态）
0x0008  0x0000  0x0005  0x0002  0x0000  0x0000     （主机请求数据：数据长度+数据包内容）
USBDSetAddress..............                  （设置设备地址请求）
USB_STATE_STATUS..............
DEV is 0x0002..............                       （设置设备地址为 2）
USB_STATE_IDLE..............
0x0008  0x0080  0x0006  0x0100  0x0000  0x0012
USBDGetDescriptor..............
USBDGetDescriptor USB_DTYPE_DEVICE..............（设置地址后，再获取设备描述符）
USB_STATE_STATUS..............
USB_STATE_IDLE..............
0x0008  0x0080  0x0006  0x0200  0x0000  0x0009
USBDGetDescriptor..............
USBDGetDescriptor USB_DTYPE_CONFIGURATION..............（获取配置描述符）
USB_STATE_STATUS..............
USB_STATE_IDLE..............
0x0008  0x0080  0x0006  0x0300  0x0000  0x00ff
USBDGetDescriptor..............
USBDGetDescriptor USB_DTYPE_STRING..............      （获取字符串描述符）
USB_STATE_STATUS..............
USB_STATE_IDLE..............
0x0008  0x0080  0x0006  0x0303  0x0409  0x00ff
USBDGetDescriptor..............
USBDGetDescriptor USB_DTYPE_STRING..............
String is 0x0003,Vaule is 0x0409,Index is 0x0003
USB_STATE_STATUS..............
USB_STATE_IDLE..............
0x0008  0x0080  0x0006  0x0200  0x0000  0x00ff
USBDGetDescriptor..............
USBDGetDescriptor USB_DTYPE_CONFIGURATION..............
USB_STATE_STATUS..............
USB_STATE_IDLE..............
0x0008  0x0080  0x0006  0x0300  0x0000  0x00ff
USBDGetDescriptor..............
USBDGetDescriptor USB_DTYPE_STRING..............
USB_STATE_STATUS..............
USB_STATE_IDLE..............
0x0008  0x0080  0x0006  0x0302  0x0409  0x00ff
USBDGetDescriptor..............
USBDGetDescriptor USB_DTYPE_STRING..............
String is 0x0002,Vaule is 0x0409,Index is 0x0002
USB_STATE_STATUS..............
USB_STATE_IDLE..............
0x0008  0x0080  0x0006  0x0300  0x0000  0x00ff
USBDGetDescriptor..............
USBDGetDescriptor USB_DTYPE_STRING..............
USB_STATE_STATUS..............
USB_STATE_IDLE..............
0x0008  0x0080  0x0006  0x0302  0x0409  0x00ff
USBDGetDescriptor..............
USBDGetDescriptor USB_DTYPE_STRING..............
String is 0x0002,Vaule is 0x0409,Index is 0x0002
```

USB_STATE_STATUS.....
USB_STATE_IDLE.....
USB_STATE_IDLE.....

以上是枚举的实际过程，从上面枚举过程中可以看出枚举一般过程：上电检测→获取设备描述符→设置设备地址→再次获取设备描述符→获取配置描述符→获取字符串描述符→枚举成功。对于不同系统，枚举过程是不一样的，以上是在 win xp 下的枚举过程，在 Linux 下，枚举过程有很大不同。