



---

## STM32原生USB 移植(simon.yuan)

### 目录

<b>STM32原生USB固件的移植，由于水平有限，难免有不当之处，欢迎交流讨论 (QQ: 343264122, QQ群: 755127489)</b>
--



1	STM32 USB鼠标的移植.....	2
2	STM32 USB Device的PMA.....	5
3	STM32 USB Device固件库运行机制.....	9
4	STM32 USB鼠标移植结果.....	10
5	STM32 USB CDC移植.....	10

## 1 STM32 USB HID鼠标移植前准备

### a) 官方USB库下载

USB底层驱动源码自己动手写不现实(如果自己写也可以, 参考USB协议文档), 这里移植ST官网写好的底层USB固件库, 那么首先需要下载官方的USB底层固件库, 下载地址及下载解压的文件如下:

[https://my.st.com/content/my\\_st\\_com/zh/products/embedded-software/mcu-mpu-embedded-software/stm32-embedded-software/stm32-standard-peripheral-library-expansion/stsw-stm32121.license=1569923627897.product=STSW-STM32121.version=4.1.0.html](https://my.st.com/content/my_st_com/zh/products/embedded-software/mcu-mpu-embedded-software/stm32-embedded-software/stm32-standard-peripheral-library-expansion/stsw-stm32121.license=1569923627897.product=STSW-STM32121.version=4.1.0.html)

 STM32_USB-FS-Device_Lib_V4.1.0	2017/6/27 21:25	文件夹	
 en.stsw-stm32121.zip	2019/10/1 17:54	WinRAR ZIP 压缩...	6,353 KB

### b) 基于ST官方的USB例程进行移植, 在以上的链接中也需要下载一份ST官方USB的user manual(en. CD00158241.pdf). 文档的内容分为几部分:

- 1) STM32微控制器介绍
- 2) STM32全速USB的固件库的架构
- 3) STM32各种USB设备的demo介绍

### c) 移植过程

1. 创建两个工作组: ApplicationInterface及STM32\_USB-FS-Device\_Driver, 并按照ST官方说明的USB应用架构对其添加相应的源文件.

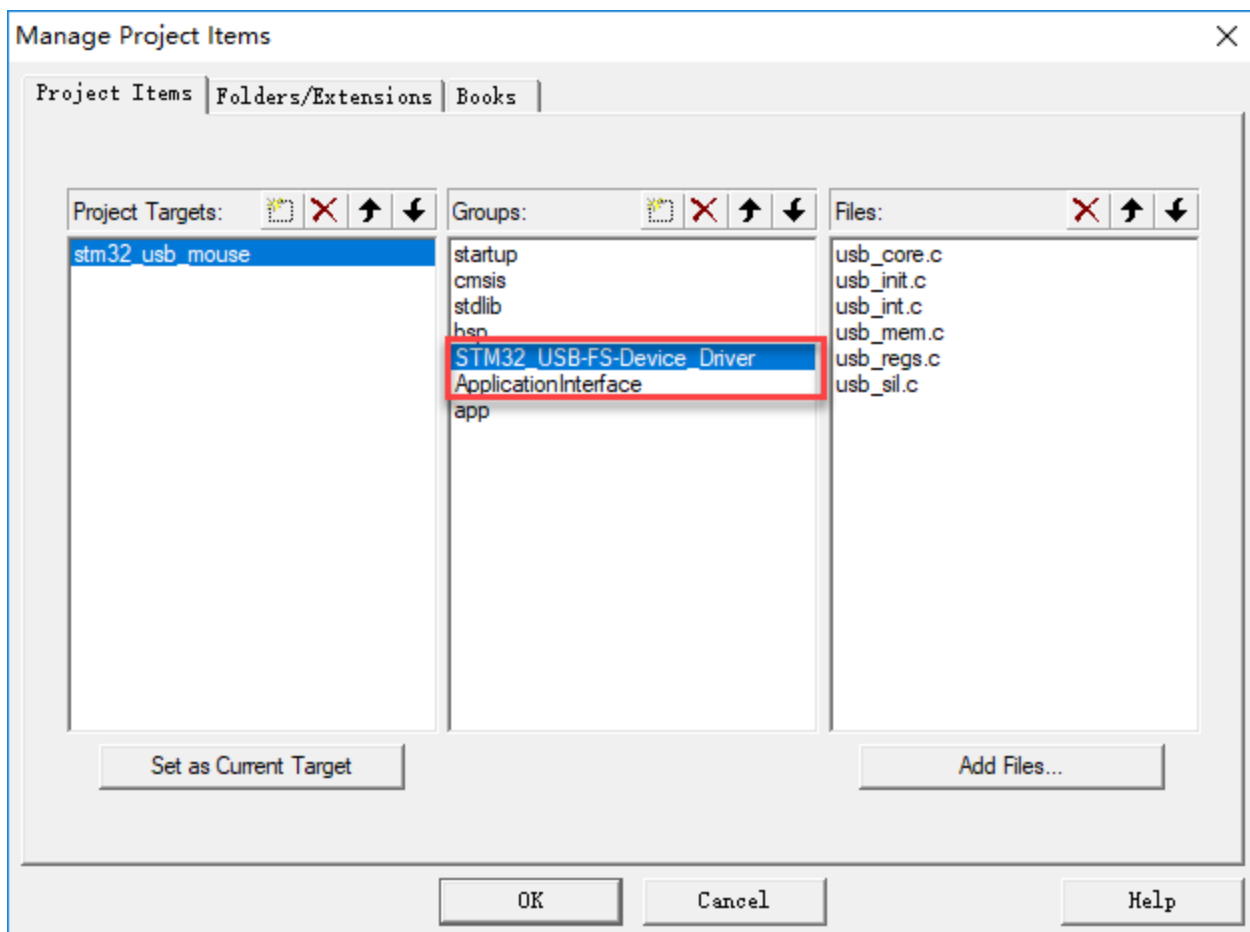
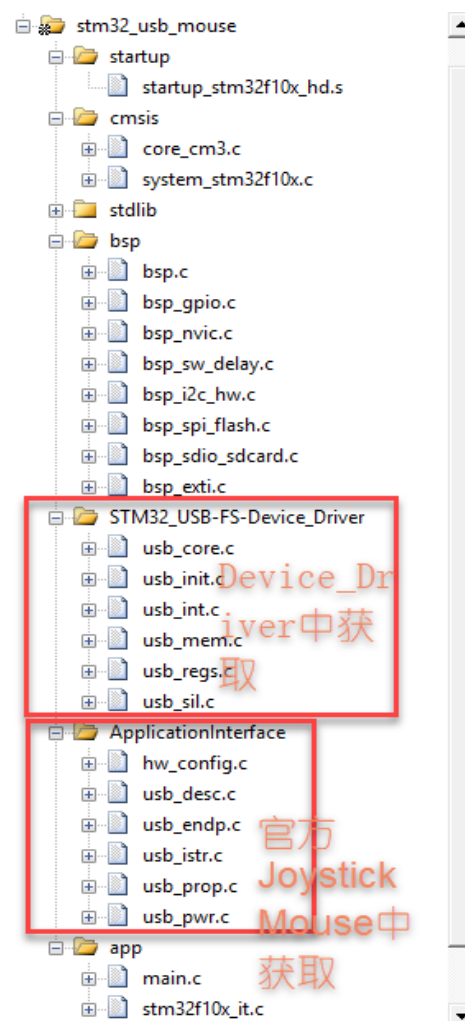
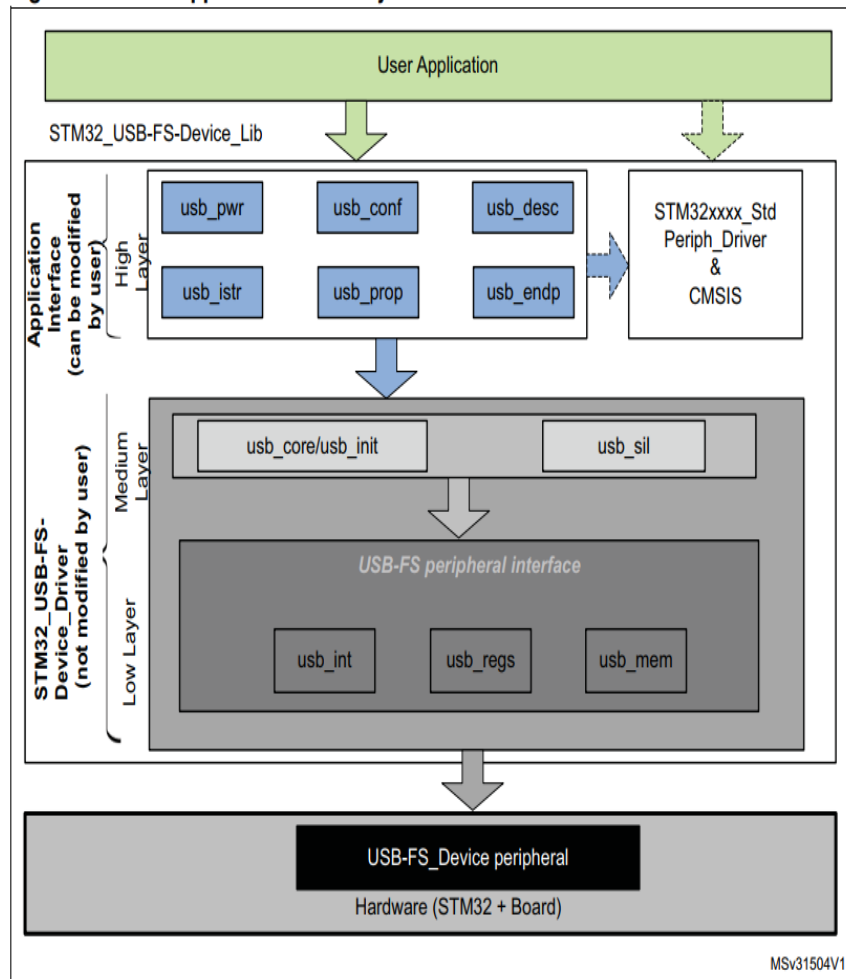


Figure 1. USB application hierarchy



2. 编写main()函数，主要是针对USB的GPIO的配置，中断优先级及时钟配置；还有就是USB设备的初始化. 中断优先级配置中需要配置USB数据相关中断还有USB唤醒中断配置。

```
int main(void)
{
    bsp_Init();
    PrintfLogo();

    /* hw_config.c */
    Set_System();
    USB_Interrupts_Config();
    Set_USBClock();

    /* usb_init.c */
    USB_Init();

    while (1)
    {
        LED1_TOGGLE;
        bsp_sw_delay_ms(200);
        LED2_TOGGLE;
        bsp_sw_delay_ms(200);
        LED3_TOGGLE;
        bsp_sw_delay_ms(200);
        LED4_TOGGLE;
        bsp_sw_delay_ms(200);
    }
}
```

```
NVIC_InitTypeDef NVIC_InitStructure;

/* 2 bit for pre-emption priority, 2 bits for subpriority */
NVIC_PriorityGroupConfig(NVIC_PriorityGroup_2);

/* Enable the USB interrupt */
NVIC_InitStructure.NVIC_IRQChannel = USB_LP_CAN1_RX0_IRQn;
NVIC_InitStructure.NVIC_IRQChannelPreemptionPriority = 5;
NVIC_InitStructure.NVIC_IRQChannelSubPriority = 0;
NVIC_InitStructure.NVIC_IRQChannelCmd = ENABLE;
NVIC_Init(&NVIC_InitStructure);

/* Enable the USB Wake-up interrupt */
NVIC_InitStructure.NVIC_IRQChannel = USBWakeUp_IRQn;
NVIC_InitStructure.NVIC_IRQChannelPreemptionPriority = 0;
NVIC_InitStructure.NVIC_IRQChannelSubPriority = 0;
NVIC_Init(&NVIC_InitStructure);
```



3. 中断服务函数添加. 在stm32f10x\_it.c中添加以下代码.

```
/******  
* Function Name : USB_LP_CAN_RX0_IRQHandler  
* Description : This function handles USB Low Priority or CAN RX0 interrupts  
* requests.  
* Input : None  
* Output : None  
* Return : None  
*****/  
void USB_LP_CAN1_RX0_IRQHandler(void)  
{  
    USB_Istr();  
}
```

4. 对项目进行编译, 对编译出错的部分进行适当删减, 去掉不需要的部分. 保证项目编译成功.  
5. 更新报告描述符, 厂商描述符, 产品描述符, 序列号描述符及语言描述符.  
6. HID报告的发送.

- 组织需要的报告数据
- 将待发送的报告写入PMA(packet buffer memory area)  
uint32\_t USB\_SIL\_Write(uint8\_t bEpAddr, uint8\_t\* pBufferPointer, uint32\_t wBufferSize)
- 使能相应的发送端点将数据发送到主机端  
void SetEPTxValid(uint8\_t bEpNum)

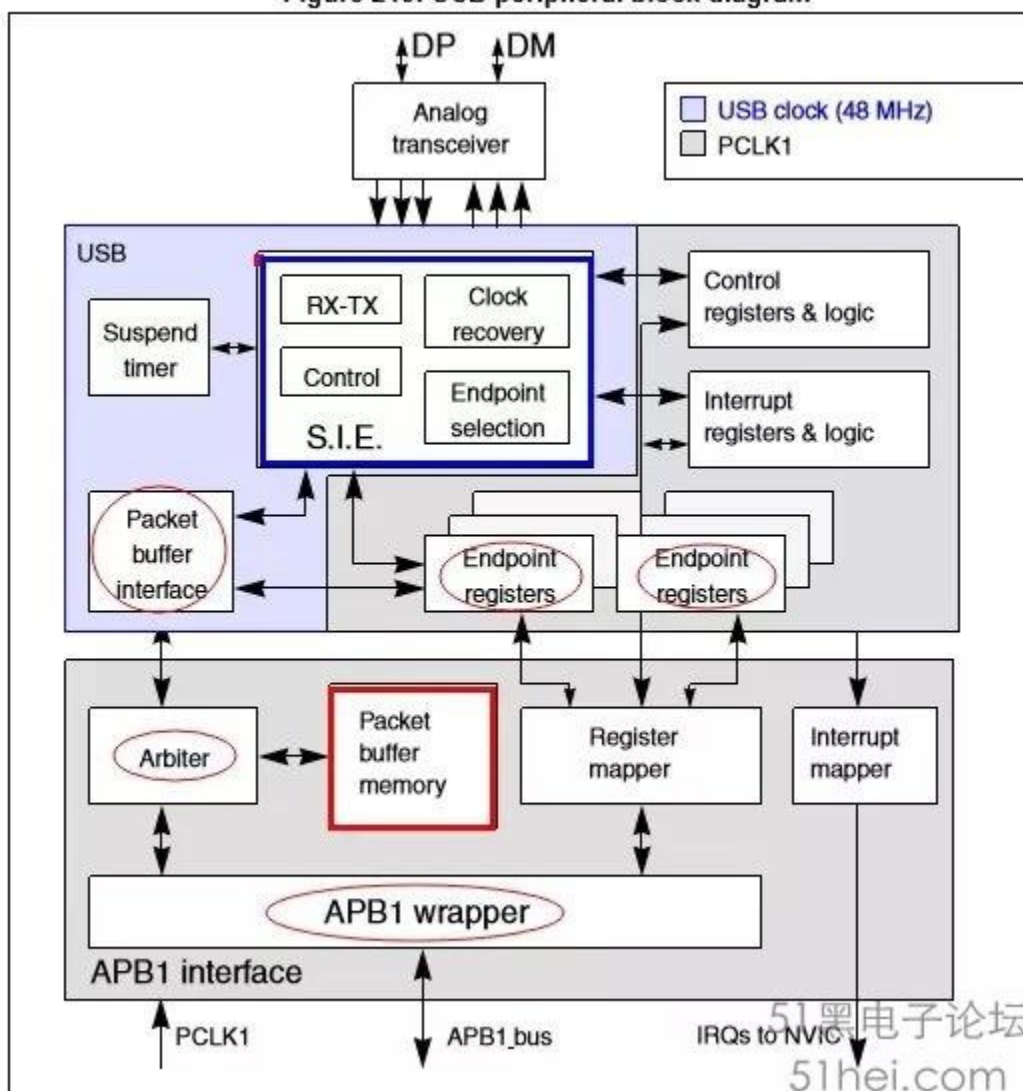
## 2 STM32 USB Device应用中的PMA包缓冲

STM32 系列 MCU 大多具有 USB 外设，其中一部分具有 USB FS 模块，作为 DEVICE 使用。另外一部分具备 OTG 模块，可以实现 HOST/DEVICE 双重角色的功能。这里聊聊关于 STM32F1/F3/L1 系列的 USB FS 模块中的包数据缓冲话题，即 Packet Buffer Memory。STM32F1/F3/L1 这三个系列的 USB FS 模块基本具有相同的结构，这里不妨以 STM32F1 系列的非互联型芯片的 USB FS 模块为例来介绍下 Packet Buffer Memory Area,后面简称 PMA。

这个 PMA 的作用就是 USB 设备模块用来实现 MCU 与主机进行数据通信的一个专门的数据缓冲区，我们称之为 USB 硬件缓冲区。说得具体点就是 USB 模块把来自主机的数据接收进来后先放到 PMA，然后再被拷贝到用户数据缓存区；或者 MCU 要发送到主机的数据，先从用户数据缓存区拷贝进 PMA，再通过 USB 模块负责发送给主机。

不少人在利用 ST 官方提供的参考工程库文件来开发自己的 USB 应用时，围绕 PMA 的配置和使用地方有时会出错卡壳，或者说即使做完了，即使关于 PMA 可能还有些疑惑。这里一起聊聊 PMA 话题。不妨先看看 STM32F1 系列的非互联型芯片的 USB FS 模块功能结构图。

Figure 219. USB peripheral block diagram



上图中的红色方框部分就是 Packet Buffer Memory，该系列芯片的的最大容量为 512 字节，在 USB 模块内部按半字访问，即 256 个半字。该存储区域既可以被 USB 内核通过 Packer buffer Interface【包缓冲接口】寻址访问，亦可以被 CPU 通过 APB1 总线访问，某时刻具体谁去访问它由图中的 Arbiter【仲裁器】决定，APB1 总线具有更高优先级。要注意的是，USB 内核访问该区域是 16 位数据宽

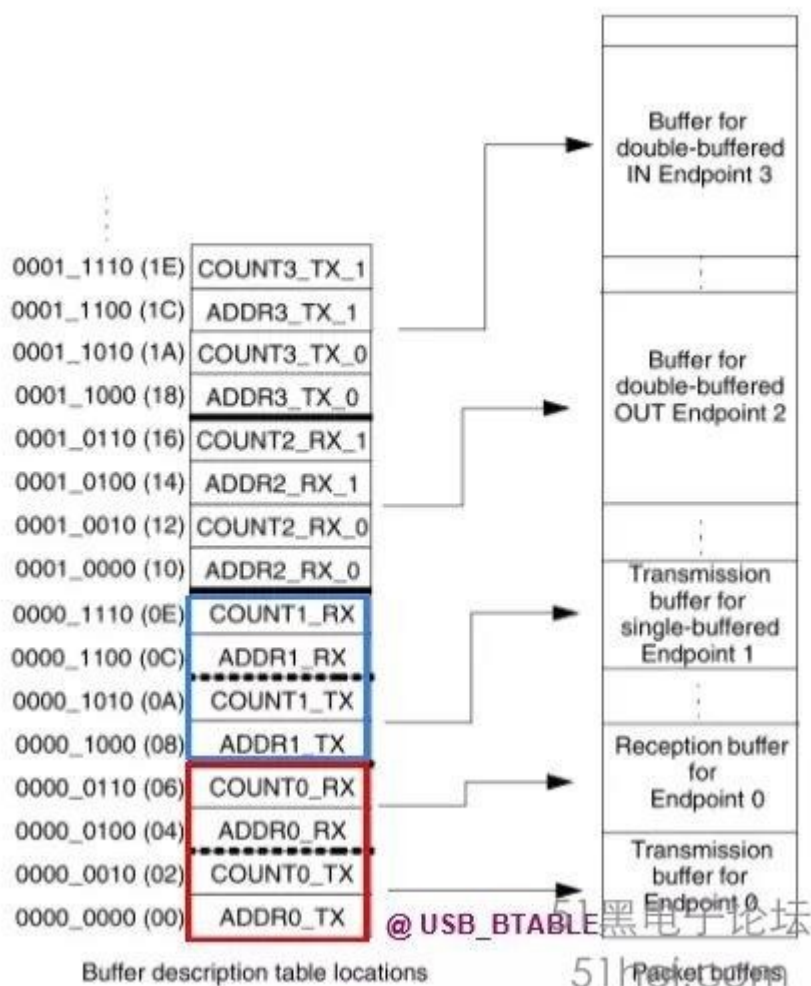


度对齐，而 APB1 总线访问它是以 32 位数据宽度对齐。另外，USB 模块中的端点相关寄存器的访问跟 PMA 一样，也可以分别被 USB 内核或 APB1 总线访问，同样存在两种访问对齐的问题。

好，继续聊聊 PMA 的具体内部结构和用法。我们知道，【如果不知道就假设知道吧，其实无数的理论都是从假设或约定开始的：-）】每个双向端点对应 2 个 Packet Buffer, 分别用作发送数据包的缓冲和接收数据包的缓冲【**双缓冲端点可视为两个单向端点轮流跟 HOST 固定地做 IN 或 OUT 方向的单向通信，轮换使用 2 个同向缓冲区**】。这些 Packet buffer 的大小和位置在 PMA 内可以配置，具体由缓冲描述表【buffer description table】来指定，该描述表也放在 PMA 里面。它的起始地址由 USB\_BTABLER 寄存器指定。即整个 PMA 放置的内容就是缓冲描述表和该表所指定的各端点相应的接收或发送缓冲区。

现在问题来了，既然缓冲描述表负责指定使用到的各端点的包缓冲地址及大小，那缓冲描述表自身在 PMA 中所占存储空间的大小怎么定的呢？

下面就是 PMA 的内容框架结构示意图。PMA 内存放着 USB 应用中用到的各端点包缓冲【PACKET BUFFER】和指定这些 PACKET BUFFER 的地址和大小的缓冲描述表【buffer description table】两部分内容。



从上面 PMA 的内部框架图可以比较直观地看出缓冲描述表是由各端点两个方向的缓冲区起始地址及相关数据长度之内容所占据，每个双向端点占 4 个连续的半字，在 PMA 内部按照地址从低往高的顺序依次存放各端点的发送缓冲区的地址、待发送数据的长度、接收缓冲区的地址、接收缓冲区的长度。这 4 个连续的半字组成缓冲描述表的一个表项[ENTRY]。各表项又按照端点序号依次从小往大的朝地址递增方向连续存放。

前面说过缓冲描述表在 PMA 中的起始位置的偏移量由 USB\_BTABLER 寄存器决定。如无特别需要，一般把它设置为零，即从 PMA 的硬件起始地址开始存放包缓冲描述表，确切地说，从 PMA 的起始地址开始沿着端点序号由小到大的顺序依次存放或预留各端点表项，直到程序代码中指定的最大端点号为止。每个表项占 4 个半字，即 8 个字节。



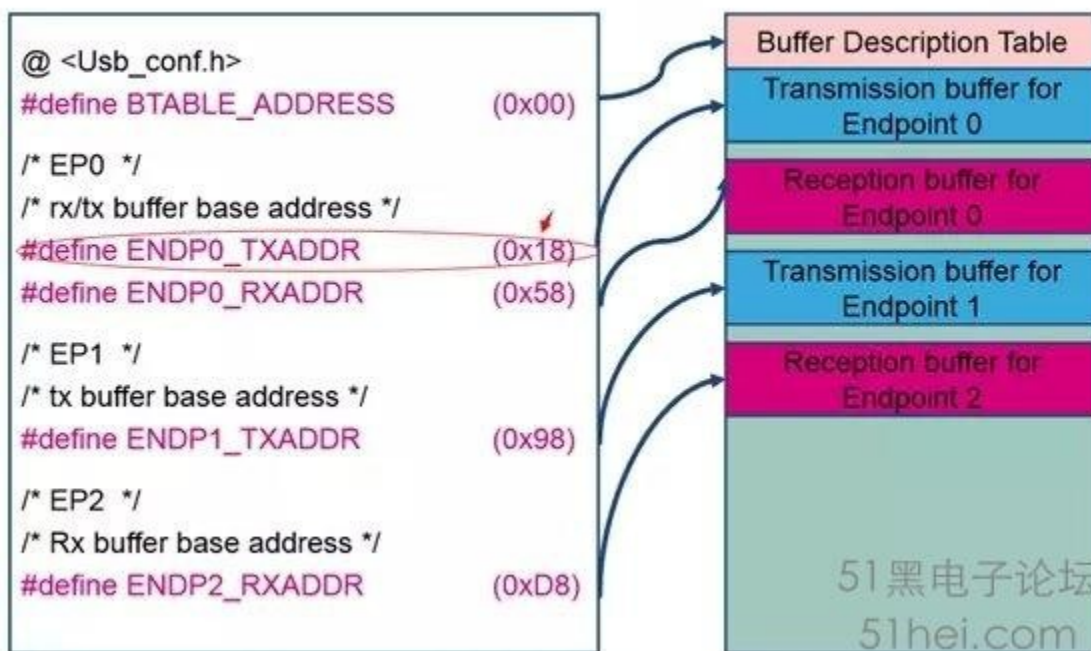
比方说，如果你用到端点 0,1,3 共三个端点，那缓冲描述表里除了安排端点 0,1,3 的表项外，其中 2 号端点的表项空间【8 字节】依然会预留在表项 1 与表项 3 之间的存储空间。也就是说这种情况下，缓冲描述表自身还是占  $4 \times 8 = 32$  字节的空间，其中表项 2 的 8 字节空间可以说是浪费了【其实也可以用，只是使用起来不太方便】。

当然，我这里只是举个例子。一般来讲，比方中的端点安排完全可以调整为 0,1,2，那 3 号端点就用不着了。这样的话缓冲描述表自身所占的 PMA 空间为  $3 \times 8 = 24$  字节，节省出 8 个字节可以给后面的包收发缓冲区之用。

我们已经知道，PMA 里面的各端点的包缓冲地址及大小是由缓冲描述表指定的，而缓冲描述表同时又跟这些包缓冲共同占据整个 PMA。那么，如果缓冲描述表自身在什么位置、占多少空间不清的话，而去给端点指定缓冲地址及大小难免有点抓瞎的味道。有人在做 STM32 USB DEVICE 开发应用时，正是在对缓冲描述表本身位置及自身用到多少 PMA 存储空间不清楚的情况下而去给用到的端点指派缓冲地址及大小，经常出现调试或通信异常。

经过上面的介绍，了解了缓冲描述表的结构和存储模式后，对于不同端点需求的情况下，缓冲描述表自身占多少空间就不难算出了，知道了起始地址，结尾地址自然就知道了。下面一起来看看大家常见的一个 STM32 USB DEVICE 应用实例。

## Packet Buffer 的设置



显然，这里用到了 0,1,2 三个端点。具体是 0 号双向端点，1 号 IN 端点，2 号 OUT 端点。配置代码里有 5 个数据，其中第一行的数据 0x00 告知缓冲描述表的起始位置位于 PMA 的起始位置，另外 4 个数据用来明确给出应用中用到的各端点数据缓冲区起始地址【或发或收】。顺便提下，USB 硬件不会把本端点的数据溢出到相邻端点的缓冲区。从这几句代码可以看出使用到的各端点数据缓冲区的起始地址及大致范围，但似乎并不能明确看出前面提到的缓冲描述表所占的地址空间及尾地址。尤其 ENDP0\_TXADDR 的起始地址 0x18 怎么定出来的呢？是否还可以变动？

结合上面关于缓冲描述表的介绍，一起来分析下。这里用到 3 个端点，而且是三个编号连续的端点，对应 3 个表项，那么该缓冲描述表自身所占空间长度为  $3 \times 8 = 24$ ，16 进制是 0x18；因为 BTABLE\_ADDRESS=0，所以缓冲描述表在 PMA 里存放位置就是 0x00-0x17 这段存储空间，从 0x18 开始以上的空间就可用作包缓冲区了。为了充分利用 PMA 空间，所以 0 号端点的发送缓冲区起始地址 ENDP0\_TXADDR 就从 0x18 开始了。其它缓冲区的地址位置和大小也是在结合各自端点传输特性和相邻包缓冲大小的基础上拟定，避免相邻缓冲区的交叉重叠。

那这个起始地址 0x18 是否可以变动呢？如果在端点安排不变的前提下，这个 0x18 是否可以变动，答案是肯定的。只要不放进缓冲描述表地址段里面就好，完全可以弄成 0x20,0x28,0x48 等不小于 0x17 的偶数，因为 USB 模块对 PMA 存储区的访问是双字节对齐。当然，如果你把 0x18 调整后，其它相邻的包缓冲地址往往需要做相应的调整。



那假如在上面图示的基础上，因为项目功能需求调整，减少 1 个端点，比如把 2 号 OUT 端点取消，那上面的 PMA 地址配置需要怎样调整呢？【其它相关问题这里就不延伸了】

如果取消 OUT 2 端点的话，上面有关 PMA 地址的配置基本可以不动，屏蔽掉关于 OUT2 端点接收缓冲地址定义的进行。当然，这里的端点数由之前的 3 个变少到 2 个，缓冲描述表自身的长度实际上变少为  $2 \times 8 = 16$  字节了，也就是说从 0x10 开始以上的空间都可以用来做包缓冲区，如果包缓冲区继续保持在 0x18 开始的地方也无妨。

假如就在上面图示基础上，因为功能的需求调整，增加 1 个单向端点，比如增加 3 号 IN 端点，那上面的 PMA 地址配置需要怎样调整呢？

这里的端点数由之前的 3 个变成 4 个，缓冲描述表自身的长度就变多变为  $4 \times 8 = 32$  字节了，即 0x20 以下的空间都给缓冲描述表占了，换句话说，只有从 0x20 开始以上的空间才能用作包缓冲区。那端点 0 发送缓冲地址【ENP0\_TXADDR】如果还保持 0x18 的话肯定不行，这里不调整的话可能就乱大了。经常有人在以 ST 官方参考库的基础上增加端点后，只是添加新增端点的缓冲区地址和大小及相应描述配置等，却没有顾及到这个包缓冲的起始地址的问题而遇到调试障碍。具体到这里，ENP0\_TXADDR 缓冲区的地址不得小于 0x20，其它缓冲地址往往需要适当调整移动。当然，如果觉得剩余 PMA 空间够用的话，你也完全可以在既不影响其它端点缓冲大小又不影响自身缓冲需要的前提下，大刀阔斧的把 ENP0\_TXADDR 缓冲起始地址换到一个全新的地址，不一定非得围着 0x18 或 0x20 转来转去，比方换到 0x138。

还是接着上面的实例继续聊，看看根据上面的初始化配置，相关缓冲描述表的表项内容在 PMA 中是如何存放的。

```
SetEPRxAddr(ENDP0, 0x18);
SetEPRxCount(ENDP0, 0x40);
SetEPTxAddr(ENDP0, 0x58);

SetEPTxAddr(ENDP1, 0x98);

SetEPRxAddr(ENDP2, 0xD8);
SetEPRxCount(ENDP2, 0x40);
```

低字节 → 高字节

地址	TX-ADDR	TX-CNT	RX-ADDR	RX-CNT	
0x40006000	00000058	0000XXXX	00000018	00008400	EP0
0x40006010	00000098	0000YYYY	0000....	0000....	EP1
0x40006020	0000....	0000....	000000D8	00008400	EP2

显然，整个缓冲描述表用到 3 个端点，共占空间  $3 \times 8 = 24$  个字节。

因为 OUT 1 端点和 IN 2 端点没有用到，但他们在表项中的椭圆形圆圈圈出来的灰色部分空间还预留出来了，一共 8 个字节夹在中间基本算是浪费了。如果把端点 OUT 2 换到 OUT 1，那整个缓冲描述表只用到 2 个端点，所需空间就变为 16 字节了。多余出来的 8 字节就可以方便的留给包缓冲用。另外，表格中的有些数据可能需要结合参考手册的 USB 寄存器部分看看。那个 XXXX/YYYY 表示初始化时不定的，准备做数据传输时才确定。地址 0x40006000 是 PMA 的起始地址。

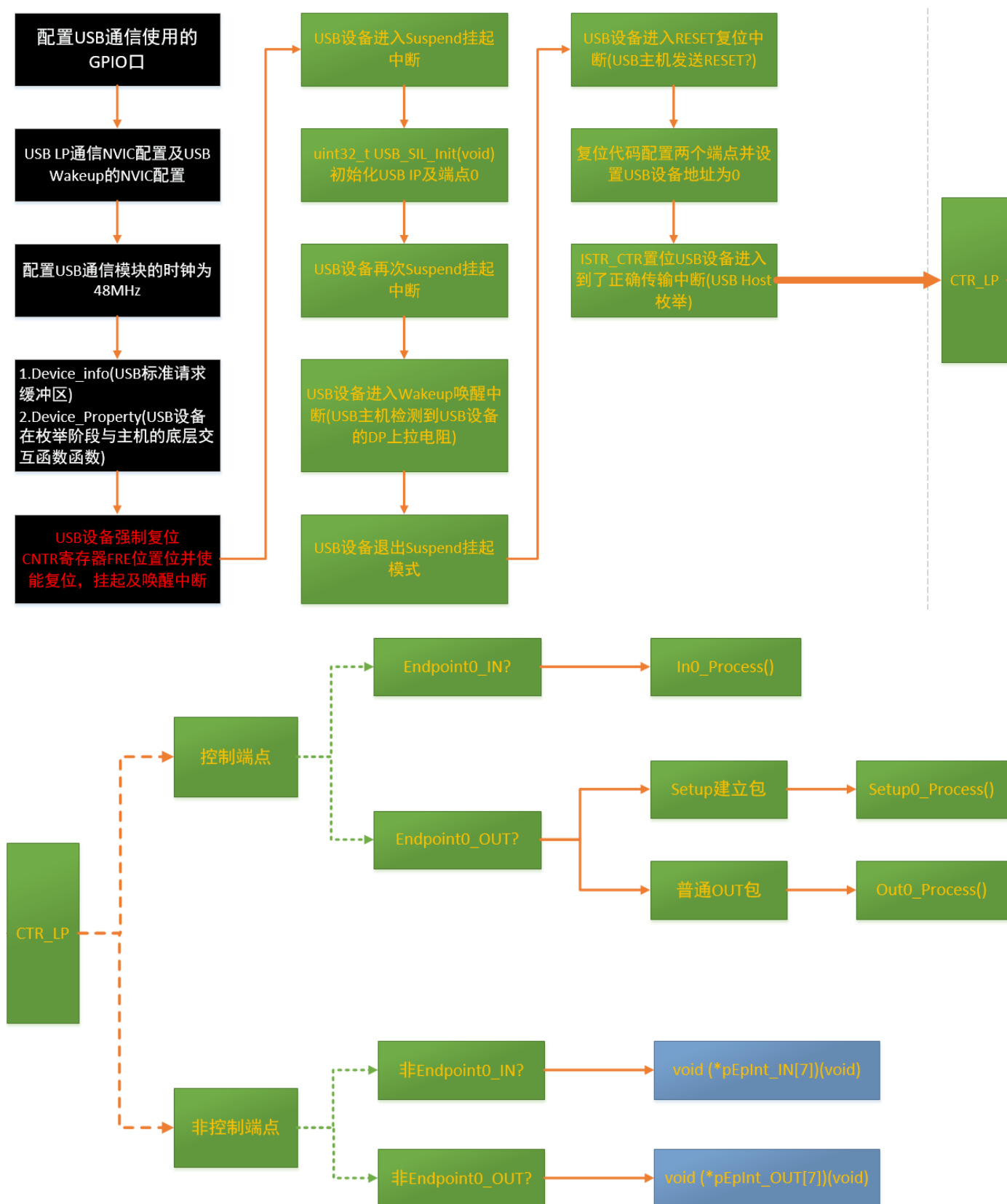
小结下，缓冲描述表的起始地址和所占空间都是可以明确指定和计算出来的，包缓冲地址的安排往往跟它有关联。只有明确知道了缓冲描述表的起始地址和自身所需存储空间长度后，才能放心地用剩余的 PMA 给用到的端点分别指定缓冲地址及大小。

缓冲描述表里的表项是根据端点号在 PMA 里按序填充的，合理安排端点可以减少缓冲描述表的 PMA 空间使用量，而留下更多空间给端点包缓冲区。

最后，注意每次传输实际收到的数据包长度不会超过各端点接收缓冲区容量的限制，长了会被截掉。



### 3 STM32 USB Device固件库运行机制





#### 4 STM32 USB鼠标移植结果

##### 鼠标、键盘和笔



Majestouch Convertible 2  
蓝牙已关闭



OnePlus



Rapoo Gaming Device



Rapoo Gaming Mouse

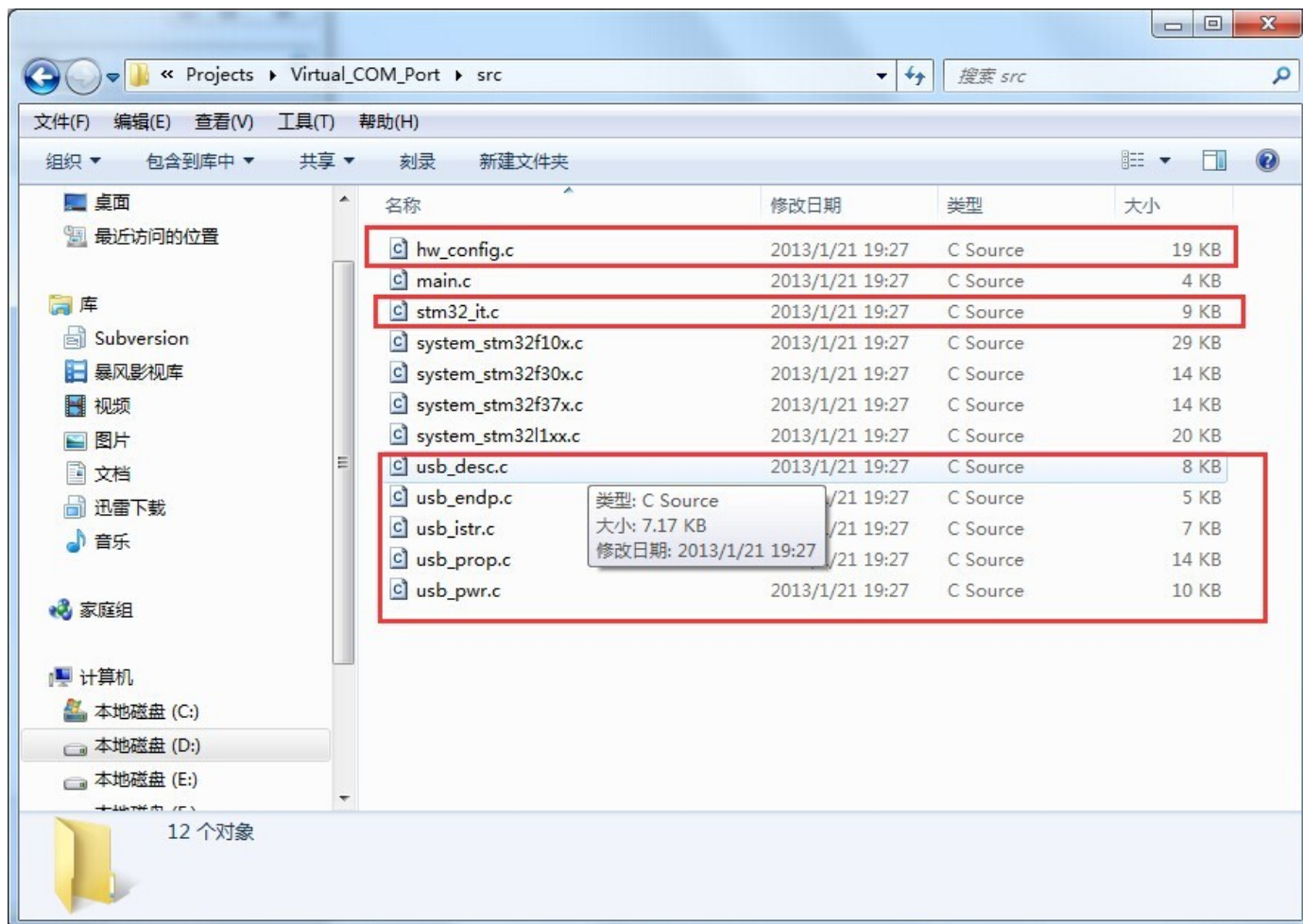


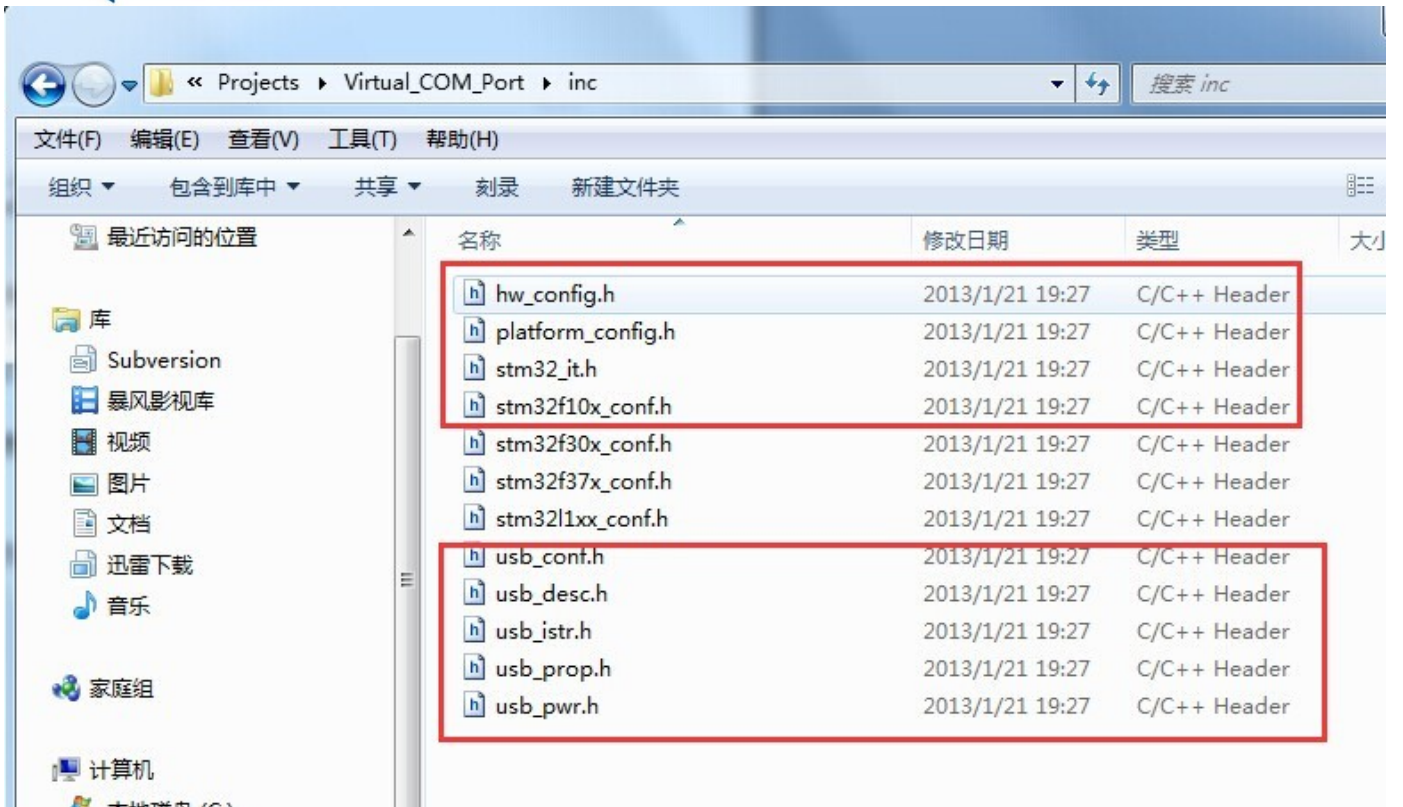
勤进电子科技HID鼠标

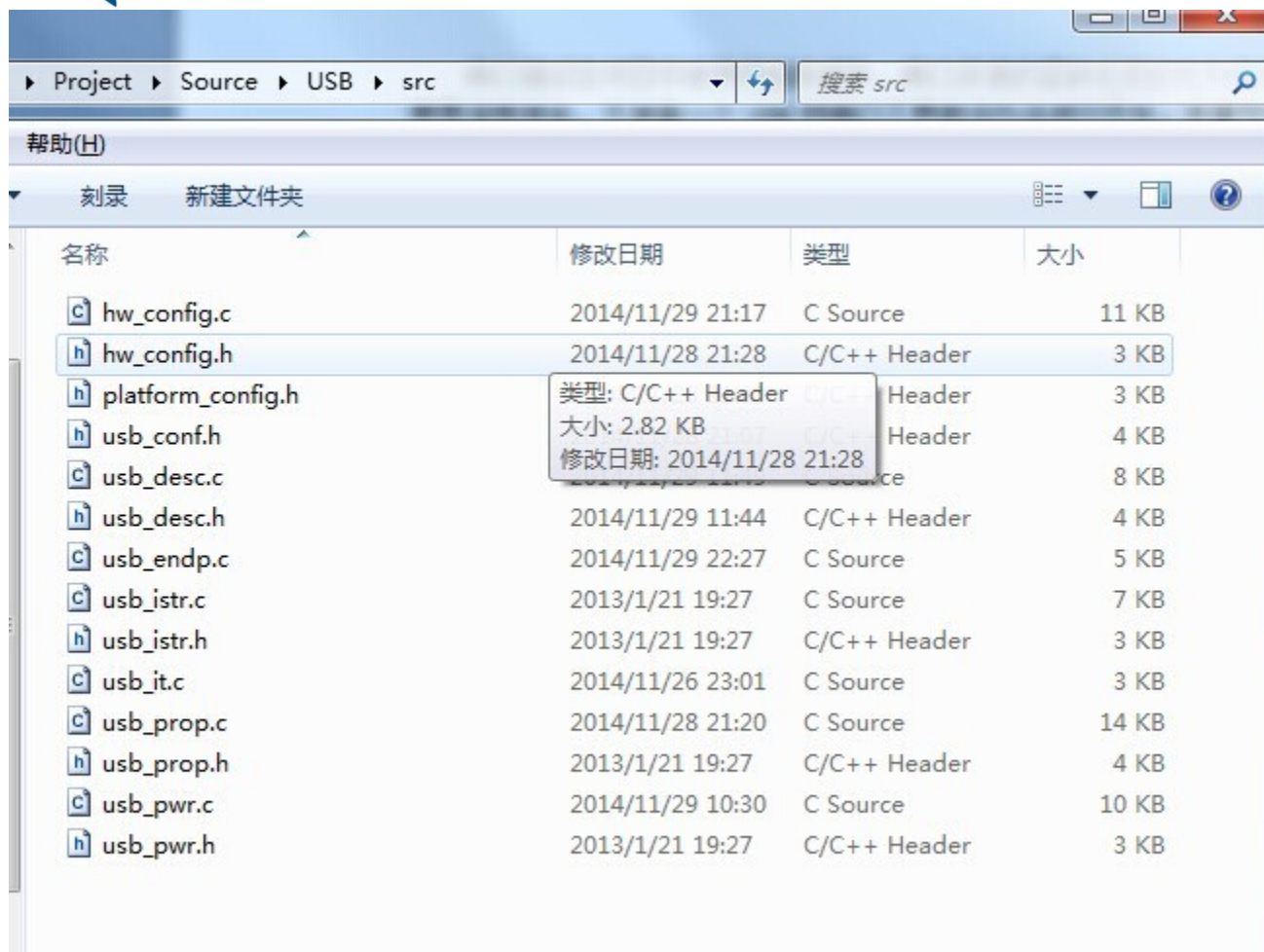
## 5 STM32 USB CDC移植

串口调试在项目中被使用越来越多，串口资源的紧缺也变的尤为突出。很多本本人群，更是深有体会，不准备一个USB转串口工具就没办法进行开发。本章来简单概述STM32低端芯片上的USB虚拟串口的移植。在官方DEMO中已经提供了现成的程序，这里对修改方法做简单说明。

首先打开官方 demo 我们开始进行移植，第一步复制我们可用的文件，操作如下：Projects\Virtual\_COM\_Port 文件夹下，复制红线部分

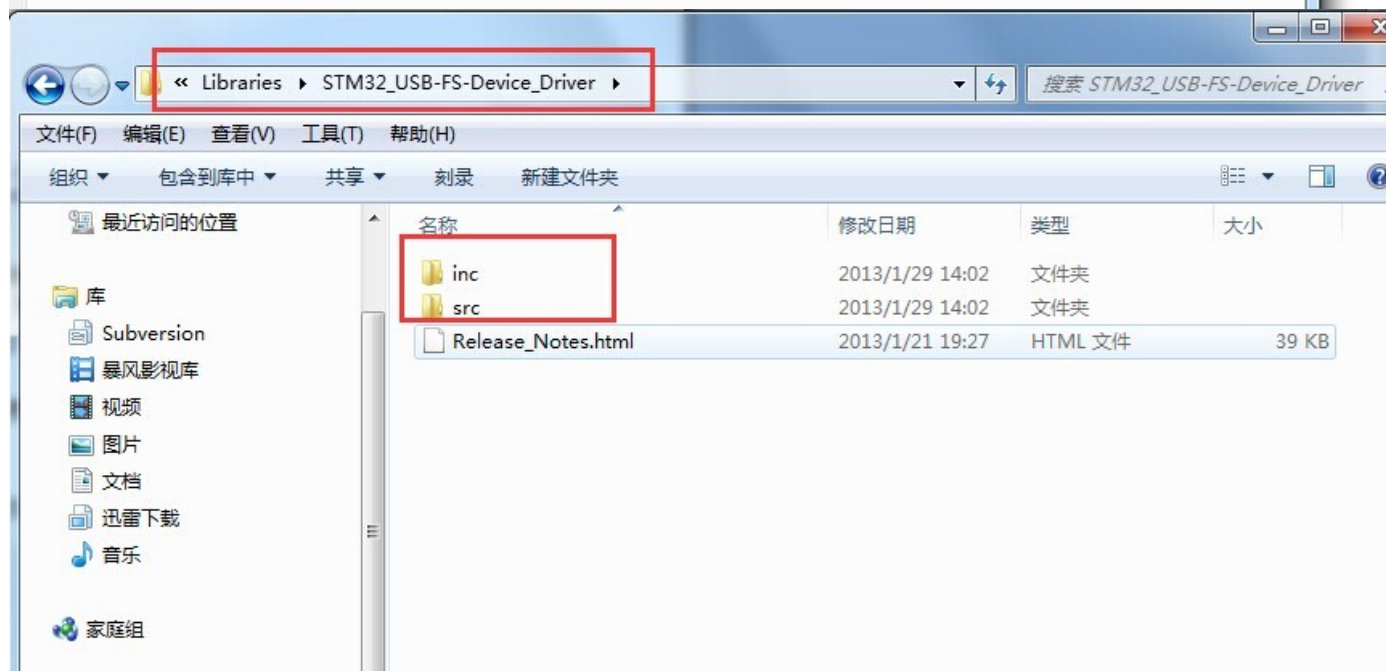




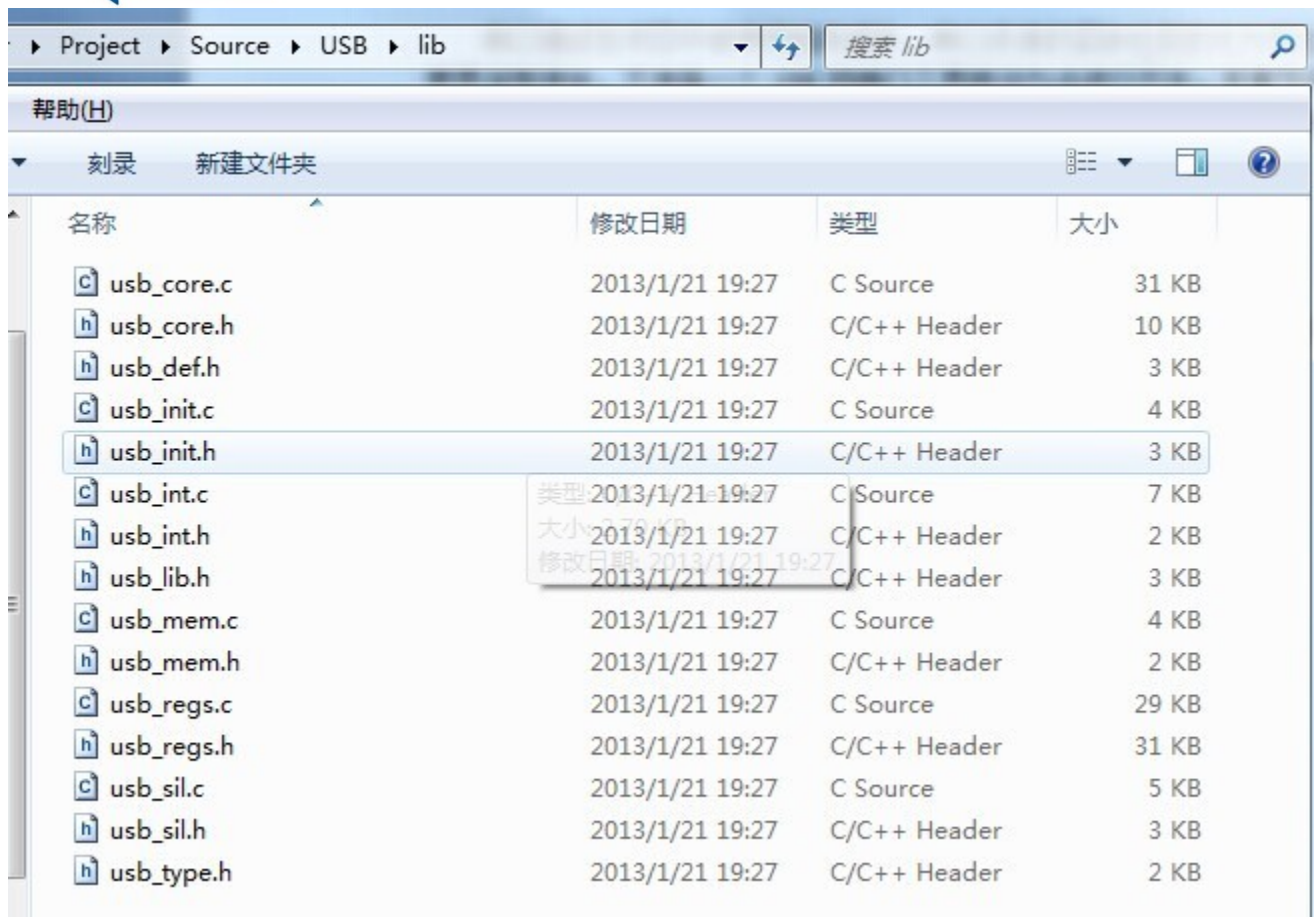


很多  
简单  
程序，

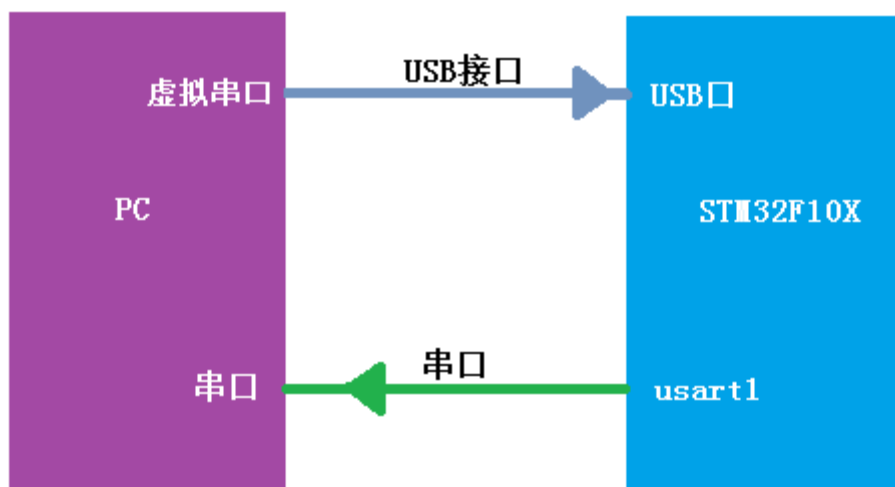
操作







好了现在所需要的文件我们以复制完了。这里先讲一下DEMO程序的主要工作流程：



由上图可知，PC通过虚拟串口发送数据到STM32 usb口，STM32再通过uart1发送数据到PC串口。我们做项目时，只用USB虚拟串口即可。所以我们现在需要把串口发送部分删除。把USB做为一个COM口来使用。我们要如何使用这个USB口呢？demo中是把USB发送数据做了一个缓存，先把要发送的数据存入缓存中，然后由USB自动发送出去。而接收部分是直接通过串口透传。我们在应用时就需要用到两个FIFO，1是发送，这个和demo方式是样；2是接收，接收也做一个缓存，我们通过查询来判断是否收到新数据。这下大家应该明白为什么使用两个FIFO了。我这里有写好的FIFO库函数可直接使用Queue.c文件。

现在开始修改：

1, stm32\_it.c 更名为 usb\_it.c 删除无用代码，只保留 usb 中断函数，和唤醒函数。代码如下：



```
1 /* Includes -----*/
2 #include "hw_config.h"
3 #include "usb_lib.h"
4 #include "usb_istr.h"
5
6
7 /*****
8 * Function Name : USB_IRQHandler
9 * Description : This function handles USB Low Priority interrupts
10 * requests.
11 * Input : None
12 * Output : None
13 * Return : None
14 *****/
15 #if defined (STM32L1XX_MD) || defined (STM32L1XX_HD) || defined (STM32L1XX_MD_PLUS) || defined (STM32F37X)
16 void USB_LP_IRQHandler(void)
17 #else
18 void USB_LP_CAN1_RX0_IRQHandler(void)
19 #endif
20 {
21 USB_Istr();
22 }
23
24 /*****
25 * Function Name : USB_FS_WKUP_IRQHandler
26 * Description : This function handles USB WakeUp interrupt request.
27 * Input : None
28 * Output : None
29 * Return : None
30 *****/
31
32 #if defined (STM32L1XX_MD) || defined (STM32L1XX_HD) || defined (STM32L1XX_MD_PLUS)
33 void USB_FS_WKUP_IRQHandler(void)
34 #else
35 void USBWakeUp_IRQHandler(void)
36 #endif
37 {
38 EXTI_ClearITPendingBit(EXTI_Line18);
39 }
```

2, 修改代码 hw\_config.c 删除无用代码, 新建立 2 组, 读 FIFO 和写 FIFO 的函数。后面会用到。

这里要讲一下为什么要屏蔽 SystemInit(), 因为 demo 只运行虚拟串口功能, 在 USB 未插入的情况下, 是进入低功耗状态, 插入时从低功耗状态退出后会调用此函数。当然我们在项目中一般不会这样, 系统是否运行和插 USB 接口没有联系。所以我在下文中把进入低功耗代码屏蔽了, 自然也就不需要唤醒代码了。

代码如下:



```
00121:
00122: /*****
00123: * Function Name : Leave_LowPowerMode
00124: * Description : Restores system clocks and power while exiting suspend mode
00125: * Input : None.
00126: * Return : None.
00127: *****/
00128: void Leave_LowPowerMode(void)
00129: {
00130:     DEVICE_INFO *pInfo = &Device_Info;
00131:
00132:     /* Set the device state to the correct state */
00133:     if (pInfo->Current_Configuration != 0)
00134:     {
00135:         /* Device configured */
00136:         bDeviceState = CONFIGURED;
00137:     }
00138:     else
00139:     {
00140:         bDeviceState = ATTACHED;
00141:     }
00142:     /*Enable SystemCoreClock*/
00143:     // SystemInit();
00144: }
00145:
```

关于USB口使能控制引脚，需要根据开发板的引脚定义来修改宏定义platform\_config.h文件中，根据需要修改hw\_config.c代码如下：

```
1 /*****
2 * Function Name : USB_Cable_Config
3 * Description : Software Connection/Disconnection of USB Cable
4 * Input : None.
5 * Return : Status
6 *****/
7 void USB_Cable_Config (FunctionalState NewState)
8 {
9     if (NewState == DISABLE)
10     {
11         GPIO_ResetBits(USB_DISCONNECT, USB_DISCONNECT_PIN);
12     }
13     else
14     {
15         GPIO_SetBits(USB_DISCONNECT, USB_DISCONNECT_PIN);
16     }
17 }
```

3, 现在修改USB 回调函数中的代码usb\_endp.c文件。使用下文代码替换：

```
1 /* Includes -----*/
2 #include "usb_lib.h"
3 #include "usb_desc.h"
4 #include "usb_mem.h"
5 #include "hw_config.h"
6 #include "usb_istr.h"
7 #include "usb_pwr.h"
8
9 /* Private typedef -----*/
10 /* Private define -----*/
11
12 /* Interval between sending IN packets in frame number (1 frame = 1ms) */
```



```
13 #define VCOMPORT_IN_FRAME_INTERVAL 5
14
15 /* Private macro -----*/
16 /* Private variables -----*/
17 static uint8_t txBuffter[VIRTUAL_COM_PORT_DATA_SIZE] = {0};
18 static volatile uint8_t txFlg = 0;
19 static volatile uint32_t FrameCount = 0;
20
21
22 /* Private function prototypes -----*/
23 /* Private functions -----*/
24
25 /*****
26 * Function Name : EP1_IN_Callback
27 * Description :
28 * Input : None.
29 * Output : None.
30 * Return : None.
31 *****/
32 void EP1_IN_Callback (void)
33 {
34     uint16_t len = 0;
35
36     if (1 == txFlg)
37     {
38         len = USB_TxRead(txBuffter, sizeof(txBuffter));
39
40         if (len > 0)
41         {
42             UserToPMABufferCopy(txBuffter, ENDP1_TXADDR, len);
43             SetEPTxCount(ENDP1, len);
44             SetEPTxValid(ENDP1);
45             FrameCount = 0;
46         }
47         else
48         {
49             txFlg = 0;
50         }
51     }
52 }
53
54 /*****
55 * Function Name : EP3_OUT_Callback
56 * Description :
57 * Input : None.
58 * Output : None.
59 * Return : None.
60 *****/
61 void EP3_OUT_Callback(void)
62 {
63     static uint8_t buffter[VIRTUAL_COM_PORT_DATA_SIZE] = {0};
64
65     uint16_t USB_Rx_Cnt;
66
67     /* Get the received data buffer and update the counter */
```



```
68 USB_Rx_Cnt = USB_SIL_Read(EP3_OUT, buffter);
69
70 /* USB data will be immediately processed, this allow next USB traffic being
71 NAKed till the end of the USART Xfer */
72 USB_RxWrite(buffter, USB_Rx_Cnt);
73
74 /* Enable the receive of data on EP3 */
75 SetEPRxValid(ENDP3);
76
77 }
78
79
80 /*****
81 * Function Name : SOF_Callback / INTR_SOFINTR_Callback
82 * Description   :
83 * Input        : None.
84 * Output       : None.
85 * Return       : None.
86 *****/
87 void SOF_Callback(void)
88 {
89     uint16_t len = 0;
90
91     if(bDeviceState == CONFIGURED)
92     {
93         if (0 == txFlag)
94         {
95             if (FrameCount++ == VCOMPORT_IN_FRAME_INTERVAL)
96             {
97                 /* Reset the frame counter */
98                 FrameCount = 0;
99
100                 /* Check the data to be sent through IN pipe */
101                 len = USB_TxRead(txBuffter, sizeof(txBuffter));
102
103                 if (len > 0)
104                 {
105                     UserToPMABufferCopy(txBuffter, ENDP1_TXADDR, len);
106                     SetEPTxCount(ENDP1, len);
107                     SetEPTxValid(ENDP1);
108
109                     txFlag = 1;
110                 }
111             }
112         }
113     }
114 }
115 /***** (C) COPYRIGHT STMicroelectronics *****END OF FILE*****/
```

这里讲下大概意思，函数 EP3\_OUT\_Callback 是在 USB 口收到数据后，将数据存入 FIFO 中。

函数 SOF\_Callback 定时查询用户是否有要发送的数据，如果有则进行发送，在发送完成后会触发发送中断 EP1\_IN\_Callback 函数，如果发送完毕就不调用 SetEPTxValid(ENDP1)函数，发送完成后就不会再触发 EP1\_IN\_Callback 函数。

4，修改 usb\_pwr.c 在前文中说到：不让系统进入休眠状态，这里屏蔽 185 行 \_\_WFI();

5，修改 usb\_prop.c 屏蔽 COM 初始化代码。137 行 USART\_Config\_Default(); 237 行 USART\_Config();





6, 修改 `usb_desc.c` 这里修改需要参考一些 USB 专业的书籍, 推荐圈圈的书, 讲的通俗易懂。关于本程序的驱动, 笔者在 win7 下测试可以自动安装, 如果无法自动安装可使用文章开始的链接中的驱动程序。本文件如果修改需谨慎, 其中 `pid,vid` 是制造商 ID 和产品编号, 如果修改了那驱动也要对应修改, 官方驱动就无法自动进行安装了。

到这里移植就差不多完成了, 下面进行测试。由于 USB 虚拟串口不受波特率限制, 所以笔者进行过 50k/s 的压力测试, 运行半小时未丢 1 个字节。