第8章

U 盘

U盘这个东西大家都很熟悉了，是一种移动存储设备。早在几年前，软盘还比较流行，几乎每台PC上都有一个软盘驱动器。U盘的出现和发展，打破了这种格局，现在的PC上几乎没人再装软驱了。U盘刚开始出现时，容量只有几十MB，经过短短几年的发展，容量已达到几G甚至更大。

8.1 USB大容量存储设备

在USB协议中，规定了一类大容量存储设备(mass storage device)，U盘就属于大容量存储设备。USB大容量存储设备还包括USB移动硬盘、USB移动光驱等。大容量存储设备的接口类代码(bInterfaceClass字段)为0x08，接口子类代码(bInterfaceSubClass字段)有好几种，但大部分U盘都使用代码0x06，即SCSI透明命令集。协议代码(bInterfaceProtocol字段)有3种：0x00、0x01、0x50，前两种需要使用中断传输，最后一种仅使用批量传输。本实例将使用最后一种协议。

既然是大容量存储设备，那么必须要有一个大容量存储器，它可以是FLASH、硬盘、光盘等。在我们的实验板上，有一个IDE接口，可以在上面挂一块IDE接口的硬盘来作为存储器。不过，考虑到很多读者并没有闲置的硬盘，所以本章第一个实例将在8952单片机的内部模拟一个FAT16的文件系统，做一个假U盘。第二个实例将使用IDE接口的硬盘来作为存储器，即IDE转USB。

将第7章的USB MIDI键盘实例复制一份，改名为UsbDisk。之所以选择它作为修改的模板，是因为它使用了端点2。将key.c、song.c文件从工程中移除，并将代码中无关的部分也删除，这里不再用到它们了。

8.2 设备描述符

设备描述符只需要修改产品ID号(PID)即可，这里是第九个实验，我们改为0x0009。其他字段保持不变。

8.3 字符串描述符

产品字符串改为“《圈圈教你玩 USB》之假 U 盘”，产品序列号改为“2008 - 08 - 14”。注意，如果是制作真正的 U 盘，每块 U 盘的序列号都要不一样，否则可能会几块相同的 U 盘无法同时使用。其实圈圈这里设置的这个产品序列号是不符合 USB 仅批量传输协议的，协议里规定该序列号最后至少有 12 位十六进制的数据位。不过 Windows 对此检查并不严格，所以还是可以正常使用。如果是做产品，最好还是按照协议来做。

8.4 配置描述符集合

先将配置描述符集合中与 MIDI 键盘相关的类特殊描述符删除，包括类特殊接口描述符、类特殊端点描述符等。只保留配置描述符、第一个接口描述符和两个标准端点描述符。同时要修改描述符集合的总长度。

8.4.1 配置描述符

配置描述符仅需修改接口数量，原来的 MIDI 键盘为 2 个接口，这里只有一个，将配置描述符的接口数量(bNumInterfaces 字段)改为 0x01。

8.4.2 接口描述符

该接口使用两个批量端点，修改 bNumEndpoints 字段为 0x02。该接口所使用的类为大容量存储设备(代码为 0x08)，修改 bInterfaceClass 字段为 0x08。子类为 SCSI 透明命令集(代码为 0x06)，修改 bInterfaceSubClass 字段为 0x06。协议使用仅批量传输(bulk only transport)协议(代码为 0x50)，修改 bInterfaceProtocol 字段为 0x50。修改好的接口描述符代码如下：

```
/*******************接口描述符*********************/
 //bLength 字段。接口描述符的长度为 9 字节
 0x09,
 //bDescriptorType 字段。接口描述符的编号为 0x04
 0x04,
 //bInterfaceNumber 字段。该接口的编号，第一个接口，编号为 0
 0x00,
 //bAlternateSetting 字段。该接口的备用编号，为 0
 0x00,
 //bNumEndpoints 字段。非 0 端点的数目。该接口有 2 个批量端点
```

```
0x02,
//bInterfaceClass 字段。该接口所使用的类。大容量存储设备接口类的代码为 0x08
0x08,
//bInterfaceSubClass 字段。该接口所使用的子类。SCSI 透明命令集的子类代码为 0x06
0x06,
//bInterfaceProtocol 字段。协议为仅批量传输,代码为 0x50
0x50,
//iConfiguration 字段。该接口的字符串索引值。这里没有,为 0
0x00,
```

8.4.3 端点描述符

端点描述符使用原来的端点 2 描述符即可,不用修改。

8.5 测 试

在描述符改完之后,接下来似乎就不知道该改什么了。事实上,在大容量存储设备的仅批量传输协议中规定了两个类特殊请求:Bulk-Only Mass Storage Reset 和 Get Max LUN。这里为了增加感性认识,先不实现这两个类请求,而是打开调试信息,根据调试信息显示的内容进入下一步操作。如图 8.5.1 所示,就是修改好描述符,运行后返回的调试信息。从图中可以看到,主机发送了一个类输入请求,通过查看 USB 大容量存储设备的协议,知道它正是前面提到过的 Get Max LUN 请求。另外,使用 BUS Hound 捕捉数据时,发现发送了一个 Set Interface 的请求,但是设备并没有收到(调试信息中未显示出来),估计这个请求是上层驱动发给下层驱动的,而不是发送给设备。

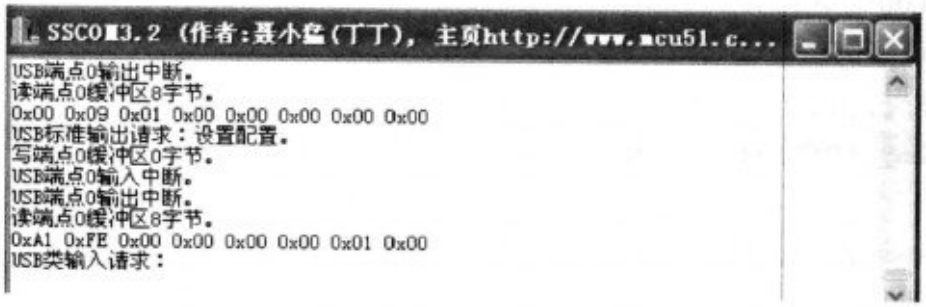

图 8.5.1 修改好描述符后的调试信息

8.6 类特殊请求

在USB大容量存储设备的Bulk Only Transport协议中，规定了两个类特殊请求：Bulk-Only Mass Storage Reset和Get Max LUN。前者是复位到命令状态的请求，后者是获取最大逻辑单元请求。

8.6.1 Get Max LUN请求

Get Max LUN请求的格式如表8.6.1所列。由bmRequestType可知，它是发送到接口的类输入请求，bRequest值为0xFE，wIndex的值为请求的接口号，本实例中为接口0。传输的数据长度为1字节，设备将在数据过程返回1字节的数据。该字节表示设备有多少个逻辑单元，值为0时表示有一个逻辑单元，为1时表示有两个逻辑单元，以此类推，最大可以取15。通过与前面的调试信息相对照，可以知道刚刚的那个类输入请求正是这个Get Max LUN请求。

表8.6.1 Get Max LUN请求的格式

bmRequestType	bRequest	wValue	wIndex	wLength	Data
10100001b	11111110b	0000h	Interface	0001h	1 byte

在端点0输出中断处理函数中的类输入请求分支下，增加对该请求的响应代码。首先要定义一个最大逻辑单元的变量。跟描述符类型一样，我们使用数组的方式来定义，尽管它只有一个元素。本实例仅实现一个逻辑单元，所以返回的值为0。因为只有一个逻辑单元，所以在后面的命令处理中，就忽略了逻辑单元这个字段。如果将最大逻辑单元改成1，那么系统将会认为有两个逻辑单元，从而分别读两次磁盘数据，结果读到了一样的数据(因为对命令处理时，忽略了逻辑单元字段，返回了一样的数据)，那么就会显示两块参数和内容一样的磁盘。改成n(最大15)，就可以得到$n+1$块磁盘，感兴趣的读者可以自己修改来试试(圈圈曾尝试过设置n为最大值15，经过一个漫长的枚举过程后，“我的电脑”中就多了16块新硬盘，场面很是壮观，圈圈长这么大还第一次见那么多的硬盘，盘符都到“Y”了)。新增的部分代码如下：

```
switch(bRequest)
{
 case GET_MAX_LUN:                          //请求为 GET_MAX_LUN(0xFE)
  #ifdef DEBUG0
   Prints("获取最大逻辑单元。\r\n");
  #endif

  pSendData = MaxLun;                       //要返回的数据位置
  SendLength = 1;                           //长度为 1 字节
```

```
    //如果请求的长度比实际长度短,则仅返回请求长度
    if(wLength<SendLength)
    {
     SendLength = wLength;
    }
    //将数据通过 EP0 返回
    UsbEp0SendData();
   break;
```

8.6.2 Bulk-Only Mass Storage Reset 请求

Bulk-Only Mass Storage Reset 请求是通知设备接下来的批量端点输出数据为命令块封包 CBW(Command Block Wrapper),其结构如表 8.6.2 所列。在这个请求处理中,仅需要设置一下状态,说明接下来的数据为 CBW,然后返回一个 0 长度的状态数据包即可。

表 8.6.2 Bulk-Only Mass Storage Reset 请求的结构

bmRequestType	bRequest	wValue	wIndex	wLength	Data
00100001b	11111111b	0000h	Interface	0000h	none

实现该请求的代码如下:

```
switch(bRequest)
{
 case MASS_STORAGE_RESET:
  #ifdef DEBUG0
   Prints("大容量存储设备复位。\r\n");
  #endif

  //接下来的数据为命令阶段(CBW)
  TransportStage = COMMAND_STAGE;
  //返回一个 0 长度的数据包
  SendLength = 0;
  NeedZeroPacket = 1;
  //将数据通过 EP0 返回
  UsbEp0SendData();
 break;
```

8.7 仅批量传输协议的数据流模型

前面的工作完成之后,接下来就是通过批量端点传输数据了。在仅批量传输协

议中，规定了数据传输的结构和过程，共分成三个阶段：命令阶段、数据阶段和状态阶段。这有点类似控制传输，但又不完全相同。命令阶段由主机通过批量端点发送一个 CBW（命令块封包）的结构，在 CBW 中定义了要操作的命令以及传输数据的方向和数量。数据阶段的传输方向由命令阶段决定，而状态阶段则总是由设备返回该命令完成的状态。

8.7.1 命令块封包 CBW 的结构

CBW（Command Block Wrapper）的结构如表 8.7.1 所列，总共有 31 字节。

表 8.7.1 CBW 的结构

位 / 字节	7	6	5	4	3	2	1	0
0～3	dCBWSignature							
4～7	dCBWTag							
8～11（08h～0Bh）	dCBWDataTransferLength							
12（0Ch）	bmCBWFlags							
13（0Dh）	保留（值为 0）				bCBWLUN			
14（0Eh）	保留（值为 0）				bCBWCBLength			
15～30（0Fh～1Eh）	CBWCB							

dCBWSignature 该字段为 CBW 的标志，为字符串 USBC（即 USB 命令的缩写）。用 ASCII 码来表示就是 0x55、0x53、0x42、0x43。如果用小端模式的 4 字节整数来表示，其值就是 0x43425355。

dCBWTag CBW 的标签，由主机分配，设备在完成该命令返回状态时，需要在 CSW（命令状态封包）中的 dCSWTag 字段中填入命令的 dCBWTag。

dCBWDataTransferLength 需要在数据阶段传输数据的字节数。小端结构，即低字节在先。

bmCBWFlags CBW 的标志。最高位（D7）表示数据传输的方向，0 表示输出数据（从主机到设备），1 表示输入数据。其他位为 0。

bCBWLUN 该字段为目标逻辑单元的编号。当有多个逻辑单元时，使用该字段来区分不同的目标单元。该字段仅使用低 4 位，高 4 位为保留值 0。

bCBWCBLength CBWCB 的长度。该字段仅使用 5 位，有效的取值范围为 1～16。不同的命令（由 CBWCB 决定），其长度可能是不一样的。如果命令的长度不足 16 字节，则后面的部分值为 0。

CBWCB 需要执行的命令，由选择的子类决定使用哪些命令。

8.7.2 命令状态封包 CSW 的结构

CSW(Command Status Wrapper)的结构如表 8.7.2 所列,共有 13 字节。

表 8.7.2 CSW 的结构

字节 \ 位	7	6	5	4	3	2	1	0
0～3	dCSWSignature							
4～7	dCSWTag							
8～11(8－Bh)	dCSWDataResidue							
12(Ch)	bCSWStatus							

dCSWSignature 该字段为 CSW 的标志,为字符串 USBS(即 USB 状态的缩写)。用 ASCII 码来表示就是 0x55、0x53、0x42、0x53。如果用小端模式的 4 字节整数来表示,其值就是 0x53425355。

dCSWTag 该命令状态封包的标签,其值为 CBW 中的 dCBWTag,响应哪个 CBW,就设置为哪个 CBW 的 dCBWTag。

dCSWDataResidue 命令完成时的剩余字节数。它表示实际完成传输的字节数与主机在 CBW 中设置的长度 dCBWDataTransferLength 之间的差额。

bCSWStatus 命令执行的状态。0x00 表示命令成功执行,0x01 表示命令执行失败,0x02 表示阶段错误。其他值为保留值。

通常,命令都能够成功完成,这时只需要设置前面两个字段为相应的值,后面两个字段都设置为 0 即可。

8.7.3 对批量数据的处理

第一次批量数据,肯定是 CBW。定义一个缓冲区,用来接收命令块封包 CBW。然后进入到数据阶段,在数据阶段中,对 CBW 进行解码,返回或者接收相应的数据。数据发送或接收完毕后,进入到状态阶段,返回命令执行的情况。然后再次进入命令阶段,等待主机发送 CBW 命令块封包。

8.8 SCSI 命令集和 UFI 命令集

SCSI(Small Computer System Interface)是小型计算机系统接口的缩写,有一套完整的协议规定其命令和命令数据的响应。SCSI 命令有很多,但是在 U 盘中常用的就几个:INQUIRY 命令、READ CAPACITY 命令、READ(10)命令、WRITE(10)命令等。

然而,虽然在接口子类代码(bInterfaceSubClass 字段)中指定使用 SCSI 透明命

令集，但是圈圈却发现 Windows 并不按照这一套来搞，而使用的是 USB 定义的 UFI 协议。但是如果将接口子类代码指定为 UFI(即 0x04)，Windows 的标准驱动又认不出来，弄不明白系统是咋回事。没办法，这里只好指定接口子类代码为 SCSI 透明命令集，而实际使用 UFI 协议的命令。虽然程序中的文件名、函数及变量名用的是 SCSI，但实际上用的是 UFI 命令集，读者要注意这一点。另外，它又并不是完全遵循 UFI 协议，例如在命令中的逻辑单元一项始终为 0，不过好在逻辑单元可以从 CBW 中获取；还有主机会发送一个操作代码为 0x1A 的命令，这在 UFI 协议中找不到(在 SCSI 命令中倒是有)，只好返回命令执行失败。

USB 软盘接口 UFI(USB Floppy Interface)命令是结合 SCSI－2 和 SFF－8070i 命令集而抽取出来的一些命令，所以在 SCSI 协议以及 SCSI 块设备命令中能够找到很多命令的定义，主要是一些强制命令。

UFI 命令出现在 CBW 的 CBWCB 字段中，最多可以有 16 字节，如果不足 16 字节，多余的将被忽略，通常设置为 0。所有 UFI 命令的第一字节(在 CBW 中偏移量为 15)为操作代码，在程序中根据不同的操作代码进行处理。

以下分别对要用到的几个 UFI 命令以及返回数据的格式作详细介绍，另外还有一些 UFI 命令没有在此详述，可以参看 USB UFI 协议文档。

8.8.1 查询命令 INQUIRY

INQUIRY 命令请求目标设备的一些基本信息，其格式如表 8.8.1 所列，操作代码为 0x12。其中 EVPD 字段和页码字段只支持 0。分配的缓冲区长度为主机接收该命令返回数据所分配的缓冲区，缓冲区长度通常为 0x24(即 36 字节)。

表 8.8.1 INQUIRY 命令的格式

位 / 字节	7	6	5	4	3	2	1	0
0	操作码(0x12)							
1	逻辑单元号			保留(0)				EVPD(0)
2	页码(只支持 0)							
3	保留(0)							
4	为返回数据分配的存储空间长度，通常为 36 字节							
5～11	保留							

INQUIRY 命令返回的数据为 36 字节，如表 8.8.2 所列。外设类型为 0 表示直接寻址设备(例如磁盘)。

表 8.8.2 INQUIRY 命令响应数据格式

字节 \ 位	7	6	5	4	3	2	1	0
0	保留 0			外设类型				
1	RMB	保留						
2	ISO 版本号(0)		ECMA 版本号(0)			ANSI 版本号(0)		
3	保留				响应数据格式(0x01)			
4	附加数据长度(31 字节)							
5～7	保留							
8～15	厂商信息							
16～31	产品信息							
32～35	产品版本信息(n. nn)							

RMB 位表示存储媒介是否可移除，0 为不可移除，1 为可移除。

如果设置为不可移除的，那么将显示在“我的电脑”中的“硬盘”区；如果设置为可移除的，那么将显示在“有可移动存储的设备”区，两者的图标也不一样，如图 8.8.1 所示。

图 8.8.1 不同 RMB 属性时的磁盘

ISO、ECMA、ANSI 等各种版本号规定为 0，“响应数据格式”为 0x01。

附加数据长度是后面附加数据的长度，为 31 字节。

厂商信息、产品信息、产品版本号可以根据自己的需要设置。

具体的返回信息内容可参看源代码中的 DiskInf 数组，它在 SCSI.c 文件中。

8.8.2 读格式化容量命令 READ FORMAT CAPACITIES

READ FORMAT CAPACITIES 命令可让主机读取设备各种可能的格式化容量的列表，如果设备中没有存储媒介，则设备返回最大能够支持的格式化容量。该命令的格式表 8.8.3 所列。

表 8.8.3 READ FORMAT CAPACITIES 命令格式

字节＼位	7	6	5	4	3	2	1	0
0	操作代码(0x23)							
1	逻辑单元号			保留				
2～6	保留							
7	分配的缓冲区长度(高字节)							
8	分配的缓冲区长度(低字节)							
9	保留							
10	保留							
11	保留							

表 8.8.4 是没有存储媒介时返回最大格式化容量的数据格式。其中，容量列表长度为 8 字节(即后面的 8 字节)，描述符代码为 3。容量的计算方法为

$$容量 = 块数 \times 每块字节数$$

另外，还有几种格式化容量的列表格式，本实例用不到，读者可以参看 UFI 协议文档。

通常每块字节数为 512 字节(即 0x200)，块数可以设置得大一些，它表示该设备最大支持的格式化容量，实际的容量要由存储媒介来决定。

表 8.8.4 返回的最大格式化容量格式

字节＼位	7	6	5	4	3	2	1	0
0	保留							
1	保留							
2	保留							
3	容量列表的长度							

续表 8.8.4

字节 \ 位	7	6	5	4	3	2	1	0
4	(MSB)							
5								
6				块数(高字节在先)				
7								(LSB)
8	保留					描述符代码		
9	(MSB)							
10				每块字节数(高字节在先)				
11								(LSB)

8.8.3 读容量命令 READ CAPACITY

READ CAPACITY 命令可以让主机读取到当前存储媒介的容量，其格式如表 8.8.5 所列，操作代码为 0x25。该命令除了操作代码为 0x25 之外，其他各字段的值都为 0。

READ CAPACITY 命令的数据响应格式如表 8.8.6 所列。最后逻辑块地址为存储媒介能够被访问的最大逻辑块地址，块(逻辑块)大小为每个逻辑块的字节数，通常为每块 512 字节。磁盘的总容量计算方法为

磁盘容量(字节数)=(最大逻辑块地址+1)×块大小

由于逻辑块地址是从 0 开始的，所以需要加 1。

READ FORMAT CAPACITIES 命令读到的是设备所支持最大的容量，而 READ CAPACITY 命令读到的才是实际的磁盘容量，注意二者的不同。

表 8.8.5 READ CAPACITY 命令格式

字节 \ 位	7	6	5	4	3	2	1	0
0	操作代码(0x25)							
1	逻辑单元号			保留				RelAdr
2	(MSB)							
3								
4				逻辑块地址(LBA)				
5								(LSB)

续表 8.8.5

位 / 字节	7	6	5	4	3	2	1	0
6	保留							
7	保留							
8	保留							PMI
9	保留							
10	保留							
11	保留							

表 8.8.6 READ CAPACITY 命令返回数据格式

位 / 字节	7	6	5	4	3	2	1	0
0	(MSB)							
1	最后逻辑块地址							
2	最后逻辑块地址							
3	最后逻辑块地址							(LSB)
4	(MSB)							
5	块大小(字节数)							
6	块大小(字节数)							
7	块大小(字节数)							(LSB)

8.8.4 READ(10)命令

主机通常使用 READ(10)命令来读取实际的磁盘数据，其命令格式如表 8.8.7 所列，操作代码为 0x28。另外还有一个 READ(12)命令(操作代码为 0xA8)，它的格式跟 READ(10)命令差不多，仅传输长度字段不一样，它的字节 6～9 都为传输长度，而 READ(10)命令只有字节 7～8 为传输长度。

其中 DPO、FUA、RelAdr 等字段都为 0 值。

逻辑块地址字段的值为需要读取数据的起始块的地址。在磁盘设备中，读、写通常都是按照块来操作的(所以叫做块设备)。一般来说，一个逻辑块就是一个扇区，大小为 512 字节。当然也有其他大小的逻辑块，例如在光盘文件系统中一个块就是 2 048字节。

传输长度字段的值为需要传输的逻辑块的数量，实际传输的字节数为传输长度值乘以每块大小。

设备根据命令中指定的逻辑块地址，从存储媒介中读取数据并通过批量端点返回。当全部数据都返回后，再返回命令执行状态 CSW。

表 8.8.7 READ(10)命令格式

字节 \ 位	7	6	5	4	3	2	1	0
0	操作代码(0x28)							
1	逻辑单元号			DP0	FUA	保留		RelAdr
2	(MSB) 逻辑块地址							
3								
4								
5								(LSB)
6	保留							
7	传输长度(MSB)							
8	传输长度(LSB)							
9	保留							
10	保留							
11	保留							

8.8.5 WRITE(10)命令

主机通常使用 WRITE(10)命令往设备写入实际的磁盘数据，其命令格式如表 8.8.8 所列，操作代码为 0x2A。另外还有一个 WRITE(12)命令(操作代码为 0xAA)，它的格式跟命令 WRITE(10)差不多，仅传输长度字段不一样，它的字节 6～9 都为传输长度，而 WRITE(10)命令只有字节 7～8 为传输长度。

该命令的各参数跟 READ(10)命令类似，请参看 READ(10)命令。

主机在发送此命令之后，接着就会发出实际要传送的数据，设备在收到全部数据后，返回命令执行的情况 CSW。

表 8.8.8 WRITE(10)命令格式

字节 \ 位	7	6	5	4	3	2	1	0
0	操作代码(0x2A)							
1	逻辑单元号			DP0	FUA	保留		RelAdr
2	(MSB) 逻辑块地址							
3								
4								
5								(LSB)

续表 8.8.8

位 / 字节	7	6	5	4	3	2	1	0
6	保留							
7	传输长度(MSB)							
8	传输长度(LSB)							
9	保留							
10	保留							
11	保留							

8.8.6 REQUEST SENSE 命令

REQUEST SENSE 命令用来探测上一个命令执行失败的原因，主机可在每个命令之后使用该命令来读取命令执行的情况。其命令代码为 0x03，格式如表 8.8.9 所列。

表 8.8.9 REQUEST SENSE 命令的格式

位 / 字节	7	6	5	4	3	2	1	0
0	操作代码(0x03)							
1	逻辑单元号			保留				
2	保留							
3	保留							
4	为返回数据分配的缓冲区长度							
5～11	保留							

该请求返回的数据格式如表 8.8.10 所列。其中 Valid 字段指示信息字段是否符合 UFI 规范，Valid 为 1 时，说明信息字段符合 UFI 规范。在 UFI 协议中，信息字段通常用来返回哪个逻辑块地址出现了错误。Sense Key、Additional Sense Code (ASC)、Additional Sense Code Qualifier(ASCQ)表示出错的代码，可以在 UFI 协议的最后找到。在本实例中，仅返回一种错误原因：无效的命令操作码，其 Sense Key 为 0x05，ASC 为 0x20，ASCQ 为 0。当然，对于一个实际的 U 盘，可能会有更多的出错原因，这就需要根据具体的情况来决定了。

表 8.8.10 REQUEST SENSE 命令返回的数据格式

位 / 字节	7	6	5	4	3	2	1	0
0	Valid	错误代码(固定为 0x70)						
1	保留							
2	保留				Sense Key			

续表 8.8.10

字节 \ 位	7	6	5	4	3	2	1	0
3	(MSB)							
4				信息				
5								
6								(LSB)
7	附加长度(固定为 10 字节)							
8～11	保留							
12	Additional Sense Code(ASC)							
13	Additional Sense Code Qualifier(ASCQ)							
14～17	保留							

8.8.7 TEST UNIT READY 命令

TEST UNIT READY 命令用来测试设备的某个逻辑单元是否准备好，操作代码为 0x00，其格式如表 8.8.11 所列。该命令的响应比较简单，如果设备已经准备好，则在状态阶段返回命令执行成功；否则返回命令执行失败。当主机使用 REQUEST SENSE 命令来探测错误原因时，设置 Sense Key 为 NOT READY。本实例中总是返回成功，即逻辑单元已准备好。

表 8.8.11 TEST UNIT READY 命令格式

字节 \ 位	7	6	5	4	3	2	1	0
0	操作代码(0x00)							
1	逻辑单元号			保留				
2～11	保留							

8.9 FAT 文件系统

由于在实验板上没有额外的 FLASH 等存储设备，因而本实验要在 ROM 中模拟一个 FAT 文件系统。如果有存储设备，则可以让 Windows 帮忙格式化，而不用自己建立文件系统。FAT 文件系统稍微有点复杂，读者可以在网上去搜索一篇叫做《4.5 万字透视 FAT32 系统》的文章，里面说得比较详细，当然也可以参看官方的 FAT 文件系统文档。在这里仅简单地说说在模拟 FAT 文件系统时需要用到的一些内容。

FAT(File Allocation Table)是文件分配表的缩写，是为了方便文件的存储、检

索、添加和删除等操作而提出的一种链表式的文件组织结构。如果给每个文件分配同样大小的存储空间这显然是不现实的，因此在 FAT 文件系统中，文件大小是不固定的。这些文件是以簇为最小单位，将每个簇号以链表的格式保存在一张专门的表格中，这张表格就叫做 FAT。目录也可以看作一个特殊的文件，这个文件被分成一个个的目录项，每个目录项保存了文件的文件名、文件长度、创建日期、起始簇号等重要信息。每个逻辑磁盘都有一个根目录，所有的子目录和文件都放在根目录下。在 FAT12/16 文件系统中，根目录位于整个磁盘的文件数据开始处，这样系统就能够找到根目录。通过读取根目录的数据，就能找到各个文件（包括目录）的目录项并打开文件。

一块磁盘应该包括主引导记录 MBR（Master Boot Record）、扩展引导记录 EBR（Extended Boot Record）、磁盘操作系统引导记录 DBR（DOS Boot Record）、文件分配表（File Allocation Table）、根目录区和文件数据区等几个重要部分。

其中，主引导记录 MBR 中记录了 MBR 引导代码、磁盘分区表等重要信息。当主引导记录中的磁盘分区表不够用时，就要用到扩展引导记录 EBR。DBR 则是每个逻辑分区的一个引导记录，里面记录了该分区的众多重要信息以及引导代码，所以十分重要。每个 MBR、EBR、DBR 通常分别只占用一个扇区（512 字节）。

在 U 盘系统中，可以没有 MBR 和 EBR，而仅有 DBR。在 DBR 之后，就是 FAT 区（通常有两个，一个为副本），接着就是文件（包括目录）数据区。FAT12/16 文件系统跟 FAT32 文件系统的根目录区有点不一样，FAT32 的根目录是不固定大小的，跟普通目录一样，而 FAT12/16 的根目录是固定大小的，所以通常在 FAT12/16 文件系统中把根目录区单独列出来。

在 FAT 文件系统中，使用的都是小端结构，即低字节在先。

本实例将使用 FAT16 文件系统，以下内容仅针对 FAT16 文件系统（除非特别说明），FAT32 所使用的结构跟 FAT16 的有所差别。

8.9.1 关于 DBR

DBR 占据逻辑分区的 0 扇区（逻辑块地址），大小通常为 512 字节，DBR 各个部分的意义如表 8.9.1 所列。

表 8.9.1 中跳转指令为引导操作系统时用，如果该分区不作为启动盘，可以不用理会该字段。但是操作系统在识别磁盘时，会检查这些值是否有效，所以不能任意设置，可以找个现成的、已经格式化好的 U 盘来查看这些数据是什么（用 BUS Hound 很容易捕捉），然后添加进来。OEM 字段为厂商定义的字符串，通常为格式化该分区的操作系统的类型以及版本号，为了向旧版本兼容，通常 Windows 格式化时会设置该字段为字符串 MSDOS5.0。每扇区字节数就是一个扇区的字节数（也就是前面所说的块大小），可以选择为 512、1 024、2 048、4 096，通常为 512 字节。每簇扇区数就是一簇所含有的扇区数，必须为 2 的整数次方，并且要保证每簇字节数不大于

32 KB(在本实例中,每簇扇区数设置为 32,每扇区字节数设置为 512,因此每簇大小为 16 KB)。保留扇区数在 FAT16 文件系统中通常规定为 1,即扇区 0 的 DBR。FAT 数为该分区中 FAT 的个数,通常为两个,另一个是备份。根目录项数就是根目录有多少个目录项,每个目录项为 32 字节,该值乘以 32 必须为每扇区字节的整数倍,即要求所有的根目录项刚好放入整数个扇区内,为了保持兼容性,该值最好设置为 512。小扇区数和大扇区数用来表示该分区有多少个扇区,当值小于 65 536 时,使用小扇区数;否则使用大扇区数。"每 FAT 扇区数"表示每个 FAT 占用多少个扇区,在 FAT16 中,每个 FAT 表项为 2 字节,因此每 FAT 扇区数就决定了总共有多少个 FAT 表项,也就决定了该分区最多可能的簇数。隐藏扇区数为该分区之前隐藏的扇区数,具体的值由操作系统格式化时决定,可以设置为 0。DBR 最后两字节必须为 0x55、0xAA,否则为无效。

表 8.9.1 FAT16 文件系统的 DBR 的结构

偏移量	字段长度	字段名	说 明
0x00	3	跳转指令	跳转到引导代码处的跳转指令
0x03	8	OEM	厂商定义的字符串
0x0B	2	每扇区字节数	每扇区的字节数,通常为 512 字节
0x0D	1	每簇扇区数	每簇的扇区数
0x0E	2	保留扇区数	保留的扇区数,FAT16 通常为 1
0x10	1	FAT 数	该分区的 FAT 数量,通常为 2(一个备份)
0x11	2	根目录项数	该字段仅被 FAT12/16 使用,FAT32 为 0
0x13	2	小扇区数	该分区的扇区数量(小于 65535 个扇区时使用)
0x15	1	媒体描述符	0xF8 表示硬盘,0xF0 表示高密度 3.5 寸软盘
0x16	2	每 FAT 扇区数	每个 FAT 所占用的扇区数量,只被 FAT12/16 使用
0x18	2	每道扇区数	每个磁道上的扇区数量
0x1A	2	磁头数	磁头的数量
0x1C	4	隐藏扇区数	引导扇区之前的扇区数
0x20	4	大扇区数	该分区的扇区数量(大于 65 535 个扇区时使用)
0x24	1	物理驱动器号	0x80 表示硬盘,0x00 表示软盘
0x25	1	保留	值为 0
0x26	1	扩展引导标签	为 0x29,表明接下来 3 个域可用
0x27	4	标卷序列号	由时间和日期构成的 32 位数
0x2B	11	磁盘标卷	与根目录的磁盘标卷一致,无标卷时为 NO NAME
0x36	8	文件系统类型	为字符串"FAT16"
0x3E	448	引导代码	启动操作系统的引导代码
0x1FE	2	有效结束标志	有效的结束标志为 0x55、0xAA

8.9.2 关于FAT表

在FAT16文件系统中，FAT表紧跟在DBR之后（因为只有一个保留扇区，即0扇区的DBR），也就是说，FAT表从1扇区开始。通常FAT有一个副本，该副本紧跟在第一个FAT之后。

FAT是一张保存了根据当前簇号就能找到下一簇号的表格。FAT是一块连续的数据区，在这块连续的数据区中分成很多大小一样的项，并从0开始编号。根据FAT文件系统类型的不同，分项时所使用的长度也不一样。FAT文件系统后面所跟的数字就表示分项时所使用的位（Bit）数，例如FAT16就表示每个项为16位，即2字节，而FAT32则为32位，即4字节。在每个项中，保存了文件所在的下一簇的簇号（或者文件结束标志）。由于要保存簇号，因而不同长度的项就对最大簇号有了不同的限制，例如FAT32就比FAT16能够表达更多的簇号，从而能够管理的分区容量也更大。理论上来说，FAT16能够分配的最大分区容量为65 536×32 KB=2 GB（实际上要比这个略微小一点，因为有些表项值有特殊意义）。

那么FAT的表项编号跟数据区之间有什么关系呢？前面提到过，文件空间分配是以簇为最小单位的，也就是说，整个数据区是被分成一个个簇的。而FAT呢？也刚好是把整个FAT区分成一个个项。如果我们把FAT区、根目录区分别当作一个簇（虽然其大小不是刚好等于一个簇，但是可以在逻辑上这样去划分），那么整个分区都被分成一个个簇了。将这些簇也从0开始编号（叫做簇号），那么每个簇刚好在FAT表中有个对应，即簇0（也就是FAT表区）对应着FAT表中的项0，而簇1（即根目录区）对应着FAT表中的项1，真正的数据区从簇2（项2）开始。

在文件目录项中，保存了一个文件（包括目录）的起始簇号，通过它可以找到文件的第一簇的数据，并且在该簇号对应的FAT表项中记录了下一簇的簇号。根据下一簇的簇号，又可以找到下一簇的数据，并且在该簇号对应的FAT表项中又记录了下下簇的簇号……按照这个关系，可以一直将整个文件的数据找出来。那么怎么知道文件已经结束了呢？有一些特殊的簇号（例如FAT16的0xFFF8～0xFFFF）来表示文件结束，当某个FAT表项中保存的簇号为文件结束簇号时，就知道该簇是该文件的最后一簇了。当然，簇的单位比较粗糙，不能精确到多少字节，在目录项中还有一个精确的文件长度，它保存了整个文件的字节数。如果最后一簇数据都已经读入后还不足目录项中所记录的文件长度时，则说明该文件已经损坏。

在FAT表项中，0表示该簇未被使用，0xFFF0～0xFFF6表示保留簇，0xFFF7表示该簇已经损坏，0xFFF8～0xFFFF表示文件最后一簇，其他簇号就用来表示该文件的下一簇的簇号。前面提到过，簇0和簇1分别可以看作FAT表区和根目录区，它们都已经被使用了，所以其对应的FAT表项没有实际的意义，因而规定FAT项0的值为0xFFF8，而项1的值为0xFFFF。当一个文件被删除时，并没有真正地

删除文件的数据，而只是将文件的目录项设置为删除状态，并将其占用的FAT表项设置为0。

8.9.3 关于目录项

目录也是一个特殊的文件(FAT12/16的根目录除外，它是固定的)，里面的数据被划分为一个个目录项。每个目录项大小都为32字节，其结构如表8.9.2所列。该结构为短文件名的格式，另外还有长文件名的格式。在这里仅介绍短文件名，并且在实例中也仅使用短文件名。

其中文件名为8个字节，不包括点和扩展名。扩展名由后面的3字节指定，而文件名和扩展名之间的点是由操作系统负责增加的。短文件名目录项的文件名和扩展名必须使用大写字母，如果文件名或扩展名长度不够，则用空格(0x20)来填充。注意文件名的第一个字节具有特殊意义。当第一字节为0时，表示该目录项为空，可以使用，并且其后面所有该目录下的目录项都为空。当第一字节为0xE5时，表示该目录项曾经被使用过，但是现在该文件已经被删除了，该目录项目前为空，可用。当第一字节为0x05时，表示ASCII为0xE5(在日文中为有效字符)的字符，避免系统误认为该文件已删除。如果文件名为".."，则表示当前目录；文件名为"..."，则表示其父目录。文件名的第一字节不可为空格(0x20)。

偏移为0x0B的字段为属性字节，它表示一个文件的属性，每个属性由一个二进制位指示。当某个二进制位为1时，就表示该文件具有这样的属性，例如一个只读的隐藏文件属性为00000011B，即0x03。

时间格式(16位)为：位15～11表示小时，可以取值为0～23；位10～5表示分，可以取值为0～59；位4～0表示秒，可以取值为0～29，每单位为2 s，即实际的秒值为该值的2倍。

日期格式(16位)为：位15～9表示年份，可以取值为0～127，它表示距离1980年差值，即实际的年份为该值加上1980，最大可表示到2107年；位8～5表示月份，可以取值为1～12；位4～0表示几号，可以取值为1～31。

在FAT16中，簇号只有2字节，因此只使用起始簇号的低字，高字部分为0，在FAT32中会使用高字。文件长度为4字节(32位)，因此单个文件最大长度为4 GB。注意目录项中的所有整数为小端结构，即低字节在先。

在每个分区的根目录下，有个特殊的目录项，就是磁盘标卷。它的属性字节值为0x08。其文件名就是所显示的磁盘名。

表 8.9.2 目录项的结构

偏移量	字节数	字段名	说 明
0x00	8	文件名	保存文件名的 8 字节(不包括扩展名和点)
0x08	3	扩展名	文件的扩展名
0x0B	1	属性字节(为右边所列出的属性的组合)	00000000B 可读/写文件(B 表示二进制,下同)
			00000001B 只读文件
			00000010B 隐藏文件
			00000100B 系统文件
			00001000B 标卷
			00010000B 子目录
			00100000B 归档
0x0C	1	保留	系统保留,值必须设置为 0
0x0D	1	创建时间(ms)	文件创建时的毫秒时间,协议中说单位为 1/10 s。但是又说取值为 0～199,不知是否搞错了
0x0E	2	创建时间	文件创建的时间
0x10	2	创建日期	文件创建时的日期
0x12	2	最后访问日期	文件最后被访问的日期。如果是写访问,必须要等于最后修改日期
0x14	2	起始簇号高字	该文件第一簇所在簇号的高字,FAT16 必须为 0
0x16	2	最后修改时间	该文件最后被修改的时间
0x18	2	最后修改日期	该文件最后被修改的日期
0x1A	2	起始簇号低字	该文件第一簇所在簇号的低字
0x1C	4	文件长度	该文件的长度(字节数)

8.10 模拟一个 FAT16 文件系统

如何来模拟一个 FAT16 文件系统呢？一个完整的 FAT16 系统需要 4 个部分：DBR、FAT 表、根目录和数据区。由于实验板上的 ROM 空间有限，仅设置关键部分的数据，其余部分数据设置为 0 即可。那么哪些部分的数据是关键的呢？分别是：整个 DBR、两个 FAT(包括一个备份)的前 6 字节(这里只设置一个文件，因而只用到表项 2)、根目录区的前两个目录项(系统标卷和一个文件)、数据区的文件内容部分。将这些关键数据以及 0 以数组的格式保存起来，在适当的时候返回即可。那么又怎样去判断何时返回这些必要数据和数据 0 呢？根据主机发送的 READ(10)命令的逻辑块地址以及当前返回的字节数即可判断出来。

本实例打算模拟一个 128 MB 的 U 盘，扇区大小为 512 字节，每簇大小为 16 KB（即每簇扇区数为 32），总共有 128 MB/16 KB＝8 K 簇，相应地，也需要这么多个 FAT 表项。FAT16 的每个 FAT 表项为 2 字节，因此一个扇区可以保存 256 个 FAT 表项，那么保存 8 K 个表项需要的扇区数为 8 192/256＝32 个，即每 FAT 扇区数为 32。根目录项数设置为 512 个，每个目录项需要 32 字节，因此总共需要 32 个扇区保存根目录。

根据以上分析，可以得出整个模拟磁盘的结构如下：0 扇区为 DBR（前 62 字节需要自己设计，后面的引导代码可以直接复制一个现成 U 盘的，注意跳转指令和后面的引导代码部分等不要设置为 0，否则会让操作系统认不出来或者提示未格式化），1～32 扇区为 FAT 表，33～64 扇区为 FAT 表备份，65～96 扇区为根目录区，97 扇区至结束为数据区。其中后面的 FAT、根目录、数据区等只有前面小部分数据为非 0，其他部分全部填充为 0，如表 8.10.1 所列。当主机使用 READ(10)读数据时，将逻辑块地址 LBA 转换为字节地址，然后查看它在哪个区间，就可以知道该返回什么数据了。当然，主机一次读操作至少为一个扇区，所以每发送完一个数据包后都要重新计算字节地址，并再次获取需要返回数据的地址。由于 D12 的端点 2 大小为 64 字节，所以这些数组至少为 64 字节，一次数据包返回才正确。对于连续的数据（例如 0 扇区的 DBR），在发送数据时会自动调整指针位置，只需要设置数据的起点位置即可。

表 8.10.1 模拟磁盘的结构

扇区地址	内 容	数 据	字节地址范围
0	DBR	前 62 字节关键数据以及后面的引导代码，整个 DBR 扇区为一个数组	0～511
1～32	FAT(1)	前 64 字节关键数据（仅前 6 字节的表项有用）	512～575
		填充 0	576～16 895
33～64	FAT(2)	前 64 字节关键数据（仅前 6 字节的表项有用）	16 896～16 959
		填充 0	16 960～33 279
65～96	根目录	前 64 字节关键数据（磁盘标卷以及测试文件）	33 280～33 343
		填充 0	33 344～49 663
97～结束	数据区	前半段测试文件数据（不足一扇区）	49 664～50 175
		填充 0	大于 501 765

8.11 实验结果

前面对这些命令的结构、返回数据格式都说得比较详细，这里就不再贴出处理的

源代码了，读者可以对照书中的内容来阅读光盘中的源代码，主要在 SCSI.c 和 FAT.c 两个文件中。由于实验板上没有存储设备，所以对于输出的数据就直接丢弃了。但是还是可以往这个 U 盘中复制数据的，并且少量数据还可以正常读回来，这主要是因为 Windows 的缓冲机制。为了加快访问速度，Windows 会将少量的文件数据缓冲在内存中，当下次读取时，就直接从缓冲区中读取，而不是直接访问磁盘。

当枚举成功后，会在任务栏的右下角出现拔下 U 盘的图标，双击它会弹出安全删除硬件的对话框。在这里可以看到在 INQUIRY 命令中返回的厂商信息和产品信息，如图 8.11.1 所示。

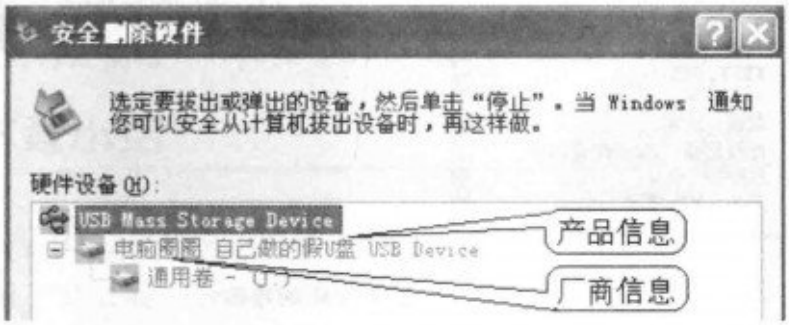

图 8.11.1 安全删除硬件的对话框

进入“我的电脑”，可以看到多出一块磁盘(如果你将 MXA LUN 设置为更大，可以获得多块磁盘)，右击，选择属性，可以看到它的大小为 127 MB，已用空间为 16 KB，如图 8.11.2 所示。因为里面有一个测试文件 TEST.TXT，所以占用了 16 KB 的空间。虽然这个文件只有 300 字节，但是前面说过，文件是以簇为最小单位分配空间的，而这里每簇大小为16 KB，所以要占用 16 KB 的空间。在磁盘上，很多小文件通常会占用很大的空间，就是因为按簇分配空间导致的，将它们压缩成单个文件，就会节省不少空间。

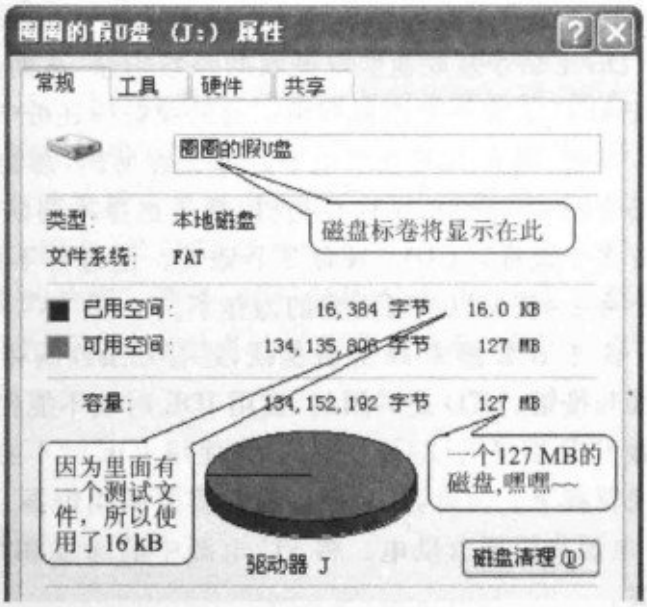

图 8.11.2 假 U 盘的属性

进入这个假 U 盘，可以看到里面有一个名为“TEST.TXT”的文本文件，这是在程序中模拟的一个文件。文件名、文件的创建时间、修改时间(在文件属性中查看)等信息保存在根目录的第二个目录项中，而文件中的内容保存在 TestFileData 数组中。打开这个文本文件，可以看到我们

所设置的文本内容，如图 8.11.3 所示。

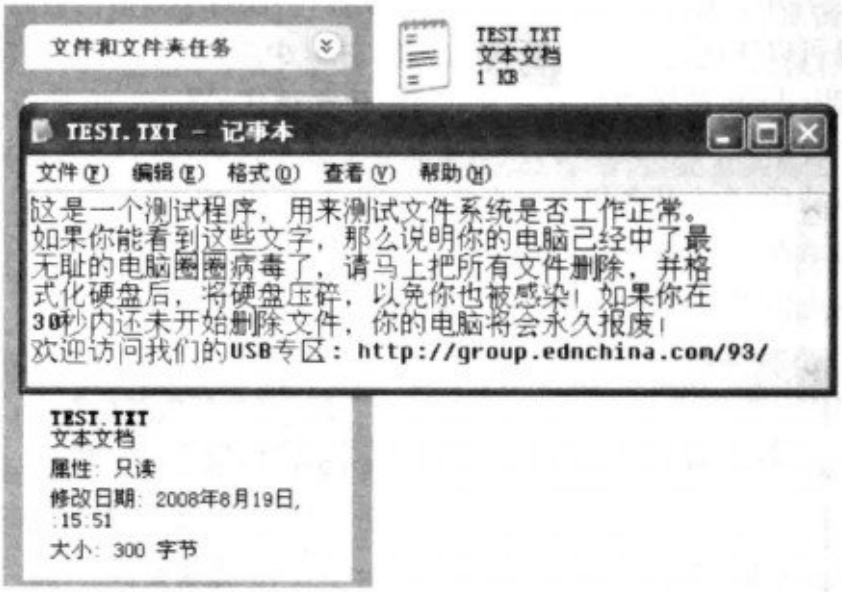

图 8.11.3 测试文件中的内容

8.12 IDE 转 USB 的实现

在上面假 U 盘的基础上，实现 IDE 转 USB 就很容易了。只要在 READ(10)命令处理中从硬盘读取数据，在 WRITE(10)命令处理中将数据写入到硬盘即可。原来程序中模拟磁盘的 DBR、FAT、根目录、数据等都不再需要了，因为它们都保存在硬盘中。当然磁盘容量不再是固定的了，而是由具体的硬盘来决定。可以使用 Identify Drive 命令从硬盘获取硬盘的参数信息，从而获得磁盘容量大小，然后在 READ CAPACITY 命令中返回即可。另外设备描述符中的 PID 应该也要改一改。

不过，操作 IDE 接口也不是那么容易的，需要查看 IDE 接口的文档。IDE 接口的操作跟 D12 的操作有点类似，都是选择不同的寄存器，然后操作。不同的是 IDE 具有多个地址，而 D12 只有 2 个地址。通过往不同的寄存器写不同的值，然后再写一个命令，就可以执行需要的操作了。

由于 IDE 需要 16 条数据线，还有几条控制和地址线，所以在学习板上 IDE 的数据线与按键、LED 是共用的，使用 IDE 时就不能使用按键和 LED 了。还有串口是用来做 IDE 接口的读/写控制的，在使用 IDE 时不能使用串口，并且需要将串口的两个跳线冒拔下。另外，IDE 硬盘通常需要两组电源(12 V 和 5 V)，可以找一个闲置的 PC 电源来给硬盘供电。将 PC 电源中的绿线和黑色的地线连在一起，就可以启动电源。

IDE 硬盘可以设置为逻辑块地址(LBA)访问方式，这样在读/写命令时，可以直接将收到的 LBA 地址写入到硬盘的 LBA 寄存器中。通过配置 Drive/Head 寄存器

来选择为 LBA 模式；另外，还有一个扇区计数(sector count)寄存器，用来保存读/写操作时需要传输的扇区个数。配合 LBA 寄存器、扇区计数寄存器就可以使用 Read Sector 或 Write Sector 命令来读或写扇区了。

需要注意的是，IDE 口的数据线是 16 位的，每次读/写操作都是 2 字节的数据，操作时序可以参看 IDE(ATA)协议。

具体如何实现在这里就不再详述了，仅在光盘中给出实现的程序，感兴趣的读者可以参考阅读。需要注意的是，如果读者使用的是 Cepark V2.0 的 PCB，那么应该使用光盘中“UsbTolde(Cepark V2.0 版)”下的代码。因为 Cepark V2.0 的这个 PCB 与圈圈设计的第一版 PCB 不一样，它将原来连接在 P1、P2 口的数据线移到了 P0、P1 口，并且将连接在 74HC573 上的 8 条数据线倒置了过来。

8.13 本章小结

本章通过对 U 盘的描述符以及命令的分析，实现了一个简单的假 U 盘，并在最后简单地介绍了 IDE 转 USB 的实现。当然，这里仅是一个实验性地实现，要真正实现一个产品级的 U 盘，还有很多路要走。例如，U 盘通常选择 FLASH 作为存储媒介，而 FLASH 有一个特点就是擦除必须按块操作。在没有大容量 RAM 缓冲的条件下实现 FLASH 块的擦写是不容易的。另外，如何做一个有效的算法来尽量平均地使用 FLASH 中的各块，以增加存储器的寿命也是一个难题。此外还要尽可能快地提高传输速度。

如今，自己再来开发 U 盘已经没有多大的商机，因为这东西现在几乎是白菜价了。使用通用 USB 芯片来开发 U 盘那更是不值得了，可以考虑使用专门的 U 盘芯片。但是，学习 U 盘的开发也并不是完全无用了，在很多场合需要在现有的系统中整合一个 U 盘的功能，例如：具有 USB 接口的手机、便携式多媒体播放器、仪表仪器(可以通过 U 盘的方式将数据导入到计算机)、固件更新(可以利用 U 盘功能，将固件直接复制进去来实现更新)等等。在这些场合可以由程序实现或者模拟一个 U 盘，而不宜直接使用专用的 U 盘芯片。因此，学习 U 盘开发的主要价值就在于在已有 USB 接口的基础上，增加一个 U 盘的功能。另外，对开发具有读/写 U 盘功能的嵌入式 USB 主机也有一定的帮助，因为当你知道设备的工作原理之后，设计主机的思路也就有了。