第 12 章

移植到 AVR 单片机上

有很多读者反映说书中所选的芯片 PDIUSBD12 太古老了，不容易买，而且还要外挂一个 MCU，都极力推荐圈圈用一颗较新的、带 USB 接口的 MCU 来做学习板。其实我们真正要学习的是 USB 的这种协议和实现方法，与具体的平台关系并不大。为了让大家学习如何将代码移植到其他平台，圈圈特意在第 2 版中增加了本书的最后几章。本章就将详细地介绍如何将之前的代码移植到 AVR 单片机 AT90USB162 中。

12.1 AT90USB162 芯片介绍

AT90USB162 芯片是一款集成了 USB2.0 全速设备的 8 位高性能、低功耗 AVR 单片机。它使用了先进的精简指令集(RISC)架构，具有 125 条功能强大的指令，而大部分指令都能在一个时钟周期执行完毕。有 32 个 8 位的通用寄存器，当运行在 16MHz 主频时，最大能达到 16MIPS 的运行速度。内置了 16K 字节的片内 FLASH，可支持一万次擦写。内置了 512 字节的 EEPROM，可支持 10 万次擦写。内置 512 字节的 SRAM。在 USB 方面，具有如下特点：

- 可通过 USB 口进行在线烧录升级。
- 内置 48MHz PLL 供 USB 全速模式使用。
- 完全独立的 176 字节 DPRAM 供 USB 端点使用。
- 端点 0(控制端点)的包长度支持 8 到 64 字节。
- 4 个可编程的输入/输出端点，可支持批量/中断/等时传输，包大小支持 8 至 64 字节，可选的单/双缓冲。
- 当 USB 总线复位时，可选择让 MCU 也跟着复位，而不用重新插拔设备。
- USB 数据线和 PS/2 数据线的引脚复用，可直接连接 USB 接口或 PS/2 接口。

这个芯片只需要很少的外围器件就可以构成一个 USB 开发板，同时由于具有 USB 在线烧录功能，连烧录器也省了，烧录软件直接使用 ATMEL 公司官方提供的 FLIP 软件即可，下载地址：http://www.atmel.com/dyn/products/tools_card.asp?tool_id=3886。在烧录时，首先将 PD7 口拉低，然后给系统上电，就进入到 USB 升级模式了，然后再通过 FLIP 软件，将编译好的固件下载到芯片中即可。需要注意的是，当要使用 USB 升级功能时，必须使用 16M 的晶体，并且芯片中出厂的引导程序

未被擦除。

AVR 的编译环境也有很多，在这里圈圈使用的是一个比较古老的软件：CodeVisionAVR. v2. 03. 4。

12.2　硬件抽象层的移植

这里我们以第 6 章的 USB 转串口为例，详细介绍如何将它移植到 AT90USB162 芯片中。

在原来的代码中，有 PDIUSBD12. h 和 PDIUSBD12. c 两个硬件抽象层的文件，更换芯片后，我们将这两个分别改名为 AT90USB. h 和 AT90USB. c，实现 AT90USB162 芯片的硬件抽象层。AT90USB. h 中主要是一些宏定义和函数声明，这里不再进行说明，请读者自行阅读光盘中的代码。由于 USB 控制器已经包含在单片机中了，因此原来在 D12 中的那些模拟硬件时序的函数和宏定义都可以不要了，直接改成寄存器访问的方式。为了方便将来的移植，我们这里将原来与硬件绑定的函数名进行了修改，改成了 UsbChip 为前缀，例如原来的读端点缓冲区的函数 D12ReadEndpointBuffer，改成了函数 UsbChipReadEndpointBuffer。这样下次载移植到其他方案时，对这些函数名就可以不再修改了。

由于每个 USB 芯片在使用之前都需要初始化，因此这里我们专门增加了一个 USB 芯片初始化的函数 UsbChipInit，在 main 函数中调用它进行 USB 部分的初始化工作。这个函数的实现代码如下：

```
/*****************************************
函数功能：USB 芯片初始化。
入口参数：无。
返　　回：无。
备　　注：无。
*****************************************/
void UsbChipInit(void)
{
 UsbDisconnect();                    //先断开 USB 连接
 PLLCSR = 0x04;                      //PLL clock Prescaler 为 2 分频
 PLLCSR| = 0x02;                     //启动 PLL
 while(! (PLLCSR&0x01));             //等待 PLL 启动完成
 USBCON = 0x00;                      //复位 USB 模块
 USBCON = 0x80;                      //使能 USB 模块
 UDPADDH = 0x00;                     //FIFO
 UPOE = 0x00;
 //disable all USB interrupts
```

```
    UDIEN = 0x00;
    UEIENX = 0x00;
    ConfigValue = 0;                    //配置值初始化为 0
    UsbConnect();                       //将 USB 连接上
}
//////////////////////End of function//////////////////////////
```

在该函数中，首先会调用函数 UsbDisconnect，将设备从 USB 总线上断开 USB 的连接，实际上就是将内部的 1.5 kΩ 上拉电阻断开，这样的话主机就会以为设备已经被拔出了，我们的设备就可以安心地做初始化工作而不被打断。另外，当我们将 MCU 复位后，也确实需要重新产生插拔事件，以让主机重新检测到我们的设备，因此在初始化时我们先断开 USB 连接。然后设置 USB 的 PLL 时钟预分频系数，由于在该芯片中，PLL 的倍频系数固定为 6，而我们的系统时钟为 16 MHz，因此要输出 48 MHz 的话，必须要先 2 分频。接着对 USB 模块进行复位，对一些寄存器、变量进行初始化等。之所以这里要关闭 USB 的中断，是因为跟原来的例子一样，这里也不使用中断，而是查询模式。然后调用 UsbConnect 函数，将内部的 1.5 kΩ 上拉电阻接到总线上，这样主机就能识别到有设备连接到总线上，并开始进入枚举过程了。UsbDisconnect 和 UsbConnect 函数的实现代码如下，至于具体的寄存器功能，请参看芯片的数据手册，里面描述得很详细。在断开连接的函数中，加延时 1 s 的目的是为了确保让主机检测到设备已经断开连接了。

```
/*****************************************
函数功能:USB 断开连接函数。
入口参数:无。
返　　回:无。
备　　注:无。
*****************************************/
void UsbDisconnect(void)
{
#ifdef DEBUG0
 Prints("断开 USB 连接。\r\n");
#endif
 UDCON = 0x01;              //Disconnect pull - up resistor
 DelayXms(1000);            //延迟 1 秒
}
//////////////////////End of function//////////////////////////
/*****************************************
函数功能:USB 连接函数。
入口参数:无。
返　　回:无。
```

```
备　　注:无。
*********************************************/
void UsbConnect(void)
{
#ifdef DEBUG0
 Prints("连接 USB。\r\n");
#endif
 USBCON = 0x80;              //使能时钟
 UDCON = 0x00;               //Connect pull - up resistor
}
//////////////////////End of function//////////////////////////
```

当主机检测到设备连接到 USB 总线上后，主机就会复位 USB 总线。在这个事件中，我们需要对芯片的端点进行复位，并设置配置值为 0，对一些变量进行重新初始化等。这部分代码在原来的例子中也有，读者可以对比阅读。下面是 USB 复位事件时的处理函数：


```
/*********************************************
函数功能:总线复位中断处理函数。
入口参数:无。
返　　回:无。
备　　注:无。
*********************************************/
void UsbBusReset(void)
{
#ifdef DEBUG0
 Prints("USB 总线复位。\r\n");
#endif
 UsbChipResetEndpoint();                    //复位端点
 ConfigValue = 0;                           //配置值初始化为 0
 UsbChipSetConfig(0);                       //设置芯片的配置值为 0
 NeedZeroPacket = 0;
 SendLength = 0;
 Ep1InIsBusy = 0;                           //复位后端点 1 输入缓冲区空闲。
 Ep3InIsBusy = 0;                           //复位后端点 3 输入缓冲区空闲。
}
//////////////////////End of function//////////////////////////
```

在这个函数中，调用到了 UsbChipResetEndpoint 函数和 UsbChipSetConfig 这两个函数，它们的实现代码都在 AT90USB.c 中，代码如下：

```
/*********************************************
函数功能:USB 端点复位。
```

```
入口参数:无。
返　　回:无。
备　　注:无。
*************************************************/
void UsbChipResetEndpoint(void)
{
 UERST = 0x1F;                                    //复位端点
 UERST = 0x00;                                    //复位端点完成

 UENUM = 0;
 UEINTX = 0x00;                                   //清除中断标志
 UECONX = 0x01;                                   //使能端点 0
 UECFG0X = 0x00;                                  //设置为控制输出端点
 UECFG1X = 0x02;                                  //设置为 8 字节、单缓冲,分配内存
 USBCON = 0x80;                                   //使能 USB 模块
}
//////////////////////////End of function//////////////////////////////

/*************************************************
函数功能:设置芯片配置状态。
入口参数:Value:配置值。
返　　回:无。
备　　注:无。
*************************************************/
void UsbChipSetConfig(uint8 Value)
{
 //无操作
 Value = 0;
}
//////////////////////////End of function//////////////////////////////
```

复位端点的函数中，对芯片的一些 USB 寄存器进行了操作，具体的含义请读者参看芯片的数据手册。需要注意的是，在这里必须要使能端点 0，并配置为控制端点，因为主机接下来就要通过端点 0 进行枚举了。由于这个芯片在设置配置时不需要进行任何操作，所以在这里 UsbChipSetConfig 函数其实什么事也没干。

那么对于非 0 端点，应该在什么地方使能呢？协议里面规定，只有在 SET_CONFIG 请求设置为非 0 的配置时，才能使能其他端点。因此，这里增加了一个使能非 0 端点的函数 UsbChipSetEndpointEnable，代码如下：

```
/*************************************
函数功能:使能端点函数。
入口参数:Enable: 是否使能。0 值为不使能,非 0 值为使能。
返　　回:无。
```

```
备    注:无。
*****************************************/
void UsbChipSetEndpointEnable(uint8 Enable)
{
 if(Enable! = 0)
 {
  UENUM = 1;
  UECONX = 1;                    //使能端点 1
  UECFG0X = 0xC1;                //设置为中断输入端点
  UECFG1X = 0x12;                //设置为 16 字节、单缓冲,分配内存
  UENUM = 2;
  UECONX = 1;                    //使能端点 2
  UECFG0X = 0xC0;                //设置为中断输出端点
  UECFG1X = 0x12;                //设置为 16 字节、单缓冲,分配内存
  UENUM = 3;
  UECONX = 1;                    //使能端点 3
  UECFG0X = 0x81;                //设置为批量输入端点
  UECFG1X = 0x32;                //设置为 64 字节、单缓冲,分配内存
  UENUM = 4;
  UECONX = 1;                    //使能端点 4
  UECFG0X = 0x80;                //设置为批量输出端点
  UECFG1X = 0x32;                //设置为 64 字节、单缓冲,分配内存
 }
 else
 {
 }
}
//////////////////////End of function//////////////////////////
```

当总线复位完成后,主机就会通过端点 0 进行枚举了,这时使用的是控制传输。控制传输首先要发送 SETUP 包,因此我们要从 USB 模块那里知道,当前端点 0 中的数据是否为 SETUP 包,所以增加了一个函数专门判断是否为 SETUP 包,代码如下:

```
/*****************************************
函数功能:判断是否是 SETUP 包。
入口参数:无。
返    回:0:不是 SETUP;非 0:是 SETUP。
备    注:无。
*****************************************/
int UsbChipIsSetup(uint8 Endp)
{
```

```
    UENUM = Endp;                          //选择端点
    if(UEINTX&(1 << 3))
    {
     return 0xFF;                          //是 setup 包,返回非 0
    }
    else
    {
     return 0;                             //不是 setup 包,返回 0
    }
}
//////////////////////////End of function//////////////////////////////
```

当我们收到 SETUP 包后，我们需要把数据从端点缓冲区中提取出来，这要用到函数 UsbChipReadEndpointBuffer。数据读出来之后，还要清除掉端点的缓冲区，这样它才能接收下一次数据，这要用到函数 UsbChipClearBuffer。而对于 SETUP 包，考虑到它的重要性，需要一个特殊的动作才能清除掉端点缓冲区，这要用到函数 UsbChipAcknowledgeSetup。这三个函数在原来的 D12 代码里也有实现，请读者对比阅读。以下是这三个函数的代码：


```
/*****************************************
函数功能:清除接收端点缓冲区的函数。
入口参数:无。
返　　回:无。
备　　注:只有使用该函数清除端点缓冲后,该接收端点才能接收新的数据包。
*****************************************/
void UsbChipClearBuffer(uint8 Endp)
{
 UENUM = Endp;                             //选择端点
 UEINTX = ~(1<<7);                         //清除 FIFOCON - FIFO Control Bit
 UsbLedBlink = 2;                          //闪烁 LED,表示有数据通信
}
//////////////////////////End of function//////////////////////////////
/*****************************************
函数功能:应答建立包的函数。
入口参数:Endp:端点号。
返　　回:无。
备　　注:无。
*****************************************/
void UsbChipAcknowledgeSetup(uint8 Endp)
{
```

```
    UENUM = Endp;                          //选择端点
    UEINTX = ~(1<<3);                      //清除 RXSTPI
    UsbLedBlink = 2;                       //闪烁 LED,表示有数据通信
}
//////////////////////End of function//////////////////////////////

/*****************************************
函数功能:读取端点缓冲区函数。
入口参数:Endp:端点号;Len:需要读取的长度;Buf:保存数据的缓冲区。
返    回:实际读到的数据长度。
备    注:无。
*****************************************/
uint8 UsbChipReadEndpointBuffer(uint8 Endp, uint8 Len, uint8 * Buf)
{
    uint8 i,j;
    UENUM = Endp;                          //选择端点
    j = UEBCLX;                            //获取数据长度
    if(j>Len)                              //如果要读的字节数比实际接收到的数据长
    {
      j = Len;                             //则只读指定的长度数据
    }
#ifdef DEBUG1                              //如果定义了 DEBUG1,则需要显示调试信息
    Prints("读端点");
    PrintLongInt(Endp);
    Prints("缓冲区");
    PrintLongInt(j);                       //实际读取的字节数
    Prints("字节。\r\n");
#endif
    for(i = 0;i<j;i++)
    {
      *(Buf + i) = UEDATX;                 //从 FIFO 中读一字节数据
#ifdef DEBUG1
      PrintHex(*(Buf + i));                //如果需要显示调试信息,则显示读到的数据
      if(((i + 1) % 16) == 0)Prints("\r\n"); //每 16 字节换行一次
#endif
    }
#ifdef DEBUG1
    if((j % 16)!= 0)Prints("\r\n");        //换行。
#endif
    return j;                              //返回实际读取的字节数。
}
//////////////////////End of function//////////////////////////////
```

当我们收到 SETUP 包之后，需要将数据返回给主机，这就需要用到把数据写到端点的函数 UsbChipWriteEndpointBuffer。很不幸的是，圈圈所选择的 AVR 编译器不能自动将指向 RAM 和指向 ROM 的指针进行转换，这样使用一个函数就无法实现发送 RAM 和 ROM 中的数据的功能了。因此圈圈写了两个函数，一个函数用来发送 RAM 中的数据，另外一个函数用来发送 ROM(Flash)中的数据，这两个函数的代码分别如下：

```
/*****************************************
函数功能：将处于 RAM 中的数据写入端点缓冲区函数。
入口参数：Endp：端点号；Len：需要发送的长度；Buf：保存数据的缓冲区。
返　　回：Len 的值。
备　　注：此函数从 RAM 中读取数据并写入到端点缓冲区。
*****************************************/
uint8 DataInRam_WriteEndpointBuffer(uint8 Endp,uint8 Len,uint8 * Buf)
{
 uint8 i;
#ifdef DEBUG1 //如果定义了 DEBUG1，则需要显示调试信息
 Prints("写端点");
 PrintLongInt(Endp);
 Prints("缓冲区");
 PrintLongInt(Len);                    //写入的字节数
 Prints("字节。\r\n");
#endif
 UENUM = Endp;                         //选择端点
 for(i = 0;i<Len;i++)
 {
  UEDATX = *(Buf + i);                 //将数据写到 FIFO 中
#ifdef DEBUG1
  PrintHex(*(Buf + i));                //如果需要显示调试信息，则显示发送的数据
  if(((i + 1) % 16) == 0)Prints("\r\n"); //每 16 字节换行一次
#endif
  }
#ifdef DEBUG1
 if((Len % 16)! = 0)Prints("\r\n");     //换行
#endif
 UsbChipValidateBuffer(Endp);          //使端点数据有效
 return Len;                           //返回 Len
}
//////////////////////End of function//////////////////////////
/*****************************************
函数功能：将处于 FALSH 中的数据写入端点缓冲区函数。
```

```
入口参数:Endp:端点号;Len:需要发送的长度;Buf:保存数据的缓冲区。
返　　回:Len 的值。
备　　注:此函数从 Flash 中读取数据并写入到端点缓冲区。
*****************************************/
uint8 DataInFlash_WriteEndpointBuffer(uint8 Endp,uint8 Len, flash uint8 * Buf)
{
 uint8 i;
#ifdef DEBUG1 //如果定义了 DEBUG1,则需要显示调试信息
 Prints("写端点");
 PrintLongInt(Endp);
 Prints("缓冲区");
 PrintLongInt(Len);                          //写入的字节数
 Prints("字节。\r\n");
#endif
 UENUM = Endp;                               //选择端点
 for(i = 0;i<Len;i++)
 {
  UEDATX = *(Buf + i);                       //将数据写到 FIFO 中
#ifdef DEBUG1
  PrintHex(*(Buf + i));                      //如果需要显示调试信息,则显示发送的数据
  if(((i + 1) % 16) == 0)Prints("\r\n");     //每 16 字节换行一次
#endif
 }
#ifdef DEBUG1
 if((Len % 16)! = 0)Prints("\r\n");          //换行
#endif
 UsbChipValidateBuffer(Endp);                //使端点数据有效
 return Len;                                 //返回 Len
}
//////////////////////End of function//////////////////////////
```

对于 UsbChipWriteEndpointBuffer 函数,我们用宏定义,将它定义成发送 RAM 中数据的函数 DataInRam_WriteEndpointBuffer。由于目前要返回 ROM 中常量的数据都是通过端点 0 发送的,所以我们只要在 UsbEp0SendData 函数中调用 DataIn-Flash_WriteEndpointBuffer 函数即可。在之前 D12 的代码中,UsbEp0SendData 函数中发送的数据是从 pSendData 指针中获取的,这里将 pSendData 改成了一个 32 位的整数,用它的最高 8 位来表示数据存储的类型。0 表示 FLASH 的数据,1 表示 RAM 中的数据。之所以这样选择,是因为原来的代码中,大部分都是 FLASH 中的常数,只有一个地方是 RAM 中的数据,这样代码修改的地方就很少。而它的低 16 位则用来表示数据存储的地址(即指针的值),因为这里只有 512 字节的 RAM 空间,

所以 16 位就足够了。这样就实现类似 Keil C51 中的 RAM 和 FLASH 空间指针自动识别的功能。

在上面两个函数中，我们还调用另外一个函数 UsbChipValidateBuffer，它是用来启动端点发送数据的函数。当我们把数据写入到端点缓冲区后，必须通知 USB 控制器："端点中数据已经写好了，主机下次来取时你就可以发送出去了"。这个函数在原来的 D12 代码中也有实现，请读者再次对比阅读。这个函数的代码如下：

```
/*****************************************
函数功能：使能发送端点缓冲区数据有效的函数。
入口参数：Endp：端点号。
返　　回：无。
备　　注：只有使用该函数使能发送端点数据有效之后，数据才能发送出去。
*****************************************/
void UsbChipValidateBuffer(unsigned char Endp)
{
 UENUM = Endp;              //选择端点
 if(Endp == 0)
 {
  //对于端点 0，清除端点 0 的输入完成中断标志位，将使能数据发送
  UEINTX = ~(1 << 0);
 }
 else                       //对于非 0 端点
 {
  UEINTX = ~(1 << 7);       //清除 FIFOCON - FIFO Control Bit 使能数据发送
 }
 UsbLedBlink = 2;           //闪烁 LED，表示有数据通信
}
//////////////////////End of function//////////////////////////
```

当主机发送设置地址命令后，要将收到的新地址写入到 USB 控制器中，这部分操作需要调用到一个函数 UsbChipWriteAddress，而在该函数中，又会调用到另外一个函数 UsbChipSetAddressStatus，让芯片启用新的地址。这两个函数的代码实现如下：

```
/*************************************
函数功能：设置芯片进入设置地址状态。
入口参数：Value：地址状态，非 0 为设置地址，0 为未设置地址。
返　　回：无。
备　　注：无。
*************************************/
void UsbChipSetAddressStatus(uint8 Value)
{
```

```
 if(Value == 0)
 {
  UDADDR& = ~(1 << 7);              //默认状态
 }
 else
 {
  UDADDR| = (1 << 7);               //进入到设置地址阶段
 }
}
//////////////////////////End of function//////////////////////////////

/*********************************************
函数功能:设置 USB 芯片功能地址函数。
入口参数:Addr:要设置的地址值。
返　　回:无。
备　　注:无。
*********************************************/
void UsbChipWriteAddress(uint8 Addr)
{
 UDADDR = Addr;                     //设置地址
 //等待前一个数据包(实际上是状态阶段)发送完毕
 //等待中断产生
 while(1)
 {
  if(UEINTX&(1 << 0))break;         //发送完毕
  if(UDINT&((1 << 0)||(1 << 3)))return;  //如果产生复位、挂起则直接返回
 }
 if(Addr! = 0)                      //如果地址非 0
 {
  UsbChipSetAddressStatus(1);       //设置进入到设置地址阶段
 }
}
//////////////////////////End of function//////////////////////////////
```

12.3　main.c 和 usbcore.c 的修改

在 main 函数中，对芯片的一些模块进行了初始化，然后在主循环中查询 USB 的各个中断寄存器，并调用相关的函数进行处理，这部分代码和之前的代码结构基本上是一样的，具体代码如下：

```
/*****************************************
函数功能:主函数。
```

```
入口参数:无。
返　　回:无。
备　　注:无。
*****************************************/
void main(void)
{
#ifdef DEBUG0
 int i;
#endif
 #asm("cli");
 MCUSR = 0;
 MCUSR &= ~(1 << 3);
 WDTCSR |= (1 << 4) | (1 << 3);
 /* Turn off WDT */
 WDTCSR = 0x00;
 SystemClockInit();              //系统时钟初始化
 LedInit();                      //LED 对应的管脚初始化
 Timer0Init();                   //定时器 0 初始化,用来产生 5ms 的定时扫描信号
 OnLed2();                       //电源指示灯
 Uart1Init();                    //串口 0 初始化
#ifdef DEBUG0
 for(i = 0;i<17;i++)             //显示头信息
 {
  Prints(HeadTable[i]);
 }
#endif
 UsbChipInit();                  //初始化 USB 部分
 while(1)
 {
  if(UDINT&(1 << 0))             //SUSPI - Suspend Interrupt Flag
  {
   UDINT = ~(1 << 0);            //清除中断
   UsbBusSuspend();              //总线挂起中断处理
  }
  if(UDINT&(1 << 3))             //EORSTI - End Of Reset Interrupt Flag
  {
   UDINT = ~(1 << 3);            //清除中断
   UsbBusReset();                //总线复位中断处理
  }
```

```
UENUM = 0;                          //选择端点 0
if(UEINTX&(3 << 2))   //FIFOCON - FIFO Control Bit//如果是 SETUP 包、输出数据等
{
 UsbEp0Out();                       //端点 0 输出中断处理
}
if((SendLength! = 0)||(NeedZeroPacket)) //如果端点 0 有数据要发送
{
 if(UEINTX&(1 << 0))                //并且端点 0 缓冲区空闲
 {
  UsbEp0In();                       //端点 0 输入中断处理
 }
}
UENUM = 1;                          //选择端点 1
if(UEINTX&(1 << 0))
{
 UsbEp1In();                        //端点 1 输入中断处理
}
UENUM = 2;                          //选择端点 2
if(UEINTX&(1 << 2))
{
 UsbEp2Out();                       //端点 2 输出中断处理
}
UENUM = 3;                          //选择端点 3
if(UEINTX&(1 << 0))
{
 UsbEp3In();                        //端点 3 输入中断处理
}
UENUM = 4;                          //选择端点 4
if(UEINTX&(1 << 2))
{
 UsbEp4Out();                       //端点 4 输出中断处理
}

if(ConfigValue! = 0)                //如果已经设置为非 0 的配置,则可以处理数据
{
 if(Ep3InIsBusy == 0)               //如果端点 3 空闲,则发送串口数据到端点 3
 {
  SendUartDataToEp3();              //调用函数将缓冲区数据发送到端点 3
 }
 if(UsbEp4ByteCount! = 0)           //端点 4 接收缓冲区中还有数据未发送,则发送到串口
 {
                                    //发送一字节到串口
```

```
            UartPutChar(UsbEp4Buffer[UsbEp4BufferOutputPoint]);
            UsbEp4BufferOutputPoint ++ ;              //发送位置后移 1
            UsbEp4ByteCount -- ;                      //计数值减 1
          }
        }
      }
    }
    ////////////////////////End of function//////////////////////////
```

在 usbcore.c 中，改动的内容并不多，主要是一些修饰变量的关键词、函数名、描述符的修改。例如，原来在 Keil C51 中，使用的 idata 关键词，这里就没有了；原来的 code 关键词，改成了 const 关键词。这主要是因为不同的嵌入式编译器之间存在着一些差异。如果不想一个个修改，也可以使用宏定义的方式进行修改。函数名方面，主要就是之前 D12 开头的一些函数，变成了 UsbChip 开头了。描述符方面，主要是端点、字符串等发生了改变。例如在设备描述符中，端点 0 最大包长由原来的 16 字节改成了 8 字节，PID 由原来的 0x0007 改成 0x2202；而配置描述符里面只修改了两个传输数据的批量端点的地址，原来在 D12 中使用的是输入端点 2/输出端点 2，而在这里修改成了输入端点 3/输出端点 4。而函数 UsbDisconnect 和 UsbCconnect 已经被移动到了硬件抽象层文件 AT90USB.c 中去了。而函数 DelayXms 则被移动到 LED.c 文件中去了。UsbBusReset 函数前面已经介绍过了，主要是对一些变量的初始化改变了。而函数 UsbEp0SendData 中，则增加了对不同指针类型的判断，从而调用不同的发送数据的函数，代码如下：

```
if((pSendData&0xFF000000) == 0x01000000)  //RAM 中的数据
{
 UsbChipWriteEndpointBuffer(0, DeviceDescriptor[7], (uint8 * )(pSendData&0xFFFF));
}
else //FLASH 中的数据
{
 DataInFlash_WriteEndpointBuffer(0,
                  DeviceDescriptor[7], (flash uint8 * )(pSendData&0xFFFF));
}
```

另外就是 pSendData 变量的定义和赋值方式发生了改变，原来是一个指针，这里改成了 uint32 类型。在赋值时，将目标指针进行了强制类型转换，并加上指针类型的标识。例如，对于 FLASH 中的数据指针，只要强制类型转换成 uint32 型即可，因为它的类型标识为 0。而 RAM 中的数据，在强制类型转换后，还需要加上 0x01000000，因为它的类型标识为 1。

另外还有两个批量端点数据的处理，分别增加了 UsbEp3In 和 UsbEp4Out 函数。当然，相应的缓冲区的变量名也进行了修改。

这样使用文字描述不够清晰，建议读者安装一个代码对比工具，例如 Winmerge 之类的软件，将两个工程的 usbcore.c 文件进行对比，就可以很清楚地看到代码中进行了哪些修改了。

当然，其他的一些功能也要进行相应的移植，例如控制 LED 灯、串口驱动、延时函数、按键扫描等。但由于这些与 USB 无关，因此这里不进行介绍，读者可以参考光盘中的代码。

总体来说，整个移植的工作量还是不大的，如果对芯片熟悉的话，应该用几个小时就可以移植完了。移植好的 USB 转串口代码见光盘中的 AT90USB162\Usb2Uart\文件夹。

12.4　USB 鼠标的移植

接下来我们再来介绍一下 USB 鼠标的移植。有了上面串口工程的基础，我们只要将 USB 鼠标中的一些描述符、类请求、数据发送等移植进来即可，同时将一些串口相关的无用请求删除。这些修改主要集中在 main.c 和 usbcore.c 文件中，例如增加 HID 设备的报告描述符，获取报告描述符的标准请求，设置空闲请求等。另外还把按键部分的代码也加了进来，不过圈圈并没有实现读取按键的函数 KeyGetValue，而是在 Timer0Isr 中，每隔 4 s 模拟一次鼠标右键。读者可以根据自己的硬件情况，来实现 KeyGetValue 函数。

移植好的 USB 鼠标代码见光盘中的 AT90USB162\UsbMouse\文件夹。

12.5　本章小节

本章介绍了将 USB 转串口和 USB 鼠标移植到 AVR USB 芯片的过程。由于代码很多是重复的，因此这里并没有进行非常详细的讲解，只抽取了部分重要内容进行说明。读者应该以光盘中的代码为主，使用文本对比工具，对比不同项目中代码的改动部分。也许还是会有读者会觉得 AT90USB162 这个芯片很老了，的确如此，现在的电子技术发展得很快，每天都有不同的产品推出。对于一个芯片，过时是迟早的事，但我们要掌握的是编程的思路和方法，以不变应万变。而且，读者也可以大胆尝试在新的芯片上去移植，去实践。另外，圈圈的代码结构也许并不好，而现在很多芯片原厂也会提供他们的 DEMO 源码，读者只要掌握了描述符的构造，数据的传输等内容，就很容易在这些 DEMO 代码上实现自己所需要的 USB 设备了。