15.2　USB2.0 程序设计实现

由于 USB 接口支持的设备众多，一个 USB 设备的程序会涉及多种协议的描述，因此程序设计需要分层编写。这样做的好处是对于诸如 USB2.0 这样设备都需要使用的协议的源文件仅需较小的修改即可使用，并且可以依据不同的设备将其他不同的协议规范的源文件添加进入工程以满足需要。

本文档介绍如何编程实现 USB2.0 规范的两层，一是 USB 层，二是 USB 标准请求层，本次程序设计主干框如图 15.28 所示。

图 15.28　程序的主干结构

15.2.1 的第 1 小节介绍如何控制 USB 控制寄存器，对应的函数为 usb_read_reg( ) 和 usb_write_reg( )。

15.2.1 的第 2 小节介绍如何通过 FIFO 寄存器来实现主机和设备之间的数据收发，对应的函数为 usb_read_fifo( ) 和 usb_write_fifo( )。

15.2.1 的第 3 小节介绍 USB 控制传输类型的实现方法，对应的函数为 usb_ctrl_in( ) 和 usb_ctrl_out( )。

15.2.2 的第 1 小节介绍的 USB 初始化的重要工作是打开 USB 有关的中断，对应的函数为 usb_init( )。

15.2.2 的第 2 小节介绍讲述如果触发了中断，设备如何依据不同的中断类型进行相应的操作，1.5 小结对应的函数为 usb_isr( )，对应的中断号为 25。

15.2.2 的第 3 小节介绍 USB 通信中最开始的 SETUP 过程如何实现，对应的函数为 usb_setup( )

15.2.2 的第 4 和第 5 小结分别介绍 USB 标准请求和特定类请求的实现方法，对应的函数为 usb_req_std( ) 和 usb_req_class( )。

控制传输的主要流程为：主机向设备不断发送 SETUP 数据包，设备接收 SETUP 数据包解析并依据解析的内容进行响应，响应内容可以是设备软件状态的设置，或者对主机回复特定的信息。该过程结束后，若设备为 HID 设备，则可能进入中断传输阶段，该阶段为设备的主要工作状态。

15.2.1　USB 的寄存器操作和数据传输

1. USB 控制寄存器操作方式

1. 读取 USB 控制寄存器方式：

先判断 USBADR 寄存器的最高位是否为 1，启动 USB 控制寄存器的读操作，地址由 US-

BADR 设定。此时将需要操作的 USB 控制寄存器的地址，通过按位或上 0x80 的方式传给 USBADR 寄存器，然后返回 USBDAT 寄存器的内容即可，USBDAT 的内容即为需要读取的 USB 控制寄存器的内容，其具体的写法代码如代码清单 15-1 所示。

代码清单 15-1　读取 USB 控制器的 C 代码描述

```
/************************读取 USB 控制寄存器的内容**********************/
BYTE usb_read_reg(BYTE addr)
{
    BYTE dat;
    while (USBADR & 0x80);          //按位与，直到读取忙时跳出循环
    USBADR = addr | 0x80;           //将传入的 USB 控制寄存器的地址赋给 USBADR 寄存器
    while (USBADR & 0x80);          //按位与，直到读取忙时跳出循环
    dat = USBDAT;                   //读取 USB 控制寄存器内容
    return dat;
}
```

2. 写 USB 控制寄存器方式：

先判断 USBADR 寄存器的最高位是否为 1，若为 1 表示 USBDAT 寄存器中的数据无效，USB 正在读取间接寄存器。此时将需要操作的 USB 控制寄存器的地址，通过按位与上 0x7f 的方式赋值给 USBADR 寄存器，然后将传入的 USB 控制寄存器的值赋给 USBDAT 寄存器，即完成对 USB 控制寄存器的写操作，其具体的写法代码如代码清单 15-2 所示。

代码清单 15-2　写 USB 控制寄存器的 C 语言代码

```
/************************写 USB 控制寄存器**********************/
void usb_write_reg(BYTE addr, BYTE dat)
{
    while (USBADR & 0x80);                  //按位与，直到读取忙时跳出循环
    USBADR = addr & 0x7f;                   //最高位清零 USBDAT 寄存器中的数据有效
    USBDAT = dat;                           //将传入的数值赋给 USBDAT 寄存器
}
```

2. FIFO 的操作方式

FIFO 寄存器是 USB 数据传输时的核心寄存器。在 USB 通信中，主机向设备发送的数据，是通过 FIFO 寄存器以字节为单位读入的；而通过设备向主机发送的数据，同样也是通过 FIFO 寄存器以字节为单位发送的。

1. 读取 FIFO 寄存器方式（接收主机的数据：传输类型 OUT）：

通过读取 COUNT0 寄存器以获取端点 0 最后接收到的 OUT 数据包的长度，依据数据包的长度以字节为单位读入传入的指针所指向的地址间中，即完成一次数据的接收。其具体的写法代码如代码清单 15-3 所示。

代码清单 15-3　读取 FIFO 的 C 语言代码

```
/*************************读取 FIFO*****************************/
BYTE usb_read_fifo(BYTE fifo, BYTE *pdat)
{
    BYTE cnt;
    BYTE ret;
    ret = cnt = usb_read_reg(COUNT0);//获取端点 0 最后接收到的 OUT 数据包长度
    while (cnt--)
    {
        *pdat++ = usb_read_reg(fifo);
    }
    return ret;                          //返回端点 0 最后接收到的 OUT 数据包长度(单位:字节)
}
```

2. 写 FIFO 寄存器方式(向主机发送数据:传输类型 IN):

直接依据设备需要发送的数据大小,以字节为单位写入 FIFO 中,即完成一次数据的发送。其具体的写法代码如代码清单 15-4 所示。

代码清单 15-4　写 FIFO 的 C 语言代码

```
/*****************************写 FIFO ******************************/
void usb_write_fifo(BYTE fifo, BYTE *pdat, BYTE cnt)
{
    while (cnt--)
    {
        usb_write_reg(fifo, *pdat++);
    }
}
```

3. USB 控制传输类型实现

首先,控制传输仅使用端点 0 进行数据的传输,其依据传输方向分为 IN 类型和 OUT 类型。与前面所讲的 IN 类型与 OUT 类型相区别的是,其必须通过端点 0 传输数据,且该传输仅在标准请求以及特定类请求中使用。

1. 控制 IN 类型:设备向主机发送数据

首先将 INDEX 寄存器指向端点 0,然后读取 CSR0 寄存器的内容,依据 CSR 寄存器的 IPRDY 为是否为 1 来判定是否已经将 IN 数据包装入了 FIFO,若已经完成装入,则直接返回,否则控制需要向主机发送的数据以字节为单位装入 FIFO 中进行发送,由于端点 0 大小 EP0_SIZE 有限,如果发送的数据大小超过端点 0 大小,则需要分批次发送,其具体的写法代码如代码清单 15-5 所示。

代码清单 15-5　USB 控制传输 IN 类型的 C 语言代码

```
/************************USB 控制传输 IN 类型************************/
void usb_ctrl_in()
{
```

```
    BYTE csr;                                   //创建 8 位 csr 变量
    BYTE cnt;                                   //创建 8 位 cnt 变量
    usb_write_reg(INDEX, 0);                    //INDEX 寄存器指向端点 0
    csr = usb_read_reg(CSR0);                   //csr 变量读取 CSR0 寄存器的内容
    if (csr & IPRDY) return;                    //如果 IN 数据包已经装入 FIFO,则直接返回

    cnt = Ep0State.wSize > EP0_SIZE ? EP0_SIZE : Ep0State.wSize;
    usb_write_fifo(FIFO0, Ep0State.pData, cnt);
                                                //将结构体 Ep0State 中的 pDATA 数据写入 FIFO
    Ep0State.wSize -= cnt;                      //wsize 减去 cnt 后的值
    Ep0State.pData += cnt;                      //pdata 加上 cnt 后的值
    if (Ep0State.wSize == 0)                    //如果数据包大小为 0
    {
        usb_write_reg(CSR0, IPRDY | DATEND);
        //置 1IPRDY 和 DATEND 表示发送了一个 0 长的数据包(发送完毕)
        Ep0State.bState = EPSTATE_IDLE;//端点 0 的状态指向 IDLE 状态
    }
    else
    {
        usb_write_reg(CSR0, IPRDY);
        //如果数据包大小不为 0 则置 1IPRDY,表示要发送的数据包装入到端点 0 的 FIFO(未发
          完还要接着发)
    }
```

2. 控制 OUT 类型:主机向设备发送数据,设备接收数据

首先将 INDEX 寄存器指向端点 0,然后读取 CSR0 寄存器的内容,依据 CSR 寄存器的 OPRDY 为是否为 0 来判定 OUT 数据包是否已经收到,若收到则返回。否则调用读取 FIFO 寄存器的函数对数据进行读取,同理由于端点 0 大小 EP0_SIZE 有限,如果读取的数据大小超过端点 0 大小,则需要分批次读取,其具体的写法代码如代码清单 15-6 所示。

代码清单 15-6　USB 控制传输 OUT 类型的 C 语言代码

```
/************************USB 控制传输 OUT 类型 ************************/
void usb_ctrl_out()
{
    BYTE csr;                                   //创建 8 位 csr 变量
    BYTE cnt;                                   //创建 8 位 cnt 变量
    usb_write_reg(INDEX, 0);                    //INDEX 寄存器指向端点 0
    csr = usb_read_reg(CSR0);                   //csr 变量读取 CSR0 寄存器的内容
    if (!(csr & OPRDY)) return;                 //如果 OUT 数据包已经收到,则退出
    cnt = usb_read_fifo(FIFO0, Ep0State.pData); //cnt=FIFO 数据包的大小
    Ep0State.wSize -= cnt;                      //wsize 减去 cnt 后的值
    Ep0State.pData += cnt;                      //pdata 加上 cnt 后指向的地址
```

```
        if (Ep0State.wSize == 0)                    //如果全部读取完毕
        {
            usb_write_reg(CSR0, SOPRDY | DATEND); //清零 OPRDY,表示接收完最后一个数据包
            Ep0State.bState = EPSTATE_IDLE;         //端点 0 的状态指向 IDLE 状态
        }
        else
        {
            usb_write_reg(CSR0, SOPRDY);
            //如果数据包大小不为 0 则清零 OPRDY,表示完成一个 OUT 数据包的接收(未读完还需接
            着读)
        }
    }
```

15.2.2 USB SETUP 阶段实现

1. USB 初始化的工作

USB 初始化最重要的工作是初始化 USB 设备状态,器包括 USB 设备的硬件状态,还有软件状态,以下为 USB 初始化的内容:

1. 硬件状态设置(寄存器)

1) FADDR 寄存器:使最后的 UADDR 地址生效,更新 USB 的功能地址

2) POWER 寄存器:强制产生异步 USB 复位,使能挂起检测

3) INTRIN1E:打开所有 IN 端点中断

4) INTROUT1E 寄存器:打开所有 OUT 端点中断

5) INTRUSBE 寄存器:允许 USB 复位信号中断,允许 USB 恢复信号中断,允许 USB 挂起信号中断

6) POWER 寄存器:使能挂起检测,当检测到总线上的挂起信号,USB 将进入挂起方式

2. 软件状态设置

1) 设备状态:默认

2) 端点 0 状态:IDLE

3) IN 端点状态:0

3) OUT 端点状态:0

具体写法如代码清单 15-7 所示。

代码清单 15-7　USB 初始化 C 语言代码

```
/************************USB 初始化*********************/
void usb_init()
{
    USBCLK = 0x00;
    USBCON = 0x90;

    usb_write_reg(FADDR, 0x00);
    //使最后的 UADDR 地址生效,更新 USB 的功能地址,FADDR 低 7 位为保存 USB 的七位功能地址
    usb_write_reg(POWER, 0x09);              //强制产生异步 USB 复位,使能挂起检测
```

```
    usb_write_reg(INTRIN1E, 0x3f);                    //打开所有 IN 端点中断
    usb_write_reg(INTROUT1E, 0x3f);                   //打开所有 OUT 端点中断
    usb_write_reg(INTRUSBE, 0x07);
    //允许 USB 复位信号中断,允许 USB 恢复信号中断,允许 USB 挂起信号中断
    usb_write_reg(POWER, 0x01);
    //使能挂起检测,当检测到总线上的挂起信号,USB 将进入挂起方式

    DeviceState = DEVSTATE_DEFAULT;                   //设置设备起始状态默认
    Ep0State.bState = EPSTATE_IDLE;                   //设置端点 0 状态为 IDLE 状态
    InEpState = 0x00;                                 //设置 IN 端点为
    OutEpState = 0x00;                                //设置 OUT 端点为

    EUSB = 1;
}
```

2. USB 中断处理内容

会触发进入中断处理函数的中断:参照 USB 初始化一节中 USB 有关的中断已打开的那些中断。

中断处理的内容:

1) 有关电源的中断

- 如果中断来自恢复信号,则跳转至 USB 恢复函数
- 如果中断来自复位信号,则跳转至 USB 复位函数
- 如果中断来自挂起信号,则跳转至 USB 挂起函数

2) 有关 USB 端点的中断

(1) 如果中断来自端点 0,即中断来自控制端点,则进入 USB 的 SETUP 阶段,SETUP 阶段是 USB 重要的通信阶段,解释 SETUP 阶段(重点),其具体写法详见 USB SETUP 阶段的实现一节;

(2) 如果中断来自其他端点,则进入对应的 USB 端点函数,USB 端点函数就是 USB 在 SETUP 阶段后的主要工作的内容,可以理解为 USB 设备的一般的使用功能就在对应的端点函数中(重点),例如:键盘在工作时使用 IN1 端点向主机发送键盘码值的报告数据,而主机仅需将对应的报告数据翻译成对应的动作即可。

具体写法如代码清单 15-8 所示。

代码清单 15-8　USB 中断句柄/中断服务程序的 C 语言代码

```
void usb_isr() interrupt USB_VECTOR     //USB 中断向量
{
    BYTE intrusb;                       //usb 中断
    BYTE intrin;                        //in 中断
    BYTE introut;                       //out 中断
    BYTE adrTemp;                       //临时保存地址变量
    adrTemp = USBADR;
    //USBADR 现场保存,避免主循环里写完 USBADR 后产生中断,在中断里修改了 USBADR 内容

    intrusb = usb_read_reg(INTRUSB);    //usb 中断=USB 电源中断标志
```

```
    intrin = usb_read_reg(INTRIN1);     //USB 端点 IN 中断标志位
    introut = usb_read_reg(INTROUT1);  //USB 端点 OUT 中断标志位

    if (intrusb & RSUIF) usb_resume();  //若恢复信号中断有效,则 USB 恢复(空的不用看)
    if (intrusb & RSTIF) usb_reset();    //若复位信号中断有效,则 USB 复位

    if (intrin & EP0IF) usb_setup();
    //若端点 0 的 IN/OUT 中断有效,USB 进入 setup 阶段

#ifdef EN_EP1IN
    if (intrin & EP1INIF) usb_in_ep1(); //若端点 1 的 in 中断有效,则进入对应函数
#endif
.
.                                       //后续不同 IN 端点的写法与端点 1IN 相同
.
#ifdef EN_EP1OUT
    if (introut & EP1OUTIF) usb_out_ep1();
                                        //若端点 1 的 out 中断有效,则进入对应的函数
#endif
.
.                                       //后续不同 OUT 端点的写法与端点 1OUT 相同
.
.
    if (intrusb & SUSIF) usb_suspend();
    USBADR = adrTemp;                   //USBADR 现场恢复
}
```

3. USB SETUP 阶段的实现

SETUP 阶段最重要的任务:一是依据主机向设备发送的不同请求,从而对设备进行对应的硬件状设置,以及软件状态设置;二是主机向设备发送请求后,可能需要设备回复某些信息,这些信息以数据包的形式从设备发送给主机。

例如:一个完整的设备描述符信息会在 SETUP 阶段中,主机发送向设备发送 GET_DESCRIPTOR 请求后,由设备将设备描述符信息打包发送给主机。

SETUP 阶段的具体内容为:

1. 先将端点定位到控制端点 0(因为在 USB 协议中,控制端点 0 就是服务于 SETUP 阶段中的,其执行着控制传输的任务),之后我们读取 USB 端点 0 控制状态寄存器 CSR0 中的内容,依据 CSR0 中的 SDSTL 位来设置端点零的状态,SDSTL 的状态表示我们是否已经完成 STALL 信号的发送,依据 USB 协议内容:如果 STALL 信号发送完成,则我们可以设置端点 0 的状态为 EPSTATE_IDLE 状态,此状态表示我们可以接收来自主机的请求并进行请求处理。如果 SETUP 阶段结束,我们需要清零 CSR0 寄存器的 SSUEND 位

2. 请求分为三大类:1USB 标准请求,对应的函数为 usb_req_std( )2 特定类请求,对应的函数为 usb_req_class( )3 厂商请求,对应的函数为 usb_req_vendor( )。其中厂商请求在此次实验

中不会涉及,故如果主机发送该请求,我们默认将 USB 设备在 SETUP 阶段挂起。如果请求不是三大类中的任意一种,则我们默认将设备挂起。USB 标准请求以及特定类请求参照 USB 标准请求一节,以及类请求一节。

3. 在 SETUP 阶段还可能存在单独的数据过程,该过程可能是 CTRL_IN 或者 CTRL_OUT 过程。

SETUP 具体的写法可以参照如下代码清单 15-9 所示。

代码清单 15-9　USB 的 SETUP 阶段的 C 语言代码

```
/*************************SETUP 阶段*******************************/
void usb_setup()
{
    BYTE csr;                                          //csr 寄存器变量
    usb_write_reg(INDEX, 0);                           //选择端点 0
    csr = usb_read_reg(CSR0);                          //读取 USB 端点 0 控制状态寄存器
    if (csr & STSTL)                                   //如果 STALL 信号发送完成
    {
        usb_write_reg(CSR0, csr & ~SDSTL);             //未接收到错误条件,SDSTL 清零
        Ep0State.bState = EPSTATE_IDLE;                //端点 0 设置 IDLE 状态
    }
    if (csr & SUEND)                                   //如果 SETUP 阶段结束
    {
        usb_write_reg(CSR0, csr & ~SSUEND);            //清零 SSUEND
    }
    switch (Ep0State.bState)                           //依据端点 0 状态进入状态机
    {
    case EPSTATE_IDLE:                                 //IDLE 状态
        if (csr & OPRDY)                               //如果收到一个 OUT 数据包
        {
            usb_read_fifo(FIFO0, (BYTE *)&Setup);//从 FIFO0 中读取 Setup 信息
            Setup.wLength = reverse2(Setup.wLength);
            //wlength 字节高 8 位和低 8 位交换
            switch (Setup.bmRequestType & REQUEST_MASK)
            //依据 setup 中请求数据的类型进入状态机
            {
            case STANDARD_REQUEST:                     //标准请求(usb 请求核心)
                usb_req_std();
                break;
            case CLASS_REQUEST:                        //类请请求(hid 请求核心)
                usb_req_class();
                break;
            case VENDOR_REQUEST:                       //厂商请求
                usb_req_vendor();                      //实验不涉及厂商请求故不作内容编写
                break;
            default:                                   //默认
```

```
                usb_setup_stall();//建立阶段挂起
                return;
            }
        }
        break;
    case EPSTATE_DATAIN://数据输入状态
        usb_ctrl_in();
        break;
    case EPSTATE_DATAOUT://数据输出状态
        usb_ctrl_out();
        break;
    }
}
```

4. USB 标准请求

本文此处开始讲解 USB 标准请求层,该层对应 usb_req_std.c 源文件。此节强烈建议配合提供的例程进行阅读,这样不仅有利于理解协议内容,更可以理解如何将协议内容转化为计算机编程语言进行实现。

1) USB 标准设备请求数据结构

在协议中 USB 标准设备请求数据结构被称为 SETUP 数据结构,这也是为什么在编程时将其定义为一个 SETUP 结构体的原因。在程序设计中,可以知道 USB 标准的数据结构需要从 USB 控制寄存器中的 FIFO 中读取出来,USB 标准设备请求数据结构十分重要,我们可以通过读取到的 USB 标准设备请求数据结构对进行解析,从而得到不同的信息,我们可以根据这些信息对不同的 USB 设备请求进行响应,表 15.29 给出类 USB 标准设备请求的数据结构。

表 15.29 SETUP 数据结构

偏移量	字段	大小/字节	取值	含义
0	bmRequestType	1		请求特性 D7:数据传输方向 0=从主机到设备 1=从设备到主机 D6-D5:请求的类型 0=标准类型 1=类类型 2=厂商类型 3=保留 D4-0:请求的接收者 0=设备 1=接口 2=端点 3=其他 其余:保留

续表

偏移量	字段	大小/字节	取值	含义
1	bRequest	1	数值	请求代码
2	wValue	2	数值	该域的意义由具体请求决定
4	wIndex	2	索引或偏移量	该域的意义由具体请求决定
6	wLength	2	字节数	数据过程所需要传输的字节数

在程序设计时,我们可以通过定义如下 SETUP 结构体来定义一个 USB 标准设备请求数据结构,如代码清单 15-10 所示。

代码清单 15-10　USB 标准设备请求的数据结构

```
typedef struct                          //USB 标准设备请求的数据结构
{
    BYTE    bmRequestType;              //请求类型(子级)
    BYTE    bRequest;                   //请求类型(父级)

    BYTE    wValueL;            //值低 8 位
    BYTE    wValueH;            //值高 8 位
    BYTE    wIndexL;            //指针低 8 位
    BYTE    wIndexH;            //指针高 8 位
    WORD    wLength;            //长度
}SETUP;                         //(获取方式:从 FIFO 中读取)
```

2) USB 标准请求

在以上 SETUP 结构体中,USB 标准请求信息包含在 bRequest 字段中,通常来讲,所有 USB 设备需要都需要能够支持 USB 标准请求,对于标准请求的分类,不同标准请求对应的 bRequest 字段的值,还有不同标准请求功能如表 15.30 所示。

表 15.30　标准请求及其功能

请求	bRequest 字段值	功能
ClearFeature	1	清除设备、接口的某种特征(或性能)
GetConfigration	8	获取指定设备当前的配置值
GetDescriptor	6	获取设备的某种标准描述符
GetInterface	10	获取设备接口当前工作的选择设置值
GetStatus	0	获取设备、接口或端点的某种状态
SetAddress	5	为设备设置唯一的地址
SetConfiguration	9	激活设备的某个配置
SetDescriptor	7	主机会更新或创立描述符
SetFeature	3	主机启用一个在设备、接口或端点上的特征
SetInterface	11	主机激活设备的某个接口的设置值
SynchFrame	12	在实时传输中,用于同步某个帧开始传输序列

对于标准请求，根据 bRequest 字段值，我们可以使用状态机来进入不同的请求，其写法具体如代码清单 15-11 所示。

代码清单 15-11　标准请求的定义

```
#define GET_STATUS              0x00
#define CLEAR_FEATURE           0x01
#define SET_FEATURE             0x03
#define SET_ADDRESS             0x05
#define GET_DESCRIPTOR          0x06
#define SET_DESCRIPTOR          0x07
#define GET_CONFIGURATION       0x08
#define SET_CONFIGURATION       0x09
#define GET_INTERFACE           0x0A
#define SET_INTERFACE           0x0B
#define SYNCH_FRAME             0x0C
void usb_req_std()
{
    switch (Setup.bRequest)          //依据 SETUP 请求类型进入状态机
    {
    case GET_STATUS:
        usb_get_status();            //获取状态
        break;
    case CLEAR_FEATURE:              //清除特征
        usb_clear_feature();
        break;
    case SET_FEATURE:                //设置特征
        usb_set_feature();
        break;
    case SET_ADDRESS:                //设置地址
        usb_set_address();
        break;
    case GET_DESCRIPTOR:             //获取描述符
        usb_get_descriptor();
        break;
    case SET_DESCRIPTOR:             //设置描述符
        usb_set_descriptor();
        break;
    case GET_CONFIGURATION:          //获取配置值
        usb_get_configuration();
        break;
    case SET_CONFIGURATION:          //激活某个配置
        usb_set_configuration();
        break;
```

```
        case GET_INTERFACE:          //获取接口
            usb_get_interface();
            break;
        case SET_INTERFACE:          //设置接口
            usb_set_interface();
            break;
        case SYNCH_FRAME:            //帧同步
            usb_synch_frame();
            break;
        default:
            usb_setup_stall();       //建立阶段挂起
            return;
        }
    }
```

表 15.31 将列出所有标准 USB 请求的结构和需要传输的数据

表 15.31　标准 USB 请求内容

bRequest 字段	wValue 字段	wIndex 字段	wLength 字段	数据过程
CLEAR_FEATURE	特性选择	0 接口号 端点号	0	没有
GET_CONFIGURATION	0	0	1	配置值
GET_DESCRIPTOR	描述符类型和索引	0 或者语言 ID	描述符长度	描述符
GET_INTERFACE	0	接口号	1	备用接口号
GET_STATUS	0	0 接口号 端点号	2	设备、接口或者端点状态
SET_ADDRESS	设备地址	0	0	没有
SET_CONFIGURATION	配置值	0	0	没有
SET_DESCRIPTOR	描述符类型和索引	0 或者语言 ID	描述符的长度	描述符
SET_FEATURE	特性选择	0 接口号 端点号	0	没有
SET_INTERFACE	备用接口号	接口号	0	没有
SYNCH_FRAME	0	端点号	2	帧号

3）标准请求函数的参考写法

为了标准起见，本书的标准请求函数分为三个步骤：1. 特殊情况处理 2. 依据状态发包 3. 结束状态，即进行数据操作（如发送数据，或者接收数据，或者不进行数据操作）。此部分的代

码请参考工程 HID 协议范例文件 usb_req_std.c 内容。

(1) GET_STATUS

依据以表 15.32,可以对该请求的特殊情况进行设计

表 15.32 GET_STATUS 内容

bmRequestType	bRequest	wValue	wIndex	wLength	数据过程
0x80 0x81 0x82	GET_STATUS	0	0 接口号 端点号	2	设备、接口或者端点状态

第 1 步:特殊情况的处理,可以加入一判断语句,将不满足上表 GET_STATUS 的所有 USB 标准设备请求数据结构进行统一处理,其意义在于当所设计的程序错误进入该请求时,程序可以通过挂起的方式以终止该请求的处理。

第 2 步:依据 USB 协议,需要选择不同的数据包发送给主机此处仅依据本次模拟 HID 实验对其中出现的几个动作进行程序的编写。如表 15.33,根据 bmRequestType 进行不同的动作:

表 15.33 bmRequestType 表

bmRequestType	值	功能
DEVICE_RECIPIENT	0x80	获取设备状态
INTERFACE_RECIPIENT	0x81	获取接口状态
ENDPOINT_RECIPIENT	0x82	获取端点状态

如果 bmRequestType 不包含以上状态,默认状态都为挂起状态。

第 3 步:进入标准请求结束状态(进行数据过程)。例如,bmRequestType 为 DEVICE_RECIPIENT,这时在步骤 2 中软件已经依据 USB 协议内容将 PACKET0 数据包装入了数据缓冲区,然后软件再通过调用 usb_setup_in(),完成数据包的发送。

注意:usb_setup_in()的内容是:先将控制端点 0 设置为 DATAIN 状态,控制寄存器 CSR0 的 SPORDY 写 1 以清零 OPRDY,随后调用 usb_ctrl_in()进行数据发送,即使用控制 IN 类型向主机发送数据包。

(2) CLEAR_FEATURE

如表 15.34 所示,可以对该请求的特殊情况进行设计

表 15.34 CLEAR_FEATURE 内容

bmRequestType	bRequest	wValue	wIndex	wLength	数据过程
0x00 0x01 0x02	CLEAR_FEATURE	特性选择	0 接口号 端点号	0	无

第 1 步:特殊情况的处理,与 GET_STATUS 请求的设计思路一致,依旧是特殊情况处理。如果 USB 标准设备请求数据结构不满足表 15.34 所有条件,则挂起。

第 2 步:由于本次实验仅涉及到依据 wIndex 指向的端点号寄存器操作,故仅对端点号的

情况进行处理;如果端点定义成 IN 端点,需要将 INDEX 寄存器根据端点号指向对应的端点,如端点号为 1,则 INDEX 寄存器需要指向端点 1,之后我们操作 INCSR1 寄存器的 CLRDT 位置 1(意思为数据切换位复位到 0),最后将端点索引器 INDEX 指向端点 0,以便后续的控制操作。同理,端点定义成 OUT 端点,则需要操作的是 OUTCSR1 的 CLRDT 位置 1,使得数据切换位复位到 0。如果未设置端点的 IN 状态或 OUT 状态,则默认挂起。

第 3 步:进入请求结束状态:端点 0 设置成空闲状态,寄存器 CSR0 的 SPORDY 位以及 DATEND 位置位,表示处理完成从端点 0 接收到的数据包,且数据结束。

(3) SET_FEATURE

依据表 15.35,可以对该请求的特殊情况进行设计

表 15.35　SET_FEATURE 内容

bmRequestType	bRequest	wValue	wIndex	wLength	数据过程
0x00 0x01 0x02	SET_FEATURE	特性选择	0 接口号 端点号	0	无

第 1 步:特殊情况的处理,与 GET_STATUS 请求的设计思路一致,依旧是特殊情况处理。如果 USB 标准设备请求数据结构不满足表 15.35 所有条件,则挂起。

第 2 步:由于本次实验仅涉及到依据 wIndex 指向的端点号寄存器操作,故仅对端点号的情况进行处理;如果端点定义成 IN 端点,需要将 INDEX 寄存器根据端点号指向对应的端点,如端点号为 1,则 INDEX 寄存器需要指向端点 1,之后我们操作 INCSR1 寄存器的 SDSTL 位置 1(意思为产生 STALL 信号作为对一个 IN 令牌的应答),最后将端点索引器 INDEX 指向端点 0,以便后续的控制操作。同理,端点定义成 OUT 端点,则需要操作的是 OUTCSR1 的 SDSTL 位置 1,产生 STALL 信号作为对一个 OUT 令牌的应答。如果未设置端点的 IN 状态或 OUT 状态,则默认挂起。

第 3 步:进入请求结束状态:端点 0 设置成空闲状态,寄存器 CSR0 的 SPORDY 位以及 DATEND 位置位,表示处理完成从端点 0 接收到的数据包,且数据结束。

(4) SET_ADDRESS

依据以表 15.36,可以对该请求的特殊情况进行设计

表 15.36　SET_ADDRESS 内容

bmRequestType	bRequest	wValue	wIndex	wLength	数据过程
0x00	SET_ADDRESS	设备地址	0	0	无

第 1 步:特殊情况的处理,与 GET_STATUS 请求的设计思路一致,依旧是特殊情况处理。如果 USB 标准设备请求数据结构不满足表 15.36 所有条件,则挂起。

第 2 步:依据 USB 标准设备请求数据结构 wValue 字段确定主机给设备分配的唯一地址,之后将主机分配给设备的唯一地址写入 FADDR 寄存器中,表示设备以接收并设置了地址;最后依据地址是否为零设置设备的状态,如果地址为零,设备为默认状态;地址不为零,设备状态为 SET_ADDRESS。

第 3 步:进入请求结束状态:端点 0 设置成空闲状态,寄存器 CSR0 的 SPORDY 位以及

DATEND 位置位,表示处理完成从端点 0 接收到的数据包,且数据结束。

(5) GET_DESCRIPTOR

GET_DESCRIPTOR 请求是 USB 通信中最常使用的请求,有关此请求的内容十分重要,可以说 STEUP 阶段的主要数据负载就在此请求的响应中,首先,按照惯例,依据表 15.37,可以对该请求的特殊情况进行设计。

表 15.37 GET_DESCRIPTOR 内容

bmRequestType	bRequest	wValue	wIndex	wLength	数据过程
0x80	GET_DESCRIPTOR	描述符类型和索引	0 或者语言 ID	描述符长度	描述符

第 1 步:特殊情况的处理,与 GET_STATUS 请求的设计思路一致,依旧是特殊情况处理。如果 USB 标准设备请求数据结构不满足表 15.37 所有条件,则挂起。

第 2 步:依据 wValue 的值判定需要发送的描述符类型,表 15.38 将列出本次实验中 wValue 所确定的所有描述符类型。

表 15.38 wValue 所确定的所有描述符类型

wValue	值	描述符类型
DESC_DEVICE	0x0100	设备描述符
DESC_CONFIGURATION	0x0200	配置描述符
DESC_STRING	0x03000x03010x0302	LANGIDDESCMANUFACTDESCMANUFACTDESC
DESC_HIDREPORT	0x2200	报告描述符

若 wValue 不为以上情况,则默认为挂起操作。

注意:

1. wValue 的第一字节(字符)表示同一中描述类型(比如字符串描述符)中具体的某个描述符(如厂商或者产品字符),第二字节表示描述类型的编号。

2. 对于全速和低速模式,获取描述符的标准请求只有三种:获取设备、配置、字符串的描述符,另外的接口和端点描述符是跟随配置描述符一并返回的,不能单独请求返回。

3. 描述符的数据结构参照 1.3 节中,描述符

第 3 步:进入标准请求结束状态(进行数据过程),例如:在步骤 2 中软件已经依据 USB 协议内容将 DEVICEDESC 设备描述符数据包装入了数据缓冲区,然后软件再通过调用 usb_setup_in(),完成数据包的发送。

注意:usb_setup_in()的内容是:先将控制端点 0 设置为 DATAIN 状态,控制寄存器 CSR0 的 SPORDY 写 1 以清零 OPRDY,随后调用 usb_ctrl_in()进行数据发送,即使用控制 IN 类型向主机发送数据包。

(6) SET_DESCRIPTOR

此次实验不涉及该标准请求,故不对内容进行编写,若进入该函数,则调用 USB 挂起函数以替代。

(7) GET_CONFIGURATION

依据表 15.39,可以对该请求的特殊情况进行设计

表 15.39　GET_CONFIGURATION 内容

bmRequestType	bRequest	wValue	wIndex	wLength	数据过程
0x80	GET_CONFIGURATION	0	0	1	配置值

第 1 步:特殊情况的处理,与 GET_STATUS 请求的设计思路一致,依旧是特殊情况处理。如果 USB 标准设备请求数据结构不满足表 15.39 所有条件,则挂起。

第 2 步:依据设备的状态选择需要发送的数据包,如果设备已配置,则发送 PACKET1 数据包,如果未被配置则发送 PACKET0 数据包。

第 3 步:进入标准请求结束状态(进行数据过程),例如:在步骤 2 中软件已经依据 USB 协议内容将 PACKET0 数据包装入了数据缓冲区,然后软件再通过调用 usb_setup_in( ),完成数据包的发送。

注意:usb_setup_in( )的内容是:先将控制端点 0 设置为 DATAIN 状态,控制寄存器 CSR0 的 SPORDY 写 1 以清零 OPRDY,随后调用 usb_ctrl_in( )进行数据发送,即使用控制 IN 类型向主机发送数据包。

(8) SET_CONFIGURATION

SET_CONFIGURATION 和 SET_ADDRESS 请求很类似,区别是 wValue 的意义:SET_ADDRESS 中,wValue 的第一字节(低字节)表示设备的地址;SET_CONFIGURATION 则为配置的值。该值与配置描述符的配置编号一致时,表示选中该配置,通常为 1,因为大多数 USB 设备只有一种设置;若为 0,则设备进入地址设置状态。

依据表 15.40,可以对该请求的特殊情况进行设计。

表 15.40　SET_CONFIGURATION 内容

bmRequestType	bRequest	wValue	wIndex	wLength	数据过程
0x00	SET_CONFIGURATION	配置值(0x0001)	0	0	无

第 1 步:特殊情况的处理,与 GET_STATUS 请求的设计思路一致,依旧是特殊情况处理。如果 USB 标准设备请求数据结构不满足表 15.40 所有条件,则挂起。

第 2 步:依据 wValue 的低字节确定配置值,在操作寄存器配置设备状态前先设置设备状态为配置状态,并将 IN 端点状态以及 OUT 端点状态设为 0 状态,之后我们依据需要使用的端点配置相应的寄存器,例如:我们需要使用端点 1 作为 IN 端点,则先配置 INDEX 寄存器定位到 1,然后配置 INCSR2 寄存器的 MODEIN 位为 1,从而设置端点的方向,最后配置 INMAXP 寄存器来设置 USB 的 IN 端点 1 最大数据包的大小。如果 wValue 值不为 0x0001,则设备回到配置地址的状态。

第 3 步:进入请求结束状态:端点 0 设置成空闲状态,寄存器 CSR0 的 SPORDY 位以及 DATEND 位置位,表示处理完成从端点 0 接收到的数据包,且数据结束。

(9) GET_INTERFACE

依据表 15.41,可以对该请求的特殊情况进行设计。

表 15.41　GET_INTERFACE 内容

bmRequestType	bRequest	wValue	wIndex	wLength	数据过程
0x80	GET_INTERFACE	0	接口号	1	备用接口号

第 1 步:特殊情况的处理,与 GET_STATUS 请求的设计思路一致,依旧是特殊情况处理。

如果 USB 标准设备请求数据结构不满足表 15.41 所有条件,则挂起。

第 2 步:选择发送 PACKET0 数据包

第 3 步:进入标准请求结束状态(进行数据过程),例如:在步骤 2 中软件已经依据 USB 协议内容将 PACKET0 数据包装入了数据缓冲区,然后软件再通过调用 usb_setup_in( ),完成数据包的发送。

注意:usb_setup_in( )的内容是:先将控制端点 0 设置为 DATAIN 状态,控制寄存器 CSR0 的 SPORDY 写 1 以清零 OPRDY,随后调用 usb_ctrl_in( )进行数据发送,即使用控制 IN 类型向主机发送数据包。

(10) SET_INTERFACE

依据表 15.42,可以对该请求的特殊情况进行设计。

表 15.42 SET_INTERFACE 内容

bmRequestType	bRequest	wValue	wIndex	wLength	数据过程
0x01	SET_INTERFACE	备用接口号	接口号	0	无

第 1 步:特殊情况的处理,与 GET_STATUS 请求的设计思路一致,依旧是特殊情况处理。如果 USB 标准设备请求数据结构不满足表 15.42 所有条件,则挂起。

第 2 步:无操作。

第 3 步:进入请求结束状态:端点 0 设置成空闲状态,寄存器 CSR0 的 SPORDY 位以及 DATEND 位置位,表示处理完成从端点 0 接收到的数据包,且数据结束。

(11) SYNCH_FRAME

此次实验不涉及该标准请求,故不对内容进行编写,若进入该函数,则调用 USB 挂起函数以替代。