第5章

用户自定义的 USB HID 设备

俗话说得好，打铁要趁热。本章将继续讲述 HID 设备。与前两章不同的是，本章的 HID 设备是用户自定义的，也就是说，它不是标准的系统设备。在 Windows 下，标准的系统设备通常是操作系统独占的，应用程序无法直接访问这些设备的驱动程序。而用户自定义的 HID 设备，Windows 操作系统自身并不会访问它们。在 Windows 操作系统下，自带了 HID 设备的驱动程序，因而无需用户自己开发驱动程序。不过，HID 设备也有其固有的缺点，那就是数据只能使用中断或控制传输。由于对中断传输查询的时间间隔最小为一个帧(或者微帧)，因而 HID 设备速度受到了限制。对于一些数据量较少的场合(例如按键输入、LED 显示，甚至一些小容量的芯片烧录器等)，使用用户自定义的 HID 设备是很合适的。

5.1 MyUsbHid 工程的建立

本章程序将在第 4 章的 USB 键盘程序上进行修改。因为 USB 键盘本身是个 HID 设备，在 USB 键盘的基础上，只要进行少量的代码修改(甚至只修改应用集合的用途即可实现，这在过滤驱动程序的开发时会讲)即可实现。

将 USB 键盘程序 UsbKeyboard 复制一份，改名为 MyUsbHid。

5.2 描述符的修改

首先要修改的是设备描述符中的产品 ID 号(idProduct 字段)，这里是第六个实例程序，取 0x0006。其次是要修改配置描述符集合中的接口描述符，因为自定义的 HID 设备不使用子类和协议，将 bInterfaceSubClass 字段和 bInterfaceProtocol 都改为 0。然后将产品字符串改为“《圈圈教你玩 USB》之用户自定义的 USB HID 设备”，设备序列号改为“2008 - 07 - 19”。这些描述符修改都比较容易，剩下就是大头——报告描述符的修改。

将报告描述符中开应用集合的用途改为 0x00，在普通桌面页(Generic Desktop Page)中，用途 ID 值 0x00 是未定义的。如果使用该用途来开集合，那么系统将不会把它当作标准系统设备，从而就成了一个用户自定义的 HID 设备。这里我们决定使

用8字节的输入报告和输出报告，它们的逻辑最小值为0，逻辑最大值为255。用途可以随便定义，就从1到8吧。至于这些数据具体怎么用，用户可以自己决定。例如像本实验中，输入报告的第一字节用来描述8个按键的状态，第二到第五字节返回报告的次数(增加一个长整型的变量Count，每发送一次报告就加1)。而输出报告的第一字节则用来控制板上8个LED的状态，第二字节(非0时)用来清除上面的报告计数器Count，其他字节未用。修改好的报告描述符代码如下：

```
//USB报告描述符的定义
code uint8 ReportDescriptor[] =
{
 //每行开始的第一字节为该条目的前缀，前缀的格式为：
 //D7～D4：bTag。D3～D2：bType；D1～D0：bSize
 //以下分别对每个条目注释
 //这是一个全局(bType为1)条目，将用途页选择为普通桌面Generic Desktop Page
 //后面跟1字节数据(bSize为1)，后面的字节数就不注释了，自己根据bSize来判断
 0x05, 0x01,                    // USAGE_PAGE (Generic Desktop)

 //这是一个局部(bType为2)条目，用途选择为0x00。在普通桌面页中，该用途是未定义的，如果使用
 //该用途来开集合，那么系统将不会把它当作标准系统设备，从而就成了一个用户自定义的HID设备
 0x09, 0x00,                    // USAGE (0)

 //这是一个主条目(bType为0)，开集合，后面跟的数据0x01表示该集合是一个应用集合
 //它的性质在前面由用途页和用途定义为用户自定义
 0xa1, 0x01,                    // COLLECTION (Application)

 //这是一个全局条目，说明逻辑值最小值为0
 0x15, 0x00,                    // LOGICAL_MINIMUM (0)

 //这是一个全局条目，说明逻辑值最大值为255
 0x25, 0xff,                    // LOGICAL_MAXIMUM (255)

 //这是一个局部条目，说明用途的最小值为1
 0x19, 0x01,                    // USAGE_MINIMUM (1)

 //这是一个局部条目，说明用途的最大值为8
 0x29, 0x08,                    // USAGE_MAXIMUM (8)

 //这是一个全局条目，说明数据域的数量为8个
 0x95, 0x08,                    // REPORT_COUNT (8)

 //这是一个全局条目，说明每个数据域的长度为8位，即1字节
 0x75, 0x08,                    // REPORT_SIZE (8)

 //这是一个主条目，说明有8个长度为8位的数据域作为输入
 0x81, 0x02,                    // INPUT (Data,Var,Abs)
```

```
    //这是一个局部条目,说明用途的最小值为1
    0x19, 0x01,                    // USAGE_MINIMUM (1)
    //这是一个局部条目,说明用途的最大值为8
    0x29, 0x08,                    // USAGE_MAXIMUM (8)
    //这是一个主条目。定义输出数据(8字节,注意前面的全局条目)
    0x91, 0x02,                    // OUTPUT (Data,Var,Abs)
    //下面这个主条目用来关闭前面的集合。bSize为0,所以后面没数据
    0xc0                           // END_COLLECTION
};
//////////////////////////报告描述符完毕//////////////////////////
```

通过上面的报告描述符的定义,我们知道返回的输入报告具有8字节。输出报告也有8字节。至于这8字节的数据是干什么用的,就由用户自己来决定了。

5.3 报告的实现

在发送报告的处理中,首先对计数器Count加1。然后将8个按键的情况写入到报告的第一字节,将Count(4字节)分别写入到报告的第二至第五字节,最后将8字节的报告写入到端点1中。发送报告的处理代码如下:

```
/*************************************************************
函数功能:根据按键情况返回报告的函数
入口参数:无
返    回:无
备    注:无
*************************************************************/
void SendReport(void)
{
 //需要返回的8字节报告的缓冲。在本测试程序中,只使用前5字节
 uint8 Buf[8] = {0,0,0,0,0,0,0,0};

 //每发送一次数据,则将Count增加1
 Count ++ ;

 //根据不同的按键设置输入报告。这里将8个按键情况放在第一字节
 Buf[0] = KeyPress;

 //根据Count的值设置报告的第二到第五字节
 Buf[1] = (Count&0xFF);                    //最低字节
 Buf[2] = ((Count>>8)&0xFF);               //次低字节
 Buf[3] = ((Count>>16)&0xFF);              //次高字节
 Buf[4] = ((Count>>24)&0xFF);              //最高字节
```

```
    //报告准备好了,通过端点 1 返回,长度为 8 字节
    D12WriteEndpointBuffer(3,8,Buf);
    Ep1InIsBusy = 1;                                    //设置端点忙标志

    //记得清除 KeyUp 和 KeyDown
    KeyUp = 0;
    KeyDown = 0;
}
/////////////////////End of function/////////////////////////////
```

再到端点 1 输出中断处理函数中,修改输出报告相关处理。原来缓冲区为 1 字节,这里要改为 8 字节,同时读取端点 1 输出缓冲时,要读 8 字节。第一字节的功能为控制 LED,这里不用改动。第二字节为非 0 值时,将清除发送计数器 Count 的值。修改后的代码如下:

```
/*************************************************************
函数功能:端点 1 输出中断处理函数
入口参数:无
返　　回:无
备　　注:无
*************************************************************/
void UsbEp1Out(void)
{
    uint8 Buf[8];              //用来保存 8 字节的输出报告
#ifdef DEBUG0
    Prints("USB 端点 1 输出中断。\r\n");
#endif

    //读端点最后状态,这将清除端点 1 输出的中断标志位
    D12ReadEndpointLastStatus(2);

    //从端点 1 输出缓冲读回 8 字节数据
    D12ReadEndpointBuffer(2,8,Buf);

    //清除端点缓冲区
    D12ClearBuffer();

    //输出报告第一字节为 LED 状态,某位为 1 时,表示 LED 亮
    LEDs = ~Buf[0];

    //输出报告的第二字节非 0 时,清除发送计数器 Count
    if(Buf[1]! = 0)
    {
        Count = 0;
    }
}
```

//////////////////////End of function//////////////////////////////

至此，用户自定义的 USB HID 设备的代码就修改完成了。然后编译工程，并下载到学习板中运行，就会发现新硬件。在设备管理器中，展开“人体学输入设备”，下面新增了一个“USB-compliant device”和一个“USB 人体学输入设备”，如图 5.3.1 所示。

“USB-compliant device”是由“USB 人体学输入设备”产生的，这个关系跟我们前面的 USB 鼠标、USB 键盘与“USB 人体学输入设备”之间的关系一样。如果在 USB 键盘中，将它下面的“USB 人体学输入设备”所读到的报告描述符内容改了（过滤驱动程序可以干这个活），将它的应用集合用途改为 0 或 0xFF，那么它就不再产生键盘设备了，而是像这里一样，变成了一个“USB-compliant device”。实现这个功能的过滤驱动程序可以安装在“USB 人体输入学设备”之下（这样的过滤驱动叫做下层过滤驱动，相应地，还有上层过滤驱动），那么很自然的，它就位于“USB 人体学输入设备”和 USB 总线驱动程序之间了，是它在中间做了手脚。

图 5.3.1　设备管理器中新增的设备

5.4　对用户自定义的 USB HID 设备的访问

要访问 HID 设备，就必须跟 HID 设备的驱动程序打交道。在 Windows 操作系统环境下，设备通常被当作特殊文件处理。要打开这个设备，就需要知道该设备的路径（设备接口名）。要找到设备的路径，通常使用 GUID 来查找。

在设备安装时，Windows 安装器和驱动程序负责将相应的设备与对应的 GUID 联系起来，并写入到注册表中。这样通过 GUID 就可找到对应设备。例如，HID 设备的接口类 GUID 是{4D1E55B2-F16F-11CF-88CB-001111000030}。但是这个值在不同的系统下也许会不一样，所以在本程序中并不直接使用这个 GUID，而是使用一个 API 函数来获取它。把系统中所有 HID 类的设备都列举出来，然后检查它的 VID、PID 以及设备版本号，看是不是要访问的设备。如果是，就可以对设备进行各种操作了。

5.5 访问HID设备时所用到的相关函数

本实例程序使用VC++ 6.0开发，使用其他开发环境（例如VB、Delphi等）开发也是类似的，都是调用一些API函数。下面分别介绍访问HID设备时需要使用的一些函数，这些函数的原型可以在MSDN中查找，或者在网上搜索，例如http://www.hszxcx.cn。

5.5.1 获取HID设备的接口类GUID的函数

```
void __stdcall HidD_GetHidGuid(OUT LPGUID HidGuid);
```

调用该函数可以获取HID设备的接口类GUID（Interface class GUID）。其中，OUT是个空的宏定义，仅是为了方便阅读，说明参数的作用是接收数据的；LPGUID是指向GUID结构体的指针类型，因此参数HidGuid就是一个GUID结构体指针。GUID是一个128位（16字节）的整数，分成不同的段用结构体来保存，详细结构可以参看代码中GUID结构体的定义。

5.5.2 获取指定类的所有设备信息集合的函数

```
HDEVINFO SetupDiGetClassDevs(const GUID * ClassGuid,
                             PCTSTR Enumerator,
                             HWND hwndParent,
                             DWORD Flags);
```

调用该函数将返回由ClassGuid指定的所有设备的一个信息集合的句柄。当信息集合使用完毕之后，需要调用函数SetupDiDestroyDeviceInfoList()去销毁。

入口参数ClassGuid就是要指定访问的设备类的GUID，可以是设备接口类GUID（Device Interface class GUID），也可以是安装类GUID（Setup class GUID）。这两种GUID有什么区别呢？设备接口类GUID就是由驱动程序负责添加的GUID，例如上面的那个HID设备的GUID为{4D1E55B2-F16F-11CF-88CB-001111000030}，该GUID将出现在系统注册表中的HKEY_LOCAL_MACHINE\SYSTEM\CurrentControlSet\Control\DeviceClasses子键下。接口类GUID下有曾经安装过的设备，键值就是设备的路径（设备接口名）。而安装类GUID则是在设备安装时，由Windows安装器添加到注册表中。通常安装器从安装驱动的inf文件中获取这个安装类GUID，例如安装HID设备的inf文件是Windows/inf文件夹下的input.inf，打开它可以找到ClassGuid = {745a17a0-74d3-11d0-b6fe-00a0c90f57da}，这就指定了HID设备的安装类GUID为{745A17A0-74D3-11D0-B6FE-00A0C90F57DA}。设备的安装类GUID出现在注册表的HKEY_LOCAL_MACHINE\SYSTEM\CurrentControlSet\Control\Class子键下，与接口类GUID

类似，安装类GUID下也有曾经安装过的设备，不过这里是用数字表示的。另外，在注册表的HKEY_LOCAL_MACHINE\SYSTEM\CurrentControlSet\Enum子键下，有不同设备的分类，在分类下可以找到对应的设备，里面记录了设备安装时的安装类GUID。搞清楚这两个GUID是很有必要的，如果类型搞错了，通常就会找不到设备，因为同一个设备的接口类GUID和安装类GUID通常是不同的。那么SetupDiGetClassDevs()函数调用时，如何决定传入的GUID是接口类的还是安装类，这由最后一个参数Flags里面的位4来决定，即DIGCF_DEVICEINTERFACE标志。当位4的值为0时，指定使用安装类GUID；当位4的值为1时，指定使用接口类GUID。另外Flags还有其他几个标志，读者可以自己去看该函数的帮助。本实例程序下要用到它的一个标志是DIGCF_PRESENT，它是Flags的位1。当位1为1时，表示只列举出已经连接的设备，否则将列出全部安装过的设备。

入口参数Enumerator是可选的，当其值为NULL时，将搜索全部设备；它也可以指定一个设备的PnP名字的字符串，从而限制搜索。

参数hwndParent为父窗口的句柄，可以指定为NULL。

当该函数调用失败时，将返回INVALID_HANDLE_VALUE，调用GetLastError()函数可以获取失败的原因。

5.5.3 从设备信息集合中获取一个设备接口信息的函数

```
BOOL SetupDiEnumDeviceInterfaces (HDEVINFO DeviceInfoSet,
                                  PSP_DEVINFO_DATA DeviceInfoData,
                                  const GUID * InterfaceClassGuid,
                                  DWORD MemberIndex,
                                  PSP_DEVICE_INTERFACE_DATA DeviceInterfaceData);
```

调用该函数可以从设备信息集合中获取某个设备接口信息。当该函数调用成功时，返回非0值，当调用失败时(例如指定的设备不存在)，将返回0值。

入口参数DeviceInfoSet是一个设备信息集合的句柄，可以用5.5.2小节中介绍的SetupDiGetClassDevs()函数来获取。

入口参数DeviceInfoData是一个SP_DEVINFO_DATA型的结构体变量，可以用来强制获取某个设备的信息。该参数是可选的，值为NULL表示不使用该参数。

入口参数InterfaceClassGuid是一个指向设备的接口类GUID的指针。

入口参数MemberIndex是整型变量，它指定获取设备信息集合中某个特定的设备信息。0表示第一个设备，1表示第二个设备，以此类推。当该值超出实际的设备数量时，函数就会返回0，表示调用失败。此时使用GetLastError()函数将会得到一个没有更多条目的出错代码：ERROR_NO_MORE_ITEMS。

入口参数DeviceInterfaceData是一个SP_DEVICE_INTERFACE_DATA的结构体，用来接收设备的信息。在函数调用之前，必须先设置好该结构体的cbSize成

员为该结构体的大小。

5.5.4 获取指定设备接口详细信息的函数

```
BOOL SetupDiGetDeviceInterfaceDetail(
    HDEVINFO DeviceInfoSet,
    PSP_DEVICE_INTERFACE_DATA DeviceInterfaceData,
    PSP_DEVICE_INTERFACE_DETAIL_DATA DeviceInterfaceDetailData,
    DWORD DeviceInterfaceDetailDataSize,
    PDWORD RequiredSize,
    PSP_DEVINFO_DATA DeviceInfoData);
```

调用该函数可以获取一个指定设备接口的详细信息，例如设备的路径(设备接口名)。函数调用成功将会返回非0值，调用失败时返回0。

入口参数DeviceInfoSet是设备信息集合的句柄。

入口参数DeviceInterfaceData是保存设备信息的结构体，可以使用前面介绍的函数SetupDiEnumDeviceInterfaces()来获取。

入口参数DeviceInterfaceDetailData是一个PSP_DEVICE_INTERFACE_DETAIL_DATA的结构体指针，用来接收设备接口的详细信息。该结构体的成员变量DevicePath中就保存着用来打开设备的路径(或者叫设备接口名)，打开设备的API函数CreateFile可以使用该参数来打指定的设备。在调用函数之前，必须要初始化该结构体中的成员变量cbSize为该结构体的大小(注意：不包括后面用来保存路径的缓冲区大小)。该参数可以为NULL，此时DeviceInterfaceDetailDataSize参数必须为0。

入口参数DeviceInterfaceDetailDataSize是DeviceInterfaceDetailData缓冲区的大小，当该值比实际需要的字节数少时，函数调用失败，返回0。

入口参数RequiredSize是一个指向整型变量的指针，该变量用来保存设备接口详细信息实际需要的缓冲区大小。该参数是可选的。

入口参数DeviceInfoData是一个用来接收该接口所在设备的设备信息的结构体指针。该参数是可选的，值为NULL表示该参数无效。

通常，在获取设备详细信息时调用该函数两次。第一次是为了获取保存设备详细信息所需要的缓冲区长度，这时设置参数DeviceInterfaceDetailData为NULL，参数DeviceInterfaceDetailDataSize为0，同时提供一个用来接收字节数变量的指针RequiredSize。这样函数就会调用失败，调用GetLastError()函数将会获取到一个缓冲区不足的错误代码。同时会将所需要的缓冲区大小写入到RequiredSize所指向的变量中。接着，根据所需要的字节数，去申请一个缓冲区作为接收设备接口详细信息的结构体，并设置该结构体的cbSize成员值为该结构体的大小。然后再次调用该函数，将参数DeviceInterfaceDetailData设置为缓冲区的首地址，而DeviceInterfaceDetailDataSize则为该缓冲区的大小。此时就不再需要参数RequiredSize了，将它置为NULL。

5.5.5 打开设备的函数

```
HANDLE CreateFile(LPCTSTR lpFileName,
                  DWORD dwDesiredAccess,
                  DWORD dwShareMode,
                  LPSECURITY_ATTRIBUTES lpSecurityAttributes,
                  DWORD dwCreationDisposition,
                  DWORD dwFlagsAndAttributes,
                  HANDLE hTemplateFile);
```

调用该函数可以打开指定的设备，并返回一个指向该设备的句柄。

入口参数 lpFileName 是一个指向要打开的设备的名称(或者叫做路径)的字符串，通过获取设备详细信息可以找到设备的路径。

入口参数 dwDesiredAccess 是指访问的方式，可以是只读、只写或者读/写方式，也可以设置为无读/写操作(NULL)，这样并不访问设备，但是可以获取到设备的属性。例如 USB 鼠标、USB 键盘等是操作系统的独占设备，因而使用读/写方式是无法打开文件的，但是如果设置为无读/写操作，那么可以获取到这些设备的属性，例如 VID、PID 等。

入口参数 dwShareMode 是共享模式，可以是无共享、共享读、共享写或者共享读/写。当设置为无共享并成功打开设备后，该设备就不能再被其他程序打开来访问了。

入口参数 lpSecurityAttributes 是一个指向 SECURITY_ATTRIBUTES 结构体的指针，用来决定返回的句柄能否继承到子过程中去。当该值为 NULL 时，句柄不能继承。

入口参数 dwCreationDisposition 用来决定当被打开的设备或者文件不存在时，执行怎样的操作。可以选择为始终创建新文件，当文件不存在时创建新文件，或者只能打开已经存在的设备等。打开物理设备时，一般选择为只打开已经存在的设备(设备是一个特殊文件)。

入口参数 dwFlagsAndAttributes 为访问的标志和属性，可以选择普通访问，或者是异步调用等，具体的选项很多，可以参考 MSDN 帮助。一般同步调用时使用参数 FILE_ATTRIBUTE_NORMAL，异步调用时使用参数 FILE_FLAG_OVERLAPPED。

入口参数 hTemplateFile 可以指定一个临时文件的句柄，该参数可以为 NULL，即不使用临时文件。

使用该函数打开设备后，就可以使用该函数返回的句柄来获取设备的属性，以及对设备进行读/写操作。

5.5.6 获取HID设备属性的函数

```
BOOLEAN __stdcall HidD_GetAttributes(IN HANDLE HidDeviceObject,
                                     OUT PHIDD_ATTRIBUTES Attributes);
```

调用该函数可以获取指定设备的属性，例如VID、PID、设备版本号等。调用成功后返回非0值，调用失败时返回0。

入口参数HidDeviceObject是被访问设备的句柄，可调用CreateFile打开一个设备从而获得该句柄。

入口参数Attributes是一个指向HIDD_ATTRIBUTES结构体的指针，用来保存HID设备的属性。

5.5.7 从设备读取数据的函数

```
BOOL ReadFile(HANDLE hFile,
              LPVOID lpBuffer,
              DWORD nNumberOfBytesToRead,
              LPDWORD lpNumberOfBytesRead,
              LPOVERLAPPED lpOverlapped);
```

调用该函数将从指定的设备读取数据。当操作成功时，返回非0值；当操作失败时，返回0值。注意当CreateFile打开文件时，如果选择了异步打开，那么该函数调用后即使操作未完成也会立即返回。此时返回值为0，但这可能并不表示操作失败，通过调用GetLastError()函数可以获取出错的代码。如果出错代码为ERROR_IO_PENDING，则表示驱动程序将这个IRP挂起了，等数据成功返回时，这个IRP才会被完成。

入口参数hFile是需要访问的设备或文件的句柄。

入口参数lpBuffer为保存接收数据的缓冲区。

入口参数nNumberOfBytesToRead为需要读取数据的长度(字节数)。

入口参数lpNumberOfBytesRead为一个指向保存实际读取到多少字节的变量的指针。当lpOverlapped不为NULL时，该参数可以为NULL。

入口参数lpOverlapped为一个指向OVERLAPPED结构体的指针。当选择为异步调用时，必须提供一个lpOverlapped参数。该参数指向的结构体提供一个事件的句柄，当IRP完成时会触发该事件。调用该函数时，事件会自动设置为无效状态，调用者不用负责清除事件标志。另外，在该结构体中，还提供了读取数据的偏移量。使用了lpOverlapped参数后，将不再使用lpNumberOfBytesRead所指向的变量来保存实际接收到的字节数了，而是通过函数GetOverlappedResult()来获取。

对于USB HID设备，使用该函数只能从中断端点获取报告数据，如果要从控制端点获取报告，则使用函数HidD_GetInputReport()。

5.5.8 往设备写数据的函数

```
BOOL WriteFile(HANDLE hFile,
               LPCVOID lpBuffer,
               DWORD nNumberOfBytesToWrite,
               LPDWORD lpNumberOfBytesWritten,
               LPOVERLAPPED lpOverlapped);
```

调用该函数将把指定的数据发送到指定的设备。该函数的返回值的意义跟ReadFile()函数一样。

入口参数 hFile 是需要访问的设备或文件的句柄。

入口参数 lpBuffer 是指向待写入数据的缓冲区。

入口参数 nNumberOfBytesToWrite 为需要写入的字节数。

入口参数 lpNumberOfBytesWritten 为指向一个用来保存实际写入字节数的变量的指针。

入口参数 lpOverlapped 为一个指向 OVERLAPPED 结构体的指针，用途跟ReadFile()函数一样，请参看 ReadFile 的说明。

对于 USB HID 设备，使用该函数只能从中断端点发送报告，如果要从控制端点发送报告，则使用函数 HidD_SetOutputReport()。

5.5.9 通过控制端点0读取报告的函数

```
BOOLEAN __stdcall HidD_GetInputReport (IN HANDLE HidDeviceObject,
                                       OUT PVOID ReportBuffer,
                                       IN ULONG ReportBufferLength);
```

调用该函数可以通过控制端点 0 获取报告，调用成功时，返回非 0 值；调用失败时，返回 0。调用该函数后，驱动程序将会发送获取报告的类输入请求，设备在数据阶段通过控制端点 0 返回其报告。如果设备在规定的时间内未返回报告，该函数将会超时返回。

入口参数 HidDeviceObjec 为已打开的设备的句柄。

入口参数 ReportBuffer 是用来接收报告的缓冲区。

入口参数 ReportBufferLength 是接收缓冲区的长度。

5.5.10 通过控制端点0发送报告的函数

```
BOOLEAN __stdcall HidD_SetOutputReport (IN HANDLE HidDeviceObject,
                                        IN PVOID ReportBuffer,
                                        IN ULONG ReportBufferLength);
```

调用该函数可以通过控制端点 0 发送报告，调用成功时，返回非 0 值；调用失败时，返回 0。调用该函数后，驱动程序将会发送设置报告的类输出请求，并在数据阶

段将报告发送到设备的控制端点0。如果设备在规定的时间内未能接收报告，那么该函数将会超时返回。

入口参数 HidDeviceObjec 为已打开的设备的句柄。

入口参数 ReportBuffer 是保存待发送报告的缓冲区。

入口参数 ReportBufferLength 是需要发送报告的长度。

5.5.11 关闭句柄的函数

```
BOOL CloseHandle(HANDLE hObject);
```

调用该函数将关闭已打开的句柄 hObject。

入口参数 hObject 为需要关闭的设备或文件的句柄。

5.5.12 需要包含的库文件

函数名中包含 Hid 字样的函数属于 hid.lib，函数名中包含 Setup 字样的函数属于 setupapi.lib，需要在工程设置选项的 Link 下的 Library Modules 下加入 hid.lib 和 setupapi.lib，否则就会出现链接错误。不过，VC 自己并没有自带这个 hid.lib，在 DDK 中才有，有些读者可能没有安装 DDK，因此圈圈将这个 hid.lib 复制了一份放在工程包中。注意这些函数的声明，是标准的 C 函数格式，因此在 C++文件中引用头文件时加上 extern "C"，否则会链接不上。

5.6 访问 USB HID 设备的上位机软件的实现

5.6.1 上位机程序编写的思路

查找指定的 USB HID 设备的思路和步骤如下：

① 使用 HidD_GetHidGuid()函数获取 HID 设备的接口类 GUID。

② 使用 SetupDiGetClassDevs()函数获取 HID 类中所有设备的信息集合。

③ 在该设备信息集合中，使用 SetupDiEnumDeviceInterfaces()函数去获取一个设备的信息。如果调用该函数时返回失败，则说明已经到了设备集合的末尾，后面已经没有设备了，此时应该退出查找。

④ 使用 SetupDiGetDeviceInterfaceDetail()函数获取某个设备的详细信息。要获取某个设备的详细信息，SetupDiGetDeviceInterfaceDetail()函数必须调用两次。第一次调用是为了得到保存设备详细信息需要多大的缓冲区，第二次调用才是真正的获取设备详细信息。

⑤ 获得设备的详细信息后，就可以使用 CreateFile()函数打开指定的设备了。

⑥ 打开设备后，再使用 HidD_GetAttributes()函数获取设备的属性，在属性中包含了 VID、PID 以及产品版本号等信息。然后再比较 VID、PID 以及产品版本号是

否跟所指定的一致。如果一致，则退出查找，说明设备已经找到；如果不一致，说明这个设备不是我们需要的设备，切换到下一个设备，然后重复前面的步骤③～⑥。

找到指定的设备后，就可以对设备进行读/写数据了。由于我们设计的HID设备是使用中断端点来传输数据，因此应该调用ReadFile()和WriteFile()函数来读、写报告。如果要操作的设备是使用默认的控制端点来读/写报告的，应该调用HidD_GetInputReport()函数和HidD_SetOutputReport()函数。

为了演示应用程序跟实验板之间的通信，在界面上放置两排8个的LED按键。一排用来控制板上的LED状态，可以用鼠标单击在开和关之间切换，另一排用来显示学习板上按键的情况。当某个按键按下时，界面上对应的LED就会亮。

实现开关状态显示以及计数值显示的编程思路是：在读报告线程中调用ReadFile()函数发送读取报告的请求，然后等待事件的发生。当学习板上有按键变动时，就会返回数据。此时ReadFile()发送的请求就会被完成，并设置事件为有效状态。这将唤醒读报告线程，从而根据接收到的数据设置LED状态以及计数器值。处理完毕后再发送读取报告的请求，等待下一次报告的返回。

实现开关控制以及清除计数值的编程思路是：当对应的按键按下后，就设置好要发送的报告数据，然后使用WriteFile()函数将报告发送到设备。同时设置一个标志，说明数据正在发送中。当数据发送完毕，就会触发事件，然后在事件响应代码中将该标志清除。在每次发送数据前，应该先检查该标志。如果数据正在发送中，那么就不能发送数据，操作无效。

需要注意的是，虽然我们的设备并没有设置报告ID，但是数据在从“USB人体学输入设备”传到“USB-compliant device”时，会自动增加一个值为0的报告ID。这样获取报告时实际上会读到9字节的数据，后面8字节才是设备真正返回的数据。同样，在发送数据时，也要事先增加一个值为0的报告ID，下层的驱动会自动将前面的报告ID去掉，然后将剩余的数据发送到设备。当然，如果设备指定了报告ID，就不存在上面这个问题。

当设备在操作过程中被拔下时，应用程序应该要能够感知到这一事件，并停止对设备的继续操作。另外，当不再需要使用设备或者关闭程序时，应该要关闭已经打开的设备句柄。

下面给出部分该测试程序的关键代码，只要把这些代码搞清楚了，应该就会知道如何去查找设备、打开设备以及操作设备了。

5.6.2 查找及打开HID设备的代码

在该程序的操作界面上，有一个打开设备的按钮，当点击该按钮时，就会在所有已经连接的HID设备中搜索目的HID设备。当指定的设备找到后，就会退出查找，并分别以读方式和写方式打开设备。然后禁止打开设备的按钮，使能其他操作按钮。该函数中使用了好几个上面介绍过的API函数，记得需要包括相应的头文件以及在

Link 选项中增加相应的库文件。函数中使用了一些全局变量，用来保存设备路径名及打开设备的句柄等。代码如下：

```
//点击打开设备按钮的处理函数
void CMyUsbHidTestAppDlg::OnOpenDevice()
{
 //定义一个 GUID 的结构体 HidGuid 来保存 HID 设备的接口类 GUID
 GUID HidGuid;
 //定义一个 DEVINFO 的句柄 hDevInfoSet 来保存获取到的设备信息集合句柄
 HDEVINFO hDevInfoSet;
 //定义 MemberIndex，表示当前搜索到第几个设备，0 表示第一个设备
 DWORD MemberIndex;
 //DevInterfaceData，用来保存设备的驱动接口信息
 SP_DEVICE_INTERFACE_DATA DevInterfaceData;
 //定义一个 BOOL 变量，保存函数调用是否返回成功
 BOOL Result;
 //定义一个 RequiredSize 的变量，用来接收需要保存详细信息的缓冲长度
 DWORD RequiredSize;
 //定义一个指向设备详细信息的结构体指针
 PSP_DEVICE_INTERFACE_DETAIL_DATA pDevDetailData;
 //定义一个用来保存打开设备的句柄
 HANDLE hDevHandle;
 //定义一个 HIDD_ATTRIBUTES 的结构体变量，保存设备的属性
 HIDD_ATTRIBUTES DevAttributes;

 //初始化设备未找到
 MyDevFound = FALSE;

 //获取在文本框中设置的 VID、PID、PVN
 GetMyIDs();

 //初始化读、写句柄为无效句柄
 hReadHandle = INVALID_HANDLE_VALUE;
 hWriteHandle = INVALID_HANDLE_VALUE;

 //对 DevInterfaceData 结构体的 cbSize 初始化为结构体大小
 DevInterfaceData.cbSize = sizeof(DevInterfaceData);
 //对 DevAttributes 结构体的 Size 初始化为结构体大小
 DevAttributes.Size = sizeof(DevAttributes);

 //调用 HidD_GetHidGuid()函数获取 HID 设备的 GUID，并保存在 HidGuid 中
 HidD_GetHidGuid(&HidGuid);

 //根据 HidGuid 来获取设备信息集合。其中 Flags 参数设置为 DIGCF_DEVICEINTERFACE|
DIGCF_PRESENT
 //前者表示使用的 GUID 为接口类 GUID，后者表示只列举正在使用的设备
```

```
        //因为我们这里只查找已经连接上的设备,返回的句柄保存在 hDevinfo 中
        //注意设备信息集合在使用完毕后要使用函数 SetupDiDestroyDeviceInfoList()销毁
        //不然会造成内存泄漏
        hDevInfoSet = SetupDiGetClassDevs(&HidGuid,
                                          NULL,
                                          NULL,
                                          DIGCF_DEVICEINTERFACE|DIGCF_PRESENT);
        AddToInfOut("开始查找设备");
        //对设备集合中每个设备进行列举,检查是否是我们要找的设备
        //当找到指定的设备,或者设备已经查找完毕时,就退出查找
        //首先指向第一个设备,即将 MemberIndex 置为 0
        MemberIndex = 0;
        while(1)
        {
         //调用 SetupDiEnumDeviceInterfaces,在设备信息集合中获取编号为 MemberIndex 的设备信息
         Result = SetupDiEnumDeviceInterfaces(hDevInfoSet,
                                              NULL,
                                              &HidGuid,
                                              MemberIndex,
                                              &DevInterfaceData);
        //如果获取信息失败,则说明设备已经查找完毕,退出循环
        if(Result == FALSE) break;

        //将 MemberIndex 指向下一个设备
        MemberIndex ++ ;

        //如果获取信息成功,则继续获取该设备的详细信息。在获取设备详细信息时,需要先知道保存
        //详细信息需要多大的缓冲区,这通过第一次调用函数 SetupDiGetDeviceInterfaceDetail 来获取
        //这时提供缓冲区和长度都为 NULL 的参数,并提供一个用来保存需要多大缓冲区的变量 RequiredSize
        Result = SetupDiGetDeviceInterfaceDetail(hDevInfoSet,
                                                 &DevInterfaceData,
                                                 NULL,
                                                 NULL,
                                                 &RequiredSize,
                                                 NULL);
        //然后,分配一个大小为 RequiredSize 的缓冲区,用来保存设备详细信息
        pDevDetailData = (PSP_DEVICE_INTERFACE_DETAIL_DATA)malloc(RequiredSize);
        if(pDevDetailData == NULL) //如果内存不足,则直接返回
```

```
{
 MessageBox("内存不足!");
 SetupDiDestroyDeviceInfoList(hDevInfoSet);
 return;
}
//设置 pDevDetailData 的 cbSize 为结构体的大小(注意只是结构体大小,不包括后面缓冲区)
pDevDetailData->cbSize = sizeof(SP_DEVICE_INTERFACE_DETAIL_DATA);
//再次调用 SetupDiGetDeviceInterfaceDetail()函数来获取设备的详细信息
//这次调用设置使用的缓冲区以及缓冲区大小
Result = SetupDiGetDeviceInterfaceDetail(hDevInfoSet,
                                         &DevInterfaceData,
                                         pDevDetailData,
                                         RequiredSize,
                                         NULL,
                                         NULL);
//将设备路径复制出来,然后销毁刚刚申请的内存
MyDevPathName = pDevDetailData->DevicePath;
free(pDevDetailData);
//如果调用失败,则查找下一个设备
if(Result == FALSE) continue;
//如果调用成功,则使用不带读/写访问的 CreateFile()函数来获取设备的属性,包括 VID、PID、
版本号等
//对于一些独占设备(例如 USB 键盘),使用读访问方式是无法打开的,
//而使用不带读/写访问的格式才可以打开这些设备,从而获取设备的属性
hDevHandle = CreateFile(MyDevPathName,
                        NULL,
                        FILE_SHARE_READ|FILE_SHARE_WRITE,
                        NULL,
                        OPEN_EXISTING,
                        FILE_ATTRIBUTE_NORMAL,
                        NULL);
//如果打开成功,则获取设备属性
if(hDevHandle != INVALID_HANDLE_VALUE)
{
 //获取设备的属性并保存在 DevAttributes 结构体中
 Result = HidD_GetAttributes(hDevHandle,
                             &DevAttributes);
 //关闭刚刚打开的设备
 CloseHandle(hDevHandle);
```

```
//获取失败,查找下一个
if(Result == FALSE) continue;

//如果获取成功,则将属性中的 VID、PID 以及设备版本号与我们需要的进行比较
//如果都一致,则说明它就是要找的设备
if(DevAttributes.VendorID == MyVid)                    //如果 VID 相等
 if(DevAttributes.ProductID == MyPid)                  //并且 PID 相等
  if(DevAttributes.VersionNumber == MyPvn)             //并且设备版本号相等
  {
   MyDevFound = TRUE;                                  //设置设备已经找到
   AddToInfOut("设备已经找到");

   //分别使用读、写方式打开之,保存其句柄并且选择为异步访问方式
   //读方式打开设备
   hReadHandle = CreateFile(MyDevPathName,
                            GENERIC_READ,
                            FILE_SHARE_READ|FILE_SHARE_WRITE,
                            NULL,
                            OPEN_EXISTING,
                            FILE_ATTRIBUTE_NORMAL|FILE_FLAG_OVERLAPPED,
                            NULL);
   if(hReadHandle! = INVALID_HANDLE_VALUE)
     AddToInfOut("读访问打开设备成功");
   else AddToInfOut("读访问打开设备失败");

   //写方式打开设备
   hWriteHandle = CreateFile(MyDevPathName,
                             GENERIC_WRITE,
                             FILE_SHARE_READ|FILE_SHARE_WRITE,
                             NULL,
                             OPEN_EXISTING,
                             FILE_ATTRIBUTE_NORMAL|FILE_FLAG_OVERLAPPED,
                             NULL);
   if(hWriteHandle! = INVALID_HANDLE_VALUE)
     AddToInfOut("写访问打开设备成功");
   else AddToInfOut("写访问打开设备失败");

   DataInSending = FALSE;              //可以发送数据

   //手动触发事件,让读报告线程恢复运行。因为在这之前并没有调用读数据的函数
   //也就不会引起事件的产生,所以需要先手动触发一次事件,让读报告线程恢复运行
   SetEvent(ReadOverlapped.hEvent);

   //显示设备的状态
   SetDlgItemText(IDC_DS,"设备已打开");
```

```
        //找到设备,退出循环。本程序只检测一个目标设备,查找到后就退出查找了
        //如果你需要将所有的目标设备都列出来,可以设置一个数组,
        //找到后就保存在数组中,直到所有设备都查找完毕才退出查找
        break;
      }
    }
    //如果打开失败,则查找下一个设备
    else continue;
  }

  //调用 SetupDiDestroyDeviceInfoList()函数销毁设备信息集合
  SetupDiDestroyDeviceInfoList(hDevInfoSet);
  //如果设备已经找到,那么应该使能各操作按钮,并同时禁止打开设备按钮
  if(MyDevFound)
  {
    //禁止打开设备按键,使能关闭设备,清计数器按键
    GetDlgItem(IDC_OPEN_DEVICE)->EnableWindow(FALSE);
    GetDlgItem(IDC_CLOSE_DEVICE)->EnableWindow(TRUE);
    GetDlgItem(IDC_CLEAR_COUNTER)->EnableWindow(TRUE);
    //使能 LED 控制按键
    GetDlgItem(IDC_LED1)->EnableWindow(TRUE);
    GetDlgItem(IDC_LED2)->EnableWindow(TRUE);
    GetDlgItem(IDC_LED3)->EnableWindow(TRUE);
    GetDlgItem(IDC_LED4)->EnableWindow(TRUE);
    GetDlgItem(IDC_LED5)->EnableWindow(TRUE);
    GetDlgItem(IDC_LED6)->EnableWindow(TRUE);
    GetDlgItem(IDC_LED7)->EnableWindow(TRUE);
    GetDlgItem(IDC_LED8)->EnableWindow(TRUE);
  }
  else
  {
    AddToInfOut("设备未找到");
  }
}
//////////////////////////////End of function/////////////////////
```

5.6.3　读输入报告线程的代码

为了避免读操作时应用程序界面失去响应，单独创建一个读输入报告的线程。该线程的作用就是使用 ReadFile()函数不断地从设备读取数据，然后根据读取到的数据设置按键情况及计数值。由于使用的是异步调用，因而在调用 ReadFile()函数

时提供一个 Overlapped 的结构，该结构中含有一个事件的句柄。事件的创建以及 Overlapped 结构的初始化在主窗口初始化时完成。

当读报告线程被创建时，设备还没有打开，因而等待事件触发。当设备打开后，手动设置事件为有效状态，从而唤醒读报告线程。此时读报告线程检测到设备已经打开，使用 ReadFile()函数请求设备输入数据。然后读报告线程等待事件被触发，进入挂起状态。当 ReadFile()函数完成后，会设置事件为有效状态，从而唤醒读报告线程。接着读报告线程就可以对返回的数据做相关处理了。数据处理完毕后，又可以再次调用 ReadFile()函数来读取数据，从而实现一直请求数据输入的功能。

需要注意的是，要让该线程能够开始读取报告，必须要在打开设备后手动触发一次，或者调用一次读数据的 ReadFile()函数，不然它就会一直等待事件的触发。

对返回的数据处理，主要是检查返回的数据量以及报告 ID 是否正确，如果正确，那么就设置界面上各开关的状态以及计数器的值。

由于该函数并不是 CMyUsbHidTestAppDlg 类(就是该工程的主窗口类)中的成员函数，所以无法直接调用 CMyUsbHidTestAppDlg 类中的成员函数。在创建该线程时，通过 pParam 参数传递了一个 this 指针，将参数 pParam 强制转化为 CMyUsbHidTestAppDlg 类的指针即可访问 CMyUsbHidTestAppDlg 类中的成员函数。

读报告的线程代码如下：

```
//读报告的线程
UINT ReadReportThread(LPVOID pParam)
{
 CMyUsbHidTestAppDlg * pAppDlg;
 DWORD Length, Counter;
 UINT i;
 CString Str;

 //将参数 pParam 取出，并转换为 CMyUsbHidTestAppDlg 型指针，以供下面调用其成员函数
 pAppDlg = (CMyUsbHidTestAppDlg * )pParam;

 //该线程是个死循环，直到程序退出时，它才退出
 while(1)
 {
  //设置事件为无效状态
  ResetEvent(ReadOverlapped.hEvent);

  //如果设备已经找到
  if(MyDevFound == TRUE)
  {
   if(hReadHandle == INVALID_HANDLE_VALUE) //如果读句柄无效
   {
    pAppDlg->AddToInfOut("无效的读报告句柄，可能打开设备时失败");
   }
```

```
else   //否则,句柄有效
{
 //则调用 ReadFile()函数请求 9 字节的报告数据
 ReadFile(hReadHandle,
          ReadReportBuffer,
          9,
          NULL,
          &ReadOverlapped);
}
//等待事件触发
WaitForSingleObject(ReadOverlapped.hEvent,INFINITE);
//如果等待过程中设备被拔出,也会导致事件触发,但此时 MyDevFound 被设置为假
//因此如果在这里判断 MyDevFound 为假,则就进入下一轮循环
if(MyDevFound == FALSE) continue;
//如果设备没有被拔下,则是 ReadFile()函数正常操作完成
//通过 GetOverlappedResult()函数来获取实际读取到的字节数
GetOverlappedResult(hReadHandle,&ReadOverlapped,&Length,TRUE);
//如果字节数不为 0,则将读到的数据显示到信息框中
if(Length! = 0)
{
 pAppDlg->AddToInfOut("读取报告" + pAppDlg->itos(Length) + "字节");
 Str = "";
 for(i = 0;i<Length;i ++ )
 {
  Str + = pAppDlg->itos(ReadReportBuffer[i],16).Right(2) + " ";
 }
 pAppDlg->AddToInfOut(Str, FALSE);
}

//如果字节数为 9,则说明获取到了正确的 9 字节报告
if(Length == 9)
{
 //第一字节为报告 ID,应该为 0
 if(ReadReportBuffer[0] == 0)
 {
  //第二字节为按键状态,将其保存到 KeyStatus 中
  KeyStatus = ReadReportBuffer[1];
  //刷新按键的情况
  pAppDlg->SetKeyStatus();
  //第三、四、五、六字节为设备返回的发送次数值。计算出值后并显示
```

```
        Counter = ReadReportBuffer[5];
        Counter = (Counter<<8) + ReadReportBuffer[4];
        Counter = (Counter<<8) + ReadReportBuffer[3];
        Counter = (Counter<<8) + ReadReportBuffer[2];
        pAppDlg->SetCounterNumber(Counter);
       }
      }
     }
     else
     {
      //阻塞线程,直到下次事件被触发
      WaitForSingleObject(ReadOverlapped.hEvent,INFINITE);
     }
    }
    return 0;
}
//////////////////////////////End of function/////////////////////
```


5.6.4 写输出报告的代码(发送LED的状态)

当用鼠标点击界面上的LED灯时,将触发对应按钮的单击事件。在事件响应代码中,将当前LED设置的状态组织成报告的格式,再通过WriteFile()函数发送给设备。设备接收到报告之后就会根据报告中的数据来设置板上LED的状态。

发送输出报告的函数代码如下:

```
//发送LED的状态。
BOOL CMyUsbHidTestAppDlg::SendLedStatus()
{
 BOOL Result;
 UINT LastError;
 UINT i;
 CString Str;

 //如果设备没有找到,则返回失败
 if(MyDevFound == FALSE)
 {
  AddToInfOut("设备未找到");
  return FALSE;
 }

 //如果句柄无效,则说明打开设备失败
 if(hWriteHandle == INVALID_HANDLE_VALUE)
 {
```

```
 AddToInfOut("无效的写报告句柄,可能是打开设备时失败");
 return FALSE;
}
//如果数据仍在发送中,则返回失败
if(DataInSending == TRUE)
{
 AddToInfOut("数据正在发送中,暂时不能发送");
 return FALSE;
}
//设置要发送报告的数据
WriteReportBuffer[0] = 0x00;                    //报告 ID 为 0
WriteReportBuffer[1] = LedStatus;               //将 LED 状态放到缓冲区中
//显示发送数据的信息
AddToInfOut("发送输出报告 9 字节");
Str = "";
for(i = 0;i<9;i ++ )
{
 Str + = itos(WriteReportBuffer[i],16).Right(2) + " ";
}
AddToInfOut(Str,FALSE);
//设置正在发送标志
DataInSending = TRUE;
//调用 WriteFile()函数发送数据
Result = WriteFile(hWriteHandle,
                   WriteReportBuffer,
                   9,
                   NULL,
                   &WriteOverlapped);
//如果函数返回失败,则可能是真的失败,也可能是 I/O 挂起了
if(Result == FALSE)
{
 //获取最后错误代码
 LastError = GetLastError();
 //看是否是真的 I/O 挂起
 if((LastError == ERROR_IO_PENDING)||(LastError == ERROR_SUCCESS))
 {
  return TRUE;
 }
 //否则,是函数调用时发生错误,显示错误代码
 else
```

```
    {
     DataInSending = FALSE;
     AddToInfOut("发送失败,错误代码:" + itos(LastError));
     //如果最后错误为1,说明该设备不支持该函数
     if(LastError == 1)
     {
      AddToInfOut("该设备不支持 WriteFile 函数。", FALSE);
     }
     return FALSE;
    }
   }
   //否则,函数返回成功
  else
  {
   DataInSending = FALSE;
   return TRUE;
  }
}
//////////////////////////End of function//////////////////////
```

5.6.5 写输出报告线程的代码

写输出报告线程的功能比较简单,它只是简单地等待事件被触发,然后清除一个正在发送数据的标志 DataInSending。在主线程中,需要发送数据时先检查 DataInSending 标志,如果可以发送数据,则调用 WriteFile()函数将数据发送到设备,并设置 DataInSending 标志为忙状态。数据发送完成后,就会触发事先设置的事件,从而唤醒写报告线程。写报告线程唤醒后就将数据正在发送的标志 DataInSending 清除,又进入到等待事件的状态。具体代码如下:

```
//写报告的线程
UINT WriteReportThread(LPVOID pParam)
{
 while(1)
 {
  //设置事件为无效状态
  ResetEvent(WriteOverlapped.hEvent);
  //等待事件触发
  WaitForSingleObject(WriteOverlapped.hEvent,INFINITE);
  //清除数据正在发送标志
  DataInSending = FALSE;
  //WriteReportBuffer[2]为非0值时将让设备清除它的计数值,
```

```
    //当点击清除计数器按键时,将会设置该值为非0,等数据发送完毕后,将它改回0
    //这样在发送LED状态时,就可以不用去设置WriteReportBuffer[2]的值了
    WriteReportBuffer[2] = 0;
  }
  return 0;
}
//////////////////////////////End of function//////////////////////
```

5.6.6 线程的创建以及设备插拔事件的注册

使用多线程时,需要在合适的时候创建好需要的线程。本程序在窗口显示时分别创建读报告线程和写报告线程,并同时将窗口的this指针作为参数传递给它们,让它们能够对窗口进行相关操作。这两个线程一旦被创建,就一直处于运行状态。因为里面是一个死循环,只有当程序退出时才被销毁。

另外,异步访问时需要给ReadFile()和WriteFile()函数提供一个Overlapped的结构体。该结构体在使用之前要先初始化,包括设置读数据的偏移量、事件的句柄等。通过调用创建事件的函数来获取一个事件的句柄,然后将其赋给Overlapped结构体中的事件句柄成员。本实例中分别给ReadFile()和WriteFile()函数创建了Overlapped结构体和对应的事件。

另外,当设备被插入或者拔下时,应用程序应该要能够感知到,并做出相关动作。要实现这个功能,就需要事先申请注册接收该设备类状态改变时产生的通知。注册设备状态改变通知使用的函数为RegisterDeviceNotification()。注册成功后,当设备的状态发生改变(例如插、拔)时,事件的接收者(在注册时指定)就会收到一个WM_DEVICECHANGE的消息。

以上这些初始化处理工作都在窗口显示时完成,具体的代码如下:

```
//初始化写报告时用的Overlapped结构体
//偏移量设置为0
WriteOverlapped.Offset = 0;
WriteOverlapped.OffsetHigh = 0;
//创建一个事件,提供给WriteFile使用,当WriteFile完成时,会设置该事件为触发状态
WriteOverlapped.hEvent = CreateEvent(NULL,TRUE,FALSE,NULL);

//初始化读报告时用的Overlapped结构体偏移量设置为0
ReadOverlapped.Offset = 0;
ReadOverlapped.OffsetHigh = 0;
//创建一个事件,提供给ReadFile使用,当ReadFile完成时,会设置该事件为触发状态
ReadOverlapped.hEvent = CreateEvent(NULL,TRUE,FALSE,NULL);

//创建写报告的线程(处于挂起状态)
pWriteReportThread = AfxBeginThread(WriteReportThread,
                                    this,
                                    THREAD_PRIORITY_NORMAL,
```

```
                                        0,
                                        CREATE_SUSPENDED,
                                        NULL);
    //如果创建成功,则恢复该线程的运行
    if(pWriteReportThread! = NULL)
    {
     pWriteReportThread->ResumeThread();
    }
    //创建一个读报告的线程(处于挂起状态)
    pReadReportThread = AfxBeginThread(ReadReportThread,
                                        this,
                                        THREAD_PRIORITY_NORMAL,
                                        0,
                                        CREATE_SUSPENDED,
                                        NULL);
    //如果创建成功,则恢复该线程的运行
    if(pReadReportThread! = NULL)
    {
     pReadReportThread->ResumeThread();
    }
    //获取 HID 设备的接口类 GUDI
    HidD_GetHidGuid(&HidGuid);
    //设置 DevBroadcastDeviceInterface 结构体,用来注册设备改变时的通知
    DevBroadcastDeviceInterface.dbcc_size = sizeof(DevBroadcastDeviceInterface);
    DevBroadcastDeviceInterface.dbcc_devicetype = DBT_DEVTYP_DEVICEINTERFACE;
    DevBroadcastDeviceInterface.dbcc_classguid = HidGuid;
    //注册设备改变时收到通知
    RegisterDeviceNotification(m_hWnd,
                               &DevBroadcastDeviceInterface,
                               DEVICE_NOTIFY_WINDOW_HANDLE);
```

5.6.7 对设备状态改变事件的处理

前面说过,成功注册了接收某类设备状态改变时的通知后,当设备状态改变时就会收到一个 WM_DEVICECHANGE 消息。在该消息处理中,可以获知设备的状态,例如设备插入并已经可用,设备已经拔下等。另外,还可以获知发生状态改变的设备的路径名。通过对比路径名,可以判断发生状态改变的设备是否是我们指定的设备。如果是,则做相关处理。例如,如果检测到指定的设备已经被拔出,则设置设备未找到,并关闭原来打开过的设备。具体的实现代码如下:

```
//设备状态改变时的处理函数
LRESULT CMyUsbHidTestAppDlg::OnDeviceChange(WPARAM nEventType, LPARAM dwData)
{
 PDEV_BROADCAST_DEVICEINTERFACE pdbi;
 CString DevPathName;
```

```
//dwData 是一个指向 DEV_BROADCAST_DEVICEINTERFACE 结构体的指针，
//在该结构体中保存了设备的类型、路径名等参数。通过跟我们指定设备的路径名比较，
//即可以判断是否是我们指定的设备拔下或者插入了
pdbi = (PDEV_BROADCAST_DEVICEINTERFACE)dwData;
switch(nEventType)                          //参数 nEventType 中保存着事件的类型
{
 //设备连接事件
 case DBT_DEVICEARRIVAL:
  if(pdbi->dbcc_devicetype == DBT_DEVTYP_DEVICEINTERFACE)
  {
   DevPathName = pdbi->dbcc_name;       //保存发生状态改变的设备的路径名
   //比较是否是我们指定的设备
   if(MyDevPathName.CompareNoCase(DevPathName) == 0)
   {
    AddToInfOut("设备已连接");
   }
  }
 return TRUE;
 //设备拔出事件
 case DBT_DEVICEREMOVECOMPLETE:
  if(pdbi->dbcc_devicetype == DBT_DEVTYP_DEVICEINTERFACE)
  {
   DevPathName = pdbi->dbcc_name;       //保存发生状态改变的设备的路径名
   //比较是否是我们指定的设备
   if(MyDevPathName.CompareNoCase(DevPathName) == 0)
   {
    AddToInfOut("设备被拔出");
    //设备被拔出，应该关闭设备(如果处于打开状态的话)，停止操作
    if(MyDevFound == TRUE)
    {
     MyDevFound = FALSE;
     OnCloseDevice();
    }
   }
  }
 return TRUE;
default:
 return TRUE;
}
}
//////////////////////////////End of function/////////////////////
```

5.7 软件界面以及使用方法

最终设计的软件界面如图 5.7.1 所示。当打开设备后，就可以使用鼠标点击

LED按钮来控制学习板上LED的情况了(图中LED7被点亮)。下面一行则显示学习板上按键的情况,当某个键被按住时,对应的键就会变成红色。下方的信息框显示更详细的操作、数据、时间等信息。旁边的VID、PID、PVN分别对应着厂商ID、产品ID、产品版本号。也可以在这里填入另外一个HID设备的ID值,例如USB鼠标、USB键盘等,看能否找到。至于自己的HID设备的各种ID值如何去找,通过前面这么多内容的学习,应该知道了吧!圈圈知道至少有四种方法可以找到:设备管理器中、注册表中、本程序查找设备时、使用Bus Hound抓包分析。不过,由于USB鼠标、USB键盘是系统独占设备,所以该应用程序将无法打开它们。对于USB鼠标和键盘,虽然以写访问方式能够打开,但是在调用WriteFile()函数时也会出错,错误原因是不支持WriteFile()函数。

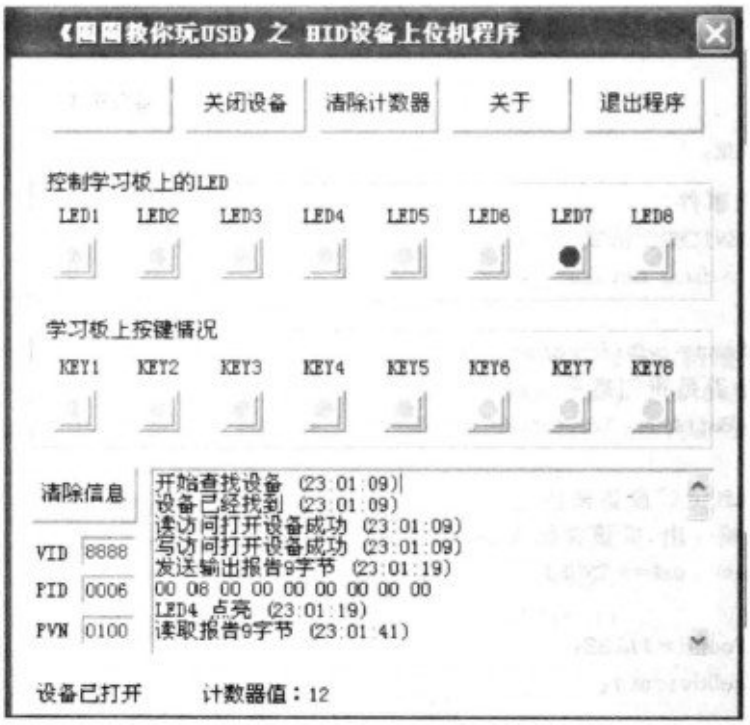

图5.7.1 软件的界面图

5.8 本章小结

本章通过用户自定义的USB HID设备实例,介绍了用户自定义的USB HID设备的实现方法,并详细介绍了访问HID设备的上位机软件的设计方法及思路。本章内容以代码为主,因为这个操作思路是比较简单的,主要是需要知道一些函数的具体用法。如果要学会编写访问USB设备的上位机软件,这些API函数是必须要掌握的,至少要知道有这些函数。至于函数的具体参数、格式,问题不大,因为可以使用MSDN帮助文档或者上网搜索来找到。