**9月17日**

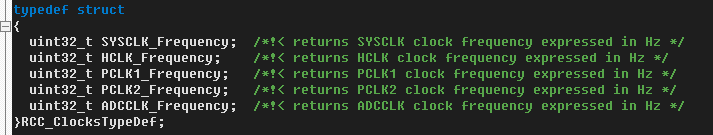
1.查看系统时钟频率

//RCC\_ClocksTypeDef Rcc\_Clk;

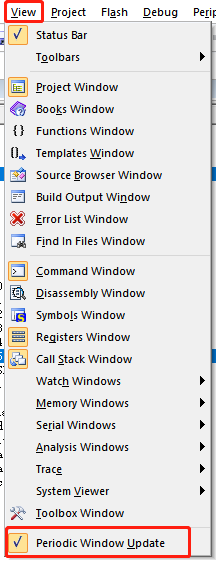
//RCC\_GetClocksFreq(&Rcc\_Clk); //获取时钟频率

然后查看Rcc\_Clk来了解各种时钟频率

结构体 RCC\_ClocksTypeDef



1. 仿真实时更新



1. 硬件仿真时，串口无法使用，发送标志位无效。软件仿真时才可以使用

。

1. 软件仿真使用虚拟串口

下载虚拟串口工具VSPD

创建连接的虚拟串口

在DEBUG界面：

MODE COM2 115200,0,8,1 //配置串口

ASSIGN COM2 <S1IN> S1OUT

串口1就<S1IN> S1OUT 串口2就<S2IN> S2OUT ...

注意：硬件仿真是不能使用的

5.

TTL电平信号规定，+5V等价于逻辑“1”，0V等价于逻辑“0”(采用二进制来表示数据时)。这样的数据通信及电平规定方式，被称做TTL（晶体管-晶体管逻辑电平）信号系统。这是计算机处理器控制的设备内部各部分之间通信的标准技术。

6.

上电后PA13、PA14、PA15、PB3、PB4默认为JTAG的调试接口，不能作为IO使用；若想做为IO使用需要禁止所有调试功能释放IO；

7.

char： 1个字节

short： 2个字节

int： 4个字节

long： 4个字节

原码、反码、补码的关系。

原码：符号位加上真值的绝对值, 即用第一位表示符号, 其余位表示值。

反码：

正数的反码是其本身

负数的反码是在其原码的基础上, 符号位不变，其余各个位取反.

补码：

正数的补码就是其本身

负数的补码是在其原码的基础上, 符号位不变, 其余各位取反, 最后+1. (即在反码的基础上+1)

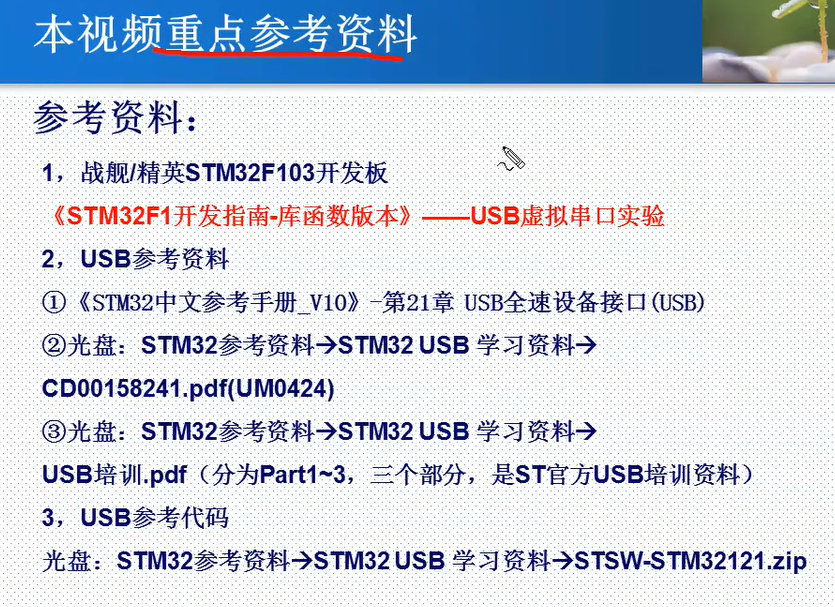
-10:

　正数的原码、反码和补码都相同。

　　负数原码和反码的相互转换：符号位不变，数值位按位取反。

负数原码和补码的相互转换：符号位不变，数值位按位取反,末位再加1。

**9月21日**



USB主机和USB从机的区别：

USB设备分bai为HOST（主设备）和SLAVE（从设备），只有当一du台HOST与一台SLAVE连接时才能实现数据zhi的传输。dao 简单的说，如果一个数码设备支持USB HOST，那么它就可以从另外一个USB设备中取得数据。

USB device：USB设备，从硬件角度看就是一个带有usb client控制器的设备；从软件角度看，就是一个挂在usb总线上的一个普通意义上的设备，只不过它们的驱动是基于host驱动之上的。

两者就是USB主机和USB设备的区别。

设备插在主机上才能实现连接和数据交换。

STM32 编译后不能运行的几个原因

<http://blog.sina.com.cn/s/blog_af34fd470102vajq.html>

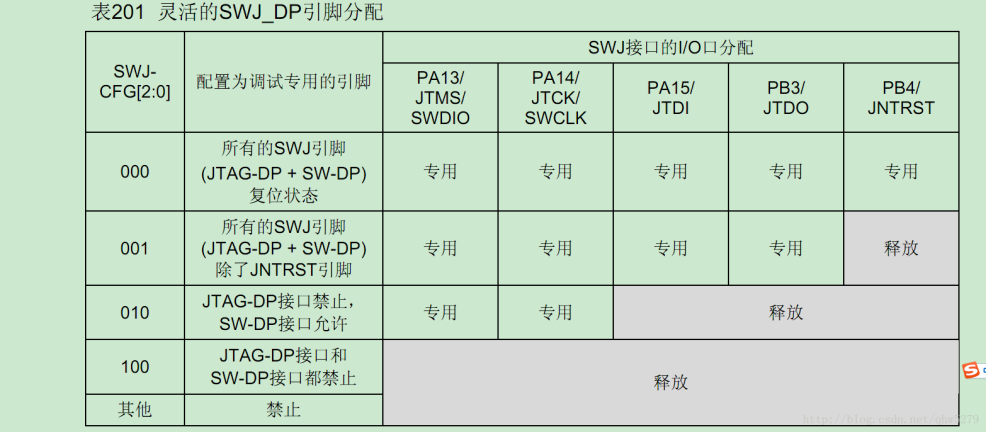
STM32的调试方式、更新程序、仿真以及补救措施

https://blog.csdn.net/weibo1230123/article/details/80056275?ops\_request\_misc=%257B%2522request%255Fid%2522%253A%2522160068017819725222410574%2522%252C%2522scm%2522%253A%252220140713.130102334..%2522%257D&request\_id=160068017819725222410574&biz\_id=0&utm\_medium=distribute.pc\_search\_result.none-task-blog-2~all~sobaiduend~default-2-80056275.pc\_ecpm\_v3\_pc\_rank\_v3&utm\_term=STM32%E5%9C%A8%E7%BA%BF%E4%BB%BF%E7%9C%9F&spm=1018.2118.3001.4187

Xmind快速度入门

1.tab （产生子主题）  
2.enter （**在下方产生并列主题**） shift+enter （**在上方产生并列主题**）  
3.Alt+Enter （给某个主题添加标注）  
4.按住Ctrl，选中连续的几个模块，再按下Ctrl+B，把他们用方框框起来  
5.按住Ctrl，选中几个模块，再按下Ctrl+]，即用大括号括起来，给它们添加概要  
6.选中一个模块，按下Ctrl+L，添加一个带箭头的连接线  
7.添加图片，直接拖进来  
8.选中模块，按下Ctrl+Enter，会在他前面产生一个父主题





usb通信的最基本形式是通过USB设备里的endpoint，而主机和endpoit之间的数据传输就是通过pipe。

Endpoint 端点是有方向的，主机到从机成为out端点，从机到主机成为in端点。从这个说明看出端点是单方向的。（除了0端点）

主机以控制传输(Control Transfer)的方式，通过端点0(Endpoint 0)对设备发送各种请求，设备收到主机发来的请求后回复相应的信息，进行枚举（Enumerate）操作。所有的USB设备必须支持标准请求（StandardRequests），控制传输方式（Control Transfer）和端点0（Endpoint 0）。

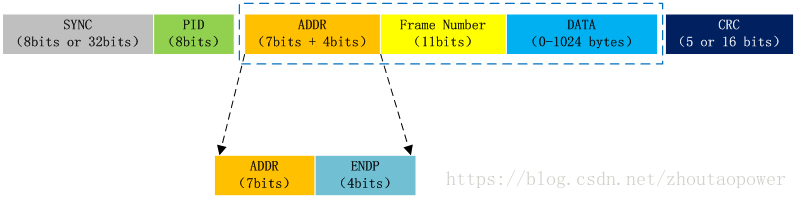
SOF： Start of Frame

EOF： End of Frame.

Frame 帧

Data frame：数据帧

USB包由五部分组成，即同步字段（SYNC）、包标识符字段（PID）、数据字段、循环冗余校验字段（CRC）和包结尾字段（EOP）



DATA0和DATA1包的区别

协议规定,数据包每两个包前面要带不同的PID,据说这样可以起到帧同步的作用.至于数据内容,则与包的类型是DATA0和DATA1无关.比如说我要发送一段数据,而这些数据在分成2个包,那么前一半的数据,由于加上了DATA0的PID,就叫DATA0包.后面的加上了DATA1的PID,就叫DATA1包.多个IN或OUT事务接连发生时，通常会DATA0和DATA1包交换使用，它们只是PID字段不同，没有区它区别。

**批量传输**的翻转同步：发送端按照 DATA0-DATA1-DATA0-…的顺序发送数据包，只有成功的事务传输才会导致 PID 翻转，也就是说发送端只有在接收到 ACK 后才会翻转 PID，发送下一个数据包，否则会重试本次事务传输。同样，若在接收端发现接收到到的数据包不是按照此顺序翻转的，比如连续收到两个 DATA0，那么接收端认为第二个 DATA0 是前一个 DATA0 的重传。（重传机制）

**中断传输**也采用PID翻转的机制来保证收发端数据同步。

**批量传输**由

1) token

2) data

3) handshake

**中断传输**由IN和OUT事务组成

**同步传输**由两种包组成

1) token

1. Data

同步传输是不可靠的传输，所以它没有握手包，也不支持PID翻转。

每个USB设备只有一个设备描述符，而一个设备中可包含一个或多个配置描述符，即USB设备可以有多种配置。设备的每一个配置中又可以包含一个或多个接口描述符，即USB设备可以支持多种功能（接口），接口的特性通过描述符提供。

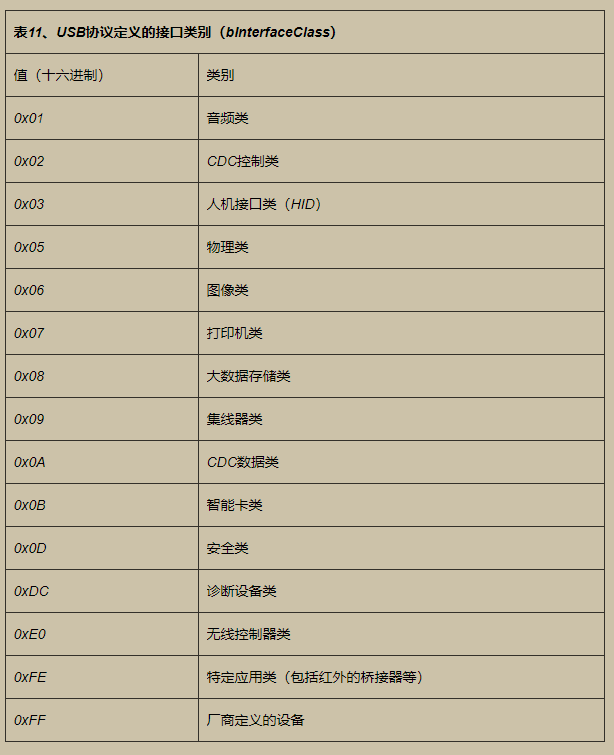
一设备至少要包含设备描述符、配置描述符和接口描述符，如果USB设备没有端点描述符，则它仅仅用默认管道与主机进行数据传输。

**设备描述符：**是设备连接到总线上时USB主机所读取的第一个描述符，它包含了14个字段。

**配置描述符：**中包括了描述符的长度（属于此描述符的所有接口描述符和端点描述符的长度的和）、供电方式（自供电/总线供电）、最大耗电量等。

**接口描述符：**配置描述符中包含了一个或多个接口描述符，这里的“接口”并不是指物理存在的接口，在这里把它称之为“功能”更易理解些，例如一个设备既有录音的功能又有扬声器的功能，则这个设备至少就有两个“接口”。如果一个配置描述符不止支持一个接口描述符，并且每个接口描述符都有一个或多个端点描述符，那么在响应USB主机的配置描述符命令时，USB设备的端点描述符总是紧跟着相关的接口描述符后面，作为配置描述符的一部分发送给主机。如果一个接口仅使用端点0，则接口描述符以后就不再返回端点描述符，并且此接口表现的是一个控制接口的特性，它使用与端点0相关联的默认管道进行数据传输。接口描述符在说明端点个数并不把端点0计算在内。

在接口描述符的 bInterfaceClass 字段，表示接口所属的类别，USB协议根据功能将不同的接口划分成不的类。



USB枚举过程：

0 插入设备

1 获取USB设备的设备描述符 : 使用默认地址0，使用稍微长一些的数据长度（比如：0x40）

2 成功后复位设备

3 对设备进行地址设置 : 使设备从地址状态进入寻址状态

4 再次获取设备描述符 : 使用新的地址

5 获取设备配置符

6 根据设备配置符中的配置符总长度，获取其他配置符（接口，端点等）

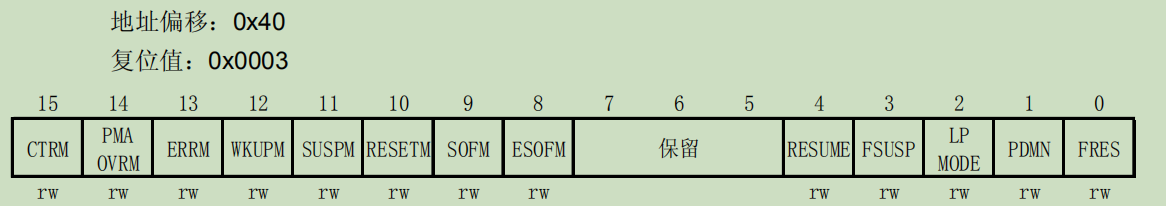
7 设置配置 : 使设备从地址状态进入配置状态

**PC主机和微控制器之间的数据传输**是通过共享一专用的数据缓冲区来完成的，该数据缓冲区能被USB外设直接访问。

USB寄存器

1.通用寄存器

**USB控制寄存器USB\_CNTR**



CTRM：正确传输(CTR)中断屏蔽位 (Correct transfer interrupt mask)

PMAOVRM：分组缓冲区溢出中断屏蔽位 (Packet memory area over / underrun interrupt mask)

ERRM：出错中断屏蔽位 (Error interrupt mask)

WKUPM：唤醒中断屏蔽位 (Wakeup interrupt mask)

SUSPM：挂起中断屏蔽位 (Suspend mode interrupt mask)

RESETM：USB复位中断屏蔽位 (USB reset interrupt mask)

SOFM：帧首中断屏蔽位 (Start of frame interrupt mask)

ESOFM：期望帧首中断屏蔽位 (Expected start of frame interrupt mask)

RESUME：唤醒请求 (Resume request)

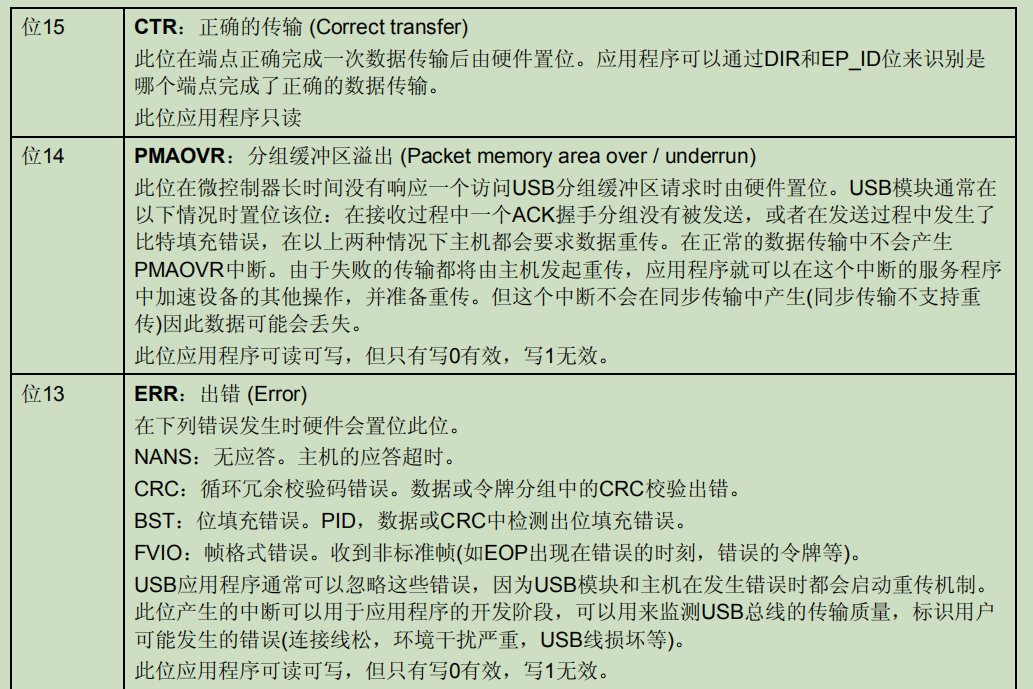
FSUSP：强制挂起 (Force suspend)

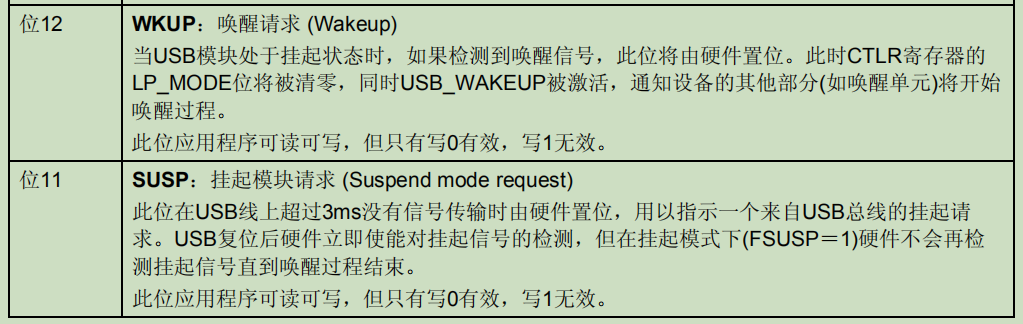
LP\_MODE：低功耗模式 (Low-power mode)

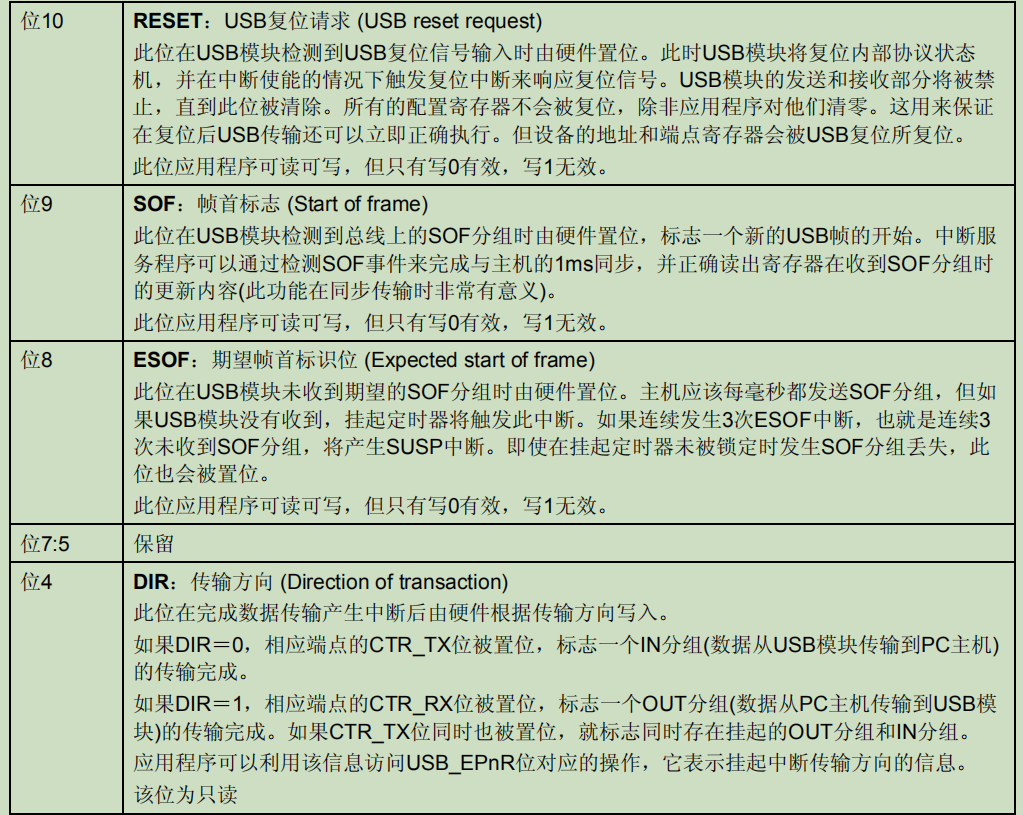
PDWN：断电模式 (Power down)

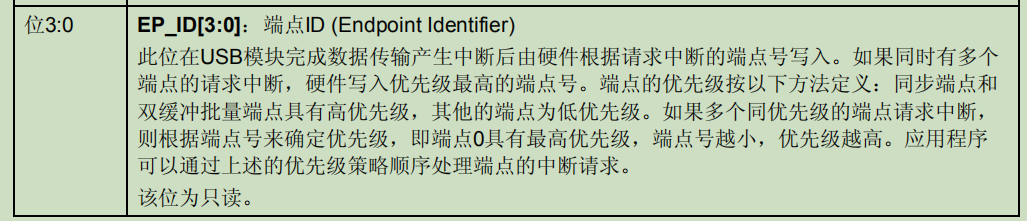
FRES：强制USB复位 (Force USB Reset)

USB中断状态寄存器USB\_ISTR

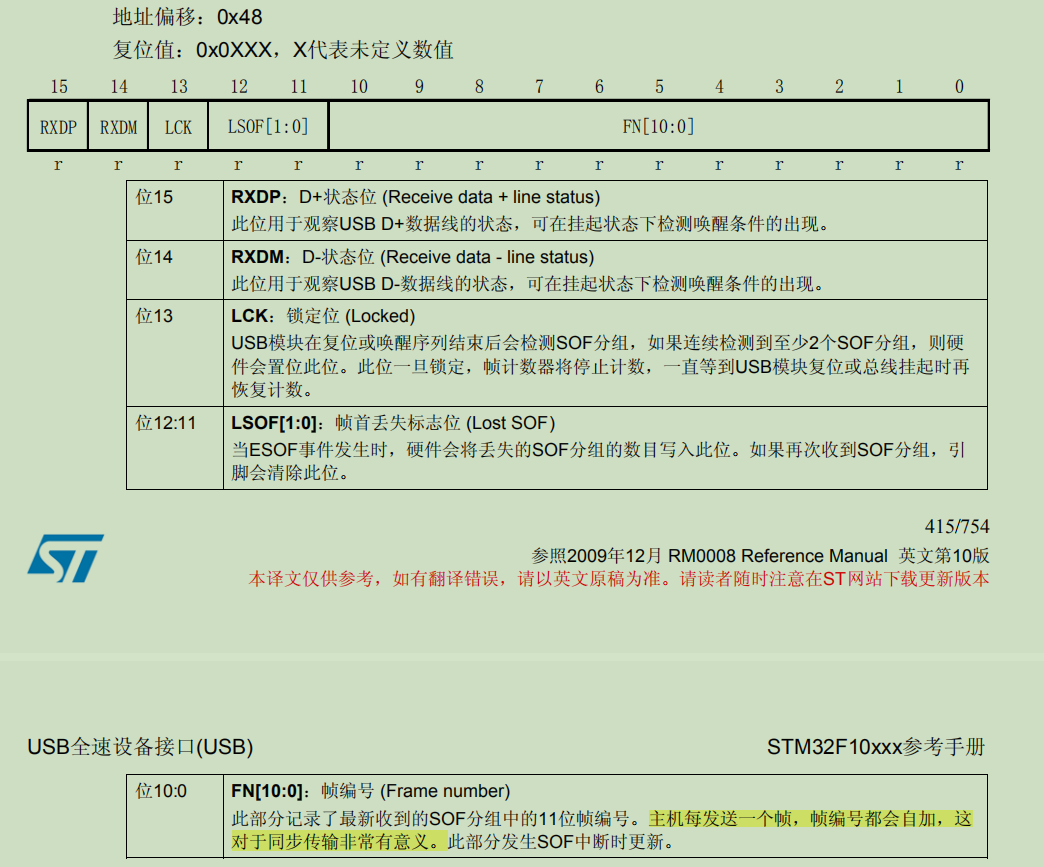




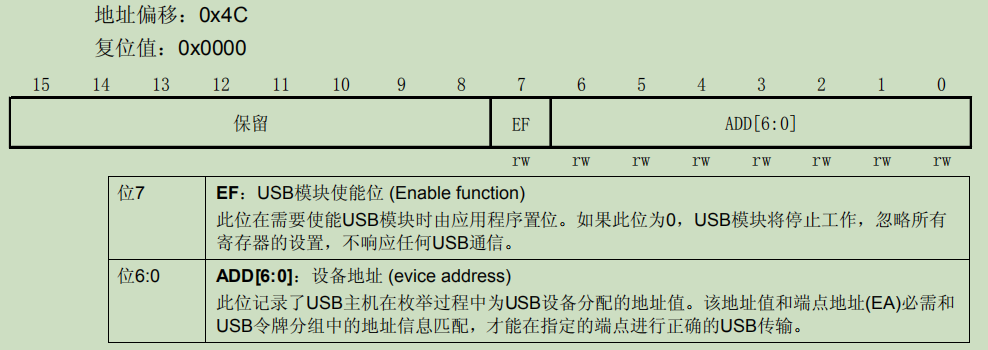




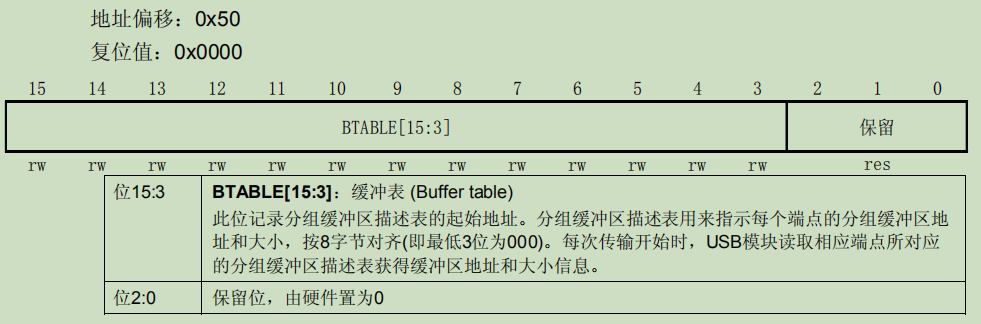
**帧编号寄存器**



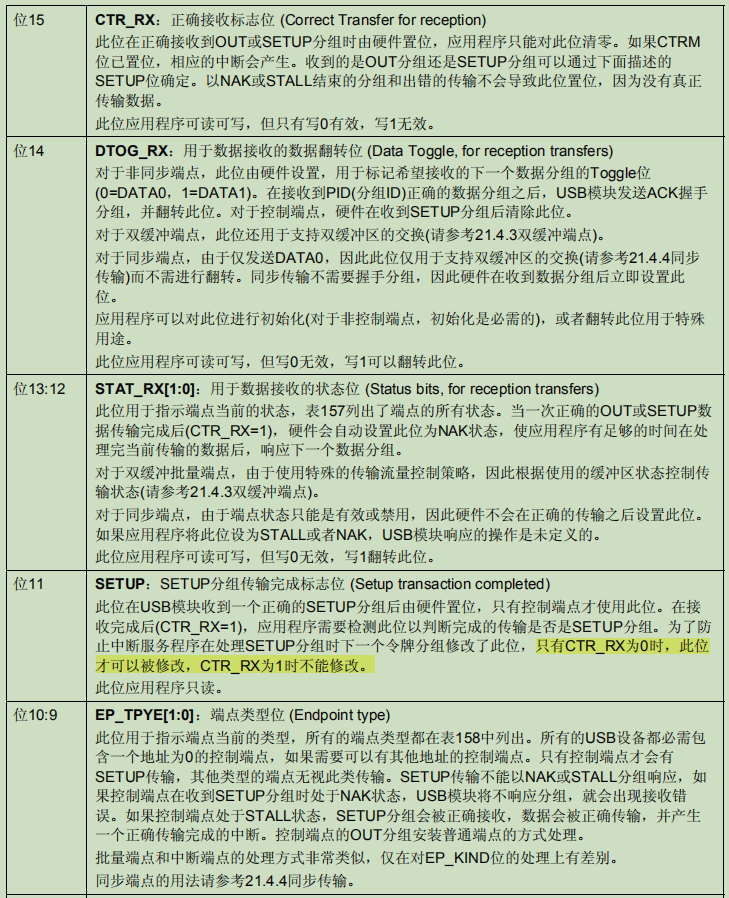
**USB设备地址寄存器**

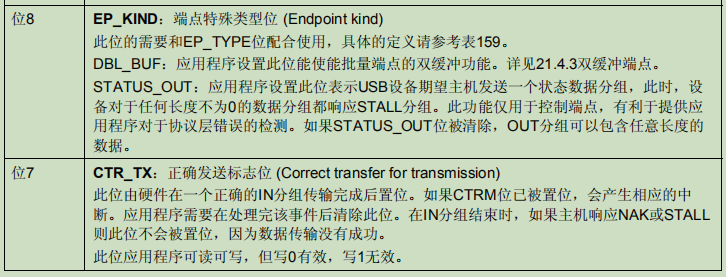


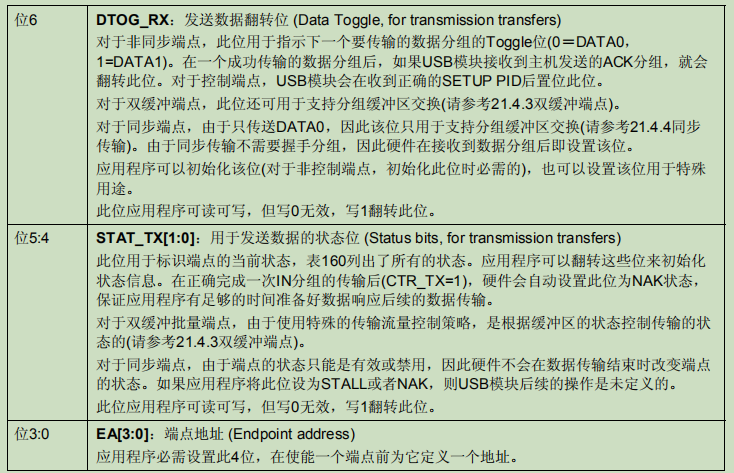
**USB分组缓冲区描述地址寄存器**



端点寄存器









**USB控制寄存器CNTR**

**USB中断状态寄存器USB\_ISTR**

**帧编号寄存器FNR**

**USB设备地址寄存器 DADDR**

**USB分组缓冲区描述地址寄存器 BTABLE**

**端点寄存器 EPnR**

**结构体函数作用及示例**

**<https://blog.csdn.net/sinat_29891353/article/details/83067747>**

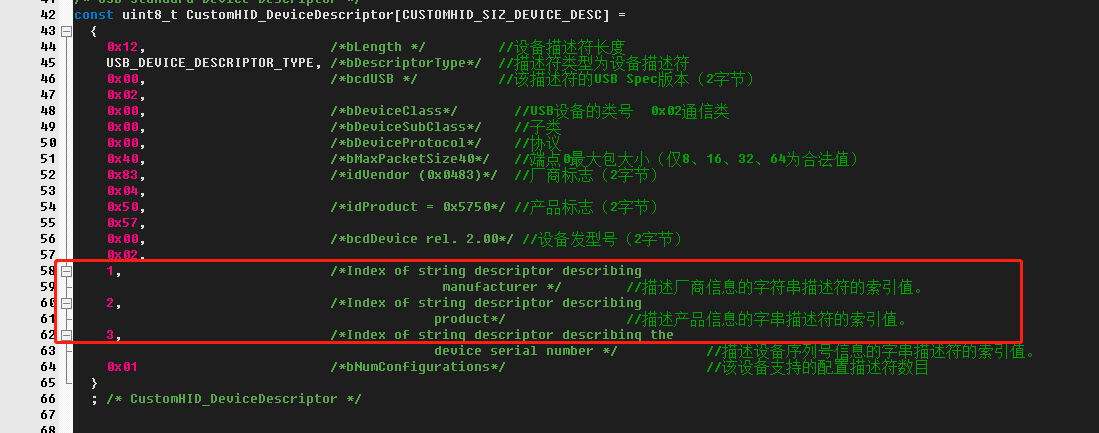
**已解决**

**<http://www.openedv.com/thread-317300-1-1.html>**

**函数名是函数的入口地址吗？**

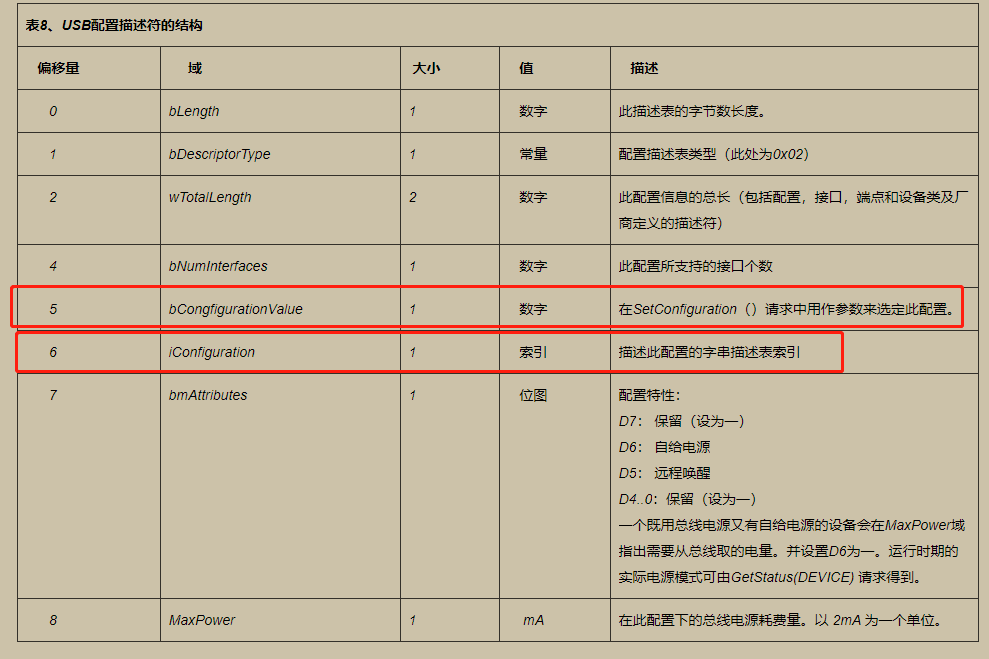
**函数名并不是函数地址的代表，这种误解与数组名就是指针一样犯了相同的错误。函数名是函数实体的代表，不是地址的代表，当然，你马上就会有疑问，平时我们不都是把函数名作为函数的地址吗？是的，我可以告诉你，函数名可以作为函数的地址，但是，绝大多数人都忽略了一个条件，从函数到指针的隐式转换是函数名在表达式中的行为，就是说，这个转换仅在表达式中才会发生，这仅是函数名众多性质中的一个，而非本质，函数名的本质就是函数实体的代表。**

**到了C++，由于C++规定，非静态成员函数的左值不可获得，因此非静态成员函数不存在隐式左值转换，即不存在像常规函数那样的从函数到指针的隐式转换，所以必须在非静态成员函数前使用&操作符才能获得地址。**



描述符索引值是干什么的？

配置描述符中



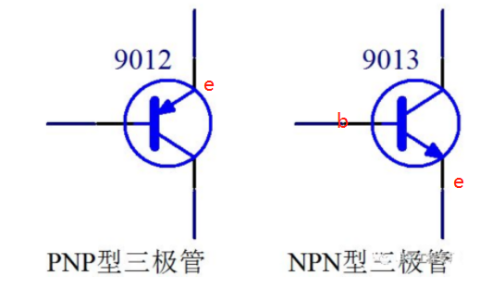
这些是干什么的，没看懂

双向端点

例如，1号端点可以启用它的输出设置（地址0x01），也可以启用它的输入设置(0x81)，也可以两个都启用。输入和输出只是公用同一个号而已，两者是独立的，EP1\_IN和EP1\_OUT是不同的。

没有调用中断函数，可能是因为没有进行宏定义。

USB的中断传输不是真正意义上的（或者说通常的）中断传输，而是按照特定的周期访问可引起中断的端点，看是否有中断发生。由于它可保证主机在最短的时间里响应和进行数据传输，所以称之为中断传输方式。显然，要求响应的时间越短，中断传输所占用带宽的比例就越大。



对于PNP 而言，e 极电压只要高于 b 级 0.7V 以上，这个三极管 e 级和 c 级之间就可以顺利导通。也就是说，控制端在 b 和 e 之间，被控制端是 e 和 c 之间。同理，NPN 型三极管的导通电压是 b 极比 e 极高 0.7V，总之是箭头的始端比末端高 0.7V 就可以导通三极管的 e 极和 c 极。

鼠标发送给PC的数据每次4个字节

BYTE1 BYTE2 BYTE3 BYTE4

定义分别是：

BYTE1 --

|--bit7: 1 表示 Y 坐标的变化量超出－256 ~ 255的范围,0表示没有溢出

|--bit6: 1 表示 X 坐标的变化量超出－256 ~ 255的范围，0表示没有溢出

|--bit5: Y 坐标变化的符号位，1表示负数，即鼠标向下移动

|--bit4: X 坐标变化的符号位，1表示负数，即鼠标向左移动

|--bit3: 恒为1

|--bit2: 1表示中键按下

|--bit1: 1表示右键按下

|--bit0: 1表示左键按下

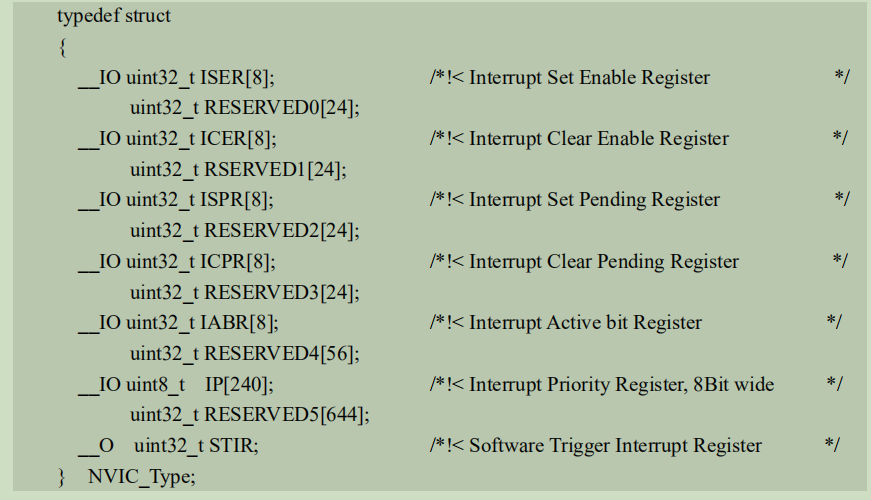
BYTE2 -- X坐标变化量，与byte的bit4组成9位符号数,负数表示向左移，正数表右移。用补码表示变化量

BYTE3 -- Y坐标变化量，与byte的bit5组成9位符号数，负数表示向下移，正数表上移。用补码表示变化量

BYTE4 -- 滚轮变化。

由于手上没有USB鼠标，对BYTE1的4-7位没有测试，对于BYTE2 BYTE3做个测试，BYTE1的4-7全为0的时候，BYTE2 BYTE3的正负表示鼠标移动方向

STM32-串口没有初始化但是调用了打印函数，导致程序死在打印函数中。



这里边是什么语法

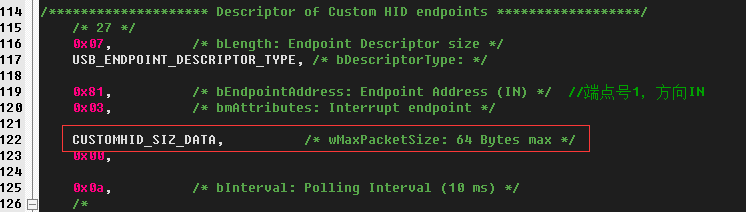
USB 2.0接口标准的三种速度规格

High-speed 25Mbps～400Mbps(最大480Mbps) (高速)

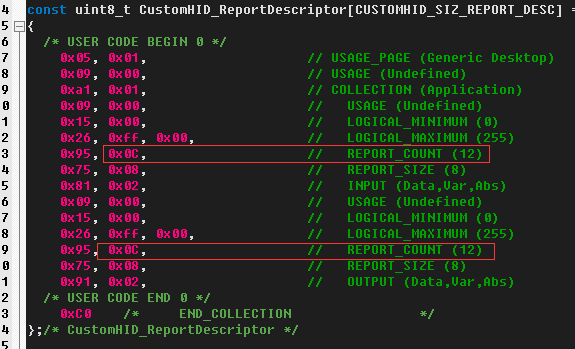
Full-speed 500Kbps～10Mbps(最大12Mbps) （全速）

Low-speed 10Kbps～100Kbps（最大1.5Mbps）

端点描述符里关于字节大小是最大的字节大小



报告描述符里的关于字节的大小是描述实际传输数据大



已解决。BUSHound能抓到数据但是上位机读不到数据的原因：

报告描述符里关于数据大小的描述为64字节

但实际上传的数据是12字节

可能因此导致HOST处理数据的时候出错。

修改报告描述符关于端点数据字节大小与实际传输的数据大小一致就可以了

结论：

BUS Hound抓到数据只能说明总线有数据，但是HOST要根据描述符来处理这些数据，因此，如果报告描述符对实际的数据描述有误就可能导致HOST处理数据异常。