

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 个字节）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:

① 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
```

② 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 指定更具体的对象，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	bDescriptorType	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	bDeviceProtocol	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	厂商信息字符串索引
15	iProduct	1	索引	产品信息字符串索引
16	iSerialNumber	1	索引	设备序列号信息字符串索引
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;
    //VID
    unsigned short idVendor;
    //PID
    unsigned short idProduct;
    //设备发行号
    unsigned short bcdDevice;
    //厂商信息字符串索引
    unsigned char iManufacturer;
    //产品信息字符串索引
    unsigned char iProduct;
    //设备序列号信息字符串索引
    unsigned char iSerialNumber;
    //配置描述符数目
    unsigned char bNumConfigurations;
}
tDeviceDescriptor;
```

① bDescriptorType 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
```

② 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	总线电源耗电量，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;
    //配置描述符的字符串索引
    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	接口字符串描述索引

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

字符串描述符是可有可无的，如果一个设备无字符串描述符，所有其它描述符中有关字符串描述表的索引都必须为 0。**字符串描述符使用 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	字符串描述表类型
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隔
---	-----------	---	----	-------------

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;
```

③ 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
```

④ 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_psUSBStdRequests[] =
{
    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
        USBReadAndDispatchRequest(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
        USBReadAndDispatchRequest(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，USBReadAndDispatchRequest(0) 获取主机发送的请求，在内部进行主机请求解析并响应。USBReadAndDispatchRequest() 函数如下：

```

USBReadAndDispatchRequest(unsigned long ulIndex)
{
    unsigned long ulSize;
    tUSBRequest *pRequest;
    pRequest = (tUSBRequest *)g_pucDataBufferIn;
    //端点 0 的最大数据包大小
    ulSize = EPO_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_psUSBStdRequests 调用标准请求处理函数
    if((pRequest->bRequest <
        (sizeof(g_psUSBStdRequests) / sizeof(tStdRequest))) &&
        (g_psUSBStdRequests[pRequest->bRequest] != 0))
    {
        g_psUSBStdRequests[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 PEO 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 = USBStringIndexFromRequest(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。

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

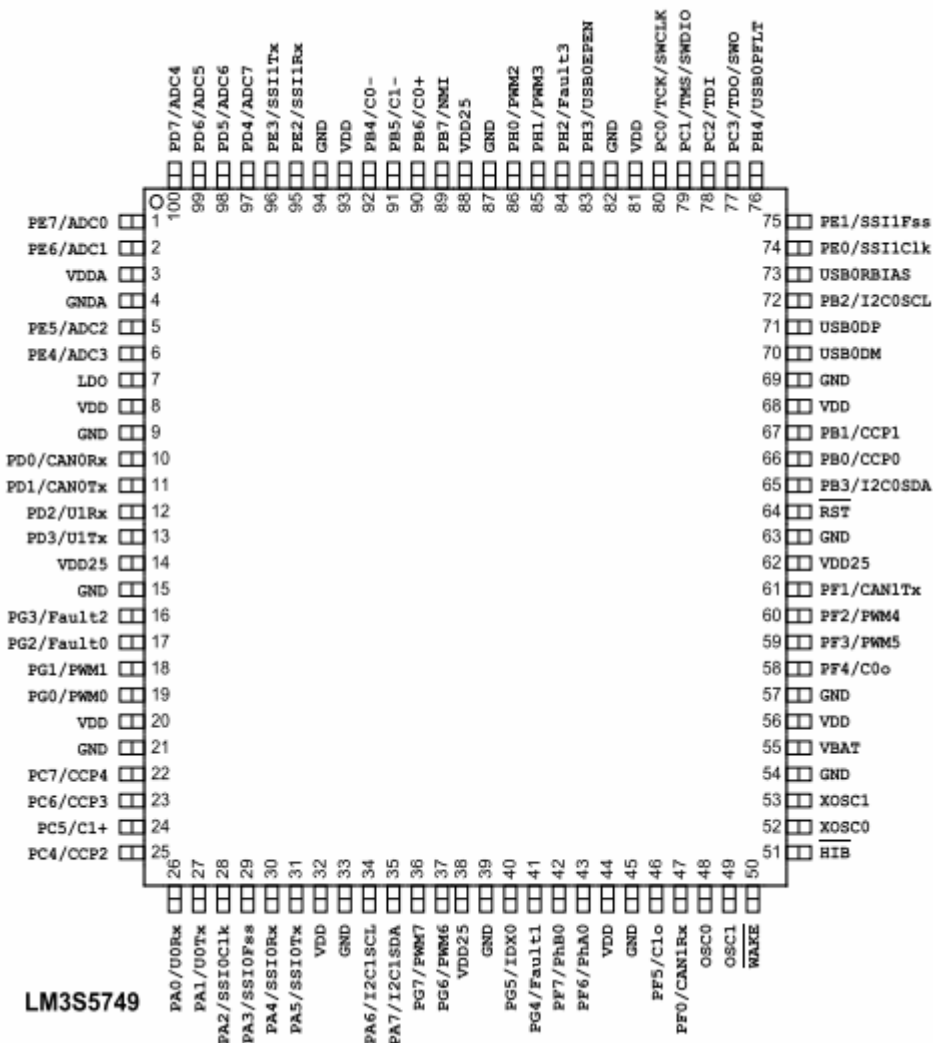
TEL: 15882446438

E-mail: paulhyde@126.com

第二章 LM3S USB 处理器

2.1 LM3S 处理器简介

Luminary Micr 公司 Stellaris 所提供一系列的微控制器是首款基于 Cortex-m3 的控制器，它们为对成本尤其敏感的嵌入式微控制器应用方案带来了高性能的 32 位运算能力。这些具备领先技术的芯片使用户能够以传统的 8 位和 16 位器件的价位来享受 32 位的性能，而且所有型号都是以小占位面积的封装形式提供。



Stellaris 系列芯片能够提供高效的性能、广泛的集成功能以及按照要求定位的选择，适用于各种关注成本并明确要求具有的过程控制以及连接能力的应用方案。LM3S3000、LM3S5000 系列，支持最大主频为 50 MHz，128 KByte FLASH, 32~64 KByte SRAM，LQFP-64/LQFP-100 封装，集成 CAN 控制器、睡眠模块、正交编码器、ADC、带死区 PWM、温度传感器、模拟比较器、UART、SSI、通用定时器，I2C、CCP、DMA 控制器等外设，芯片内部固化驱动库，支持 USB Host/Device/OTG 功能，可运行在全速和低速模式，它符合 USB2.0 标准，包含挂起和唤醒信号。它包含 32 个端点，其中包含 2 个用于控制传输的专用连接端点（一个用于输入，一个用于输出），其他 30 个端点带有可软件动态定义大小的 FIFO 并支持多包队列。FIFO 支持 μ DMA，可有效降低系统资源的占用。USB Device 启动方式灵活，可软件控制是否在启动时连接。USB 控制器遵从 OTG 标准的会话请求协议 (SRP) 和主机协商协议 (HNP)。

Stellaris USB 模块特性:

- 符合 USB-IF 认证标准
- 支持 USB 2.0 全速模式 (12 Mbps) 和低速模式 (1.5 Mbps)
- 集成 PHY
- 传输类型: 控制传输 (Control)，中断传输 (Interrupt)，批量传输 (Bulk)，等时传输 (Isochronous)

e. 32 端点:1 个专用的输入控制端点和 1 个专用输出控制端点; 15 个可配置的输入端点和 15 个可配置的输出端点。

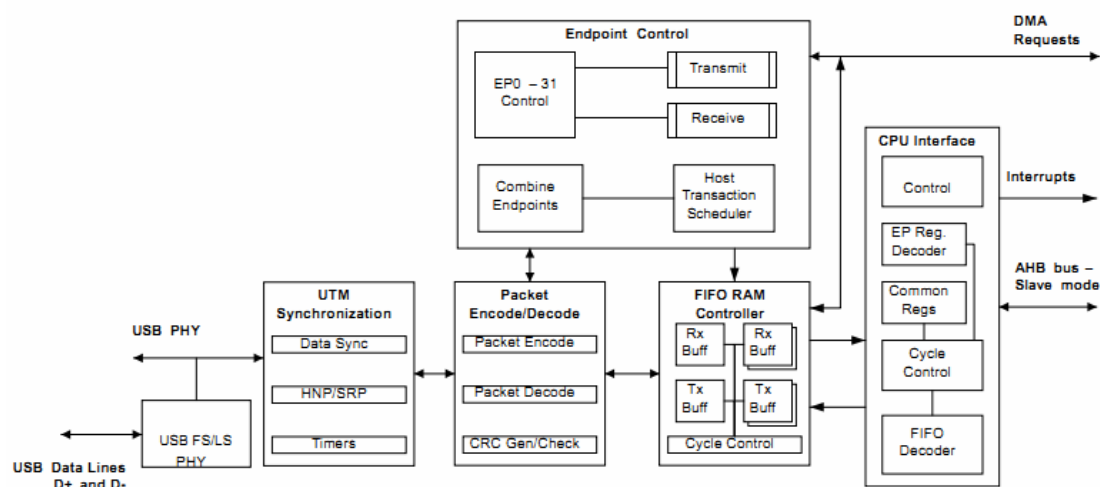
f. 4 KB 专用端点内存空间: 可支持双缓存的 1023 字节最大包长的等时传输。

g. 支持 VBUS 电压浮动(droop)和有效 ID 检测, 并产生中断信号

h. 高效传输 μ DMA :用于发送和接收的独立通道多达 3 个输入端点和 3 个输出端点, 当 FIFO 中包含需要的大量数据时, 触发通道请求。

本书主要讲 Stellaris USB 处理器的相关使用与 USB 协议, 有关 LM3S 系列处理器其它外设操作与使用请参考《嵌入式系统原理与应用—基于 Cortex-m3 和 uc/os-II》, ISBN: 978564709310, 电子科技大学出版社。

2.2 Stellaris USB 模块



USB 模块框图

一些 USB 控制器的信号是 GPIO 的复用功能, 这些管脚在复位时默认设置为 GPIO 信号。当需要使用 USB 功能时, 应将相关 GPIO 备选功能选择寄存器 (GPIOAFSEL) 中的 AFSEL 位置位, 表示启用 GPIO 的备选功能; 同时还应将括号内的数字写入 GPIO 端口控制寄存器 (GPIOCTRL) 的 PMCN 位域, 表示将 USB 信号分配给指定的 GPIO 管脚。USB0VBUS 和 USB0ID 信号通过清除 GPIO 数字使能寄存器 (GPIODEN) 中相应的 DEN 位来配置。关于配置 GPIO 的详细信息, 请参阅《嵌入式系统原理与应用—基于 Cortex-m3 和 uc/os-II》(ISBN: 978564709310, 电子科技大学出版社) 及相关数据手册。

当用于 OTG 模式时, 由于 USB0VBUS 和 USB0ID 是 USB 专用的管脚, 不需要配置, 直接连接到 USB 连接器的 VBUS 和 ID 信号。如果 USB 控制器专用于主机或设备, USB 通用控制和状态寄存器 (USBGPCS) 中的 DEVMODOTG 和 DEVMOD 位用于连接 USB0VBUS 和 USB0ID 到内部固定电平, 释放 PB0 和 PB1 管脚用于通用 GPIO。当用作自供电的设备时, 需要监测 VBUS 值, 来确定主机是否断开 VBUS, 从而禁止自供电设备 D+/D- 上的上拉电阻。此功能可通过将一个标准 GPIO 连接到 VBUS 实现。

2.2.1 功能描述

在管脚 USB0RBIAS 和地之间需要接一个 1%精度的 9.1 k Ω 电阻, 且该电阻离 USB0RBIAS 管脚越近越好。由于损耗在该电阻上的功率很小, 可以采用贴片电阻。Stellaris USB 支持



OTG 标准的回话请求协议 (SRP) 和主机协商协议 (HNP)，提供完整的 OTG 协商。回话请求协议 (SRP) 允许连接在 USB 线缆 B 端的 B 类设备向连接在 A 端的 A 类设备发出请求，通知 A 类设备打开 VBUS 电源。主机协商协议 (HNP) 用于在初始会话请求协议提供供电后，决定 USB 线缆哪端的设备作为 USB Host 主控制器。当连接到非 OTG 外设或设备时，OTG 控制器可以探测出线缆的另一端接入是 USB Host 主机还是 Device 设备，并通过一个寄存器指示 OTG 运行在 Host 主机还是 Device 设备角色，上述过程是 USB 控制器自动完成的。基于这种自动探测机制，系统使用 A 类/B 类连接器取代 AB 类连接器，可支持与另外的 OTG 设备完整的 OTG 协商。

另外，USB 控制器支持接入非 OTG 外设或 Host 主控制器。它可以被设置为专用于 Host 或 Device 功能，此时 USB0VBUS 和 USB0ID 管脚可被设置作为的 GPIO 口使用。当 USB 控制器被用作自供电的 Device 时，必须将 VBUS 接到 GPIO 或模拟比较器的输入，在 VBUS 掉电时产生中断，用于关闭 USB0DP 的上拉电阻。

注意：当使用 USB 模块时，系统必须运行在 20MHz 频率或以上。

2.2.2 作为设备

USB 控制器作为 USB 设备 (Device) 操作时，输入事务通过使用端点的发送端点寄存器，由端点的发送接口进行控制。输出事务通过使用端点的接收端点寄存器，由端点的接收接口进行控制。当配置端点的 FIFO 大小时，需要考虑最大数据包大小。

- a. 批量传输：批量端点的 FIFO 可配置为最大包长 (最大 64 字节)，如果使用双包缓存，则需要配置为 2 倍于最大包长大小 (后面的章节将详细描述)。
- b. 中断传输：中断端点的 FIFO 可配置为最大包长 (最大 64 字节)，如果使用双包缓存，则需要配置为 2 倍于最大包长大小。
- c. 等时传输：等时端点的 FIFO 比较灵活，最大支持 1023 字节。
- d. 控制传输：USB 设备可能指定一个独立的控制端点，但在大多数情况，USB 设备使用 USB 控制器的端点 0 作为专用的控制端点。

① 端点

USB 控制器作为设备运行时，提供两个专用的控制端点 (输入和输出) 和用于与主机通讯的 30 个可配置的端点 (15 输入和 15 输出)。端点的端点号和方向与对应的相关寄存器有直接联系。比如，当主机发送到端点 1，所有的配置和数据存在于端点 1 发送寄存器接口中。端点 0 是专用的控制端点，用于枚举期间的端点 0 的所有控制传输或其他端点 0 的控制请求。端点 0 使用 USB 控制器 FIFO 内存 (RAM) 的前 64 字节，此内存对于输入事务和输出事务是共享的。其余 30 个端点可配置为控制端点、批量端点、中断端点或等时端点。他们应被作为 15 个可配置的输入端点和 15 个可配置的输出端点来对待。这些成对的端点的输入和输出端点不需要必须配置为相同类型，比如端点对 (endpoint pairs) 的输出部分可以设置为批量端点，而输入部分可以设置为中断端点。每个端点的 FIFO 的地址和大小可以根据应用需求来修改。

② 输入事务

输入事务是相对主机而言的，输入事务的数据通过发送端点的 FIFO 来处理。15 个可配置的输入端点的 FIFO 大小由 USB 发送 FIFO 起始地址寄存器 (USBTXFIFOADD) 决定，传输时发送端点 FIFO 中的最大数据包大小可编程配置，该大小由写入该端点的 USB 端点 n 最大发送数据寄存器 (USBTXMAXPn) 中的值决定，USBTXMAXPn 值不能大于 FIFO 的大小。端点的 FIFO 可配置为双包缓存或单包缓存，当双包缓存使能时，FIFO 中可缓冲两个数据包，这需要 FIFO 至少为两个数据包大小。当不使用双包缓存时，即使数据包的大小小于 FIFO 大小的一半，也只能缓冲一个数据包。

单包缓冲：如果发送端点 FIFO 的大小小于该端点最大包长的两倍时(由 USB 发送动态 FIFO 大小寄存器 USBTXFIFOSZ 设定)，只能使用单包缓冲，在 FIFO 中缓冲一个数据包。当数据包已装载到 TXFIFO 中时，USB 端点 n 发送控制和状态低字节寄存器 USBTXCSRLn 中的 TXRDY 位必须被置位，如果 USB 端点 n 发送控制和状态高字节寄存器 USBTXCSRHn 中的 AUTOSET 位被置 1，TXRDY 位将在最大包长的包装载到 FIFO 中时自动置位；如果数据包小于最大包长，位 TXRDY 必须手动置位。当 TXRDY 位被手动或自动置 1 时，表明要发送的数据包已准备好。如果数据包成功发送，TXRDY 位和 FIFONE 位将被清 0，同时产生相应的中断信号，此时下一包数据可装载到 FIFO 中。

双包缓存：如果发送端点 FIFO 的大小至少两倍于该端点最大包长时，允许使用双包缓存，FIFO 中可以缓冲两个数据包。当数据包已装载到 TXFIFO 中时，USBTXCSRLn 中的 TXRDY 位必须被置位，如果寄存器 USBTXCSRHn 中的 AUTOSET 位被置 1，TXRDY 位将在最大包长的包装载到 FIFO 中时自动置位；如果数据包小于最大包长，位 TXRDY 必须手动置位。当 TXRDY 位被手动或自动置 1 时，表明要发送的数据包已准备好。在装载完第一个包后，TXRDY 位立即清除，同时产生中断信号；此时第二个数据包可装载到 TXFIFO 中，TXRDY 位重新置位(手动或自动)，此时，两个要发送的包都已准备好，如果任一数据包成功发送，TXRDY 位和 FIFONE 位将被清 0，同时产生相应的发送端点中断信号，此时下一包数据可装载到 TXFIFO 中。寄存器 USBTXCSRLn 中的 FIFONE 位的状态表明此时可以装载几个包，如果 FIFONE 位置 1，表明 FIFO 中还有一个包未发送，只能装载一个数据包；如果 FIFONE 位为 0，表明 FIFO 中没有未发送的包，可以装载两个数据包。

如果 USB 发送双包缓存禁止寄存器 USBTXDPKTBUDIS 中的 EPn 位置位，相应的端点禁止双包缓存。此位缺省为置 1，需要使能双包缓存时必须清 0 该位。

③ 输出事务

输出事务是相对主机而言的，输出事务的数据通过接收端点的 FIFO 来处理。15 个可配置的输出端点的 FIFO 大小由 USB 接收 FIFO 起始地址寄存器(USBRXFIFOADD)决定，传输时接收端点 FIFO 中的最大数据包大小可编程配置，该大小由写入该端点的 USB 端点 n 最大接收数据寄存器 (USBRXMAXPn)中的值决定。端点的 FIFO 可配置为双包缓存或单包缓存，当双包缓存使能时 FIFO 中可缓冲两个数据包。当不使用双包缓存时，即使数据包的大小小于 FIFO 大小的一半，也只能缓冲一个数据包。

单包缓存：如果接收端点 FIFO 的大小小于该端点最大包长的两倍时，只能使用单包缓冲，在 FIFO 中缓冲一个数据包。当数据包已接收到 RXFIFO 中时，USB 端点 n 接收控制和状态低字节寄存器 USBRXCSRLn 中的 RXRDY 和 FULL 位置位，同时发出相应的中断信号，表明接收 FIFO 中有一个数据包需要读出。当数据包从 FIFO 中读出时，RXRDY 位必须被清 0 以允许接收后面的数据包，同时向 USB 主机发送确认信号。如果 USB 端点 n 接收控制和状态高字节寄存器 USBRXCSRHn 中的 AUTOCl 位被置 1，RXRDY 和 FULL 位将在最大包长的包从 FIFO 中读出时自动清 0；如果数据包小于最大包长，位 RXRDY 必须手动清 0。

双包缓存：如果接收端点的 FIFO 大小不小于该端点的最大包长的 2 倍时，可以使用双缓冲机制缓存两个数据包。当第一个数据包被接收缓存到 RXFIFO，寄存器 USBRXCSRLn 中的 RXRDY 位置位，同时产生相应的接收端点中断信号，指示有一个数据包需要从 RXFIFO 中读出。当第一个数据包被接收时，寄存器 USBRXCSRLn 的 FULL 位不置位，该位只有在第二个数据包被接收缓存到 FIFO 时才置位。当从 FIFO 中读出一个包时，RXRDY 位必须清 0 以允许接收后面的包。如果 USB 端点 n 接收控制和状态高字节寄存器 USBRXCSRHn 中的 AUTOCl 位被置 1，RXRDY 位将在最大包长的包从 FIFO 中读出时自动清 0；如果数据包小于最大包长，RXRDY 位必须手动清 0。当 RXRDY 位清 0 时，FULL 位为 1，USB 控制器先清除 FULL 位，然后再置位 RXRDY 位，表明 FIFO 中的另一个数据包等待被读出。

如果 USB 接收双包缓存禁止寄存器 USBRXDPKTBUFDIS 中的 EPn 位置位，相应的端点禁止双包缓存。此位缺省为置 1，需要使能双包缓存时必须清 0 该位。

④ 调度

传输事务由 Host 主机控制器调度决定，Device 设备无法控制事务调度。设备等待 Host 主控制器发出请求，随时可建立传输事务。当传输事务完成或由于某些原因被终止时，会产生中断信号。当 Host 主控制器发起请求，而 Device 设备还没有准备好，设备会返回一个 NAK 忙信号。

⑤ 设备挂起 (SUSPEND)

USB 总线空闲达 3ms 时，USB 控制器自动进入挂起 (SUSPEND) 模式。如果 USB 中断使能寄存器 USBIE 中使能挂起 (SUSPEND) 中断，此时会发出一个中断信号。当 USB 控制器进入挂起模式，USB PHY 也将进入挂起模式。当检测到唤醒 (RESUME) 信号时，USB 控制器退出挂起模式，同时使 USB PHY 退出挂起模式，此时如果唤醒中断使能，将产生中断信号。**设置 USB 电源寄存器 USBPOWER 中的 RESUME 位同样可以强制 USB 控制器退出挂起模式。当此位置位，USB 控制器退出挂起模式，同时在总线上发出唤醒信号。RESUME 位必须在 10ms (最大 15ms) 后清 0 来结束唤醒信号。**为满足电源功耗需求，LM3S USB 控制器可进入深睡眠模式。

⑥ 帧起始

当 USB 控制器运行在设备模式，它每 1ms 收到一次主机发出的帧起始包 (SOF)。当收到 SOF 包时，包中包含的 11 位帧号写入 USB 帧值寄存器 USBFRAME 中，同时发出 SOF 中断信号，由应用程序处理。一旦 USB 控制器开始收到 SOF 包，它将预期每 1ms 收到 1 次。如果超过 1.00358ms 没有收到 SOF 包，将假定此包丢失，寄存器 USBFRAME 也将不更新。当 SOF 包重新成功接收时，USB 控制器继续，并重新同步这些脉冲。

使用 SOF 中断可以每 1ms 获得一个基本时钟，可当 tick 使用，并且使用方便。后面会有重要应用。

⑥ USB 复位

当 USB 控制器处于设备模式，如果检测到 USB 总线上复位信号，USB 控制将自动：清除寄存器 USBFADDR，清除 USB 端点索引寄存器 (USBEPIDX)，清空所有端点 FIFO，清除所有控制/状态寄存器，使能所有端点中断，产生复位中断信号。刚接入到主机的 USB 设备，USB 复位意味着要进行枚举了。

2.2.3 作为主机

当 Stellaris USB 控制器运行在主机模式时，可与其他 USB 设备的点对点通讯，也可用于连接到集线器，与多个设备进行通讯。USB 控制器支持全速和低速设备。它自动执行必要的事务传输，允许 USB 2.0 集线器使用低速设备和全速设备。支持控制传输、批量传输、等时传输和中断传输。输入事务由端点的接收接口进行控制；输出事务使用端点的发送端点寄存器。当配置端点的 FIFO 大小时，需要考虑最大数据包大小。

① 端点

端点寄存器用于控制 USB 端点接口，通过接口可与设备进行通讯。主机端点由 1 个专用控制输入端点、1 个专用控制输出端点、15 个可配置的输出端点和 15 个可配置的输入端点组成。控制端点只能与设备的端点 0 进行控制传输，用于设备枚举或其他使用设备端点 0 的控制功能。**控制端点输入输出事务共享一个 FIFO 存储空间，并在 FIFO 的前 64 字节。**其余输入和输出端点可配置为：控制传输端点、批量传输端点、中断传输端点或等时传输端点。**输入和输出控制有成对的三组寄存器，它们可以与不同类型的端点以及不同设备的不同端点进行通讯。例如，第一对端点可分开控制，输出部分与设备的批量输出端点 1 通讯，同时输入部分与设备的中断端点 2 通讯。**FIFO 的地址和大小可以软件设置，并且可以指定用于某一个端点输入或者输出传输。

无论点对点通信还是集线器通信,在访问设备之前,必须设置端点 n 的接收功能地址寄存器 USBRXFUNCADDRn 和端点 n 的发送功能地址寄存器 USBTXFUNCADDRn。LM3S 系列 USB 控制器支持通过集线器连接设备,一个寄存器就可以实现,该寄存器说明集线器地址和每个传输的端口。**在本书中的所有例程都未使用集线器访问设备。**

② 输入事务

输入事务,相对主机而言是数据输入,与设备输出事务类似的,**但传输数据必须通过设置寄存器 USBCSRLO 中的 REQPKT 位开始,向事务调度表明此端点存在一个活动的传输。**此时事务调度向目标设备发送一个输入令牌包。当主机 RXFIFO 中接收到数据包时,寄存器 USBCSRLO 的 RXRDY 位置位,同时产生相应的接收端点中断信号,指示 RXFIFO 中有数据包需要读出。

当数据包被读出时, RXRDY 位必须清 0。寄存器 USBRXCSRHn 中的 AUTOCL 位可用于当最大包长的包从 RXFIFO 中读出时将 RXRDY 位自动清 0。寄存器 USBRXCSRHn 中的 AUTORQ 位用于当 RXRDY 位清 0 时将 REQPKT 位自动置位。AUTOCL 和 AUTORQ 位用于 μ DMA 访问,在主处理器不干预时完成批量传输。

当 RXRDY 位清 0 时,控制器向设备发送确认信号。当传输一定数据包时,需要将端点 n 的 USBRQPKTCOUNTn 寄存器配置为**要传输包数量**,每一次传输, USBRQPKTCOUNTn 寄存器中的值减 1,当减到 0 时, AUTORQ 位清 0 来阻止后面试图进行的数据传输。如传输数量未知, USBRQPKTCOUNTn 寄存器必须清 0, AUTORQ 位保持置位,直到收到结束包, AUTORQ 位才清 0 (小于 USBRXMAXPn 寄存器中的 MAXLOAD 值)。

用图表示传输过程。

③ 输出事务

当数据包装载到 TXFIFO 中时, USBTXCSRn 寄存器中的 TXRDY 位必须置位。如果置位了 USBTXCSRn 寄存器中的 AUTOSET 位,当最大包长的数据包装载到 TXFIFO 中时, TXRDY 位自动置位。此外, AUTOSET 位与 μ DMA 控制器配合使用,可以在不需要软件干预的情况下完成批量传输。

④ 调度

调度由 USB 主机控制器自动处理。**中断传输可以是每 1 帧进行一次,也可以每 255 帧进行一次,可以在 1 帧到 255 之间以 1 帧增量调度。批量端点不允许调度参数,但在设备的端点不响应时,允许 NAK 超时。等时端点可以在每 1 帧到每 216 帧之间调度(2 的幂)。**

USB 控制器维持帧计数,并发送 SOF 包, SOF 包发送后, USB 主机控制器检查所有配置好的端点,寻找激活的传输事务。**REQPKT 位置位的接收端点或 TXRDY 或 FIFONE 位置位的发送端点,被视为存在激活的传输事务。**

如果传输建立在一帧的第一个调度周期,而且端点的间隔计数器减到 0,则等时传输和中断传输开始。所以每个端点的中断传输和等时传输每 N 帧才发生一次, N 是通过 USB 主机端点 n 的 USBTXINTERVALn 寄存器或 USB 主机端点 n 的 USBRXINTERVALn 寄存器设置的间隔。

如果在帧中下一个 SOF 包之前有足够的提供足够的时间完成传输,则激活的批量传输立即开始。如果传输需要重发时(例如,收到 NAK 或设备未响应),需要在调度器先检查完其他所有端点是否有激活的传输之后,传输才能重传。这保证了一个发送大量 NAK 响应的端点不阻塞总线上的其他传输正常进行。

⑤ USB集线器

以下过程只适用于 USB2.0 集线器的主机。当低速设备或全速设备通过 USB 2.0 集线器连接到 USB 主机时,集线器地址和端口信息必须记录在相应的 USB 端点 n 的 USBRXHUBADDRn 寄存器和 USB 端点 n 的 USBRXHUBPORTn 寄存器或者 USB 端点 n 的 USBTXHUBADDRn 寄存器和 USB 端点 n 的 USBTXHUBPORTn 寄存器。

此外,设备的运行速度(全速或低速)必须记录在 USB 端点 0 的 USBTYPE0 寄存器,和设

备访问主机 USB 端点 n 的 USBTXTYPE_n 寄存器，或者主机 USB 端点 n 的 USBRXTYPE_n 寄存器。

对于集线器通讯，这些寄存器的设置记录了 USB 设备当前相应端点的配置。为了支持更多数量的设备，USB 主机控制器允许通过更新这些寄存器配置来实现。

2.2.4 OTG 模式

OTG 就是 On The Go，正在进行中的意思，可以进行“主机与设备”模式切换。USB OTG 允许使用时才给 VBUS 上电，不使用 USB 总线时，则关断 VBUS。VBUS 由总线上的 A 设备提供电源。OTG 控制器通过 PHY 采样 ID 输入信号分辨 A 设备和 B 设备。ID 信号拉低时，检测到插入 A 设备(表示 OTG 控制器作为 A 设备角色)；ID 信号为高时，检测到插入 B 设备(表示 OTG 控制器作为 B 设备角色)。注意当在 OTG A 和 OTG B 之间切换时，控制器保留所有的寄存器内容。关于 OTG 使用不在本书重点介绍。

2.3 寄存器描述

使用 USB 处理器进行 USB 主机、设备开发离不开相关寄存器操作，USB 本身就相当复杂，寄存器相当多，下面就几个常用的寄存器进行介绍。本节主要讲主机与设备模式下的寄存器使用，关于 GTO 模式下寄存器使用不具体讲解，因为 GTO A 设备相当于主机，GTO B 设备相当于设备。USB 寄存器如下表：

寄存器名称	类型	复位值	寄存器描述
USBFADDR	R/W	0x00	USB 地址
USBPOWER	R/W	0x20	USB 电源
USBTXIS	RO	0x0000	发送中断状态
USBRXIS	RO	0x0000	接收中断状态
USBTXIE	R/W	0xFFFF	发送中断使能
USBRXIE	R/W	0xFFFE	接收中断使能
USBIS	RO	0x00	USB 处理模块中断状态
USBIE	R/W	0x06	USB 处理模块中断使能
USBFRAME	RO	0x0000	USB 帧值
USBEPIDX	R/W	0x00	端点索引
USBTEST	R/W	0x00	测试模式
USBFIFO _n	R/W	0x0000.0000	端点 n FIFO
USBDEVCTL	R/W	0x80	设备控制
USBTXFIFOSZ	R/W	0x00	动态 TXFIFO 大小
USBRXFIFOSZ	R/W	0x00	动态 RXFIFO 大小
USBTXFIFOADD	R/W	0x0000	动态 TXFIFO 起始地址
USBRXFIFOADD	R/W	0x0000	动态 RXFIFO 起始地址
USBCONTIM	R/W	0x5C	USB 连接时序
USBVPLEN	R/W	0x3C	USB OTG VBUS 脉冲时序
USBFSEOF	R/W	0x77	USB 全速模式下最后的传输与帧结束时序
USBLSEOF	R/W	0x72	USB 低速模式下最后的传输与帧结束时序

USBTXFUNCADDRn	R/W	0x00	发送端点 n 功能地址
USBTXHUBADDRn	R/W	0x00	发送端点 n 集线器 (Hub) 地址
USBTXHUBPORTn	R/W	0x00	发送端点 n 集线器 (Hub) 端口
USBRXFUNCADDRn	R/W	0x00	接收端点 n 功能地址 (n 不为 0)
USBRXHUBADDRn	R/W	0x00	接收端点 n 集线器 (Hub) 地址 (n 不为 0)
USBRXHUBPORTn	R/W	0x00	接收端点 n 集线器 (Hub) 端口 (n 不为 0)
USBCSRL0	W1C	0x00	端点 0 控制和状态低字节
USBCSRH0	W1C	0x00	端点 0 控制和状态高字节
USBCOUNT0	RO	0x00	端点 0 接收字节数量
USBTYPE0	R/W	0x00	端点 0 类型
USBNAKLMT	R/W	0x00	USB NAK 限制
USBTXMAXPn	R/W	0x0000	发送端点 n 最大传输数据
USBTXC SRLn	R/W	0x00	发送端点 n 控制和状态低字节
USBTXC SRHn	R/W	0x00	发送端点 n 控制和状态高字节
USBRXMAXPn	R/W	0x0000	接收端点 n 最大传输数据
USBRXC SRLn	R/W	0x00	接收端点 n 控制和状态低字节
USBRXC SRHn	R/W	0x00	接收端点 n 控制和状态高字节
USBRXCOUNTn	RO	0x0000	端点 n 接收字节数量
USBTXTYPEn	R/W	0x00	端点 n 主机发送配置类型
USBTXINTERVALn	R/W	0x00	端点 n 主机发送间隔
USBRXTYPEn	R/W	0x00	端点 n 主机配置接收类型
USBRXINTERVALn	R/W	0x00	端点 n 主机接收巡检间隔
USBRQPKTCOUNTn	R/W	0x0000	端点 n 块传输中请求包数量
USBRXDPKTBUFDIS	R/W	0x0000	接收双包缓存禁止
USBTXDPKTBUFDIS	R/W	0x0000	发送双包缓存禁止
USBEPc	R/W	0x0000.0000	USB 外部电源控制
USBEPcRIS	RO	0x0000.0000	USB 外部电源控制原始中断状态
USBEPcIM	R/W	0x0000.0000	USB 外部电源控制中断屏蔽
USBEPcISC	R/W	0x0000.0000	USB 外部电源控制中断状态和清除
USBDRRIS	RO	0x0000.0000	USB 设备唤醒 (RESUME) 原始中断状态
USBDRIIM	R/W	0x0000.0000	USB 设备唤醒 (RESUME) 中断屏蔽
USBDRIISC	W1C	0x0000.0000	USB 设备唤醒 (RESUME) 中断状态和清除
USBGPCS	R/W	0x0000.0000	USB 通用控制和状态
USBVDC	R/W	0x0000.0000	VBUS 浮动控制
USBVDCRIS	RO	0x0000.0000	VBUS 浮动控制原始中断状态
USBVDCIM	R/W	0x0000.0000	VBUS 浮动控制中断屏蔽
USBVDCISC	R/W	0x0000.0000	VBUS 浮动控制中断状态\清除

USBIDVRIS	RO	0x0000.0000	ID 有效检测原始中断状态
USBIDVIM	R/W	0x0000.0000	ID 有效检测中断屏蔽
USBIDVISC	R/W1C	0x0000.0000	ID 有效检测中断状态与清除
USBDMASEL	R/W	0x0033.2211	DMA 选择

表 1. USB 寄存器

2.3.1 控制状态寄存器

USBFADDR, 设备地址寄存器。包含 7 位设备地址, 将通过 SET ADDRESS(USBREQ_SET_ADDRESS) 请求收到的地址 (pUSBRequest->wValue 值) 写入该寄存器。USBFADDR 寄存器描述如下表:

位	名称	类型	复位	描述
8	保留	RO	0	保持不变
[7:0]	FUNCADDR	R/W	0	设备地址

表 2. USBFADDR 寄存器

例如, 在枚举时会用到设置 USB 设备地址:

```
void USBDevAddrSet(unsigned long ulBase, unsigned long ulAddress)
{
    HWREGB(ulBase + USB_O_FADDR) = (unsigned char)ulAddress;
}
```

其中 ulBase 为设备基地址, LM3S USB 处理器一般只有一个 USB 模块, 所以只有 USB0_BASE (0x40050000); ulAddress 为主机 USBREQ_SET_ADDRESS 请求命令时 tUSBRequest.wValue 值。

USBPOWER, USB 电源控制寄存器, 8 位。主机模式下, USBPOWER 寄存器描述如下表:

位	名称	类型	复位	描述
[7:4]	保留	RO	0x02	保持不变
3	RESET	R/W	0	总线复位
2	RESUME	R/W	0	总线唤醒, 置位后 20ms 后软件清除
1	SUSPEND	R/W1S	0	总线挂起
0	PWRDNPHY	R/W	0	PHY 掉电

表 2. USBPOWER 寄存器

USBPOWER, USB 电源控制寄存器, 8 位。设备模式下, USBPOWER 寄存器描述如下表:

位	名称	类型	复位	描述
[5:4]	保留	RO	0x02	保持不变
7	ISOUP	R/W	0	ISO 传输时, TXRDY 置位时, 收到输入令牌包, 将发送 0 长度的数据包。
6	SOFTCONN	R/W	0	软件连接/断开 USB
3	RESET	RO	0	总线复位
2	RESUME	R/W	0	总线唤醒, 置位后 10ms 后软件清除
1	SUSPEND	RO	0	总线挂起
0	PWRDNPHY	R/W	0	PHY 掉电

表 3. USBPOWER 寄存器

```
#define USB_POWER_ISOUP      0x00000080 // Isochronous Update
#define USB_POWER_SOFTCONN   0x00000040 // Soft Connect/Disconnect
#define USB_POWER_RESET      0x00000008 // RESET Signaling
#define USB_POWER_RESUME     0x00000004 // RESUME Signaling
#define USB_POWER_SUSPEND    0x00000002 // SUSPEND Mode
#define USB_POWER_PWRDNPHY   0x00000001 // Power Down PHY
```

例如：在主机模式下，把 USB 总线挂起。

```
void USBHostSuspend(unsigned long ulBase)
{
    //ulBase 为设备基地址
    HWREGB(ulBase + USB_O_POWER) |= USB_POWER_SUSPEND;
}
```

例如：在主机模式下，复位 USB 总线。

```
void USBHostReset(unsigned long ulBase, tBoolean bStart)
{
    //ulBase 为设备基地址，bStart 确定复位开始/结束。
    if(bStart)
    {
        HWREGB(ulBase + USB_O_POWER) |= USB_POWER_RESET;
    }
    else
    {
        HWREGB(ulBase + USB_O_POWER) &= ~USB_POWER_RESET;
    }
}
```

例如：在主机模式下，唤醒 USB 总线，唤醒开始 20ms 后必须手动结束。

```
void USBHostResume(unsigned long ulBase, tBoolean bStart)
{
    //ulBase 为设备基地址，bStart 确定唤醒开始/结束。
    if(bStart)
    {
        HWREGB(ulBase + USB_O_POWER) |= USB_POWER_RESUME;
    }
    else
    {
        HWREGB(ulBase + USB_O_POWER) &= ~USB_POWER_RESUME;
    }
}
```

例如：在设备模式下，设备连接/断开主机。

```
void USBDevConnect(unsigned long ulBase)
{
    //ulBase 为设备基地址。
    HWREGB(ulBase + USB_O_POWER) |= USB_POWER_SOFTCONN;
}
void USBDevDisconnect(unsigned long ulBase)
{
    //ulBase 为设备基地址。
    HWREGB(ulBase + USB_O_POWER) &= (~USB_POWER_SOFTCONN);
}
```

USBFRAME，16 位只读寄存器，保存最新收到的帧编号。USBFRAME 寄存器描述如下表：

位	名称	类型	复位	描述
[15:11]	保留	RO	0	保持不变
[10:0]	FRAME	RO	0	帧编号

表 4. USBFRAME 寄存器

例如：获取最新帧编号。

```
unsigned long USBFrameNumberGet(unsigned long ulBase)
{
    //ulBase 为设备基地址。
    return(HWREGH(ulBase + USB_O_FRAME));
}
```

USBTEST, 测试模式, 回应 USBREQ_SET_FEATURE 的 USB_FEATURE_TEST_MODE 命令, 使 USB 控制器进入测试模式, 任何时候, 只能有一位被设置, 但不用于正常操作。在主机模式下, USBTEST 寄存器描述如下表:

位	名称	类型	复位	描述
[4:0]	保留	RO	0	保持不变
7	FORCEH	R/W	0	强制为主机模式
6	FIFOACC	R/WIS	0	将端点 0 的 TxFIFO 数据传送到 RxFIFO
5	FORCEFS	R/W	0	强制全速模式

表 5. USBTEST 寄存器

在设备模式下, USBTEST 寄存器描述如下表:

位	名称	类型	复位	描述
[4:0]、7	保留	RO	0	保持不变
6	FIFOACC	R/WIS	0	将端点 0 的 TxFIFO 数据传送到 RxFIFO
5	FORCEFS	R/W	0	强制全速模式

表 6. USBTEST 寄存器

USBDEVCTL, USB 设备控制器, 提供 USB 控制器当前状态信息, 可指示接入设备是全速还是低速。USBDEVCTL 寄存器描述如下表:

位	名称	类型	复位	描述
7	DEV	RO	1	设备模式, OTG A 端。
6	FSDEV	RO	0	全速设备
5	LSDEV	RO	0	低速设备
[4:3]	VBUS	RO	0	VBUS 电平
2	HOST	RO	0	Host 主机模式
1	HOSTREQ	R/W	0	主机请求
0	SESSION	R/W	0	会话开始/结束

表 7. USBDEVCTL 寄存器

```
#define USB_DEVCTL_DEV      0x00000080 // Device Mode
#define USB_DEVCTL_FSDEV    0x00000040 // Full-Speed Device Detected
#define USB_DEVCTL_LSDEV    0x00000020 // Low-Speed Device Detected
#define USB_DEVCTL_VBUS_M   0x00000018 // VBUS Level
#define USB_DEVCTL_VBUS_NONE 0x00000000 // Below SessionEnd
#define USB_DEVCTL_VBUS_SEND 0x00000008 // Above SessionEnd, below AValid
#define USB_DEVCTL_VBUS_AVALID 0x00000010 // Above AValid, below VBUSValid
#define USB_DEVCTL_VBUS_VALID 0x00000018 // Above VBUSValid
#define USB_DEVCTL_HOST     0x00000004 // Host Mode
#define USB_DEVCTL_HOSTREQ  0x00000002 // Host Request
#define USB_DEVCTL_SESSION  0x00000001 // Session Start/End
```

例如: 获取当前 USB 工作模式。

```
unsigned long USBModeGet(unsigned long ulBase)
{
    return(HWREGB(ulBase + USB_0_DEVCTL) &
           (USB_DEVCTL_DEV | USB_DEVCTL_HOST | USB_DEVCTL_SESSION |
            USB_DEVCTL_VBUS_M));
}
```

返回下面参数:

```
#define USB_DUAL_MODE_HOST    0x00000001
#define USB_DUAL_MODE_DEVICE  0x00000081
#define USB_DUAL_MODE_NONE    0x00000080
#define USB_OTG_MODE_ASIDE_HOST 0x0000001d
```

```
#define USB_OTG_MODE_ASIDE_NPWR 0x00000001
#define USB_OTG_MODE_ASIDE_SESS 0x00000009
#define USB_OTG_MODE_ASIDE_AVAL 0x00000011
#define USB_OTG_MODE_ASIDE_DEV 0x00000019
#define USB_OTG_MODE_BSIDE_HOST 0x0000009d
#define USB_OTG_MODE_BSIDE_DEV 0x00000099
#define USB_OTG_MODE_BSIDE_NPWR 0x00000081
#define USB_OTG_MODE_NONE 0x00000080
```

USBCONTIM，连接时序控制寄存器，用于控制 USB 连接。USBCONTIM 寄存器描述如下表：

位	名称	类型	复位	描述
[7:4]	WTCON	R/W	0x5	连接等待
[3:0]	WTID	R/W	0xC	ID 等待

表 8. USBCONTIM 寄存器

WTCON，此位域可按需求配置等待延时，满足连接/断开的滤波需求；单位为 533.3ns，复位后为 0x5，默认为 2.667μs。

WTID，此位域配置 ID 等待延时，等待 ID 值有效时才使能 OTG 的 ID 检测。单位为 4.369 ms，复位后为 0xC，默认为 52.43 ms。

USBVPLEN，VBUS 脉冲时序配置寄存器，控制 VBUS 脉冲充电的持续时间。USBVPLEN 寄存器描述如下表：

位	名称	类型	复位	描述
[7:0]	VPLEN	R/W	0x3C	VBUS 脉冲宽度

表 9. USBVPLEN 寄存器

VPLEN，用于配置 VBUS 脉冲充电的持续时间，单位 546.1 μs，默认为 32.77 ms。

USBFSEOF，用于配置全速模式下最后传输开始与帧结束(EOF)之间的最小时间间隔。USBFSEOF 寄存器描述如下表：

位	名称	类型	复位	描述
[7:0]	FSEOFG	R/W	0x77	全速模式帧间隙

表 10. USBFSEOF 寄存器

FSEOFG，在全速传输中用于配置最后的传输与帧结束之间的最小时间间隔，单位 533.3ns，默认为 63.46 μs。

USBLSEOF，用于配置低速模式下最后传输开始与帧结束(EOF)之间的最小时间间隔。USBLSEOF 寄存器描述如下表：

位	名称	类型	复位	描述
[7:0]	LSEOFG	R/W	0x72	低速模式帧间隙

表 11. USBLSEOF 寄存器

FSEOFG，在低速传输中用于配置最后的传输与帧结束之间的最小时间间隔，单位 1.067 μs，默认为 121.6 μs。

注意：USBCONTIM、USBVPLEN、USBFSEOF、USBLSEOF 一般不用配置，保持默认情况就能满足 USB 通信。

USBGPCS，USB 通用控制状态寄存器，提供内部 ID 信号的状态。

USBGPCS 寄存器描述如下表：

位	名称	类型	复位	描述
[31:2]	保留	RO	0	保持不变

1	DEVMODOTG	R/W	0	使能 ID 控制模式
0	DEVMOD	R/W	0	模式控制

表 12. USBGPCS 寄存器

```
#define USB_GPCS_DEVMODOTG    0x00000002 // Enable Device Mode
#define USB_GPCS_DEVMOD      0x00000001 // Device Mode
```

例如：设置主机模式。

```
void USBHostMode(unsigned long ulBase)
{
    //ulBase 为设备基地址。
    HWREGB(ulBase + USB_O_GPCS) &= ~(USB_GPCS_DEVMOD);
}
```

2.3.2 中断控制

USBTXIS，发送中断状态寄存器，16 位只读寄存器，读取数据时，清除相应标志位。

USBTXIS 寄存器描述如下表：

位	名称	类型	复位	描述
15..0	EPn	RO	0	端点发送中断标志

表 13. USBTXIS 寄存器

USBRIXS，接收中断状态寄存器。USBRIXS 寄存器描述如下表：

位	名称	类型	复位	描述
15..1	EPn	R/W	0	端点接收中断标志

表 14. USBRIXS 寄存器

USBTXIE，发送中断使能寄存器，16 位只读寄存器，读取数据时，清除相应标志位。

USBTXIE 寄存器描述如下表：

位	名称	类型	复位	描述
15..0	EPn	RO	0	端点发送中断标志

表 15. USBTXIE 寄存器

USBRIXE，接收中断使能寄存器。USBRIXE 寄存器描述如下表：

位	名称	类型	复位	描述
15..1	EPn	R/W	0	端点接收中断标志

表 16. USBRIXE 寄存器

USBIS，通用中断状态寄存器，8 位只读寄存器，读取数据时，清除相应标志位。

USBIS 寄存器描述如下表：

位	名称	类型	复位	描述
7	VBUSERR	RO	0	VBUS 中断（只有主机模式使用）
6	SESREQ	RO	0	会话请求中断（只有主机模式使用）
5	DISCON	RO	0	连接断开中断
4	CONN	RO	0	连接中断
3	SOF	RO	0	帧起始中断
2	BABBLE	RO	0	Babble 中断
1	RESUME	RO	0	唤醒中断
0	SUSPEND	RO	0	挂起中断（只有设备模式使用）

表 17. USBIS 寄存器

USBIE，通用中断使能寄存器，USBIE 寄存器描述如下表：

位	名称	类型	复位	描述
---	----	----	----	----

7	VBUSERR	RO	0	VBUS 中断使能（只有主机模式使用）
6	SESREQ	RO	0	会话请求中断使能（只有主机模式使用）
5	DISCON	RO	0	连接断开中断使能
4	CONN	RO	0	连接中断使能
3	SOF	RO	0	帧起始中断使能
2	BABBLE	RO	0	Babble 中断使能
1	RESUME	RO	0	唤醒中断使能
0	SUSPEND	RO	0	挂起中断使能（只有设备模式使用）

表 18. USBIE 寄存器

此外还有 USBEPC、USBEPKRIS、USBEPK、USBEPKRIS、USBEPKIM、USBEPKISC 电源管理中断；USBIDVISC、USBIDVIM、USBIDVRIS 等 ID 检测中断控制；USBDKRIS、USBDKRIM、USBDKRISC 唤醒中断控制；USBVDC、USBVDCRIS、USBVDCIM、USBVDCISC 的 VBUS Droop 控制中断。这些中断使用较少，不在此详细描述。

例如，写一组函数，控制、管理上面中断。

USBIntEnable()、USBIntDisable()的 ulFlags 参数， USBIntStatus()返回参数：

```
#define USB_INTCTRL_ALL      0x000003FF // All control interrupt sources
#define USB_INTCTRL_STATUS   0x000000FF // Status Interrupts
#define USB_INTCTRL_VBUS_ERR 0x00000080 // VBUS Error
#define USB_INTCTRL_SESSION   0x00000040 // Session Start Detected
#define USB_INTCTRL_SESSION_END 0x00000040 // Session End Detected
#define USB_INTCTRL_DISCONNECT 0x00000020 // Disconnect Detected
#define USB_INTCTRL_CONNECT   0x00000010 // Device Connect Detected
#define USB_INTCTRL_SOF       0x00000008 // Start of Frame Detected
#define USB_INTCTRL_BABBLE    0x00000004 // Babble signaled
#define USB_INTCTRL_RESET     0x00000004 // Reset signaled
#define USB_INTCTRL_RESUME    0x00000002 // Resume detected
#define USB_INTCTRL_SUSPEND   0x00000001 // Suspend detected
#define USB_INTCTRL_MODE_DETECT 0x00000200 // Mode value valid
#define USB_INTCTRL_POWER_FAULT 0x00000100 // Power Fault detected

//端点中断控制
#define USB_INTEP_ALL        0xFFFFFFFF // Host IN Interrupts
#define USB_INTEP_HOST_IN    0xFFFFE000 // Host IN Interrupts
#define USB_INTEP_HOST_IN_15 0x80000000 // Endpoint 15 Host IN Interrupt
#define USB_INTEP_HOST_IN_14 0x40000000 // Endpoint 14 Host IN Interrupt
#define USB_INTEP_HOST_IN_13 0x20000000 // Endpoint 13 Host IN Interrupt
#define USB_INTEP_HOST_IN_12 0x10000000 // Endpoint 12 Host IN Interrupt
#define USB_INTEP_HOST_IN_11 0x08000000 // Endpoint 11 Host IN Interrupt
#define USB_INTEP_HOST_IN_10 0x04000000 // Endpoint 10 Host IN Interrupt
#define USB_INTEP_HOST_IN_9  0x02000000 // Endpoint 9 Host IN Interrupt
#define USB_INTEP_HOST_IN_8  0x01000000 // Endpoint 8 Host IN Interrupt
#define USB_INTEP_HOST_IN_7  0x00800000 // Endpoint 7 Host IN Interrupt
#define USB_INTEP_HOST_IN_6  0x00400000 // Endpoint 6 Host IN Interrupt
#define USB_INTEP_HOST_IN_5  0x00200000 // Endpoint 5 Host IN Interrupt
#define USB_INTEP_HOST_IN_4  0x00100000 // Endpoint 4 Host IN Interrupt
#define USB_INTEP_HOST_IN_3  0x00080000 // Endpoint 3 Host IN Interrupt
#define USB_INTEP_HOST_IN_2  0x00040000 // Endpoint 2 Host IN Interrupt
#define USB_INTEP_HOST_IN_1  0x00020000 // Endpoint 1 Host IN Interrupt
#define USB_INTEP_DEV_OUT    0xFFFFE000 // Device OUT Interrupts
#define USB_INTEP_DEV_OUT_15 0x80000000 // Endpoint 15 Device OUT Interrupt
#define USB_INTEP_DEV_OUT_14 0x40000000 // Endpoint 14 Device OUT Interrupt
#define USB_INTEP_DEV_OUT_13 0x20000000 // Endpoint 13 Device OUT Interrupt
#define USB_INTEP_DEV_OUT_12 0x10000000 // Endpoint 12 Device OUT Interrupt
#define USB_INTEP_DEV_OUT_11 0x08000000 // Endpoint 11 Device OUT Interrupt
#define USB_INTEP_DEV_OUT_10 0x04000000 // Endpoint 10 Device OUT Interrupt
```



```

#define USB_INTEP_DEV_OUT_9    0x02000000 // Endpoint 9 Device OUT Interrupt
#define USB_INTEP_DEV_OUT_8    0x01000000 // Endpoint 8 Device OUT Interrupt
#define USB_INTEP_DEV_OUT_7    0x00800000 // Endpoint 7 Device OUT Interrupt
#define USB_INTEP_DEV_OUT_6    0x00400000 // Endpoint 6 Device OUT Interrupt
#define USB_INTEP_DEV_OUT_5    0x00200000 // Endpoint 5 Device OUT Interrupt
#define USB_INTEP_DEV_OUT_4    0x00100000 // Endpoint 4 Device OUT Interrupt
#define USB_INTEP_DEV_OUT_3    0x00080000 // Endpoint 3 Device OUT Interrupt
#define USB_INTEP_DEV_OUT_2    0x00040000 // Endpoint 2 Device OUT Interrupt
#define USB_INTEP_DEV_OUT_1    0x00020000 // Endpoint 1 Device OUT Interrupt
#define USB_INTEP_HOST_OUT     0x0000FFFE // Host OUT Interrupts
#define USB_INTEP_HOST_OUT_15  0x00008000 // Endpoint 15 Host OUT Interrupt
#define USB_INTEP_HOST_OUT_14  0x00004000 // Endpoint 14 Host OUT Interrupt
#define USB_INTEP_HOST_OUT_13  0x00002000 // Endpoint 13 Host OUT Interrupt
#define USB_INTEP_HOST_OUT_12  0x00001000 // Endpoint 12 Host OUT Interrupt
#define USB_INTEP_HOST_OUT_11  0x00000800 // Endpoint 11 Host OUT Interrupt
#define USB_INTEP_HOST_OUT_10  0x00000400 // Endpoint 10 Host OUT Interrupt
#define USB_INTEP_HOST_OUT_9   0x00000200 // Endpoint 9 Host OUT Interrupt
#define USB_INTEP_HOST_OUT_8   0x00000100 // Endpoint 8 Host OUT Interrupt
#define USB_INTEP_HOST_OUT_7   0x00000080 // Endpoint 7 Host OUT Interrupt
#define USB_INTEP_HOST_OUT_6   0x00000040 // Endpoint 6 Host OUT Interrupt
#define USB_INTEP_HOST_OUT_5   0x00000020 // Endpoint 5 Host OUT Interrupt
#define USB_INTEP_HOST_OUT_4   0x00000010 // Endpoint 4 Host OUT Interrupt
#define USB_INTEP_HOST_OUT_3   0x00000008 // Endpoint 3 Host OUT Interrupt
#define USB_INTEP_HOST_OUT_2   0x00000004 // Endpoint 2 Host OUT Interrupt
#define USB_INTEP_HOST_OUT_1   0x00000002 // Endpoint 1 Host OUT Interrupt
#define USB_INTEP_DEV_IN       0x0000FFFE // Device IN Interrupts
#define USB_INTEP_DEV_IN_15    0x00008000 // Endpoint 15 Device IN Interrupt
#define USB_INTEP_DEV_IN_14    0x00004000 // Endpoint 14 Device IN Interrupt
#define USB_INTEP_DEV_IN_13    0x00002000 // Endpoint 13 Device IN Interrupt
#define USB_INTEP_DEV_IN_12    0x00001000 // Endpoint 12 Device IN Interrupt
#define USB_INTEP_DEV_IN_11    0x00000800 // Endpoint 11 Device IN Interrupt
#define USB_INTEP_DEV_IN_10    0x00000400 // Endpoint 10 Device IN Interrupt
#define USB_INTEP_DEV_IN_9     0x00000200 // Endpoint 9 Device IN Interrupt
#define USB_INTEP_DEV_IN_8     0x00000100 // Endpoint 8 Device IN Interrupt
#define USB_INTEP_DEV_IN_7     0x00000080 // Endpoint 7 Device IN Interrupt
#define USB_INTEP_DEV_IN_6     0x00000040 // Endpoint 6 Device IN Interrupt
#define USB_INTEP_DEV_IN_5     0x00000020 // Endpoint 5 Device IN Interrupt
#define USB_INTEP_DEV_IN_4     0x00000010 // Endpoint 4 Device IN Interrupt
#define USB_INTEP_DEV_IN_3     0x00000008 // Endpoint 3 Device IN Interrupt
#define USB_INTEP_DEV_IN_2     0x00000004 // Endpoint 2 Device IN Interrupt
#define USB_INTEP_DEV_IN_1     0x00000002 // Endpoint 1 Device IN Interrupt
#define USB_INTEP_0             0x00000001 // Endpoint 0 Interrupt

```

中断使能函数:

```

void USBIntEnable(unsigned long ulBase, unsigned long ulFlags)
{
    //发送中断控制
    if(ulFlags & (USB_INT_HOST_OUT | USB_INT_DEV_IN | USB_INT_EP0))
    {
        HWREGH(ulBase + USB_O_TXIE) |=
            ulFlags & (USB_INT_HOST_OUT | USB_INT_DEV_IN | USB_INT_EP0);
    }
    //接收中断控制
    if(ulFlags & (USB_INT_HOST_IN | USB_INT_DEV_OUT))
    {
        HWREGH(ulBase + USB_O_RXIE) |=
            ((ulFlags & (USB_INT_HOST_IN | USB_INT_DEV_OUT)) >>
             USB_INT_RX_SHIFT);
    }
    //通用中断控制
}

```

```

if(ulFlags & USB_INT_STATUS)
{
    HWREGB(ulBase + USB_O_IE) |=
        (ulFlags & USB_INT_STATUS) >> USB_INT_STATUS_SHIFT;
}
//电源中断控制
if(ulFlags & USB_INT_POWER_FAULT)
{
    HWREG(ulBase + USB_O_EPCIM) = USB_EPCIM_PF;
}
//ID 中断控制
if(ulFlags & USB_INT_MODE_DETECT)
{
    HWREG(USB0_BASE + USB_O_IDVIM) = USB_IDVIM_ID;
}
}

```

中断禁止函数:

```

void USBIntDisable(unsigned long ulBase, unsigned long ulFlags)
{
    //发送中断控制
    if(ulFlags & (USB_INT_HOST_OUT | USB_INT_DEV_IN | USB_INT_EP0))
    {
        HWREGH(ulBase + USB_O_TXIE) &=
            ~(ulFlags & (USB_INT_HOST_OUT | USB_INT_DEV_IN | USB_INT_EP0));
    }
    //接收中断控制
    if(ulFlags & (USB_INT_HOST_IN | USB_INT_DEV_OUT))
    {
        HWREGH(ulBase + USB_O_RXIE) &=
            ~((ulFlags & (USB_INT_HOST_IN | USB_INT_DEV_OUT)) >>
                USB_INT_RX_SHIFT);
    }
    //通用中断控制
    if(ulFlags & USB_INT_STATUS)
    {
        HWREGB(ulBase + USB_O_IE) &=
            ~(ulFlags & USB_INT_STATUS) >> USB_INT_STATUS_SHIFT;
    }
    //电源中断控制
    if(ulFlags & USB_INT_POWER_FAULT)
    {
        HWREG(ulBase + USB_O_EPCIM) = 0;
    }
    //ID 中断控制
    if(ulFlags & USB_INT_MODE_DETECT)
    {
        HWREG(USB0_BASE + USB_O_IDVIM) = 0;
    }
}

```

获取中断标志并清除函数:

```

unsigned long USBIntStatus(unsigned long ulBase)
{
    unsigned long ulStatus;
    ulStatus = (HWREGB(ulBase + USB_O_TXIS));
    ulStatus |= (HWREGB(ulBase + USB_O_RXIS) << USB_INT_RX_SHIFT);
    ulStatus |= (HWREGB(ulBase + USB_O_IS) << USB_INT_STATUS_SHIFT);
    if(HWREG(ulBase + USB_O_EPCISC) & USB_EPCISC_PF)
    {
        ulStatus |= USB_INT_POWER_FAULT;
    }
}

```

```

        HWREGB(ulBase + USB_O_EPDISC) |= USB_EPDISC_PF;
    }

    if(HWREG(USB0_BASE + USB_O_IDVISC) & USB_IDVRIS_ID)
    {
        ulStatus |= USB_INT_MODE_DETECT;
        HWREG(USB0_BASE + USB_O_IDVISC) |= USB_IDVRIS_ID;
    }
    return(ulStatus);
}

端点中断使用较为频繁，可单独控制。
void USBIntDisableEndpoint(unsigned long ulBase, unsigned long ulFlags)
{
    HWREGH(ulBase + USB_O_TXIE) &=
        ~(ulFlags & (USB_INTEP_HOST_OUT | USB_INTEP_DEV_IN | USB_INTEP_0));
    HWREGH(ulBase + USB_O_RXIE) &=
        ~((ulFlags & (USB_INTEP_HOST_IN | USB_INTEP_DEV_OUT)) >>
            USB_INTEP_RX_SHIFT);
}

void USBIntEnableEndpoint(unsigned long ulBase, unsigned long ulFlags)
{
    HWREGH(ulBase + USB_O_TXIE) |=
        ulFlags & (USB_INTEP_HOST_OUT | USB_INTEP_DEV_IN | USB_INTEP_0);
    HWREGH(ulBase + USB_O_RXIE) |=
        ((ulFlags & (USB_INTEP_HOST_IN | USB_INTEP_DEV_OUT)) >>
            USB_INTEP_RX_SHIFT);
}

unsigned long USBIntStatusEndpoint(unsigned long ulBase)
{
    unsigned long ulStatus;
    ulStatus = HWREGH(ulBase + USB_O_TXIS);
    ulStatus |= (HWREGH(ulBase + USB_O_RXIS) << USB_INTEP_RX_SHIFT);
    return(ulStatus);
}

```

2.3.3 端点寄存器

USBEPIDX, USB 端点索引寄存器。设置端点 FIFO 大小和起始地址要配合 USBEPIDX 使用。USBEPIDX 寄存器描述如下表：

位	名称	类型	复位	描述
[7:4]	保留	RO	0	保留
[3:0]	EPIDX	R/W	0	端点索引

表 19. USBEPIDX 寄存器

USBRXFIFOSZ、USBTXFIFOSZ，USB 接收/发送 FIFO 大小寄存器。USBRXFIFOSZ、USBTXFIFOSZ 寄存器描述如下表：

位	名称	类型	复位	描述
[7:5]	保留	RO	0	保留
4	DPB	R/W	0	双包缓存
[3:0]	SIZE	R/W	0	最大包 8*(2^SIZE)

表 20. USBRXFIFOSZ、USBTXFIFOSZ 寄存器

USBRXFIFOADD、USBTXFIFOADD，USB 接收/发送 FIFO 首地址寄存器。USBRXFIFOADD、USBTXFIFOADD 寄存器描述如下表：

位	名称	类型	复位	描述
[15: 9]	保留	RO	0	保留

[8:0]	ADDR	R/W	0	最大包 8*SIZE
-------	------	-----	---	------------

表 21. USBRXFIFOADD、USBTXFIFOADD 寄存器

注意：访问要用到索引寄存器的 FIFO 寄存器只有 USBRXFIFOSZ、USBTXFIFOSZ、USBRXFIFOADD、USBTXFIFOADD 四个。

例如：写一个函数，控制端点的 FIFO。

```
static void USBIndexWrite(unsigned long ulBase, unsigned long ulEndpoint,
                          unsigned long ulIndexedReg, unsigned long ulValue,
                          unsigned long ulSize)
{
    unsigned long ulIndex;
    // 保存当前索引寄存器值.
    ulIndex = HWREGB(ulBase + USB_0_EPIDX);
    // 写新值到端点索引寄存器
    HWREGB(ulBase + USB_0_EPIDX) = ulEndpoint;
    //根据寄存器大小写入新值到 FIFO 寄存器。
    if(ulSize == 1)
    {
        HWREGB(ulBase + ulIndexedReg) = ulValue;
    }
    else
    {
        HWREGB(ulBase + ulIndexedReg) = ulValue;
    }
    //恢复以前索引寄存器值。
    HWREGB(ulBase + USB_0_EPIDX) = ulIndex;
}
```

USBTXFUNCADDRn，发送端点功能地址寄存器。在主机模式下，用于设置端点 n 访问的目标地址。USBTXFUNCADDRn 寄存器描述如下表：

位	名称	类型	复位	描述
7	保留	RO	0	保留
[6:0]	ADDR	R/W	0	设备总线地址

表 22 USBTXFUNCADDRn 寄存器

USBTXHUBPORTn，发送端点 n 集线器端口号寄存器。USBTXHUBPORTn 寄存器描述如下表：

位	名称	类型	复位	描述
7	保留	RO	0	保留
[6:0]	ADDR	R/W	0	集线器端口

表 23 USBTXHUBPORTn 寄存器

USBTXHUBADDRn，发送端点 n 集线器地址寄存器，USBTXHUBADDRn 寄存器描述如下表：

位	名称	类型	复位	描述
7	MULTTRAN	RO	0	多路开关
[6:0]	ADDR	R/W	0	集线器地址

表 24 USBTXHUBADDRn 寄存器

USBRXFUNCADDRn，接收端点功能地址寄存器。在主机模式下，用于设置端点 n 访问的目标地址。USBRXFUNCADDRn 寄存器描述如下表：

位	名称	类型	复位	描述
7	保留	RO	0	保留
[6:0]	ADDR	R/W	0	设备总线地址

表 25 USBRXFUNCADDRn 寄存器

USBRXHUBPORTn，接收端点 n 集线器端口号寄存器。USBRXHUBPORTn 寄存器描述如下表：

位	名称	类型	复位	描述
7	保留	RO	0	保留
[6:0]	ADDR	R/W	0	集线器端口

表 26 USBRXHUBPORTn 寄存器

USBXHXUBADDRn, 接收端点 n 集线器地址寄存器, USBXHXUBADDRn 寄存器描述如下表:

位	名称	类型	复位	描述
7	MULTTRAN	RO	0	多路开关
[6:0]	ADDR	R/W	0	集线器地址

表 27 USBXHXUBADDRn 寄存器

注意: USBTXFUNCADDR0、USBTXHUBPORT0 、USBTXHUBADDR0 同时用于端点 0 的接收和发送。

例如: 写一个函数, 主机端点访问某个设备地址。

USBHostAddrSet() 的 ulEndpoint 参数:

```
#define USB_EP_0          0x00000000 // Endpoint 0
#define USB_EP_1          0x00000010 // Endpoint 1
#define USB_EP_2          0x00000020 // Endpoint 2
#define USB_EP_3          0x00000030 // Endpoint 3
#define USB_EP_4          0x00000040 // Endpoint 4
#define USB_EP_5          0x00000050 // Endpoint 5
#define USB_EP_6          0x00000060 // Endpoint 6
#define USB_EP_7          0x00000070 // Endpoint 7
#define USB_EP_8          0x00000080 // Endpoint 8
#define USB_EP_9          0x00000090 // Endpoint 9
#define USB_EP_10         0x000000A0 // Endpoint 10
#define USB_EP_11         0x000000B0 // Endpoint 11
#define USB_EP_12         0x000000C0 // Endpoint 12
#define USB_EP_13         0x000000D0 // Endpoint 13
#define USB_EP_14         0x000000E0 // Endpoint 14
#define USB_EP_15         0x000000F0 // Endpoint 15
#define NUM_USB_EP        16          // Number of supported endpoints

void USBHostAddrSet(unsigned long ulBase, unsigned long ulEndpoint,
                    unsigned long ulAddr, unsigned long ulFlags)
{
    //根据 ulFlags 设置发送还是接地址
    if(ulFlags & USB_EP_HOST_OUT)
    {
        HWREGB(ulBase + USB_O_TXFUNCADDR0 + (ulEndpoint >> 1)) = ulAddr;
    }
    else
    {
        HWREGB(ulBase + USB_O_TXFUNCADDR0 + 4 + (ulEndpoint >> 1)) = ulAddr;
    }
}
```

例如: 写一个函数, 主机端点通过集线器访问某个设备地址。

```
void USBHostHubAddrSet(unsigned long ulBase, unsigned long ulEndpoint,
                       unsigned long ulAddr, unsigned long ulFlags)
{
    ///根据 ulFlags 设置发送还是接地址
    if(ulFlags & USB_EP_HOST_OUT)
    {
        HWREGB(ulBase + USB_O_TXHUBADDR0 + (ulEndpoint >> 1)) = ulAddr;
    }
    else
    {

```

```

        HWREGB(ulBase + USB_O_TXHUBADDR0 + 4 + (ulEndpoint >> 1)) = ulAddr;
    }
}

```

USBTXMAXPn, 发送端点 n 最大传输数据寄存器, 定义发送端点单次可传输的最大数据长度。USBTXMAXPn 寄存器描述如下表:

位	名称	类型	复位	描述
[15:11]	保留	RO	0	保留
[10:0]	MAXLOAD	R/W	0	单次传输数据最大字节数

表 28 USBTXMAXPn 寄存器

USBRCOUNTn, USB 端点 n 接收字节数寄存器, 从 RXFIFO 中读出数据的字节数。USBRCOUNTn 寄存器描述如下表:

位	名称	类型	复位	描述
[15:13]	保留	RO	0	保留
[12:0]	COUNT	R/W	0	接收字节数

表 29 USBRCOUNTn 寄存器

USBTXTYPEn, 主机发送类型寄存器, 配置发送端点的目标端点号, 传输协议, 以及其运行速度。USBTXTYPEn 寄存器描述如下表:

位	名称	类型	复位	描述
[7:6]	SPEED	R/W	0	运行速度
[5:4]	PROTO	R/W	0	协议
[3:0]	TEP	R/W	0	目标端点号

表 30 USBTXTYPEn 寄存器

USBTXTYPEn, 主机发送类型寄存器, 配置发送端点的目标端点号, 传输协议, 以及其运行速度。USBTXTYPEn 寄存器描述如下表:

位	名称	类型	复位	描述
[7:6]	SPEED	R/W	0	运行速度
[5:4]	PROTO	R/W	0	协议
[3:0]	TEP	R/W	0	目标端点号

表 31 USBTXTYPEn 寄存器

USBTXINTERVALn, 主机发送间隔寄存器。USBTXINTERVALn 寄存器描述如下表:

位	名称	类型	复位	描述
[7:0]	TXPOLL / NAKLMT	R/W	0	NAK 超时时间间隔

表 32 USBTXINTERVALn 寄存器

USBRXTYPEn, 主机接收类型寄存器, 配置接收端点的目标端点号, 传输协议, 以及其运行速度。USBRXTYPEn 寄存器描述如下表:

位	名称	类型	复位	描述
[7:6]	SPEED	R/W	0	运行速度
[5:4]	PROTO	R/W	0	协议
[3:0]	TEP	R/W	0	目标端点号

表 33 USBRXTYPEn 寄存器

USBRXINTERVALn, 主机接收间隔寄存器。USBRXINTERVALn 寄存器描述如下表:

位	名称	类型	复位	描述
[7:0]	TXPOLL / NAKLMT	R/W	0	NAK 超时时间间隔

表 34 USBRXINTERVALn 寄存器

USBRQPKTCOUNTn, 块传输请求包数量寄存器。USBRQPKTCOUNTn 寄存器描述如下表:

位	名称	类型	复位	描述
[15:0]	COUNT	R/W	0	块传输包数量

表 35 USBRQPKTCOUNTn 寄存器

USBXDPKTBUDIS，接收双包缓存禁止寄存器，USBXDPKTBUDIS 寄存器描述如下表：

位	名称	类型	复位	描述
15..0	EPn	R/W	0	接收双包缓存禁止

表 36 USBXDPKTBUDIS 寄存器

USBTXDPKTBUDIS，发送双包缓存禁止寄存器，USBTXDPKTBUDIS 寄存器描述如下表：

位	名称	类型	复位	描述
15..0	EPn	R/W	0	发送双包缓存禁止

表 37 USBTXDPKTBUDIS 寄存器

USBFIFOn，FIFO 端点寄存器，主要进行 FIFO 访问。写操作，将向 TXFIFO 中写入数据；读操作，将从 RXFIFO 中读出数据。USBFIFOn 寄存器描述如下表：

位	名称	类型	复位	描述
[31:0]	EPDATA	R/W	0	端点数据

表 38 USBFIFOn 寄存器

USBCSRL0，端点 0 控制和状态低字节寄存器，为端点 0 提供控制和状态位。USBCSRL0 寄存器描述如下表：

位	名称	类型	复位	描述	主机/设备
7	NAKTO/SETENDC	R/W	0	NAK 超时/SETEND 清 0	
6	STATUS/RXRDYC	R/W	0	状态包/清 RXRDY 位	
5	REQPKT/STALL	R/W	0	请求包/发送 STALL 握手	
4	ERROR/SETEND	R/W	0	错误/Setup End	
3	SETUP/DATAEND	R/W	0	建立令牌包/数据结束	
2	STALLED	R/W	0	端点挂起	
1	TXRDY	R/W	0	发送包准备好	
0	RXRDY	R/W	0	接收包准备好	

表 39 USBCSRL0 寄存器

USBCSRH0，端点 0 控制和状态高字节寄存器，为端点 0 提供控制和状态位。USBCSRH0 寄存器描述如下表：

位	名称	类型	复位	描述
[7:3]	保留	RO	0	保留
2	DTWE	R/W	0	数据切换写使能（只有主机模式）
1	DT	R/W	0	数据切换（只有主机模式）
0	FLUSH	R/W	0	清空 FIFO

表 40 USBCSRH0 寄存器

USBTXCSSLn，发送端点 n(非端点 0)控制和状态低字节寄存器，为端点 n 提供控制和状态位。USBTXCSSLn 寄存器描述如下表：

位	名称	类型	复位	描述	主机/设备
7	NAKTO	R/W	0	NAK 超时（只有主机）	
6	CLRDT	R/W	0	清除数据转换	
5	STALLED	R/W	0	端点挂起	
4	SETUP/STALL	R/W	0	建立令牌包/发送 STALL	

3	FLUSH	R/W	0	清空 FIFO
2	ERROR/UNDRN	R/W	0	错误/欠运转
1	FIFONE	R/W	0	FIFO 不空
0	TXRDY	R/W	0	发送包准备好

表 41 USBTXCSRn 寄存器

USBTXCSRn, 发送端点 n(非端点 0)控制和状态高字节寄存器, 为端点 n 提供控制和状态位。USBTXCSRn 寄存器描述如下表:

位	名称	类型	复位	描述
7	AUTOSET	R/W	0	自动置位
6	ISO	R/W	0	ISO 传输 (只有设备模式)
5	MODE	R/W	0	模式
4	DMAEN	R/W	0	DMA 请求使能
3	FDT	R/W	0	强制数据切换
2	DMAMOD	R/W	0	DMA 请求模式
1	DTWE	R/W	0	数据切换写使能 (只有主机模式)
0	DT	R/W	0	数据切换 (只有主机模式)

表 42 USBRXCSRn 寄存器

USBRXCSRn, 接收端点 n 控制和状态低字节寄存器, USBRXCSRn 寄存器描述如下表:

位	名称	类型	复位	描述 主机/设备
7	CLRDT	R/W	0	清除数据转换
6	STALLED	R/W	0	端点挂起
5	REQPKT/ STALL	R/W	0	请求包/发送 STALL 握手
4	FLUSH	R/W	0	清空 FIFO
3	DATAERR\NAKTO/ DATAERR	R/W	0	数据错误\NAK 超时 /数据错误
2	ERROR/OVER	R/W	0	错误/Overrun
1	FULL	R/W	0	错误
0	RXRDY	R/W	0	接收包准备好

表 43 USBRXCSRn 寄存器

USBRXCSRn, 接收端点 n 控制和状态高字节寄存器, 为接收端点提供额外的控制和状态位。USBRXCSRn 寄存器描述如下表:

位	名称	类型	复位	描述
7	AUTOCL	R/W	0	自动清除
6	AUTORQ	R/W	0	自动请求
5	DMAEN	R/W	0	DMA 请求使能
4	PIDERR	R/W	0	PID 错误
3	DMAMOD	R/W	0	DMA 请求模式
2	DTWE	R/W	0	数据切换写使能 (只有主机模式)
1	DT	R/W	0	数据切换 (只有主机模式)
0	保留	RO	0	保留

表 44 USBRXCSRn 寄存器

例如: 写一个函数, 配置端点。

USBHostEndpointConfig() 和 USBDevEndpointConfigSet() 的 ulFlags 参数:

```
#define USB_EP_AUTO_SET      0x00000001 // Auto set feature enabled
#define USB_EP_AUTO_REQUEST 0x00000002 // Auto request feature enabled
```



```

#define USB_EP_AUTO_CLEAR      0x00000004 // Auto clear feature enabled
#define USB_EP_DMA_MODE_0      0x00000008 // Enable DMA access using mode 0
#define USB_EP_DMA_MODE_1      0x00000010 // Enable DMA access using mode 1
#define USB_EP_MODE_ISOC       0x00000000 // Isochronous endpoint
#define USB_EP_MODE_BULK       0x00000100 // Bulk endpoint
#define USB_EP_MODE_INT        0x00000200 // Interrupt endpoint
#define USB_EP_MODE_CTRL       0x00000300 // Control endpoint
#define USB_EP_MODE_MASK       0x00000300 // Mode Mask
#define USB_EP_SPEED_LOW       0x00000000 // Low Speed
#define USB_EP_SPEED_FULL      0x00001000 // Full Speed
#define USB_EP_HOST_IN         0x00000000 // Host IN endpoint
#define USB_EP_HOST_OUT        0x00002000 // Host OUT endpoint
#define USB_EP_DEV_IN          0x00002000 // Device IN endpoint
#define USB_EP_DEV_OUT         0x00000000 // Device OUT endpoint

```

ulFIFOSize 参数:

```

#define USB_FIFO_SZ_8          0x00000000 // 8 byte FIFO
#define USB_FIFO_SZ_16         0x00000001 // 16 byte FIFO
#define USB_FIFO_SZ_32         0x00000002 // 32 byte FIFO
#define USB_FIFO_SZ_64         0x00000003 // 64 byte FIFO
#define USB_FIFO_SZ_128        0x00000004 // 128 byte FIFO
#define USB_FIFO_SZ_256        0x00000005 // 256 byte FIFO
#define USB_FIFO_SZ_512        0x00000006 // 512 byte FIFO
#define USB_FIFO_SZ_1024       0x00000007 // 1024 byte FIFO
#define USB_FIFO_SZ_2048       0x00000008 // 2048 byte FIFO
#define USB_FIFO_SZ_4096       0x00000009 // 4096 byte FIFO
#define USB_FIFO_SZ_8_DB       0x00000010 // 8 byte double buffered FIFO
#define USB_FIFO_SZ_16_DB      0x00000011 // 16 byte double buffered FIFO
#define USB_FIFO_SZ_32_DB      0x00000012 // 32 byte double buffered FIFO
#define USB_FIFO_SZ_64_DB      0x00000013 // 64 byte double buffered FIFO
#define USB_FIFO_SZ_128_DB     0x00000014 // 128 byte double buffered FIFO
#define USB_FIFO_SZ_256_DB     0x00000015 // 256 byte double buffered FIFO
#define USB_FIFO_SZ_512_DB     0x00000016 // 512 byte double buffered FIFO
#define USB_FIFO_SZ_1024_DB    0x00000017 // 1024 byte double buffered FIFO
#define USB_FIFO_SZ_2048_DB    0x00000018 // 2048 byte double buffered FIFO

```

端点转化到其状态控制器寄存器地址:

```

#define EP_OFFSET(Endpoint)    (Endpoint - 0x10)

```

主机端点配置函数:

```

void USBHostEndpointConfig(unsigned long ulBase, unsigned long ulEndpoint,
                           unsigned long ulMaxPayload,
                           unsigned long ulNAKPollInterval,
                           unsigned long ulTargetEndpoint, unsigned long ulFlags)
{
    unsigned long ulRegister;
    //判断是否是端点 0, 端点 0 的发送与接收配置使用同一寄存器。
    if(ulEndpoint == USB_EP_0)
    {
        HWREGB(ulBase + USB_O_NAKLMT) = ulNAKPollInterval;
        HWREGB(ulBase + EP_OFFSET(ulEndpoint) + USB_O_TYPE0) =
            ((ulFlags & USB_EP_SPEED_FULL) ? USB_TYPE0_SPEED_FULL :
            USB_TYPE0_SPEED_LOW);
    }
    //端点 1-15 配置
    else
    {
        ulRegister = ulTargetEndpoint;
        if(ulFlags & USB_EP_SPEED_FULL)
        {
            ulRegister |= USB_TXTYPE1_SPEED_FULL;
        }
    }
}

```

```

else
{
    ulRegister |= USB_TXTYPE1_SPEED_LOW;
}
switch(ulFlags & USB_EP_MODE_MASK)
{
    case USB_EP_MODE_BULK:
    {
        ulRegister |= USB_TXTYPE1_PROTO_BULK;
        break;
    }
    case USB_EP_MODE_ISOC:
    {
        ulRegister |= USB_TXTYPE1_PROTO_ISOC;
        break;
    }
    case USB_EP_MODE_INT:
    {
        ulRegister |= USB_TXTYPE1_PROTO_INT;
        break;
    }
    case USB_EP_MODE_CTRL:
    {
        ulRegister |= USB_TXTYPE1_PROTO_CTRL;
        break;
    }
}
//发送/接收端点配置
if(ulFlags & USB_EP_HOST_OUT)
{
    HWREGB(ulBase + EP_OFFSET(ulEndpoint) + USB_0_TXTYPE1) =
        ulRegister;
    HWREGB(ulBase + EP_OFFSET(ulEndpoint) + USB_0_TXINTERVAL1) =
        ulNAKPollInterval;
    HWREGB(ulBase + EP_OFFSET(ulEndpoint) + USB_0_TXMAXP1) =
        ulMaxPayload;
    ulRegister = 0;
    if(ulFlags & USB_EP_AUTO_SET)
    {
        ulRegister |= USB_TXCSRH1_AUTOSET;
    }
    if(ulFlags & USB_EP_DMA_MODE_1)
    {
        ulRegister |= USB_TXCSRH1_DMAEN | USB_TXCSRH1_DMAMOD;
    }
    else if(ulFlags & USB_EP_DMA_MODE_0)
    {
        ulRegister |= USB_TXCSRH1_DMAEN;
    }
    HWREGB(ulBase + EP_OFFSET(ulEndpoint) + USB_0_TXCSRH1) =
        (unsigned char)ulRegister;
}
else
{
    HWREGB(ulBase + EP_OFFSET(ulEndpoint) + USB_0_RXTYPE1) =
        ulRegister;
    HWREGB(ulBase + EP_OFFSET(ulEndpoint) + USB_0_RXINTERVAL1) =
        ulNAKPollInterval;
    ulRegister = 0;
}

```

```

        if (ulFlags & USB_EP_AUTO_CLEAR)
        {
            ulRegister |= USB_RXCSRH1_AUTOCL;
        }
        //DMA 控制
        if (ulFlags & USB_EP_DMA_MODE_1)
        {
            ulRegister |= USB_RXCSRH1_DMAEN | USB_RXCSRH1_DMAMOD;
        }
        else if (ulFlags & USB_EP_DMA_MODE_0)
        {
            ulRegister |= USB_RXCSRH1_DMAEN;
        }
        HWREGB(ulBase + EP_OFFSET(ulEndpoint) + USB_0_RXCSRH1) =
            (unsigned char)ulRegister;
    }
}

```

设备端点配置函数:

```

void USBDevEndpointConfigSet(unsigned long ulBase, unsigned long ulEndpoint,
                             unsigned long ulMaxPacketSize, unsigned long ulFlags)
{
    unsigned long ulRegister;
    if (ulFlags & USB_EP_DEV_IN)
    {
        HWREGB(ulBase + EP_OFFSET(ulEndpoint) + USB_0_TXMAXP1) =
            ulMaxPacketSize;
        ulRegister = 0;
        if (ulFlags & USB_EP_AUTO_SET)
        {
            ulRegister |= USB_TXCSRH1_AUTOSET;
        }
        if (ulFlags & USB_EP_DMA_MODE_1)
        {
            ulRegister |= USB_TXCSRH1_DMAEN | USB_TXCSRH1_DMAMOD;
        }
        else if (ulFlags & USB_EP_DMA_MODE_0)
        {
            ulRegister |= USB_TXCSRH1_DMAEN;
        }
        if ((ulFlags & USB_EP_MODE_MASK) == USB_EP_MODE_ISOC)
        {
            ulRegister |= USB_TXCSRH1_ISO;
        }
        HWREGB(ulBase + EP_OFFSET(ulEndpoint) + USB_0_TXCSRH1) =
            (unsigned char)ulRegister;
        HWREGB(ulBase + EP_OFFSET(ulEndpoint) + USB_0_TXCSRL1) =
            USB_TXCSRL1_CLRDT;
    }
    else
    {
        HWREGB(ulBase + EP_OFFSET(ulEndpoint) + USB_0_RXMAXP1) =
            ulMaxPacketSize;
        ulRegister = 0;
        if (ulFlags & USB_EP_AUTO_CLEAR)
        {
            ulRegister = USB_RXCSRH1_AUTOCL;
        }
        if (ulFlags & USB_EP_DMA_MODE_1)
    }
}

```

```

        {
            ulRegister |= USB_RXCSRH1_DMAEN | USB_RXCSRH1_DMAMOD;
        }
        else if (ulFlags & USB_EP_DMA_MODE_0)
        {
            ulRegister |= USB_RXCSRH1_DMAEN;
        }
        if ((ulFlags & USB_EP_MODE_MASK) == USB_EP_MODE_ISOC)
        {
            ulRegister |= USB_RXCSRH1_ISO;
        }
        HWREGB(ulBase + EP_OFFSET(ulEndpoint) + USB_0_RXCSRH1) =
            (unsigned char)ulRegister;
        HWREGB(ulBase + EP_OFFSET(ulEndpoint) + USB_0_RXCSRL1) =
            USB_RXCSRL1_CLRDT;
    }
}

```

从 FIFO 中读取数据函数:

```

long USBEndpointDataGet(unsigned long ulBase, unsigned long ulEndpoint,
                        unsigned char *pucData, unsigned long *pulSize)
{
    unsigned long ulRegister, ulByteCount, ulFIFO;
    if (ulEndpoint == USB_EP_0)
    {
        ulRegister = USB_0_CSRL0;
    }
    else
    {
        ulRegister = USB_0_RXCSRL1 + EP_OFFSET(ulEndpoint);
    }
    //判断数据是否准备好。
    if ((HWREGH(ulBase + ulRegister) & USB_CSRL0_RXRDY) == 0)
    {
        *pulSize = 0;
        return(-1);
    }
    //获取要读取的数据个数
    ulByteCount = HWREGH(ulBase + USB_0_COUNT0 + ulEndpoint);
    ulByteCount = (ulByteCount < *pulSize) ? ulByteCount : *pulSize;
    *pulSize = ulByteCount;
    ulFIFO = ulBase + USB_0_FIF00 + (ulEndpoint >> 2);
    //从 FIFO 中读取
    for (; ulByteCount > 0; ulByteCount--)
    {
        *pucData++ = HWREGB(ulFIFO);
    }
    return(0);
}

```

写数据到 FIFO 中函数:

```

long USBEndpointDataPut(unsigned long ulBase, unsigned long ulEndpoint,
                        unsigned char *pucData, unsigned long ulSize)
{
    unsigned long ulFIFO;
    unsigned char ucTxPktRdy;
    if (ulEndpoint == USB_EP_0)
    {
        ucTxPktRdy = USB_CSRL0_TXRDY;
    }
}

```

```

else
{
    ucTxPktRdy = USB_TXCSRL1_TXRDY;
}
//判断是否可以写入数据
if(HWREGB(ulBase + USB_0_CSRL0 + ulEndpoint) & ucTxPktRdy)
{
    return(-1);
}
//计算 FIFO 地址
ulFIFO = ulBase + USB_0_FIF00 + (ulEndpoint >> 2);
//写入 FIFO
for(; ulSize > 0; ulSize--)
{
    HWREGB(ulFIFO) = *pucData++;
}
return(0);
}

```

发送刚写入 FIFO 的数据函数：

```

long USBEndpointDataSend(unsigned long ulBase, unsigned long ulEndpoint,
                        unsigned long ulTransType)
{
    unsigned long ulTxPktRdy;
    if(ulEndpoint == USB_EP_0)
    {
        ulTxPktRdy = ulTransType & 0xff;
    }
    else
    {
        ulTxPktRdy = (ulTransType >> 8) & 0xff;
    }
    // 判断发送数据包是否准备好
    if(HWREGB(ulBase + USB_0_CSRL0 + ulEndpoint) & USB_CSRL0_TXRDY)
    {
        return(-1);
    }
    //发送刚写入的数据
    HWREGB(ulBase + USB_0_CSRL0 + ulEndpoint) = ulTxPktRdy;
    return(0);
}

```

以上介绍了常用 USB 寄存器，不常用寄存器请读者参考相关数据手册。通过以上寄存器操作可以开发 USB 主机与设备模块，可以完成全部 USB2.0 支持的全速和低速系统开发。一般情况下，不使用寄存器级编程，为了方便使用，LM3S 处理器官方已经制作好了 C 语言“driverlib.lib”库，并免费为开发人员提供，在 LM3S5000 系列 MCU 中已将“driverlib.lib”库预先放入 ROM 中，节约程序储存空间，并且调用速度更快。第三章将重点介绍 Stellaris 处理器的 USB 底层驱动函数。已经编译到 driverlib.lib 中，只要加入 usb.h 头文件就可以进行开发。第四章及以后的章节会介绍使用 usb.lib 开发 USB 主机与设备模块。

2.4 USB 处理配置使用

与其它处理器一样，在使用其相关外设资源时，要先进行初始化配置。初始化与配置分为四步：

① 内核配置

使用 USB 处理器前必须配置 RCGC2 寄存器，使其外设时钟；使能 USB 的 PLL 为 PHY 提供时钟；使能相应 USB 中断。

② 管脚配置

配置 RCGC2 使能相应 GPIO 模块；配置 GPIOPCTL 寄存器中的 PMCN 位，分配 USB 信号到合适的管脚上（根据具体芯片配置，有的芯片不需要此步）；作为设备时，必须禁止向 VBUS 供电，使用外部主机控制器供电。通常使用 USB0EPEN 信号用于控制外部稳压器，禁止使能外部稳压器，避免同时驱动电源管脚 USB0VBUS。

③ 端点配置

在主机模式，在和设备端点建立连接时配置；在设备模式，设备枚举之前配置。实际使用时端点很少需要设置，但都很重要。

④ 建立通信

硬件配置完成后还需进行协议配置，比如描述符、类协议之类，与 USB 通信相关的协议部分都需要考虑。

例如：使用 USB 处理器做鼠标，开始要进行以下配置（①、②步骤，③、④后面详细讲）。

```
//配置内核 CPU 时钟，作设备时不能低于 20MHz.
HWREG(SYSCTL_RCC) &= ~(SYSCTL_RCC_MOSCDIS);
SysCtlDelay(524288);
HWREG(SYSCTL_RCC) = ((HWREG(SYSCTL_RCC) &
                        ~(SYSCTL_RCC_PWRDN | SYSCTL_RCC_XTAL_M |
                          SYSCTL_RCC_OSCSRC_M)) |
                      SYSCTL_RCC_XTAL_8MHZ | SYSCTL_RCC_OSCSRC_MAIN);
SysCtlDelay(524288);
HWREG(SYSCTL_RCC) = ((HWREG(SYSCTL_RCC) & ~(SYSCTL_RCC_BYPASS |
        SYSCTL_RCC_SYSDIV_M)) | SYSCTL_RCC_SYSDIV_4 |
                      SYSCTL_RCC_USESYSDIV);

//使能 USB 设备时钟
SysCtlPeripheralEnable(SYSCTL_PERIPH_USB0);
// 打开 USB Phy 时钟.
SysCtlUSBPllEnable();
//清除中断标志。
USBIntStatusControl(USB0_BASE);
USBIntStatusEndpoint(USB0_BASE);
//使能相关中断
USBIntEnableControl(USB0_BASE, USB_INTCTRL_RESET |
                    USB_INTCTRL_DISCONNECT |
                    USB_INTCTRL_RESUME |
                    USB_INTCTRL_SUSPEND |
                    USB_INTCTRL_SOF);
USBIntEnableEndpoint(USB0_BASE, USB_INTEP_ALL);
//开总中断
IntEnable(INT_USB0);
//引脚配置
SysCtlPeripheralEnable(SYSCTL_PERIPH_GPIOF);
GPIOPinTypeGPIOOutput(GPIO_PORTF_BASE, 0xf0);
GPIOPinWrite(GPIO_PORTF_BASE, 0xf0, 0);
GPIOPinTypeGPIOInput(GPIO_PORTF_BASE, 0x0f);
HWREG(GPIO_PORTF_BASE+GPIO_O_PUR) = 0x0f;
```

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

第三章 底层库函数

3.1 底层库函数简介

寄存器级编程直接、效率高，但不易编写与移植。一般情况下，不使用寄存器级编程。为了让开发者在最短时间内完成产品设计，Luminary Micro Stellaris 外围驱动程序库是一系列用来访问 Stellaris 系列的基于 Cortex-M3 微处理器上的外设的驱动程序。尽管从纯粹的操作系统的理解上它们不是驱动程序，但这些驱动程序确实提供了一种机制，使器件的外设使用起来很容易。

对于许多应用来说，驱动程序直接使用就能满足一般应用的功能、内存或处理要求。外设驱动程序库提供二个编程模型：直接寄存器访问模型和软件驱动程序模型。根据应用的需要或者开发者所需要的编程环境，每个模型可以独立使用或组合使用。

每个编程模型有优点也有弱点。使用直接寄存器访问模型通常得到比使用软件驱动程序模型更少和更高效的代码。然而，直接寄存器访问模型一定要求了解每个寄存器、位段、它们之间的相互作用以及任何一个外设适当操作所需的先后顺序的详细内容；而开发者使用软件驱动程序模型，则不需要知道这些详细内容，通常只需更短的时间开发应用。

驱动程序能够对外设进行完全的控制，在 USB 产品开发时，可以直接使用驱动库函数编程，从而缩短开发周期。开发模型如下图：

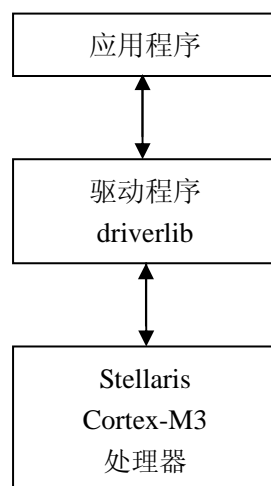


图 1 驱动程序开发模型

驱动程序包含 Stellaris 处理器的全部外设资源控制,下面就 USB 开发过程中会使用到常用底层库函数进行分类说明。

3.2 通用库函数

通用库函数,包含了内核操作、中断控制、GPIO 控制、USB 基本操作。能完成内核控制的全部操作,包器件的时钟、使能的外设、器件的配置、处理复位;能控制嵌套向量中断控制器(NVIC),使能和禁止中断、注册中断处理程序和设置中断的优先级;提供对多达 8 个独立 GPIO 管脚(实际出现的管脚数取决于 GPIO 端口和器件型号)的控制;能进行寄存器级操作 USB 外设模块。

3.2.1 内核操作

在处理器使用之前要进行必要的系统配置,包括内核电压、CPU 主频、外设资源使能等;在应用程序开发中还常常用到获取系统时钟、延时等操作。这些操作都通过内核系统操作函数进行访问。

```
void SysCtlLD0Set(unsigned long ulVoltage)
```

作用: 配置内核电压。

参数: ulVoltage, 内核电压参数。在使用 PLL 之前,最好设置内核电压为 2.75V。保证处理器稳定。

返回: 无

例如: 设置 CPU 主机为 2.75V

```
SysCtlLD0Set(SYSCTL_LD0_2_75V);
```

```
void SysCtlClockSet(unsigned long ulConfig)
```

作用: 配置 CPU 统主频。

参数: ulConfig, 时钟配置。格式: (振荡源 | 晶体频率 | PLL 是否使用 | 分频)。

如果使用 PLL, 则配置的时钟 = 200Mhz / 分频。

返回: 无

例如: 外接晶振 8MHz, 设置系统频率为 50MHz。(200MHz / 4 = 50MHz)

```
SysCtlClockSet(SYSCTL_XTAL_8MHZ | SYSCTL_SYSDIV_4 | SYSCTL_USE_PLL | SYSCTL_OSC_MAIN);
```

```
unsigned long SysCtlClockGet(void)
```

作用: 获取 CPU 统主频。

参数：无。

返回：当前 CPU 主频。

```
void SysCtlPeripheralEnable(unsigned long ulPeripheral)
```

作用：使能外设资源，外设使用前必须先使能，使能对应外设资源时钟。

参数：ulPeripheral，外设资源。

返回：无

例如：使能 GPIO 的端口 A；使能 USB0 外设资源。

```
SysCtlPeripheralEnable(SYSCTL_PERIPH_GPIOA);
```

```
SysCtlPeripheralEnable(SYSCTL_PERIPH_USB0);
```

```
void SysCtlDelay(unsigned long ulCount)
```

作用：延时 ulCount * 3 个系统时钟周期。

参数：ulCount，延时计数。延时为 ulCount * 3 个系统时钟周期。

返回：无

例如：延时 1 秒。

```
SysCtlDelay(SysCtlClockGet() / 3);
```

例如：在使用 USB 设备时会先配置处理器，使用本节函数完成系统配置：外接晶振 6MHz，设置内核电压为 2.75V，配置 CPU 主频为 25MHz，使能 USB0 模块，使能端口 F，并延时 100ms。

```
SysCtlLDOSet(SYSCTL_LDO_2_75V);
```

```
SysCtlClockSet(SYSCTL_XTAL_6MHZ | SYSCTL_SYSDIV_8 | SYSCTL_USE_PLL | SYSCTL_OSC_MAIN);
```

```
SysCtlPeripheralEnable(SYSCTL_PERIPH_USB0);
```

```
SysCtlPeripheralEnable(SYSCTL_PERIPH_GPIOF);
```

```
SysCtlDelay(SysCtlClockGet() / 30);
```

3.2.2 系统中断控制

下面介绍两组中断函数，进行处理器中断控制。对于 Stellaris 处理器，中断类型很多、控制相当复杂。中断层次如下图：

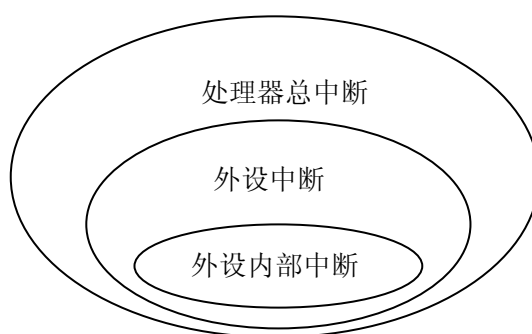


图 2 中断控制模型

```
void IntEnable(unsigned long ulInterrupt)
```

作用：使能外设资源中断。

参数：ulInterrupt，外设资源中断。

返回：无

例如：使能 USB0 中断。

```
IntEnable(INT_USB0);
```

```
void IntDisable(unsigned long ulInterrupt)
```

作用：禁止外设资源中断。

参数：ulInterrupt，外设资源中断。

返回：无

例如：禁止 USB0 中断。

```
IntDisable (INT_USB0);
```

```
tBoolean IntMasterEnable(void)
```

作用：使能总中断。

参数：无。

返回：中断使能是否成功。

例如：使能总中断。

```
IntMasterEnable();
```

```
tBoolean IntMasterDisable(void)
```

作用：禁止总中断。

参数：无。

返回：中断禁止是否成功。

例如：禁止总中断。

```
IntMasterDisable();
```

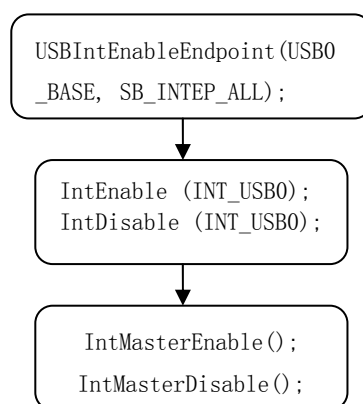


图 3 中断控制实际操作模型

从图 2 可以看出，处理器中断控制必须进行三层配置，图 3 显示中断控制的实际操作过程和配置顺序为从上到下依次配置。首先配置外设模块内部中断，比如 USBIntEnableEndpoint 使能 USB 模块的端点中断；再配置外设中断，比如 IntEnable 使能 USB0 模块中断；最后配置总中断，IntMasterEnable 使能总中断，如果不使能总中断，系统外设中断和外设内部中断都无法正常工作。

3.2.3 GPIO 控制

GPIO，通用输入输出端口。对于处理器来说，GPIO 占有重要的应用地位。无论是通信还是控制，都离不开 GPIO。Stellaris GPIO 模块由 7 个物理 GPIO 模块组成，分别对端口

A、端口 B、端口 C、端口 D、端口 E、端口 F 和端口 G，各种 Stellaris Cortex-3 的模块数量不一样，具体模块数请参考对应数据手册；每个模块支持 8 个通用输入输出端口（GPIO），即 GPIO_PIN_0 到 GPIO_PIN_7，Stellaris Cortex-3 可支持 5-42 个独立的 GPIO；每个 GPIO 可以根据具体使用情况独立配置成输入\输出。每种 Stellaris Cortex-3 模块数量、模块上的 GPIO 数量不一样，但是使用方法都一样。

在使用下列函数前，必须 SysCtlPeripheralEnable 使能对应 GPIO 模块。

```
void GPIOPinTypeGPIOInput(unsigned long ulPort, unsigned char ucPins);
void GPIOPinTypeGPIOOutput(unsigned long ulPort, unsigned char ucPins);
void GPIOPinTypeUSBAnalog(unsigned long ulPort, unsigned char ucPins);
void GPIOPinTypeUSBDigital(unsigned long ulPort, unsigned char ucPins);
long GPIOPinRead(unsigned long ulPort, unsigned char ucPins);
void GPIOPinWrite(unsigned long ulPort, unsigned char ucPins,
                  unsigned char ucVal);
void GPIOIntTypeSet(unsigned long ulPort, unsigned char ucPins,
                  unsigned long ulIntType);
void GPIOPinIntEnable(unsigned long ulPort, unsigned char ucPins);
void GPIOPinIntDisable(unsigned long ulPort, unsigned char ucPins);
long GPIOPinIntStatus(unsigned long ulPort, tBoolean bMasked);
void GPIOPinIntClear(unsigned long ulPort, unsigned char ucPins);
```

引脚定义：

```
#define GPIO_PIN_0      0x00000001 // GPIO pin 0
#define GPIO_PIN_1      0x00000002 // GPIO pin 1
#define GPIO_PIN_2      0x00000004 // GPIO pin 2
#define GPIO_PIN_3      0x00000008 // GPIO pin 3
#define GPIO_PIN_4      0x00000010 // GPIO pin 4
#define GPIO_PIN_5      0x00000020 // GPIO pin 5
#define GPIO_PIN_6      0x00000040 // GPIO pin 6
#define GPIO_PIN_7      0x00000080 // GPIO pin 7
```

```
void GPIOPinTypeGPIOInput(unsigned long ulPort, unsigned char ucPins);
```

作用：把 ulPort 端口上的 ucPins 引脚设置为输入。

参数：ulPort，指定端口，GPIO_PORTn_BASE，n 取对应端口号（A-G）；ucPins，指定控制引脚，GPIO_PIN_n，n 取对应引脚号（0-7）。

返回：无。

例如：设置端口 A 的 3、4 引脚为输入。

```
GPIOPinTypeGPIOInput(GPIO_PORTA_BASE, (GPIO_PIN_3 | GPIO_PIN_4));
```

```
void GPIOPinTypeUSBAnalog(unsigned long ulPort, unsigned char ucPins);
```

作用：把 ulPort 端口上的 ucPins 引脚配置为 USB 使用的模拟信号引脚。

参数：ulPort，指定端口，GPIO_PORTn_BASE，n 取对应端口号（A-G）；ucPins，指定控制引脚，GPIO_PIN_n，n 取对应引脚号（0-7）。

返回：无。

```
void GPIOPinTypeUSBDigital (unsigned long ulPort, unsigned char ucPins);
```

作用：把 ulPort 端口上的 ucPins 引脚配置为 USB 使用的数字信号引脚。

参数: ulPort, 指定端口, GPIO_PORTn_BASE, n 取对应端口号 (A-G); ucPins, 指定控制引脚, GPIO_PIN_n, n 取对应引脚号 (0-7)。

返回: 无。

GPIOPinTypeUSBAAnalog 和 GPIOPinTypeUSBDigital 不能把所有引脚都设置为 USB 功能引脚, 必须要相应引脚有 USB 引脚功能, 才可配置。不同 USB 处理器 USB 功能引脚不一样, 具体情况请参考对应芯片的数据手册。由于以上两个函数不常用, 在使用 lm3s3xxxx 与 lm3s5xxxx 系列时一般不使用。Lm3s9xxxx 系列或以后更高版本由于引脚复用情况较多, 所以必须使用以上两个函数, 确定 USB 功能引脚。

```
void GPIOPinWrite(unsigned long ulPort, unsigned char ucPins,  
                  unsigned char ucVal);
```

作用: 向 ulPort 端口上的 ucPins 引脚写入 ucVal 值。

参数: ulPort, 指定端口, GPIO_PORTn_BASE, n 取对应端口号 (A-G); ucPins, 指定控制引脚, GPIO_PIN_n, n 取对应引脚号 (0-7); ucVal, 待写入的数据。

返回: 无。

```
long GPIOPinRead(unsigned long ulPort, unsigned char ucPins);
```

作用: 把 ulPort 端口的 ucPins 引脚状态读出。

参数: ulPort, 指定端口, GPIO_PORTn_BASE, n 取对应端口号 (A-G); ucPins, 指定控制引脚, GPIO_PIN_n, n 取对应引脚号 (0-7)。

返回: ucPins 指定引脚状态组合。

例如: A 口输入数据从 B 口输出。

```
GPIOPinWrite(GPIO_PORTB_BASE, 0xff, GPIOPinRead(GPIO_PORTA_BASE, 0xff));
```

//GPIO 中断类型

```
#define GPIO_FALLING_EDGE      0x00000000 // Interrupt on falling edge
```

```
#define GPIO_RISING_EDGE       0x00000004 // Interrupt on rising edge
```

```
#define GPIO_BOTH_EDGES        0x00000001 // Interrupt on both edges
```

```
#define GPIO_LOW_LEVEL         0x00000002 // Interrupt on low level
```

```
#define GPIO_HIGH_LEVEL        0x00000007 // Interrupt on high level
```

```
void GPIOIntTypeSet(unsigned long ulPort, unsigned char ucPins,  
                    unsigned long ulIntType);
```

作用: 设置 ulPort 端口的 ucPins 引脚中断类型。

参数: ulPort, 指定端口, GPIO_PORTn_BASE, n 取对应端口号 (A-G); ucPins, 指定控制引脚, GPIO_PIN_n, n 取对应引脚号 (0-7); ulIntType, 指定中断类型。

返回: 无

```
void GPIOPinIntEnable(unsigned long ulPort, unsigned char ucPins);
```

作用: 使能 ulPort 端口的 ucPins 引脚中断。

参数: ulPort, 指定端口, GPIO_PORTn_BASE, n 取对应端口号 (A-G); ucPins, 指定控制引脚, GPIO_PIN_n, n 取对应引脚号 (0-7)。

返回: 无

```
void GPIOPinIntDisable(unsigned long ulPort, unsigned char ucPins);
```

作用: 禁止 ulPort 端口的 ucPins 引脚中断。

参数: ulPort, 指定端口, GPIO_PORTn_BASE, n 取对应端口号 (A-G); ucPins, 指定控制引脚, GPIO_PIN_n, n 取对应引脚号 (0-7)。

返回：无

```
long GPIOPinIntStatus(unsigned long ulPort, tBoolean bMasked);
```

作用：获取 ulPort 端口的中断状态。

参数：ulPort，指定端口，GPIO_PORTn_BASE，n 取对应端口号（A-G）；bMasked，是否获取屏蔽后中断状态，一般使用 true。

返回：引脚中断状态。

```
void GPIOPinIntClear(unsigned long ulPort, unsigned char ucPins);
```

作用：清除 ulPort 端口的 ucPins 引脚中断标志。

参数：ulPort，指定端口，GPIO_PORTn_BASE，n 取对应端口号（A-G）；ucPins，指定控制引脚，GPIO_PIN_n，n 取对应引脚号（0-7）。

返回：无

例如：使用已学的函数，利用 GPIO 中断，完成 PC4-7 上按键控制 PB0、PB1、PB4、PB5 上 LED 灯的任务。KEY1 控制 LED1，KEY2 控制 LED2，KEY3 控制 LED3，KEY3 控制 LED4，当按键按下，对应 LED 取当前的相反状态，如 LED 现在亮，当按下 KEY 时，对应的 LED 灭，再次按下 KEY 时，LED 又亮。

```
#include <sysctl.h>
#include <gpio.h>
#include <hw_memmap.h>
#include <hw_ints.h>
#include <interrupt.h>
#include <lm3s8962.h>

#define u32      unsigned long
//LED 引脚定义  PORTB
#define LED1     GPIO_PIN_1
#define LED2     GPIO_PIN_4
#define LED3     GPIO_PIN_5
#define LED4     GPIO_PIN_6
#define LED      (LED1 | LED2 | LED3 | LED4)
//按键引脚定义  PORTC
#define KEY1     GPIO_PIN_4
#define KEY2     GPIO_PIN_5
#define KEY3     GPIO_PIN_6
#define KEY4     GPIO_PIN_7
#define KEY      (KEY1 | KEY2 | KEY3 | KEY4)

int main(void)
{
    //设置内核电压与 CPU 主频。
    SysCtlLD0Set(SYSCTL_LD0_2_75V);
    SysCtlClockSet(SYSCTL_XTAL_6MHZ | SYSCTL_SYSDIV_8 | SYSCTL_USE_PLL | SYSCTL_OSC_MAIN);
    //使能 GPIO 外设
    SysCtlPeripheralEnable(SYSCTL_PERIPH_GPIOB);
    SysCtlPeripheralEnable(SYSCTL_PERIPH_GPIOC);
    //LED IO 设置
    GPIOPinTypeGPIOOutput(GPIO_PORTB_BASE, LED);
```

```

//KEY IO 设置
GPIOPinTypeGPIOInput(GPIO_PORTC_BASE, KEY);
//KEY 中断设置, 配置为上升沿触发。
GPIOIntTypeSet(GPIO_PORTC_BASE, KEY, GPIO_RISING_EDGE);
GPIOPinIntEnable(GPIO_PORTC_BASE, KEY);
//打开端口 C 中断
IntEnable(INT_GPIOC);
//打开全局中断。
IntMasterEnable();
//打开所有 LED
GPIOPinWrite(GPIO_PORTB_BASE, LED, LED);
//等待中断。
while(1)
{
}
}

void GPIO_Port_C_ISR(void)
{
    u32 ulStatus = GPIOPinIntStatus(GPIO_PORTC_BASE, 1);
    GPIOPinIntClear(GPIO_PORTC_BASE, KEY);
    if(ulStatus & KEY1)
    {
        GPIOPinWrite(GPIO_PORTB_BASE, LED1, ~GPIOPinRead(GPIO_PORTB_BASE, LED1));
    }
    else if(ulStatus & KEY2)
    {
        GPIOPinWrite(GPIO_PORTB_BASE, LED2, ~GPIOPinRead(GPIO_PORTB_BASE, LED2));
    }
    else if(ulStatus & KEY3)
    {
        GPIOPinWrite(GPIO_PORTB_BASE, LED3, ~GPIOPinRead(GPIO_PORTB_BASE, LED3));
    }
    else if(ulStatus & KEY4)
    {
        GPIOPinWrite(GPIO_PORTB_BASE, LED4, ~GPIOPinRead(GPIO_PORTB_BASE, LED4));
    }
}

```

掌握好以上 API 后, 能够为外设资源控制打好基础, 为学习 USB 底层驱动 API 函数打好基础。在实际 USB 工程开发中, 以上 API 函数使用次数较多, 在开发中占有重要地位。有关 MCU 外设的其它 API 函数请参考其它书籍。

3.3 USB 基本操作

USB API 提供了用来访问 Stellaris USB 设备控制器或主机控制器的函数集。API 分组如下: USBDev、USBHost、USBOTG、USBEndpoint 和 USBFIFO。USB 设备控制器只使用 USBDev

组 API；USB 主机控制器只使用 USBHost 中 API；OTG 接口的微控制器使用 USBOTG 组 API。USB OTG 微控制器，一旦配置完，则使用设备或主机 API。余下的 API 均可被 USB 主机和 USB 设备控制器使用，USBEndpoint 组 API 一般用来配置和访问端点，USBFIFO 组 API 则配置 FIFO 大小和位置。所以在本书中把 USB API 分为三类：USB 基本操作 API、设备库函数 API、主机库函数 API。“USB 基本操作 API”包含设备和主机都使用的 API：USBEndpoint 组 API、USBFIFO 组 API、其它公用 API。“设备库函数 API”只包含设备能够使用的 API 函数。“主机库函数 API”只包含主机能够使用的 API 函数。OTG 配置完成时，使用“设备库函数 API”和“主机库函数 AP”。

本节主要介绍 USB 基本操作 API，包含 USBEndpoint 组 API、USBFIFO 组 API、其它公用 API。

端点控制函数：

```
unsigned long USBEndpointDataAvail(unsigned long ulBase, unsigned long ulEndpoint);
long USBEndpointDataGet(unsigned long ulBase, unsigned long ulEndpoint,
                        unsigned char *pucData, unsigned long *pulSize);
long USBEndpointDataPut(unsigned long ulBase, unsigned long ulEndpoint,
                        unsigned char *pucData, unsigned long ulSize);
long USBEndpointDataSend(unsigned long ulBase, unsigned long ulEndpoint,
                        unsigned long ulTransType);
void USBEndpointDataToggleClear(unsigned long ulBase,
                                unsigned long ulEndpoint,
                                unsigned long ulFlags);
unsigned long USBEndpointStatus(unsigned long ulBase,
                                unsigned long ulEndpoint);
void USBEndpointDMAChannel(unsigned long ulBase,
                           unsigned long ulEndpoint,
                           unsigned long ulChannel);
void USBEndpointDMAEnable(unsigned long ulBase, unsigned long ulEndpoint,
                           unsigned long ulFlags);
void USBEndpointDMADisable(unsigned long ulBase,
                            unsigned long ulEndpoint,
                            unsigned long ulFlags);
```

FIFO 控制函数：

```
unsigned long USBFIFOAddrGet(unsigned long ulBase,
                             unsigned long ulEndpoint);
void USBFIFOConfigSet(unsigned long ulBase, unsigned long ulEndpoint,
                     unsigned long ulFIFOAddress,
                     unsigned long ulFIFOSize, unsigned long ulFlags);
void USBFIFOConfigGet(unsigned long ulBase, unsigned long ulEndpoint,
                     unsigned long *pulFIFOAddress,
                     unsigned long *pulFIFOSize,
                     unsigned long ulFlags);
void USBFIFOFlush(unsigned long ulBase, unsigned long ulEndpoint,
                  unsigned long ulFlags);
```

中断控制函数：

```

void USBIntEnableControl(unsigned long ulBase,
                        unsigned long ulIntFlags);
void USBIntDisableControl(unsigned long ulBase,
                        unsigned long ulIntFlags);
unsigned long USBIntStatusControl(unsigned long ulBase);
void USBIntEnableEndpoint(unsigned long ulBase,
                        unsigned long ulIntFlags);
void USBIntDisableEndpoint(unsigned long ulBase,
                        unsigned long ulIntFlags);
unsigned long USBIntStatusEndpoint(unsigned long ulBase);
void USBIntEnable(unsigned long ulBase, unsigned long ulIntFlags);
void USBIntDisable(unsigned long ulBase, unsigned long ulIntFlags);
unsigned long USBIntStatus(unsigned long ulBase);

```

其它控制函数:

```

unsigned long USBFrameNumberGet(unsigned long ulBase);
void USBOTGSessionRequest(unsigned long ulBase, tBoolean bStart);
unsigned long USBModeGet(unsigned long ulBase);

```

端点控制函数包含所有端点控制: 端点设置、端点数据发送与接收、端点 DMA 控制等。是 USB 通信最基本的一组函数。使用下面函数前要进行端点控制, 如果是 USB 主机, 使用 USBHostEndpointConfig 配置主机端点; 如果是 USB 设备, 使用 USBDevEndpointConfigSet 配置设备端点。

```

unsigned long USBEndpointDataAvail(unsigned long ulBase,
                                unsigned long ulEndpoint)

```

作用: 检查接收端点中有多少数据可用。

参数: ulBase, 指定 USB 模块, 在 Stellaris USB 处理器中只有一个 USB 模块, 所以参数为 USB0_BASE。ulEndpoint, 指定接收端点, USB_EP_n (n=0..15)。

返回: 接收端点中的可用数据个数。

```

long USBEndpointDataGet(unsigned long ulBase, unsigned long ulEndpoint,
                        unsigned char *pucData, unsigned long *pulSize);

```

作用: 从 ulEndpoint 的 RXFIFO 中读取 pulSize 指定长度的数据到 pucData 指定数据中。

参数: ulBase, 指定 USB 模块, 在 Stellaris USB 处理器中只有一个 USB 模块, 所以参数为 USB0_BASE。ulEndpoint, 指定接收端点, USB_EP_n (n=0..15)。pucData 指定接收数据的数组指针。pulSize 指定接收数据的长度。

返回: 函数是否成功执行。

```

long USBEndpointDataPut(unsigned long ulBase, unsigned long ulEndpoint,
                        unsigned char *pucData, unsigned long ulSize);

```

作用: pucData 中的 ulSize 个数据放入 ulEndpoint 的 TXFIFO 中, 等待发送。

参数: ulBase, 指定 USB 模块, 在 Stellaris USB 处理器中只有一个 USB 模块, 所以参数为 USB0_BASE。ulEndpoint, 指定操作的端点号, USB_EP_n (n=0..15)。pucData 指定待发送数据的数组指针。pulSize 指定发送数据的长度。

返回: 函数是否成功执行。

```

long USBEndpointDataSend(unsigned long ulBase, unsigned long ulEndpoint,
                        unsigned long ulTransType);

```


作用：触发 ulEndpoint 的 TXFIFO 发送数据。

参数：ulBase，指定 USB 模块，在 Stellaris USB 处理器中只有一个 USB 模块，所以参数为 USB0_BASE。ulEndpoint，指定操作的端点号，USB_EP_n (n=0..15)。ulTransType，指定发送类型。

返回：函数是否成功执行。

ulTransType 可选参数：

```
#define USB_TRANS_OUT          0x00000102 // Normal OUT transaction
#define USB_TRANS_IN           0x00000102 // Normal IN transaction
#define USB_TRANS_IN_LAST      0x0000010a // Final IN transaction (for
                                         // endpoint 0 in device mode)
#define USB_TRANS_SETUP        0x0000110a // Setup transaction (for endpoint
                                         // 0)
#define USB_TRANS_STATUS       0x00000142 // Status transaction (for endpoint
                                         // 0)
```

```
void USBEndpointDataToggleClear(unsigned long ulBase,
                                unsigned long ulEndpoint,
                                unsigned long ulFlags);
```

作用：清除 Toggle。

参数：ulBase，指定 USB 模块，在 Stellaris USB 处理器中只有一个 USB 模块，所以参数为 USB0_BASE。ulEndpoint，指定操作的端点号，USB_EP_n (n=0..15)。ulFlags，指定访问 IN/OUT 端点。

返回：无。

```
unsigned long USBEndpointStatus(unsigned long ulBase,
                                unsigned long ulEndpoint);
```

作用：获取 ulEndpoint 指定端点的状态。

参数：ulBase，指定 USB 模块，在 Stellaris USB 处理器中只有一个 USB 模块，所以参数为 USB0_BASE。ulEndpoint，指定操作的端点号，USB_EP_n (n=0..15)。

返回：ulEndpoint 端点的状态。

在设备模式下返回 USB_DEV_XX，主机模式下返回 USB_HOST_XX。返回数据：

```
#define USB_HOST_IN_PID_ERROR  0x01000000 // Stall on this endpoint received
#define USB_HOST_IN_NOT_COMP   0x00100000 // Device failed to respond
#define USB_HOST_IN_STALL      0x00400000 // Stall on this endpoint received
#define USB_HOST_IN_DATA_ERROR 0x00080000 // CRC or bit-stuff error
                                         // (ISOC Mode)
#define USB_HOST_IN_NAK_TO     0x00080000 // NAK received for more than the
                                         // specified timeout period
#define USB_HOST_IN_ERROR      0x00040000 // Failed to communicate with a
                                         // device
#define USB_HOST_IN_FIFO_FULL  0x00020000 // RX FIFO full
#define USB_HOST_IN_PKTRDY     0x00010000 // Data packet ready
#define USB_HOST_OUT_NAK_TO    0x00000080 // NAK received for more than the
                                         // specified timeout period
#define USB_HOST_OUT_NOT_COMP  0x00000080 // No response from device
```

```

// (ISOC mode)
#define USB_HOST_OUT_STALL      0x00000020 // Stall on this endpoint received
#define USB_HOST_OUT_ERROR      0x00000004 // Failed to communicate with a
// device
#define USB_HOST_OUT_FIFO_NE    0x00000002 // TX FIFO is not empty
#define USB_HOST_OUT_PKTEND      0x00000001 // Transmit still being transmitted
#define USB_HOST_EP0_NAK_TO      0x00000080 // NAK received for more than the
// specified timeout period
#define USB_HOST_EP0_STATUS      0x00000040 // This was a status packet
#define USB_HOST_EP0_ERROR      0x00000010 // Failed to communicate with a
// device
#define USB_HOST_EP0_RX_STALL    0x00000004 // Stall on this endpoint received
#define USB_HOST_EP0_RXPKTRDY    0x00000001 // Receive data packet ready
#define USB_DEV_RX_SENT_STALL    0x00400000 // Stall was sent on this endpoint
#define USB_DEV_RX_DATA_ERROR    0x00080000 // CRC error on the data
#define USB_DEV_RX_OVERRUN       0x00040000 // OUT packet was not loaded due to
// a full FIFO
#define USB_DEV_RX_FIFO_FULL     0x00020000 // RX FIFO full
#define USB_DEV_RX_PKT_RDY       0x00010000 // Data packet ready
#define USB_DEV_TX_NOT_COMP      0x00000080 // Large packet split up, more data
// to come
#define USB_DEV_TX_SENT_STALL    0x00000020 // Stall was sent on this endpoint
#define USB_DEV_TX_UNDERRUN      0x00000004 // IN received with no data ready
#define USB_DEV_TX_FIFO_NE       0x00000002 // The TX FIFO is not empty
#define USB_DEV_TX_TXPKTRDY      0x00000001 // Transmit still being transmitted
#define USB_DEV_EP0_SETUP_END    0x00000010 // Control transaction ended before
// Data End seen
#define USB_DEV_EP0_SENT_STALL   0x00000004 // Stall was sent on this endpoint
#define USB_DEV_EP0_IN_PKTEND    0x00000002 // Transmit data packet pending
#define USB_DEV_EP0_OUT_PKT_RDY  0x00000001 // Receive data packet ready
void USBEndpointDMAChannel(unsigned long ulBase,
                           unsigned long ulEndpoint,
                           unsigned long ulChannel);

```

作用：端点 DMA 控制。

参数：ulBase，指定 USB 模块，在 Stellaris USB 处理器中只有一个 USB 模块，所以参数为 USB0_BASE。ulEndpoint，指定操作的端点号，USB_EP_n (n=0..15)。ulChannel，指定 DMA 通过。

返回：无。

```

void USBEndpointDMAEnable(unsigned long ulBase, unsigned long ulEndpoint,
                           unsigned long ulFlags);

```

作用：端点 DMA 使能。

参数：ulBase，指定 USB 模块，在 Stellaris USB 处理器中只有一个 USB 模块，所以参数为 USB0_BASE。ulEndpoint，指定操作的端点号，USB_EP_n (n=0..15)。

返回：无。

```
void USBEndpointDMADisable(unsigned long ulBase,
                           unsigned long ulEndpoint,
                           unsigned long ulFlags);
```

作用：端点 DMA 禁止。

参数：ulBase，指定 USB 模块，在 Stellaris USB 处理器中只有一个 USB 模块，所以参数为 USB0_BASE。ulEndpoint，指定操作的端点号，USB_EP_n (n=0..15)。

返回：无。

在 USB 通信中，实际是端点与端点间通信，掌握端点控制函数是非常必要的。例如，在枚举过程中，会大量使用端点 0 与主机进行数据传输。下面程序介绍了枚举过程中端点 0 的数据传输。

```
tDeviceInstance g_psUSBDevice[1];
static void USBDEP0StateTx(unsigned long ulIndex)
{
    unsigned long ulNumBytes;
    unsigned char *pData;
    g_psUSBDevice[0].eEP0State = USB_STATE_TX;
    //设置端点待发送。
    ulNumBytes = g_psUSBDevice[0].ulEP0DataRemain;
    //端点 0 最大传输包为 64，传输数据包大于 64，就分批次传输。
    if(ulNumBytes > EP0_MAX_PACKET_SIZE)
    {
        ulNumBytes = EP0_MAX_PACKET_SIZE;
    }
    //数据指针，指向待发送数据组。
    pData = (unsigned char *)g_psUSBDevice[0].pEP0Data;
    //分批次发送
    g_psUSBDevice[0].ulEP0DataRemain -= ulNumBytes;
    g_psUSBDevice[0].pEP0Data += ulNumBytes;
    //把刚才数据放入端点 0 的 TXFIFO 中
    USBEndpointDataPut(USB0_BASE, USB_EP_0, pData, ulNumBytes);
    // 判断 ulNumBytes 大小，如果等于最大包长度。
    if(ulNumBytes == EP0_MAX_PACKET_SIZE)
    {
        //普通发送，对于主机来说，设备通过输入端点发送。
        USBEndpointDataSend(USB0_BASE, USB_EP_0, USB_TRANS_IN);
    }
    else
    {
        g_psUSBDevice[0].eEP0State = USB_STATE_STATUS;
        //发送数据结束，并发送标志位
        USBEndpointDataSend(USB0_BASE, USB_EP_0, USB_TRANS_IN_LAST);
        // 判断是否有 callback 函数。
        if((g_psUSBDevice[0].psInfo->sCallbacks.pfnDataSent) &&
            (g_psUSBDevice[0].ulOUTDataSize != 0))
```

```

    {
        //通过 callback 告诉应用程序，发送结束。
        g_psUSBDevice[0].psInfo->sCallbacks.pfnDataSent(
            g_psUSBDevice[0].pvInstance, g_psUSBDevice[0].ulOUTDataSize);
        g_psUSBDevice[0].ulOUTDataSize = 0;
    }
}
}

```

从上面程序中可以看出，如果发送数据大于端点最大包大小，则分批次发送，发送到最后一个数据包时，要传送包结束标志。

```

unsigned long USBFIFOAddrGet(unsigned long ulBase,
                             unsigned long ulEndpoint);

```

作用：通过端点获取端点对应的 FIFO 地址。

参数：ulBase，指定 USB 模块，在 Stellaris USB 处理器中只有一个 USB 模块，所以参数为 USB0_BASE。ulEndpoint，指定操作的端点号，USB_EP_n (n=0..15)。

返回：FIFO 地址。

```

void USBFIFOConfigSet(unsigned long ulBase, unsigned long ulEndpoint,
                      unsigned long ulFIFOAddress,
                      unsigned long ulFIFOSize,
                      unsigned long ulFlags);

```

作用：FIFO 配置。

参数：ulBase，指定 USB 模块，在 Stellaris USB 处理器中只有一个 USB 模块，所以参数为 USB0_BASE。ulEndpoint，指定操作的端点号，USB_EP_n (n=0..15)。ulFIFOAddress，配置 ulEndpoint 端点的 FIFO 首地址。ulFIFOSize，FIFO 大小。ulFlags，端点类型。

返回：无。

ulFIFOSize 参数：

```

#define USB_FIFO_SZ_8          0x00000000 // 8 byte FIFO
#define USB_FIFO_SZ_16         0x00000001 // 16 byte FIFO
#define USB_FIFO_SZ_32         0x00000002 // 32 byte FIFO
#define USB_FIFO_SZ_64         0x00000003 // 64 byte FIFO
#define USB_FIFO_SZ_128        0x00000004 // 128 byte FIFO
#define USB_FIFO_SZ_256        0x00000005 // 256 byte FIFO
#define USB_FIFO_SZ_512        0x00000006 // 512 byte FIFO
#define USB_FIFO_SZ_1024       0x00000007 // 1024 byte FIFO
#define USB_FIFO_SZ_2048       0x00000008 // 2048 byte FIFO
#define USB_FIFO_SZ_4096       0x00000009 // 4096 byte FIFO
#define USB_FIFO_SZ_8_DB       0x00000010 // 8 byte double buffered FIFO
#define USB_FIFO_SZ_16_DB      0x00000011 // 16 byte double buffered FIFO
#define USB_FIFO_SZ_32_DB      0x00000012 // 32 byte double buffered FIFO
#define USB_FIFO_SZ_64_DB      0x00000013 // 64 byte double buffered FIFO
#define USB_FIFO_SZ_128_DB     0x00000014 // 128 byte double buffered FIFO
#define USB_FIFO_SZ_256_DB     0x00000015 // 256 byte double buffered FIFO
#define USB_FIFO_SZ_512_DB     0x00000016 // 512 byte double buffered FIFO

```

```

#define USB_FIFO_SZ_1024_DB    0x00000017 // 1024 byte double buffered FIFO
#define USB_FIFO_SZ_2048_DB    0x00000018 // 2048 byte double buffered FIFO
ulFlags 参数:
#define USB_EP_AUTO_SET        0x00000001 // Auto set feature enabled
#define USB_EP_AUTO_REQUEST    0x00000002 // Auto request feature enabled
#define USB_EP_AUTO_CLEAR      0x00000004 // Auto clear feature enabled
#define USB_EP_DMA_MODE_0      0x00000008 // Enable DMA access using mode 0
#define USB_EP_DMA_MODE_1      0x00000010 // Enable DMA access using mode 1
#define USB_EP_MODE_ISOC       0x00000000 // Isochronous endpoint
#define USB_EP_MODE_BULK       0x00000100 // Bulk endpoint
#define USB_EP_MODE_INT        0x00000200 // Interrupt endpoint
#define USB_EP_MODE_CTRL       0x00000300 // Control endpoint
#define USB_EP_MODE_MASK       0x00000300 // Mode Mask
#define USB_EP_SPEED_LOW       0x00000000 // Low Speed
#define USB_EP_SPEED_FULL      0x00001000 // Full Speed
#define USB_EP_HOST_IN         0x00000000 // Host IN endpoint
#define USB_EP_HOST_OUT        0x00002000 // Host OUT endpoint
#define USB_EP_DEV_IN          0x00002000 // Device IN endpoint
#define USB_EP_DEV_OUT         0x00000000 // Device OUT endpoint

```

```

void USBFIFOConfigGet(unsigned long ulBase, unsigned long ulEndpoint,
                      unsigned long *pulFIFOAddress,
                      unsigned long *pulFIFOSize,
                      unsigned long ulFlags);

```

作用：获取 FIFO 配置。

参数：ulBase，指定 USB 模块，在 Stellaris USB 处理器中只有一个 USB 模块，所以参数为 USB0_BASE。ulEndpoint，指定操作的端点号，USB_EP_n (n=0..15)。ulFIFOAddress，获取 ulEndpoint 端点的 FIFO 首地址。ulFIFOSize，获取 FIFO 大小。ulFlags，端点类型。

返回：无。

```

void USBFIFOFlush(unsigned long ulBase, unsigned long ulEndpoint,
                  unsigned long ulFlags);

```

作用：清空 FIFO。

参数：ulBase，指定 USB 模块，在 Stellaris USB 处理器中只有一个 USB 模块，所以参数为 USB0_BASE。ulEndpoint，指定操作的端点号，USB_EP_n (n=0..15)。ulFlags，端点类型。

返回：无。

中断控制在 USB 开发中具有重要地位，USB 处理速度高达 12MHz，每一种状态都可以触发中断，并让处理器进行数据处理；USB 中断类型多于 20 个。在这种复杂的控制中，中断管理和相应数据处理是非常重要的。下面这些函数可以有效的进行 USB 中断控制。在使用这些函数前必须使能处理器总中断和外设模块中断。

```

void USBIntEnableControl(unsigned long ulBase, unsigned long ulIntFlags);

```

作用：使能 USB 通用中断，除端点中断外的所有 USB 外设内部中断。

参数: ulBase, 指定 USB 模块, 在 Stellaris USB 处理器中只有一个 USB 模块, 所以参数为 USB0_BASE。ulIntFlags, 中断标志。

返回: 无。

ulIntFlags 可用参数:

```
#define USB_INTCTRL_ALL          0x000003FF // All control interrupt sources
#define USB_INTCTRL_STATUS       0x000000FF // Status Interrupts
#define USB_INTCTRL_VBUS_ERR     0x00000080 // VBUS Error
#define USB_INTCTRL_SESSION      0x00000040 // Session Start Detected
#define USB_INTCTRL_SESSION_END  0x00000040 // Session End Detected
#define USB_INTCTRL_DISCONNECT   0x00000020 // Disconnect Detected
#define USB_INTCTRL_CONNECT      0x00000010 // Device Connect Detected
#define USB_INTCTRL_SOF          0x00000008 // Start of Frame Detected
#define USB_INTCTRL_BABBLE       0x00000004 // Babble signaled
#define USB_INTCTRL_RESET        0x00000004 // Reset signaled
#define USB_INTCTRL_RESUME       0x00000002 // Resume detected
#define USB_INTCTRL_SUSPEND      0x00000001 // Suspend detected
#define USB_INTCTRL_MODE_DETECT  0x00000200 // Mode value valid
#define USB_INTCTRL_POWER_FAULT  0x00000100 // Power Fault detected
```

```
void USBIntDisableControl(unsigned long ulBase, unsigned long ulIntFlags);
```

作用: 禁止 USB 通用中断, 除端点中断外的所有 USB 外设内部中断。

参数: ulBase, 指定 USB 模块, 在 Stellaris USB 处理器中只有一个 USB 模块, 所以参数为 USB0_BASE。ulIntFlags, 中断标志。

返回: 无。

```
unsigned long USBIntStatusControl(unsigned long ulBase);
```

作用: 获取中断标志。

参数: ulBase, 指定 USB 模块, 在 Stellaris USB 处理器中只有一个 USB 模块, 所以参数为 USB0_BASE。

返回: 返回中断标志, 与 USBIntEnableControl 的 ulIntFlags 标志相同。

```
void USBIntEnableEndpoint(unsigned long ulBase, unsigned long ulIntFlags);
```

作用: 使能 USB 端点中断。

参数: ulBase, 指定 USB 模块, 在 Stellaris USB 处理器中只有一个 USB 模块, 所以参数为 USB0_BASE。ulIntFlags, 中断标志。

返回: 无。

ulIntFlags 可选参数:

```
#define USB_INTEP_ALL          0xFFFFFFFF // Host IN Interrupts
#define USB_INTEP_HOST_IN      0xFFFE0000 // Host IN Interrupts
#define USB_INTEP_HOST_IN_15   0x80000000 // Endpoint 15 Host IN Interrupt
#define USB_INTEP_HOST_IN_14   0x40000000 // Endpoint 14 Host IN Interrupt
#define USB_INTEP_HOST_IN_13   0x20000000 // Endpoint 13 Host IN Interrupt
#define USB_INTEP_HOST_IN_12   0x10000000 // Endpoint 12 Host IN Interrupt
#define USB_INTEP_HOST_IN_11   0x08000000 // Endpoint 11 Host IN Interrupt
#define USB_INTEP_HOST_IN_10   0x04000000 // Endpoint 10 Host IN Interrupt
#define USB_INTEP_HOST_IN_9     0x02000000 // Endpoint 9 Host IN Interrupt
```

```

#define USB_INTEP_HOST_IN_8    0x01000000 // Endpoint 8 Host IN Interrupt
#define USB_INTEP_HOST_IN_7    0x00800000 // Endpoint 7 Host IN Interrupt
#define USB_INTEP_HOST_IN_6    0x00400000 // Endpoint 6 Host IN Interrupt
#define USB_INTEP_HOST_IN_5    0x00200000 // Endpoint 5 Host IN Interrupt
#define USB_INTEP_HOST_IN_4    0x00100000 // Endpoint 4 Host IN Interrupt
#define USB_INTEP_HOST_IN_3    0x00080000 // Endpoint 3 Host IN Interrupt
#define USB_INTEP_HOST_IN_2    0x00040000 // Endpoint 2 Host IN Interrupt
#define USB_INTEP_HOST_IN_1    0x00020000 // Endpoint 1 Host IN Interrupt
#define USB_INTEP_DEV_OUT      0xFFFE0000 // Device OUT Interrupts
#define USB_INTEP_DEV_OUT_15   0x80000000 // Endpoint 15 Device OUT Interrupt
#define USB_INTEP_DEV_OUT_14   0x40000000 // Endpoint 14 Device OUT Interrupt
#define USB_INTEP_DEV_OUT_13   0x20000000 // Endpoint 13 Device OUT Interrupt
#define USB_INTEP_DEV_OUT_12   0x10000000 // Endpoint 12 Device OUT Interrupt
#define USB_INTEP_DEV_OUT_11   0x08000000 // Endpoint 11 Device OUT Interrupt
#define USB_INTEP_DEV_OUT_10   0x04000000 // Endpoint 10 Device OUT Interrupt
#define USB_INTEP_DEV_OUT_9    0x02000000 // Endpoint 9 Device OUT Interrupt
#define USB_INTEP_DEV_OUT_8    0x01000000 // Endpoint 8 Device OUT Interrupt
#define USB_INTEP_DEV_OUT_7    0x00800000 // Endpoint 7 Device OUT Interrupt
#define USB_INTEP_DEV_OUT_6    0x00400000 // Endpoint 6 Device OUT Interrupt
#define USB_INTEP_DEV_OUT_5    0x00200000 // Endpoint 5 Device OUT Interrupt
#define USB_INTEP_DEV_OUT_4    0x00100000 // Endpoint 4 Device OUT Interrupt
#define USB_INTEP_DEV_OUT_3    0x00080000 // Endpoint 3 Device OUT Interrupt
#define USB_INTEP_DEV_OUT_2    0x00040000 // Endpoint 2 Device OUT Interrupt
#define USB_INTEP_DEV_OUT_1    0x00020000 // Endpoint 1 Device OUT Interrupt
#define USB_INTEP_HOST_OUT     0x0000FFFE // Host OUT Interrupts
#define USB_INTEP_HOST_OUT_15  0x00008000 // Endpoint 15 Host OUT Interrupt
#define USB_INTEP_HOST_OUT_14  0x00004000 // Endpoint 14 Host OUT Interrupt
#define USB_INTEP_HOST_OUT_13  0x00002000 // Endpoint 13 Host OUT Interrupt
#define USB_INTEP_HOST_OUT_12  0x00001000 // Endpoint 12 Host OUT Interrupt
#define USB_INTEP_HOST_OUT_11  0x00000800 // Endpoint 11 Host OUT Interrupt
#define USB_INTEP_HOST_OUT_10  0x00000400 // Endpoint 10 Host OUT Interrupt
#define USB_INTEP_HOST_OUT_9   0x00000200 // Endpoint 9 Host OUT Interrupt
#define USB_INTEP_HOST_OUT_8   0x00000100 // Endpoint 8 Host OUT Interrupt
#define USB_INTEP_HOST_OUT_7   0x00000080 // Endpoint 7 Host OUT Interrupt
#define USB_INTEP_HOST_OUT_6   0x00000040 // Endpoint 6 Host OUT Interrupt
#define USB_INTEP_HOST_OUT_5   0x00000020 // Endpoint 5 Host OUT Interrupt
#define USB_INTEP_HOST_OUT_4   0x00000010 // Endpoint 4 Host OUT Interrupt
#define USB_INTEP_HOST_OUT_3   0x00000008 // Endpoint 3 Host OUT Interrupt
#define USB_INTEP_HOST_OUT_2   0x00000004 // Endpoint 2 Host OUT Interrupt
#define USB_INTEP_HOST_OUT_1   0x00000002 // Endpoint 1 Host OUT Interrupt
#define USB_INTEP_DEV_IN       0x0000FFFE // Device IN Interrupts
#define USB_INTEP_DEV_IN_15    0x00008000 // Endpoint 15 Device IN Interrupt
#define USB_INTEP_DEV_IN_14    0x00004000 // Endpoint 14 Device IN Interrupt
#define USB_INTEP_DEV_IN_13    0x00002000 // Endpoint 13 Device IN Interrupt

```

```

#define USB_INTEP_DEV_IN_12    0x00001000 // Endpoint 12 Device IN Interrupt
#define USB_INTEP_DEV_IN_11    0x00000800 // Endpoint 11 Device IN Interrupt
#define USB_INTEP_DEV_IN_10    0x00000400 // Endpoint 10 Device IN Interrupt
#define USB_INTEP_DEV_IN_9     0x00000200 // Endpoint 9 Device IN Interrupt
#define USB_INTEP_DEV_IN_8     0x00000100 // Endpoint 8 Device IN Interrupt
#define USB_INTEP_DEV_IN_7     0x00000080 // Endpoint 7 Device IN Interrupt
#define USB_INTEP_DEV_IN_6     0x00000040 // Endpoint 6 Device IN Interrupt
#define USB_INTEP_DEV_IN_5     0x00000020 // Endpoint 5 Device IN Interrupt
#define USB_INTEP_DEV_IN_4     0x00000010 // Endpoint 4 Device IN Interrupt
#define USB_INTEP_DEV_IN_3     0x00000008 // Endpoint 3 Device IN Interrupt
#define USB_INTEP_DEV_IN_2     0x00000004 // Endpoint 2 Device IN Interrupt
#define USB_INTEP_DEV_IN_1     0x00000002 // Endpoint 1 Device IN Interrupt
#define USB_INTEP_0            0x00000001 // Endpoint 0 Interrupt

```

void USBIntDisableEndpoint(unsigned long ulBase, unsigned long ulIntFlags);

作用：禁止 USB 端点中断。

参数：ulBase，指定 USB 模块，在 Stellaris USB 处理器中只有一个 USB 模块，所以参数为 USB0_BASE。ulIntFlags，中断标志。

返回：无。

unsigned long USBIntStatusEndpoint(unsigned long ulBase);

作用：获取 USB 端点中断标志。

参数：ulBase，指定 USB 模块，在 Stellaris USB 处理器中只有一个 USB 模块，所以参数为 USB0_BASE。

返回：中断标志，与 USBIntEnableEndpoint 的 ulIntFlags 标志相同。

void USBIntEnable(unsigned long ulBase, unsigned long ulIntFlags);

作用：使能 USB 中断，包括端点中断。

参数：ulBase，指定 USB 模块，在 Stellaris USB 处理器中只有一个 USB 模块，所以参数为 USB0_BASE。ulIntFlags，中断标志。

返回：无。

ulIntFlags 可选参数：

```

#define USB_INT_ALL            0xFF030E0F // All Interrupt sources
#define USB_INT_STATUS        0xFF000000 // Status Interrupts
#define USB_INT_VBUS_ERR      0x80000000 // VBUS Error
#define USB_INT_SESSION_START 0x40000000 // Session Start Detected
#define USB_INT_SESSION_END   0x20000000 // Session End Detected
#define USB_INT_DISCONNECT    0x20000000 // Disconnect Detected
#define USB_INT_CONNECT       0x10000000 // Device Connect Detected
#define USB_INT_SOF           0x08000000 // Start of Frame Detected
#define USB_INT_BABBLE        0x04000000 // Babble signaled
#define USB_INT_RESET         0x04000000 // Reset signaled
#define USB_INT_RESUME        0x02000000 // Resume detected
#define USB_INT_SUSPEND       0x01000000 // Suspend detected
#define USB_INT_MODE_DETECT    0x00020000 // Mode value valid
#define USB_INT_POWER_FAULT    0x00010000 // Power Fault detected

```



```

#define USB_INT_HOST_IN          0x00000E00 // Host IN Interrupts
#define USB_INT_DEV_OUT          0x00000E00 // Device OUT Interrupts
#define USB_INT_HOST_IN_EP3      0x00000800 // Endpoint 3 Host IN Interrupt
#define USB_INT_HOST_IN_EP2      0x00000400 // Endpoint 2 Host IN Interrupt
#define USB_INT_HOST_IN_EP1      0x00000200 // Endpoint 1 Host IN Interrupt
#define USB_INT_DEV_OUT_EP3      0x00000800 // Endpoint 3 Device OUT Interrupt
#define USB_INT_DEV_OUT_EP2      0x00000400 // Endpoint 2 Device OUT Interrupt
#define USB_INT_DEV_OUT_EP1      0x00000200 // Endpoint 1 Device OUT Interrupt
#define USB_INT_HOST_OUT         0x0000000E // Host OUT Interrupts
#define USB_INT_DEV_IN           0x0000000E // Device IN Interrupts
#define USB_INT_HOST_OUT_EP3     0x00000008 // Endpoint 3 HOST_OUT Interrupt
#define USB_INT_HOST_OUT_EP2     0x00000004 // Endpoint 2 HOST_OUT Interrupt
#define USB_INT_HOST_OUT_EP1     0x00000002 // Endpoint 1 HOST_OUT Interrupt
#define USB_INT_DEV_IN_EP3       0x00000008 // Endpoint 3 DEV_IN Interrupt
#define USB_INT_DEV_IN_EP2       0x00000004 // Endpoint 2 DEV_IN Interrupt
#define USB_INT_DEV_IN_EP1       0x00000002 // Endpoint 1 DEV_IN Interrupt
#define USB_INT_EP0              0x00000001 // Endpoint 0 Interrupt

```

```
void USBIntDisable(unsigned long ulBase, unsigned long ulIntFlags);
```

作用：禁止 USB 通用中断，包括端点中断。

参数：ulBase，指定 USB 模块，在 Stellaris USB 处理器中只有一个 USB 模块，所以参数为 USB0_BASE。ulIntFlags，中断标志。

返回：无。

```
unsigned long USBIntStatus(unsigned long ulBase);
```

作用：获取中断标志。

参数：ulBase，指定 USB 模块，在 Stellaris USB 处理器中只有一个 USB 模块，所以参数为 USB0_BASE。

返回：返回中断标志，与 USBIntEnable 的 ulIntFlags 标志相同。

这些函数中 USBIntEnableControl、USBIntDisableControl、USBIntStatusControl 控制除端点外的 USB 通用中断；USBIntEnableEndpoint、USBIntDisableEndpoint、USBIntStatusEndpoint 控制 USB 端点中断，端点数量可达 16 个，包括端点 0；USBIntEnable、USBIntDisable、USBIntStatus 控制 USB 中断，包括通用中断和端点中断，但是端点中断只可以控制 4 个，端点 0 到端点 3。

```
unsigned long USBFrameNumberGet(unsigned long ulBase);
```

作用：获取当前帧编号。

参数：ulBase，指定 USB 模块，在 Stellaris USB 处理器中只有一个 USB 模块，所以参数为 USB0_BASE。

返回：返回当前帧编号。

```
void USBOTGSessionRequest(unsigned long ulBase, tBoolean bStart);
```

作用：OTG 启动会话。

参数：ulBase，指定 USB 模块，在 Stellaris USB 处理器中只有一个 USB 模块，所以参数为 USB0_BASE。bStart，会话启动还是停止。

返回：无。

```
unsigned long USBModeGet(unsigned long ulBase);
```

作用：获取 USB 工作模式。

参数：ulBase，指定 USB 模块，在 Stellaris USB 处理器中只有一个 USB 模块，所以参数为 USB0_BASE。

返回：工作模式。

返回值为：

```
#define USB_DUAL_MODE_HOST      0x00000001 // Dual mode controller is in Host
#define USB_DUAL_MODE_DEVICE    0x00000081 // Dual mode controller is in Device
#define USB_DUAL_MODE_NONE      0x00000080 // Dual mode controller mode not set
#define USB_OTG_MODE_ASIDE_HOST 0x0000001d // OTG A side
#define USB_OTG_MODE_ASIDE_NPWR 0x00000001 // OTG A side
#define USB_OTG_MODE_ASIDE_SESS 0x00000009 // OTG A side of cable Session Valid.
#define USB_OTG_MODE_ASIDE_AVAL 0x00000011 // OTG A side of the cable A valid.
#define USB_OTG_MODE_ASIDE_DEV  0x00000019 // OTG A side of the cable.
#define USB_OTG_MODE_BSIDE_HOST 0x0000009d // OTG B side of the cable.
#define USB_OTG_MODE_BSIDE_DEV  0x00000099 // OTG B side of the cable.
#define USB_OTG_MODE_BSIDE_NPWR 0x00000081 // OTG B side of the cable.
#define USB_OTG_MODE_NONE       0x00000080 // OTG controller mode is not set.
```

本节主要介绍 USB 基本操作 API，包含 USBEndpoint 组 API、USBFIFO 组 API、其它公用 API。通过以上学习，可以对 USB 底层操作做一个大概了解，为以后协议理解打好基础。

3.4 设备库函数

设备库函数 API，只包含设备能够使用的 API 函数。要开发 USB 设备还需要与前面章节介绍的 API 结合，才能完成 USB 设备功能。

```
void USBDevAddrSet(unsigned long ulBase, unsigned long ulAddress);
unsigned long USBDevAddrGet(unsigned long ulBase);
void USBDevConnect(unsigned long ulBase);
void USBDevDisconnect(unsigned long ulBase);
void USBDevEndpointConfigSet(unsigned long ulBase,
                              unsigned long ulEndpoint,
                              unsigned long ulMaxPacketSize,
                              unsigned long ulFlags);
void USBDevEndpointConfigGet(unsigned long ulBase,
                              unsigned long ulEndpoint,
                              unsigned long *pulMaxPacketSize,
                              unsigned long *pulFlags);
void USBDevEndpointDataAck(unsigned long ulBase,
                            unsigned long ulEndpoint,
                            tBoolean bIsLastPacket);
void USBDevEndpointStall(unsigned long ulBase, unsigned long ulEndpoint,
                          unsigned long ulFlags);
void USBDevEndpointStallClear(unsigned long ulBase,
                              unsigned long ulEndpoint,
                              unsigned long ulFlags);
void USBDevEndpointStatusClear(unsigned long ulBase,
```

```

        unsigned long ulEndpoint,
        unsigned long ulFlags);

void USBDevAddrSet(unsigned long ulBase, unsigned long ulAddress);
作用：设置设备地址。
参数：ulBase，指定 USB 模块，在 Stellaris USB 处理器中只有一个 USB 模块，所以参
数为 USB0_BASE。ulAddress，主机 USBREQ_SET_ADDRESS 请求命令时
tUSBRequest.wValue 值，用于配置设备地址。
返回：无。

unsigned long USBDevAddrGet(unsigned long ulBase);
作用：获取设备地址。
参数：ulBase，指定 USB 模块，在 Stellaris USB 处理器中只有一个 USB 模块，所以参
数为 USB0_BASE。
返回：设备地址。

void USBDevConnect(unsigned long ulBase);
作用：软件连接到 USB 主机。
参数：ulBase，指定 USB 模块，在 Stellaris USB 处理器中只有一个 USB 模块，所以参
数为 USB0_BASE。
返回：无。

void USBDevDisconnect(unsigned long ulBase);
作用：软件断开设备与主机的连接。
参数：ulBase，指定 USB 模块，在 Stellaris USB 处理器中只有一个 USB 模块，所以参
数为 USB0_BASE。
返回：无。

void USBDevEndpointConfigSet(unsigned long ulBase, unsigned long ulEndpoint,
        unsigned long ulMaxPacketSize, unsigned long ulFlags);
作用：在设备模式下，配置端点的最大数据包大小，及其它配置。
参数：ulBase，指定 USB 模块，在 Stellaris USB 处理器中只有一个 USB 模块，所以参
数为 USB0_BASE。ulEndpoint，指定操作的端点号，USB_EP_n(n=0..15)。ulMaxPacketSize，
最大数据包大小。ulFlags，关于端点的其它配置。
返回：无。

ulFlags 可选参数：
#define USB_EP_AUTO_SET          0x00000001 // Auto set feature enabled
#define USB_EP_AUTO_REQUEST      0x00000002 // Auto request feature enabled
#define USB_EP_AUTO_CLEAR        0x00000004 // Auto clear feature enabled
#define USB_EP_DMA_MODE_0        0x00000008 // Enable DMA access using mode 0
#define USB_EP_DMA_MODE_1        0x00000010 // Enable DMA access using mode 1
#define USB_EP_MODE_ISOC         0x00000000 // Isochronous endpoint
#define USB_EP_MODE_BULK         0x00000100 // Bulk endpoint
#define USB_EP_MODE_INT          0x00000200 // Interrupt endpoint
#define USB_EP_MODE_CTRL         0x00000300 // Control endpoint
#define USB_EP_MODE_MASK         0x00000300 // Mode Mask
#define USB_EP_SPEED_LOW         0x00000000 // Low Speed
#define USB_EP_SPEED_FULL        0x00001000 // Full Speed
#define USB_EP_HOST_IN           0x00000000 // Host IN endpoint
#define USB_EP_HOST_OUT          0x00002000 // Host OUT endpoint
#define USB_EP_DEV_IN            0x00002000 // Device IN endpoint

```

```
#define USB_EP_DEV_OUT          0x00000000 // Device OUT endpoint
```

```
void USBDevEndpointConfigGet(unsigned long ulBase, unsigned long ulEndpoint,  
                             unsigned long *pulMaxPacketSize, unsigned long *pulFlags);
```

作用：在设备模式下，获取端点配置。

参数：ulBase，指定 USB 模块，在 Stellaris USB 处理器中只有一个 USB 模块，所以参数为 USB0_BASE。ulEndpoint，指定操作的端点号，USB_EP_n(n=0..15)。ulMaxPacketSize，返回最大数据包大小指针。ulFlags，返回端点配置指针。

返回：无。

```
void USBDevEndpointDataAck(unsigned long ulBase, unsigned long ulEndpoint,  
                           tBoolean bIsLastPacket);
```

作用：在设备模式下，收到数据响应。

参数：ulBase，指定 USB 模块，在 Stellaris USB 处理器中只有一个 USB 模块，所以参数为 USB0_BASE。ulEndpoint，指定操作的端点号，USB_EP_n(n=0..15)。bIsLastPacket，数据是否是最后一个包。

返回：无。

```
void USBDevEndpointStall(unsigned long ulBase, unsigned long ulEndpoint,  
                        unsigned long ulFlags);
```

作用：在设备模式下，停止端点。

参数：ulBase，指定 USB 模块，在 Stellaris USB 处理器中只有一个 USB 模块，所以参数为 USB0_BASE。ulEndpoint，指定操作的端点号，USB_EP_n(n=0..15)。ulFlags，指定是 IN 端点还是 OUT 端点。

返回：无。

```
void USBDevEndpointStallClear(unsigned long ulBase, unsigned long ulEndpoint,  
                             unsigned long ulFlags);
```

作用：在设备模式下，清除端点的停止条件。

参数：ulBase，指定 USB 模块，在 Stellaris USB 处理器中只有一个 USB 模块，所以参数为 USB0_BASE。ulEndpoint，指定操作的端点号，USB_EP_n(n=0..15)。ulFlags，指定是 IN 端点还是 OUT 端点。

返回：无。

```
void USBDevEndpointStatusClear(unsigned long ulBase,  
                               unsigned long ulEndpoint,  
                               unsigned long ulFlags);
```

作用：在设备模式下，清除端点的状态。

参数：ulBase，指定 USB 模块，在 Stellaris USB 处理器中只有一个 USB 模块，所以参数为 USB0_BASE。ulEndpoint，指定操作的端点号，USB_EP_n(n=0..15)。ulFlags，与 USBEndpointStatus 返回值相同，用于指定清除端点的状态。

返回：无。

以上介绍的 API 函数只能在设备模式下调用，并且在设备模式下使用频繁。如果不正确使用这些 API 函数，可能枚举不成功；枚举成功，数据传输也可能有问题。所以开发 USB 设备，掌握这些 API 函数是必要的。

3.5 主机库函数

主机库函数 API，只包含主机能够使用的 API 函数。要开发 USB 主机还需要与前面章节介绍的 API 结合，才能完成 USB 主机功能。

```
void USBHostAddrSet(unsigned long ulBase, unsigned long ulEndpoint,  
                   unsigned long ulAddr, unsigned long ulFlags);
```

```

unsigned long USBHostAddrGet(unsigned long ulBase,
                             unsigned long ulEndpoint,
                             unsigned long ulFlags);
void USBHostEndpointConfig(unsigned long ulBase,
                            unsigned long ulEndpoint,
                            unsigned long ulMaxPacketSize,
                            unsigned long ulNAKPollInterval,
                            unsigned long ulTargetEndpoint,
                            unsigned long ulFlags);
void USBHostEndpointDataAck(unsigned long ulBase,
                             unsigned long ulEndpoint);
void USBHostEndpointDataToggle(unsigned long ulBase,
                                unsigned long ulEndpoint,
                                tBoolean bDataToggle,
                                unsigned long ulFlags);
void USBHostEndpointStatusClear(unsigned long ulBase,
                                 unsigned long ulEndpoint,
                                 unsigned long ulFlags);
void USBHostHubAddrSet(unsigned long ulBase, unsigned long ulEndpoint,
                       unsigned long ulAddr, unsigned long ulFlags);
unsigned long USBHostHubAddrGet(unsigned long ulBase,
                                unsigned long ulEndpoint,
                                unsigned long ulFlags);

void USBHostPwrEnable(unsigned long ulBase);
void USBHostPwrDisable(unsigned long ulBase);
void USBHostPwrConfig(unsigned long ulBase, unsigned long ulFlags);
void USBHostPwrFaultEnable(unsigned long ulBase);
void USBHostPwrFaultDisable(unsigned long ulBase);
void USBHostRequestIN(unsigned long ulBase, unsigned long ulEndpoint);
void USBHostRequestStatus(unsigned long ulBase);
void USBHostReset(unsigned long ulBase, tBoolean bStart);
void USBHostResume(unsigned long ulBase, tBoolean bStart);
void USBHostSuspend(unsigned long ulBase);
void USBHostMode(unsigned long ulBase);
unsigned long USBHostSpeedGet(unsigned long ulBase);

```

```

void USBHostAddrSet(unsigned long ulBase, unsigned long ulEndpoint,
                    unsigned long ulAddr, unsigned long ulFlags);

```

作用：主机端点与设备通信时，设置端点访问的设备地址。

参数：ulBase，指定 USB 模块，在 Stellaris USB 处理器中只有一个 USB 模块，所以参数为 USB0_BASE。ulEndpoint，指定操作的端点号，USB_EP_n (n=0..15)。ulFlags，指定是 IN 端点还是 OUT 端点。ulAddr，指定设备地址。

返回：无。

```

unsigned long USBHostAddrGet(unsigned long ulBase,

```

```

        unsigned long ulEndpoint,
        unsigned long ulFlags);

```

作用：主机端点与设备通信时，获取端点访问的设备地址。

参数：ulBase，指定 USB 模块，在 Stellaris USB 处理器中只有一个 USB 模块，所以参数为 USB0_BASE。ulEndpoint，指定操作的端点号，USB_EP_n (n=0..15)。ulFlags，指定是 IN 端点还是 OUT 端点。

返回：设备地址。

```

void USBHostEndpointConfig(unsigned long ulBase,
                           unsigned long ulEndpoint,
                           unsigned long ulMaxPacketSize,
                           unsigned long ulNAKPollInterval,
                           unsigned long ulTargetEndpoint,
                           unsigned long ulFlags);

```

作用：主机端点配置。

参数：ulBase，指定 USB 模块，在 Stellaris USB 处理器中只有一个 USB 模块，所以参数为 USB0_BASE。ulEndpoint，指定操作的端点号，USB_EP_n (n=0..15)。ulMaxPacketSize，ulFlags，指定是 IN 端点还是 OUT 端点。ulMaxPayload，端点的最大数据包大小。ulNAKPollInterval，NAK 超时限制或查询间隔，这取决于端点的类型。ulTargetEndpoint，目标端点。

返回：无。

```

void USBHostEndpointDataAck(unsigned long ulBase, unsigned long ulEndpoint);

```

作用：主机端点响应。

参数：ulBase，指定 USB 模块，在 Stellaris USB 处理器中只有一个 USB 模块，所以参数为 USB0_BASE。ulEndpoint，指定操作的端点号，USB_EP_n (n=0..15)。

返回：无。

```

void USBHostEndpointDataToggle(unsigned long ulBase, unsigned long ulEndpoint,
                                tBoolean bDataToggle, unsigned long ulFlags);

```

作用：主机端点数据转换。

参数：ulBase，指定 USB 模块，在 Stellaris USB 处理器中只有一个 USB 模块，所以参数为 USB0_BASE。ulEndpoint，指定操作的端点号，USB_EP_n (n=0..15)。bDataToggle，是否进行数据转换。ulFlags，指定是 IN 端点还是 OUT 端点。

返回：无。

```

void USBHostEndpointStatusClear(unsigned long ulBase,
                                 unsigned long ulEndpoint, unsigned long ulFlags);

```

作用：清除主机端点状态。

参数：ulBase，指定 USB 模块，在 Stellaris USB 处理器中只有一个 USB 模块，所以参数为 USB0_BASE。ulEndpoint，指定操作的端点号，USB_EP_n (n=0..15)。ulFlags，指定是 IN 端点还是 OUT 端点。

返回：无。

```

void USBHostHubAddrSet(unsigned long ulBase, unsigned long ulEndpoint,
                       unsigned long ulAddr, unsigned long ulFlags);

```

作用：设置集线器地址。

参数: ulBase, 指定 USB 模块, 在 Stellaris USB 处理器中只有一个 USB 模块, 所以参数为 USB0_BASE。ulEndpoint, 指定操作的端点号, USB_EP_n (n=0..15)。ulFlags, 指定是 IN 端点还是 OUT 端点。ulAddr, 集线器地址。

返回: 无。

```
unsigned long USBHostHubAddrGet(unsigned long ulBase,  
                                unsigned long ulEndpoint,  
                                unsigned long ulFlags);
```

作用: 获取集线器地址。

参数: ulBase, 指定 USB 模块, 在 Stellaris USB 处理器中只有一个 USB 模块, 所以参数为 USB0_BASE。ulEndpoint, 指定操作的端点号, USB_EP_n (n=0..15)。ulFlags, 指定是 IN 端点还是 OUT 端点。

返回: 集线器地址。

```
void USBHostPwrEnable(unsigned long ulBase);
```

作用: 主机使能外部电源。

参数: ulBase, 指定 USB 模块, 在 Stellaris USB 处理器中只有一个 USB 模块, 所以参数为 USB0_BASE。

返回: 无。

```
void USBHostPwrDisable(unsigned long ulBase);
```

作用：主机禁止外部电源。

参数：ulBase，指定 USB 模块，在 Stellaris USB 处理器中只有一个 USB 模块，所以参数为 USB0_BASE。

返回：无。

```
void USBHostPwrConfig(unsigned long ulBase, unsigned long ulFlags);
```

作用：外部电源配置。

参数：ulBase，指定 USB 模块，在 Stellaris USB 处理器中只有一个 USB 模块，所以参数为 USB0_BASE。ulFlags，电源配置。

返回：无。

```
void USBHostPwrFaultEnable(unsigned long ulBase);
```

作用：电源异常使能。

参数：ulBase，指定 USB 模块，在 Stellaris USB 处理器中只有一个 USB 模块，所以参数为 USB0_BASE。

返回：无。

```
void USBHostPwrFaultDisable(unsigned long ulBase);
```

作用：电源异常禁止。

参数：ulBase，指定 USB 模块，在 Stellaris USB 处理器中只有一个 USB 模块，所以参数为 USB0_BASE。

返回：无。

```
void USBHostRequestIN(unsigned long ulBase, unsigned long ulEndpoint);
```

作用：发送请求。

参数：ulBase，指定 USB 模块，在 Stellaris USB 处理器中只有一个 USB 模块，所以参数为 USB0_BASE。ulEndpoint，指定操作的端点号，USB_EP_n (n=0..15)。

返回：无。

```
void USBHostRequestStatus(unsigned long ulBase);
```

作用：在端点 0 上发送设备状态标准请求。

参数：ulBase，指定 USB 模块，在 Stellaris USB 处理器中只有一个 USB 模块，所以参数为 USB0_BASE。

返回：无。

```
void USBHostReset(unsigned long ulBase, tBoolean bStart);
```

作用：总线复位。

参数：ulBase，指定 USB 模块，在 Stellaris USB 处理器中只有一个 USB 模块，所以参数为 USB0_BASE。bStart，复位开始还是停止。复位开始后 20mS 后软件复位停止。

返回：无。

```
void USBHostResume(unsigned long ulBase, tBoolean bStart);
```

作用：总线唤醒。这个函数在设备模式下可以调用。

参数：ulBase，指定 USB 模块，在 Stellaris USB 处理器中只有一个 USB 模块，所以参数为 USB0_BASE。bStart，唤醒开始还是停止。主机模式下，唤醒开始后 20mS 后软件唤醒停止。在设备模式下，大于 10ms 小于 15ms。

返回：无

```
void USBHostSuspend(unsigned long ulBase);
```

作用：总线挂起。

参数：ulBase，指定 USB 模块，在 Stellaris USB 处理器中只有一个 USB 模块，所以参数为 USB0_BASE。

返回：无。

void USBHostMode(unsigned long ulBase);

作用：设置 USB 处理器工作于主机模式。

参数：ulBase，指定 USB 模块，在 Stellaris USB 处理器中只有一个 USB 模块，所以参数为 USB0_BASE。

返回：无。

unsigned long USBHostSpeedGet(unsigned long ulBase);

作用：获取与主机连接的设备速度。

参数：ulBase，指定 USB 模块，在 Stellaris USB 处理器中只有一个 USB 模块，所以参数为 USB0_BASE。

返回：设备速度。

以上介绍的 API 函数（除 USBHostResume 外）只能在主机模式下调用，并且在主机模式下使用频繁。所有主机都会使用到本节 API 函数，这些函数在 USB 主机开发中占有重要地位。

例如，主机对设备进行枚举过程控制，简单请求过程。

```
static void USBHCDEnumHandler(void)
{
    unsigned long ulEPStatus;
    unsigned long ulDataSize;
    //获取端点 0 的状态
    ulEPStatus = USBEndpointStatus(USB0_BASE, USB_EP_0);
    //判断端点 0 状态
    if(ulEPStatus == USB_HOST_EP0_ERROR)
    {
        //如果端点 0 出现问题，清除端点 0 状态。
        USBHostEndpointStatusClear(USB0_BASE, USB_EP_0, USB_HOST_EP0_ERROR);
        //清空端点 0 的 FIFO 数据
        USBFIFOFlush(USB0_BASE, USB_EP_0, 0);
        g_sUSBHEP0State.eState = EP0_STATE_ERROR;
        return;
    }
    //根据端点 0 的状态进行处理
    switch(g_sUSBHEP0State.eState)
    {
        case EP0_STATE_STATUS:
        {
            //处理接收到的数据
            if(ulEPStatus & (USB_HOST_EP0_RXPKTRDY | USB_HOST_EP0_STATUS))
            {
                //清除接收状态
                USBHostEndpointStatusClear(USB0_BASE, USB_EP_0,
                    (USB_HOST_EP0_RXPKTRDY |
                     USB_HOST_EP0_STATUS));
            }
            g_sUSBHEP0State.eState = EP0_STATE_IDLE;
        }
    }
}
```

```

        break;
    }
    case EPO_STATE_STATUS_IN:
    {
        //主机发送状态请求
        USBHostRequestStatus(USB0_BASE);
        g_sUSBHEP0State.eState = EPO_STATE_STATUS;
        break;
    }
    case EPO_STATE_IDLE:
    {
        break;
    }
    case EPO_STATE_SETUP_OUT:
    {
        //通过端点 0 发送数据
        USBHCDEP0StateTx();
        break;
    }
    case EPO_STATE_SETUP_IN:
    {
        //主机发出请求
        USBHostRequestIN(USB0_BASE, USB_EP_0);
        g_sUSBHEP0State.eState = EPO_STATE_RX;
        break;
    }
    case EPO_STATE_RX:
    {
        if(ulEPStatus & USB_HOST_EP0_RX_STALL)
        {
            g_sUSBHEP0State.eState = EPO_STATE_IDLE;
            USBHostEndpointStatusClear(USB0_BASE, USB_EP_0, ulEPStatus);
            break;
        }
        if(g_sUSBHEP0State.ulBytesRemaining > MAX_PACKET_SIZE_EP0)
        {
            ulDataSize = MAX_PACKET_SIZE_EP0;
        }
        else
        {
            ulDataSize = g_sUSBHEP0State.ulBytesRemaining;
        }
        if(ulDataSize != 0)
        {

```

```

        //从端点 0 中获取数据
        USBEndpointDataGet(USB0_BASE, USB_EP_0, g_sUSBHEP0State.pData,
                            &ulDataSize);
    }
    g_sUSBHEP0State.pData += ulDataSize;
    g_sUSBHEP0State.ulBytesRemaining -= ulDataSize;
    //主机响应
    USBDevEndpointDataAck(USB0_BASE, USB_EP_0, false);
    if((ulDataSize < g_sUSBHEP0State.ulMaxPacketSize) ||
        (g_sUSBHEP0State.ulBytesRemaining == 0))
    {
        g_sUSBHEP0State.eState = EPO_STATE_STATUS;
        //数据接收完，发送 NULL 包
        USBEndpointDataSend(USB0_BASE, USB_EP_0, USB_TRANS_STATUS);
    }
    else
    {
        //主机请求
        USBHostRequestIN(USB0_BASE, USB_EP_0);
    }
    break;
}
case EPO_STATE_STALL:
{
    g_sUSBHEP0State.eState = EPO_STATE_IDLE;
    break;
}
default:
{
    ASSERT(0);
    break;
}
}
}

```

端点 0 发送数据:

```

static void USBHCDEP0StateTx(void)
{
    unsigned long ulNumBytes;
    unsigned char *pData;
    g_sUSBHEP0State.eState = EPO_STATE_SETUP_OUT;
    //设置发送数据字节数
    ulNumBytes = g_sUSBHEP0State.ulBytesRemaining;
    //判断大于端点 0 的最大包长度 64。
    if(ulNumBytes > 64)

```

```

{
    ulNumBytes = 64;
}
// 指向待发数据
pData = (unsigned char *)g_sUSBHEP0State.pData;
g_sUSBHEP0State.ulBytesRemaining -= ulNumBytes;
g_sUSBHEP0State.pData += ulNumBytes;
// 放入 FIO 中
USBEndpointDataPut(USB0_BASE, USB_EP_0, pData, ulNumBytes);
//判断是否是最后一个包，并存入包结束标志
if(ulNumBytes == 64)
{
    USBEndpointDataSend(USB0_BASE, USB_EP_0, USB_TRANS_OUT);
}
else
{
    USBEndpointDataSend(USB0_BASE, USB_EP_0, USB_TRANS_OUT);
    g_sUSBHEP0State.eState = EP0_STATE_STATUS_IN;
}
}

```

本章介绍了 USB 处理器底层 API 操作函数的使用,有助于加深读者对 USB 设备和主机控制。对于具体的 USB 开发,使用现有的 API 函数即可完成。由于 USB 协议的复杂性,从第四章开始学习使用 Luminary Micro 公司提供的 Stellaris Cortex-m3 USB 处理器的 USB 应用性库函数进行实际 USB 工程开发。

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

第四章 USB 库介绍

4.1 USB 库函数简介

Luminary Micro 公司提供 USB 处理器的 USB 库函数，应用在 Stellaris 处理器上，为 USB 设备、USB 主机、OTG 开发提供 USB 协议框架和 API 函数，适用于多种开发环境：Keil、CSS、IAR、CRT、CCS 等。本书中的所有例程都在 Keil uv4 中编译。

使用 USB 库开发时，要加入两个已经编译好的 .lib。KEIL 中建立 USB 开发工程结构如图 1 所示：

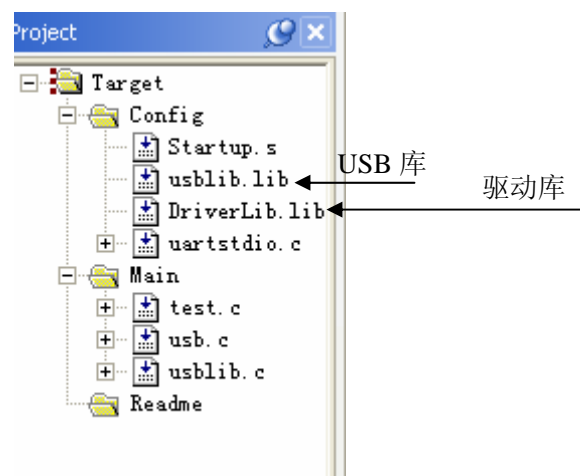


图 1 文件组织结构

在使用 USB 库之前必须了解 USB 库的结构，有助于开发者其理解与使用。USB 库分为三个层次：USB 设备 API、设备类驱动、设备类，如图 2 USB 库架构：

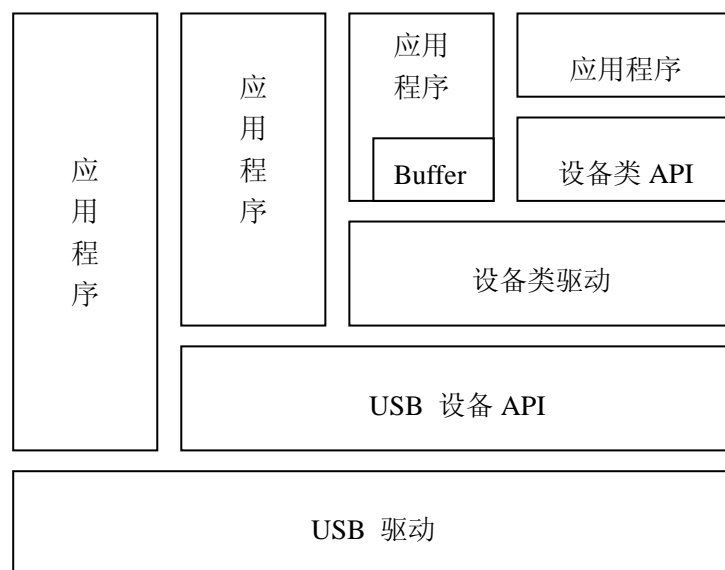


图 2 USB 库架构

从图 2 中可以看出，最底层驱动是第三章讲的 USB 驱动程序，只使用 USB 驱动程序可以进行简单的 USB 开发。对于更为复杂的 USB 工程，仅仅使用驱动程序开发是很困难的。在引入 USB 库后，可以很方便、简单进行复杂的 USB 工程设计。USB 库提供三层 API，底层为 USB 设备 API，提供最基础的 USB 协议和类定义；USB 设备驱动是在 USB 设备 API 基础上扩展的 USB 各种设备驱动，比如 HID 类、CDC 类等类驱动；为了方便程序员使用，还提供设备类 API，扩展 USB 库的使用范围，进一步减轻开发人员的负担，在不用考虑更底层驱动情况下完成 USB 工程开发。

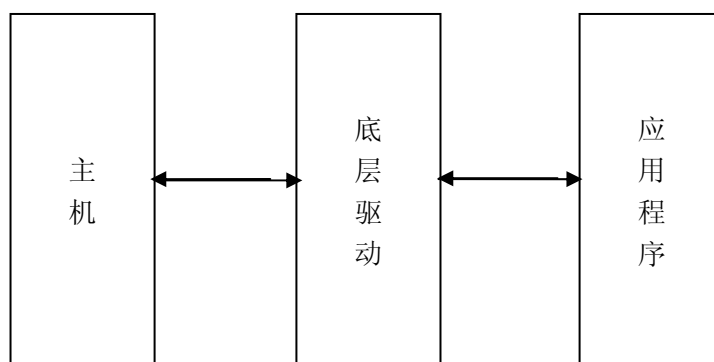


图 3

同时开发人员可有多种选择，开发 USB 设备。如图 3，开发人员可以使用最底层的 API 驱动函数进行开发，应用程序通过底层驱动与 USB 主机通信、控制。但要求开发人员对 USB 协议完全了解，并熟练于协议编写。

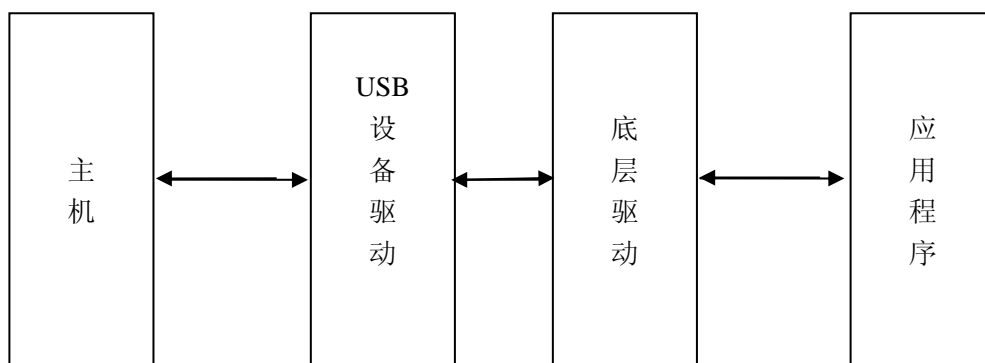


图 4

如图 4，开发人员可以在最底层的 API 驱动函数基础上使用 USB 库函数的 USB 设备驱动函数，进行 USB 控制，应用程序通过底层驱动和 USB 设备驱动与 USB 主机通信、控制。减轻了开发人员的负担。



图 5

如图 5，开发人员可以利用最底层的 API 驱动函数、USB 设备驱动函数和设备类驱动函数，进行 USB 开发。设备类驱动主要提供各种 USB 设备类的驱动，比如 Audio 类驱动、HID 类驱动、Composite 类驱动、CDC 类驱动、Bulk 类驱动、Mass Storage 类驱动等 6 种基本类驱动，其它设备类驱动可以参考这 5 种驱动的源码，由开发者自己编写。

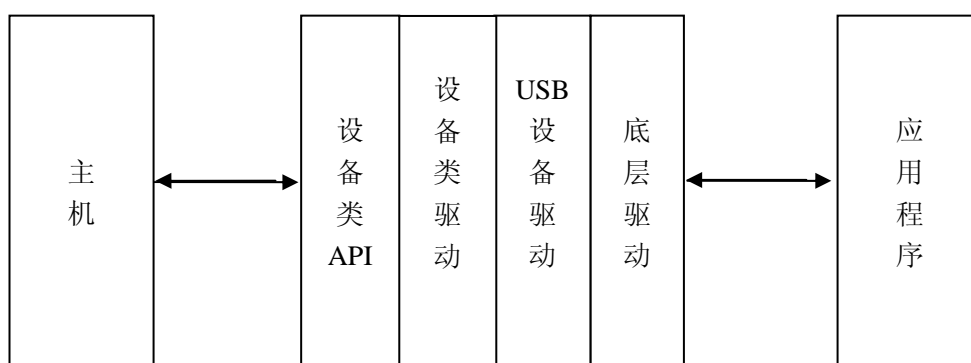


图 6

如图 6，开发人员可以利用最底层的 API 驱动函数、USB 设备驱动函数、设备类驱动函数和设备类 API，进行 USB 开发。设备类 API 主要提供各种 USB 设备类操作相关的函数，比

如 HID 中的键盘、鼠标操作接口。设备类 API 也只提供了 5 种 USB 设备类 API，其它不常用的需要开发者自己编写。

Luminary Micro 公司提供 USB 函数库支持多种使用方法，完全能够满足 USB 产品开发，并且使用方便、快捷。

4.2 使用底层驱动开发

使用最底层 USB 驱动开发，要求开发人员对 USB 协议及相关事务彻底了解，开发 USB 产品有一定难度，但是这种开发模式占用内存少，运行效率高。但有容易出现 bug。

例如，使用底层 USB 驱动开发，开发一个音频设备。

第一：初始化 usb 处理器，包括内核电压、CPU 主频、USB 外设资源等。

```
SysCtlLDOSet(SYSCTL_LDO_2_75V);
// 主频 50MHz
SysCtlClockSet(SYSCTL_XTAL_8MHZ | SYSCTL_SYSDIV_4 | SYSCTL_USE_PLL | SYSCTL_OSC_MAIN);
//打开 USB 外设
SysCtlPeripheralEnable(SYSCTL_PERIPH_USB0);
//打开 USB 主时钟
SysCtlUSBPLLEnable();
//清除中断标志，并重新打开中断
USBIntStatusControl(USB0_BASE);
USBIntStatusEndpoint(USB0_BASE);
USBIntEnableControl(USB0_BASE, USB_INTCTRL_RESET |
                    USB_INTCTRL_DISCONNECT |
                    USB_INTCTRL_RESUME |
                    USB_INTCTRL_SUSPEND |
                    USB_INTCTRL_SOF);
USBIntEnableEndpoint(USB0_BASE, USB_INTEP_ALL);
//断开→连接
USBDevDisconnect(USB0_BASE);
SysCtlDelay(SysCtlClockGet() / 30);
USBDevConnect(USB0_BASE);
//使能总中断
IntEnable(INT_USB0);
//音频设备会传输大量数据，打开 DMA 最好。
uDMAChannelControlSet(psDevice->psPrivateData->ucOUTDMA,
                      (UDMA_SIZE_32 | UDMA_SRC_INC_NONE |
                       UDMA_DST_INC_32 | UDMA_ARB_16));
USBEndpointDMAChannel(USB0_BASE, psDevice->psPrivateData->ucOUTEndpoint,
                      psDevice->psPrivateData->ucOUTDMA);
```

第二：USB 音频设备描述符。

```
//语言描述符
const unsigned char g_pLangDescriptor[] =
{
    4,
    USB_DTYPE_STRING,
    USBShort(USB_LANG_EN_US)
};
//制造商 字符串 描述符
const unsigned char g_pManufacturerString[] =
{
    (17 + 1) * 2,
    USB_DTYPE_STRING,
    'T', 0, 'e', 0, 'x', 0, 'a', 0, 's', 0, ' ', 0, 'I', 0, 'n', 0, 's', 0,
    't', 0, 'r', 0, 'u', 0, 'm', 0, 'e', 0, 'n', 0, 't', 0, 's', 0,
};
//产品 字符串 描述符
```



```

const unsigned char g_pProductString[] =
{
    (13 + 1) * 2,
    USB_DTYPE_STRING,
    'A', 0, 'u', 0, 'd', 0, 'i', 0, 'o', 0, ' ', 0, 'E', 0, 'x', 0, 'a', 0,
    'm', 0, 'p', 0, 'l', 0, 'e', 0
};
//产品 序列号 描述符
const unsigned char g_pSerialNumberString[] =
{
    (8 + 1) * 2,
    USB_DTYPE_STRING,
    '1', 0, '2', 0, '3', 0, '4', 0, '5', 0, '6', 0, '7', 0, '8', 0
};
//设备接口字符串描述符
const unsigned char g_pInterfaceString[] =
{
    (15 + 1) * 2,
    USB_DTYPE_STRING,
    'A', 0, 'u', 0, 'd', 0, 'i', 0, 'o', 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符
const unsigned char g_pConfigString[] =
{
    (20 + 1) * 2,
    USB_DTYPE_STRING,
    'A', 0, 'u', 0, 'd', 0, 'i', 0, 'o', 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
};
//字符串描述符集合.
const unsigned char * const g_pStringDescriptors[] =
{
    g_pLangDescriptor,
    g_pManufacturerString,
    g_pProductString,
    g_pSerialNumberString,
    g_pInterfaceString,
    g_pConfigString
};
//音频设备描述符
static unsigned char g_pAudioDeviceDescriptor[] =
{
    18, // Size of this structure.
    USB_DTYPE_DEVICE, // Type of this structure.
    USBShort(0x110), // USB version 1.1 (if we say 2.0, hosts assume
                    // high-speed - see USB 2.0 spec 9.2.6.6)
    USB_CLASS_AUDIO, // USB Device Class (spec 5.1.1)
    USB_SUBCLASS_UNDEFINED, // USB Device Sub-class (spec 5.1.1)
    USB_PROTOCOL_UNDEFINED, // USB Device protocol (spec 5.1.1)
    64, // Maximum packet size for default pipe.
    USBShort(0x1111), // Vendor ID (filled in during USBDAudioInit).
    USBShort(0xffee), // Product ID (filled in during USBDAudioInit).
    USBShort(0x100), // Device Version BCD.
    1, // Manufacturer string identifier.
    2, // Product string identifier.
    3, // Product serial number.
    1 // Number of configurations.
};

```

```

};
//音频配置描述符
static unsigned char g_pAudioDescriptor[] =
{
    9, // Size of the configuration descriptor.
    USB_DTYPE_CONFIGURATION, // Type of this descriptor.
    USBShort(32), // The total size of this full structure.
    2, // The number of interfaces in this
        // configuration.
    1, // The unique value for this configuration.
    0, // The string identifier that describes this
        // configuration.
    USB_CONF_ATTR_BUS_PWR, // Bus Powered, Self Powered, remote wake up.
    250, // The maximum power in 2mA increments.
};
//音频接口描述符
unsigned char g_pIADAudioDescriptor[] =
{
    8, // Size of the interface descriptor.
    USB_DTYPE_INTERFACE_ASC, // Interface Association Type.
    0x0, // Default starting interface is 0.
    0x2, // Number of interfaces in this association.
    USB_CLASS_AUDIO, // The device class for this association.
    USB_SUBCLASS_UNDEFINED, // The device subclass for this association.
    USB_PROTOCOL_UNDEFINED, // The protocol for this association.
    0 // The string index for this association.
};
//音频控制接口描述符
const unsigned char g_pAudioControlInterface[] =
{
    9, // Size of the interface descriptor.
    USB_DTYPE_INTERFACE, // Type of this descriptor.
    AUDIO_INTERFACE_CONTROL, // The index for this interface.
    0, // The alternate setting for this interface.
    0, // The number of endpoints used by this
        // interface.
    USB_CLASS_AUDIO, // The interface class
    USB_ASC_AUDIO_CONTROL, // The interface sub-class.
    0, // The interface protocol for the sub-class
        // specified above.
    0, // The string index for this interface.

    // Audio Header Descriptor.
    9, // The size of this descriptor.
    USB_DTYPE_CS_INTERFACE, // Interface descriptor is class specific.
    USB_ACDSTYPE_HEADER, // Descriptor sub-type is HEADER.
    USBShort(0x0100), // Audio Device Class Specification Release
        // Number in Binary-Coded Decimal.
        // Total number of bytes in
        // g_pAudioControlInterface
    USBShort((9 + 9 + 12 + 13 + 9)),
    1, // Number of streaming interfaces.
    1, // Index of the first and only streaming
        // interface.

    // Audio Input Terminal Descriptor.
    12, // The size of this descriptor.
    USB_DTYPE_CS_INTERFACE, // Interface descriptor is class specific.

```

```

USB_ACDSTYPE_IN_TERMINAL, // Descriptor sub-type is INPUT_TERMINAL.
AUDIO_IN_TERMINAL_ID,     // Terminal ID for this interface.
                           // USB streaming interface.
USBShort(USB_TTYPE_STREAMING),
0,                         // ID of the Output Terminal to which this
                           // Input Terminal is associated.
2,                         // Number of logical output channels in the
                           // Terminal 组 output audio channel cluster.
USBShort((USB_CHANNEL_L | // Describes the spatial location of the
          USB_CHANNEL_R)), // logical channels.
0,                         // Channel Name string index.
0,                         // Terminal Name string index.

// Audio Feature Unit Descriptor
13,                       // The size of this descriptor.
USB_DTYPE_CS_INTERFACE,   // Interface descriptor is class specific.
USB_ACDSTYPE_FEATURE_UNIT, // Descriptor sub-type is FEATURE_UNIT.
AUDIO_CONTROL_ID,         // Unit ID for this interface.
AUDIO_IN_TERMINAL_ID,     // ID of the Unit or Terminal to which this
                           // Feature Unit is connected.
2,                         // Size in bytes of an element of the
                           // bmaControls() array that follows.
                           // Master Mute control.
USBShort(USB_ACONTROL_MUTE),
                           // Left channel volume control.
USBShort(USB_ACONTROL_VOLUME),
                           // Right channel volume control.
USBShort(USB_ACONTROL_VOLUME),
0,                         // Feature unit string index.

// Audio Output Terminal Descriptor.
9,                         // The size of this descriptor.
USB_DTYPE_CS_INTERFACE,   // Interface descriptor is class specific.
USB_ACDSTYPE_OUT_TERMINAL, // Descriptor sub-type is INPUT_TERMINAL.
AUDIO_OUT_TERMINAL_ID,    // Terminal ID for this interface.
                           // Output type is a generic speaker.
USBShort(USB_ATTTYPE_SPEAKER),
AUDIO_IN_TERMINAL_ID,     // ID of the input terminal to which this
                           // output terminal is connected.
AUDIO_CONTROL_ID,         // ID of the feature unit that this output
                           // terminal is connected to.
0,                         // Output terminal string index.
};
//音频流接口描述符
const unsigned char g_pAudioStreamInterface[] =
{
    9,                     // Size of the interface descriptor.
    USB_DTYPE_INTERFACE,   // Type of this descriptor.
    AUDIO_INTERFACE_OUTPUT, // The index for this interface.
    0,                     // The alternate setting for this interface.
    0,                     // The number of endpoints used by this
                           // interface.
    USB_CLASS_AUDIO,       // The interface class
    USB_ASC_AUDIO_STREAMING, // The interface sub-class.
    0,                     // Unused must be 0.
    0,                     // The string index for this interface.

    // Vendor-specific Interface Descriptor.
    9,                     // Size of the interface descriptor.

```

```

USB_DTYPE_INTERFACE,      // Type of this descriptor.
1,                        // The index for this interface.
1,                        // The alternate setting for this interface.
1,                        // The number of endpoints used by this
                          // interface.
USB_CLASS_AUDIO,          // The interface class
USB_ASC_AUDIO_STREAMING,  // The interface sub-class.
0,                        // Unused must be 0.
0,                        // The string index for this interface.

// Class specific Audio Streaming Interface descriptor.
7,                        // Size of the interface descriptor.
USB_DTYPE_CS_INTERFACE,   // Interface descriptor is class specific.
USB_ASDSTYPE_GENERAL,     // General information.
AUDIO_IN_TERMINAL_ID,     // ID of the terminal to which this streaming
                          // interface is connected.
1,                        // One frame delay.
USBShort(USB_ADF_PCM),    //

// Format type Audio Streaming descriptor.
11,                       // Size of the interface descriptor.
USB_DTYPE_CS_INTERFACE,   // Interface descriptor is class specific.
USB_ASDSTYPE_FORMAT_TYPE, // Audio Streaming format type.
USB_AF_TYPE_TYPE_I,       // Type I audio format type.
2,                        // Two audio channels.
2,                        // Two bytes per audio sub-frame.
16,                       // 16 bits per sample.
1,                        // One sample rate provided.
USB3Byte(48000),          // Only 48000 sample rate supported.

// Endpoint Descriptor
9,                        // The size of the endpoint descriptor.
USB_DTYPE_ENDPOINT,       // Descriptor type is an endpoint.
                          // OUT endpoint with address
                          // ISOC_OUT_ENDPOINT.
USB_EP_DESC_OUT | USB_EP_TO_INDEX(ISOC_OUT_ENDPOINT),
USB_EP_ATTR_ISOC |        // Endpoint is an adaptive isochronous data
USB_EP_ATTR_ISOC_ADAPT |  // endpoint.
USB_EP_ATTR_USAGE_DATA,
USBShort(ISOC_OUT_EP_MAX_SIZE), // The maximum packet size.
1,                        // The polling interval for this endpoint.
0,                        // Refresh is unused.
0,                        // Synch endpoint address.

// Audio Streaming Isochronous Audio Data Endpoint Descriptor
7,                        // The size of the descriptor.
USB_ACSDT_ENDPOINT,       // Audio Class Specific Endpoint Descriptor.
USB_ASDSTYPE_GENERAL,     // This is a general descriptor.
USB_EP_ATTR_ACG_SAMPLING, // Sampling frequency is supported.
USB_EP_LOCKDELAY_UNDEF,   // Undefined lock delay units.
USBShort(0),              // No lock delay.
};

```

第三：USB 音频设备枚举。

/枚举用到函数

```

static void USBDGetStatus(tUSBRequest *pUSBRequest);
static void USBDClearFeature(tUSBRequest *pUSBRequest);
static void USBDSetFeature(tUSBRequest *pUSBRequest);
static void USBDSetAddress(tUSBRequest *pUSBRequest);
static void USBDGetDescriptor(tUSBRequest *pUSBRequest);

```

```

static void USBSetDescriptor(tUSBRequest *pUSBRequest);
static void USBGetConfiguration(tUSBRequest *pUSBRequest);
static void USBSetConfiguration(tUSBRequest *pUSBRequest);
static void USBGetInterface(tUSBRequest *pUSBRequest);
static void USBSetInterface(tUSBRequest *pUSBRequest);
static void USBDEP0StateTx(void);
static long USBStringIndexFromRequest(unsigned short usLang,
                                       unsigned short usIndex);

```

//该结构能够完整表述 USB 设备枚举过程。

```

typedef struct
{
    //当前 USB 设备地址，也可以能过 DEV_ADDR_PENDING 最高位改变.
    volatile unsigned long ulDevAddress;
    //保存设备当前生效的配置.
    unsigned long ulConfiguration;
    //当前设备的接口设置
    unsigned char ucAltSetting;
    //指向端点 0 正发接收或者发送的数据组。
    unsigned char *pEP0Data;
    //指示端点 0 正发接收或者发送数据的剩下数据量。
    volatile unsigned long ulEP0DataRemain;
    //端点 0 正发接收或者发送数据的数据总量
    unsigned long ulOUTDataSize;
    //当前设备状态
    unsigned char ucStatus;
    //在处理过程中是否使用 wakeup 信号。
    tBoolean bRemoteWakeup;
    //bRemoteWakeup 信号计数。
    unsigned char ucRemoteWakeupCount;
}
tDeviceState;

```

//定义端点输出/输入。

```

#define HALT_EP_IN          0
#define HALT_EP_OUT        1
//端点 0 的状态，在枚举过程中使用。
typedef enum
{
    //等待主机请求。
    USB_STATE_IDLE,
    //通过 IN 端口 0 给主机发送数块。
    USB_STATE_TX,
    //通过 OUT 端口 0 从主机接收数据块。
    USB_STATE_RX,
    //端点 0 发送/接收完成，等待主机应答。
    USB_STATE_STATUS,
    //端点 0STALL，等待主机响应 STALL。
    USB_STATE_STALL
}
tEP0State;
//端点 0 最大传输包大小。
#define EP0_MAX_PACKET_SIZE    64
//用于指示设备地址改变
#define DEV_ADDR_PENDING       0x80000000
//总线复位后，默认的配置编号。
#define DEFAULT_CONFIG_ID      1

```

```

//REMOTE_WAKEUP 的信号毫秒数，在协议中定义为 1ms-15ms.
#define REMOTE_WAKEUP_PULSE_MS 10
//REMOTE_WAKEUP 保持 20ms.
#define REMOTE_WAKEUP_READY_MS 20
//端点 0 的读数据缓存。
static unsigned char g_pucDataBufferIn[EPO_MAX_PACKET_SIZE];
//定义当前设备状态信息实例。
static volatile tDeviceState g_sUSBDeviceState;
//定义当前端点 0 的状态
static volatile tEPOState g_eUSBDEPOState = USB_STATE_IDLE;
//请求函数表。
static const tStdRequest g_psUSBStdRequests[] =
{
    USBDGetStatus,
    USBDClearFeature,
    0,
    USBDSetFeature,
    0,
    USBDSetAddress,
    USBDGetDescriptor,
    USBDSetDescriptor,
    USBDGetConfiguration,
    USBDSetConfiguration,
    USBDGetInterface,
    USBDSetInterface,
};
//在读取 usb 中断时合并使用。
#define USB_INT_RX_SHIFT 8
#define USB_INT_STATUS_SHIFT 24
#define USB_RX_EPSTATUS_SHIFT 16
//端点控制状态寄存器转换
#define EP_OFFSET(Endpoint) (Endpoint - 0x10)

//从端点 0 的 FIFO 中获取数据。
long USBEndpoint0DataGet(unsigned char *pucData, unsigned long *pulSize)
{
    unsigned long ulByteCount;
    //判断端点 0 的数据是否接收完成。
    if((HWREGH(USB0_BASE + USB_0_CSRL0) & USB_CSRL0_RXRDY) == 0)
    {
        *pulSize = 0;
        return(-1);
    }
    //USB_0_COUNT0 指示端点 0 收到的数据量。
    ulByteCount = HWREGH(USB0_BASE + USB_0_COUNT0 + USB_EP_0);
    //确定读回的数据量。
    ulByteCount = (ulByteCount < *pulSize) ? ulByteCount : *pulSize;
    *pulSize = ulByteCount;
    //从 FIFO 中读取数据。
    for(; ulByteCount > 0; ulByteCount--)
    {
        *pucData++ = HWREGB(USB0_BASE + USB_0_FIF00 + (USB_EP_0 >> 2));
    }
    return(0);
}
//端点 0 应答。
void USBDevEndpoint0DataAck(tBoolean bIsLastPacket)
{
    HWREGB(USB0_BASE + USB_0_CSRL0) =

```

```

        USB_CSRLO_RXRDYC | (bIsLastPacket ? USB_CSRLO_DATAEND : 0);
    }
    // 向端点 0 中放入数据。
    long USBEndpoint0DataPut(unsigned char *pucData, unsigned long ulSize)
    {
        if(HWREGB(USB0_BASE + USB_0_CSRLO + USB_EP_0) & USB_CSRLO_TXRDY)
        {
            return(-1);
        }
        for(; ulSize > 0; ulSize--)
        {
            HWREGB(USB0_BASE + USB_0_FIFO0 + (USB_EP_0 >> 2)) = *pucData++;
        }
        return(0);
    }
    //向端点 0 中写入数据。
    long USBEndpoint0DataSend(unsigned long ulTransType)
    {
        //判断是否已经有数据准备好。
        if(HWREGB(USB0_BASE + USB_0_CSRLO + USB_EP_0) & USB_CSRLO_TXRDY)
        {
            return(-1);
        }
        HWREGB(USB0_BASE + USB_0_CSRLO + USB_EP_0) = ulTransType & 0xff;
        return(0);
    }
    //端点 0 从主机上获取数据。
    void USBRequestDataEP0(unsigned char *pucData, unsigned long ulSize)
    {
        g_eUSBDEP0State = USB_STATE_RX;
        g_sUSBDeviceState.pEP0Data = pucData;
        g_sUSBDeviceState.ulOUTDataSize = ulSize;
        g_sUSBDeviceState.ulEP0DataRemain = ulSize;
    }
    //端点 0 请求发送数据。
    void USBSendDataEP0(unsigned char *pucData, unsigned long ulSize)
    {
        g_sUSBDeviceState.pEP0Data = pucData;
        g_sUSBDeviceState.ulEP0DataRemain = ulSize;
        g_sUSBDeviceState.ulOUTDataSize = ulSize;
        USBDEP0StateTx();
    }
    //端点 0 处于停止状态。
    void USBStallEP0(void)
    {
        HWREGB(USB0_BASE + USB_0_CSRLO) |= (USB_CSRLO_STALL | USB_CSRLO_RXRDYC);
        g_eUSBDEP0State = USB_STATE_STALL;
    }
    //从端点 0 中获取一个请求。
    static void USBDReadAndDispatchRequest(void)
    {
        unsigned long ulSize;
        tUSBRequest *pRequest;
        pRequest = (tUSBRequest *)g_pucDataBufferIn;
        ulSize = EP0_MAX_PACKET_SIZE;
        USBEndpoint0DataGet(g_pucDataBufferIn, &ulSize);
        if(!ulSize)
        {
            return;
        }
    }

```

```

}
//判断是否是标准请求。
if((pRequest->bRequestType & USB_RTYPE_TYPE_M) != USB_RTYPE_STANDARD)
{
    UARTprintf("非标准请求.....\r\n");
}
else
{
    //调到标准请求处理函数中。
    if((pRequest->bRequest <
        (sizeof(g_psUSBStdRequests) / sizeof(tStdRequest))) &&
        (g_psUSBStdRequests[pRequest->bRequest] != 0))
    {
        g_psUSBStdRequests[pRequest->bRequest](pRequest);
    }
    else
    {
        USBStallEP0();
    }
}
}

//枚举过程。
// USB_STATE_IDLE --> USB_STATE_TX --> USB_STATE_STATUS -->USB_STATE_IDLE
//          |               |               |
//          |--> USB_STATE_RX -               |
//          |               |               |
//          |--> USB_STATE_STALL ----->-----
//
// -----
// | Current State | State 0 | State 1 |
// | ----- | ----- | ----- |
// | USB_STATE_IDLE | USB_STATE_TX/RX | USB_STATE_STALL |
// | USB_STATE_TX | USB_STATE_STATUS | |
// | USB_STATE_RX | USB_STATE_STATUS | |
// | USB_STATE_STATUS | USB_STATE_IDLE | |
// | USB_STATE_STALL | USB_STATE_IDLE | |
// | ----- | ----- | ----- |
//
void USBDeviceEnumHandler(void)
{
    unsigned long uLEPStatus;
    //获取中断状态。
    uLEPStatus = HWREGH(USB0_BASE + EP_OFFSET(USB_EP_0) + USB_0_TXCSRL1);
    uLEPStatus |= ((HWREGH(USB0_BASE + EP_OFFSET(USB_EP_0) + USB_0_RXCSRL1)) <<
        USB_RX_EPSTATUS_SHIFT);

    //端点 0 的状态。
    switch(g_eUSBDEPOState)
    {
        case USB_STATE_STATUS:
        {
            UARTprintf("USB_STATE_STATUS.....\r\n");
            g_eUSBDEPOState = USB_STATE_IDLE;
            //判断地址改变。
            if(g_sUSBDeviceState.ulDevAddress & DEV_ADDR_PENDING)
            {
                //设置地址。
                g_sUSBDeviceState.ulDevAddress &= ~DEV_ADDR_PENDING;
                HWREGB(USB0_BASE + USB_0_FADDR) =
                    (unsigned char)g_sUSBDeviceState.ulDevAddress;
            }
        }
    }
}

```



```

    }
    //端点 0 接收包准备好。
    if(ulEPStatus & USB_DEV_EP0_OUT_PKTRDY)
    {
        USBReadAndDispatchRequest();
    }
    break;
}
//等待从主机接收数据。
case USB_STATE_IDLE:
{
    if(ulEPStatus & USB_DEV_EP0_OUT_PKTRDY)
    {
        USBReadAndDispatchRequest();
    }
    break;
}
//数据处理好, 准备发送。
case USB_STATE_TX:
{
    USBDEPOStateTx();
    break;
}
//接收数据。
case USB_STATE_RX:
{
    unsigned long ulDataSize;
    if(g_sUSBDeviceState.ulEP0DataRemain > EP0_MAX_PACKET_SIZE)
    {
        ulDataSize = EP0_MAX_PACKET_SIZE;
    }
    else
    {
        ulDataSize = g_sUSBDeviceState.ulEP0DataRemain;
    }
    USBEndpoint0DataGet(g_sUSBDeviceState.pEP0Data, &ulDataSize);
    if(g_sUSBDeviceState.ulEP0DataRemain < EP0_MAX_PACKET_SIZE)
    {
        USBDevEndpoint0DataAck(true);
        g_eUSBDEPOState = USB_STATE_IDLE;
        if(g_sUSBDeviceState.ulOUTDataSize != 0)
        {
            {
            }
        }
    }
    else
    {
        USBDevEndpoint0DataAck(false);
    }
    g_sUSBDeviceState.pEP0Data += ulDataSize;
    g_sUSBDeviceState.ulEP0DataRemain -= ulDataSize;
    break;
}
//停止状态
case USB_STATE_STALL:
{
    if(ulEPStatus & USB_DEV_EP0_SENT_STALL)
    {
        HWREGB(USB0_BASE + USB_0_CSRL0) &= ~(USB_DEV_EP0_SENT_STALL);
        g_eUSBDEPOState = USB_STATE_IDLE;
    }
}

```

```

        }
        break;
    }

    default:
    {
        break;
    }
}
}

```

设备枚举过程中 USBDSetAddress 和 USBDGetDescriptor 很重要，下面只列出这两个函数的具体内容。

```

// SET_ADDRESS 标准请求。
static void USBDSetAddress(tUSBRequest *pUSBRequest)
{
    USBDevEndpoint0DataAck(true);
    g_sUSBDeviceState.ulDevAddress = pUSBRequest->wValue | DEV_ADDR_PENDING;
    g_eUSBDEPOState = USB_STATE_STATUS;
    //HandleSetAddress();
}

//GET_DESCRIPTOR 标准请求。
static void USBDGetDescriptor(tUSBRequest *pUSBRequest)
{
    USBDevEndpoint0DataAck(false);
    switch(pUSBRequest->wValue >> 8)
    {
        case USB_DTYPE_DEVICE:
        {
            g_sUSBDeviceState.pEP0Data =
                (unsigned char *)g_pDFUDeviceDescriptor;
            g_sUSBDeviceState.ulEP0DataRemain = g_pDFUDeviceDescriptor[0];
            break;
        }
        case USB_DTYPE_CONFIGURATION:
        {
            unsigned char ucIndex;
            ucIndex = (unsigned char)(pUSBRequest->wValue & 0xFF);
            if(ucIndex != 0)
            {
                USBStallEP0();
                g_sUSBDeviceState.pEP0Data = 0;
                g_sUSBDeviceState.ulEP0DataRemain = 0;
            }
            else
            {
                g_sUSBDeviceState.pEP0Data =
                    (unsigned char *)g_pDFUConfigDescriptor;
                g_sUSBDeviceState.ulEP0DataRemain =
                    *(unsigned short *)&(g_pDFUConfigDescriptor[2]);
            }
            break;
        }
        case USB_DTYPE_STRING:
        {
            long lIndex;
            lIndex = USBDStringIndexFromRequest(pUSBRequest->wIndex,
                                                pUSBRequest->wValue & 0xFF);

            if(lIndex == -1)
            {

```

```

        USBStallEP0();
        break;
    }
    g_sUSBDeviceState.pEP0Data =
        (unsigned char *)g_pStringDescriptors[lIndex];
    g_sUSBDeviceState.ulEP0DataRemain = g_pStringDescriptors[lIndex][0];
    break;
}
case 0x22:
{
    //USBDevEndpoint0DataAck(false);
    g_sUSBDeviceState.pEP0Data = (unsigned char *)ReportDescriptor;
    g_sUSBDeviceState.ulEP0DataRemain = sizeof(&ReportDescriptor[0]);
    //USBDEP0StateTx();
}
default:
{
    USBStallEP0();
    break;
}
}
if(g_sUSBDeviceState.pEP0Data)
{
    if(g_sUSBDeviceState.ulEP0DataRemain > pUSBRequest->wLength)
    {
        g_sUSBDeviceState.ulEP0DataRemain = pUSBRequest->wLength;
    }
    USBDEP0StateTx();
}
}

```

第四：USB 音频数据处理与控制。此过程包括数据处理，音量控制，静音控制等，控制过程较为复杂，在此不在一一讲解，可以参考相关 USB 音频设备书籍。在第 6 章有讲其它方法进行开发。

4.3 使用 USB 库开发

使用 USB 库函数进行开发，开发人员可以不深入研究 USB 协议，包括枚举过程、中断处理、数据处理等。使用库函数提供的 API 接口函数就可以完成开发工作。使用 USB 库函数方便、快捷、缩短开发周期、不易出现 bug，但占用存储空间、内存较大，由于 Stellaris USB 处理器的存储空间达 128K，远远超过程序需要的存储空间，所以使用 USB 库函数开发是比较好的方法。

例如：使用库函数开发一个 USB 鼠标。

1. 完成字符串描述符。

```

//语言描述符
const unsigned char g_pLangDescriptor[] =
{
    4,
    USB_DTYPE_STRING,
    USBShort(USB_LANG_EN_US)
};
//制造商 字符串 描述符
const unsigned char g_pManufacturerString[] =
{
    (17 + 1) * 2,
    USB_DTYPE_STRING,
    'T', 0, 'e', 0, 'x', 0, 'a', 0, 's', 0, ' ', 0, 'I', 0, 'n', 0, 's', 0,
    't', 0, 'r', 0, 'u', 0, 'm', 0, 'e', 0, 'n', 0, 't', 0, 's', 0,
}

```

```

};
//产品 字符串 描述符
const unsigned char g_pProductString[] =
{
    (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
};
//产品 序列号 描述符
const unsigned char g_pSerialNumberString[] =
{
    (8 + 1) * 2,
    USB_DTYPE_STRING,
    '1', 0, '2', 0, '3', 0, '4', 0, '5', 0, '6', 0, '7', 0, '8', 0
};
//设备接口字符串描述
const unsigned char g_pHIDInterfaceString[] =
{
    (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
};
// 配置字符串描述
const unsigned char g_pConfigString[] =
{
    (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
};
//字符串描述符集合
const unsigned char * const g_pStringDescriptors[] =
{
    g_pLangDescriptor,
    g_pManufacturerString,
    g_pProductString,
    g_pSerialNumberString,
    g_pHIDInterfaceString,
    g_pConfigString
};
#define NUM_STRING_DESCRIPTORS (sizeof(g_pStringDescriptors) / \
                                sizeof(unsigned char *))

```

2. 了解鼠标设备 `tUSBHIDMouseDevice` 在库函数中的定义，并完成鼠标设备实例。

```

//定义 USB 鼠标实例
tHIDMouseInstance g_sMouseInstance;
//定义 USB 鼠标相关信息
const tUSBHIDMouseDevice g_sMouseDevice =
{
    USB_VID_STELLARIS,
    USB_PID_MOUSE,
    500,
    USB_CONF_ATTR_SELF_PWR,
    MouseHandler,
    (void *)&g_sMouseDevice,
    g_pStringDescriptors,

```

```

    NUM_STRING_DESCRIPTOR,
    &g_sMouseInstance
};
3. 初始化 USB 鼠标设备，并进行数据处理。
//初始化 USB 鼠标设备，只需用这个函数就完成配置，包括枚举配置。
USBHIDMouseInit(0, (tUSBHIDMouseDevice *)&g_sMouseDevice);
//数据处理，改变鼠标位置和按键状态。
USBHIDMouseStateChange((void *)&g_sMouseDevice,
                        (char)lDeltaX, (char)lDeltaY,
                        ucButtons);

```

从库函数开发 USB 鼠标设备过程中可以看出，使用 USB 库函数开发非常简单、方便、快捷，不用考虑底层的驱动、类协议。在 HID 类中，报告符本身就很复杂，但是使用 USB 库函数开发完全屏蔽报告符配置过程。从第五章开始介绍使用 USB 库函数开发 USB 设备与主机。

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

第五章 HID 设备

5.1 HID 介绍

为简化 USB 设备的开发过程，USB 提出了设备类的概念。所有设备类都必须支持标准 USB 描述符和标准 USB 设备请求。如果有必要，设备类还可以自行定义其专用的描述符和设备请求，这分别被称为设备类定义描述符和设备类定义请求。另外，一个完整的设备类还将指明其接口和端点的使用方法，如接口所包含端点的个数、端点的最大数据包长度等。

HID 设备类就是设备类的一类，HID 是 Human Interface Device 缩写，人机交互设备，例如键盘、鼠标与游戏杆等。不过 HID 设备并不一定要有人机接口，只要符合 HID 类别规范的设备都是 HID 设备。

HID 设备既可以是低速设备也可以是全速设备，其典型的数据传输类型为中断 IN 传输，即它适用于主机接收 USB 设备发来的小量到中等量的数据。HID 具有以下功能特点：适用于传输少量或中量的数据；传输的数据具有突发性；传输的最大速率有限制；无固定的传输率。

HID 设备类除支持标准 USB 描述符外（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），还自行定义了 3 种类描述符，分别为 HID 描述符（主要用于识别 HID 设备所包含的其他类描述符）、报告描述符（提供 HID 设备和主机间交换数据的格式）和物理描述符。一个 HID 设备只能支持一个 HID 描述符；可以支持一个或多个报告描述符；物理描述符是可选的，大多数 HID 设备不需要使用它。

5.2 HID 类描述符

除了标准 USB 描述符外，HID 设备自行定义了三种描述：HID 描述符、报告描述符、物理描述符，物理描述符不常用，不在此讲解。三种描述分别定义为：

```
#define USB_HID_DTYPE_HID      0x21
#define USB_HID_DTYPE_REPORT   0x22
#define USB_HID_DTYPE_PHYSICAL 0x23
```

HID 描述符，提供 HID 设备的相关信息。HID 描述符描述符如下表：

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	bcdHID	2	数字	版本号（BCD 码）

4	bCountryCode	1	数字	国家语言代码
5	bNumDescriptors	1	数字	描述符个数
6	bType	1	索引	下一个描述符类型
7	wLength	2	位图	下一个描述符长度

表 1. HID 描述符号

C 语言 HID 描述符结构体为：

```
typedef struct
{
    //HID 描述符长度
    unsigned char bLength;
    // USB_HID_DTYPE_HID (0x21).
    unsigned char bDescriptorType;

    //HID 协议版本号
    unsigned short bcdHID;
    //国家语言
    unsigned char bCountryCode;
    //描述符个数，至少为 1
    unsigned char bNumDescriptors;
    //下一个描述符类型
    unsigned char bType;
    //下一个描述符长度
    unsigned short wLength;
}
```

tHIDDescriptor;

例如：定义一个实际的 HID 设备描述符。

```
static const tHIDDescriptor g_sKeybHIDDescriptor =
{
    9,                                // bLength
    USB_HID_DTYPE_HID,               // bDescriptorType
    0x111,                            // bcdHID (version 1.11 compliant)
    USB_HID_COUNTRY_US,              // bCountryCode (not localized)
    1,                                // bNumDescriptors
    USB_HID_DTYPE_REPORT,            // Report descriptor
    sizeof(Report)                    // Size of report descriptor
};
```

国家语言定义：

```
#define USB_HID_COUNTRY_NONE        0x00
#define USB_HID_COUNTRY_ARABIC      0x01
#define USB_HID_COUNTRY_BELGIAN     0x02
#define USB_HID_COUNTRY_CANADA_BI   0x03
#define USB_HID_COUNTRY_CANADA_FR   0x04
#define USB_HID_COUNTRY_CZECH_REPUBLIC 0x05
#define USB_HID_COUNTRY_DANISH      0x06
```

```

#define USB_HID_COUNTRY_FINNISH          0x07
#define USB_HID_COUNTRY_FRENCH           0x08
#define USB_HID_COUNTRY_GERMAN           0x09
#define USB_HID_COUNTRY_GREEK            0x0A
#define USB_HID_COUNTRY_HEBREW            0x0B
#define USB_HID_COUNTRY_HUNGARY           0x0C
#define USB_HID_COUNTRY_INTERNATIONAL_ISO 0x0D
#define USB_HID_COUNTRY_ITALIAN           0x0E
#define USB_HID_COUNTRY_JAPAN_KATAKANA    0x0F
#define USB_HID_COUNTRY_KOREAN            0x10
#define USB_HID_COUNTRY_LATIN_AMERICAN    0x11
#define USB_HID_COUNTRY_NETHERLANDS       0x12
#define USB_HID_COUNTRY_NORWEGIAN         0x13
#define USB_HID_COUNTRY_PERSIAN            0x14
#define USB_HID_COUNTRY_POLAND             0x15
#define USB_HID_COUNTRY_PORTUGUESE        0x16
#define USB_HID_COUNTRY_RUSSIA            0x17
#define USB_HID_COUNTRY_SLOVAKIA          0x18
#define USB_HID_COUNTRY_SPANISH            0x19
#define USB_HID_COUNTRY_SWEDISH            0x1A
#define USB_HID_COUNTRY_SWISS_FRENCH       0x1B
#define USB_HID_COUNTRY_SWISS_GERMAN       0x1C
#define USB_HID_COUNTRY_SWITZERLAND        0x1D
#define USB_HID_COUNTRY_TAIWAN             0x1E
#define USB_HID_COUNTRY_TURKISH_Q          0x1F
#define USB_HID_COUNTRY_UK                 0x20
#define USB_HID_COUNTRY_US                 0x21
#define USB_HID_COUNTRY_YUGOSLAVIA         0x22
#define USB_HID_COUNTRY_TURKISH_F          0x23

```

在中国使用较多的是美国语言 USB_HID_COUNTRY_US。

USB HID 设备是通过报告来给传送数据的，报告有输入报告和输出报告。输入报告 (Input) 是 USB 设备发送给主机的，例如 USB 鼠标将鼠标移动和鼠标点击等信息返回给电脑，键盘将按键数据返回给电脑等；输出报告 (Output) 是主机发送给 USB 设备的，例如键盘上的数字键盘锁定灯和大写字母锁定灯等。报告是一个数据包，里面包含的是所要传送的数据。输入报告是通过中断输入端点输入的，而输出报告有点区别，当没有中断输出端点时，可以通过控制输出端点 0 发送，当有中断输出端点时，通过中断输出端点发出。

而报告描述符，是描述一个报告以及报告里面的数据是用来干什么用的。通过它，USB HOST 可以分析出报告里面的数据所表示的意思。它通过控制输入端点 0 返回，主机使用获取报告描述符命令来获取报告描述符，注意这个请求是发送到接口的，而不是到设备。一个报告描述符可以描述多个报告，不同的报告通过报告 ID 来识别，报告 ID 在报告最前面，即第一个字节。当报告描述符中没有规定报告 ID 时，报告中就没有 ID 字段，开始就是数据。更详细的说明请参看 USB HID 协议。

下面以一个实例介绍鼠标报告描述符：


```

static const unsigned char g_pucMouseReportDescriptor[]=
{
    UsagePage(USB_HID_GENERIC_DESKTOP),          //通用桌面
    Usage(USB_HID_MOUSE),                          //HID 鼠标
    Collection(USB_HID_APPLICATION),              //应用集合，以 EndCollection 结束
        Usage(USB_HID_POINTER),                  //指针设备
        Collection(USB_HID_PHYSICAL),            //集合
            UsagePage(USB_HID_BUTTONS),          //按键
            UsageMinimum(1),                     //最小值
            UsageMaximum(3),                     //最大值
            LogicalMinimum(0),                   //逻辑最小值
            LogicalMaximum(1),                   //逻辑最大值
            ReportSize(1),                       //Report 大小为 1bit
            ReportCount(3),                      //Report 3 个位
            Input(USB_HID_INPUT_DATA | USB_HID_INPUT_VARIABLE |
                USB_HID_INPUT_ABS),              //发送给主机的报告格式
            //剩余 5 位填满
            ReportSize(5),
            ReportCount(1),
            Input(USB_HID_INPUT_CONSTANT | USB_HID_INPUT_ARRAY |
                USB_HID_INPUT_ABS),
            UsagePage(USB_HID_GENERIC_DESKTOP),  //通用桌面
            Usage(USB_HID_X),
            Usage(USB_HID_Y),
            LogicalMinimum(-127),
            LogicalMaximum(127),
            ReportSize(8),
            ReportCount(2),
            Input(USB_HID_INPUT_DATA | USB_HID_INPUT_VARIABLE |
                USB_HID_INPUT_RELATIVE),
            ReportSize(8),
            ReportCount(MOUSE_REPORT_SIZE - 3),
            Input(USB_HID_INPUT_CONSTANT | USB_HID_INPUT_ARRAY |
                USB_HID_INPUT_ABS),
        EndCollection,
    EndCollection,
};

```

UsagePage 常用参数:

```

#define USB_HID_GENERIC_DESKTOP 0x01
#define USB_HID_BUTTONS          0x09
#define USB_HID_USAGE_POINTER    0x0109
#define USB_HID_USAGE_BUTTONS    0x0509
#define USB_HID_USAGE_LEDS       0x0508
#define USB_HID_USAGE_KEYCODES   0x0507

```

```
#define USB_HID_X          0x30
#define USB_HID_Y          0x31
```

Usage 常用参数:

```
#define USB_HID_X          0x30
#define USB_HID_Y          0x31
#define USB_HID_POINTER    0x01
#define USB_HID_MOUSE      0x02
#define USB_HID_KEYBOARD   0x06
```

Collection 常用参数:

```
#define USB_HID_APPLICATION 0x00
#define USB_HID_PHYSICAL    0x01
```

Input 常用参数:

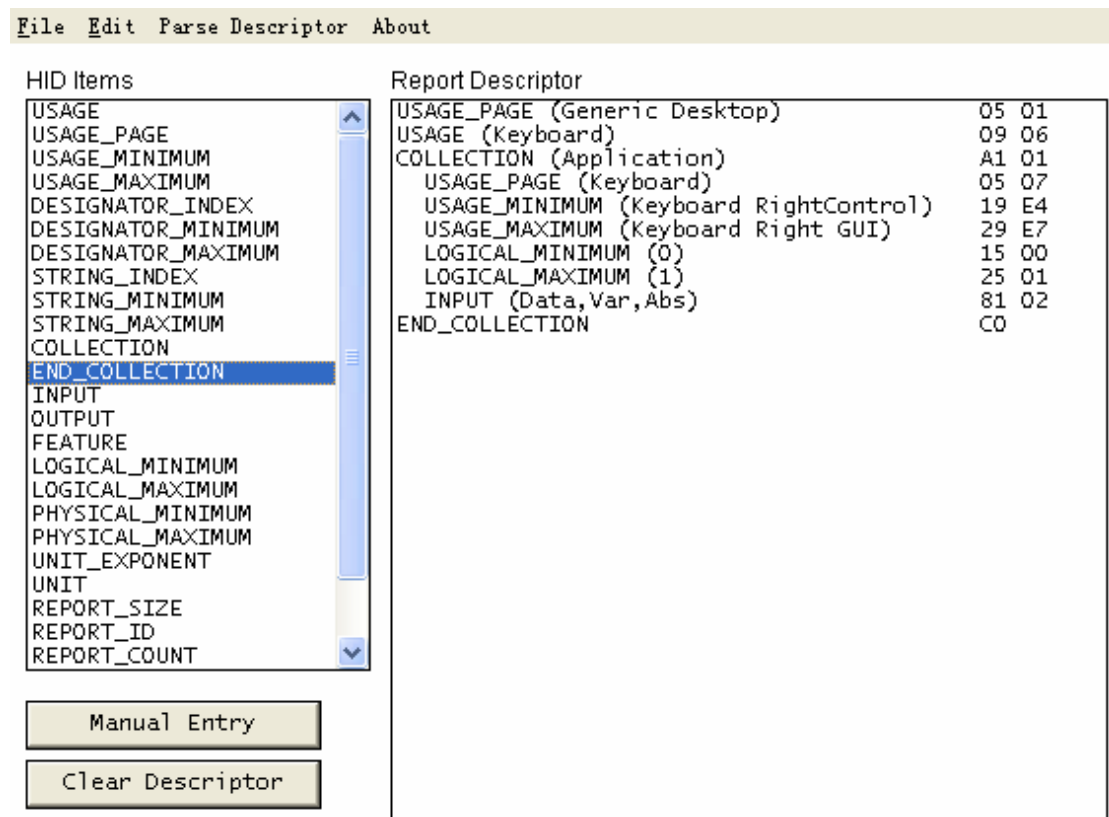
```
#define USB_HID_INPUT_DATA    0x0000
#define USB_HID_INPUT_CONSTANT 0x0001
#define USB_HID_INPUT_ARRAY    0x0000
#define USB_HID_INPUT_VARIABLE 0x0002
#define USB_HID_INPUT_ABS      0x0000
#define USB_HID_INPUT_RELATIVE 0x0004
#define USB_HID_INPUT_NOWRAP   0x0000
#define USB_HID_INPUT_WRAP     0x0008
#define USB_HID_INPUT_LINEAR   0x0000
#define USB_HID_INPUT_NONLINEAR 0x0010
#define USB_HID_INPUT_PREFER    0x0000
#define USB_HID_INPUT_NONPREFER 0x0020
#define USB_HID_INPUT_NONULL    0x0000
#define USB_HID_INPUT_NULL      0x0040
#define USB_HID_INPUT_BITF      0x0100
#define USB_HID_INPUT_BYTES     0x0000
```

Output 常用参数:

```
#define USB_HID_OUTPUT_DATA    0x0000
#define USB_HID_OUTPUT_CONSTANT 0x0001
#define USB_HID_OUTPUT_ARRAY    0x0000
#define USB_HID_OUTPUT_VARIABLE 0x0002
#define USB_HID_OUTPUT_ABS      0x0000
#define USB_HID_OUTPUT_RELATIVE 0x0004
#define USB_HID_OUTPUT_NOWRAP   0x0000
#define USB_HID_OUTPUT_WRAP     0x0008
#define USB_HID_OUTPUT_LINEAR   0x0000
#define USB_HID_OUTPUT_NONLINEAR 0x0010
#define USB_HID_OUTPUT_PREFER    0x0000
#define USB_HID_OUTPUT_NONPREFER 0x0020
#define USB_HID_OUTPUT_NONULL    0x0000
#define USB_HID_OUTPUT_NULL      0x0040
#define USB_HID_OUTPUT_BITF      0x0100
```

```
#define USB_HID_OUTPUT_BYTES    0x0000
```

报告描述符使用复杂，可从 [Http://www.usb.org](http://www.usb.org) 下载 HID Descriptor tool（如图 1 所示）来生成。HID Descriptor tool 是由 USB 官方编写的专用报告符生成工具。有关报告符具体内容请参阅 [Http://www.usb.org](http://www.usb.org) 网上的 HID Usage Tables 文档。使用 USB 库函数开发 USBHID 设备可以不用考虑报告符，在库中已经定义。



如图 1

HID Descriptor tool 界面简洁，操作方便，集结了所有 HID 设备报告定义，如键盘、鼠标、操作杆、手写设备等，对于报告符不熟悉的开发者相当方便。

5.3 USB 键盘

在 USB 库中已经定义好 USB 键盘的数据类型、API，开发 USB 键盘非常快捷方便。相关定义和数据类型放在“usbhidkeyb.h”中。

5.3.1 数据类型

usbhidkeyb.h 中已经定义好 USB 键盘使用的所有数据类型和函数，下面介绍 USB 键盘使用的数据类型。

```
#define KEYB_MAX_CHARS_PER_REPORT    6
```

KEYB_MAX_CHARS_PER_REPORT 定义 USB 键盘一次发送 6 个数据给主机，如果每次发送的数据多于定义的 6 个，将会通过 USBHIDKeyboardKeyStateChange() 函数将返回发送数据太多的错误：KEYB_ERR_TOO_MANY_KEYS。KEYB_MAX_CHARS_PER_REPORT 这个值的定义由键盘报告决定。

```
typedef enum
{
    //状态还没有定义
    HID_KEYBOARD_STATE_UNCONFIGURED,
```

```

        //空闲状态，没有按键按下或者没有等待数据。
        HID_KEYBOARD_STATE_IDLE,
        //等待主机发送数据
        HID_KEYBOARD_STATE_WAIT_DATA,
        //等待数据发送
        HID_KEYBOARD_STATE_SEND
    }
    tKeyboardState;
    tKeyboardState，定义键盘状态，用于在 USB 键盘工作时，保存键盘状态。
#define KEYB_IN_REPORT_SIZE 8
#define KEYB_OUT_REPORT_SIZE 1
KEYB_IN_REPORT_SIZE、KEYB_OUT_REPORT_SIZE 定义键盘 IN 报告符、OUT 报告符长度。
typedef struct
{
    // 指示当前 USB 设备是否配置成功。
    unsigned char ucUSBConfigured;
    // USB 键盘使用的子协议：USB_HID_PROTOCOL_BOOT 或者 USB_HID_PROTOCOL_REPORT
    // 将会传给设备描述符和端点描述符
    unsigned char ucProtocol;
    // 键盘 LED 灯当前状态
    volatile unsigned char ucLEDStates;
    // 记录有几个键按下
    unsigned char ucKeyCount;
    // 中断 IN 端点状态.
    volatile tKeyboardState eKeyboardState;
    // 指示当前是否有键状态改变
    volatile tBoolean bChangeMade;
    // 用于接收 OUT 报告
    unsigned char pucDataBuffer[KEYB_OUT_REPORT_SIZE];
    // 用于收送 IN 报告，保存最新一次报告
    unsigned char pucReport[KEYB_IN_REPORT_SIZE];
    // 按下键的 usage 代码
    unsigned char pucKeysPressed[KEYB_MAX_CHARS_PER_REPORT];
    // 为 IN 报告定义时间。
    tHIDReportIdle sReportIdle;
    // HID 设备实例，保存键盘 HID 信息。
    tHIDInstance sHIDInstance;
    // HID 设备驱动
    tUSBDHIDDevice sHIDDevice;
}
tHIDKeyboardInstance;
tHIDKeyboardInstance，键盘实例。在 tHIDInstance 和 tUSBDHIDDevice 的基础上，用于保存全部 USB 键盘的配置信息，包括描述符、callback 函数、按键事件等。
typedef struct

```

```

{
    //VID
    unsigned short usVID;
    //PID
    unsigned short usPID;
    //设备最大耗电量
    unsigned short usMaxPowermA;
    //电源属性
    unsigned char ucPwrAttributes;
    //函数指针，处理返回事务
    tUSBCallback pfnCallback;
    //Callback 第一个入口参数
    void *pvCBData;
    //指向字符串描述符集合
    const unsigned char * const *ppStringDescriptors;
    //字符串描述符个数 (1 + (5 * (num languages)))
    unsigned long ulNumStringDescriptors;
    //键盘实例，保存 USB 键盘的相关信息。
    tHIDKeyboardInstance *psPrivateHIDKbdData;
}

tUSBHIDKeyboardDevice;

```

tUSBHIDKeyboardDevice, USB 键盘类。定义了 VID、PID、电源属性、字符串描述符等，还包括了一个 USB 键盘实例。其它 HID 设备描述符、配置信息通过 API 函数储入 tHIDKeyboardInstance 定义的 USB 键盘实例中。

5.3.2 API 函数

在 USB 键盘 API 库中定义了 7 个函数，完成 USB 键盘初始化、配置及数据处理。下面为 usbdhidkeyb.h 中定义的 API 函数：

```

void *USBHIDKeyboardInit(unsigned long ulIndex, const tUSBHIDKeyboardDevice *psDevice);
void *USBHIDKeyboardCompositeInit(unsigned long ulIndex,
                                   const tUSBHIDKeyboardDevice *psDevice);
unsigned long USBHIDKeyboardKeyStateChange(void *pvInstance, unsigned char ucModifiers,
                                             unsigned char ucUsageCode, tBoolean bPressed);
void USBHIDKeyboardTerm(void *pvInstance);
void *USBHIDKeyboardSetCBData(void *pvInstance, void *pvCBData);
void USBHIDKeyboardPowerStatusSet(void *pvInstance, unsigned char ucPower);
tBoolean USBHIDKeyboardRemoteWakeupRequest(void *pvInstance);
void *USBHIDKeyboardInit(unsigned long ulIndex,
                          const tUSBHIDKeyboardDevice *psDevice);

```

作用：初始化键盘硬件、协议，把其它配置参数填入 psDevice 的键盘实例中。

参数：ulIndex, USB 模块代码，固定值：USB_BASE0。psDevice, USB 键盘类。

返回：指向配置后的 tUSBHIDKeyboardDevice。

```

void *USBHIDKeyboardCompositeInit(unsigned long ulIndex,
                                   const tUSBHIDKeyboardDevice *psDevice);

```

作用：初始化键盘协议，本函数在 USBHIDKeyboardInit 中已经调用。

参数: ulIndex, USB 模块代码, 固定值: USB_BASE0。psDevice, USB 键盘类。

返回: 指向配置后的 tUSBHIDKeyboardDevice。

```
unsigned long USBHIDKeyboardKeyStateChange(void *pvInstance,
                                           unsigned char ucModifiers,
                                           unsigned char ucUsageCode,
                                           tBoolean bPressed);
```

作用: 键盘状态改变, 并发送报告给主机。

参数: pvInstance, 指向 tUSBHIDKeyboardDevice, 本函数将修改其按键状态等。
ucModifiers, 功能按键代码。ucUsageCode, 普通按键代码。bPressed, 是否加入到报告中并发送给主机。

返回: 程序错误代码。

```
void USBHIDKeyboardTerm(void *pvInstance);
```

作用: 结束 usb 键盘。

参数: pvInstance, 指向 tUSBHIDKeyboardDevice。

返回: 无。

```
void *USBHIDKeyboardSetCBData(void *pvInstance, void *pvCBData);
```

作用: 修改 tUSBHIDKeyboardDevice 中的 pvCBData 指针。

参数: pvInstance, 指向 tUSBHIDKeyboardDevice。pvCBData, 数据指针, 用于替换 tUSBHIDKeyboardDevice 中的 pvCBData 指针。

返回: 以前 tUSBHIDKeyboardDevice 的 pvCBData 的指针。

```
void USBHIDKeyboardPowerStatusSet(void *pvInstance, unsigned char ucPower);
```

作用: 设置键盘电源模式。

参数: pvInstance, 指向 tUSBHIDKeyboardDevice。ucPower, 电源工作模式, USB_STATUS_SELF_PWR 或者 USB_STATUS_BUS_PWR。

返回: 无。

```
tBoolean USBHIDKeyboardRemoteWakeupRequest(void *pvInstance);
```

作用: 唤醒请求。

参数: pvInstance, 指向 tUSBHIDKeyboardDevice。

返回: 是否成功唤醒。

这些 API 中使用最多是 USBHIDKeyboardInit 和 USBHIDKeyboardPowerStatusSet 两个函数, 在首次使用 USB 键盘时, 要初始化设备, 使用 USBHIDKeyboardInit 完成 USB 键盘初始化、打开 USB 中断、枚举设备、描述符补全等; USBHIDKeyboardPowerStatusSet 设置按键状态, 并通过报告发送给主机, 这是键盘与主机进行数据通信最主要的接口函数, 使用频率最高。

5.3.3 USB 键盘开发

USB 键盘开发只需要 4 步就能完成。如图 2 所示, 键盘设备配置(主要是字符串描述符)、callback 函数编写、USB 处理器初始化、按键处理。

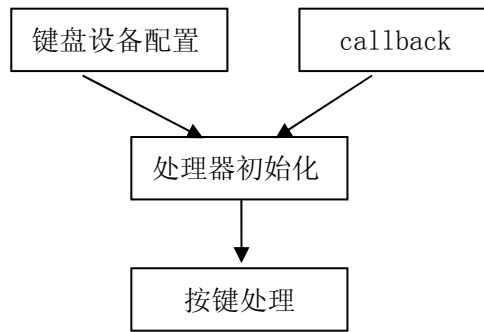


图 2

第一步：键盘设备配置（主要是字符串描述符），按字符串描述符标准完成串描述符配置，进而完成键盘设备配置。

```

#include "inc/hw_ints.h"
#include "inc/hw_memmap.h"
#include "inc/hw_gpio.h"
#include "inc/hw_types.h"
#include "driverlib/debug.h"
#include "driverlib/gpio.h"
#include "driverlib/interrupt.h"
#include "driverlib/sysctl.h"
#include "driverlib/systick.h"
#include "driverlib/usb.h"
#include "usblib/usblib.h"
#include "usblib/usbhid.h"
#include "usblib/usb-ids.h"
#include "usblib/device/usbdevice.h"
#include "usblib/device/usbdhid.h"
#include "usblib/device/usbdhidkeyb.h"
//声明函数原型
unsigned long KeyboardHandler(void *pvCBData,
                             unsigned long ulEvent,
                             unsigned long ulMsgData,
                             void *pvMsgData);

//*****
// 语言描述符
//*****

const unsigned char g_pLangDescriptor[] =
{
    4,
    USB_DTYPE_STRING,
    USBShort(USB_LANG_EN_US)
};

//*****

```

```

// 制造商 字符串 描述符
//*****
const unsigned char g_pManufacturerString[] =
{
    (17 + 1) * 2,
    USB_DTYPE_STRING,
    'T', 0, 'e', 0, 'x', 0, 'a', 0, 's', 0, ' ', 0, 'I', 0, 'n', 0, 's', 0,
    't', 0, 'r', 0, 'u', 0, 'm', 0, 'e', 0, 'n', 0, 't', 0, 's', 0,
};
//*****

//产品 字符串 描述符
//*****
const unsigned char g_pProductString[] =
{
    (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
};
//*****

// 产品 序列号 描述符
//*****
const unsigned char g_pSerialNumberString[] =
{
    (8 + 1) * 2,
    USB_DTYPE_STRING,
    '1', 0, '2', 0, '3', 0, '4', 0, '5', 0, '6', 0, '7', 0, '8', 0
};
//*****

// 设备接口字符串描述符
//*****
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符
//*****
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
};

//*****
// 字符串描述符集合，一定要按这个顺序排列，因为在描述符中已经定义好描述索引。
//*****

const unsigned char * const g_pStringDescriptors[] =
{
    g_pLangDescriptor,
    g_pManufacturerString,
    g_pProductString,
    g_pSerialNumberString,
    g_pHIDInterfaceString,
    g_pConfigString
};

#define NUM_STRING_DESCRIPTOR (sizeof(g_pStringDescriptors) / \
                                sizeof(unsigned char *))

//*****
//键盘实例，键盘配置并为键盘设备信息提供空间
//*****
tHIDKeyboardInstance g_KeyboardInstance;

//*****
//键盘设备配置
//*****

const tUSBHIDKeyboardDevice g_sKeyboardDevice =
{
    USB_VID_STELLARIS,          //自行定义 VID.
    USB_PID_KEYBOARD,          //自行定义 PID.
    500,
    USB_CONF_ATTR_SELF_PWR | USB_CONF_ATTR_RWAKE,
    KeyboardHandler,
    (void *)&g_sKeyboardDevice,
    g_pStringDescriptors,
    NUM_STRING_DESCRIPTOR,
    &g_KeyboardInstance
};

```

第二步：完成 Callback 函数。Callback 函数用于处理按键事务。可能是主机发出，也可能是状态信息。USB 键盘设备中包含了以下事务：USB_EVENT_CONNECTED、USB_EVENT_DISCONNECTED、USB_HID_KEYB_EVENT_SET_LEDS、USB_EVENT_SUSPEND、USB_EVENT_RESUME、USB_EVENT_TX_COMPLETE。如下表：

名称	说明
USB_EVENT_CONNECTED	USB 设备已经连接到主机
USB_EVENT_DISCONNECTED	USB 设备已经与主机断开
USBD_HID_KEYB_EVENT_SET_LEDS	USB 键盘有 LED 灯设置，查询功能按键再确定 LED
USB_EVENT_SUSPEND	挂起
USB_EVENT_RESUME	唤醒
USB_EVENT_TX_COMPLETE	发送完成

表 2. USB 键盘事务

根据以上事务编写 Callback 函数：

```

unsigned long KeyboardHandler(void *pvCBData, unsigned long ulEvent,
                             unsigned long ulMsgData, void *pvMsgData)
{
    switch (ulEvent)
    {
        case USB_EVENT_CONNECTED:
        {
            //连接成功时，点亮 LED1
            GPIOPinWrite(GPIO_PORTF_BASE, 0x10, 0x10);
            break;
        }
        case USB_EVENT_DISCONNECTED:
        {
            //断开连接时，LED1 灭
            GPIOPinWrite(GPIO_PORTF_BASE, 0x10, 0x00);
            break;
        }
        case USB_EVENT_TX_COMPLETE:
        {
            //发送完成时，LED2 亮
            GPIOPinWrite(GPIO_PORTF_BASE, 0x20, 0x20);
            break;
        }
        case USB_EVENT_SUSPEND:
        {
            //发送完成时，LED2 亮
            GPIOPinWrite(GPIO_PORTF_BASE, 0x20, 0x0);
            break;
        }
        case USB_EVENT_RESUME:
        {
            break;
        }
        case USBD_HID_KEYB_EVENT_SET_LEDS:
        {

```

```

        //HID_KEYB_CAPS_LOCK 灯
        GPIOPinWrite(GPIO_PORTF_BASE, 0x80, ((char)u1MsgData &
            HID_KEYB_CAPS_LOCK)?0 : 0x80);

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

```

第三步：系统初始化，配置内核电压、系统主频、使能端口、配置按键端口、LED 控制等，本例中使用 4 个按键模拟普通键盘，使用 4 个 LED 进行指示。原理图如图 3 所示：

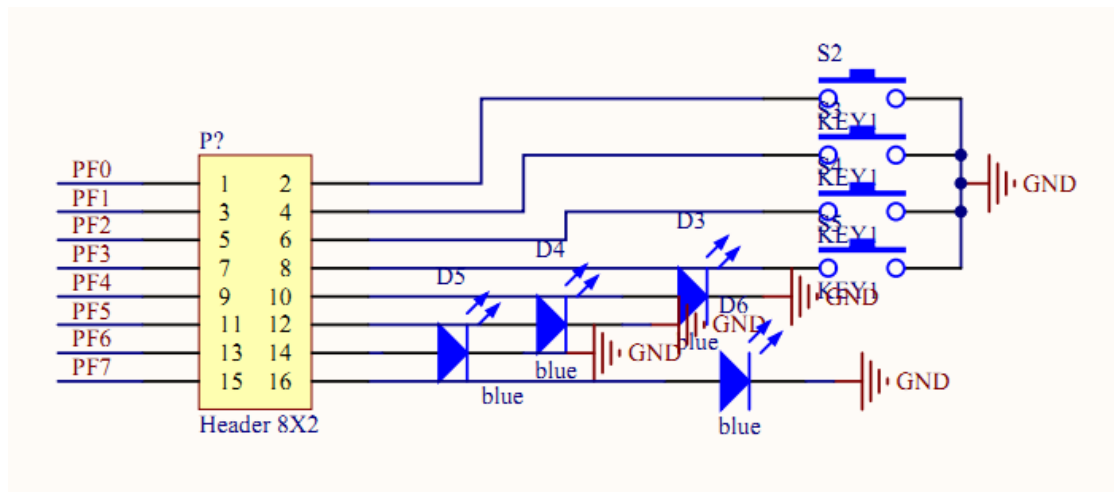


图 3

系统初始化：

```

//设置系统内核电压 与 主频
SysCtlLDOSet(SYSCTL_LD0_2_75V);
SysCtlClockSet(SYSCTL_XTAL_8MHZ | SYSCTL_SYSDIV_4 | SYSCTL_USE_PLL | SYSCTL_OSC_MAIN);
//使能端口
SysCtlPeripheralEnable(SYSCTL_PERIPH_GPIOF);
GPIOPinTypeGPIOOutput(GPIO_PORTF_BASE, 0xf0);
GPIOPinTypeGPIOInput(GPIO_PORTF_BASE, 0x0f);
HWREG(GPIO_PORTF_BASE+GPIO_O_PUR) |= 0x0f;
//初始化键盘设备
USBDHIDKeyboardInit(0, &g_sKeyboardDevice);

```

第四步：按键处理。主要使用 USBDHIDKeyboardPowerStatusSet 设置按键状态，并通过报告发送给主机。

```

while(1)
{
    USBDHIDKeyboardKeyStateChange((void *)&g_sKeyboardDevice, HID_KEYB_CAPS_LOCK,

```

```

        HID_KEYB_USAGE_A,
        (GPIOPinRead(GPIO_PORTF_BASE, 0x0f) & GPIO_PIN_0)
        ? false : true);
    USBDHIDKeyboardKeyStateChange((void *)&g_sKeyboardDevice, 0,
        HID_KEYB_USAGE_DOWN_ARROW,
        (GPIOPinRead(GPIO_PORTF_BASE, 0x0f) & GPIO_PIN_1)
        ? false : true);
    USBDHIDKeyboardKeyStateChange((void *)&g_sKeyboardDevice, 0,
        HID_KEYB_USAGE_UP_ARROW,
        (GPIOPinRead(GPIO_PORTF_BASE, 0x0f) & GPIO_PIN_2)
        ? false : true);
    USBDHIDKeyboardKeyStateChange((void *)&g_sKeyboardDevice, 0,
        HID_KEYB_USAGE_ESCAPE,
        (GPIOPinRead(GPIO_PORTF_BASE, 0x0f) & GPIO_PIN_3)
        ? false : true);
    SysCtlDelay(SysCtlClockGet()/3000);
}

```

使用上面四步就完成 USB 键盘开发，与普通 USB 键盘没有什么差别。由于在这个例子中使用的是 Demo 开发板，只模拟了四个按键，但功能与普通 USB 键盘一样。USB 键盘开发时要加入两个 lib: usblib.lib 和 DriverLib.lib，在启动代码中加入 USB0DeviceIntHandler 中断服务函数。USB 键盘源码如下：

```

#include "inc/hw_ints.h"
#include "inc/hw_memmap.h"
#include "inc/hw_gpio.h"
#include "inc/hw_types.h"
#include "driverlib/debug.h"
#include "driverlib/gpio.h"
#include "driverlib/interrupt.h"
#include "driverlib/sysctl.h"
#include "driverlib/systick.h"
#include "driverlib/usb.h"
#include "usblib/usblib.h"
#include "usblib/usbhid.h"
#include "usblib/usb-ids.h"
#include "usblib/device/usbdevice.h"
#include "usblib/device/usbdhid.h"
#include "usblib/device/usbdhidkeyb.h"
//声明函数原型
unsigned long KeyboardHandler(void *pvCBData,
                                unsigned long ulEvent,
                                unsigned long ulMsgData,
                                void *pvMsgData);

//*****
// 语言描述符

```

```

//*****
const unsigned char g_pLangDescriptor[] =
{
    4,
    USB_DTYPE_STRING,
    USBShort(USB_LANG_EN_US)
};
//*****
// 制造商 字符串 描述符
//*****
const unsigned char g_pManufacturerString[] =
{
    (17 + 1) * 2,
    USB_DTYPE_STRING,
    'T', 0, 'e', 0, 'x', 0, 'a', 0, 's', 0, ' ', 0, 'I', 0, 'n', 0, 's', 0,
    't', 0, 'r', 0, 'u', 0, 'm', 0, 'e', 0, 'n', 0, 't', 0, 's', 0,
};
//*****
//产品 字符串 描述符
//*****
const unsigned char g_pProductString[] =
{
    (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
};
//*****
// 产品 序列号 描述符
//*****
const unsigned char g_pSerialNumberString[] =
{
    (8 + 1) * 2,
    USB_DTYPE_STRING,
    '1', 0, '2', 0, '3', 0, '4', 0, '5', 0, '6', 0, '7', 0, '8', 0
};
//*****
// 设备接口字符串描述符
//*****
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符
//*****
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
};

//*****
// 字符串描述符集合
//*****
const unsigned char * const g_pStringDescriptors[] =
{
    g_pLangDescriptor,
    g_pManufacturerString,
    g_pProductString,
    g_pSerialNumberString,
    g_pHIDInterfaceString,
    g_pConfigString
};

#define NUM_STRING_DESCRIPTOR (sizeof(g_pStringDescriptors) / \
                                sizeof(unsigned char *))

//*****
//键盘实例，键盘配置并为键盘设备信息提供空间
//*****
tHIDKeyboardInstance g_KeyboardInstance;

//*****
//键盘设备配置
//*****
const tUSBHIDKeyboardDevice g_sKeyboardDevice =
{
    USB_VID_STELLARIS,
    USB_PID_KEYBOARD,
    500,
    USB_CONF_ATTR_SELF_PWR | USB_CONF_ATTR_RWAKE,
    KeyboardHandler,

```

```

        (void *)&g_sKeyboardDevice,
        g_pStringDescriptors,
        NUM_STRING_DESCRIPTORS,
        &g_KeyboardInstance
    };

    /*******
    //键盘 callback 函数
    /*******
    unsigned long KeyboardHandler(void *pvCBData, unsigned long ulEvent,
                                unsigned long ulMsgData, void *pvMsgData)
    {
        switch (ulEvent)
        {
            case USB_EVENT_CONNECTED:
            {
                //连接成功时, 点亮 LED1
                GPIOPinWrite(GPIO_PORTF_BASE, 0x10, 0x10);

                break;
            }
            case USB_EVENT_DISCONNECTED:
            {
                //断开连接时, LED1 灭
                GPIOPinWrite(GPIO_PORTF_BASE, 0x10, 0x00);

                break;
            }
            case USB_EVENT_TX_COMPLETE:
            {
                //发送完成时, LED2 亮
                GPIOPinWrite(GPIO_PORTF_BASE, 0x20, 0x20);

                break;
            }
            case USB_EVENT_SUSPEND:
            {
                //发送完成时, LED2 亮
                GPIOPinWrite(GPIO_PORTF_BASE, 0x20, 0x0);

                break;
            }
            case USB_EVENT_RESUME:
            {
                break;
            }
            case USBD_HID_KEYB_EVENT_SET_LEDS:
            {
                //HID_KEYB_CAPS_LOCK 灯

```

```

        GPIOPinWrite(GPIO_PORTF_BASE, 0x80, ((char)ulMsgData & HID_KEYB_CAPS_LOCK)?0 :
0x80);

        break;
    }
    default:
    {
        break;
    }
}

return (0);
}

//*****
//主函数
//*****

int main(void)
{
    //系统初始化。
    SysCtlLDOSet(SYSCTL_LD0_2_75V);
    SysCtlClockSet(SYSCTL_XTAL_8MHZ | SYSCTL_SYSDIV_4 | SYSCTL_USE_PLL | SYSCTL_OSC_MAIN);

    SysCtlPeripheralEnable(SYSCTL_PERIPH_GPIOF);
    GPIOPinTypeGPIOOutput(GPIO_PORTF_BASE, 0xf0);
    GPIOPinTypeGPIOInput(GPIO_PORTF_BASE, 0x0f);
    HWREG(GPIO_PORTF_BASE+GPIO_O_PUR) |= 0x0f;
    USBDHIDKeyboardInit(0, &g_sKeyboardDevice);
    while(1)
    {

        USBDHIDKeyboardKeyStateChange((void *)&g_sKeyboardDevice, HID_KEYB_CAPS_LOCK,
                                        HID_KEYB_USAGE_A,
                                        (GPIOPinRead(GPIO_PORTF_BASE, 0x0f) & GPIO_PIN_0)
                                        ? false : true);

        USBDHIDKeyboardKeyStateChange((void *)&g_sKeyboardDevice, 0,
                                        HID_KEYB_USAGE_DOWN_ARROW,
                                        (GPIOPinRead(GPIO_PORTF_BASE, 0x0f) & GPIO_PIN_1)
                                        ? false : true);

        USBDHIDKeyboardKeyStateChange((void *)&g_sKeyboardDevice, 0,
                                        HID_KEYB_USAGE_UP_ARROW,
                                        (GPIOPinRead(GPIO_PORTF_BASE, 0x0f) & GPIO_PIN_2)
                                        ? false : true);

        USBDHIDKeyboardKeyStateChange((void *)&g_sKeyboardDevice, 0,
                                        HID_KEYB_USAGE_ESCAPE,
                                        (GPIOPinRead(GPIO_PORTF_BASE, 0x0f) & GPIO_PIN_3)
                                        ? false : true);
    }
}

```



```

        SysCtlDelay(SysCtlClockGet()/3000);
    }
}

```

5.4 USB 鼠标

在 USB 库中已经定义好 USB 鼠标的数据类型、API，开发 USB 鼠标非常快捷方便。相关定义和数据类型放在“usbhidmouse.h”中。

5.4.1 数据类型

usbhidmouse.h 中已经定义好 USB 鼠标使用的所有数据类型和函数，下面介绍 USB 鼠标使用的数据类型。

```
#define MOUSE_REPORT_SIZE    3
```

MOUSE_REPORT_SIZE 定义鼠标 IN 报告的大小，可用于判断 IN 报告是否正确，如果超出 3 个字节会返回错误代码。

```
typedef enum
{
    //状态还没有定义
    HID_MOUSE_STATE_UNCONFIGURED,
    //空闲状态，没有按键按下或者没有等待数据。
    HID_MOUSE_STATE_IDLE,
    //等待主机发送数据
    HID_MOUSE_STATE_WAIT_DATA,
    //等待数据发送
    HID_MOUSE_STATE_SEND
}

```

```
tMouseState;
```

tMouseState，定义 USB 鼠标状态，USB 鼠标正常工作时，保存鼠标工作状态。

```
typedef struct
{
    // 指示当前 USB 设备是否配置成功。
    unsigned char ucUSBConfigured;
    // USB 键盘使用的子协议：USB_HID_PROTOCOL_BOOT 或者 USB_HID_PROTOCOL_REPORT
    // 将会传给设备描述符和端点描述符
    unsigned char ucProtocol;
    // 用于收送 IN 报告，保存最新一次报告
    unsigned char pucReport[MOUSE_REPORT_SIZE];
    // 中断 IN 端点状态。
    volatile tMouseState eMouseState;
    // 为 IN 报告定义时间。
    tHIDReportIdle sReportIdle;
    // HID 设备实例，保存键盘 HID 信息。
    tHIDInstance sHIDInstance;
    // HID 设备驱动
    tUSBDHIDDevice sHIDDevice;
}

```

```
tHIDMouseInstance;
```

tHIDMouseInstance, USB 鼠标实例。在 tHIDInstance 和 tUSBHIDDevice 的基础上, 用于保存全部 USB 鼠标的配置信息, 包括描述符、callback 函数、按键事件等。

```
typedef struct
{
    //VID
    unsigned short usVID;
    //PID
    unsigned short usPID;
    //设备最大耗电量
    unsigned short usMaxPowermA;
    //电源属性
    unsigned char ucPwrAttributes;
    //函数指针, 处理返回事务
    tUSBCallback pfnCallback;
    //Callback 第一个入口参数
    void *pvCBData;
    //指向字符串描述符集合
    const unsigned char * const *ppStringDescriptors;
    //字符串描述符个数 (1 + (5 * (num languages)))
    unsigned long ulNumStringDescriptors;
    //鼠标实例, 保存 USB 鼠标的相关信息。
    tHIDMouseInstance *psPrivateHIDMouseData;
}
tUSBHIDMouseDevice;
```

tUSBHIDMouseDevice, USB 鼠标类。定义了 VID、PID、电源属性、字符串描述符等, 还包括了一个 USB 鼠标实例。其它 HID 设备描述符、配置信息通过 API 函数储入 tHIDMouseInstance 定义的 USB 鼠标实例中。

```
#define MOUSE_ERR_TX_ERROR    2
```

MOUSE_ERR_TX_ERROR, USB 鼠标 API 函数 USBHIDMouseStateChange 返回的错误代码。

5.4.2 API 函数

在 USB 鼠标 API 库中定义了 7 个函数, 完成 USB 键盘初始化、配置及数据处理。下面为 usbdhidkeyb.h 中定义的 API 函数:

```
void *USBHIDMouseInit(unsigned long ulIndex,
                      const tUSBHIDMouseDevice *psDevice);

void *USBHIDMouseCompositeInit(unsigned long ulIndex,
                              const tUSBHIDMouseDevice *psDevice);

unsigned long USBHIDMouseStateChange(void *pvInstance, char cDeltaX,
                                     char cDeltaY,
                                     unsigned char ucButtons);

void USBHIDMouseTerm(void *pvInstance);

void *USBHIDMouseSetCBData(void *pvInstance, void *pvCBData);

void USBHIDMousePowerStatusSet(void *pvInstance,
```

```

                                unsigned char ucPower);
tBoolean USBDHIDMouseRemoteWakeupRequest(void *pvInstance);

void *USBBDHIDMouseInit(unsigned long ulIndex,
                        const tUSBBDHIDMouseDevice *psDevice);
作用：初始化鼠标硬件、协议，把其它配置参数填入 psDevice 的鼠标实例中。
参数：ulIndex, USB 模块代码，固定值：USB_BASE0。psDevice, USB 鼠标类。
返回：指向配置后的 tUSBBDHIDMouseDevice。
void *USBBDHIDMouseCompositeInit(unsigned long ulIndex,
                                const tUSBBDHIDMouseDevice *psDevice);
作用：初始化鼠标协议，本函数在 USBBDHIDMouseInit 中已经调用。
参数：ulIndex, USB 模块代码，固定值：USB_BASE0。psDevice, USB 鼠标类。
返回：指向配置后的 tUSBBDHIDMouseDevice。
unsigned long USBBDHIDMouseStateChange(void *pvInstance, char cDeltaX,
                                       char cDeltaY,
                                       unsigned char ucButtons);

作用：鼠标状态改变，并发送报告给主机。
参数：pvInstance, 指向 tUSBBDHIDMouseDevice, 本函数将修改其 X、Y、按键状态等。
cDeltaX, X 值。cDeltaY, Y 值。ucButtons, 鼠标按键。
返回：程序错误代码。
void USBBDHIDMouseTerm(void *pvInstance);
作用：结束 usb 鼠标。
参数：pvInstance, 指向 tUSBBDHIDMouseDevice。
返回：无。
void *USBBDHIDMouseSetCBData(void *pvInstance, void *pvCBData);
作用：修改 tUSBBDHIDMouseDevice 中的 pvCBData 指针。
参数：pvInstance, 指向 tUSBBDHIDMouseDevice。pvCBData, 数据指针，用于替换
tUSBBDHIDMouseDevice 中的 pvCBData 指针。
返回：以前 tUSBBDHIDMouseDevice 的 pvCBData 的指针。
void USBBDHIDMousePowerStatusSet(void *pvInstance,
                                unsigned char ucPower);

作用：设置鼠标电源模式。
参数：pvInstance, 指向 tUSBBDHIDMouseDevice。ucPower, 电源工作模式，
USB_STATUS_SELF_PWR 或者 USB_STATUS_BUS_PWR。
返回：无。
tBoolean USBBDHIDMouseRemoteWakeupRequest(void *pvInstance);
作用：唤醒请求。
参数：pvInstance, 指向 tUSBBDHIDMouseDevice。
返回：是否成功唤醒。

```

这些 API 中使用最多是 USBBDHIDMouseInit 和 USBBDHIDMouseStateChange 两个函数，在首次使用 USB 鼠标时，要初始化 USB 设备，使用 USBBDHIDMouseInit 完成 USB 鼠标初始化、打开 USB 中断、枚举设备、描述符补全等；USBBDHIDMouseStateChange 设置鼠标 X、Y、按键状态，并通过 IN 报告发送给主机，这是 USB 鼠标与主机进行数据通信最主要的接口函数，使用频率最高。

5.4.3 USB 鼠标开发

USB 鼠标开发只需要 4 步就能完成。如图 2 所示, 鼠标设备配置(主要是字符串描述符)、callback 函数编写、USB 处理器初始化、X\Y\按键处理。

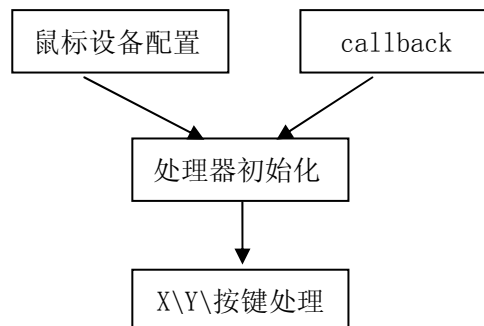


图 2

第一步: 鼠标设备配置 (主要是字符串描述符), 按字符串描述符标准完成串描述符配置, 进而完成鼠标设备配置。

```
#include "inc/hw_types.h"
#include "driverlib/usb.h"
#include "usblib/usb.h"
#include "usblib/usbhid.h"
#include "usblib/usb-ids.h"
#include "usblib/device/usbdevice.h"
#include "usblib/device/usbdhid.h"
#include "usblib/device/usbdhidmouse.h"
#include "usb_mouse_structs.h"
//声明函数原型
unsigned long MouseHandler(void *pvCBData,
                           unsigned long ulEvent,
                           unsigned long ulMsgData,
                           void *pvMsgData);

//*****
// 语言描述符
//*****
const unsigned char g_pLangDescriptor[] =
{
    4,
    USB_DTYPE_STRING,
    USBShort(USB_LANG_EN_US)
};

//*****
// 制造商 字符串 描述符
//*****
const unsigned char g_pManufacturerString[] =
```

```

{
    (17 + 1) * 2,
    USB_DTYPE_STRING,
    'T', 0, 'e', 0, 'x', 0, 'a', 0, 's', 0, ' ', 0, 'I', 0, 'n', 0, 's', 0,
    't', 0, 'r', 0, 'u', 0, 'm', 0, 'e', 0, 'n', 0, 't', 0, 's', 0,
};
//*****
//产品 字符串 描述符
//*****
const unsigned char g_pProductString[] =
{
    (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
};
//*****
// 产品 序列号 描述符
//*****
const unsigned char g_pSerialNumberString[] =
{
    (8 + 1) * 2,
    USB_DTYPE_STRING,
    '1', 0, '2', 0, '3', 0, '4', 0, '5', 0, '6', 0, '7', 0, '8', 0
};
//*****
// 设备接口字符串描述符
//*****
const unsigned char g_pHIDInterfaceString[] =
{
    (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
};
//*****
// 设备配置字符串描述符
//*****
const unsigned char g_pConfigString[] =
{
    (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
    };

    //*****
    // 字符串描述符集合
    //*****
    const unsigned char * const g_pStringDescriptors[] =
    {
        g_pLangDescriptor,
        g_pManufacturerString,
        g_pProductString,
        g_pSerialNumberString,
        g_pHIDInterfaceString,
        g_pConfigString
    };

#define NUM_STRING_DESCRIPTOR (sizeof(g_pStringDescriptors) / \
                                sizeof(unsigned char *))

    //*****
    //鼠标实例，鼠标配置并为鼠标设备信息提供空间
    //*****
    tHIDMouseInstance g_sMouseInstance;

    //*****
    //鼠标设备配置
    //*****

    const tUSBHIDMouseDevice g_sMouseDevice =
    {
        USB_VID_STELLARIS,    //开发者自己定义
        USB_PID_MOUSE,        //开发者自己定义
        500,
        USB_CONF_ATTR_SELF_PWR,
        MouseHandler,
        (void *)&g_sMouseDevice,
        g_pStringDescriptors,
        NUM_STRING_DESCRIPTOR,
        &g_sMouseInstance
    };

```

第二步：完成 Callback 函数。Callback 函数用于处理 X、Y、按键事务。可能是主机发出，也可能是状态信息。USB 鼠标设备中包含了以下事务：USB_EVENT_CONNECTED、USB_EVENT_DISCONNECTED、USB_EVENT_SUSPEND、USB_EVENT_RESUME、USB_EVENT_TX_COMPLETE。如下表：

名称	说明
USB_EVENT_CONNECTED	USB 设备已经连接到主机
USB_EVENT_DISCONNECTED	USB 设备已经与主机断开

USB_EVENT_SUSPEND	挂起
USB_EVENT_RESUME	唤醒
USB_EVENT_TX_COMPLETE	发送完成

表 2. USB 鼠标事务

根据以上事务编写 Callback 函数:

```

unsigned long MouseHandler(void *pvCBData, unsigned long ulEvent,
                           unsigned long ulMsgData, void *pvMsgData)
{
    switch (ulEvent)
    {
        case USB_EVENT_CONNECTED:
        {
            //连接成功时, 点亮 LED1
            GPIOWrite(GPIO_PORTF_BASE, 0x10, 0x10);
            break;
        }
        case USB_EVENT_DISCONNECTED:
        {
            //断开连接时, LED1 灭
            GPIOWrite(GPIO_PORTF_BASE, 0x10, 0x00);
            break;
        }
        case USB_EVENT_TX_COMPLETE:
        {
            //发送完成时, LED2 亮
            GPIOWrite(GPIO_PORTF_BASE, 0x20, 0x20);
            break;
        }
        case USB_EVENT_SUSPEND:
        {
            //发送完成时, LED2 亮
            GPIOWrite(GPIO_PORTF_BASE, 0x20, 0x00);
            break;
        }
        case USB_EVENT_RESUME:
        {
            break;
        }
        default:
        {
            break;
        }
    }
    return (0);
}

```

}

第三步：系统初始化，配置内核电压、系统主频、使能端口、配置按键端口、LED 控制等，本例中使用 4 个按键控制鼠标移动，使用 4 个 LED 进行指示动作。原理图如图 3 所示：

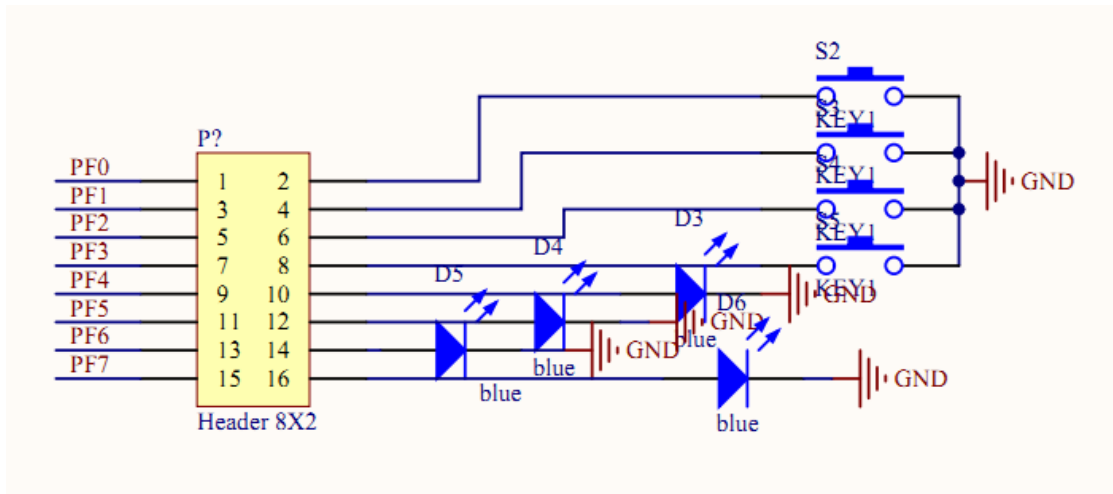


图 3

系统初始化：

```
//设置系统内核电压 与 主频
SysCtlLDOSet(SYSCTL_LD0_2_75V);
SysCtlClockSet(SYSCTL_XTAL_8MHZ | SYSCTL_SYSDIV_4 | SYSCTL_USE_PLL | SYSCTL_OSC_MAIN);
//使能端口
SysCtlPeripheralEnable(SYSCTL_PERIPH_GPIOF);
GPIOPinTypeGPIOOutput(GPIO_PORTF_BASE, 0xf0);
GPIOPinTypeGPIOInput(GPIO_PORTF_BASE, 0x0f);
HWREG(GPIO_PORTF_BASE+GPIO_O_PUR) |= 0x0f;
//初始化鼠标设备
USBDHIDMouseInit(0, &g_sMouseDevice);
```

第四步：X、Y、按键处理。主要使用 USBDHIDMouseStateChange 设置 X、Y、按键状态，并通过报告发送给主机。

```
while(1)
{
    ulTemp = (~GPIOPinRead(GPIO_PORTF_BASE, 0x0f)) & 0x0f;
    switch(ulTemp)
    {
        case 0x01:
            x = x + 1;
            key = 0;
            break;
        case 0x02:
            x = x - 1;
            key = 0;
            break;
        case 0x04:
            y = y + 1;
```



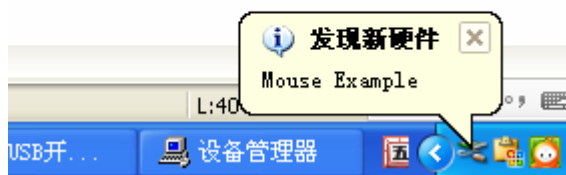
```

        key = 0;
        break;
    case 0x08:
        y = y - 1;
        key = 0;
        break;
    case 0x03:
        key = 1;
        break;
    case 0x0c:
        key = 2;
        break;
    case 0x09:
        key = 4;
        break;
    default:
        key = 0;
        break;
    }

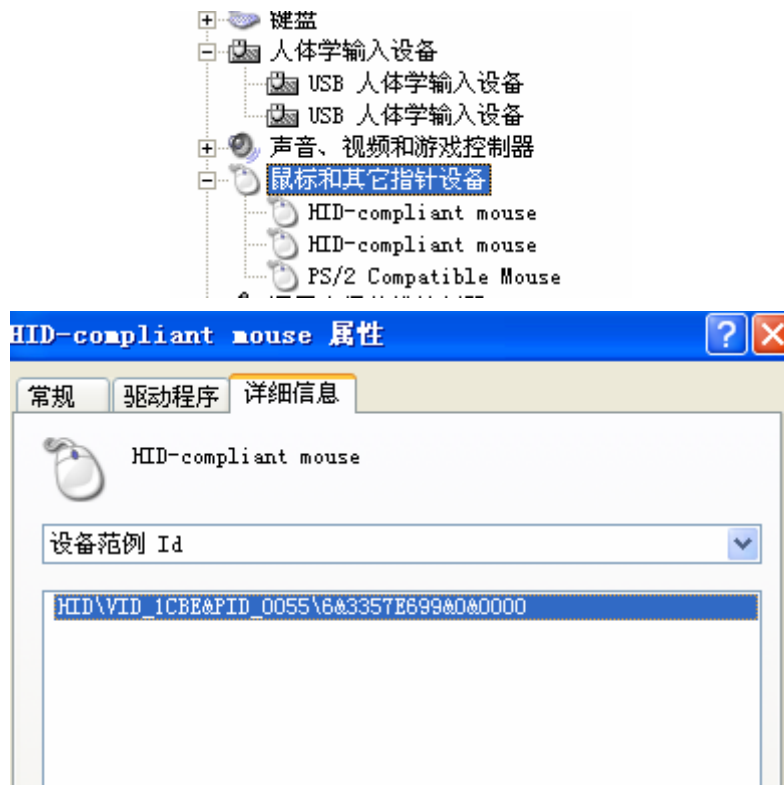
    if (ulTemp)
        USBDHIDMouseStateChange((void *)&g_sMouseDevice, x, y, key);
    SysCtlDelay(SysCtlClockGet() / 30);
}

```

使用上面四步就完成 USB 鼠标开发，与普通 USB 鼠标一样操作。由于在这个例子中使用的是 Demo 开发板，只能用四个按键，模拟鼠标移动。USB 鼠标开发时也要加入两个 lib: usblib.lib 和 DriverLib.lib，在启动代码中加入 USB0DeviceIntHandler 中断服务函数。程序运行进如下图：



从图中可以看出 USB 鼠标枚举成功，在“设备管理器”中可以看到“USB 人体学输入设备”，而且可以在“鼠标和其它指针设备”中找到“HID-compliant mouse”，如下图。其中有一个就是上面开发的 USB 鼠标，查看“属性”可以看到下图：其中 VID 和 PID 都是之前配置的。



USB 鼠标源码如下：

```
#include "inc/hw_ints.h"
#include "inc/hw_memmap.h"
#include "inc/hw_gpio.h"
#include "inc/hw_types.h"
#include "driverlib/debug.h"
#include "driverlib/gpio.h"
#include "inc/hw_types.h"
#include "driverlib/usb.h"
#include "inc/hw_sysctl.h"
#include "driverlib/sysctl.h"
#include "usblib/usblib.h"
#include "usblib/usbhid.h"
#include "usblib/usb-ids.h"
#include "usblib/device/usbdevice.h"
#include "usblib/device/usbdhid.h"
#include "usblib/device/usbdhidmouse.h"
//声明函数原型
unsigned long MouseHandler(void *pvCBData,
                           unsigned long ulEvent,
                           unsigned long ulMsgData,
                           void *pvMsgData);

//*****
// 语言描述符
//*****
```

```

const unsigned char g_pLangDescriptor[] =
{
    4,
    USB_DTYPE_STRING,
    USBShort(USB_LANG_EN_US)
};
//*****
// 制造商 字符串 描述符
//*****
const unsigned char g_pManufacturerString[] =
{
    (17 + 1) * 2,
    USB_DTYPE_STRING,
    'T', 0, 'e', 0, 'x', 0, 'a', 0, 's', 0, ' ', 0, 'I', 0, 'n', 0, 's', 0,
    't', 0, 'r', 0, 'u', 0, 'm', 0, 'e', 0, 'n', 0, 't', 0, 's', 0,
};
//*****
//产品 字符串 描述符
//*****
const unsigned char g_pProductString[] =
{
    (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
};
//*****
// 产品 序列号 描述符
//*****
const unsigned char g_pSerialNumberString[] =
{
    (8 + 1) * 2,
    USB_DTYPE_STRING,
    '1', 0, '2', 0, '3', 0, '4', 0, '5', 0, '6', 0, '7', 0, '8', 0
};
//*****
// 设备接口字符串描述符
//*****
const unsigned char g_pHIDInterfaceString[] =
{
    (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
};

//*****
// 设备配置字符串描述符
//*****
const unsigned char g_pConfigString[] =
{
    (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
};

//*****
// 字符串描述符集合
//*****
const unsigned char * const g_pStringDescriptors[] =
{
    g_pLangDescriptor,
    g_pManufacturerString,
    g_pProductString,
    g_pSerialNumberString,
    g_pHIDInterfaceString,
    g_pConfigString
};

#define NUM_STRING_DESCRIPTOR (sizeof(g_pStringDescriptors) / \
                                sizeof(unsigned char *))

//*****
//键盘实例，键盘配置并为键盘设备信息提供空间
//*****
tHIDMouseInstance g_sMouseInstance;

//*****
//键盘设备配置
//*****
const tUSBHIDMouseDevice g_sMouseDevice =
{
    USB_VID_STELLARIS,
    USB_PID_MOUSE,
    500,
    USB_CONF_ATTR_SELF_PWR,
    MouseHandler,
    (void *)&g_sMouseDevice,
    g_pStringDescriptors,
    NUM_STRING_DESCRIPTOR,

```

```

        &g_sMouseInstance
};

//*****
//键盘 callback 函数
//*****
unsigned long MouseHandler(void *pvCBData, unsigned long ulEvent,
                           unsigned long ulMsgData, void *pvMsgData)
{
    switch (ulEvent)
    {
        case USB_EVENT_CONNECTED:
        {
            //连接成功时, 点亮 LED1
            GPIOPinWrite(GPIO_PORTF_BASE, 0x10, 0x10);
            break;
        }
        case USB_EVENT_DISCONNECTED:
        {
            //断开连接时, LED1 灭
            GPIOPinWrite(GPIO_PORTF_BASE, 0x10, 0x00);
            break;
        }
        case USB_EVENT_TX_COMPLETE:
        {
            //发送完成时, LED2 亮
            GPIOPinWrite(GPIO_PORTF_BASE, 0x20, 0x20);
            break;
        }
        case USB_EVENT_SUSPEND:
        {
            //发送完成时, LED2 亮
            GPIOPinWrite(GPIO_PORTF_BASE, 0x20, 0x0);
            break;
        }
        case USB_EVENT_RESUME:
        {
            break;
        }
        default:
        {
            break;
        }
    }
    return (0);
}

```

```

}
//*****
//主函数
//*****
int main(void)
{
    //系统初始化。
    unsigned long x=0, y=0, key=0, ulTemp=0;
    SysCtlLDOSet(SYSCTL_LDO_2_75V);
    SysCtlClockSet(SYSCTL_XTAL_8MHZ | SYSCTL_SYSDIV_4 | SYSCTL_USE_PLL |
SYSCTL_OSC_MAIN );
    SysCtlPeripheralEnable(SYSCTL_PERIPH_GPIOF);
    GPIOPinTypeGPIOOutput(GPIO_PORTF_BASE, 0xf0);
    GPIOPinTypeGPIOInput(GPIO_PORTF_BASE, 0x0f);
    HWREG(GPIO_PORTF_BASE+GPIO_O_PUR) |= 0x0f;
    USBDHIDMouseInit(0, &g_sMouseDevice);
    while(1)
    {
        ulTemp = (~GPIOPinRead(GPIO_PORTF_BASE, 0x0f)) & 0x0f;
        switch(ulTemp)
        {
            case 0x01:
                x = x + 1;
                key = 0;
                break;
            case 0x02:
                x = x - 1 ;
                key = 0;
                break;
            case 0x04:
                y = y + 1;
                key = 0;
                break;
            case 0x08:
                y = y - 1;
                key = 0;
                break;
            case 0x03:
                key = 1;
                break;
            case 0x0c:
                key = 2;
                break;
            case 0x09:

```

```

        key = 4;
        break;
    default:
        key = 0;
        break;
    }
    if (ulTemp)
        USBDHIDMouseStateChange((void *)&g_sMouseDevice, x, y, key);
    SysCtlDelay(SysCtlClockGet()/30);
}
}

```

5.5 小结

经过本章节介绍，读者对 HID 设备有初步了解，如果想了解更深层次的 HID 设备类，可能参考官方数据手册。本章主要介绍了 HID 设备类的结构与编程、USB 库函数编程、USB 键盘开发与 USB 鼠标开发。当然这些代码都是简单的实现功能，真正的产口还需要在这基础之上进一步完善与优化。

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

第六章 Audio 设备

6.1 Audio 设备介绍

USB 协议制定时，为了方便不同设备的开发商基于 USB 进行设计，定义了不同的设备类来支持不同类型的设备。虽然在 USB 标准中定义了 USB_DEVICE_CLASS_AUDIO—AUDIO 设备。但是很少有此类设备问世。目前称为 USB 音箱的设备，大都使用 USB_DEVICE_CLASS_POWER，仅仅将 USB 接口作为电源使用。完全基于 USB 协议的 USB_DEVICE_CLASS_AUDIO 设备，采用一根 USB 连接线，在设备中不同的端点实现音频信号的输入，输出包括相关按键控制。

AUDIO 设备是专门针对 USB 音频设备定义的一种专用类别，它不仅定义了音频输入、输出端点的标准，还提供了音量控制、混音器配置、左右声道平衡，甚至包括对支持杜比音效解码设备的支持，功能相当强大。不同的开发者可以根据不同的需求对主机枚举自己的设备结构，主机则根据枚举的不同设备结构提供相应的服务。

AUDIO 设备采用 USB 传输模式中的 Isochronous transfers 模式，Isochronous transfers 传输模式是专门针对流媒体特点的传输方法。它依照设备在链接初始化时枚举的参数，保证提供稳定的带宽给采用该模式的设备或端点。由于对实时性的要求，它不提供相应的接收 / 应答和握手协议。这很好地适应了音频数据流量稳定、对差错相对不敏感的特点。

6.2 Audio 描述符

Audio 设备定义了三种接口描述符子类，子协议恒为 0x00。三种接口描述符子类分别定义为：

```
// Audio Interface Subclass Codes
#define USB_ASC_AUDIO_CONTROL 0x01
#define USB_ASC_AUDIO_STREAMING 0x02
#define USB_ASC_MIDI_STREAMING 0x03
```

USB_ASC_AUDIO_CONTROL，音频设备控制类；USB_ASC_AUDIO_STREAMING，音频流类；USB_ASC_MIDI_STREAMING，MIDI 流类。以上三种用于描述接口描述符的子类，确定被描述接口的功能。Audio 设备没有定义子协议，所以为定值 0x00。

Audio 设备接口描述符中要包含 Class-Specific AC Interface Header Descriptor，用于定义接口的其它功能端口。Class-Specific AC Interface Header Descriptor（AC 接口头）描述符如下表：

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 USB_DTYPE_CS_INTERFACE (36)
2	bDescriptorSubtype	1	数字	AC 接口头: USB_DSUBTYPE_HEADER
3	bcdADC	2	数字	Audio 设备发行号, BCD 码
5	wTotalLength	2	数字	接口描述符总长度, 包括标准接口描述符
6	bInCollection	1	索引	AudioStreaming、MIDIStreaming 使用代码
7	baInterfaceNr	2	位图	使用 AudioStreaming 或者 MIDIStreaming 的接口号

表 1. AC 接口头描述符

C 语言 AC 接口头描述符结构体为:

```
typedef struct
{
    //本描述符长度.
    unsigned char bLength;
    //描述类型 USB_DTYPE_CS_INTERFACE (36).
    unsigned char bDescriptorSubtype;
    //发行号 (BCD 码)
    unsigned short bcdADC;
    //接口描述符总长度, 包括标准接口描述符
    unsigned short wTotalLength;
    // AudioStreaming、MIDIStreaming 使用代码
    unsigned char bInCollection;
    // 使用 AudioStreaming 或者 MIDIStreaming 的接口号
    unsigned char baInterfaceNr;
}
tACHeader;
```

例如: 定义一个实际的 AC 接口头描述符。

```
const unsigned char g_pAudioInterfaceHeader[] =
{
    9, // The size of this descriptor.
    USB_DTYPE_CS_INTERFACE, // Interface descriptor is class specific.
    USB_ACDSTYPE_HEADER, // Descriptor sub-type is HEADER.
    USBShort(0x0100), // Audio Device Class Specification Release
    // Number in Binary-Coded Decimal.
    // Total number of bytes in
    // g_pAudioControlInterface
    USBShort((9 + 9 + 12 + 13 + 9)),
    1, // Number of streaming interfaces.
    1, // Index of the first and only streaming interface.
}
```

Audio 设备的输入端口描述符（Input Terminal Descriptor），对于输入端定义。输入端口描述符如下表：

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 USB_DTYPE_CS_INTERFACE (36)
2	bDescriptorSubtype	1	常量	USB_ACDSTYPE_IN_TERMINAL
3	bTerminalID	1	常量	本端口 ID 号
4	wTerminalType	2	常量	端口类型
6	bAssocTerminal	1	常量	对应输出端口 ID
7	bNrChannels	2	位图	通道数量
8	wChannelConfig	2	数字	通道配置
10	iChannelNames	1	常量	通道名字
11	iTerminal	1	数字	端口字符串描述符索引

表 1. HID 描述符符

C 语言输入端口描述符结构体为：

```
typedef struct
{
    unsigned char bLength;
    unsigned char bDescriptorType;
    unsigned char bDescriptorSubtype;
    unsigned char bTerminalID;
    unsigned short wTerminalType;
    unsigned char bAssocTerminal;
    unsigned char bNrChannels;
    unsigned short wChannelConfig;
    unsigned char iChannelNames;
    unsigned char iTerminal;
}
tACInputTerminal;
```

Audio 设备的输出端口描述符（Output Terminal Descriptor），对于输出端定义。输出端口描述符如下表：

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 USB_DTYPE_CS_INTERFACE (36)
2	bDescriptorSubtype	1	常量	USB_ACDSTYPE_OUT_TERMINAL
3	bTerminalID	1	常量	本端口 ID 号
4	wTerminalType	2	常量	端口类型
6	bAssocTerminal	1	常量	对应输入端口 ID
7	bSourceID	1	常量	连接端口的 ID 号

8	iTerminal	1	数字	端口字符串描述符索引
---	-----------	---	----	------------

表 1. HID 描述符符

C 语言输出端口描述符结构体为：

```
typedef struct
{
    unsigned char bLength;
    unsigned char bDescriptorType;
    unsigned char bDescriptorSubtype;
    unsigned char bTerminalID;
    unsigned short wTerminalType;
    unsigned char bAssocTerminal;
    unsigned char bSourceID;
    unsigned char iTerminal;
}
tACOutputTerminal;
```

其它与 Audio 设备相关的描述符，请读者参阅 USB_Audio_Class 手册。下面列出一个 Audio 设备的接口描述符：

```
const unsigned char g_pAudioControlInterface[] =
{
    //标准接口描述符
    9, // Size of the interface descriptor.
    USB_DTYPE_INTERFACE, // Type of this descriptor.
    AUDIO_INTERFACE_CONTROL, // 接口编号，从 0 开发编. AUDIO_INTERFACE_CONTROL = 0
    0, // The alternate setting for this interface.
    0, // The number of endpoints used by this interface.
    USB_CLASS_AUDIO, // AUDIO 设备
    USB_ASC_AUDIO_CONTROL, // 子类，USB_ASC_AUDIO_CONTROL 用于 Audio 控制.
    0, // 无协议，定值 0。
    0, // The string index for this interface.
    // Audio 接口头描述符.
    9, // The size of this descriptor.
    USB_DTYPE_CS_INTERFACE, // 描述符类型.
    USB_ACDSTYPE_HEADER, // 子类型，本描述为描述头.
    USBShort(0x0100), // Audio Device Class Specification Release
    // Number in Binary-Coded Decimal.
    // Total number of bytes in
    // g_pAudioControlInterface
    USBShort((9 + 9 + 12 + 13 + 9)),
    1, // Number of streaming interfaces.
    1, // Index of the first and only streaming
    // interface.
    // Audio 设备输入端口描述符.
    12, // The size of this descriptor.
    USB_DTYPE_CS_INTERFACE, // Interface descriptor is class specific.
```

```

USB_ACDSTYPE_IN_TERMINAL, // 本描述符为输入端口.
AUDIO_IN_TERMINAL_ID,     // Terminal ID for this interface.
                           // USB streaming interface.
USBShort(USB_TTYPE_STREAMING),
0,                         // ID of the Output Terminal to which this
                           // Input Terminal is associated.
2,                         // Number of logical output channels in the
                           // Terminal output audio channel cluster.
USBShort((USB_CHANNEL_L | // Describes the spatial location of the
          USB_CHANNEL_R)), // logical channels.
0,                         // Channel Name string index.
0,                         // Terminal Name string index.

// Audio 设备特征单元描述符
13,                       // The size of this descriptor.
USB_DTYPE_CS_INTERFACE,   // Interface descriptor is class specific.
USB_ACDSTYPE_FEATURE_UNIT, // Descriptor sub-type is FEATURE_UNIT.
AUDIO_CONTROL_ID,         // Unit ID for this interface.
AUDIO_IN_TERMINAL_ID,     // ID of the Unit or Terminal to which this
                           // Feature Unit is connected.
2,                         // Size in bytes of an element of the
                           // bmaControls() array that follows.
                           // Master Mute control.
USBShort(USB_ACONTROL_MUTE),
                           // Left channel volume control.
USBShort(USB_ACONTROL_VOLUME),
                           // Right channel volume control.
USBShort(USB_ACONTROL_VOLUME),
0,                         // Feature unit string index.
//输出端口描述符.
9,                         // The size of this descriptor.
USB_DTYPE_CS_INTERFACE,   // Interface descriptor is class specific.
USB_ACDSTYPE_OUT_TERMINAL, // 输出端口描述符.
AUDIO_OUT_TERMINAL_ID,    // Terminal ID for this interface.
                           // Output type is a generic speaker.
USBShort(USB_ATYPE_SPEAKER),
AUDIO_IN_TERMINAL_ID,     // ID of the input terminal to which this
                           // output terminal is connected.
AUDIO_CONTROL_ID,         // ID of the feature unit that this output
                           // terminal is connected to.
0,                         // Output terminal string index.

};
//音频流接口

```

```

const unsigned char g_pAudioStreamInterface[] =
{
    //标准接口描述符
    9, // Size of the interface descriptor.
    USB_DTYPE_INTERFACE, // Type of this descriptor.
    AUDIO_INTERFACE_OUTPUT, // The index for this interface.
    0, // The alternate setting for this interface.
    0, // The number of endpoints used by this
        // interface.
    USB_CLASS_AUDIO, // The interface class
    USB_ASC_AUDIO_STREAMING, // The interface sub-class.
    0, // Unused must be 0.
    0, // The string index for this interface.
    // Vendor-specific Interface Descriptor.
    9, // Size of the interface descriptor.
    USB_DTYPE_INTERFACE, // Type of this descriptor.
    1, // The index for this interface.
    1, // The alternate setting for this interface.
    1, // The number of endpoints used by this
        // interface.
    USB_CLASS_AUDIO, // The interface class
    USB_ASC_AUDIO_STREAMING, // The interface sub-class.
    0, // Unused must be 0.
    0, // The string index for this interface.
    // Class specific Audio Streaming Interface descriptor.
    7, // Size of the interface descriptor.
    USB_DTYPE_CS_INTERFACE, // Interface descriptor is class specific.
    USB_AS_DSTYPE_GENERAL, // General information.
    AUDIO_IN_TERMINAL_ID, // ID of the terminal to which this streaming
        // interface is connected.
    1, // One frame delay.
    USBShort(USB_ADF_PCM), //
    // Format type Audio Streaming descriptor.
    11, // Size of the interface descriptor.
    USB_DTYPE_CS_INTERFACE, // Interface descriptor is class specific.
    USB_AS_DSTYPE_FORMAT_TYPE, // Audio Streaming format type.
    USB_AF_TYPE_TYPE_I, // Type I audio format type.
    2, // Two audio channels.
    2, // Two bytes per audio sub-frame.
    16, // 16 bits per sample.
    1, // One sample rate provided.
    USB3Byte(48000), // Only 48000 sample rate supported.
    //端点描述符
    9, // The size of the endpoint descriptor.

```

```

USB_DTYPE_ENDPOINT,          // Descriptor type is an endpoint.
                                // OUT endpoint with address
                                // ISOC_OUT_ENDPOINT.

USB_EP_DESC_OUT | USB_EP_TO_INDEX(ISOC_OUT_ENDPOINT),
USB_EP_ATTR_ISOC |           // Endpoint is an adaptive isochronous data
USB_EP_ATTR_ISOC_ADAPT |     // endpoint.
USB_EP_ATTR_USAGE_DATA,
USBShort(ISOC_OUT_EP_MAX_SIZE), // The maximum packet size.
1,                               // The polling interval for this endpoint.
0,                               // Refresh is unused.
0,                               // Synch endpoint address.
// Audio Streaming Isochronous Audio Data Endpoint Descriptor
7,                               // The size of the descriptor.
USB_ACSDT_ENDPOINT,           // Audio Class Specific Endpoint Descriptor.
USB_ASDSTYPE_GENERAL,         // This is a general descriptor.
USB_EP_ATTR_ACG_SAMPLING,     // Sampling frequency is supported.
USB_EP_LOCKDELAY_UNDEF,      // Undefined lock delay units.
USBShort(0),                  // No lock delay.
};

```

6.3 Audio 数据类型

usbdaudio.h 中已经定义好 Audio 设备类中使用的所有数据类型和函数,下面介绍 Audio 设备类使用的数据类型。

```

typedef struct
{
    unsigned long ulUSBBase;
    //设备信息指针
    tDeviceInfo *psDevInfo;
    //配置描述符
    tConfigDescriptor *psConfDescriptor;
    //最大音量值
    short sVolumeMax;
    //最小音量值
    short sVolumeMin;
    //音量控制阶梯值
    short sVolumeStep;
    struct
    {
        //callback 入口参数
        void *pvData;
        // pvData 大小
        unsigned long ulSize;
        // 可用 pvData 大小
        unsigned long ulNumBytes;
        // Callback
    }
};

```

```

        tUSBAudioBufferCallback pfnCallback;
    } sBuffer;
    //请求类型.
    unsigned short usRequestType;
    //请求标志
    unsigned char ucRequest;
    // 更新值
    unsigned short usUpdate;
    // 当前音量设置.
    unsigned short usVolume;
    // 静音设置.
    unsigned char ucMute;
    //采样率
    unsigned long ulSampleRate;
    // 使用输出端点
    unsigned char ucOUTEndpoint;
    // 输出端点 DMA 通道
    unsigned char ucOUTDMA;
    //控制接口
    unsigned char ucInterfaceControl;
    //Audio 接口
    unsigned char ucInterfaceAudio;
}
tAudioInstance;

```

tAudioInstance, Audio 设备类实例。用于保存全部 Audio 设备类的配置信息，包括描述符、callback 函数、控制事件等。

```

#define USB_AUDIO_INSTANCE_SIZE sizeof(tAudioInstance);
#define COMPOSITE_DAUDIO_SIZE (8 + 52 + 52)

```

USB_AUDIO_INSTANCE_SIZE , 定义 Audio 设备类实例信息的大小。
COMPOSITE_DAUDIO_SIZE 定义设备描述符与所有接口描述符总长度。

```

typedef struct
{
    // VID
    unsigned short usVID;
    // PID
    unsigned short usPID;
    //8 字节供应商字符串
    unsigned char pucVendor[8];
    //16 字节产品字符串
    unsigned char pucProduct[16];
    //4 字节版本号
    unsigned char pucVersion[4];
    //最大耗电量
    unsigned short usMaxPowermA;
}

```

```

//电源属性  USB_CONF_ATTR_SELF_PWR  USB_CONF_ATTR_BUS_PWR
//USB_CONF_ATTR_RWAKE.
unsigned char ucPwrAttributes;
// callback 函数
tUSBCallback pfnCallback;
//字符串描述符集合
const unsigned char * const *ppStringDescriptors;
//字符串描述符个数
unsigned long ulNumStringDescriptors;
//最大音量
short sVolumeMax;
//最小音量
short sVolumeMin;
//音量调节步进
short sVolumeStep;
//Audio 设备类实例
tAudioInstance *psPrivateData;
}
tUSBDAudioDevice;

```

tUSBDAudioDevice, Audio 设备类。定义了 VID、PID、电源属性、字符串描述符等，还包括了一个 Audio 设备类实例。其它设备描述符、配置信息通过 API 函数储入 tAudioInstance 定义的 Audio 设备实例中。

6.4 API 函数

在 Audio 设备类 API 库中定义了 4 个函数，完成 USB Audio 设备初始化、配置及数据处理。下面为 usbdaudio.h 中定义的 API 函数：

```

void *USBDAudioInit(unsigned long ulIndex,
                    const tUSBDAudioDevice *psAudioDevice);

void *USBDAudioCompositeInit(unsigned long ulIndex,
                             const tUSBDAudioDevice *psAudioDevice);

int USBAudioBufferOut(void *pvInstance, void *pvBuffer, unsigned long ulSize,
                    tUSBAudioBufferCallback pfnCallback);

void USBDAudioTerm(void *pvInstance);

```

```

void *USBDAudioInit(unsigned long ulIndex,
                    const tUSBDAudioDevice *psAudioDevice);

```

作用：初始化 Audio 设备硬件、协议，把其它配置参数填入 psAudioDevice 实例中。

参数：ulIndex, USB 模块代码，固定值：USB_BASE0。psAudioDevice, Audio 设备类。

返回：指向配置后的 tUSBDAudioDevice。

```

void *USBDAudioCompositeInit(unsigned long ulIndex,
                             const tUSBDAudioDevice *psAudioDevice);

```

作用：初始化 Audio 设备协议，本函数在 USBDAudioInit 中已经调用。

参数：ulIndex, USB 模块代码，固定值：USB_BASE0。psAudioDevice, Audio 设备类。

返回：指向配置后的 tUSBDAudioDevice。

```

int USBAudioBufferOut(void *pvInstance, void *pvBuffer, unsigned long ulSize,

```



```
tUSBAudioBufferCallback pfnCallback);
```

作用：从输出端点获取数据，并放入 pvBuffer 中。

参数：pvInstance，指向 tUSBDAudioDevice。pvBuffer，用于存放输出端点数据。ulSize，设置数据大小。pfnCallback，输出端点返回，只有一个事件，USBD_AUDIO_EVENT_DATAOUT。

返回：无。

```
void USBDAudioTerm(void *pvInstance);
```

作用：结束 Audio 设备。

参数：pvInstance，指向 tUSBDAudioDevice。

返回：无。

在这些函数中 USBDAudioInit 和 USBAudioBufferOut 函数最重要并且使用最多，USBDAudioInit 第一次使用 Audio 设备时，用于初始化 Audio 设备的配置与控制。USBAudioBufferOut 从输出端点中获取数据，并放入数据缓存区内。

6.5 Audio 设备开发

Audio 设备开发只需要 5 步就能完成。如图 2 所示，Audio 设备配置（主要是字符串描述符）、callback 函数编写、USB 处理器初始化、DMA 控制、数据处理。

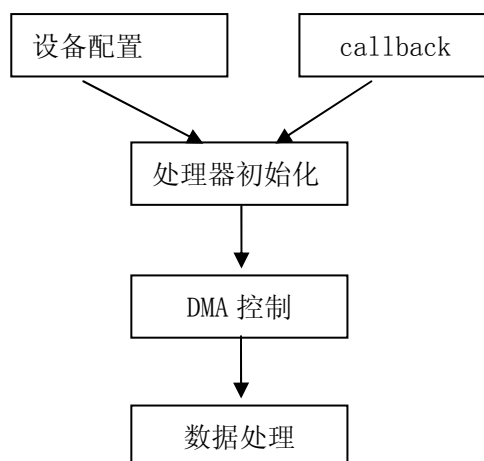


图 2

第一步：Audio 设备配置（主要是字符串描述符），按字符串描述符标准完成串描述符配置，进而完成 Audio 设备配置。

```

#include "inc/hw_ints.h"
#include "inc/hw_memmap.h"
#include "inc/hw_types.h"
#include "inc/hw_sysctl.h"
#include "inc/hw_udma.h"
#include "inc/hw_gpio.h"
#include "driverlib/gpio.h"
#include "driverlib/interrupt.h"
#include "driverlib/sysctl.h"
#include "driverlib/udma.h"
#include "usblib/usb.h"
#include "usblib/usb-ids.h"

```

```

#include "usblib/device/usbdevice.h"
#include "usblib/device/usbdaudio.h"

//根据具体 Audio 芯片修改
#define VOLUME_MAX          ((short)0x0C00)  // +12db
#define VOLUME_MIN          ((short)0xDC80)  // -34.5db
#define VOLUME_STEP         ((short)0x0180)  // 1.5db
//Audio 设备
const tUSBDAudioDevice g_sAudioDevice;
//DMA 控制
tDMAControlTable sDMAControlTable[64] __attribute__((aligned(1024)));
//*****
// 缓存与标志.
//*****
#define AUDIO_PACKET_SIZE    ((48000*4)/1000)
#define AUDIO_BUFFER_SIZE    (AUDIO_PACKET_SIZE*20)
#define SBUFFER_FLAGS_PLAYING 0x00000001
#define SBUFFER_FLAGS_FILLING 0x00000002
struct
{
    //主要 buffer, USB audio class 和 sound driver 使用.
    volatile unsigned char pucBuffer[AUDIO_BUFFER_SIZE];
    // play pointer.
    volatile unsigned char *pucPlay;
    // USB fill pointer.
    volatile unsigned char *pucFill;
    // 采样率 调整.
    volatile int iAdjust;
    // 播放状态
    volatile unsigned long ulFlags;
} g_sBuffer;
//*****
// 当前音量
//*****
short g_sVolume;
//*****
// 通过 USBDAudioInit() 函数, 完善 Audio 设备配置信息
//*****
void *g_pvAudioDevice;
// 音量更新
#define FLAG_VOLUME_UPDATE    0x00000001
// 更新静音状态
#define FLAG_MUTE_UPDATE      0x00000002
// 静音状态

```

```

#define FLAG_MUTED                0x00000004
// 连接成功

#define FLAG_CONNECTED            0x00000008
volatile unsigned long g_ulFlags;
extern unsigned long
AudioMessageHandler(void *pvCBData, unsigned long ulEvent,
                    unsigned long ulMsgParam, void *pvMsgData);
//*****
// 设备语言描述符.
//*****
const unsigned char g_pLangDescriptor[] =
{
    4,
    USB_DTYPE_STRING,
    USBShort(USB_LANG_EN_US)
};
//*****
// 制造商 字符串 描述符
//*****
const unsigned char g_pManufacturerString[] =
{
    (17 + 1) * 2,
    USB_DTYPE_STRING,
    'T', 0, 'e', 0, 'x', 0, 'a', 0, 's', 0, ' ', 0, 'I', 0, 'n', 0, 's', 0,
    't', 0, 'r', 0, 'u', 0, 'm', 0, 'e', 0, 'n', 0, 't', 0, 's', 0,
};
//*****
//产品 字符串 描述符
//*****
const unsigned char g_pProductString[] =
{
    (13 + 1) * 2,
    USB_DTYPE_STRING,
    'A', 0, 'u', 0, 'd', 0, 'i', 0, 'o', 0, ' ', 0, 'E', 0, 'x', 0, 'a', 0,
    'm', 0, 'p', 0, 'l', 0, 'e', 0,
};
//*****
// 产品 序列号 描述符
//*****
const unsigned char g_pSerialNumberString[] =
{
    (8 + 1) * 2,
    USB_DTYPE_STRING,
    '1', 0, '2', 0, '3', 0, '4', 0, '5', 0, '6', 0, '7', 0, '8', 0
};

```

```

};
//*****
// 设备接口字符串描述符
//*****
const unsigned char g_pInterfaceString[] =
{
    (15 + 1) * 2,
    USB_DTYPE_STRING,
    'A', 0, 'u', 0, 'd', 0, 'i', 0, 'o', 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符
//*****
const unsigned char g_pConfigString[] =
{
    (20 + 1) * 2,
    USB_DTYPE_STRING,
    'A', 0, 'u', 0, 'd', 0, 'i', 0, 'o', 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
};
//*****
// 字符串描述符集合
//*****
const unsigned char * const g_pStringDescriptors[] =
{
    g_pLangDescriptor,
    g_pManufacturerString,
    g_pProductString,
    g_pSerialNumberString,
    g_pInterfaceString,
    g_pConfigString
};
#define NUM_STRING_DESCRIPTOR (sizeof(g_pStringDescriptors) / \
                                sizeof(unsigned char *))
//*****
// 定义 Audio 设备实例
//*****
static tAudioInstance g_sAudioInstance;
//*****
// 定义 Audio 设备类
//*****
const tUSBDAudioDevice g_sAudioDevice =

```

```

{
    // VID
    USB_VID_STELLARIS,
    // PID
    USB_PID_AUDIO,
    // 8 字节供应商字符串.
    "TI      ",
    //16 字节产品字符串.
    "Audio Device  ",
    //4 字节版本字符串.
    "1.00",
    500,
    USB_CONF_ATTR_SELF_PWR,
    AudioMessageHandler,
    g_pStringDescriptors,
    NUM_STRING_DESCRIPTOR,
    VOLUME_MAX,
    VOLUME_MIN,
    VOLUME_STEP,
    &g_sAudioInstance
};

```

第二步：完成 Callback 函数。Callback 函数用于处理输出端点数据事务。主机发出的音频流数据，也可能是状态信息。Audio 设备中包含了以下事务：USBD_AUDIO_EVENT_IDLE、USBD_AUDIO_EVENT_ACTIVE 、 USBD_AUDIO_EVENT_MUTE 、 USBD_AUDIO_EVENT_VOLUME 、 USB_EVENT_DISCONNECTED、USB_EVENT_CONNECTED。如下表：

名称	说明
USB_EVENT_CONNECTED	USB 设备已经连接到主机
USB_EVENT_DISCONNECTED	USB 设备已经与主机断开
USBD_AUDIO_EVENT_VOLUME	更新音量
USBD_AUDIO_EVENT_MUTE	静音
USBD_AUDIO_EVENT_ACTIVE	Audio 处于活动状态
USBD_AUDIO_EVENT_IDLE	Audio 处于空闲状态

表 2. Audio 事务

根据以上事务编写 Callback 函数：

```

//*****
//USB Audio 设备类返回事件处理函数（callback）.
//*****
unsigned long AudioMessageHandler(void *pvCBData, unsigned long ulEvent,
                                unsigned long ulMsgParam, void *pvMsgData)
{
    switch(ulEvent)
    {
        //Audio 正处于空闲或者工作状态。
        case USBD_AUDIO_EVENT_IDLE:

```

```

case USBD_AUDIO_EVENT_ACTIVE:
{
    GPIOPinWrite(GPIO_PORTF_BASE, 0x10, 0x10);
    g_ulFlags |= FLAG_CONNECTED;
    break;
}
// 静音控制.
case USBD_AUDIO_EVENT_MUTE:
{
    // 检查是否静音.
    if (ulMsgParam == 1)
    {
        // 静音.
        g_ulFlags |= FLAG_MUTE_UPDATE | FLAG_MUTED;
    }
    else
    {
        // 取消静音.
        g_ulFlags &= ~(FLAG_MUTE_UPDATE | FLAG_MUTED);
        g_ulFlags |= FLAG_MUTE_UPDATE;
    }
    break;
}
// 音量控制.
case USBD_AUDIO_EVENT_VOLUME:
{
    g_ulFlags |= FLAG_VOLUME_UPDATE;
    // 最大音量.
    if (ulMsgParam == 0x8000)
    {
        // 设置为最小
        g_sVolume = 0;
    }
    else
    {
        // 声音控制器, 设置音量.
        g_sVolume = (short)ulMsgParam - (short)VOLUME_MIN;
    }
    break;
}
// 断开连接.
case USB_EVENT_DISCONNECTED:
{
    GPIOPinWrite(GPIO_PORTF_BASE, 0x10, 0x00);

```

```

        GPIOWrite(GPIO_PORTF_BASE, 0x80, 0x00);
        g_ulFlags &= ~FLAG_CONNECTED;
        break;
    }
    //连接
    case USB_EVENT_CONNECTED:
    {
        GPIOWrite(GPIO_PORTF_BASE, 0x80, 0x80);
        g_ulFlags |= FLAG_CONNECTED;
        break;
    }
    default:
    {
        break;
    }
}
return(0);
}

```

第三步：系统初始化，配置内核电压、系统主频、使能端口、配置按键端口、LED 控制等，本例中使用 4 个 LED 进行指示。原理图如图 3 所示：

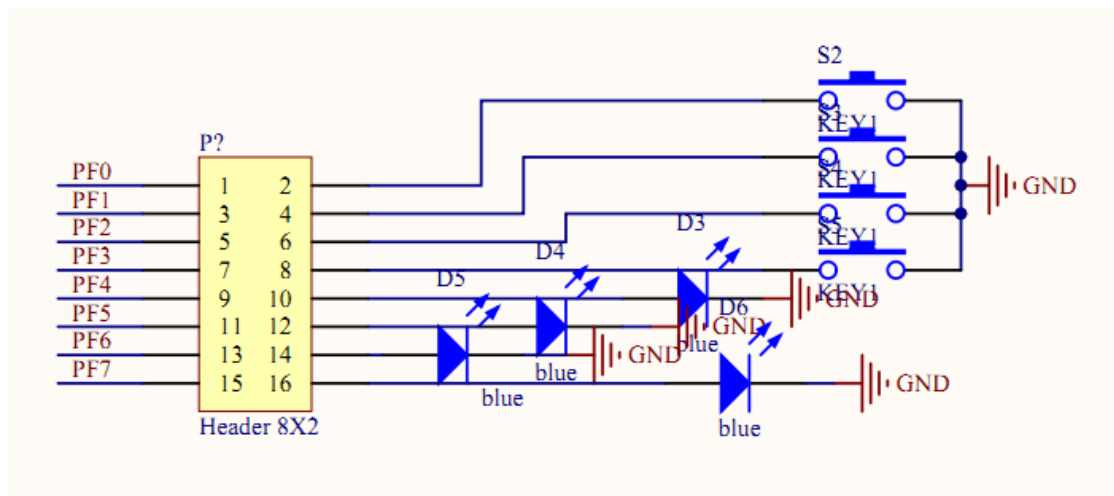


图 3

系统初始化：

```

//设置内核电压、主频 50Mhz
SysCtlLDOSet(SYSCTL_LDO_2_75V);
SysCtlClockSet(SYSCTL_XTAL_8MHZ | SYSCTL_SYSDIV_4 | SYSCTL_USE_PLL | SYSCTL_OSC_MAIN);
SysCtlPeripheralEnable(SYSCTL_PERIPH_GPIOF);
GPIOPinTypeGPIOOutput(GPIO_PORTF_BASE, 0xf0);
GPIOPinTypeGPIOInput(GPIO_PORTF_BASE, 0x0f);
HWREG(GPIO_PORTF_BASE+GPIO_O_PUR) |= 0x0f;
// 全局状态标志.
g_ulFlags = 0;
// 初始化 Audio 设备.

```

```
g_pvAudioDevice = USBDAudioInit(0, (tUSBDAudioDevice *)&g_sAudioDevice);
```

第四步：使能、配置 DMA，Audio 设备要传输大量数据，所以 USB 库函数内部已经使用了 DMA，在使用前必须使能、配置 DMA。

```
//配置使能 DMA
SysCtlPeripheralEnable(SYSCTL_PERIPH_UDMA);
SysCtlDelay(10);
uDMAControlBaseSet(&sDMAControlTable[0]);
uDMAEnable();
```

第五步：数据处理。主要使用 USBAudioBufferOut 从输出端点中获取数据并处理，并且进行 Audio 设备控制。

```
while(1)
{
    //等待连接结束.
    while((g_ulFlags & FLAG_CONNECTED) == 0)
    {
    }

    //初始化 Buffer
    g_sBuffer.pucFill = g_sBuffer.pucBuffer;
    g_sBuffer.pucPlay = g_sBuffer.pucBuffer;
    g_sBuffer.ulFlags = 0;

    //从 Audio 设备类中获取数据
    if(USBAudioBufferOut(g_pvAudioDevice,
                        (unsigned char *)g_sBuffer.pucFill,
                        AUDIO_PACKET_SIZE, USBBufferCallback) == 0)
    {
        //标记数据放入 buffer 中.
        g_sBuffer.ulFlags |= SBUFFER_FLAGS_FILLING;
    }

    //设备连接到主机.
    while(g_ulFlags & FLAG_CONNECTED)
    {
        // 检查音量是否有改变.
        if(g_ulFlags & FLAG_VOLUME_UPDATE)
        {
            // 清除更新音量标志.
            g_ulFlags &= ~FLAG_VOLUME_UPDATE;

            // 修改音量，自行添加代码. 在此以 LED 灯做指示。
            //UpdateVolume();

            GPIOPinWrite(GPIO_PORTF_BASE, 0x40, ~GPIOPinRead(GPIO_PORTF_BASE, 0x40));
        }

        //是否静音
        if(g_ulFlags & FLAG_MUTE_UPDATE)
        {

```



```

//修改静音状态, 自行添加函数. 在此以 LED 灯做指示。
//UpdateMute();
    if(g_ulFlags & FLAG_MUTED)
        GPIOPinWrite(GPIO_PORTF_BASE, 0x20, 0x20);
    else
        GPIOPinWrite(GPIO_PORTF_BASE, 0x20, 0x00);
// 清除静音标志
g_ulFlags &= ~FLAG_MUTE_UPDATE;
}
}
}

//*****
//USBAudioBufferOut 的 Callback 入口参数
//*****
void USBBufferCallback(void *pvBuffer, unsigned long ulParam, unsigned long ulEvent)
{
    //数据处理, 自行加入代码。
    // Your Codes .....
    //再一次获取数据.
    USBAudioBufferOut(g_pvAudioDevice, (unsigned char *)g_sBuffer.pucFill,
        AUDIO_PACKET_SIZE, USBBufferCallback);
}

```

使用上面五步就完成 Audio 设备开发。Audio 设备开发时要加入两个 lib 库函数：usb.lib 和 DriverLib.lib，在启动代码中加入 USB0DeviceIntHandler 中断服务函数。以上 Audio 设备开发完成，在 Win xp 下运行效果如下图所示：



在枚举过程中可以看出，在电脑右下角可以看到“Audio Example”字样，标示正在进行枚举。枚举成功后，在“设备管理器”的“声音、视频和游戏控制器”中看到“USB Audio Device”设备，如下图。现在 Audio 设备可以正式使用。



Audio 设备开发源码如下：

```

#include "inc/hw_ints.h"
#include "inc/hw_memmap.h"
#include "inc/hw_types.h"
#include "inc/hw_sysctl.h"
#include "inc/hw_udma.h"
#include "inc/hw_gpio.h"
#include "driverlib/gpio.h"
#include "driverlib/interrupt.h"
#include "driverlib/sysctl.h"
#include "driverlib/udma.h"
#include "usblib/usblib.h"
#include "usblib/usb-ids.h"
#include "usblib/device/usbdevice.h"
#include "usblib/device/usbdaudio.h"

//根据具体 Audio 芯片修改
#define VOLUME_MAX ((short)0x0C00) // +12db
#define VOLUME_MIN ((short)0xDC80) // -34.5db
#define VOLUME_STEP ((short)0x0180) // 1.5db

//Audio 设备
const tUSBDAudioDevice g_sAudioDevice;

//DMA 控制
tDMAControlTable sDMAControlTable[64] __attribute__((aligned(1024)));
//*****

// 缓存与标志.
//*****

#define AUDIO_PACKET_SIZE ((48000*4)/1000)
#define AUDIO_BUFFER_SIZE (AUDIO_PACKET_SIZE*20)
#define SBUFFER_FLAGS_PLAYING 0x00000001
#define SBUFFER_FLAGS_FILLING 0x00000002

struct
{
    //主要 buffer, USB audio class 和 sound driver 使用.
    volatile unsigned char pucBuffer[AUDIO_BUFFER_SIZE];
    // play pointer.
    volatile unsigned char *pucPlay;
    // USB fill pointer.
    volatile unsigned char *pucFill;
    // 采样率 调整.
    volatile int iAdjust;
    // 播放状态
    volatile unsigned long ulFlags;
} g_sBuffer;

//*****

```

```

// 当前音量
//*****
short g_sVolume;
//*****
// 通过 USBDAudioInit() 函数, 完善 Audio 设备配置信息
//*****
void *g_pvAudioDevice;
// 音量更新
#define FLAG_VOLUME_UPDATE      0x00000001
// 更新静音状态
#define FLAG_MUTE_UPDATE        0x00000002
// 静音状态
#define FLAG_MUTED              0x00000004
// 连接成功
#define FLAG_CONNECTED          0x00000008
volatile unsigned long g_ulFlags;
extern unsigned long
AudioMessageHandler(void *pvCBData, unsigned long ulEvent,
                    unsigned long ulMsgParam, void *pvMsgData);
//*****
// 设备语言描述符.
//*****
const unsigned char g_pLangDescriptor[] =
{
    4,
    USB_DTYPE_STRING,
    USBShort(USB_LANG_EN_US)
};
//*****
// 制造商 字符串 描述符
//*****
const unsigned char g_pManufacturerString[] =
{
    (17 + 1) * 2,
    USB_DTYPE_STRING,
    'T', 0, 'e', 0, 'x', 0, 'a', 0, 's', 0, ' ', 0, 'I', 0, 'n', 0, 's', 0,
    't', 0, 'r', 0, 'u', 0, 'm', 0, 'e', 0, 'n', 0, 't', 0, 's', 0,
};
//*****
//产品 字符串 描述符
//*****
const unsigned char g_pProductString[] =
{
    (13 + 1) * 2,

```

```

        USB_DTYPE_STRING,
        'A', 0, 'u', 0, 'd', 0, 'i', 0, 'o', 0, ' ', 0, 'E', 0, 'x', 0, 'a', 0,
        'm', 0, 'p', 0, 'l', 0, 'e', 0
    };

//*****
// 产品 序列号 描述符
//*****
const unsigned char g_pSerialNumberString[] =
{
    (8 + 1) * 2,
    USB_DTYPE_STRING,
    '1', 0, '2', 0, '3', 0, '4', 0, '5', 0, '6', 0, '7', 0, '8', 0
};

//*****
// 设备接口字符串描述符
//*****
const unsigned char g_pInterfaceString[] =
{
    (15 + 1) * 2,
    USB_DTYPE_STRING,
    'A', 0, 'u', 0, 'd', 0, 'i', 0, 'o', 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符
//*****
const unsigned char g_pConfigString[] =
{
    (20 + 1) * 2,
    USB_DTYPE_STRING,
    'A', 0, 'u', 0, 'd', 0, 'i', 0, 'o', 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
};

//*****
// 字符串描述符集合
//*****
const unsigned char * const g_pStringDescriptors[] =
{
    g_pLangDescriptor,
    g_pManufacturerString,
    g_pProductString,
    g_pSerialNumberString,
    g_pInterfaceString,

```

```

        g_pConfigString
};

#define NUM_STRING_DESCRIPTOR (sizeof(g_pStringDescriptors) /
                                sizeof(unsigned char *))

//*****
// 定义 Audio 设备实例
//*****

static tAudioInstance g_sAudioInstance;

//*****
// 定义 Audio 设备类
//*****

const tUSBDAudioDevice g_sAudioDevice =
{
    // VID
    USB_VID_STELLARIS,
    // PID
    USB_PID_AUDIO,
    // 8 字节供应商字符串.
    "TI",
    //16 字节产品字符串.
    "Audio Device",
    //4 字节版本字符串.
    "1.00",
    500,
    USB_CONF_ATTR_SELF_PWR,
    AudioMessageHandler,
    g_pStringDescriptors,
    NUM_STRING_DESCRIPTOR,
    VOLUME_MAX,
    VOLUME_MIN,
    VOLUME_STEP,
    &g_sAudioInstance
};

//*****
//USB Audio 设备类返回事件处理函数 (callback) .
//*****

unsigned long AudioMessageHandler(void *pvCBData, unsigned long ulEvent,
                                unsigned long ulMsgParam, void *pvMsgData)
{
    switch(ulEvent)
    {
        //Audio 正处于空闲或者工作状态。
        case USBD_AUDIO_EVENT_IDLE:
        case USBD_AUDIO_EVENT_ACTIVE:

```

```

{
    GPIOPinWrite(GPIO_PORTF_BASE, 0x10, 0x10);
    g_ulFlags |= FLAG_CONNECTED;
    break;
}

// 静音控制.
case USB_AUDIO_EVENT_MUTE:
{
    // 检查是否静音.
    if (ulMsgParam == 1)
    {
        // 静音.
        g_ulFlags |= FLAG_MUTE_UPDATE | FLAG_MUTED;
    }
    else
    {
        // 取消静音.
        g_ulFlags &= ~(FLAG_MUTE_UPDATE | FLAG_MUTED);
        g_ulFlags |= FLAG_MUTE_UPDATE;
    }
    break;
}

// 音量控制.
case USB_AUDIO_EVENT_VOLUME:
{
    g_ulFlags |= FLAG_VOLUME_UPDATE;
    // 最大音量.
    if (ulMsgParam == 0x8000)
    {
        // 设置为最小
        g_sVolume = 0;
    }
    else
    {
        // 声音控制器, 设置音量.
        g_sVolume = (short)ulMsgParam - (short)VOLUME_MIN;
    }
    break;
}

// 断开连接.
case USB_EVENT_DISCONNECTED:
{
    GPIOPinWrite(GPIO_PORTF_BASE, 0x10, 0x00);
    GPIOPinWrite(GPIO_PORTF_BASE, 0x80, 0x00);
}

```

```

        g_ulFlags &= ~FLAG_CONNECTED;
        break;
    }
    case USB_EVENT_CONNECTED:
    {
        GPIOPinWrite(GPIO_PORTF_BASE, 0x80, 0x80);
        g_ulFlags |= FLAG_CONNECTED;
        break;
    }
    default:
    {
        break;
    }
}
return(0);
}

//*****
//USBAudioBufferOut 的 Callback 入口参数
//*****
void USBBufferCallback(void *pvBuffer, unsigned long ulParam, unsigned long ulEvent)
{
    //数据处理，自行加入代码。
    // Your Codes .....
    //再一次获取数据.
    USBAudioBufferOut(g_pvAudioDevice, (unsigned char *)g_sBuffer.pucFill,
        AUDIO_PACKET_SIZE, USBBufferCallback);
}

//*****
// 应用主函数.
//*****
int main(void)
{
    //设置内核电压、主频 50Mhz
    SysCtlLDOSet(SYSCTL_LDO_2_75V);
    SysCtlClockSet(SYSCTL_XTAL_8MHZ | SYSCTL_SYSDIV_4 |
        SYSCTL_USE_PLL | SYSCTL_OSC_MAIN );
    SysCtlPeripheralEnable(SYSCTL_PERIPH_GPIOF);
    GPIOPinTypeGPIOOutput(GPIO_PORTF_BASE, 0xf0);
    GPIOPinTypeGPIOInput(GPIO_PORTF_BASE, 0x0f);
    HWREG(GPIO_PORTF_BASE+GPIO_O_PUR) |= 0x0f;

    //配置使能 DMA
    SysCtlPeripheralEnable(SYSCTL_PERIPH_UDMA);

```

```

SysCtlDelay(10);
uDMAControlBaseSet(&sDMAControlTable[0]);
uDMAEnable();

// 全局状态标志.
g_ulFlags = 0;
// 初始化 Audio 设备.
g_pvAudioDevice = USBDAudioInit(0, (tUSBDAudioDevice *)&g_sAudioDevice);
while(1)
{
    //等待连接结束.
    while((g_ulFlags & FLAG_CONNECTED) == 0)
    {
    }

    //初始化 Buffer
    g_sBuffer.pucFill = g_sBuffer.pucBuffer;
    g_sBuffer.pucPlay = g_sBuffer.pucBuffer;
    g_sBuffer.ulFlags = 0;
    //从 Audio 设备类中获取数据
    if(USBAudioBufferOut(g_pvAudioDevice,
                        (unsigned char *)g_sBuffer.pucFill,
                        AUDIO_PACKET_SIZE, USBBufferCallback) == 0)
    {
        //标记数据放入 buffer 中.
        g_sBuffer.ulFlags |= SBUFFER_FLAGS_FILLING;
    }
    //设备连接到主机.
    while(g_ulFlags & FLAG_CONNECTED)
    {
        // 检查音量是否有改变.
        if(g_ulFlags & FLAG_VOLUME_UPDATE)
        {
            // 清除更新音量标志.
            g_ulFlags &= ~FLAG_VOLUME_UPDATE;
            // 修改音量, 自行添加代码. 在此以 LED 灯做指示。
            //UpdateVolume();
            GPIOPinWrite(GPIO_PORTF_BASE, 0x40, ~GPIOPinRead(GPIO_PORTF_BASE, 0x40));
        }
        //是否静音
        if(g_ulFlags & FLAG_MUTE_UPDATE)
        {
            //修改静音状态, 自行添加函数. 在此以 LED 灯做指示。
            //UpdateMute();
            if(g_ulFlags & FLAG_MUTED)

```



```

        GPIOPinWrite(GPIO_PORTF_BASE, 0x20, 0x20);
    else
        GPIOPinWrite(GPIO_PORTF_BASE, 0x20, 0x00);
    // 清除静音标志
    g_ulFlags &= ~FLAG_MUTE_UPDATE;
}
}
}
}
}

```

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

第七章 Bulk 设备

7.1 bulk 设备介绍

USB 通道的数据传输格式有两种，而且这两种格式还是互斥的。有消息和流两种对于流状态，不具有 usb 数据的格式，遵循的规则就是先进先出。对于消息通道，它的通信模式符合 usb 的数据格式，一般有三个阶段组成首先是建立阶段，数据阶段，确认阶段。所有的通信的开始都是由主机方面发起的。

USB 协议制定时，为了方便不同设备的开发商基于 USB 进行设计，定义了不同的设备类来支持不同类型的设备。Bulk 也是其中一种，Bulk 设备使用端点 Bulk 传输模式，可以快速传输大量数据。

批量传输 (BULK) 采用的是流状态传输，批量传送的一个特点就是支持不确定时间内进行大量数据传输，能够保证数据一定可以传输，但是不能保证传输的带宽和传输的延迟。而且批量传输是一种单向的传输，要进行双向传输必须要使用两个通道。本章将介绍双向 BULK 的批量传输。

7.2 bulk 数据类型

usbdbulk.h、usblib.h 中已经定义好 Bulk 设备类中使用的所有数据类型和函数，下面介绍 Bulk 设备类使用的数据类型。

```
typedef struct
{
    //定义本 Buffer 是用于传输还是接收，True 为接收，False 为发送。
    tBoolean bTransmitBuffer;
    //Callback 函数，用于 Buffer 数据处理完成后
    tUSBCallback pfnCallback;
    //Callback 第一个输入参数
    void *pvCBData;
    //数据传输或者接收时调用的函数，用于完成发送或者接收的函数。
    tUSBPacketTransfer pfnTransfer;
    //数据传输或者接收时调用的函数。
    //发送时，用于检查是否有足够空间；接收时，用于检查可以接收的数量。
    tUSBPacketAvailable pfnAvailable;
```

```

//在设备模式下，设备类指针
void *pvHandle;
//用于存放发送或者接收的数据.
unsigned char *pcBuffer;
//发送或者接收数据大小
unsigned long ulBufferSize;
//RAM Buffer
void *pvWorkspace;
}
tUSBBuffer;

```

tUSBBuffer，数据缓存控制结构体，定义在 usblib.h 中，用于在 Bulk 传输过程中，发送数据或者接收数据。是 Bulk 设备传输的主要载体，结构体内部包含数据发送、接收、处理函数等。

```

typedef enum
{
    //Bulk 状态没定义
    BULK_STATE_UNCONFIGURED,
    //空闲状态
    BULK_STATE_IDLE,
    //等待数据发送或者结束
    BULK_STATE_WAIT_DATA,
    //等待数据处理.
    BULK_STATE_WAIT_CLIENT
} tBulkState;

```

tBulkState，定义 Bulk 端点状态。定义在 usdbulk.h。用于端点状态标记与控制，可以保证数据传输不相互冲突。

```

typedef struct
{
    //USB 基地址
    unsigned long ulUSBBase;
    //设备信息
    tDeviceInfo *psDevInfo;
    //配置信息
    tConfigDescriptor *psConfDescriptor;
    //Bulk 接收端点状态
    volatile tBulkState eBulkRxState;
    //Bulk 发送端点状态
    volatile tBulkState eBulkTxState;
    //标志位
    volatile unsigned short usDeferredOpFlags;
    //最后一次发送数据大小
    unsigned short usLastTxSize;
    //连接是否成功
    volatile tBoolean bConnected;
}

```

```

//IN 端点号
unsigned char ucINEndpoint;
//OUT 端点号
unsigned char ucOUTEndpoint;
//接口号
unsigned char ucInterface;
}
tBulkInstance;

```

tBulkInstance, Bulk 设备类实例。定义了 Bulk 设备类的 USB 基地址、设备信息、IN 端点、OUT 端点等信息。

```

typedef struct
{
    //VID
    unsigned short usVID;
    //PID
    unsigned short usPID;
    //最大耗电量
    unsigned short usMaxPowermA;
    //电源属性
    unsigned char ucPwrAttributes;
    //接收回调函数，主要用于接收数据处理
    tUSBCallback pfnRxCallback;
    //接收回调函数的第一个参数。
    void *pvRxCBData;
    //发送回调函数，主要用于发送数据处理
    tUSBCallback pfnTxCallback;
    //发送回调函数的第一个参数。
    void *pvTxCBData;
    //字符串描述符集合
    const unsigned char * const *ppStringDescriptors;
    //字符串描述符个数
    unsigned long ulNumStringDescriptors;
    //Bulk 设备实例
    tBulkInstance *psPrivateBulkData;
}
tUSBDBulkDevice;

```

tUSBDBulkDevice, Bulk 设备类，定义了 VID、PID、电源属性、字符串描述符等，还包括了一个 Bulk 设备类实例。其它设备描述符、配置信息通过 API 函数储入 tBulkInstance 定义的 Bulk 设备实例中。

7.3 API 函数

在 Bulk 设备类 API 库中定义了 11 个函数，完成 USB Bulk 设备初始化、配置及数据处理。以及 11 个 Buffer 操作函数，下面为 usdbulk.h 中定义的 API 函数：

```

void *USBDBulkInit(unsigned long ulIndex,
                    const tUSBDBulkDevice *psDevice);

```

```

void *USBDBulkCompositeInit(unsigned long ulIndex,
                           const tUSBDBulkDevice *psDevice);

unsigned long USBDBulkPacketWrite(void *pvInstance,
                                  unsigned char *pcData,
                                  unsigned long ullength,
                                  tBoolean bLast);

unsigned long USBDBulkPacketRead(void *pvInstance,
                                 unsigned char *pcData,
                                 unsigned long ullength,
                                 tBoolean bLast);

unsigned long USBDBulkTxPacketAvailable(void *pvInstance);
unsigned long USBDBulkRxPacketAvailable(void *pvInstance);
void USBDBulkTerm(void *pvInstance);
void *USBDBulkSetRxCBData(void *pvInstance, void *pvCBData);
void *USBDBulkSetTxCBData(void *pvInstance, void *pvCBData);
void USBDBulkPowerStatusSet(void *pvInstance, unsigned char ucPower);
tBoolean USBDBulkRemoteWakeupRequest(void *pvInstance);

```

```

void *USBDBulkInit(unsigned long ulIndex,
                  const tUSBDBulkDevice *psDevice);

```

作用：初始化 Bulk 设备硬件、协议，把其它配置参数填入 psDevice 实例中。

参数：ulIndex，USB 模块代码，固定值：USB_BASE0。psDevice，Bulk 设备类。

返回：指向配置后的 tUSBDBulkDevice。

```

void *USBDBulkCompositeInit(unsigned long ulIndex,
                           const tUSBDBulkDevice *psDevice);

```

作用：初始化 Bulk 设备协议，本函数在 USBDBulkInit 中已经调用。

参数：ulIndex，USB 模块代码，固定值：USB_BASE0。psDevice，Bulk 设备类。

返回：指向配置后的 tUSBDBulkDevice。

```

unsigned long USBDBulkPacketWrite(void *pvInstance,
                                  unsigned char *pcData,
                                  unsigned long ullength,
                                  tBoolean bLast);

```

作用：通过 Bulk 传输发送一个包数据，底层驱动，在 Buffer 中使用。

参数：pvInstance，tUSBDBulkDevice 设备指针。pcData，待写入的数据指针。ullength，待写入数据的长度。bLast，是否传输结束包。

返回：成功发送长度，可能与 ullength 长度不一样。

```

unsigned long USBDBulkPacketRead(void *pvInstance,
                                 unsigned char *pcData,
                                 unsigned long ullength,
                                 tBoolean bLast);

```

作用：通过 Bulk 传输接收一个包数据，底层驱动，在 Buffer 中使用。

参数：pvInstance，tUSBDBulkDevice 设备指针。pcData，读出数据指针。ullength，读出数据的长度。bLast，是否是束包。

返回：成功接收长度，可能与 ullength 长度不一样。

```

unsigned long USBDBulkTxPacketAvailable(void *pvInstance);
作用：获取可用发送数据长度。
参数：pvInstance, tUSBDBulkDevice 设备指针。
返回：发送包大小，用发送数据长度。
unsigned long USBDBulkRxPacketAvailable(void *pvInstance);
作用：获取接收数据长度。
参数：pvInstance, tUSBDBulkDevice 设备指针。
返回：可用接收数据个数，可读取的有效数据。
void USBDBulkTerm(void *pvInstance);
作用：结束 Bulk 设备。
参数：pvInstance, 指向 tUSBDBulkDevice。
返回：无。
void *USBDBulkSetRxCBData(void *pvInstance, void *pvCBData);
作用：改变接收回调函数的第一个参数。
参数：pvInstance, 指向 tUSBDBulkDevice。pvCBData, 用于替换的参数
返回：旧参数指针。
void *USBDBulkSetTxCBData(void *pvInstance, void *pvCBData);
作用：改变发送回调函数的第一个参数。
参数：pvInstance, 指向 tUSBDBulkDevice。pvCBData, 用于替换的参数
返回：旧参数指针。
void USBDBulkPowerStatusSet(void *pvInstance, unsigned char ucPower);
作用：修改电源属性、状态。
参数：pvInstance, 指向 tUSBDBulkDevice。ucPower, 电源属性。
返回：无。
tBoolean USBDBulkRemoteWakeupRequest(void *pvInstance);
作用：唤醒请求。
参数：pvInstance, 指向 tUSBDBulkDevice。
返回：无。

```

在这些函数中 USBDBulkInit 和 USBDBulkPacketWrite、USBDBulkPacketRead、USBDBulkTxPacketAvailable、USBDBulkRxPacketAvailable 函数最重要并且使用最多，USBDBulkInit 第一次使用 Bulk 设备时，用于初始化 Bulk 设备的配置与控制。USBDBulkPacketRead 、 USBDBulkPacketWrite 、 USBDBulkTxPacketAvailable 、 USBDBulkRxPacketAvailable 为 Bulk 传输数据的底层驱动函数用于驱动 Buffer。

usblib.h 中定义为 11 个 Buffer 操作函数，用于数据发送、接收、以及回调其它函数，下面介绍 11 个 Buffer 操作函数：

```

const tUSBBuffer *USBBufferInit(const tUSBBuffer *psBuffer);
void USBBufferInfoGet(const tUSBBuffer *psBuffer,
                      tUSBRingBufObject *psRingBuf);
unsigned long USBBufferWrite(const tUSBBuffer *psBuffer,
                             const unsigned char *pucData,
                             unsigned long ulLength);
void USBBufferDataWritten(const tUSBBuffer *psBuffer,
                          unsigned long ulLength);

```

```

void USBBufferDataRemoved(const tUSBBuffer *psBuffer,
                          unsigned long ulLength);
void USBBufferFlush(const tUSBBuffer *psBuffer);

unsigned long USBBufferRead(const tUSBBuffer *psBuffer,
                           unsigned char *pucData,
                           unsigned long ulLength);
unsigned long USBBufferDataAvailable(const tUSBBuffer *psBuffer);
unsigned long USBBufferSpaceAvailable(const tUSBBuffer *psBuffer);
void *USBBufferCallbackDataSet(tUSBBuffer *psBuffer, void *pvCBData);

unsigned long USBBufferEventCallback(void *pvCBData,
                                     unsigned long ulEvent,
                                     unsigned long ulMsgValue,
                                     void *pvMsgData);

const tUSBBuffer *USBBufferInit(const tUSBBuffer *psBuffer);
作用：初始化 Buffer，把它加入到当前设备中。首次使用 Buffer 必须使用此函数。
参数：psBuffer，待初始化的 Buffer。
返回：指向配置后的 Buffer。
void USBBufferInfoGet(const tUSBBuffer *psBuffer,
                     tUSBRingBufObject *psRingBuf);
作用：获取 Buffer 信息。psRingBuf 与 psBuffer 建立关系。
参数：psBuffer，操作的目标 Buffer。psRingBuf，申明一个 tUSBRingBufObject 变量。
返回：无。
unsigned long USBBufferWrite(const tUSBBuffer *psBuffer,
                             const unsigned char *pucData,
                             unsigned long ulLength);

作用：写入一组数据。直接写入。
参数：psBuffer，目标 Buffer。pucData，待写入数据指针。ulLength，写入长度。
返回：写入的数据长度。
void USBBufferDataWritten(const tUSBBuffer *psBuffer,
                          unsigned long ulLength);
作用：写入一组数据。使用前要调用 USBBufferInfoGet。
参数：psBuffer，目标 Buffer。ulLength，写入长度。
返回：写入的数据长度。
void USBBufferDataRemoved(const tUSBBuffer *psBuffer,
                          unsigned long ulLength);

作用：从 Buffer 中移出数据。
参数：psBuffer，目标 Buffer。ulLength，移出数据个数。
返回：无。
void USBBufferFlush(const tUSBBuffer *psBuffer);
作用：清除 Buffer 中数据。
参数：psBuffer，目标 Buffer。
返回：无。

```

```

unsigned long USBBufferRead(const tUSBBuffer *psBuffer,
                           unsigned char *pucData,
                           unsigned long ulLength);

```

作用：读取数据。

参数：psBuffer，目标 Buffer。pucData，数据存放指针。ulLength，读取个数。

返回：读取数据个数。

```

unsigned long USBBufferDataAvailable(const tUSBBuffer *psBuffer);

```

作用：可读取数据个数。

参数：psBuffer，目标 Buffer。

返回：可读取数据个数。

```

unsigned long USBBufferSpaceAvailable(const tUSBBuffer *psBuffer);

```

作用：可用数据空间大小。

参数：psBuffer，目标 Buffer。

返回：数据空间大小。

```

void *USBBufferCallbackDataSet(tUSBBuffer *psBuffer, void *pvCBData);

```

作用：修改设备 Buffer。

参数：psBuffer，用于替换的新 Buffer 指针。

返回：旧 Buffer 指针。

```

unsigned long USBBufferEventCallback(void *pvCBData,
                                     unsigned long ulEvent,
                                     unsigned long ulMsgValue,
                                     void *pvMsgData);

```

作用：Buffer 事件调用函数。

参数：pvCBData，设备指针。ulEvent，Buffer 事务。ulMsgValue，数据长度。pvMsgData 数据指针。

返回：函数是否成功执行。

以上是 11 个 Buffer 处理函数，用于 Buffer 数据接收、发送及处理。在 Bulk 传输中大量使用。

7.4 Bulk 设备开发

Bulk 设备开发只需要 4 步就能完成。如图 2 所示，Bulk 设备配置（主要是字符串描述符）、callback 函数编写、USB 处理器初始化、数据处理。

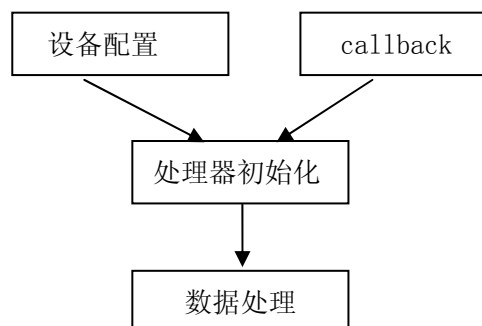


图 2

第一步：Bulk 设备配置（主要是字符串描述符），按字符串描述符标准完成串描述符配置，进而完成 Bulk 设备配置。


```

#include "inc/hw_ints.h"
#include "inc/hw_memmap.h"
#include "inc/hw_types.h"
#include "inc/hw_sysctl.h"
#include "inc/hw_udma.h"
#include "inc/hw_gpio.h"
#include "driverlib/gpio.h"
#include "driverlib/interrupt.h"
#include "driverlib/sysctl.h"
#include "driverlib/usb.h"
#include "usblib/usblib.h"
#include "usblib/usb-ids.h"
#include "usblib/device/usbdevice.h"
#include "usblib/device/usdbulk.h"
#include "uartstdio.h"
#include "ustdlib.h"
//每次传输数据大小
#define BULK_BUFFER_SIZE 256
unsigned long RxHandler(void *pvCBData, unsigned long ulEvent,
                        unsigned long ulMsgValue, void *pvMsgData);
unsigned long TxHandler(void *pvlCBData, unsigned long ulEvent,
                        unsigned long ulMsgValue, void *pvMsgData);
unsigned long EchoNewDataToHost(tUSBDBulkDevice *psDevice, unsigned char *pcData,
                                unsigned long ulNumBytes);
#define COMMAND_PACKET_RECEIVED 0x00000001
#define COMMAND_STATUS_UPDATE 0x00000002
volatile unsigned long g_ulFlags = 0;
char *g_pcStatus;
static volatile tBoolean g_bUSBConfigured = false;
volatile unsigned long g_ulTxCount = 0;
volatile unsigned long g_ulRxCount = 0;
const tUSBBuffer g_sRxBuffer;
const tUSBBuffer g_sTxBuffer;
//*****
// 设备语言描述符.
//*****
const unsigned char g_pLangDescriptor[] =
{
    4,
    USB_DTYPE_STRING,
    USBShort(USB_LANG_EN_US)
};
//*****
// 制造商 字符串 描述符

```

```

//*****
const unsigned char g_pManufacturerString[] =
{
    (17 + 1) * 2,
    USB_DTYPE_STRING,
    'T', 0, 'e', 0, 'x', 0, 'a', 0, 's', 0, ' ', 0, 'I', 0, 'n', 0, 's', 0,
    't', 0, 'r', 0, 'u', 0, 'm', 0, 'e', 0, 'n', 0, 't', 0, 's', 0,
};
//*****
//产品 字符串 描述符
//*****
const unsigned char g_pProductString[] =
{
    (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
};
//*****
// 产品 序列号 描述符
//*****
const unsigned char g_pSerialNumberString[] =
{
    (8 + 1) * 2,
    USB_DTYPE_STRING,
    '1', 0, '2', 0, '3', 0, '4', 0, '5', 0, '6', 0, '7', 0, '8', 0
};
//*****
// 设备接口字符串描述符
//*****
const unsigned char g_pDataInterfaceString[] =
{
    (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
};
//*****
// 设备配置字符串描述符
//*****
const unsigned char g_pConfigString[] =
{

```

```

        (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
    };
    //*****
    // 字符串描述符集合
    //*****
    const unsigned char * const g_pStringDescriptors[] =
    {
        g_pLangDescriptor,
        g_pManufacturerString,
        g_pProductString,
        g_pSerialNumberString,
        g_pDataInterfaceString,
        g_pConfigString
    };
    #define NUM_STRING_DESCRIPTOR (sizeof(g_pStringDescriptors) / \
                                   sizeof(unsigned char *))
    //*****
    // 定义 Bulk 设备实例
    //*****
    tBulkInstance g_sBulkInstance;
    //*****
    // 定义 Bulk 设备
    //*****
    const tUSBDBulkDevice g_sBulkDevice =
    {
        0x1234,
        USB_PID_BULK,
        500,
        USB_CONF_ATTR_SELF_PWR,
        USBBufferEventCallback,
        (void *)&g_sRxBuffer,
        USBBufferEventCallback,
        (void *)&g_sTxBuffer,
        g_pStringDescriptors,
        NUM_STRING_DESCRIPTOR,
        &g_sBulkInstance
    };
    //*****
    // 定义 Buffer
    //*****

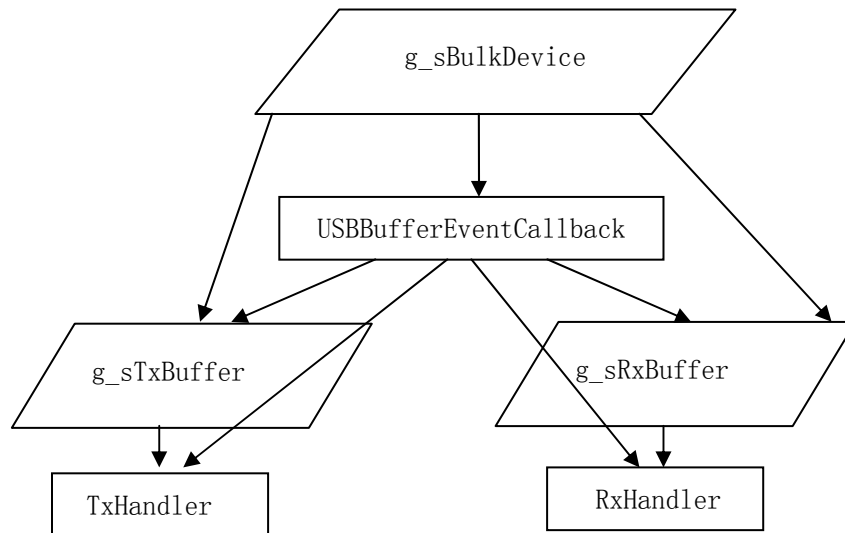
```

```

unsigned char g_pucUSBRxBuffer[BULK_BUFFER_SIZE];
unsigned char g_pucUSBTxBuffer[BULK_BUFFER_SIZE];
unsigned char g_pucTxBufferWorkspace[USB_BUFFER_WORKSPACE_SIZE];
unsigned char g_pucRxBufferWorkspace[USB_BUFFER_WORKSPACE_SIZE];
const tUSBBuffer g_sRxBuffer =
{
    false,                                // This is a receive buffer.
    RxHandler,                            // pfnCallback
    (void *)&g_sBulkDevice,              // Callback data is our device pointer.
    USBDBulkPacketRead,                   // pfnTransfer
    USBDBulkRxPacketAvailable,            // pfnAvailable
    (void *)&g_sBulkDevice,              // pvHandle
    g_pucUSBRxBuffer,                     // pcBuffer
    BULK_BUFFER_SIZE,                     // ulBufferSize
    g_pucRxBufferWorkspace                 // pvWorkspace
};
const tUSBBuffer g_sTxBuffer =
{
    true,                                // This is a transmit buffer.
    TxHandler,                            // pfnCallback
    (void *)&g_sBulkDevice,              // Callback data is our device pointer.
    USBDBulkPacketWrite,                   // pfnTransfer
    USBDBulkTxPacketAvailable,             // pfnAvailable
    (void *)&g_sBulkDevice,              // pvHandle
    g_pucUSBTxBuffer,                     // pcBuffer
    BULK_BUFFER_SIZE,                     // ulBufferSize
    g_pucTxBufferWorkspace                 // pvWorkspace
};

```

tUSBDBulkDevice g_sBulkDevice、tUSBBuffer g_sTxBuffer、tUSBBuffer g_sRxBuffer 是管理 Bulk 设备的主要结构体，它们三者关系如下图：



tUSBDBulkDevice g_sBulkDevice 主要进行上层协议与通信管理控制，通过 USBBufferEventCallback 函数处理 Buffer 数据：数据发送、接收、控制事件，通过 g_sTxBuffer 中的 USBDBulkPacketWrite 和 USBDBulkTxPacketAvailable 实现底层数据发送，并通过 TxHandler 返回处理结果；通过 g_sRxBuffer 中的 USBDBulkPacketRead 和 USBDBulkRxPacketAvailable 实现底层数据接收，并通过 RxHandler 返回处理结果。在 g_sBulkDevice 层可以直接使用 Buffer 函数对 Buffer 层操作，并通过 TxHandler 和 RxHandler 返回处理结果。所有处理过程中的数据都保存在 Buffer 层的 g_pucUSBTxBuffer 或者 g_pucUSBRxBuffer，隶属于 g_sBulkDevice 的一部分。注意：USBDBulkPacketWrite、USBDBulkTxPacketAvailable、USBDBulkPacketRead、USBDBulkRxPacketAvailable 由 Bulk 设备类 API 定义，可以直接使用。USBBufferEventCallback 为 Buffer 层定义的标准 API，用于处理、调用 USBDBulkPacketWrite、USBDBulkTxPacketAvailable、USBDBulkPacketRead、USBDBulkRxPacketAvailable、TxHandler 和 RxHandler 完成 g_sBulkDevice 层发送的数据接收与发送命令。

第二步：完成 Callback 函数。Callback 函数用于处理输出端点、输入端点数据事务。Bulk 设备接收回调函数包含以下事务：USB_EVENT_CONNECTED、USB_EVENT_DISCONNECTED、USB_EVENT_RX_AVAILABLE、USB_EVENT_SUSPEND、USB_EVENT_RESUME、USB_EVENT_ERROR。Bulk 设备发送回调函数包含了以下事务：USB_EVENT_TX_COMPLETE。如下表：

名称	属性	说明
USB_EVENT_CONNECTED	接收	USB 设备已经连接到主机
USB_EVENT_DISCONNECTED	接收	USB 设备已经与主机断开
USB_EVENT_RX_AVAILABLE	接收	有接受数据
USB_EVENT_SUSPEND	接收	挂起
USB_EVENT_RESUME	接收	唤醒
USB_EVENT_ERROR	接收	错误
USB_EVENT_TX_COMPLETE	发送	发送完成

表 2. Bulk 事务

根据以上事务编写 Callback 函数：

```

//*****
//USB Bulk 设备类返回事件处理函数（callback）。

```

```

//*****
unsigned long TxHandler(void *pvCBData, unsigned long ulEvent, unsigned long ulMsgValue,
                        void *pvMsgData)
{
    //发送完成事件
    if(ulEvent == USB_EVENT_TX_COMPLETE)
    {
        g_ulTxCount += ulMsgValue;
    }
    return(0);
}

unsigned long RxHandler(void *pvCBData, unsigned long ulEvent,
                        unsigned long ulMsgValue, void *pvMsgData)
{
    // 接收事件
    switch(ulEvent)
    {
        //连接成功
        case USB_EVENT_CONNECTED:
        {
            GPIOPinWrite(GPIO_PORTF_BASE, 0x40, 0x40);
            g_bUSBConfigured = true;
            g_pcStatus = "Host connected.";
            g_ulFlags |= COMMAND_STATUS_UPDATE;
            // Flush our buffers.
            USBBufferFlush(&g_sTxBuffer);
            USBBufferFlush(&g_sRxBuffer);

            break;
        }
        // 断开连接.
        case USB_EVENT_DISCONNECTED:
        {
            GPIOPinWrite(GPIO_PORTF_BASE, 0x40, 0x00);
            g_bUSBConfigured = false;
            g_pcStatus = "Host disconnected.";
            g_ulFlags |= COMMAND_STATUS_UPDATE;
            break;
        }
        // 有可能数据接收.
        case USB_EVENT_RX_AVAILABLE:
        {
            tUSBDBulkDevice *psDevice;
            psDevice = (tUSBDBulkDevice *)pvCBData;

```

```

        // 把接收到的数据发送回去。
        return(EchoNewDataToHost(psDevice, pvMsgData, ulMsgValue));
    }
    //挂起, 唤醒
    case USB_EVENT_SUSPEND:
    case USB_EVENT_RESUME:break;
    default:break;
}
return(0);
}

//*****
//EchoNewDataToHost 函数
//*****
unsigned long EchoNewDataToHost(tUSBDBulkDevice *psDevice, unsigned char *pcData,
                                unsigned long ulNumBytes)
{
    unsigned long ulLoop, ulSpace, ulCount;
    unsigned long ulReadIndex;
    unsigned long ulWriteIndex;
    tUSBRingBufObject sTxRing;
    // 获取 Buffer 信息.
    USBBufferInfoGet(&g_sTxBuffer, &sTxRing);
    // 有多少可能空间
    ulSpace = USBBufferSpaceAvailable(&g_sTxBuffer);
    // 改变数据
    ulLoop = (ulSpace < ulNumBytes) ? ulSpace : ulNumBytes;
    ulCount = ulLoop;
    // 更新接收到的数据个数
    g_ulRxCount += ulNumBytes;
    ulReadIndex = (unsigned long)(pcData - g_pucUSBRxBuffer);
    ulWriteIndex = sTxRing.ulWriteIndex;
    while(ulLoop)
    {
        //更新接收的数据
        if((g_pucUSBRxBuffer[ulReadIndex] >= 'a') &&
            (g_pucUSBRxBuffer[ulReadIndex] <= 'z'))
        {
            //转换
            g_pucUSBTxBuffer[ulWriteIndex] =
                (g_pucUSBRxBuffer[ulReadIndex] - 'a') + 'A';
        }
        else
        {
            //转换

```

```

        if((g_pucUSBRxBuffer[ulReadIndex] >= 'A') &&
            (g_pucUSBRxBuffer[ulReadIndex] <= 'Z'))
        {
            //转换
            g_pucUSBTxBuffer[ulWriteIndex] =
                (g_pucUSBRxBuffer[ulReadIndex] - 'Z') + 'z';
        }
        else
        {
            //转换
            g_pucUSBTxBuffer[ulWriteIndex] = g_pucUSBRxBuffer[ulReadIndex];
        }
    }
    // 更新指针
    ulWriteIndex++;
    ulWriteIndex = (ulWriteIndex == BULK_BUFFER_SIZE) ? 0 : ulWriteIndex;

    ulReadIndex++;
    ulReadIndex = (ulReadIndex == BULK_BUFFER_SIZE) ? 0 : ulReadIndex;
    ulLoop--;
}
// 发送数据
USBBufferDataWritten(&g_sTxBuffer, ulCount);
return(ulCount);
}

```

第三步：系统初始化，配置内核电压、系统主频、使能端口、LED 控制等，本例中使用 4 个 LED 进行指示数据传输。在这个例子中，Bulk 传输接收的数据发送给主机。原理图如图 3 所示：

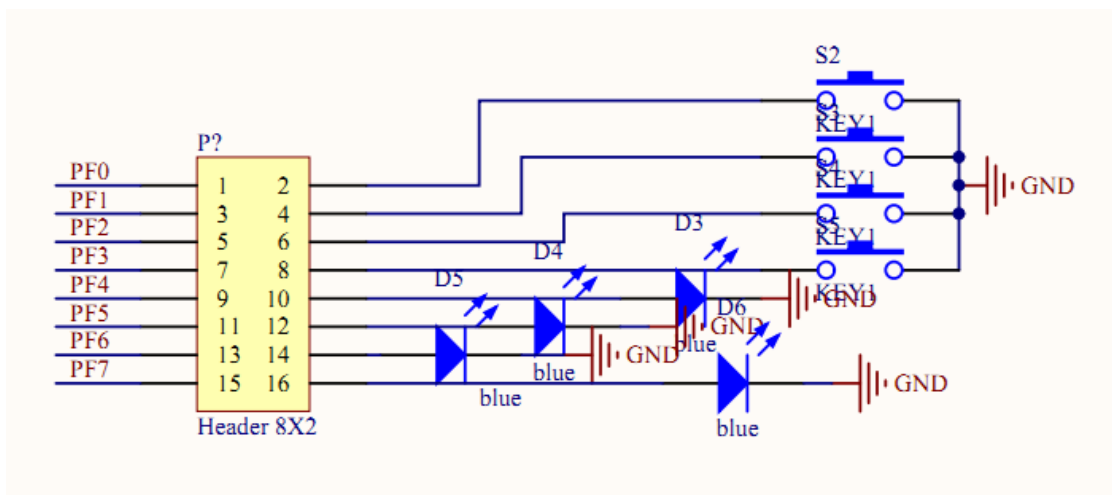


图 3

系统初始化：

```

unsigned long ulTxCount = 0;
unsigned long ulRxCount = 0;

```



```
// char pcBuffer[16];
//设置内核电压、主频 50Mhz
SysCtlLDOSet(SYSCTL_LDO_2_75V);
SysCtlClockSet(SYSCTL_XTAL_8MHZ | SYSCTL_SYSDIV_4 | SYSCTL_USE_PLL | SYSCTL_OSC_MAIN );
```

```
SysCtlPeripheralEnable(SYSCTL_PERIPH_GPIOF);
GPIONPinTypeGPIOOutput(GPIO_PORTF_BASE, 0xf0);
GPIONPinTypeGPIOInput(GPIO_PORTF_BASE, 0x0f);
HWREG(GPIO_PORTF_BASE+GPIO_O_PUR) |= 0x0f;
```

```
// 初始化发送与接收 Buffer.
USBBufferInit((tUSBBuffer *)&g_sTxBuffer);
USBBufferInit((tUSBBuffer *)&g_sRxBuffer);
// 初始化 Bulk 设备
USBDBulkInit(0, (tUSBDBulkDevice *)&g_sBulkDevice);
```

第四步：数据处理。主要使用 11 个 Buffer 处理函数，用于 Buffer 数据接收、发送及处理。在 Bulk 传输中大量使用。

```
while(1)
{
    //等待连接结束.
    while((g_ulFlags & FLAG_CONNECTED) == 0)
    {
    }

    //初始化 Buffer
    g_sBuffer.pucFill = g_sBuffer.pucBuffer;
    g_sBuffer.pucPlay = g_sBuffer.pucBuffer;
    g_sBuffer.ulFlags = 0;

    //从 Bulk 设备类中获取数据
    if(USBBulkBufferOut(g_pvBulkDevice,
                        (unsigned char *)g_sBuffer.pucFill,
                        BULK_PACKET_SIZE, USBBufferCallback) == 0)
    {
        //标记数据放入 buffer 中.
        g_sBuffer.ulFlags |= SBUFFER_FLAGS_FILLING;
    }

    //设备连接到主机.
    while(g_ulFlags & FLAG_CONNECTED)
    {
        // 检查音量是否有改变.
        if(g_ulFlags & FLAG_VOLUME_UPDATE)
        {
            // 清除更新音量标志.
            g_ulFlags &= ~FLAG_VOLUME_UPDATE;

            // 修改音量，自行添加代码. 在此以 LED 灯做指示。
```

```

        //UpdateVolume();
        GPIOPinWrite(GPIO_PORTF_BASE, 0x40, ~GPIOPinRead(GPIO_PORTF_BASE, 0x40));
    }
    //是否静音
    if(g_ulFlags & FLAG_MUTE_UPDATE)
    {
        //修改静音状态, 自行添加函数. 在此以 LED 灯做指示。
        //UpdateMute();
        if(g_ulFlags & FLAG_MUTED)
            GPIOPinWrite(GPIO_PORTF_BASE, 0x20, 0x20);
        else
            GPIOPinWrite(GPIO_PORTF_BASE, 0x20, 0x00);
        // 清除静音标志
        g_ulFlags &= ~FLAG_MUTE_UPDATE;
    }
}

//*****
//USBBulkBufferOut 的 Callback 入口参数
//*****
void USBBufferCallback(void *pvBuffer, unsigned long ulParam, unsigned long ulEvent)
{
    //数据处理, 自行加入代码。
    // Your Codes .....
    //再一次获取数据。
    USBBulkBufferOut(g_pvBulkDevice, (unsigned char *)g_sBuffer.pucFill,
        BULK_PACKET_SIZE, USBBufferCallback);
}

```

使用上面四步就完成 Bulk 设备开发。Bulk 设备开发时要加入两个 lib 库函数：usb.lib 和 DriverLib.lib，在启动代码中加入 USB0DeviceIntHandler 中断服务函数。以上 Bulk 设备开发完成，在 Win xp 下运行效果如下图所示：



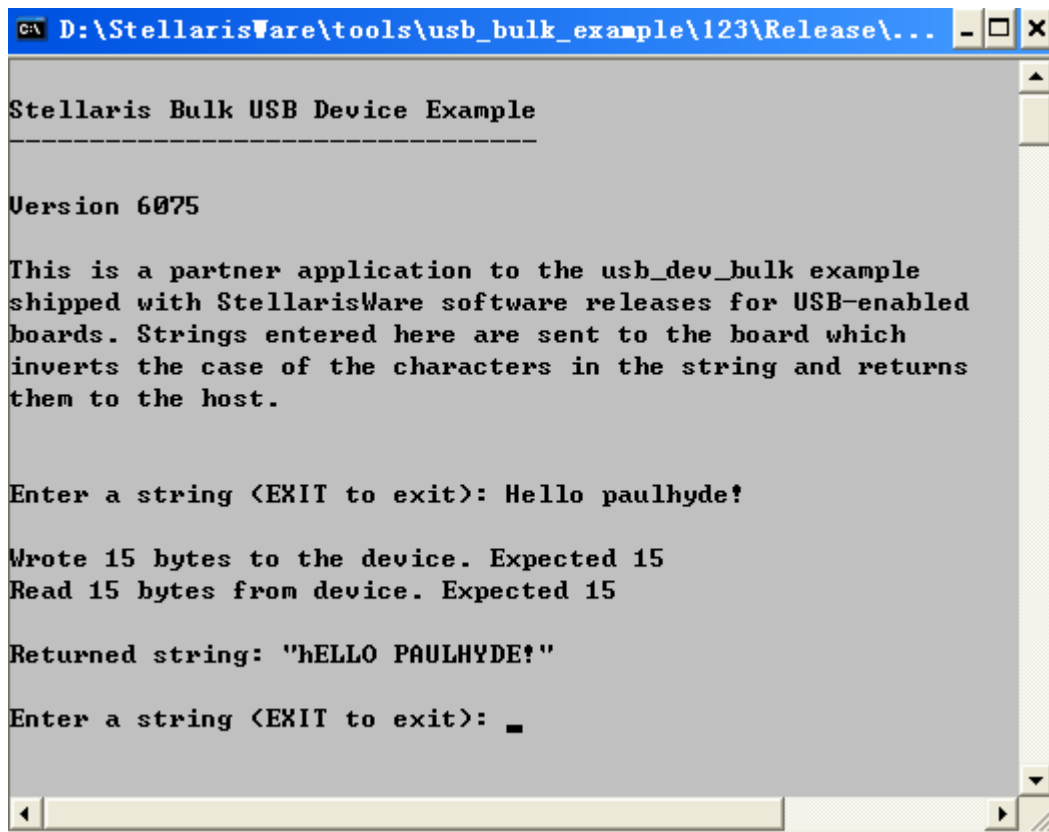


驱动安装完成

在枚举过程中可以看出，在电脑右下角可以看到“Generic Bulk Device”字样，标示正在进行枚举，并手动安装驱动。枚举成功后，在“设备管理器”的“Stellaris Bulk Device”中看到“Generic Bulk Device”设备，如下图。现在 Bulk 设备可以正式使用。



Bulk 设备要配合上位机使用，上位机发送字符串通过 USB Bulk 设备转换后发送给主机。运行图如下：



```
C:\D:\StellarisWare\tools\usb_bulk_example\123\Release\...
Stellaris Bulk USB Device Example
-----
Version 6075

This is a partner application to the usb_dev_bulk example
shipped with StellarisWare software releases for USB-enabled
boards. Strings entered here are sent to the board which
inverts the case of the characters in the string and returns
them to the host.

Enter a string <EXIT to exit>: Hello paulhyde!

Wrote 15 bytes to the device. Expected 15
Read 15 bytes from device. Expected 15

Returned string: "hELLO PAULHYDE!"

Enter a string <EXIT to exit>: _
```

上位机源码如下:

```
#include <windows.h>
#include <strsafe.h>
#include <initguid.h>
#include "lmsbdl1.h"
#include "luminary_guids.h"
//*****
// Buffer size definitions.
//*****
#define MAX_STRING_LEN 256
#define MAX_ENTRY_LEN 256
#define USB_BUFFER_LEN 1216
//*****
// The build version number
//*****
#define BLDVER "6075"
//*****
// The number of bytes we read and write per transaction if in echo mode.
//*****
#define ECHO_PACKET_SIZE 1216
//*****
// Buffer into which error messages are written.
//*****
TCHAR g_pcErrorString[MAX_STRING_LEN];
```

```

//*****
// The number of bytes transfered in the last measurement interval.
//*****
ULONG g_ulByteCount = 0;
//*****
// The total number of packets transfered.
//*****
ULONG g_ulPacketCount = 0;
//*****
LPTSTR GetSystemErrorString(DWORD dwError)
{
    DWORD dwRetcode;
    // Ask Windows for the error message description.
    dwRetcode = FormatMessage(FORMAT_MESSAGE_FROM_SYSTEM, "%0", dwError, 0,
                             g_pcErrorString, MAX_STRING_LEN, NULL);

    if(dwRetcode == 0)
    {
        return((LPTSTR)L"Unknown");
    }
    else
    {
        // Remove the trailing "\n\r" if present.
        if(dwRetcode >= 2)
        {
            if(g_pcErrorString[dwRetcode - 2] == '\r')
            {
                g_pcErrorString[dwRetcode - 2] = '\0';
            }
        }
        return(g_pcErrorString);
    }
}
//*****
// Print the throughput in terms of Kbps once per second.
//*****
void UpdateThroughput(void)
{
    static ULONG ulStartTime = 0;
    static ULONG ulLast = 0;
    ULONG ulNow;
    ULONG ulElapsed;
    SYSTEMTIME sSysTime;
    // Get the current system time.
    GetSystemTime(&sSysTime);

```

```

        ulNow = (((sSysTime.wHour * 60) +
                    sSysTime.wMinute) * 60) +
                    sSysTime.wSecond) * 1000) + sSysTime.wMilliseconds;
// If this is the first call, set the start time.
if(ulStartTime == 0)
{
    ulStartTime = ulNow;
    ulLast = ulNow;
    return;
}
// How much time has elapsed since the last measurement?
ulElapsed = (ulNow > ulStartTime) ? (ulNow - ulStartTime) : (ulStartTime - ulNow);
// We dump a new measurement every second.
if(ulElapsed > 1000)
{
    printf("\r%6dKbps   Packets:  %10d   ", ((g_ulByteCount * 8) / ulElapsed),
g_ulPacketCount);
    g_ulByteCount = 0;
    ulStartTime = ulNow;
}
}

//*****
// The main application entry function.
//*****
int main(int argc, char *argv[])
{
    BOOL bResult;
    BOOL bDriverInstalled;
    BOOL bEcho;
    char szBuffer[USB_BUFFER_LEN];
    ULONG ulWritten;
    ULONG ulRead;
    ULONG ulLength;
    DWORD dwError;
    LMUSB_HANDLE hUSB;

    // Are we operating in echo mode or not? The "-e" parameter tells the
    // app to echo everything it receives back to the device unchanged.
    bEcho = ((argc > 1) && (argv[1][1] == 'e')) ? TRUE : FALSE;
    // Print a cheerful welcome.
    printf("\nStellaris Bulk USB Device Example\n");
    printf( "-----\n\n");
    printf("Version %s\n\n", BLDVER);
    if(!bEcho)
    {

```

```

    printf("This is a partner application to the usb_dev_bulk example\n");
    printf("shipped with StellarisWare software releases for USB-enabled\n");
    printf("boards. Strings entered here are sent to the board which\n");
    printf("inverts the case of the characters in the string and returns\n");
    printf("them to the host.\n\n");
}
else
{
    printf("If run with the \"-e\" command line switch, this application\n");
    printf("echoes all data received on the bulk IN endpoint to the bulk\n");
    printf("OUT endpoint. This feature may be helpful during development\n");
    printf("and debug of your own USB devices. Note that this will not\n");
    printf("do anything exciting if run with the usb_dev_bulk example\n");
    printf("device attached since it expects the host to initiate transfers.\n\n");
}

// Find our USB device and prepare it for communication.
hUSB = InitializeDevice(BULK_VID, BULK_PID,
                       (LPGUID)&(GUID_DEVINTERFACE_STELLARIS_BULK),
                       &bDriverInstalled);

if(hUSB)
{
    // Are we operating in echo mode or not? The "-e" parameter tells the
    // app to echo everything it receives back to the device unchanged.
    if(bEcho)
    {
        printf("Running in echo mode. Press Ctrl+C to exit.\n\n"
               "Throughput:      0Kbps Packets:      0");
        while(1)
        {
            // Read a block of data from the device.
            dwError = ReadUSBPacket(hUSB, szBuffer, USB_BUFFER_LEN, &ulRead,
                                    INFINITE, NULL);

            if(dwError != ERROR_SUCCESS)
            {
                // We failed to read from the device.
                printf("\n\nError %d (%S) reading from bulk IN pipe.\n", dwError,
                      GetSystemErrorString(dwError));
                break;
            }
            else
            {
                // Update our byte and packet counters.
                g_ulByteCount += ulRead;
                g_ulPacketCount++;
            }
        }
    }
}

```

```

        // Write the data back out to the device.
        bResult = WriteUSBPacket(hUSB, szBuffer, ulRead, &ulWritten);
        if(!bResult)
        {
            // We failed to write the data for some reason.
            dwError = GetLastError();
            printf("\n\nError %d (%S) writing to bulk OUT pipe.\n", dwError,
                GetSystemErrorString(dwError));
            break;
        }
        // Display the throughput.
        UpdateThroughput();
    }
}
else
{
    // We are running in normal mode. Keep sending and receiving
    // strings until the user indicates that it is time to exit.
    while(1)
    {
        // The device was found and successfully configured. Now get a string from
        // the user...
        do
        {
            printf("\nEnter a string (EXIT to exit): ");
            fgets(szBuffer, MAX_ENTRY_LEN, stdin);
            printf("\n");
            // How many characters were entered (including the trailing '\n')?
            ulLength = (ULONG)strlen(szBuffer);
            if(ulLength <= 1)
            {
                printf("\nPlease enter some text.\n");
                ulLength = 0;
            }
        }
        else
        {
            // Get rid of the trailing '\n' if there is one there.
            if(szBuffer[ulLength - 1] == '\n')
            {
                szBuffer[ulLength - 1] = '\0';
                ulLength--;
            }
        }
    }
}

```



```

    }
    while(ulLength == 0);
    if(!(strcmp("EXIT", szBuffer)))
    {
        printf("Exiting on user request.\n");
        break;
    }
    // Write the user's string to the device.
    bResult = WriteUSBPacket(hUSB, szBuffer, ulLength, &ulWritten);
    if(!bResult)
    {
        dwError = GetLastError();
        printf("Error %d (%S) writing to bulk OUT pipe.\n", dwError,
            GetSystemErrorString(dwError));
    }
    else
    {
        // We wrote data successfully so now read it back.
        printf("Wrote %d bytes to the device. Expected %d\n",
            ulWritten, ulLength);
        // We expect the same number of bytes as we just sent.
        dwError = ReadUSBPacket(hUSB, szBuffer, ulWritten, &ulRead,
            INFINITE, NULL);
        if(dwError != ERROR_SUCCESS)
        {
            // We failed to read from the device.
            printf("Error %d (%S) reading from bulk IN pipe.\n", dwError,
                GetSystemErrorString(dwError));
        }
        else
        {
            szBuffer[ulRead] = '\0';
            printf("Read %d bytes from device. Expected %d\n",
                ulRead, ulWritten);
            printf("\nReturned string: \"%s\"\n", szBuffer);
        }
    }
}
}
else
{
    // An error was reported while trying to connect to the device.
    dwError = GetLastError();

```

```

    printf("\nUnable to initialize the Stellaris Bulk USB Device.\n");
    printf("Error code is %d (%S)\n\n", dwError, GetSystemErrorString(dwError));
    printf("Please make sure you have a Stellaris USB-enabled evaluation\n");
    printf("or development kit running the usb_dev_bulk example\n");
    printf("application connected to this system via the \"USB OTG\" or\n");
    printf("\"USB DEVICE\" connectors. Once the device is connected, run\n");
    printf("this application again.\n\n");
    printf("\nPress \"Enter\" to exit: ");
    fgets(szBuffer, MAX_STRING_LEN, stdin);
    printf("\n");
    return(2);
}

TerminateDevice(hUSB);
return(0);
}

```

Bulk 设备开发源码如下：

```

#include "inc/hw_ints.h"
#include "inc/hw_memmap.h"
#include "inc/hw_types.h"
#include "inc/hw_sysctl.h"
#include "inc/hw_udma.h"
#include "inc/hw_gpio.h"
#include "driverlib/gpio.h"
#include "driverlib/interrupt.h"
#include "driverlib/sysctl.h"
#include "driverlib/usb.h"
#include "usblib/usb.h"
#include "usblib/usb-ids.h"
#include "usblib/device/usbdevice.h"
#include "usblib/device/usdbulk.h"
#include "uartstdio.h"
#include "ustdlib.h"

//每次传输数据大小
#define BULK_BUFFER_SIZE 256

unsigned long RxHandler(void *pvCBData, unsigned long ulEvent,
                        unsigned long ulMsgValue, void *pvMsgData);

unsigned long TxHandler(void *pvlCBData, unsigned long ulEvent,
                        unsigned long ulMsgValue, void *pvMsgData);

unsigned long EchoNewDataToHost(tUSDBulkDevice *psDevice, unsigned char *pcData,
                               unsigned long ulNumBytes);

#define COMMAND_PACKET_RECEIVED 0x00000001
#define COMMAND_STATUS_UPDATE 0x00000002

volatile unsigned long g_ulFlags = 0;

char *g_pcStatus;

```

```

static volatile tBoolean g_bUSBConfigured = false;
volatile unsigned long g_ulTxCount = 0;
volatile unsigned long g_ulRxCount = 0;
const tUSBBuffer g_sRxBuffer;
const tUSBBuffer g_sTxBuffer;
//*****
// 设备语言描述符.
//*****
const unsigned char g_pLangDescriptor[] =
{
    4,
    USB_DTYPE_STRING,
    USBShort(USB_LANG_EN_US)
};
//*****
// 制造商 字符串 描述符
//*****
const unsigned char g_pManufacturerString[] =
{
    (17 + 1) * 2,
    USB_DTYPE_STRING,
    'T', 0, 'e', 0, 'x', 0, 'a', 0, 's', 0, ' ', 0, 'I', 0, 'n', 0, 's', 0,
    't', 0, 'r', 0, 'u', 0, 'm', 0, 'e', 0, 'n', 0, 't', 0, 's', 0,
};
//*****
//产品 字符串 描述符
//*****
const unsigned char g_pProductString[] =
{
    (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,
};
//*****
// 产品 序列号 描述符
//*****
const unsigned char g_pSerialNumberString[] =
{
    (8 + 1) * 2,
    USB_DTYPE_STRING,
    '1', 0, '2', 0, '3', 0, '4', 0, '5', 0, '6', 0, '7', 0, '8', 0,
};

```

```

//*****
// 设备接口字符串描述符
//*****
const unsigned char g_pDataInterfaceString[] =
{
    (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
};
//*****
// 设备配置字符串描述符
//*****
const unsigned char g_pConfigString[] =
{
    (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
};
//*****
// 字符串描述符集合
//*****
const unsigned char * const g_pStringDescriptors[] =
{
    g_pLangDescriptor,
    g_pManufacturerString,
    g_pProductString,
    g_pSerialNumberString,
    g_pDataInterfaceString,
    g_pConfigString
};
#define NUM_STRING_DESCRIPTOR (sizeof(g_pStringDescriptors) / \
                                sizeof(unsigned char *))
//*****
// 定义 Bulk 设备实例
//*****
tBulkInstance g_sBulkInstance;
//*****
// 定义 Bulk 设备
//*****
const tUSBDBulkDevice g_sBulkDevice =

```

```

{
    0x1234,
    USB_PID_BULK,
    500,
    USB_CONF_ATTR_SELF_PWR,
    USBBufferEventCallback,
    (void *)&g_sRxBuffer,
    USBBufferEventCallback,
    (void *)&g_sTxBuffer,
    g_pStringDescriptors,
    NUM_STRING_DESCRIPTOR,
    &g_sBulkInstance
};

//*****
// 定义 Buffer
//*****

unsigned char g_pucUSBRxBuffer[BULK_BUFFER_SIZE];
unsigned char g_pucUSBTxBuffer[BULK_BUFFER_SIZE];
unsigned char g_pucTxBufferWorkspace[USB_BUFFER_WORKSPACE_SIZE];
unsigned char g_pucRxBufferWorkspace[USB_BUFFER_WORKSPACE_SIZE];
const tUSBBuffer g_sRxBuffer =
{
    false,                // This is a receive buffer.
    RxHandler,            // pfnCallback
    (void *)&g_sBulkDevice, // Callback data is our device pointer.
    USBDBulkPacketRead,    // pfnTransfer
    USBDBulkRxPacketAvailable, // pfnAvailable
    (void *)&g_sBulkDevice, // pvHandle
    g_pucUSBRxBuffer,      // pcBuffer
    BULK_BUFFER_SIZE,      // ulBufferSize
    g_pucRxBufferWorkspace // pvWorkspace
};

const tUSBBuffer g_sTxBuffer =
{
    true,                // This is a transmit buffer.
    TxHandler,            // pfnCallback
    (void *)&g_sBulkDevice, // Callback data is our device pointer.
    USBDBulkPacketWrite,   // pfnTransfer
    USBDBulkTxPacketAvailable, // pfnAvailable
    (void *)&g_sBulkDevice, // pvHandle
    g_pucUSBTxBuffer,      // pcBuffer
    BULK_BUFFER_SIZE,      // ulBufferSize
    g_pucTxBufferWorkspace // pvWorkspace
};

```

```

//*****
//USB Bulk 设备类返回事件处理函数（callback）.
//*****
unsigned long TxHandler(void *pvCBData, unsigned long ulEvent, unsigned long ulMsgValue,
                        void *pvMsgData)
{
    //发送完成事件
    if(ulEvent == USB_EVENT_TX_COMPLETE)
    {
        g_ulTxCount += ulMsgValue;
    }
    return(0);
}

unsigned long RxHandler(void *pvCBData, unsigned long ulEvent,
                        unsigned long ulMsgValue, void *pvMsgData)
{
    // 接收事件
    switch(ulEvent)
    {
        //连接成功
        case USB_EVENT_CONNECTED:
        {
            GPIOPinWrite(GPIO_PORTF_BASE, 0x40, 0x40);
            g_bUSBConfigured = true;
            g_pcStatus = "Host connected.";
            g_ulFlags |= COMMAND_STATUS_UPDATE;
            // Flush our buffers.
            USBBufferFlush(&g_sTxBuffer);
            USBBufferFlush(&g_sRxBuffer);

            break;
        }
        // 断开连接.
        case USB_EVENT_DISCONNECTED:
        {
            GPIOPinWrite(GPIO_PORTF_BASE, 0x40, 0x00);
            g_bUSBConfigured = false;
            g_pcStatus = "Host disconnected.";
            g_ulFlags |= COMMAND_STATUS_UPDATE;
            break;
        }
        // 有可能数据接收.
        case USB_EVENT_RX_AVAILABLE:
        {

```

```

        tUSDBulkDevice *psDevice;
        psDevice = (tUSDBulkDevice *)pvCBData;
        // 把接收到的数据发送回去。
        return(EchoNewDataToHost(psDevice, pvMsgData, ulMsgValue));
    }

    //挂起, 唤醒
    case USB_EVENT_SUSPEND:
    case USB_EVENT_RESUME:break;
    default:break;
}

return(0);
}

//*****
//EchoNewDataToHost 函数
//*****
unsigned long EchoNewDataToHost(tUSDBulkDevice *psDevice, unsigned char *pcData,
                                unsigned long ulNumBytes)
{
    unsigned long ulLoop, ulSpace, ulCount;
    unsigned long ulReadIndex;
    unsigned long ulWriteIndex;
    tUSBRingBufObject sTxRing;
    // 获取 Buffer 信息.
    USBBufferInfoGet(&g_sTxBuffer, &sTxRing);
    // 有多少可能空间
    ulSpace = USBBufferSpaceAvailable(&g_sTxBuffer);
    // 改变数据
    ulLoop = (ulSpace < ulNumBytes) ? ulSpace : ulNumBytes;
    ulCount = ulLoop;
    // 更新接收到的数据个数
    g_ulRxCount += ulNumBytes;
    ulReadIndex = (unsigned long)(pcData - g_pucUSBRxBuffer);
    ulWriteIndex = sTxRing.ulWriteIndex;
    while(ulLoop)
    {
        //更新接收的数据
        if((g_pucUSBRxBuffer[ulReadIndex] >= 'a') &&
            (g_pucUSBRxBuffer[ulReadIndex] <= 'z'))
        {
            //转换
            g_pucUSBTxBuffer[ulWriteIndex] =
                (g_pucUSBRxBuffer[ulReadIndex] - 'a') + 'A';
        }
        else

```

```

    {
        //转换
        if((g_pucUSBRxBuffer[ulReadIndex] >= 'A') &&
            (g_pucUSBRxBuffer[ulReadIndex] <= 'Z'))
        {
            //转换
            g_pucUSBTxBuffer[ulWriteIndex] =
                (g_pucUSBRxBuffer[ulReadIndex] - 'Z') + 'z';
        }
        else
        {
            //转换
            g_pucUSBTxBuffer[ulWriteIndex] = g_pucUSBRxBuffer[ulReadIndex];
        }
    }
    // 更新指针
    ulWriteIndex++;
    ulWriteIndex = (ulWriteIndex == BULK_BUFFER_SIZE) ? 0 : ulWriteIndex;

    ulReadIndex++;
    ulReadIndex = (ulReadIndex == BULK_BUFFER_SIZE) ? 0 : ulReadIndex;
    ulLoop--;
}
// 发送数据
USBBufferDataWritten(&g_sTxBuffer, ulCount);
return(ulCount);
}

//*****
// 应用主函数.
//*****

int main(void)
{
    unsigned long ulTxCount = 0;
    unsigned long ulRxCount = 0;
    // char pcBuffer[16];
    //设置内核电压、主频 50Mhz
    SysCtlLDOSet(SYSCTL_LDO_2_75V);
    SysCtlClockSet(SYSCTL_XTAL_8MHZ | SYSCTL_SYSDIV_4 | SYSCTL_USE_PLL |
SYSCTL_OSC_MAIN );
    SysCtlPeripheralEnable(SYSCTL_PERIPH_GPIOF);
    GPIOPinTypeGPIOOutput(GPIO_PORTF_BASE, 0xf0);
    GPIOPinTypeGPIOInput(GPIO_PORTF_BASE, 0x0f);
    HWREG(GPIO_PORTF_BASE+GPIO_O_PUR) |= 0x0f;

```



```

// 初始化发送与接收 Buffer.
USBBufferInit((tUSBBuffer *)&g_sTxBuffer);
USBBufferInit((tUSBBuffer *)&g_sRxBuffer);
// 初始化 Bulk 设备
USBDBulkInit(0, (tUSBDBulkDevice *)&g_sBulkDevice);
while(1)
{
    if(g_ulFlags & COMMAND_STATUS_UPDATE)
    {
        //清除更新标志
        g_ulFlags &= ~COMMAND_STATUS_UPDATE;
        GPIOPinWrite(GPIO_PORTF_BASE, 0x30, 0x30);
    }
    // 发送完成
    if(ulTxCount != g_ulTxCount)
    {
        ulTxCount = g_ulTxCount;
        GPIOPinWrite(GPIO_PORTF_BASE, 0x10, 0x10);
        //usnprintf(pcBuffer, 16, " %d ", ulTxCount);
    }
    // 接收完成
    if(ulRxCount != g_ulRxCount)
    {
        ulRxCount = g_ulRxCount;
        GPIOPinWrite(GPIO_PORTF_BASE, 0x20, 0x20);
    }
}
}

```

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

第八章 CDC 设备

8.1 CDC 设备介绍

USB 的 CDC 类是 USB 通信设备类 (Communication Device Class) 的简称。CDC 类是 USB 组织定义的一类专门给各种通信设备(电信通信设备和中速网络通信设备)使用的 USB 子类。根据 CDC 类所针对通信设备的不同，CDC 类又被分成以下不同的模型：USB 传统纯电话业务 (POTS) 模型，USB ISDN 模型和 USB 网络模型。通常一个 CDC 类又由两个接口子类组成通信接口类 (Communication Interface Class) 和数据接口类 (Data Interface Class)。通信接口类对设备进行管理和控制，而数据接口类传送数据。这两个接口子类占有不同数量和类型的终端点 (Endpoints)，不同 CDC 类模型，其所对应的接口的终端点需求也是不同的。

8.2 CDC 数据类型

usbdc.h 中已经定义好 CDC 设备类中使用的所有数据类型和函数，同时也会使用 Buffer 数据类型及 API 函数，第 7 章有介绍 Buffer 数据类型及 API 函数，下面只介绍 CDC 设备类使用的数据类型。

```
typedef enum
{
    //CDC 状态没定义
    CDC_STATE_UNCONFIGURED,
    //空闲状态
    CDC_STATE_IDLE,
    //等待数据发送或者结束
    CDC_STATE_WAIT_DATA,
    //等待数据处理.
    CDC_STATE_WAIT_CLIENT
} tCDCState;
```

tCDCState，定义 CDC 端点状态。定义在 usbdc.h。用于端点状态标记与控制，可以保证数据传输不相互冲突。

```
typedef struct
{
    //USB 基地址
```

```

unsigned long ulUSBBase;
//设备信息
tDeviceInfo *psDevInfo;
//配置信息
tConfigDescriptor *psConfDescriptor;
//CDC 接收端点状态
volatile tCDCState eCDCRxState;
//CDC 发送端点状态
volatile tCDCState eCDCTxState;
//CDC 请求状态
volatile tCDCState eCDCRequestState;
//CDC 中断状态
volatile tCDCState eCDCInterruptState;
//请求更新标志
volatile unsigned char ucPendingRequest;
//暂时结束
unsigned short usBreakDuration;
//控制
unsigned short usControlLineState;
//UART 状态
unsigned short usSerialState;
//标志位
volatile unsigned short usDeferredOpFlags;
//最后一次发送数据大小
unsigned short usLastTxSize;
//UART 控制参数
tLineCoding sLineCoding;
//接收数据
volatile tBoolean bRxBlocked;
//控制数据
volatile tBoolean bControlBlocked;
//连接是否成功
volatile tBoolean bConnected;
//控制端点
unsigned char ucControlEndpoint;
//Bulk IN 端点
unsigned char ucBulkINEndpoint;
// Bulk Out 端点
unsigned char ucBulkOUTEndpoint;
//接口控制
unsigned char ucInterfaceControl;
//接口数据
unsigned char ucInterfaceData;
}

```

```
tCDCInstance;
```

tCDCInstance, CDC 设备类实例。定义了 CDC 设备类的 USB 基地址、设备信息、IN 端点、OUT 端点等信息。

```
typedef struct
{
    //VID
    unsigned short usVID;
    //PID
    unsigned short usPID;
    //最大耗电量
    unsigned short usMaxPowermA;
    //电源属性
    unsigned char ucPwrAttributes;
    //控制回调函数
    tUSBCallback pfnControlCallback;
    //控制回调函数的第一个参数
    void *pvControlCBData;
    //接收回调函数
    tUSBCallback pfnRxCallback;
    //接收回调函数的第一个参数
    void *pvRxCBData;
    //发送回调函数
    tUSBCallback pfnTxCallback;
    //发送回调函数的第一个参数
    void *pvTxCBData;
    //字符串描述符集合
    const unsigned char * const *ppStringDescriptors;
    //字符串描述符个数
    unsigned long ulNumStringDescriptors;
    //CDC 类实例
    tCDCSerInstance *psPrivateCDCSerData;
}
tUSBDCDCDevice;
```

tUSBDCDCDevice, CDC 设备类, 定义了 VID、PID、电源属性、字符串描述符等, 还包括了一个 CDC 设备类实例。其它设备描述符、配置信息通过 API 函数储入 tCDCSerInstance 定义的 CDC 设备实例中。

8.3 API 函数

在 CDC 设备类 API 库中定义了 13 个函数, 完成 USB CDC 设备初始化、配置及数据处理。以及 11 个 Buffer 操作函数, Buffer 第 7 章有介绍。下面为 usbdcdc.h 中定义的 API 函数:

```
void *USBDCDCInit(unsigned long ulIndex,
                  const tUSBDCDCDevice *psCDCDevice);

void *USBDCDCCompositeInit(unsigned long ulIndex,
                           const tUSBDCDCDevice *psCDCDevice);

unsigned long USBDCDCTxPacketAvailable(void *pvInstance);
```

```

unsigned long USBDCDCPacketWrite(void *pvInstance,
                                unsigned char *pcData,
                                unsigned long ulLength,
                                tBoolean bLast);

unsigned long USBDCDCRxPacketAvailable(void *pvInstance);

unsigned long USBDCDCPacketRead(void *pvInstance,
                                unsigned char *pcData,
                                unsigned long ulLength,
                                tBoolean bLast);

void USBDCDCSerialStateChange(void *pvInstance,
                               unsigned short usState);

void USBDCDCTerm(void *pvInstance);

void *USBDCDCSetControlCBData(void *pvInstance, void *pvCBData);
void *USBDCDCSetRxCBData(void *pvInstance, void *pvCBData);
void *USBDCDCSetTxCBData(void *pvInstance, void *pvCBData);
void USBDCDCPowerStatusSet(void *pvInstance, unsigned char ucPower);
tBoolean USBDCDCRemoteWakeupRequest(void *pvInstance);

```

```

void *USBDCDCInit(unsigned long ulIndex,
                  const tUSBDCDCDevice *psCDCDevice);

```

作用：初始化 CDC 设备硬件、协议，把其它配置参数填入 psCDCDevice 实例中。

参数：ulIndex，USB 模块代码，固定值：USB_BASE0。psDevice，CDC 设备类。

返回：指向配置后的 tUSBDCDCDevice。

```

void * USBDCDCCompositeInit(unsigned long ulIndex,
                             const tUSBDCDCDevice *psCDCDevice);

```

作用：初始化 CDC 设备协议，本函数在 USBDCDCInit 中已经调用。

参数：ulIndex，USB 模块代码，固定值：USB_BASE0。psDevice，CDC 设备类。

返回：指向配置后的 tUSBDCDCDevice。

```

unsigned long USBDCDCTxPacketAvailable(void *pvInstance);

```

作用：获取可用发送数据长度。

参数：pvInstance，tUSBDCDCDevice 设备指针。

返回：发送包大小，用发送数据长度。

```

unsigned long USBDCDCPacketWrite(void *pvInstance,
                                unsigned char *pcData,
                                unsigned long ulLength,
                                tBoolean bLast);

```

作用：通过 CDC 传输发送一个包数据，底层驱动，在 Buffer 中使用。

参数：pvInstance，tUSBDCDCDevice 设备指针。pcData，待写入的数据指针。ulLength，待写入数据的长度。bLast，是否传输结束包。

返回：成功发送长度，可能与 ulLength 长度不一样。

```

unsigned long USBDCDCRxPacketAvailable(void *pvInstance);

```

作用：获取接收数据长度。

参数：pvInstance，tUSBDCDCDevice 设备指针。

返回：可用接收数据个数，可读取的有效数据。

```

unsigned long USBDCDCPacketRead(void *pvInstance,
                                unsigned char *pcData,
                                unsigned long ulLength,
                                tBoolean bLast);

```

作用：通过 CDC 传输接收一个包数据，底层驱动，在 Buffer 中使用。

参数：pvInstance, tUSBDCDCDevice 设备指针。pcData, 读出数据指针。ulLength, 读出数据的长度。bLast, 是否是束包。

返回：成功接收长度，可能与 ulLength 长度不一样。

```

void USBDCDCSerialStateChange(void *pvInstance,
                                unsigned short usState);

```

作用：UART 收到数据后，调用此函数进行数据处理。

参数：pvInstance, tUSBDCDCDevice 设备指针。usState, UART 状态。

返回：无。

```

void USBDCDCTerm(void *pvInstance);

```

作用：结束 CDC 设备。

参数：pvInstance, 指向 tUSBDCDCDevice。

返回：无。

```

void *USBDCDCSetControlCBData(void *pvInstance, void *pvCBData);

```

作用：改变控制回调函数的第一个参数。

参数：pvInstance, 指向 tUSBDCDCDevice。pvCBData, 用于替换的参数

返回：旧参数指针。

```

void *USBDCDCSetRxCBData(void *pvInstance, void *pvCBData);

```

作用：改变接收回调函数的第一个参数。

参数：pvInstance, 指向 tUSBDCDCDevice。pvCBData, 用于替换的参数

返回：旧参数指针。

```

void *USBDCDCSetTxCBData(void *pvInstance, void *pvCBData);

```

作用：改变发送回调函数的第一个参数。

参数：pvInstance, 指向 tUSBDCDCDevice。pvCBData, 用于替换的参数

返回：旧参数指针。

```

void USBDCDCPowerStatusSet(void *pvInstance, unsigned char ucPower);

```

作用：修改电源属性、状态。

参数：pvInstance, 指向 tUSBDCDCDevice。ucPower, 电源属性。

返回：无。

```

tBoolean USBDCDCRemoteWakeupRequest(void *pvInstance);

```

作用：唤醒请求。

参数：pvInstance, 指向 tUSBDCDCDevice。

返回：无。

在这些函数中 USBDCDCInit 和 USBDCDCPacketWrite、USBDCDCPacketRead、USBDCDCTxPacketAvailable、USBDCDCRxPacketAvailable 函数最重要并且使用最多，USBDCDCInit 第一次使用 CDC 设备时，用于初始化 CDC 设备的配置与控制。USBDCDCPacketRead、USBDCDCPacketWrite、USBDCDCTxPacketAvailable、USBDCDCRxPacketAvailable 为 CDC 传输数据的底层驱动函数用于驱动 Buffer。

在 CDC 类中也会使用到 11 个 Buffer 处理函数，用于 Buffer 数据接收、发送及处理。在 CDC 传输中大量使用。使用方法在第 7 章有讲解。

8.4 CDC 设备开发

CDC 设备开发只需要 4 步就能完成。如图 2 所示，CDC 设备配置（主要是字符串描述符）、callback 函数编写、USB 处理器初始化、数据处理。

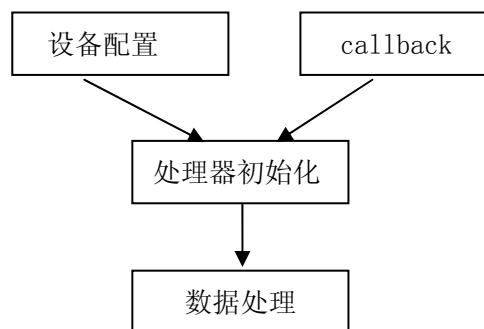


图 2

下面以“USB 转 UART”实例说明使用 USB 库开发 USB CDC 类过程：

第一步：CDC 设备配置（主要是字符串描述符），按字符串描述符标准完成串描述符配置，进而完成 CDC 设备配置。

```
#include "inc/hw_ints.h"
#include "inc/hw_memmap.h"
#include "inc/hw_types.h"
#include "inc/hw_uart.h"
#include "inc/hw_gpio.h"
#include "driverlib/debug.h"
#include "driverlib/gpio.h"
#include "driverlib/interrupt.h"
#include "driverlib/sysctl.h"
#include "driverlib/systick.h"
#include "driverlib/timer.h"
#include "driverlib/uart.h"
#include "driverlib/usb.h"
#include "usblib/usblib.h"
#include "usblib/usbcdc.h"
#include "usblib/usb-ids.h"
#include "usblib/device/usbdevice.h"
#include "usblib/device/usbdcdc.h"

#define UART_BUFFER_SIZE 256
volatile unsigned long g_ulUARTTxCount = 0;
volatile unsigned long g_ulUARTRxCount = 0;

// UART 设置.
#define USB_UART_BASE          UART0_BASE
#define USB_UART_PERIPH        SYSCTL_PERIPH_UART0
#define USB_UART_INT           INT_UART0
```

```

#define TX_GPIO_BASE          GPIO_PORTA_BASE
#define TX_GPIO_PERIPH        SYSCTL_PERIPH_GPIOA
#define TX_GPIO_PIN           GPIO_PIN_1
#define RX_GPIO_BASE          GPIO_PORTA_BASE
#define RX_GPIO_PERIPH        SYSCTL_PERIPH_GPIOA
#define RX_GPIO_PIN           GPIO_PIN_0
#define DEFAULT_BIT_RATE      115200
#define DEFAULT_UART_CONFIG    (UART_CONFIG_WLEN_8 | UART_CONFIG_PAR_NONE | \
                                UART_CONFIG_STOP_ONE)

// 发送中断标志.
static tBoolean g_bSendingBreak = false;
//系统时钟
volatile unsigned long g_ulSysTickCount = 0;
// g_ulFlags 使用的标志位.
#define COMMAND_PACKET_RECEIVED 0x00000001
#define COMMAND_STATUS_UPDATE  0x00000002
// 全局标志
volatile unsigned long g_ulFlags = 0;
//状态
char *g_pcStatus;
// 全局 USB 配置标志.
static volatile tBoolean g_bUSBConfigured = false;
unsigned long RxHandler(void *pvCBData, unsigned long ulEvent,
                        unsigned long ulMsgValue, void *pvMsgData);
unsigned long TxHandler(void *pvCBData, unsigned long ulEvent,
                        unsigned long ulMsgValue, void *pvMsgData);
unsigned long ControlHandler(void *pvCBData, unsigned long ulEvent,
                             unsigned long ulMsgValue, void *pvMsgData);

void USBUARTPrimeTransmit(unsigned long ulBase);
void CheckForSerialStateChange(const tUSBDCDCDevice *psDevice, long lErrors);
void SetControlLineState(unsigned short usState);
tBoolean SetLineCoding(tLineCoding *psLineCoding);
void GetLineCoding(tLineCoding *psLineCoding);
void SendBreak(tBoolean bSend);

const tUSBBuffer g_sTxBuffer;
const tUSBBuffer g_sRxBuffer;
const tUSBDCDCDevice g_sCDCDevice;
unsigned char g_pucUSBTxBuffer[];
unsigned char g_pucUSBRxBuffer[];

//*****
// 设备语言描述符.

```



```

//*****
const unsigned char g_pLangDescriptor[] =
{
    4,
    USB_DTYPE_STRING,
    USBShort(USB_LANG_EN_US)
};
//*****
// 制造商 字符串 描述符
//*****
const unsigned char g_pManufacturerString[] =
{
    (17 + 1) * 2,
    USB_DTYPE_STRING,
    'T', 0, 'e', 0, 'x', 0, 'a', 0, 's', 0, ' ', 0, 'I', 0, 'n', 0, 's', 0,
    't', 0, 'r', 0, 'u', 0, 'm', 0, 'e', 0, 'n', 0, 't', 0, 's', 0,
};
//*****
//产品 字符串 描述符
//*****
const unsigned char g_pProductString[] =
{
    2 + (16 * 2),
    USB_DTYPE_STRING,
    'V', 0, 'i', 0, 'r', 0, 't', 0, 'u', 0, 'a', 0, 'l', 0, ' ', 0,
    'C', 0, 'O', 0, 'M', 0, ' ', 0, 'P', 0, 'o', 0, 'r', 0, 't', 0,
};
//*****
// 产品 序列号 描述符
//*****
const unsigned char g_pSerialNumberString[] =
{
    2 + (8 * 2),
    USB_DTYPE_STRING,
    '1', 0, '2', 0, '3', 0, '4', 0, '5', 0, '6', 0, '7', 0, '9', 0,
};
//*****
// 设备接口字符串描述符
//*****
const unsigned char g_pControlInterfaceString[] =
{
    2 + (21 * 2),
    USB_DTYPE_STRING,
    'A', 0, 'C', 0, 'M', 0, ' ', 0, 'C', 0, 'o', 0, 'n', 0, 't', 0,

```

```

        'r', 0, 'o', 0, 'l', 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符

//*****

const unsigned char g_pConfigString[] =
{
    2 + (26 * 2),
    USB_DTYPE_STRING,
    'S', 0, 'e', 0, 'l', 0, 'f', 0, ' ', 0, 'P', 0, 'o', 0, 'w', 0,
    'e', 0, 'r', 0, 'e', 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
};

//*****

// 字符串描述符集合

//*****

const unsigned char * const g_pStringDescriptors[] =
{
    g_pLangDescriptor,
    g_pManufacturerString,
    g_pProductString,
    g_pSerialNumberString,
    g_pControlInterfaceString,
    g_pConfigString
};

#define NUM_STRING_DESCRIPTOR (sizeof(g_pStringDescriptors) / \
                                sizeof(unsigned char *))

//*****

// 定义 CDC 设备实例

//*****

tCDCSerInstance g_sCDCInstance;

//*****

// 定义 CDC 设备

//*****

const tUSBDCDCDevice g_sCDCDevice =
{
    0x1234,
    USB_PID_SERIAL,
    0,
    USB_CONF_ATTR_SELF_PWR,
    ControlHandler,
    (void *)&g_sCDCDevice,

```

```

        USBBufferEventCallback,
        (void *)&g_sRxBuffer,
        USBBufferEventCallback,
        (void *)&g_sTxBuffer,
        g_pStringDescriptors,
        NUM_STRING_DESCRIPTORS,
        &g_sCDCInstance
    };
//*****
// 定义 Buffer
//*****
unsigned char g_pcUSBRxBuffer[UART_BUFFER_SIZE];
unsigned char g_pucRxBufferWorkspace[USB_BUFFER_WORKSPACE_SIZE];
unsigned char g_pcUSBTxBuffer[UART_BUFFER_SIZE];
unsigned char g_pucTxBufferWorkspace[USB_BUFFER_WORKSPACE_SIZE];
const tUSBBuffer g_sRxBuffer =
{
    false,                // This is a receive buffer.
    RxHandler,            // pfnCallback
    (void *)&g_sCDCDevice, // Callback data is our device pointer.
    USBDCDCPacketRead,    // pfnTransfer
    USBDCDCRxPacketAvailable, // pfnAvailable
    (void *)&g_sCDCDevice, // pvHandle
    g_pcUSBRxBuffer,      // pcBuffer
    UART_BUFFER_SIZE,     // ulBufferSize
    g_pucRxBufferWorkspace // pvWorkspace
};
const tUSBBuffer g_sTxBuffer =
{
    true,                // This is a transmit buffer.
    TxHandler,            // pfnCallback
    (void *)&g_sCDCDevice, // Callback data is our device pointer.
    USBDCDCPacketWrite,   // pfnTransfer
    USBDCDCTxPacketAvailable, // pfnAvailable
    (void *)&g_sCDCDevice, // pvHandle
    g_pcUSBTxBuffer,      // pcBuffer
    UART_BUFFER_SIZE,     // ulBufferSize
    g_pucTxBufferWorkspace // pvWorkspace
};

```

第二步：完成 Callback 函数。Callback 函数用于处理输出端点、输入端点数据事务。CDC 设备接收回调函数包含以下事务：USB_EVENT_RX_AVAILABLE、USB_EVENT_DATA_REMAINING、USB_EVENT_REQUEST_BUFFER；CDC 设备发送回调函数包含了以下事务：USB_EVENT_TX_COMPLETE；CDC 设备控制回调函数包含了以下事务：USB_EVENT_CONNECTED、USB_EVENT_DISCONNECTED、USBDCDC_EVENT_GET_LINE_CODING、

USBD_CDC_EVENT_SET_LINE_CODING 、 USBD_CDC_EVENT_SET_CONTROL_LINE_STATE 、
USBD_CDC_EVENT_SEND_BREAK 、 USBD_CDC_EVENT_CLEAR_BREAK 、 USB_EVENT_SUSPEND 、
USB_EVENT_RESUME。如下表：

名称	属性	说明
USB_EVENT_RX_AVAILABLE	接收	有数据可接收
USB_EVENT_DATA_REMAINING	接收	剩余数据
USB_EVENT_REQUEST_BUFFER	接收	请求 Buffer
USB_EVENT_TX_COMPLETE	发送	发送完成
USB_EVENT_RESUME	控制	唤醒
USB_EVENT_SUSPEND	控制	挂起
USBD_CDC_EVENT_CLEAR_BREAK	控制	清除 Break 信号
USBD_CDC_EVENT_SEND_BREAK	控制	发送 Break 信号
USBD_CDC_EVENT_SET_CONTROL_LINE_STATE	控制	控制信号
USBD_CDC_EVENT_SET_LINE_CODING	控制	配置 UART 通信参数
USBD_CDC_EVENT_GET_LINE_CODING	控制	获取 UART 通信参数
USB_EVENT_DISCONNECTED	控制	断开
USB_EVENT_CONNECTED	控制	连接

表 2. CDC 事务

根据以上事务编写 Callback 函数：

```
//*****  
//CDC 设备类控制回调函数  
//*****  
unsigned long ControlHandler(void *pvCBData, unsigned long ulEvent,  
                             unsigned long ulMsgValue, void *pvMsgData)  
{  
    unsigned long ulIntsOff;  
    // 判断处理事务  
    switch(ulEvent)  
    {  
        //连接成功  
        case USB_EVENT_CONNECTED:  
            g_USBConfigured = true;  
            //清空 Buffer。  
            USBBufferFlush(&g_sTxBuffer);  
            USBBufferFlush(&g_sRxBuffer);  
            // 更新状态。  
            ulIntsOff = IntMasterDisable();  
            g_pcStatus = "Host connected."  
            g_ulFlags |= COMMAND_STATUS_UPDATE;  
            if(!ulIntsOff)  
            {  
                IntMasterEnable();  
            }  
        }  
    }
```

```

        break;
//断开连接.
case USB_EVENT_DISCONNECTED:
    g_bUSBConfigured = false;
    ulIntsOff = IntMasterDisable();
    g_pcStatus = "Host disconnected.";
    g_ulFlags |= COMMAND_STATUS_UPDATE;
    if(!ulIntsOff)
    {
        IntMasterEnable();
    }
    break;
// 获取 UART 通信参数.
case USBD_CDC_EVENT_GET_LINE_CODING:
    GetLineCoding(pvMsgData);
    break;
//设置 UART 通信参数。
case USBD_CDC_EVENT_SET_LINE_CODING:
    SetLineCoding(pvMsgData);
    break;
// 设置 RS232 RTS 和 DTR.
case USBD_CDC_EVENT_SET_CONTROL_LINE_STATE:
    SetControlLineState((unsigned short)ulMsgValue);
    break;
// 发送 Break 信号
case USBD_CDC_EVENT_SEND_BREAK:
    SendBreak(true);
    break;
// 清除 Break 信号
case USBD_CDC_EVENT_CLEAR_BREAK:
    SendBreak(false);
    break;
// 挂起与唤醒事务
case USB_EVENT_SUSPEND:
case USB_EVENT_RESUME:
    break;
default:
    break;
}
return(0);
}

//*****
//CDC 设备类 发送回调函数

```

```

//*****
unsigned long TxHandler(void *pvCBData, unsigned long ulEvent, unsigned long ulMsgValue,
    void *pvMsgData)
{
    switch(ulEvent)
    {
        //发送结束，在此不用处理数据
        case USB_EVENT_TX_COMPLETE:
            break;
        default:
            break;
    }
    return(0);
}

//*****
//CDC 设备类 发送回调函数
//*****
unsigned long TxHandler(void *pvCBData, unsigned long ulEvent, unsigned long ulMsgValue,
    void *pvMsgData)
{
    switch(ulEvent)
    {
        //发送结束，在此不用处理数据
        case USB_EVENT_TX_COMPLETE:
            break;
        default:
            break;
    }
    return(0);
}

//*****
//CDC 设备类 接收回调函数
//*****
unsigned long RxHandler(void *pvCBData, unsigned long ulEvent, unsigned long ulMsgValue,
    void *pvMsgData)
{
    unsigned long ulCount;
    //判断事务类型
    switch(ulEvent)
    {
        //接收数据
        case USB_EVENT_RX_AVAILABLE:
            {
                //UART 接收数据并能过 USB 发给主机。
            }
    }
}

```

```

        USBUARTPrimeTransmit(USB_UART_BASE);
        UARTIntEnable(USB_UART_BASE, UART_INT_TX);
        break;
    }
    // 检查剩余数据
    case USB_EVENT_DATA_REMAINING:
    {
        ulCount = UARTBusy(USB_UART_BASE) ? 1 : 0;
        return(ulCount);
    }
    //请求 Buffer
    case USB_EVENT_REQUEST_BUFFER:
    {
        return(0);
    }
    default:
        break;
}

return(0);
}

//*****
// 设置 RS232 RTS 和 DTR.
//*****
void SetControlLineState(unsigned short usState)
{
    // 根据 MCU 引脚自行添加。
}

//*****
// 设置 UART 通信参数
//*****
tBoolean SetLineCoding(tLineCoding *psLineCoding)
{
    unsigned long ulConfig;
    tBoolean bRetcode;
    bRetcode = true;
    // 数据长度
    switch(psLineCoding->ucDatabits)
    {
        case 5:
        {
            ulConfig = UART_CONFIG_WLEN_5;
            break;
        }
    }
}

```

```

case 6:
{
    ulConfig = UART_CONFIG_WLEN_6;
    break;
}
case 7:
{
    ulConfig = UART_CONFIG_WLEN_7;
    break;
}
case 8:
{
    ulConfig = UART_CONFIG_WLEN_8;
    break;
}
default:
{
    ulConfig = UART_CONFIG_WLEN_8;
    bRetcode = false;
    break;
}
}
// 校验位
switch(psLineCoding->ucParity)
{
    case USB_CDC_PARITY_NONE:
    {
        ulConfig |= UART_CONFIG_PAR_NONE;
        break;
    }
    case USB_CDC_PARITY_ODD:
    {
        ulConfig |= UART_CONFIG_PAR_ODD;
        break;
    }
    case USB_CDC_PARITY_EVEN:
    {
        ulConfig |= UART_CONFIG_PAR_EVEN;
        break;
    }
    case USB_CDC_PARITY_MARK:
    {
        ulConfig |= UART_CONFIG_PAR_ONE;
        break;
    }
}

```



```

    }
    case USB_CDC_PARITY_SPACE:
    {
        ulConfig |= UART_CONFIG_PAR_ZERO;
        break;
    }
    default:
    {
        ulConfig |= UART_CONFIG_PAR_NONE;
        bRetcode = false;
        break;
    }
}
//停止位
switch(psLineCoding->ucStop)
{
    case USB_CDC_STOP_BITS_1:
    {
        ulConfig |= UART_CONFIG_STOP_ONE;
        break;
    }
    case USB_CDC_STOP_BITS_2:
    {
        ulConfig |= UART_CONFIG_STOP_TWO;
        break;
    }
    default:
    {
        ulConfig = UART_CONFIG_STOP_ONE;
        bRetcode |= false;
        break;
    }
}
UARTConfigSetExpClk(USB_UART_BASE, SysCtlClockGet(), psLineCoding->ulRate,
                    ulConfig);
return(bRetcode);
}

//*****
// 获取 UART 通信参数.
//*****
void GetLineCoding(tLineCoding *psLineCoding)
{
    unsigned long ulConfig;

```

```

unsigned long ulRate;
UARTConfigGetExpClk(USB_UART_BASE, SysCtlClockGet(), &ulRate,
                    &ulConfig);
psLineCoding->ulRate = ulRate;
//发送数据长度
switch(ulConfig & UART_CONFIG_WLEN_MASK)
{
    case UART_CONFIG_WLEN_8:
    {
        psLineCoding->ucDatabits = 8;
        break;
    }
    case UART_CONFIG_WLEN_7:
    {
        psLineCoding->ucDatabits = 7;
        break;
    }
    case UART_CONFIG_WLEN_6:
    {
        psLineCoding->ucDatabits = 6;
        break;
    }
    case UART_CONFIG_WLEN_5:
    {
        psLineCoding->ucDatabits = 5;
        break;
    }
}
// 校验位
switch(ulConfig & UART_CONFIG_PAR_MASK)
{
    case UART_CONFIG_PAR_NONE:
    {
        psLineCoding->ucParity = USB_CDC_PARITY_NONE;
        break;
    }
    case UART_CONFIG_PAR_ODD:
    {
        psLineCoding->ucParity = USB_CDC_PARITY_ODD;
        break;
    }
    case UART_CONFIG_PAR_EVEN:
    {
        psLineCoding->ucParity = USB_CDC_PARITY_EVEN;

```

```

        break;
    }
    case UART_CONFIG_PAR_ONE:
    {
        psLineCoding->ucParity = USB_CDC_PARITY_MARK;
        break;
    }
    case UART_CONFIG_PAR_ZERO:
    {
        psLineCoding->ucParity = USB_CDC_PARITY_SPACE;
        break;
    }
}
//停止位
switch(ulConfig & UART_CONFIG_STOP_MASK)
{
    case UART_CONFIG_STOP_ONE:
    {
        psLineCoding->ucStop = USB_CDC_STOP_BITS_1;
        break;
    }
    case UART_CONFIG_STOP_TWO:
    {
        psLineCoding->ucStop = USB_CDC_STOP_BITS_2;
        break;
    }
}
}
//*****
// UART 发送 Break 信号
//*****
void SendBreak(tBoolean bSend)
{
    if(!bSend)
    {
        UARTBreakCtl(USB_UART_BASE, false);
        g_bSendingBreak = false;
    }
    else
    {
        UARTBreakCtl(USB_UART_BASE, true);
        g_bSendingBreak = true;
    }
}
}

```

第三步：系统初始化，配置内核电压、系统主频、使能端口、LED 控制等，本例中使用 4 个 LED 进行指示数据传输。在这个例子中，CDC 传输接收的数据发送给主机。原理图如图 3 所示：

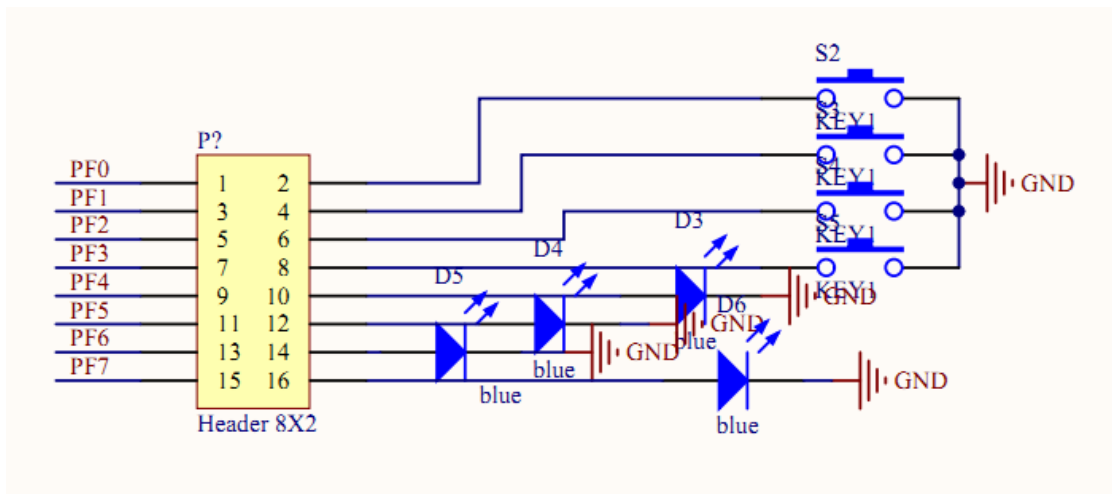


图 3

系统初始化：

```

unsigned long ulTxCount;
unsigned long ulRxCount;
// char pcBuffer[16];
//设置内核电压、主频 50Mhz
SysCtlLDOSet(SYSCTL_LDO_2_75V);
SysCtlClockSet(SYSCTL_XTAL_8MHZ | SYSCTL_SYSDIV_4 | SYSCTL_USE_PLL | SYSCTL_OSC_MAIN);

SysCtlPeripheralEnable(SYSCTL_PERIPH_GPIOF);
GPIOPinTypeGPIOOutput(GPIO_PORTF_BASE, 0xf0);
GPIOPinTypeGPIOInput(GPIO_PORTF_BASE, 0x0f);
HWREG(GPIO_PORTF_BASE+GPIO_O_PUR) |= 0x0f;

g_USBConfigured = false;
//UART 配置
SysCtlPeripheralEnable(USB_UART_PERIPH);
SysCtlPeripheralEnable(TX_GPIO_PERIPH);
SysCtlPeripheralEnable(RX_GPIO_PERIPH);
GPIOPinTypeUART(TX_GPIO_BASE, TX_GPIO_PIN);
GPIOPinTypeUART(RX_GPIO_BASE, RX_GPIO_PIN);
UARTConfigSetExpClk(USB_UART_BASE, SysCtlClockGet(), DEFAULT_BIT_RATE,
                    DEFAULT_UART_CONFIG);
UARTFIFOLevelSet(USB_UART_BASE, UART_FIFO_TX4_8, UART_FIFO_RX4_8);
// 配置和使能 UART 中断.
UARTIntClear(USB_UART_BASE, UARTIntStatus(USB_UART_BASE, false));
UARTIntEnable(USB_UART_BASE, (UART_INT_OE | UART_INT_BE | UART_INT_PE |
                                UART_INT_FE | UART_INT_RT | UART_INT_TX | UART_INT_RX));

```

```

// 初始化发送与接收 Buffer.
USBBufferInit((tUSBBuffer *)&g_sTxBuffer);
USBBufferInit((tUSBBuffer *)&g_sRxBuffer);
// 初始化 CDC 设备
USBDCDCInit(0, (tUSBDCDCDevice *)&g_sCDCDevice);

    ulRxCount = 0;
    ulTxCount = 0;
    IntEnable(USB_UART_INT);

```

第四步：数据处理。主要使用 11 个 Buffer 处理函数，用于 Buffer 数据接收、发送及处理。在 CDC 传输中大量使用。

```

//*****
//CheckForSerialStateChange 在接收到串行数据后，处理标志。
//*****
void CheckForSerialStateChange(const tUSBDCDCDevice *psDevice, long lErrors)
{
    unsigned short usSerialState;
    // 设置 TXCARRIER (DSR)和 RXCARRIER (DCD)位.
    usSerialState = USB_CDC_SERIAL_STATE_TXCARRIER |
                    USB_CDC_SERIAL_STATE_RXCARRIER;
    // 判断是什么标志
    if(lErrors)
    {
        if(lErrors & UART_DR_OE)
        {
            usSerialState |= USB_CDC_SERIAL_STATE_OVERRUN;
        }
        if(lErrors & UART_DR_PE)
        {
            usSerialState |= USB_CDC_SERIAL_STATE_PARITY;
        }
        if(lErrors & UART_DR_FE)
        {
            usSerialState |= USB_CDC_SERIAL_STATE_FRAMING;
        }
        if(lErrors & UART_DR_BE)
        {
            usSerialState |= USB_CDC_SERIAL_STATE_BREAK;
        }
        // 改变状态。
        USBDCDCSerialStateChange((void *)psDevice, usSerialState);
    }
}

```

```

}
//*****
//从 UART 中读取数据，并放在 CDC 设备 Buffer 中发送给 USB 主机
//*****
long ReadUARTData(void)
{
    long lChar, lErrors;
    unsigned char ucChar;
    unsigned long ulSpace;
    lErrors = 0;
    //检查有多少可用空间
    ulSpace = USBBufferSpaceAvailable((tUSBBuffer *)&g_sTxBuffer);
    //从 UART 中读取数据并写入到 CDC 设备类的 Buffer 中发送
    while(ulSpace && UARTCharsAvail(USB_UART_BASE))
    {
        //读一个字节
        lChar = UARTCharGetNonBlocking(USB_UART_BASE);
        //是不是控制或错误标志 。
        if(!(lChar & ~0xFF))
        {
            ucChar = (unsigned char)(lChar & 0xFF);
            USBBufferWrite((tUSBBuffer *)&g_sTxBuffer,
                           (unsigned char *)&ucChar, 1);
            ulSpace--;
        }
        else
        {
            lErrors |= lChar;
        }
        g_ulUARTRxCount++;
    }
    return(lErrors);
}
//*****
// 从 Buffer 中读取数据，通过 UART 发送
//*****
void USBUARTPrimeTransmit(unsigned long ulBase)
{
    unsigned long ulRead;
    unsigned char ucChar;
    if(g_bSendingBreak)
    {
        return;
    }
}

```

```

//检查 UART 中可用空间
while(UARTSpaceAvail(ulBase))
{
    //从 Buffer 中读取一个字节.
    ulRead = USBBufferRead((tUSBBuffer *)&g_sRxBuffer, &ucChar, 1);
    if(ulRead)
    {
        // 放在 UART TXFIFO 中发送.
        UARTCharPutNonBlocking(ulBase, ucChar);
        g_ulUARTTxCount++;
    }
    else
    {
        return;
    }
}

}

//*****
// UART 中断处理函数
//*****
void USBUARTIntHandler(void)
{
    unsigned long ulInts;
    long lErrors;
    //获取中断标志并清除
    ulInts = UARTIntStatus(USB_UART_BASE, true);
    UARTIntClear(USB_UART_BASE, ulInts);
    // 发送中断
    if(ulInts & UART_INT_TX)
    {
        // 从 USB 中获取数据并能过 UART 发送.
        USBUARTPrimeTransmit(USB_UART_BASE);
        // If the output buffer is empty, turn off the transmit interrupt.
        if(!USBBufferDataAvailable(&g_sRxBuffer))
        {
            UARTIntDisable(USB_UART_BASE, UART_INT_TX);
        }
    }
    // 接收中断.
    if(ulInts & (UART_INT_RX | UART_INT_RT))
    {
        //从 UART 中读取数据并通过 Buffer 发送给 USB 主机。
        lErrors = ReadUARTData();
    }
}

```

```

        //检查是否有控制信号或者错误信号。
        CheckForSerialStateChange(&g_sCDCDevice, lErrors);
    }
}
while(1)
{
    if(g_ulFlags & COMMAND_STATUS_UPDATE)
    {
        //清除更新标志，有数据更新。
        IntMasterDisable();
        g_ulFlags &= ~COMMAND_STATUS_UPDATE;
        IntMasterEnable();

        GPIOPinWrite(GPIO_PORTF_BASE, 0x30, 0x30);
    }
    // 发送完成
    if(ulTxCount != g_ulUARTTxCount)
    {
        ulTxCount = g_ulUARTTxCount;
        GPIOPinWrite(GPIO_PORTF_BASE, 0x10, 0x10);
        //usnprintf(pcBuffer, 16, " %d ", ulTxCount);
    }
    // 接收完成
    if(ulRxCount != g_ulUARTTxCount)
    {
        ulRxCount = g_ulUARTTxCount;
        GPIOPinWrite(GPIO_PORTF_BASE, 0x20, 0x20);
    }
}

```

使用上面四步就完成CDC设备开发。CDC设备开发时要加入两个lib库函数：usb.lib和DriverLib.lib，在启动代码中加入USB0DeviceIntHandler中断服务函数和USBUARTIntHandler中断服务程序。以上UART→RS232开发完成，在Win xp下运行效果如下图所示：



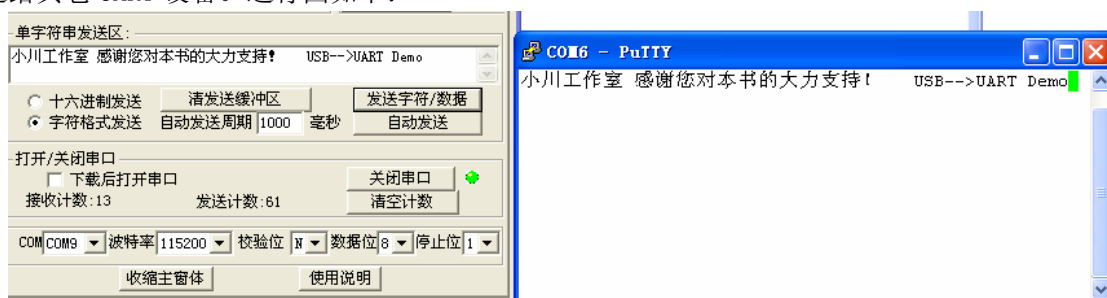


驱动安装

在枚举过程中可以看出，在电脑右下角可以看到“Virtual Com Port”字样，标示正在进行枚举，并手动安装驱动。枚举成功后，在“设备管理器”的“端口”中看到“DESCRIPTION_0”设备，如下图。现在 CDC 设备可以正式使用。



CDC 设备要配合上位机使用，上位机发送字符串通过 USB—>UART 设备转换后通过 UART 发送给其它 UART 设备。运行图如下：



CDC 设备 USB—>UART 开发源码较多，下面只列出一部分如下：

```
//*****
// 字符串描述符集合
//*****

const unsigned char * const g_pStringDescriptors[] =
{
    g_pLangDescriptor,
    g_pManufacturerString,
```

```

    g_pProductString,
    g_pSerialNumberString,
    g_pControlInterfaceString,
    g_pConfigString
};

#define NUM_STRING_DESCRIPTOR (sizeof(g_pStringDescriptors) / \
                                sizeof(unsigned char *))

//*****
// 定义 CDC 设备实例
//*****

tCDCSerInstance g_sCDCInstance;

//*****
// 定义 CDC 设备
//*****

const tUSBDCDCDevice g_sCDCDevice =
{
    0x1234,
    USB_PID_SERIAL,
    0,
    USB_CONF_ATTR_SELF_PWR,
    ControlHandler,
    (void *)&g_sCDCDevice,
    USBBufferEventCallback,
    (void *)&g_sRxBuffer,
    USBBufferEventCallback,
    (void *)&g_sTxBuffer,
    g_pStringDescriptors,
    NUM_STRING_DESCRIPTOR,
    &g_sCDCInstance
};

//*****
// 定义 Buffer
//*****

unsigned char g_pcUSBRxBuffer[UART_BUFFER_SIZE];
unsigned char g_pucRxBufferWorkspace[USB_BUFFER_WORKSPACE_SIZE];
unsigned char g_pcUSBTxBuffer[UART_BUFFER_SIZE];
unsigned char g_pucTxBufferWorkspace[USB_BUFFER_WORKSPACE_SIZE];
const tUSBBuffer g_sRxBuffer =
{
    false,                // This is a receive buffer.
    RxHandler,            // pfnCallback
    (void *)&g_sCDCDevice, // Callback data is our device pointer.
    USBDCDCPacketRead,    // pfnTransfer
    USBDCDCRxPacketAvailable, // pfnAvailable
};

```

```

        (void *)&g_sCDCDevice,          // pvHandle
        g_pcUSBRxBuffer,                // pcBuffer
        UART_BUFFER_SIZE,              // ulBufferSize
        g_pucRxBufferWorkspace          // pvWorkspace
    };

    const tUSBBuffer g_sTxBuffer =
    {
        true,                            // This is a transmit buffer.
        TxHandler,                       // pfnCallback
        (void *)&g_sCDCDevice,          // Callback data is our device pointer.
        USBDCDCPacketWrite,             // pfnTransfer
        USBDCDCTxPacketAvailable,       // pfnAvailable
        (void *)&g_sCDCDevice,          // pvHandle
        g_pcUSBTxBuffer,                // pcBuffer
        UART_BUFFER_SIZE,              // ulBufferSize
        g_pucTxBufferWorkspace          // pvWorkspace
    };

//*****
//CheckForSerialStateChange 在接收到串行数据后，处理标志。
//*****
void CheckForSerialStateChange(const tUSBDCDCDevice *psDevice, long lErrors)
{
    unsigned short usSerialState;
    // 设置 TXCARRIER (DSR)和 RXCARRIER (DCD)位.
    usSerialState = USB_CDC_SERIAL_STATE_TXCARRIER |
                    USB_CDC_SERIAL_STATE_RXCARRIER;
    // 判断是什么标志
    if(lErrors)
    {
        if(lErrors & UART_DR_OE)
        {
            usSerialState |= USB_CDC_SERIAL_STATE_OVERRUN;
        }
        if(lErrors & UART_DR_PE)
        {
            usSerialState |= USB_CDC_SERIAL_STATE_PARITY;
        }
        if(lErrors & UART_DR_FE)
        {
            usSerialState |= USB_CDC_SERIAL_STATE_FRAMING;
        }
        if(lErrors & UART_DR_BE)
        {

```

```

        usSerialState |= USB_CDC_SERIAL_STATE_BREAK;
    }
    // 改变状态。
    USBDCDCSerialStateChange((void *)psDevice, usSerialState);
}

}

//*****
//从 UART 中读取数据，并放在 CDC 设备 Buffer 中发送给 USB 主机
//*****
long ReadUARTData(void)
{
    long lChar, lErrors;
    unsigned char ucChar;
    unsigned long ulSpace;
    lErrors = 0;
    //检查有多少可用空间
    ulSpace = USBBufferSpaceAvailable((tUSBBuffer *)&g_sTxBuffer);
    //从 UART 中读取数据并写入到 CDC 设备类的 Buffer 中发送
    while(ulSpace && UARTCharsAvail(USB_UART_BASE))
    {
        //读一个字节
        lChar = UARTCharGetNonBlocking(USB_UART_BASE);
        //是不是控制或错误标志 。
        if(!(lChar & ~0xFF))
        {
            ucChar = (unsigned char)(lChar & 0xFF);
            USBBufferWrite((tUSBBuffer *)&g_sTxBuffer,
                           (unsigned char *)&ucChar, 1);
            ulSpace--;
        }
        else
        {
            lErrors |= lChar;
        }
        g_ulUARTRxCount++;
    }
    return(lErrors);
}

//*****
// 从 Buffer 中读取数据，通过 UART 发送
//*****
void USBUARTPrimeTransmit(unsigned long ulBase)
{
    unsigned long ulRead;

```

```

    unsigned char ucChar;
    if(g_bSendingBreak)
    {
        return;
    }
    //检查 UART 中可用空间
    while(UARTSpaceAvail(ulBase))
    {
        //从 Buffer 中读取一个字节.
        ulRead = USBBufferRead((tUSBBuffer *)&g_sRxBuffer, &ucChar, 1);
        if(ulRead)
        {
            // 放在 UART TXFIFO 中发送.
            UARTCharPutNonBlocking(ulBase, ucChar);
            g_ulUARTTxCount++;
        }
        else
        {
            return;
        }
    }
}

//*****
// 设置 RS232 RTS 和 DTR.
//*****

void SetControllLineState(unsigned short usState)
{
    // 根据 MCU 引脚自行添加。
}

//*****
// 设置 UART 通信参数
//*****

tBoolean SetLineCoding(tLineCoding *psLineCoding)
{
    unsigned long ulConfig;
    tBoolean bRetcode;
    bRetcode = true;
    // 数据长度
    switch(psLineCoding->ucDatabits)
    {
        case 5:
        {
            ulConfig = UART_CONFIG_WLEN_5;
            break;
        }
    }
}

```

```

    }
    case 6:
    {
        ulConfig = UART_CONFIG_WLEN_6;
        break;
    }
    case 7:
    {
        ulConfig = UART_CONFIG_WLEN_7;
        break;
    }
    case 8:
    {
        ulConfig = UART_CONFIG_WLEN_8;
        break;
    }
    default:
    {
        ulConfig = UART_CONFIG_WLEN_8;
        bRetcode = false;
        break;
    }
}

// 校验位
switch(psLineCoding->ucParity)
{
    case USB_CDC_PARITY_NONE:
    {
        ulConfig |= UART_CONFIG_PAR_NONE;
        break;
    }
    case USB_CDC_PARITY_ODD:
    {
        ulConfig |= UART_CONFIG_PAR_ODD;
        break;
    }
    case USB_CDC_PARITY_EVEN:
    {
        ulConfig |= UART_CONFIG_PAR_EVEN;
        break;
    }
    case USB_CDC_PARITY_MARK:
    {
        ulConfig |= UART_CONFIG_PAR_ONE;

```

```

        break;
    }
    case USB_CDC_PARITY_SPACE:
    {
        ulConfig |= UART_CONFIG_PAR_ZERO;
        break;
    }
    default:
    {
        ulConfig |= UART_CONFIG_PAR_NONE;
        bRetcode = false;
        break;
    }
}
//停止位
switch(psLineCoding->ucStop)
{
    case USB_CDC_STOP_BITS_1:
    {
        ulConfig |= UART_CONFIG_STOP_ONE;
        break;
    }
    case USB_CDC_STOP_BITS_2:
    {
        ulConfig |= UART_CONFIG_STOP_TWO;
        break;
    }
    default:
    {
        ulConfig = UART_CONFIG_STOP_ONE;
        bRetcode |= false;
        break;
    }
}

UARTConfigSetExpClk(USB_UART_BASE, SysCtlClockGet(), psLineCoding->ulRate,
                    ulConfig);

return(bRetcode);
}

//*****
// 获取 UART 通信参数.
//*****
void GetLineCoding(tLineCoding *psLineCoding)
{

```

```

unsigned long ulConfig;
unsigned long ulRate;
UARTConfigGetExpClk(USB_UART_BASE, SysCtlClockGet(), &ulRate,
                    &ulConfig);
psLineCoding->ulRate = ulRate;
//发送数据长度
switch(ulConfig & UART_CONFIG_WLEN_MASK)
{
    case UART_CONFIG_WLEN_8:
    {
        psLineCoding->ucDatabits = 8;
        break;
    }
    case UART_CONFIG_WLEN_7:
    {
        psLineCoding->ucDatabits = 7;
        break;
    }
    case UART_CONFIG_WLEN_6:
    {
        psLineCoding->ucDatabits = 6;
        break;
    }
    case UART_CONFIG_WLEN_5:
    {
        psLineCoding->ucDatabits = 5;
        break;
    }
}
// 校验位
switch(ulConfig & UART_CONFIG_PAR_MASK)
{
    case UART_CONFIG_PAR_NONE:
    {
        psLineCoding->ucParity = USB_CDC_PARITY_NONE;
        break;
    }
    case UART_CONFIG_PAR_ODD:
    {
        psLineCoding->ucParity = USB_CDC_PARITY_ODD;
        break;
    }
    case UART_CONFIG_PAR_EVEN:
    {

```



```

        psLineCoding->ucParity = USB_CDC_PARITY_EVEN;
        break;
    }
    case UART_CONFIG_PAR_ONE:
    {
        psLineCoding->ucParity = USB_CDC_PARITY_MARK;
        break;
    }
    case UART_CONFIG_PAR_ZERO:
    {
        psLineCoding->ucParity = USB_CDC_PARITY_SPACE;
        break;
    }
}
//停止位
switch(ulConfig & UART_CONFIG_STOP_MASK)
{
    case UART_CONFIG_STOP_ONE:
    {
        psLineCoding->ucStop = USB_CDC_STOP_BITS_1;
        break;
    }
    case UART_CONFIG_STOP_TWO:
    {
        psLineCoding->ucStop = USB_CDC_STOP_BITS_2;
        break;
    }
}
}
//*****
// UART 发送 Break 信号
//*****
void SendBreak(tBoolean bSend)
{
    if(!bSend)
    {
        UARTBreakCtl(USB_UART_BASE, false);
        g_bSendingBreak = false;
    }
    else
    {
        UARTBreakCtl(USB_UART_BASE, true);
        g_bSendingBreak = true;
    }
}

```

```

}
//*****
//CDC 设备类控制回调函数
//*****
unsigned long  ControlHandler(void *pvCBData, unsigned long ulEvent,
                             unsigned long ulMsgValue, void *pvMsgData)
{
    unsigned long ulIntsOff;
    // 判断处理事务
    switch(ulEvent)
    {
        //连接成功
        case USB_EVENT_CONNECTED:
            g_bUSBConfigured = true;
            //清空 Buffer。
            USBBufferFlush(&g_sTxBuffer);
            USBBufferFlush(&g_sRxBuffer);
            // 更新状态.
            ulIntsOff = IntMasterDisable();
            g_pcStatus = "Host connected.";
            g_ulFlags |= COMMAND_STATUS_UPDATE;
            if(!ulIntsOff)
            {
                IntMasterEnable();
            }
            break;
        //断开连接.
        case USB_EVENT_DISCONNECTED:
            g_bUSBConfigured = false;
            ulIntsOff = IntMasterDisable();
            g_pcStatus = "Host disconnected.";
            g_ulFlags |= COMMAND_STATUS_UPDATE;
            if(!ulIntsOff)
            {
                IntMasterEnable();
            }
            break;
        // 获取 UART 通信参数.
        case USBD_CDC_EVENT_GET_LINE_CODING:
            GetLineCoding(pvMsgData);
            break;
        //设置 UART 通信参数.
        case USBD_CDC_EVENT_SET_LINE_CODING:
            SetLineCoding(pvMsgData);
    }
}

```

```

        break;
// 设置 RS232 RTS 和 DTR.
case USBD_CDC_EVENT_SET_CONTROL_LINE_STATE:
    SetControlLineState((unsigned short)ulMsgValue);
    break;
// 发送 Break 信号
case USBD_CDC_EVENT_SEND_BREAK:
    SendBreak(true);
    break;
// 清除 Break 信号
case USBD_CDC_EVENT_CLEAR_BREAK:
    SendBreak(false);
    break;
// 挂起与唤醒事务
case USB_EVENT_SUSPEND:
case USB_EVENT_RESUME:
    break;
default:
    break;
}
return(0);
}

//*****
//CDC 设备类 发送回调函数
//*****
unsigned long TxHandler(void *pvCBData, unsigned long ulEvent, unsigned long ulMsgValue,
    void *pvMsgData)
{
    switch(ulEvent)
    {
        //发送结束, 在此不用处理数据
        case USB_EVENT_TX_COMPLETE:
            break;
        default:
            break;
    }
    return(0);
}

//*****
//CDC 设备类 接收回调函数
//*****
unsigned long RxHandler(void *pvCBData, unsigned long ulEvent, unsigned long ulMsgValue,
    void *pvMsgData)
{

```

```

    unsigned long ulCount;
    //判断事务类型
    switch(ulEvent)
    {
        //接收数据
        case USB_EVENT_RX_AVAILABLE:
        {
            //UART 接收数据并能过 USB 发给主机。
            USBUARTPrimeTransmit(USB_UART_BASE);
            UARTIntEnable(USB_UART_BASE, UART_INT_TX);
            break;
        }
        // 检查剩余数据
        case USB_EVENT_DATA_REMAINING:
        {
            ulCount = UARTBusy(USB_UART_BASE) ? 1 : 0;
            return(ulCount);
        }
        //请求 Buffer
        case USB_EVENT_REQUEST_BUFFER:
        {
            return(0);
        }
        default:
            break;
    }

    return(0);
}

//*****
// UART 中断处理函数
//*****
void USBUARTIntHandler(void)
{
    unsigned long ulInts;
    long lErrors;
    //获取中断标志并清除
    ulInts = UARTIntStatus(USB_UART_BASE, true);
    UARTIntClear(USB_UART_BASE, ulInts);
    // 发送中断
    if(ulInts & UART_INT_TX)
    {
        // 从 USB 中获取数据并能过 UART 发送.
    }
}

```

```

        USBUARTPrimeTransmit(USB_UART_BASE);
        // If the output buffer is empty, turn off the transmit interrupt.
        if(!USBBufferDataAvailable(&g_sRxBuffer))
        {
            UARTIntDisable(USB_UART_BASE, UART_INT_TX);
        }
    }
    // 接收中断.
    if(ulInts & (UART_INT_RX | UART_INT_RT))
    {
        //从 UART 中读取数据并通过 Buffer 发送给 USB 主机。
        lErrors = ReadUARTData();
        //检查是否有控制信号或者错误信号。
        CheckForSerialStateChange(&g_sCDCDevice, lErrors);
    }
}

//*****
// 应用主函数.
//*****
int main(void)
{
    unsigned long ulTxCount;
    unsigned long ulRxCount;
    // char pcBuffer[16];
    //设置内核电压、主频 50Mhz
    SysCtlLDOSet(SYSCTL_LDO_2_75V);
    SysCtlClockSet(SYSCTL_XTAL_8MHZ | SYSCTL_SYSDIV_4 | SYSCTL_USE_PLL |
SYSCTL_OSC_MAIN );
    SysCtlPeripheralEnable(SYSCTL_PERIPH_GPIOF);
    GPIOPinTypeGPIOOutput(GPIO_PORTF_BASE, 0xf0);
    GPIOPinTypeGPIOInput(GPIO_PORTF_BASE, 0xf0);
    HWREG(GPIO_PORTF_BASE+GPIO_O_PUR) |= 0x0f;

    g_bUSBConfigured = false;
    //UART 配置
    SysCtlPeripheralEnable(USB_UART_PERIPH);
    SysCtlPeripheralEnable(TX_GPIO_PERIPH);
    SysCtlPeripheralEnable(RX_GPIO_PERIPH);
    GPIOPinTypeUART(TX_GPIO_BASE, TX_GPIO_PIN);
    GPIOPinTypeUART(RX_GPIO_BASE, RX_GPIO_PIN);
    UARTConfigSetExpClk(USB_UART_BASE, SysCtlClockGet(), DEFAULT_BIT_RATE,
        DEFAULT_UART_CONFIG);
    UARTFIFOLevelSet(USB_UART_BASE, UART_FIFO_TX4_8, UART_FIFO_RX4_8);
    // 配置和使能 UART 中断.

```

```

UARTIntClear(USB_UART_BASE, UARTIntStatus(USB_UART_BASE, false));
UARTIntEnable(USB_UART_BASE, (UART_INT_OE | UART_INT_BE | UART_INT_PE |
                                UART_INT_FE | UART_INT_RT | UART_INT_TX | UART_INT_RX));

// 初始化发送与接收 Buffer.
USBBufferInit((tUSBBuffer *)&g_sTxBuffer);
USBBufferInit((tUSBBuffer *)&g_sRxBuffer);
// 初始化 CDC 设备
USBDCDCInit(0, (tUSBDCDCDevice *)&g_sCDCDevice);

ulRxCount = 0;
ulTxCount = 0;
IntEnable(USB_UART_INT);
while(1)
{
    if(g_ulFlags & COMMAND_STATUS_UPDATE)
    {
        //清除更新标志，有数据更新。
        IntMasterDisable();
        g_ulFlags &= ~COMMAND_STATUS_UPDATE;
        IntMasterEnable();

        GPIOPinWrite(GPIO_PORTF_BASE, 0x30, 0x30);
    }
    // 发送完成
    if(ulTxCount != g_ulUARTTxCount)
    {
        ulTxCount = g_ulUARTTxCount;
        GPIOPinWrite(GPIO_PORTF_BASE, 0x10, 0x10);
        //usnprintf(pcBuffer, 16, " %d ", ulTxCount);
    }
    // 接收完成
    if(ulRxCount != g_ulUARTTxCount)
    {
        ulRxCount = g_ulUARTTxCount;
        GPIOPinWrite(GPIO_PORTF_BASE, 0x20, 0x20);
    }
}
}

```

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

第九章 Mass Storage 设备

9.1 Mass Storage 设备介绍

USB 的 Mass Storage 类是 USB 大容量储存设备类 (Mass Storage Device Class)。专门用于大容量存储设备，比如 U 盘、移动硬盘、USB CD-ROM、读卡器等。在日常生活中经常用到。USB Mass Storage 设备开发相对简单。

9.2 MSC 数据类型

usbdmisc.h 中已经定义好 MSC 设备类中使用的所有数据类型和函数。下面介绍 MSC 设备类使用的数据类型。

```
typedef enum
{
    //设备加入
    USBDMSC_MEDIA_PRESENT,
    //设备移出
    USBDMSC_MEDIA_NOTPRESENT,
    //设备未知
    USBDMSC_MEDIA_UNKNOWN
}
```

```
tUSBDMSCMediaStatus;
```

tUSBDMSCMediaStatus，定义存储设备状态。定义在 usbdmisc.h。USBDMSCMediaChange() 函数用手修改此状态。

```
typedef struct
{
    //open 函数指针，用户自己完成该函数编写
    void *(* Open)(unsigned long ulDrive);
    //Close 函数指针，用户自己完成该函数编
    void (* Close)(void * pvDrive);
    // BlockRead 函数指针，用于读取存储设备，用户自己完成该函数编
    unsigned long (* BlockRead)(void * pvDrive, unsigned char *pucData,
                                unsigned long ulSector,
                                unsigned long ulNumBlocks);
    // BlockWrite 函数指针，用于写入存储设备，用户自己完成该函数编
```

```

        unsigned long (* BlockWrite)(void * pvDrive, unsigned char *pucData,
                                     unsigned long ulSector,
                                     unsigned long ulNumBlocks);

        // 读取当前扇区数
        unsigned long (* NumBlocks)(void * pvDrive);
    }

```

tMSCDMedia;

tMSCDMedia, 存储设备底层操作驱动。用于 MSC 设备对存储设备操作。

typedef struct

```

{
    unsigned long ulUSBBase;
    tDeviceInfo *psDevInfo;
    tConfigDescriptor *psConfDescriptor;
    unsigned char ucErrorCode;
    unsigned char ucSenseKey;
    unsigned short usAddSenseCode;
    void *pvMedia;
    volatile tBoolean bConnected;
    unsigned long ulFlags;
    tUSBDMSCMediaStatus eMediaStatus;
    unsigned long pulBuffer[DEVICE_BLOCK_SIZE>>2];
    unsigned long ulBytesToTransfer;
    unsigned long ulCurrentLBA;
    unsigned char ucINEndpoint;
    unsigned char ucINDMA;
    unsigned char ucOUTEndpoint;
    unsigned char ucOUTDMA;
    unsigned char ucInterface;
    unsigned char ucSCSIState;
}

```

tMSCInstance;

tMSCInstance, MSC 设备类实例。定义了 MSC 设备类的 USB 基地址、设备信息、IN 端点、OUT 端点等信息。

typedef struct

```

{
    unsigned short usVID;
    unsigned short usPID;
    unsigned char pucVendor[8];
    unsigned char pucProduct[16];
    unsigned char pucVersion[4];
    unsigned short usMaxPowermA;
    unsigned char ucPwrAttributes;
    const unsigned char * const *ppStringDescriptors;
    unsigned long ulNumStringDescriptors;
}

```



```

    tMSCMedia sMediaFunctions;
    tUSBCallback pfnEventCallback;
    tMSCInstance *psPrivateData;
}
tUSBDMSCDevice;

```

tUSBDMSCDevice, MSC 设备类, 定义了 VID、PID、电源属性、字符串描述符等, 还包括了一个 MSC 设备类实例。其它设备描述符、配置信息通过 API 函数储入 tMSCInstance 定义的 MSC 设备实例中。

9.3 API 函数

在 MSC 设备类 API 库中定义了 4 个函数, 完成 USB MSC 设备初始化、配置及数据处理。下面为 usbdMSC.h 中定义的 API 函数:

```

void *USBDMSCInit(unsigned long ulIndex,
                  const tUSBDMSCDevice *psMSCDevice);

void *USBDMSCCompositeInit(unsigned long ulIndex,
                           const tUSBDMSCDevice *psMSCDevice);

void USBDMSCTerm(void *pvInstance);
void USBDMSCMediaChange(void *pvInstance,
                       tUSBDMSCMediaStatus eMediaStatus);

void *USBDMSCInit(unsigned long ulIndex,
                  const tUSBDMSCDevice *psMSCDevice);

```

作用: 初始化 MSC 设备硬件、协议, 把其它配置参数填入 psMSCDevice 实例中。

参数: ulIndex, USB 模块代码, 固定值: USB_BASE0。psMSCDevice, MSC 设备类。

返回: 指向配置后的 tUSBDMSCDevice。

```

void *USBDMSCCompositeInit(unsigned long ulIndex,
                           const tUSBDMSCDevice *psMSCDevice);

```

作用: 初始化 MSC 设备协议, 本函数在 USBDMSCInit 中已经调用。

参数: ulIndex, USB 模块代码, 固定值: USB_BASE0。psMSCDevice, MSC 设备类。

返回: 指向配置后的 tUSBDMSCDevice。

```

void USBDMSCTerm(void *pvInstance);

```

作用: 结束 MSC 设备。

参数: pvInstance, 指向 tUSBDMSCDevice。

返回: 无。

```

void USBDMSCMediaChange(void *pvInstance,
                       tUSBDMSCMediaStatus eMediaStatus);

```

作用: 存储设备状态改变。

参数: pvInstance, 指向 tUSBDMSCDevice。

返回: 无。

在这些函数中 USBDMSCInit 函数最重要并且使用最多, USBDMSCInit 第一次使用 MSC 设备时, 用于初始化 MSC 设备的配置与控制。其它数据访问、控制处理由中断直接调用 tMSCMedia 定义的 5 个底层驱动函数完成。

9.4 MSC 设备开发

MSC 设备开发只需要 5 步就能完成。如图 2 所示, MSC 设备配置(主要是字符串描述符)、callback 函数编写、存储设备底层驱动编写、USB 处理器初始化、数据处理。

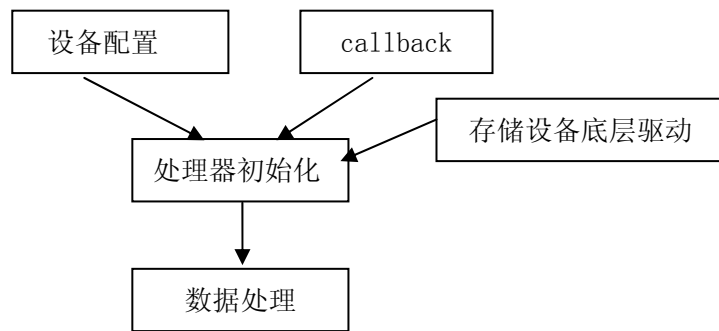


图 2

下面以“USB 转 UART”实例说明使用 USB 库开发 USB MSC 类过程：

第一步：MSC 设备配置（主要是字符串描述符），按字符串描述符标准完成串描述符配置，进而完成 MSC 设备配置。

```

#include "inc/hw_ints.h"
#include "inc/hw_memmap.h"
#include "inc/hw_gpio.h"
#include "inc/hw_types.h"
#include "inc/hw_ints.h"
#include "driverlib/sysctl.h"
#include "driverlib/gpio.h"
#include "driverlib/interrupt.h"
#include "driverlib/rom.h"
#include "driverlib/systick.h"
#include "driverlib/usb.h"
#include "driverlib/udma.h"
#include "usblib/usblib.h"
#include "usblib/usb-ids.h"
#include "usblib/device/usbdevice.h"
#include "usblib/device/usbdmhc.h"
#include "diskio.h"
#include "usbdscard.h"

//声明函数原型
unsigned long USBDMSCEventCallback(void *pvCBData, unsigned long ulEvent,
                                   unsigned long ulMsgParam,
                                   void *pvMsgData);

const tUSBDMSCDevice g_sMSCDevice;
//msc 状态
volatile enum
{
    MSC_DEV_DISCONNECTED,
    MSC_DEV_CONNECTED,

```

```

        MSC_DEV_IDLE,
        MSC_DEV_READ,
        MSC_DEV_WRITE,
    }
    g_eMSCState;
    //全局标志
#define FLAG_UPDATE_STATUS      1
    static unsigned long g_ulFlags;
    //DMA
    tDMAControlTable sDMAControlTable[64] __attribute__ ((aligned(1024)));

    /**
     * 语言描述符
     */
    const unsigned char g_pLangDescriptor[] =
    {
        4,
        USB_DTYPE_STRING,
        USBShort(USB_LANG_EN_US)
    };

    /**
     * 制造商 字符串 描述符
     */
    const unsigned char g_pManufacturerString[] =
    {
        (17 + 1) * 2,
        USB_DTYPE_STRING,
        'T', 0, 'e', 0, 'x', 0, 'a', 0, 's', 0, ' ', 0, 'I', 0, 'n', 0, 's', 0,
        't', 0, 'r', 0, 'u', 0, 'm', 0, 'e', 0, 'n', 0, 't', 0, 's', 0,
    };

    /**
     * 产品 字符串 描述符
     */
    const unsigned char g_pProductString[] =
    {
        (19 + 1) * 2,
        USB_DTYPE_STRING,
        'M', 0, 'a', 0, 's', 0, 's', 0, ' ', 0, 'S', 0, 't', 0, 'o', 0, 'r', 0,
        'a', 0, 'g', 0, 'e', 0, ' ', 0, 'D', 0, 'e', 0, 'v', 0, 'i', 0, 'c', 0,
        'e', 0,
    };

    /**
     * 产品 序列号 描述符
     */

```

```

const unsigned char g_pSerialNumberString[] =
{
    (8 + 1) * 2,
    USB_DTYPE_STRING,
    '1', 0, '2', 0, '3', 0, '4', 0, '5', 0, '6', 0, '7', 0, '8', 0
};

//*****
// 设备接口字符串描述符
//*****
const unsigned char g_pDataInterfaceString[] =
{
    (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
};

//*****
// 设备配置字符串描述符
//*****
const unsigned char g_pConfigString[] =
{
    (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
};

//*****
// 字符串描述符集合
//*****
const unsigned char * const g_pStringDescriptors[] =
{
    g_pLangDescriptor,
    g_pManufacturerString,
    g_pProductString,
    g_pSerialNumberString,
    g_pDataInterfaceString,
    g_pConfigString
};

#define NUM_STRING_DESCRIPTOR (sizeof(g_pStringDescriptors) / \
                                sizeof(unsigned char *))

//*****

```

```

//MSC 实例，配置并为设备信息提供空间
//*****

tMSCInstance g_sMSCInstance;
//*****

//设备配置
//*****

const tUSBDMSCDevice g_sMSCDevice =
{
    USB_VID_STELLARIS,
    USB_PID_MSC,
    "TI",
    "Mass Storage",
    "1.00",
    500,
    USB_CONF_ATTR_SELF_PWR,
    g_pStringDescriptors,
    NUM_STRING_DESCRIPTOR,
    {
        USBDMSCStorageOpen,
        USBDMSCStorageClose,
        USBDMSCStorageRead,
        USBDMSCStorageWrite,
        USBDMSCStorageNumBlocks
    },
    USBDMSCEventCallback,
    &g_sMSCInstance
};

```

```

#define MSC_BUFFER_SIZE 512

```

第二步：完成 Callback 函数。Callback 函数用于返回数据处理状态：

```

//*****

//callback 函数
//*****

unsigned long USBDMSCEventCallback(void *pvCBData, unsigned long ulEvent,
                                   unsigned long ulMsgParam, void *pvMsgData)
{
    switch(ulEvent)
    {
        // 正在写数据到存储设备.
        case USBD_MSC_EVENT_WRITING:
        {
            if(g_eMSCState != MSC_DEV_WRITE)
            {
                g_eMSCState = MSC_DEV_WRITE;
                g_ulFlags |= FLAG_UPDATE_STATUS;
            }
        }
    }
}

```

```

        }
        break;
    }
    //读取数据.
    case USBD_MSC_EVENT_READING:
    {
        if(g_eMSCState != MSC_DEV_READ)
        {
            g_eMSCState = MSC_DEV_READ;
            g_ulFlags |= FLAG_UPDATE_STATUS;
        }

        break;
    }
    //空闲
    case USBD_MSC_EVENT_IDLE:
    default:
    {
        break;
    }
}
return(0);
}

```

第三步：完成接口函数编写：

```

#define SDCARD_PRESENT          0x00000001
#define SDCARD_IN_USE          0x00000002

struct
{
    unsigned long ulFlags;
}
g_sDriveInformation;

//*****
//    打开存储设备
//*****
void *    USBDMSCStorageOpen(unsigned long ulDrive)
{
    unsigned char ucPower;
    unsigned long ulTemp;
    // 检查是否在使用.
    if(g_sDriveInformation.ulFlags & SDCARD_IN_USE)
    {

```

```

        return(0);
    }
    // 初始化存储设备.
    ulTemp = disk_initialize(0);
    if(ulTemp == RES_OK)
    {
        //打开电源
        ucPower = 1;
        disk_ioctl(0, CTRL_POWER, &ucPower);
        //设置标志.
        g_sDriveInformation.ulFlags = SDCARD_PRESENT | SDCARD_IN_USE;
    }
    else if(ulTemp == STA_NODISK)
    {
        // 没有存储设备.
        g_sDriveInformation.ulFlags = SDCARD_IN_USE;
    }
    else
    {
        return(0);
    }
    return((void *)&g_sDriveInformation);
}

//*****
// 关闭存储设备
//*****
void USBDMSCStorageClose(void * pvDrive)
{
    unsigned char ucPower;
    g_sDriveInformation.ulFlags = 0;
    ucPower = 0;
    disk_ioctl(0, CTRL_POWER, &ucPower);
}

//*****
// 读取扇区数据
//*****
unsigned long USBDMSCStorageRead(void * pvDrive,
                                unsigned char *pucData,
                                unsigned long ulSector,
                                unsigned long ulNumBlocks)
{
    if(disk_read (0, pucData, ulSector, ulNumBlocks) == RES_OK)
    {
        return(ulNumBlocks * 512);
    }
}

```

```

    }
    return(0);
}

//*****
// 写数据到扇区
//*****
unsigned long USBDMSCStorageWrite(void * pvDrive,
                                   unsigned char *pucData,
                                   unsigned long ulSector,
                                   unsigned long ulNumBlocks)
{
    if(disk_write(0, pucData, ulSector, ulNumBlocks) == RES_OK)
    {
        return(ulNumBlocks * 512);
    }
    return(0);
}

//*****
// 获取当前扇区
//*****
unsigned long USBDMSCStorageNumBlocks(void * pvDrive)
{
    unsigned long ulSectorCount;
    disk_ioctl(0, GET_SECTOR_COUNT, &ulSectorCount);
    return(ulSectorCount);
}

#define USBDMSC_IDLE          0x00000000
#define USBDMSC_NOT_PRESENT   0x00000001

//*****
// 存储设备当前状态
//*****
unsigned long USBDMSCStorageStatus(void * pvDrive);

第四步：系统初始化，配置内核电压、系统主频、使能端口、等。系统初始化：
//系统初始化。
SysCtlLDOSet(SYSCTL_LDO_2_75V);
SysCtlClockSet(SYSCTL_XTAL_8MHZ | SYSCTL_SYSDIV_4 | SYSCTL_USE_PLL | SYSCTL_OSC_MAIN );

SysCtlPeripheralEnable(SYSCTL_PERIPH_GPIOF);
GPIOPinTypeGPIOOutput(GPIO_PORTF_BASE, 0xf0);
GPIOPinTypeGPIOInput(GPIO_PORTF_BASE, 0x0f);
HWREG(GPIO_PORTF_BASE+GPIO_O_PUR) |= 0x0f;
// ucDMA 配置
SysCtlPeripheralEnable(SYSCTL_PERIPH_UDMA);
SysCtlDelay(10);

```



```

uDMAControlBaseSet(&sDMAControlTable[0]);
uDMAEnable();
g_ulFlags = 0;
g_eMSCState = MSC_DEV_IDLE;
//msc 设备初始化
USBDMSCInit(0, (tUSBDMSCDevice *)&g_sMSCDevice);
//初始化存储设备
disk_initialize(0);
第五步：状态处理，其它控制。
while(1)
{
    switch(g_eMSCState)
    {
        case MSC_DEV_READ:
        {
            if(g_ulFlags & FLAG_UPDATE_STATUS)
            {
                g_ulFlags &= ~FLAG_UPDATE_STATUS;
            }
            break;
        }
        case MSC_DEV_WRITE:
        {
            if(g_ulFlags & FLAG_UPDATE_STATUS)
            {
                g_ulFlags &= ~FLAG_UPDATE_STATUS;
            }
            break;
        }
        case MSC_DEV_IDLE:
        default:
        {
            break;
        }
    }
}

```

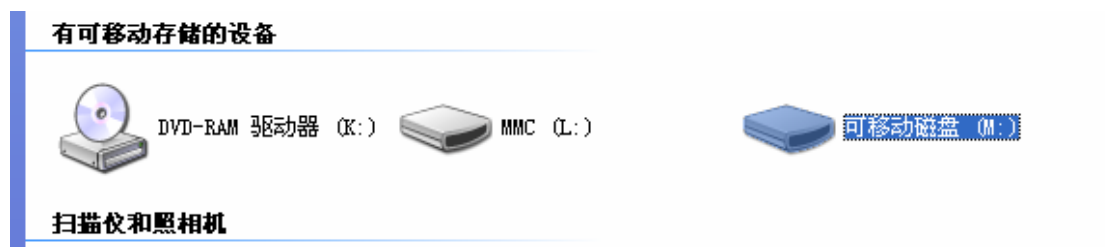
使用上面五步就完成MSC 设备开发。MSC 设备开发时要加入两个lib库函数：usb.lib和 DriverLib.lib，在启动代码中加入 USB0DeviceIntHandler 中断服务函数。以上 MSC 设备开发完成，在 Win xp 下运行效果如下图所示：



在枚举过程中可以看出，在电脑右下角可以看到“Mass Storage Device”字样，标示正在进行枚举，并手动安装驱动。枚举成功后，在“设备管理器”的“通用串行总线控制器”中看到“USB Mass Storage Device”设备，如下图。现在 MSC 设备可以正式使用。



MSC 设备可以在“我的电脑”中“有可移动存储的设备”找到：



MSC 设备开发源码较多，下面只列出一部分如下：

```
#include "inc/hw_ints.h"
#include "inc/hw_memmap.h"
#include "inc/hw_gpio.h"
#include "inc/hw_types.h"
#include "inc/hw_ints.h"
#include "driverlib/sysctl.h"
#include "driverlib/gpio.h"
#include "driverlib/interrupt.h"
#include "driverlib/rom.h"
#include "driverlib/systick.h"
```

```

#include "driverlib/usb.h"
#include "driverlib/udma.h"
#include "usblib/usblib.h"
#include "usblib/usb-ids.h"
#include "usblib/device/usbdevice.h"
#include "usblib/device/usbdmasc.h"
#include "diskio.h"
#include "usbdscard.h"

//声明函数原型
unsigned long USBDMSCEventCallback(void *pvCBData, unsigned long ulEvent,
                                   unsigned long ulMsgParam,
                                   void *pvMsgData);

const tUSBDMSCDevice g_sMSCDevice;
//msc 状态
volatile enum
{
    MSC_DEV_DISCONNECTED,
    MSC_DEV_CONNECTED,
    MSC_DEV_IDLE,
    MSC_DEV_READ,
    MSC_DEV_WRITE,
}
g_eMSCState;
//全局标志
#define FLAG_UPDATE_STATUS      1
static unsigned long g_ulFlags;
//DMA
tDMAControlTable sDMAControlTable[64] __attribute__ ((aligned(1024)));

//*****
// 语言描述符
//*****
const unsigned char g_pLangDescriptor[] =
{
    4,
    USB_DTYPE_STRING,
    USBShort(USB_LANG_EN_US)
};
//*****
// 制造商 字符串 描述符
//*****
const unsigned char g_pManufacturerString[] =

```

```

{
    (17 + 1) * 2,
    USB_DTYPE_STRING,
    'T', 0, 'e', 0, 'x', 0, 'a', 0, 's', 0, ' ', 0, 'I', 0, 'n', 0, 's', 0,
    't', 0, 'r', 0, 'u', 0, 'm', 0, 'e', 0, 'n', 0, 't', 0, 's', 0,
};

//*****
//产品 字符串 描述符
//*****
const unsigned char g_pProductString[] =
{
    (19 + 1) * 2,
    USB_DTYPE_STRING,
    'M', 0, 'a', 0, 's', 0, 's', 0, ' ', 0, 'S', 0, 't', 0, 'o', 0, 'r', 0,
    'a', 0, 'g', 0, 'e', 0, ' ', 0, 'D', 0, 'e', 0, 'v', 0, 'i', 0, 'c', 0,
    'e', 0
};

//*****
// 产品 序列号 描述符
//*****
const unsigned char g_pSerialNumberString[] =
{
    (8 + 1) * 2,
    USB_DTYPE_STRING,
    '1', 0, '2', 0, '3', 0, '4', 0, '5', 0, '6', 0, '7', 0, '8', 0
};

//*****
// 设备接口字符串描述符
//*****
const unsigned char g_pDataInterfaceString[] =
{
    (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
};

//*****
// 设备配置字符串描述符
//*****
const unsigned char g_pConfigString[] =
{
    (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
};

//*****
// 字符串描述符集合
//*****

const unsigned char * const g_pStringDescriptors[] =
{
    g_pLangDescriptor,
    g_pManufacturerString,
    g_pProductString,
    g_pSerialNumberString,
    g_pDataInterfaceString,
    g_pConfigString
};

#define NUM_STRING_DESCRIPTOR (sizeof(g_pStringDescriptors) / \
                                sizeof(unsigned char *))

//*****
//MSC 实例，配置并为设备信息提供空间
//*****
tMSCInstance g_sMSCInstance;
//*****
//设备配置
//*****
const tUSBDMSCDevice g_sMSCDevice =
{
    USB_VID_STELLARIS,
    USB_PID_MSC,
    "TI",
    "Mass Storage",
    "1.00",
    500,
    USB_CONF_ATTR_SELF_PWR,
    g_pStringDescriptors,
    NUM_STRING_DESCRIPTOR,
    {
        USBDMSCStorageOpen,
        USBDMSCStorageClose,
        USBDMSCStorageRead,
        USBDMSCStorageWrite,
        USBDMSCStorageNumBlocks
    },
};

```

```

        USBDMSCEventCallback,
        &g_MSCInstance
    };

#define MSC_BUFFER_SIZE 512
//*****
//callback 函数
//*****
unsigned long USBDMSCEventCallback(void *pvCBData, unsigned long ulEvent,
                                   unsigned long ulMsgParam, void *pvMsgData)
{
    switch(ulEvent)
    {
        // 正在写数据到存储设备.
        case USBD_MSC_EVENT_WRITING:
        {
            if(g_eMSCState != MSC_DEV_WRITE)
            {
                g_eMSCState = MSC_DEV_WRITE;
                g_ulFlags |= FLAG_UPDATE_STATUS;
            }
            break;
        }
        //读取数据.
        case USBD_MSC_EVENT_READING:
        {
            if(g_eMSCState != MSC_DEV_READ)
            {
                g_eMSCState = MSC_DEV_READ;
                g_ulFlags |= FLAG_UPDATE_STATUS;
            }

            break;
        }
        //空闲
        case USBD_MSC_EVENT_IDLE:
        default:
        {
            break;
        }
    }
    return(0);
}

//*****

```

```

//主函数
//*****
int main(void)
{
    //系统初始化。
    SysCtlLDOSet(SYSCTL_LDO_2_75V);
    SysCtlClockSet(SYSCTL_XTAL_8MHZ    |    SYSCTL_SYSDIV_4    |    SYSCTL_USE_PLL    |
SYSCTL_OSC_MAIN );
    SysCtlPeripheralEnable(SYSCTL_PERIPH_GPIOF);
    GPIOPinTypeGPIOOutput(GPIO_PORTF_BASE, 0xf0);
    GPIOPinTypeGPIOInput (GPIO_PORTF_BASE, 0x0f);
    HWREG(GPIO_PORTF_BASE+GPIO_O_PUR) |= 0x0f;
    // ucDMA 配置
    SysCtlPeripheralEnable(SYSCTL_PERIPH_UDMA);
    SysCtlDelay(10);
    uDMAControlBaseSet(&sDMAControlTable[0]);
    uDMAEnable();
    g_ulFlags = 0;
    g_eMSCState = MSC_DEV_IDLE;
    //msc 设备初始化
    USBDMSCInit(0, (tUSBDMSCDevice *)&g_sMSCDevice);
    //初始化存储设备
    disk_initialize(0);
    while(1)
    {
        switch(g_eMSCState)
        {
            case MSC_DEV_READ:
            {
                if(g_ulFlags & FLAG_UPDATE_STATUS)
                {
                    g_ulFlags &= ~FLAG_UPDATE_STATUS;
                }
                break;
            }
            case MSC_DEV_WRITE:
            {
                if(g_ulFlags & FLAG_UPDATE_STATUS)
                {
                    g_ulFlags &= ~FLAG_UPDATE_STATUS;
                }
                break;
            }
            case MSC_DEV_IDLE:

```

```
        default:
        {
            break;
        }
    }
}
```


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

第十章 Composite 设备

10.1 Composite 设备介绍

USB 的 Composite 类是 USB 复合设备类，一个 USB 设备具有多种设备功能，比如一个 USB 设备同时具有鼠标和键盘功能。单一的 USB 设备开发相对简单，但在很多时候使用的 USB 设备具有多种功能。Composite 类可以满足这种要求。

10.2 Composite 数据类型

usbcomp.h 中已经定义好 composite 设备类中使用的所有数据类型和函数。下面介绍 composite 设备类使用的数据类型。

```
typedef struct
{
    const tDeviceInfo *pDeviceInfo;
    const tConfigHeader *psConfigHeader;
    unsigned char ucIfaceOffset;
} tUSBDCompositeEntry;
tUSBDCompositeEntry, 定义 composite 设备信息。定义在 usbcomp.h。
```

```
typedef struct
{
    unsigned long ulUSBBase;
    tDeviceInfo *psDevInfo;
    tConfigDescriptor sConfigDescriptor;
    tDeviceDescriptor sDeviceDescriptor;
    tConfigHeader sCompConfigHeader;
    tConfigSection psCompSections[2];
    tConfigSection *ppsCompSections[2];
    unsigned long ulDataSize;
    unsigned char *pucData;
}
```

```
tCompositeInstance;
```

tCompositeInstance, 设备类实例。定义了 Composite 设备类的 USB 基地址、设备信息、IN 端点、OUT 端点等信息。

```
typedef struct
```

```

{
    const tDeviceInfo *psDevice;
    void *pvInstance;
}
tCompositeEntry;
tCompositeEntry, Composite 各设备的设备信息。
typedef struct
{
    unsigned short usVID;
    unsigned short usPID;
    unsigned short usMaxPowermA;
    unsigned char ucPwrAttributes;
    tUSBCallback pfncallback;
    const unsigned char * const *ppStringDescriptors;
    unsigned long ulNumStringDescriptors;
    unsigned long ulNumDevices;
    tCompositeEntry *psDevices;
    tCompositeInstance *psPrivateData;
}
tUSBDCompositeDevice;

```

tUSBDCompositeDevice, Composite 设备类, 定义了 VID、PID、电源属性、字符串描述符等, 还包括了 Composite 设备类实例。其它设备描述符、配置信息通过 API 函数储入 tCompositeInstance 定义的 Composite 设备实例中。

10.3 API 函数

在 Composite 设备类 API 库中定义了 2 个函数, 完成 USB Composite 设备初始化、配置及数据处理。下面为 usbdcomp.h 中定义的 API 函数:

```

void *USBDCompositeInit(unsigned long ulIndex,
                        tUSBDCompositeDevice *psCompDevice,
                        unsigned long ulSize,
                        unsigned char *pucData);

void USBDCompositeTerm(void *pvInstance);

void *USBDCompositeInit(unsigned long ulIndex,
                        tUSBDCompositeDevice *psCompDevice,
                        unsigned long ulSize,
                        unsigned char *pucData);

```

作用: 初始化 Composite 设备硬件、协议, 把其它配置参数填入 psCompDevice 实例中。

参数: ulIndex, USB 模块代码, 固定值: USB_BASE0。psMSCDevice, MSC 设备类。

返回: 指向配置后的 tUSBDCompositeDevice。

```
void USBDCompositeTerm(void *pvInstance);
```

作用: 结束 Composite 设备。

参数: pvInstance, 指向 tUSBDCompositeDevice。

返回: 无。

在这些函数中 USBDCompositeInit 函数最重要, 用于处理各子设备信息, 保存所有子设备配置及其它数据。

9.4 Composite 设备开发

Composite 设备开发只需要 3 步就能完成：各子设备配置、完善接口函数；Composite 设备配置、协调；各子设备数据处理。如图 2 所示，

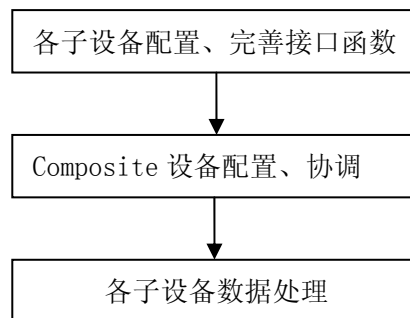


图 2

下面以“电子教鞭”实例说明使用 USB 库开发 USB Composite 设备过程，电子教鞭有两个重要功能，U 盘功能和控制功能。所以要做两个子类：大容量存储类与键盘类：

第一步：各子设备配置、完善接口函数：

```
#define DESCRIPTOR_DATA_SIZE    (COMPOSITE_DHID_SIZE + COMPOSITE_DMSC_SIZE)
unsigned char g_pucDescriptorData[DESCRIPTOR_DATA_SIZE];

//声明函数原型
unsigned long USBDMSCEventCallback(void *pvCBData, unsigned long ulEvent,
                                   unsigned long ulMsgParam,
                                   void *pvMsgData);

//声明函数原型
unsigned long KeyboardHandler(void *pvCBData,
                              unsigned long ulEvent,
                              unsigned long ulMsgData,
                              void *pvMsgData);

unsigned long EventHandler(void *pvCBData, unsigned long ulEvent,
                           unsigned long ulMsgData, void *pvMsgData);

const tUSBDMSCDevice g_sMSCDevice;
//msc 状态
volatile enum
{
    MSC_DEV_DISCONNECTED,
    MSC_DEV_CONNECTED,
    MSC_DEV_IDLE,
    MSC_DEV_READ,
    MSC_DEV_WRITE,
}
g_eMSCState;
//全局标志
#define FLAG_UPDATE_STATUS    1
```

```

static unsigned long g_ulFlags;
//DMA
tDMAControlTable sDMAControlTable[64] __attribute__ ((aligned(1024)));

//*****
// 语言描述符
//*****
const unsigned char g_pLangDescriptor[] =
{
    4,
    USB_DTYPE_STRING,
    USBShort(USB_LANG_EN_US)
};
//*****
// 制造商 字符串 描述符
//*****
const unsigned char g_pManufacturerString[] =
{
    (17 + 1) * 2,
    USB_DTYPE_STRING,
    'T', 0, 'e', 0, 'x', 0, 'a', 0, 's', 0, ' ', 0, 'I', 0, 'n', 0, 's', 0,
    't', 0, 'r', 0, 'u', 0, 'm', 0, 'e', 0, 'n', 0, 't', 0, 's', 0,
};
//*****
//产品 字符串 描述符
//*****
const unsigned char g_pProductString[] =
{
    (19 + 1) * 2,
    USB_DTYPE_STRING,
    'M', 0, 'a', 0, 's', 0, 's', 0, ' ', 0, 'S', 0, 't', 0, 'o', 0, 'r', 0,
    'a', 0, 'g', 0, 'e', 0, ' ', 0, 'D', 0, 'e', 0, 'v', 0, 'i', 0, 'c', 0,
    'e', 0,
};
//*****
// 产品 序列号 描述符
//*****
const unsigned char g_pSerialNumberString[] =
{
    (8 + 1) * 2,
    USB_DTYPE_STRING,
    '1', 0, '2', 0, '3', 0, '4', 0, '5', 0, '6', 0, '7', 0, '8', 0,
};
//*****

```

```

// 字符串描述符集合
//*****
const unsigned char * const g_pStringDescriptors[] =
{
    g_pLangDescriptor,
    g_pManufacturerString,
    g_pProductString,
    g_pSerialNumberString,
};
#define NUM_STRING_DESCRIPTOR (sizeof(g_pStringDescriptors) / \
                                sizeof(unsigned char *))
//*****
//MSC 实例，配置并为设备信息提供空间
//*****
tMSCInstance g_sMSCInstance;
//*****
//msc 设备配置
//*****
const tUSBDMSCDevice g_sMSCDevice =
{
    USB_VID_STELLARIS,
    USB_PID_MSC,
    "TI",
    "Mass Storage",
    "1.00",
    200,
    USB_CONF_ATTR_SELF_PWR,
    g_pStringDescriptors,
    NUM_STRING_DESCRIPTOR,
    {
        USBDMSCStorageOpen,
        USBDMSCStorageClose,
        USBDMSCStorageRead,
        USBDMSCStorageWrite,
        USBDMSCStorageNumBlocks
    },
    USBDMSCEventCallback,
    &g_sMSCInstance
};
#define MSC_BUFFER_SIZE 512
//*****
//键盘实例，配置并为设备信息提供空间
//*****
tHIDKeyboardInstance g_KeyboardInstance;

```

```

//*****
//键盘设备配置
//*****
const tUSBHIDKeyboardDevice g_sKeyboardDevice =
{
    USB_VID_STELLARIS,
    USB_VID_STELLARIS,
    200,
    USB_CONF_ATTR_SELF_PWR | USB_CONF_ATTR_RWAKE,
    KeyboardHandler,
    (void *)&g_sKeyboardDevice,
    0,
    0,
    &g_KeyboardInstance
};

//*****
//callback 函数
//*****
unsigned long USBDMSCEventCallback(void *pvCBData, unsigned long ulEvent,
                                   unsigned long ulMsgParam, void *pvMsgData)
{
    switch(ulEvent)
    {
        // 正在写数据到存储设备.
        case USBD_MSC_EVENT_WRITING:
        {
            break;
        }
        //读取数据.
        case USBD_MSC_EVENT_READING:
        {
            GPIOPinWrite(GPIO_PORTF_BASE, 0x10, 0x10);
            break;
        }
        //空闲
        case USBD_MSC_EVENT_IDLE:
        default:
        {
            GPIOPinWrite(GPIO_PORTF_BASE, 0x10, 0x00);
            break;
        }
    }
}

```

```

        return(0);
    }
    //*****
    //键盘 callback 函数
    //*****
    unsigned long KeyboardHandler(void *pvCBData, unsigned long ulEvent,
                                   unsigned long ulMsgData, void *pvMsgData)
    {
        switch (ulEvent)
        {
            case USB_EVENT_CONNECTED:
            {
                GPIOPinWrite(GPIO_PORTF_BASE, 0x20, 0x20);
                break;
            }
            case USB_EVENT_DISCONNECTED:
            {
                GPIOPinWrite(GPIO_PORTF_BASE, 0x20, 0x00);
                break;
            }
            case USB_EVENT_TX_COMPLETE:
            {
                break;
            }
            case USB_EVENT_SUSPEND:
            {
                break;
            }
            case USB_EVENT_RESUME:
            {
                break;
            }
            case USBD_HID_KEYB_EVENT_SET_LEDS:
            {
                break;
            }
            default:
            {
                break;
            }
        }
        return (0);
    }

```

```

}

第二步：完成 Composite 设备配置、协调：
//*****
//复合设备配置
//*****
tCompositeEntry g_psCompDevices[]=
{
    {
        &g_sMSCDeviceInfo,
        (void *)&g_sMSCDeviceInfo
    },
    {
        &g_sHIDDeviceInfo,
        (void *)&g_sHIDDeviceInfo
    }
};

#define NUM_DEVICES          (sizeof(g_psCompDevices)/sizeof(tCompositeEntry))
tCompositeInstance g_CompInstance;
unsigned long xxx[10];
tUSBDCompositeDevice g_sCompDevice =
{
    USB_VID_STELLARIS,
    0x0123,
    500,
    USB_CONF_ATTR_BUS_PWR,
    EventHandler,
    g_pStringDescriptors,
    NUM_STRING_DESCRIPTOR,
    2,
    g_psCompDevices,
    xxx,
    &g_CompInstance
};
//*****
//复合设备 callback 函数
//*****
unsigned long EventHandler(void *pvCBData, unsigned long ulEvent, unsigned long ulMsgData,
                          void *pvMsgData)
{
    unsigned long ulNewEvent;
    ulNewEvent = 1;
    switch(ulEvent)
    {
        case USB_EVENT_CONNECTED:

```



```

        {
            break;
        }
        case USB_EVENT_DISCONNECTED:
        {
            break;
        }
        case USB_EVENT_SUSPEND:
        {
            break;
        }
        case USB_EVENT_RESUME:
        {
            break;
        }

        default:
        {
            ulNewEvent = 0;
            break;
        }
    }
    if(ulNewEvent)
    {
    }
    return(0);
}

```

第三步：各子设备数据处理，主要是按键处理，U 盘功能自动调用底层驱动自动完成：

```

//系统初始化。
SysCtlLDOSet(SYSCTL_LDO_2_75V);
SysCtlClockSet(SYSCTL_XTAL_8MHZ    |    SYSCTL_SYSDIV_8    |    SYSCTL_USE_PLL    |
SYSCTL_OSC_MAIN );

SysCtlPeripheralEnable(SYSCTL_PERIPH_GPIOF);
GPIOPinTypeGPIOOutput(GPIO_PORTF_BASE, 0xf0);
GPIOPinTypeGPIOInput(GPIO_PORTF_BASE, 0x0f);
HWREG(GPIO_PORTF_BASE+GPIO_O_PUR) |= 0x0f;
// ucDMA 配置
SysCtlPeripheralEnable(SYSCTL_PERIPH_UDMA);
SysCtlDelay(10);
uDMAControlBaseSet(&sDMAControlTable[0]);
uDMAEnable();

g_ulFlags = 0;
g_eMSCState = MSC_DEV_IDLE;

```

```

//复合设备初始化
g_sCompDevice.psDevices[0].pvInstance =
    USBDMSCCompositeInit(0, &g_sMSCDevice);
g_sCompDevice.psDevices[1].pvInstance =
    USBDHIDKeyboardInit(0, &g_sKeyboardDevice);
USBDCompositeInit(0, &g_sCompDevice, DESCRIPTOR_DATA_SIZE,
    g_pucDescriptorData);
//初始化存储设备
disk_initialize(0);
while(1)
{
    USBDHIDKeyboardKeyStateChange((void *)&g_sKeyboardDevice, HID_KEYB_CAPS_LOCK,
        HID_KEYB_USAGE_A,
        (GPIOPinRead(GPIO_PORTF_BASE, 0x0f) & GPIO_PIN_0)
        ? false : true);
    USBDHIDKeyboardKeyStateChange((void *)&g_sKeyboardDevice, 0,
        HID_KEYB_USAGE_DOWN_ARROW,
        (GPIOPinRead(GPIO_PORTF_BASE, 0x0f) & GPIO_PIN_1)
        ? false : true);
    USBDHIDKeyboardKeyStateChange((void *)&g_sKeyboardDevice, 0,
        HID_KEYB_USAGE_UP_ARROW,
        (GPIOPinRead(GPIO_PORTF_BASE, 0x0f) & GPIO_PIN_2)
        ? false : true);
    USBDHIDKeyboardKeyStateChange((void *)&g_sKeyboardDevice, 0,
        HID_KEYB_USAGE_ESCAPE,
        (GPIOPinRead(GPIO_PORTF_BASE, 0x0f) & GPIO_PIN_3)
        ? false : true);
    SysCtlDelay(SysCtlClockGet()/3000);
}

```

使用上面三步就完成 Composite 设备开发。Composite 设备开发时要加入两个 lib 库函数：usb.lib 和 DriverLib.lib，在启动代码中加入 USB0DeviceIntHandler 中断服务函数。以上 Composite 设备开发完成，在 Win xp 下运行效果如下图所示：



在电脑中可以发现多了 USB MSC 设备和 HID 设备，同时还多了一个 Composite 设备。Composite 设备开发源码较多，下面只列出一部分如下：

```

#include "inc/hw_ints.h"
#include "inc/hw_memmap.h"
#include "inc/hw_gpio.h"
#include "inc/hw_types.h"

```

```

#include "inc/hw_ints.h"
#include "driverlib/sysctl.h"
#include "driverlib/gpio.h"
#include "driverlib/interrupt.h"
#include "driverlib/rom.h"
#include "driverlib/systick.h"
#include "driverlib/usb.h"
#include "driverlib/udma.h"
#include "usblib/usblib.h"
#include "usblib/usb-ids.h"
#include "usblib/device/usbdevice.h"
#include "usblib/device/usbdmhc.h"
#include "diskio.h"
#include "usbdsdcard.h"
#include "usblib/usblib.h"
#include "usblib/usbhid.h"
#include "usblib/device/usbdhid.h"
#include "usblib/device/usbdcomp.h"
#include "usblib/device/usbdhidkeyb.h"

#define DESCRIPTOR_DATA_SIZE (COMPOSITE_DHID_SIZE + COMPOSITE_DMSC_SIZE)
unsigned char g_pucDescriptorData[DESCRIPTOR_DATA_SIZE];

//声明函数原型
unsigned long USBDMSCEventCallback(void *pvCBData, unsigned long ulEvent,
                                   unsigned long ulMsgParam,
                                   void *pvMsgData);

//声明函数原型
unsigned long KeyboardHandler(void *pvCBData,
                             unsigned long ulEvent,
                             unsigned long ulMsgData,
                             void *pvMsgData);

unsigned long EventHandler(void *pvCBData, unsigned long ulEvent,
                          unsigned long ulMsgData, void *pvMsgData);

const tUSBDMSCDevice g_sMSCDevice;
//msc 状态
volatile enum
{
    MSC_DEV_DISCONNECTED,
    MSC_DEV_CONNECTED,
    MSC_DEV_IDLE,
    MSC_DEV_READ,
    MSC_DEV_WRITE,

```

```

}
g_eMSCState;
//全局标志
#define FLAG_UPDATE_STATUS      1
static unsigned long g_ulFlags;
//DMA
tDMAControlTable sDMAControlTable[64] __attribute__ ((aligned(1024)));

//*****
// 语言描述符
//*****
const unsigned char g_pLangDescriptor[] =
{
    4,
    USB_DTYPE_STRING,
    USBShort(USB_LANG_EN_US)
};
//*****
// 制造商 字符串 描述符
//*****
const unsigned char g_pManufacturerString[] =
{
    (17 + 1) * 2,
    USB_DTYPE_STRING,
    'T', 0, 'e', 0, 'x', 0, 'a', 0, 's', 0, ' ', 0, 'I', 0, 'n', 0, 's', 0,
    't', 0, 'r', 0, 'u', 0, 'm', 0, 'e', 0, 'n', 0, 't', 0, 's', 0,
};
//*****
//产品 字符串 描述符
//*****
const unsigned char g_pProductString[] =
{
    (19 + 1) * 2,
    USB_DTYPE_STRING,
    'M', 0, 'a', 0, 's', 0, 's', 0, ' ', 0, 'S', 0, 't', 0, 'o', 0, 'r', 0,
    'a', 0, 'g', 0, 'e', 0, ' ', 0, 'D', 0, 'e', 0, 'v', 0, 'i', 0, 'c', 0,
    'e', 0,
};
//*****
// 产品 序列号 描述符
//*****
const unsigned char g_pSerialNumberString[] =
{
    (8 + 1) * 2,

```

```

    USB_DTYPE_STRING,
    '1', 0, '2', 0, '3', 0, '4', 0, '5', 0, '6', 0, '7', 0, '8', 0
};
//*****
// 字符串描述符集合
//*****
const unsigned char * const g_pStringDescriptors[] =
{
    g_pLangDescriptor,
    g_pManufacturerString,
    g_pProductString,
    g_pSerialNumberString,
};
#define NUM_STRING_DESCRIPTOR (sizeof(g_pStringDescriptors) / \
                                sizeof(unsigned char *))
//*****
//MSC 实例，配置并为设备信息提供空间
//*****
tMSCInstance g_sMSCInstance;
//*****
//msc 设备配置
//*****
const tUSBDMSCDevice g_sMSCDevice =
{
    USB_VID_STELLARIS,
    USB_PID_MSC,
    "TI",
    "Mass Storage",
    "1.00",
    200,
    USB_CONF_ATTR_SELF_PWR,
    g_pStringDescriptors,
    NUM_STRING_DESCRIPTOR,
    {
        USBDMSCStorageOpen,
        USBDMSCStorageClose,
        USBDMSCStorageRead,
        USBDMSCStorageWrite,
        USBDMSCStorageNumBlocks
    },
    USBDMSCEventCallback,
    &g_sMSCInstance
};
#define MSC_BUFFER_SIZE 512

```

```

//*****
//键盘实例，配置并为设备信息提供空间
//*****
tHIDKeyboardInstance g_KeyboardInstance;
//*****
//键盘设备配置
//*****
const tUSBHIDKeyboardDevice g_sKeyboardDevice =
{
    USB_VID_STELLARIS,
    USB_VID_STELLARIS,
    200,
    USB_CONF_ATTR_SELF_PWR | USB_CONF_ATTR_RWAKE,
    KeyboardHandler,
    (void *)&g_sKeyboardDevice,
    0,
    0,
    &g_KeyboardInstance
};
//*****
//复合设备配置
//*****
tCompositeEntry g_psCompDevices[]=
{
    {
        &g_sMSCDeviceInfo,
        (void *)&g_sMSCDeviceInfo
    },
    {
        &g_sHIDDeviceInfo,
        (void *)&g_sHIDDeviceInfo
    }
};
#define NUM_DEVICES (sizeof(g_psCompDevices)/sizeof(tCompositeEntry))
tCompositeInstance g_CompInstance;
unsigned long xxx[10];
tUSBCompositeDevice g_sCompDevice =
{
    USB_VID_STELLARIS,
    0x0124,
    500,
    USB_CONF_ATTR_BUS_PWR,
    EventHandler,
    g_pStringDescriptors,

```

```

    NUM_STRING_DESCRIPTOR,
    2,
    g_psCompDevices,
    xxx,
    &g_CompInstance
};

//*****
//callback 函数
//*****
unsigned long USBDMSCEventCallback(void *pvCBData, unsigned long ulEvent,
                                   unsigned long ulMsgParam, void *pvMsgData)
{
    switch(ulEvent)
    {
        // 正在写数据到存储设备.
        case USBDMSC_EVENT_WRITING:
        {
            break;
        }
        //读取数据.
        case USBDMSC_EVENT_READING:
        {
            GPIOWrite(GPIO_PORTF_BASE, 0x10, 0x10);
            break;
        }
        //空闲
        case USBDMSC_EVENT_IDLE:
        default:
        {
            GPIOWrite(GPIO_PORTF_BASE, 0x10, 0x00);
            break;
        }
    }
    return(0);
}

//*****
//键盘 callback 函数
//*****
unsigned long KeyboardHandler(void *pvCBData, unsigned long ulEvent,
                             unsigned long ulMsgData, void *pvMsgData)
{
    switch (ulEvent)

```

```

{
    case USB_EVENT_CONNECTED:
    {
        GPIOPinWrite(GPIO_PORTF_BASE, 0x20, 0x20);
        break;
    }
    case USB_EVENT_DISCONNECTED:
    {
        GPIOPinWrite(GPIO_PORTF_BASE, 0x20, 0x00);
        break;
    }
    case USB_EVENT_TX_COMPLETE:
    {
        break;
    }
    case USB_EVENT_SUSPEND:
    {
        break;
    }
    case USB_EVENT_RESUME:
    {
        break;
    }
    case USBD_HID_KEYB_EVENT_SET_LEDS:
    {
        break;
    }
    default:
    {
        break;
    }
}

return (0);
}

//*****
//复合设备 callback 函数
//*****
unsigned long EventHandler(void *pvCBData, unsigned long ulEvent, unsigned long ulMsgData,
    void *pvMsgData)
{
    unsigned long ulNewEvent;
    ulNewEvent = 1;

```



```

switch(ulEvent)
{
    case USB_EVENT_CONNECTED:
    {
        break;
    }
    case USB_EVENT_DISCONNECTED:
    {
        break;
    }
    case USB_EVENT_SUSPEND:
    {
        break;
    }
    case USB_EVENT_RESUME:
    {
        break;
    }

    default:
    {
        ulNewEvent = 0;
        break;
    }
}

if(ulNewEvent)
{
}

return(0);
}

//*****
//主函数
//*****

int main(void)
{
    //系统初始化。
    SysCtlLD0Set(SYSCTL_LDO_2_75V);
    SysCtlClockSet(SYSCTL_XTAL_8MHZ    |   SYSCTL_SYSDIV_8    |   SYSCTL_USE_PLL    |
SYSCTL_OSC_MAIN );
    SysCtlPeripheralEnable(SYSCTL_PERIPH_GPIOF);
    GPIOPinTypeGPIOOutput(GPIO_PORTF_BASE, 0xf0);
    GPIOPinTypeGPIOInput(GPIO_PORTF_BASE, 0x0f);
    HWREG(GPIO_PORTF_BASE+GPIO_O_PUR) |= 0x0f;
    // ucDMA 配置

```

```

SysCtlPeripheralEnable(SYSCTL_PERIPH_UDMA);
SysCtlDelay(10);
uDMAControlBaseSet(&sDMAControlTable[0]);
uDMAEnable();

g_ulFlags = 0;
g_eMSCState = MSC_DEV_IDLE;
//复合设备初始化
g_sCompDevice.psDevices[0].pvInstance =
    USBDMSCCompositeInit(0, &g_sMSCDevice);
g_sCompDevice.psDevices[1].pvInstance =
    USBDHIDKeyboardInit(0, &g_sKeyboardDevice);
USBDCompositeInit(0, &g_sCompDevice, DESCRIPTOR_DATA_SIZE,
    g_pucDescriptorData);
//初始化存储设备
disk_initialize(0);
while(1)
{
    USBDHIDKeyboardKeyStateChange((void *)&g_sKeyboardDevice, HID_KEYB_CAPS_LOCK,
        HID_KEYB_USAGE_A,
        (GPIOPinRead(GPIO_PORTF_BASE, 0x0f) & GPIO_PIN_0)
        ? false : true);
    USBDHIDKeyboardKeyStateChange((void *)&g_sKeyboardDevice, 0,
        HID_KEYB_USAGE_DOWN_ARROW,
        (GPIOPinRead(GPIO_PORTF_BASE, 0x0f) & GPIO_PIN_1)
        ? false : true);
    USBDHIDKeyboardKeyStateChange((void *)&g_sKeyboardDevice, 0,
        HID_KEYB_USAGE_UP_ARROW,
        (GPIOPinRead(GPIO_PORTF_BASE, 0x0f) & GPIO_PIN_2)
        ? false : true);
    USBDHIDKeyboardKeyStateChange((void *)&g_sKeyboardDevice, 0,
        HID_KEYB_USAGE_ESCAPE,
        (GPIOPinRead(GPIO_PORTF_BASE, 0x0f) & GPIO_PIN_3)
        ? false : true);
    SysCtlDelay(SysCtlClockGet()/3000);
}
}

```