



一个基于固件的软 USB 设备

古墓学派



欢迎关注我的不常更新的微信公众号

背景

大约在 2006 年，我在互联网上发现了一个神奇的项目——PowerSwitch。这是一个通过 USB 接口控制的八通道电子开关，但它的主控 MCU(ATMEL AT90S2313)本身并没有 USB 硬件接口，设计者使用两个 GPIO 口线连接 USB HOST 的 D+/D-线，通过固件捕获 D+/D-上的信号并进行解码，最终实现了一个 USB 低速设备。

AT90S2313 芯片的高速度是这个项目成功的关键，使用 12MHz 的石英晶体驱动时它具有 12MIPS 的高性能，每 8 个指令周期刚好对应了 USB 低速通信比特流（1.5Mbps）中的一个比特。所以固件可以准确地采样每一个比特，并且在采样间隔期间对比特流进行处理。由于 12MIPS 仍然不够快，所以固件只做了 NRZI 解码和 BIT UNSTUFF，没有进行 CRC 校验。在发送数据时，固件会事先计算出 CRC16 校验码并附加在数据之后，且在数据和 CRC 码准备好之前一直向主机返回 NAK。

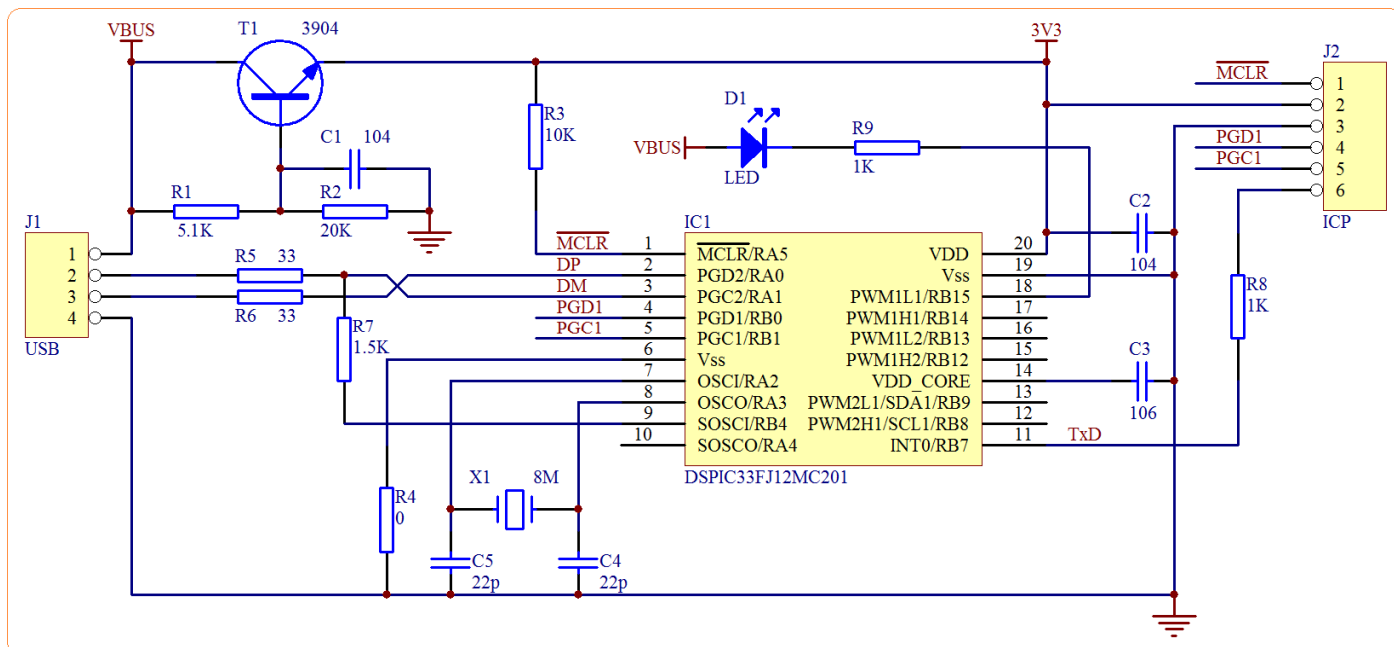
非常精巧的设计，ATMEL 公司有一篇应用笔记（AVR309）记录了这一技术的要点。不过这个项目的核心代码使用了 AVR 汇编语言，而我当时并不熟悉 AT90 系列芯片，所以我没有对此项目进行深入研究。到目前为止 PowerSwitch 演变为一个独立项目“V-USB”，移植到 ATMEGA 系列芯片上并派生出一些实用产品，比如 USBAsp，但从未在非 ATMEL 的芯片上实现过。^①

<https://www.obdev.at/products/vusb/index.html>

几年前我曾经设计过一些基于 USB 接口的产品，由于我使用的芯片都有 USB 硬件接口，所以我对 USB 的底层协议不是特别关注。这一次我准备仿照“V-USB”重新实现一个基于固件的软 USB 设备，目的是更深入地学习 USB 底层协议。我选择了 Microchip 公司的 PIC24F/dsPIC33 系列芯片，使用汇编语言和 C 语言完成编码，如果你不是 Microchip 系列 MCU 产品的专家，这个项目可能会带给你相当大的困难。又或者说，带给你一个深入研究 Microchip 系列产品的绝好途径。

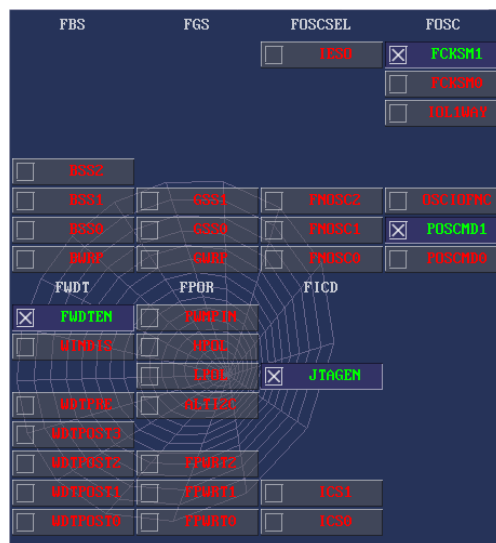
^① 最近在 Github 上发现一个基于 STM32F030 的同类项目：<https://github.com/ads830e/stm32f030-vusb>。

硬件电路



我选择了 dsPIC33FJ12MC201 这个芯片，该芯片具有 20 引脚的 SSOP 小封装，提供最大 40MIPS 的高性能。我使用 8MHz 石英晶体 X1 并通过 PLL 为 CPU 核提供 30MHz ($F_{PLLOUT}=30\text{MHz}$) 主时钟，这样 CPU 具有 15MIPS 的运行速度，每 10 个机器周期对应了 1.5Mbps 比特流中的 1 个比特。我使用了 GPIO A 中的 RA0/RA1 口线连接 USB D+/D-，由于 RA0/RA1 可直接触发“Change Notification”中断，因此不必额外再使用 INT0 口线。注意 USB D+/D- 必须连接到同一 GPIO 的两根口线上，将 D+ 连接到 RAx 上而将 D- 连接到 RBx 上是不允许的。USB D- 上的 1.5K 上拉电阻 R7 并未直连到 3.3V，而是由 GPIO B 中的 RB4 控制。RB15 口线上连接了一个 LED，主机端的测试程序将通过 USB 口发送一些数据控制这个 LED 的状态。MCLR/PGC1/PGD1 用做 ICSP 接口 J2，可以使用诸如 PICKit3 之类的编程器烧写固件代码。RB7 口线连接到 ICSP 接口 J2 的第 6 脚上，用于在必要时通过 UART 发送一些调试信息。注意 ICSP 接口 J2 的第 6 脚应该是对应 PICKit3 的 LVP 信号，我没用过 PICKit3，不知道将这个引脚连接到 RB7 上是否会影响 PICKit3。我用了一个 NPN 晶体管 T1 产生 3.3V 电源，在我的实验板上则是一颗 AMS1117-33，你也可以尝试在 VBUS 之后串接一个红色 LED 代替晶体管 T1，将 +5V 电压降至 3.2V。芯片第 6 脚到 GND 之间可以直接焊一个锡桥代替 R4，不必真正焊一个 0 欧姆电阻。我增加 R4 目的是允许把 dsPIC33FJ12MC201 更换成 PIC24F16KA101，这只需要断开第 6 脚到 GND 的连接并去掉第 14 脚上的 10 μ 电容 C3，同时把石英晶体换成 30MHz 即可。

源代码中没有直接定义熔丝位，所以在烧写代码时需手工设置“FCKSM1/POSCMD1/FWDTEN/JTAGEN”为“编程”状态，也就是 0，其它位都设为“未编程”状态即可。若使用 PIC24F16KA101 芯片外接 30MHz 晶体，则需设置“FNOSC2/FNOSC0/POSCMD0/FWDTEN”。



软件编程环境

这个项目在 WINDOWS 平台上开发并测试。我使用了 Microchip 的 XC16 编译套件，版本号 1.25。除此外我没有使用集成开发环境（MPLAB IDE）和调试器（如 ICD 4 等），也没有购买任何商用许可证。没有商用许可证意味着其 GCC 编译器仅能支持-O1 级别的优化，这基本够用了。你可以访问这个页面下载 XC16 编译器，更高版本的应该也能正常使用。

<https://www.microchip.com/development-tools/pic-and-dspic-downloads-archive>

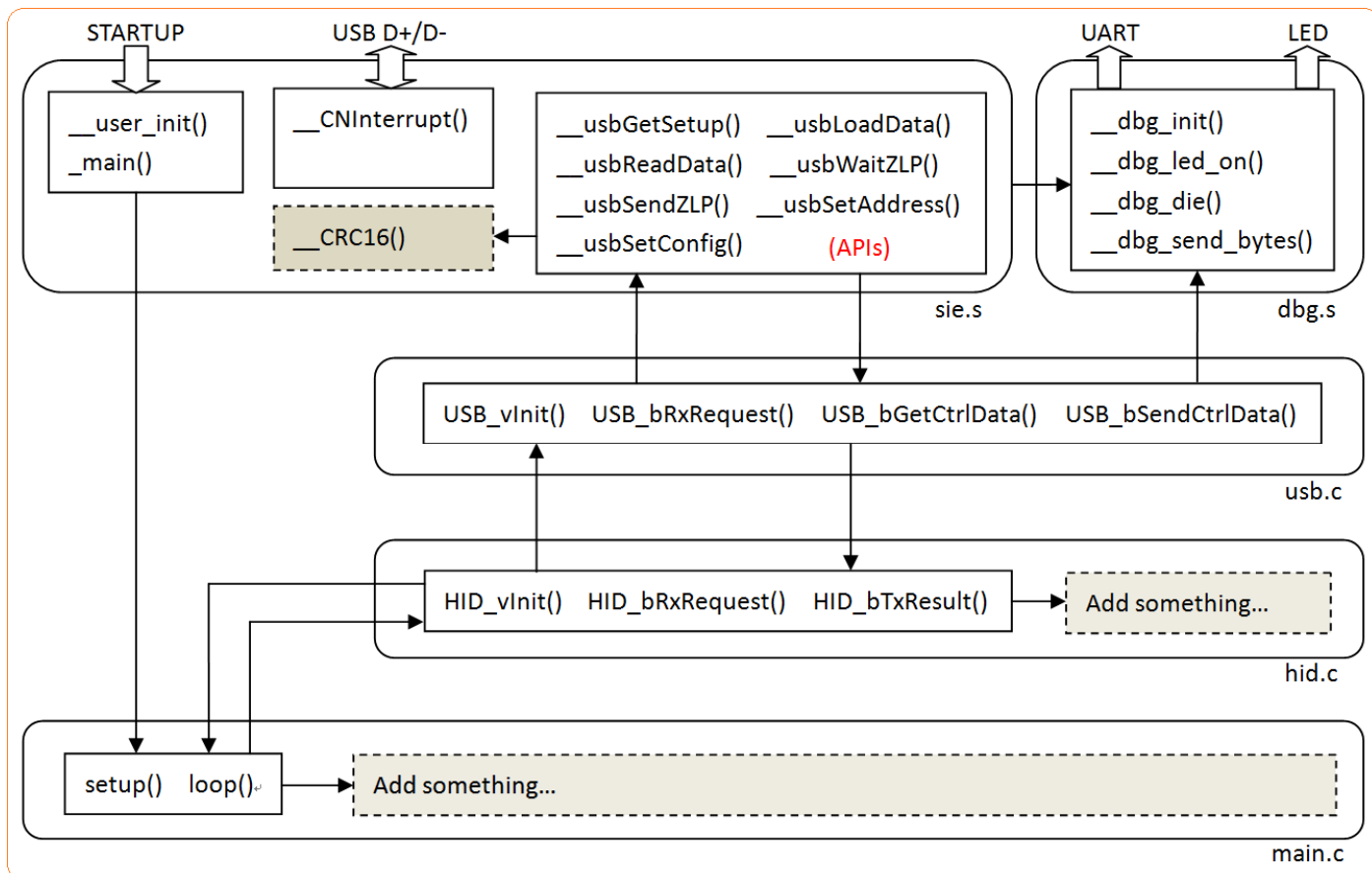
我直接使用一个批处理文件（usb.bat）对源码进行编译：

```
xc16-gcc -mcpu=33FJ12MC201 -O1 main.c hid.c usb.c sie.s dbg.s -o main.elf -T p33FJ12MC201.gld
-Wl,--defsym,__has_user_init=1,-Map=main.map
xc16-bin2hex main.elf
xc16-objdump -D main.elf >main.txt
pause
```

我给连接器传了一个参数“-Wl,--defsym,__has_user_init=1”，这样在源码文件中 sie.s 的“__user_init”函数就会被 C 启动代码调用，在 XC16 编译器安装目录中找一下名为“libpic30.zip”的库源码包，其中的“crt0_standard.s”文件引用了“__has_user_init”这个符号。

运行于主机端的测试例程是基于 Visual Studio 2008 开发的。我有一个自制的固件代码烧写器代替 PICKit3，有关这个烧写器的详情我会另外撰文。

源代码的结构



这个项目包括 5 个源码文件，其中两个是汇编语言写的（`sie.s/dbg.s`）。`dbg.s` 是用于调试的，通过 UART 输出一些信息以及通过 LED 显示一些状态，这个源码完全可以省略掉。其余三个是 C 语言源码，`usb.c` 用于处理部分 USB 标准请求，如“取描述符”等。`hid.c` 用于处理 HID 协议的“Set/Get Report”请求。`main.c` 则包括程序的初始化代码和主任务循环。需要注意的是 C 语言的入口点“main 函数”定义在 `sie.s` 之中，`main.c` 中只有 `setup` 函数和 `loop` 函数，这很象 arduino 的源码。

`sie.s` 是这个项目的核心代码，它包括了“Change Notification”中断服务程序，USB 底层的所有处理都在这个中断服务程序中进行。它还包括了存储通信数据的缓冲区和几个用于记录状态的全局变量，以及一组供 `usb.c` 调用的 API 函数，这个 API 层可以避免 `usb.c` 直接访问 `sie.s` 定义的变量。

源代码详解

参考资料和网上资源

DS52106A : MPLAB_XC16_Asm_Link_Users_Guide.pdf

DS50002071E : MPLAB_XC16_C_Compiler_Users_Guide.pdf

以上文档包含在 XC16 编译器套件中，安装之后即可看到。

DS70265A : dsPIC33FJ12MC201/202 Data Sheet

http://ww1.microchip.com/downloads/cn/DeviceDoc/70265a_cn.pdf

DS39927C : PIC24F16KA101/102 Data Sheet

http://ww1.microchip.com/downloads/cn/DeviceDoc/39927c_cn.pdf

DS70193C : dsPIC33F/PIC24H Family Reference Manual Section 10 – I/O Ports

http://ww1.microchip.com/downloads/cn/DeviceDoc/70193c_cn.pdf

DS70157F : 16-bit MCU and DSC Programmer's Reference Manual

http://ww1.microchip.com/downloads/cn/DeviceDoc/70157f_cn.pdf

USB2.0 Specification : usb_20_20190524.zip

<https://usb.org/document-library/usb-20-specification>

Device Class Definition for HID1.11 : hid1_11.pdf

<https://usb.org/document-library/device-class-definition-hid-111>

HID Usage Tables 1.12 : hut1_12v2.pdf

<https://usb.org/document-library/hid-usage-tables-112>

A Fast Compact CRC5 Checker for Microcontrollers : crc5check.pdf

<https://www.michael-joost.de/crc5check.pdf>

Cyclic Redundancy Checks in USB : crcdes.pdf

<https://usb.org/document-library/cyclic-redundancy-checks-usb>

<https://docs.microsoft.com/zh-cn/windows-hardware/drivers/usbcon/>

<https://docs.microsoft.com/en-us/windows-hardware/drivers/ddi/hidsdi/>

如果无法从 Microchip 以及 USB 组织官网上下载到这些文件，可以从我这个站点上下载。下载包中部分文件是中英文双版本的，建议阅读英文版，以免中文翻译有错漏。

<http://wiki.geniekits.com/lib/exe/fetch.php?media=downloads:ya-vusb-refs.zip>

我使用了我的前雇主飞天诚信科技股份有限公司（<https://www.ftsafe.com.cn/>）的 USB Vendor ID，PID 是我自己随便设的。如果你需要一个专属的 Vendor ID，可以访问 USB.org 官网（<https://usb.org/developers>），这里告知了如何获得独立的 Vendor ID。

这个项目中所有由我编写的源代码及文档都采用 MIT 开源许可证，所包含的其它参考资料都是在互联网上公开发表的，版权属于其发布者。我仅出于方便大家获取的目的包含了这些资料。

源码中的要点

/*-----*/
 /*- <1> 参考资料摘要 -*/
 /*-----不华丽的分割线-*/

- USB 总线供电电压：VBUS=4.75V – 5.25V
- D+/D-线输出的高电平：VOH=2.8V – 3.6V
- D+/D-线输入的高电平（最低）：VIH=2.0V
- D+/D-线输出的低电平：VOL=0.0V – 0.3V
- D+/D-线输入的低电平（最高）：VIL=0.8V

以上参考 USB2.0 Specification 第 178 页 7.3.2 节。对比 DS70265E 第 229 页 TABLE24-9 和第 232 页 TABLE24-10，可知我们可以直接将 USB D+/D-线连接到 dsPIC33 芯片的 GPIO 口上，并通过固件读取或输出 0/1 信号。

- D+/D-线上的单端 1 (SE1)：D+ and D- > 0.8V (RA1-RA0 = 11b)
- D+/D-线上的单端 0 (SE0)：D+ and D- < 0.3V (RA1-RA0 = 00b)
- D+/D-线上的差分 1 (DF1)：D+ > 2.8V and D- < 0.3V (RA1-RA0 = 01b)
- D+/D-线上的差分 0 (DF0)：D+ < 0.3V and D- > 2.8V (RA1-RA0 = 10b)

以上参考 USB2.0 Specification 第 144 页 7.1.7 节。我们使用 RA0 连接 D+，RA1 连接 D-，括号中为通过 GPIO A 口读入或输出的信号。对低速设备来说，差分 0 (DF0) 对应着 J 状态，也相当于 D+/D-上无任何活动时的 IDLE 状态，差分 1 (DF1) 对应 K 状态。

参考 USB2.0 Specification 第 141 页 7.1.5.1 节，我们可以看到 USB 设备连接到主机或 HUB 端口之后的电路细节。从下图中可见主机或 HUB 端口的 D+/D-被两个 15K 电阻下拉，低速 USB 设备的 D-线上有一个 1.5K 的上拉电阻 R_{pu}，上拉至 3.3V，主机或 HUB 通过此上拉电阻感知设备插入并确认此设备为低速设备。

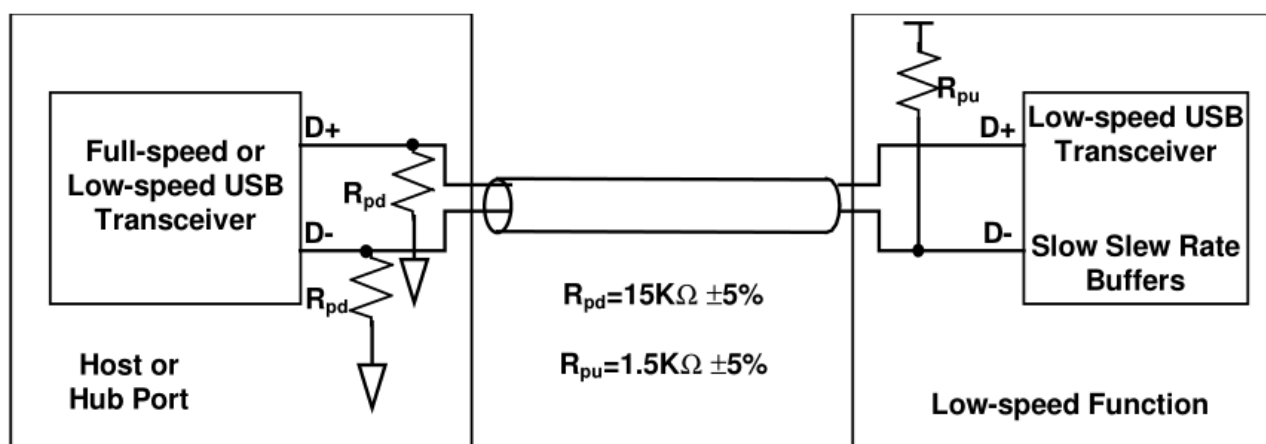


Figure 7-21. Low-speed Device Cable and Resistor Connections

在感知到设备插入后，主机或 USB HUB 会首先发出一个“BUS RESET”信号，由一个持续时间不小于 10ms 的 SE0 形成，注意必须区分“BUS RESET”和“Keep-alive”两者的不同之处。参考 USB2.0 Specification 第 153 页 7.1.7.5 节和第 332 页 11.8.4.1 节。之后主机会使用初始设备地址 (0000000b) 向设备的 0 号端点发送 SETUP 请求，直至主

机为设备分配了新的地址。

主机或 USB HUB 总是分三步完成一次数据传输，第一步是发送一个令牌包 (TOKEN) 给设备，这一步的传输方向是从主机到设备；第二步是传输令牌附加的数据包 (DATA)，方向既可能是从主机到设备又可能是从设备到主机，这在前面的令牌中有指示；第三步则是在数据包之后，接收数据的一方需要发出一个握手包 (HANDSHAKE)。每一个步骤都带有一个字节的 PID，令牌包使用的 PID 包括“SETUP/OUT/IN”三个，其中“SETUP/OUT”指示后面附加的数据包是主机发给设备的，“IN”则指示出设备需要向主机发回数据包，令牌之后的数据包使用“DATA0/DATA1”这两个 PID，而最后的握手包使用“ACK/NAK/STALL”这三个 PID。PID 的定义请参考 USB2.0 Specification 第 196 页 Table8-1。

一个令牌包共有 3 个字节，包括 1 字节“PID”和 2 字节的“设备地址/端点号/CRC5 校验码”。设备地址初始为 0，之后再由主机分配后发送给设备。端点用于组织不同类型的数据传输，我们目前只使用端点 0 进行通信。设备地址和端点可以参考 USB2.0 Specification 第 197 页 8.3.2 节，我在后文还会有更多的说明。令牌包的定义可以参考 USB2.0 Specification 第 199 页 8.4.1 节。

之后的数据包共有最多 11 字节，因为我们使用最多 8 字节的有效数据长度，所以 11 个字节中包括 1 字节 PID 加 8 字节有效数据加 2 字节 CRC16 校验码。若有效数据不够 8 字节，CRC16 校验码将直接附加在有效数据之后，此类数据包我称之为“短包”。有效数据可以是 0 字节，这时数据包就只包括 PID 和 CRC16 校验码，此类数据包我称之为“空包”或者“零长度包”。参考 USB2.0 Specification 第 206 页 8.4.4 节。标识数据包的两个 PID (DATA0/DATA1) 交替使用，若前一个数据包以“DATA0”引导并被成功接收，则后一个数据包就以“DATA1”引导，成功之后再切换回“DATA0”，如此交替进行。

握手包只有 1 字节 PID。我的代码仅处理“ACK/NAK”两种类型的握手包，当主机发出“SETUP”或“OUT”这两类令牌包以及后附数据包之后，设备发出“ACK”表示令牌包及数据包正常收取了，允许主机进行后续数据通信。设备发出“NAK”则表示令牌包及数据包无法正常收取，要求主机重新发送。请参考 USB2.0 Specification 第 206 页 8.4.5 节。必须注意如果主机发出的令牌包是“SETUP”且后附了数据包，设备必须收取数据并发出“ACK”，不允许对“SETUP”令牌包发回“NAK”或“STALL”。请参考 USB2.0 Specification 第 209 页 8.4.6.4 节。当主机发出“IN”这个令牌包时，如果设备没有准备好发给主机的数据，就需要发出“NAK”，主机会重新发出“IN”这个令牌包。如果设备准备好了数据，就直接发出数据包，然后等待主机发来“ACK”或者“NAK”这两种握手包。若主机发来“ACK”，说明数据正常收取了，允许设备进行后续通信。若主机发来“NAK”，则设备必须重新发送刚才的数据包。

进行 CRC5 和 CRC16 校验时，PID 字节并不包含在被校验的数据中。PID 字节的低 4 位由 USB2.0 Specification 第 196 页 Table8-1 定义，高 4 位是低 4 位的反码。例如“SETUP”定义为 1011b，其反码是 0100b，所以完整的 PID 字节是 01001011b (0x2D)。

USB 数据传输总是由主机发起，由设备响应。所有的数据传输阶段（令牌/数据/握手）在发送之前都预先发出一个 SYNC 字节引导，数据包发送之后都附加一个 EOP (End Of Packet) 指示发送完毕。SYNC 字节的值是 10000000b (0x80)，由于是低位先发，所以设备将收到 00000001b 这样一个比特流。USB 总线采用 NRZI 编码，具体来说当被发送的比特为 0 时，D+/D-的状态将发生切换，从 J 状态改变到 K 状态，或从 K 改变到 J，当被发送的比特为 1 时，D+/D-的状态保持不变。由于 D+/D-初始为 J 状态，所以 00000001 这个 bit 流将被编码为“KJKJKJKK”。请参考 USB2.0 Specification 第 157 页 7.1.8 节和第 159 页 7.1.10 节。EOP 由 2 比特的 SE0 和 1 比特的 J 状态组成。请参考 USB2.0 Specification 第 145 页 Table 7-2 和第 183 页 Table7-10。

如果 NRZI 码流之中出现连续的 6 个“1”，也就是“KKKKKKK”或“JJJJJJ”，这时数据发送方会在 6 个“1”之后自动插入一个“0”，强制翻转 D+/D-线的状态，这个机制被称为“BIT-STUFF”。这个自动插入的“0”并不属于有效的数据，接收方需要剔除这个多余的“0”。请参考 USB2.0 Specification 第 157 页 7.1.9 节。

USB 总线以“帧”的形式组织连续的数据传输，帧间隔时间为 1mS。我的理解是一次完整的数据传输过程是不能跨越帧边界的，不能在前一帧里发出令牌包而在下一帧里发出这个令牌包附加的数据包。当前这一帧剩余时间不足以完成一次完整的数据传输时，这一点剩余时间就会被放弃，数据传输将在下一帧开始。请参考 USB2.0

Specification 第 36 页 5.3.3 节和第 159 页 7.1.12 节。注意“Keep-alive”信号会在每一帧开始时都会发出，即大约每毫秒就会产生一次，这个信号是由 2 比特的 SE0 加 1 比特的 J 状态组成的，和 EOP 信号一致。请参考 USB2.0 Specification 第 144 页的 7.1.7.1 节和第 332 页的 11.8.4.1 节。

对于 PIC24F/dsPIC33 这款 CPU 而言，大多数指令都在一个机器周期内完成，无条件转移指令需要两个机器周期，条件转移指令当条件满足真正发生了转移时需两个机器周期，条件不满足转移未能发生时只需一个机器周期。另外，需注意芯片内建的中断控制器具有 5 机器周期的延迟，而从中断（或子程序）返回需三个周期。参考 DS70157F 第 39 页 3.2.1 节和第 62 页 4.3 节，以及 DS70265E 第 3 页对 Interrupt Controller 的描述。

我们使用的时间单位，不仅有常规的秒/毫秒/微秒，还常用“bit”做为时间单位。对于 USB 低速设备来说，传输 1bit 所用时间是 1/1.5 微秒，约 0.667 微秒，对应了 CPU (@15MIPS) 的 10 个机器（指令）周期。

```
/*-----*/
/*- <2> D-线上的上拉电阻 -*/
/*-----不华丽的分割线-*/
```

在硬件原理图中可见 D-线的上拉电阻由 RB4 口控制，只有固件通过 RB4 输出高电平时主机或 HUB 才会感知到设备插入，这允许固件在通过 RB4 输出高电平之前执行一些较复杂的耗时比较长的初始化代码。我在源码“sie.s”中的“__user_init()”函数里设置 RB4 为高，这步操作也可以转移到源码“main.c”的“setup()”函数中进行。任何时候固件通过 RB4 输出低电平或高阻态，都会导致主机或 HUB 认为设备已经拔掉了，而设备无需真正拔除。请参考 USB2.0 Specification 第 149 页 7.1.7.3 节。

如果将 D-线的上拉电阻直连 3.3V 电源，源码“sie.s”是不用修改的，除非你需要将 RB4 线用作其它用途。我没有试过使用 RA 口的某个口线控制上拉电阻，你可以检查一下源码，看看在操作 RA0/RA1 两个口线时是否影响了其它 RA 口线。

我使用的第一个带有 USB 接口的 MCU 是 CYPRESS 公司的 CY63001，这个芯片使用 5V 电源，因此 USB 接口的 D-线是由一个 7.5K 电阻上拉至 5V 的。7.5K 和主机端的 15K 下拉电阻分压可使 D-线偏置在 3.3V，满足 USB 规范的要求。

附加一句提示，在函数“__user_init()”中，我禁用了所有引脚的模拟输入和开漏输出功能。这没有什么必要，你可以重新启用你需要的模拟输入与开漏输出功能。

```
/*-----*/
/*- <3> 设备插入主机端口 -*/
/*-----不华丽的分割线-*/
```

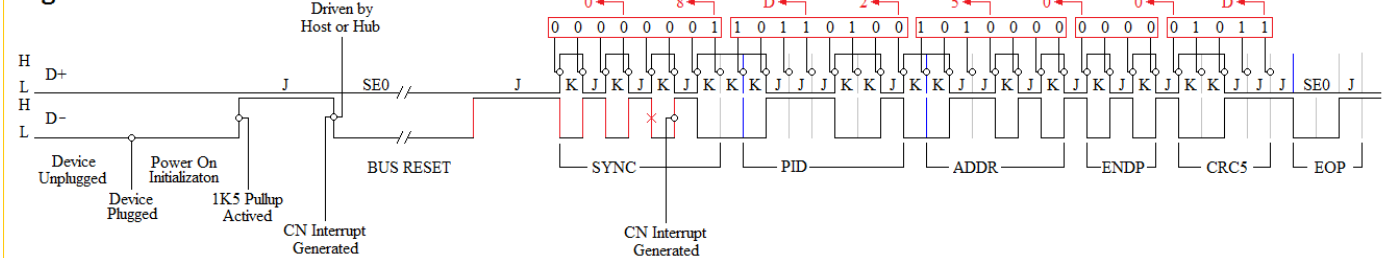
当设备插入主机或 HUB 端口中并被探测到之后，主机或 HUB 会首先发出一个“BUS RESET”信号，由于总线在 IDLE 状态时 D-被 1.5K 电阻上拉至高电平，在“BUS RESET”期间 D-被驱动到低电平，因此我启用了 D-（RA1）线触发 Change Notification（CN3）中断（参考 DS70193D 第 10-7 页）。注意如果你还需要启用其它中断，必须确保 CN 中断的优先级是最高的。

需要注意的是主机或 HUB 探测到低速设备之后，会在相应端口自动发送“Keep-alive”信号，在每一个数据帧开始时发出，因此在 Change Notification 中断服务程序中，必须区分“BUS RESET”和“Keep-alive”信号。如果采用 D+（RA0）线触发 Change Notification（CN2）中断，硬件可以忽略“BUS RESET”和“Keep-alive”信号，这可以减少代码量和中断频率。

下图（figure01）中显示了设备插入主机或 HUB 后接收到的比特流。在“BUS RESET”之后，主机或 HUB 向设备发出的第一个字节是“SYNC”。每一次 D+/D-的状态切换（红色的脉冲边沿）都会触发 CN 中断，固件在 CN 中断服务程序中尝试捕获“SYNC”字节最后的“JKK”三个比特。当成功捕到“JKK”之后，中断服务程序就不会退出，继

续捕获后面的所有比特直至收到 EOP 后才会结束。

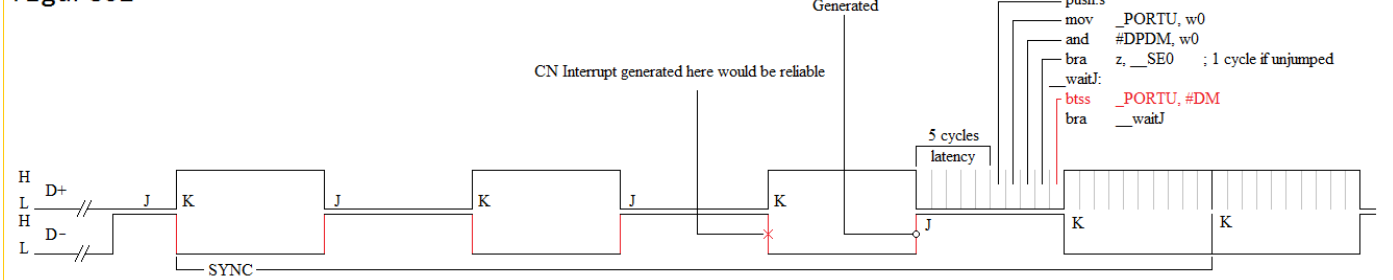
figure01



/*-----*/
 /*- <4> 捕获 SYNC 字节 -*/
 /*-----不华丽的分割线-----*/

进入 CN 中断服务程序之后，固件首先使用“push.s”指令将工作寄存器 W0-W3 存入影子寄存器，然后使用“mov _PORTU, w0”指令读取 D+/D- 的状态，_PORTU 已经预定义为 PORTA。由于中断控制器具有 5 机器周期的延迟，指令“push.s”在第 6 周期执行，所以 D+/D- 的状态是在第 7 个机器周期读取的。1 个比特对应 10 个机器周期，在第 7 周期采样 D+/D- 的状态略有迟缓，但并未影响可靠性。同时读取 D+ 和 D- 两根线的状态是为了判断此时 D+/D- 线上是否为一个 SE0，如果是则转到“__SE0:”处继续区分当前是“BUS RESET”还是“Keep-alive”。

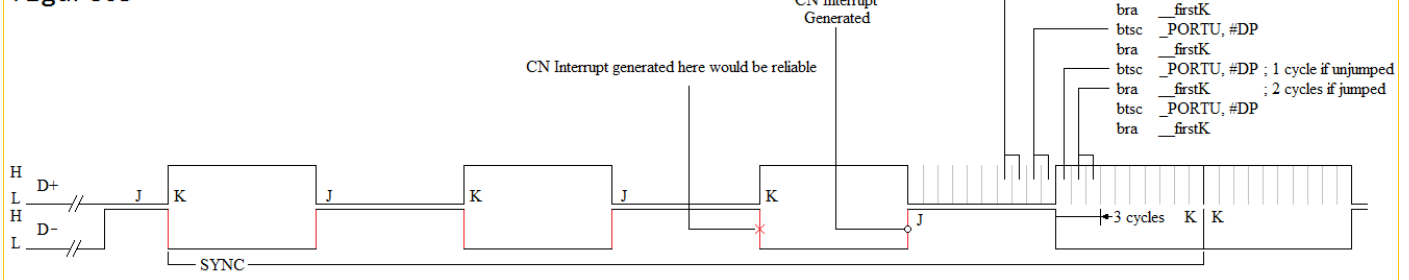
figure02



为了捕到“JKK”这三个 bit，固件首先利用一个循环确保 D+/D- 当前处于 J 状态。从上图（figure02）中看如果 CN 中断发生在 J 位的上升沿这里（由黑色的 o 标记出），则首次执行“btss _PORTU, #DM”指令时已经是这个 J 位的最后一个周期了，因此这个 J 状态可能很难可靠地捕获到。所以我认为应该是由前一个 K 状态的下降沿（由红色 X 标记出）触发 CN 中断，在“__waitJ:”循环中等到所需的 J 状态。

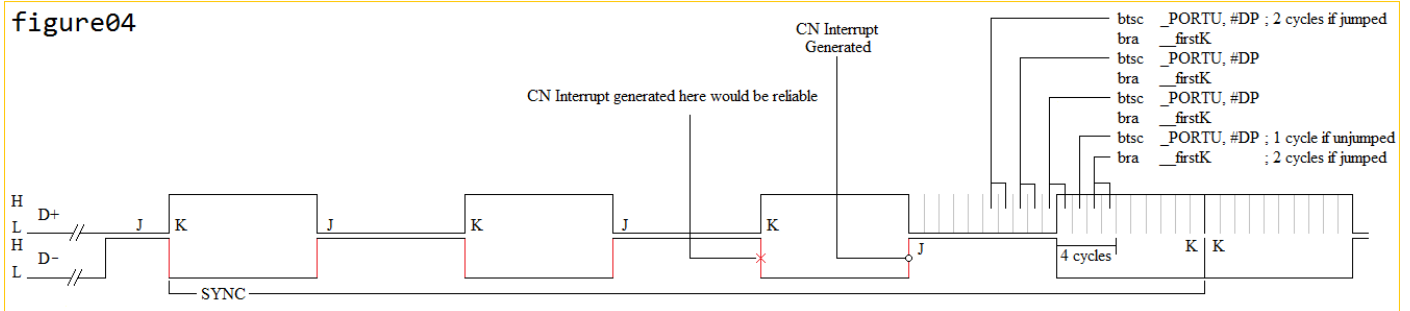
捕获下一个 K 位使用的代码略有奇怪，这样做有助于获得比较一致的延迟时间。从下面两图（figure03/figure04）中我们可以看到指令执行的时机和延迟之间的关系。可以看到延迟时间为 3 或 4 个机器周期。

figure03



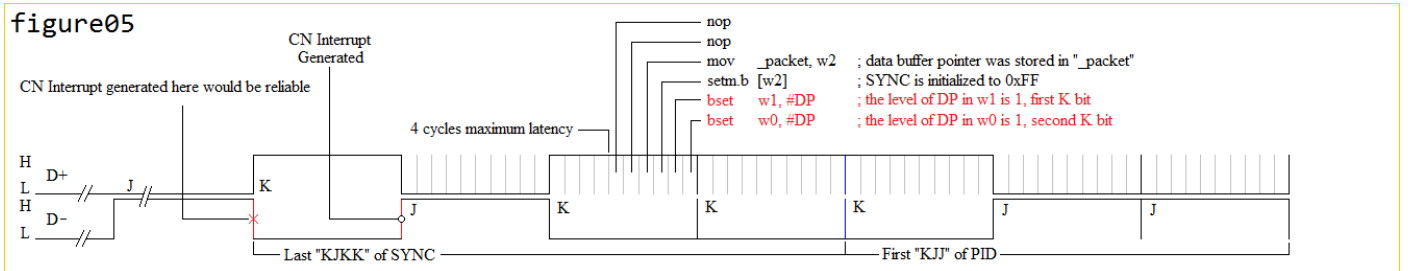
上图显示了指令“btsc _PORTU, #DP”在第一个 K 位的第一周期执行时形成的 3 机器周期的延迟。下图则显示了指令“btsc _PORTU, #DP”在第一个 K 位的第二周期执行时形成的 4 机器周期的延迟。

figure04



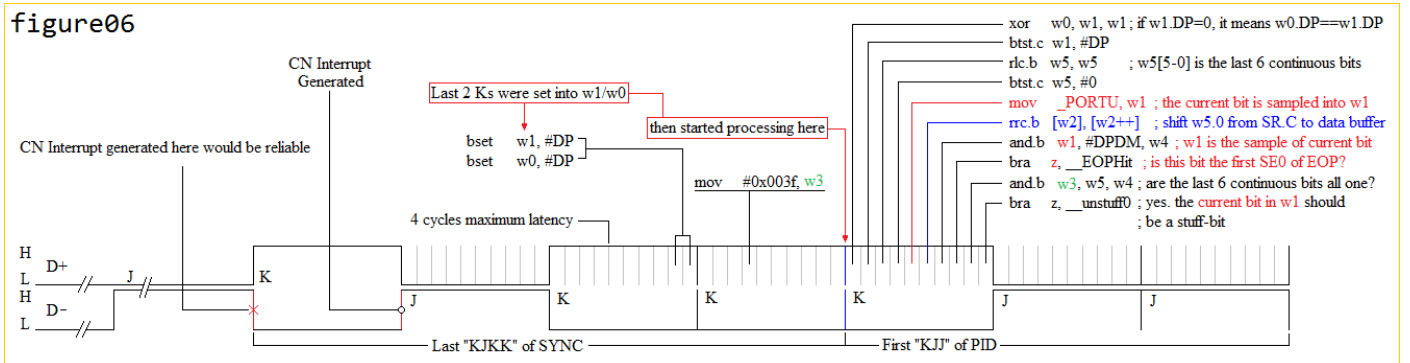
在成功捕获到“SYNC”字节最后的“JKK”位之后，固件需要完整地接收这一个比特流，包括“SYNC”字节。因为“SYNC”字节是固定的且已经错过了前面的“KJKJK”这 5 个位，所以“SYNC”字节应该直接填入缓存才好，但我做的有所不同。我先将缓存中的“SYNC”字节初始化为“0xFF”，然后使用只有 10 条指令的一小段代码获得“SYNC”字节的最后一个比特，也就是最后两个“KK”位所表示的 1。程序计算出的这个比特是 0，所以缓存中的“SYNC”字节最终是“0x7F”，也就是“0x80 的反码”。USB 总线上传的所有数据都以反码的形式保存到缓存中。下图 (figure05) 显示了计算“SYNC”字节最后一个比特之前所做的准备工作：在寄存器 W0/W1 各置入一个 K 状态。

figure05



下图 (figure06) 是计算这个比特的 10 条指令。固件使用“异或”操作判断两次采样的结果是相同还是相反，当两次对 DP 线采样为相同值时，指令“xor w0, w1, w1”的计算结果是 0，恰好与 NRZI 编码规则相反，所以我们收到的数据都是反码。此计算结果被指令“btst.c w1, #DP”送入 C 标志位，然后分别移入 W5 寄存器和[W2]指向的数据缓存。因为我想确保在第 5 周期（每一位的中点）采样 D+/D- 当前的状态，所以指令“mov _PORTU, w1”插入到“btst.c w5, #0”和“rrc.b [w2], [w2++]”之间了。

figure06



这段代码计算出每字节的最后一位，并采样后一字节的第 0 位，所以“rrc.b”指令的目的操作数是“[w2++]”。在图例 (figure06) 中这段代码计算出 SYNC 字节的 bit7 位，同时采样了 PID 字节的 bit0 位。

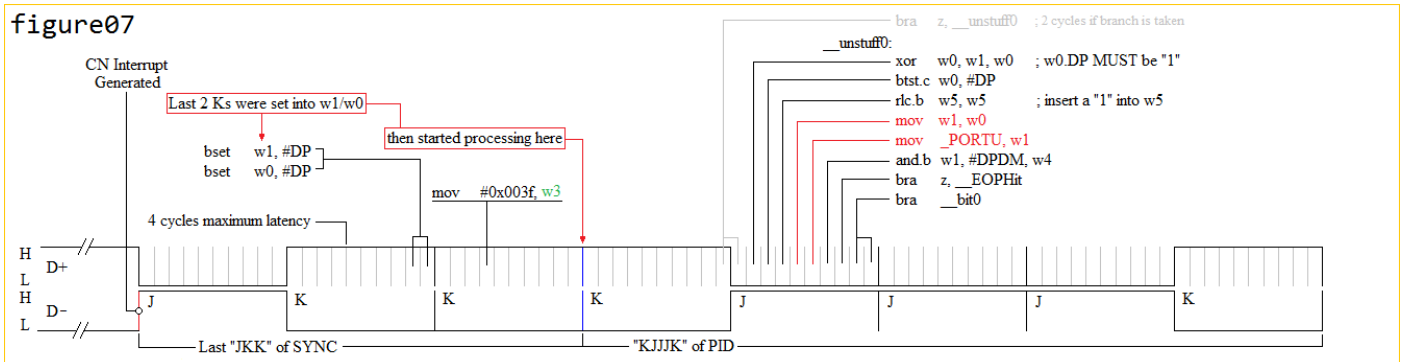
/*-----*/
/*- <5> 探测并移除 stuff bit -----*/
/*-----不华丽的分割线-----*/

从图例 (figure06) 中可以看到，寄存器 w5 用来观察数据流中是否出现了 6 个连续的 1。如果在第 9 周期（指

令“and.b w3, w5, w4”) 探测到一个“stuff-bit”(w5 寄存器低 6 位为 000000b), 则固件在第 10 周期(指令“bra z, __unstuff0”) 转移至“__unstuff0:” 继续执行。固件将抛弃 w0 寄存器中的采样, 将 w1 寄存器中对当前位的采样移入 w0 中, 并把下一个位采样到 w1 寄存器中。下图(figure07)中所示的 PID 字节的第 1 个 K 状态当然不是一个“stuff-bit”了, 但这部分程序用于接收完整的比特流, 在后续数据中遇到“stuff-bit”时就会转移到“__unstuff0:” 这里。

从下图(figure07)中还可以看到前 3 个指令用于向 w5 寄存器中插入一个“1”, 这可以简化成“bset w5, #0” 加一个“nop”, 这样可以使指令“mov _PORTU, w1” 在第 5 周期执行, 现在是在第 6 周期执行此指令。第 4 个指令“mov w1, w0” 将寄存器 w1 中的采样复制到 w0 中, 下一个指令“mov _PORTU, w1” 重新采样 D+/D-的电平。最后的“bra __bit0” 指令需要两个机器周期。

figure07



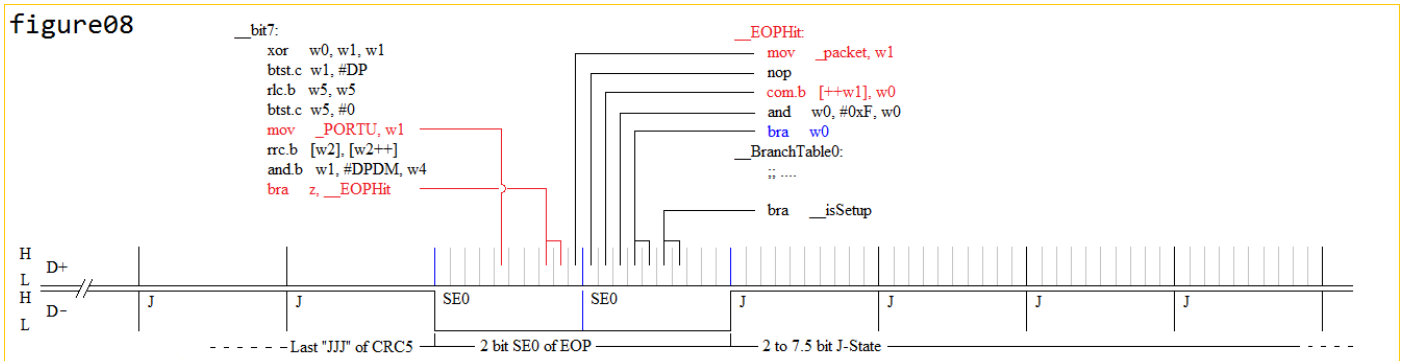
上图(figure07)中我们假设 PID 字节的第一个 K 位是需要丢弃的 STUFF 位。在图例(figure06)中这个 K 位是采样到寄存器 w1 中的, 这里将它复制到 w0 中, 并将当前的 J 位采样到 w1 中。这样图例(figure07)中蓝色竖线标出的这个数位就不再计算了。

/*-----*/
/*- <6> EOP 信号和 bit7 位 -*/
/*-----不华丽的分割线-----*/

在对每一个字节的 bit0 和 bit1 进行采样之后, 固件都测试采样值是否为“SE0”, 这是因为 USB HUB 有一个“EOP dribble”的问题。参考 USB2.0 Specification 第 157 页 7.1.9.1 节。固件这个处理机制似乎不大可靠, 设备连接于 HUB 上时通信偶有出错。参见文后“已知的 BUG”一节。下图(figure08)中左侧代码是在计算 CRC5 字节的 bit7 位同时采样下一字节的 bit0 位, 这一位已经是 EOP 信号的第一个 SE0 了。

对 bit7 进行采样后的处理流程有所不同, “bra z, __unstuff7” 指令在第 8 周期执行, 而非第 10 周期, 这是为了把第 9 和第 10 周期留给指令“bra __bit7”, 以形成一个循环收取完整的 bit 流, 这也导致了“__unstuff7:” 这部分代码有所不同。我没有使用 dsPIC33 特有的无开销硬件循环(DO-LOOP), 这有助于向 PIC24F 芯片上移植。我没有制作一张图展示这段代码, 你可以阅读源码文件“sie.s”中标号“__bit6:”之下的 9 个指令。

figure08

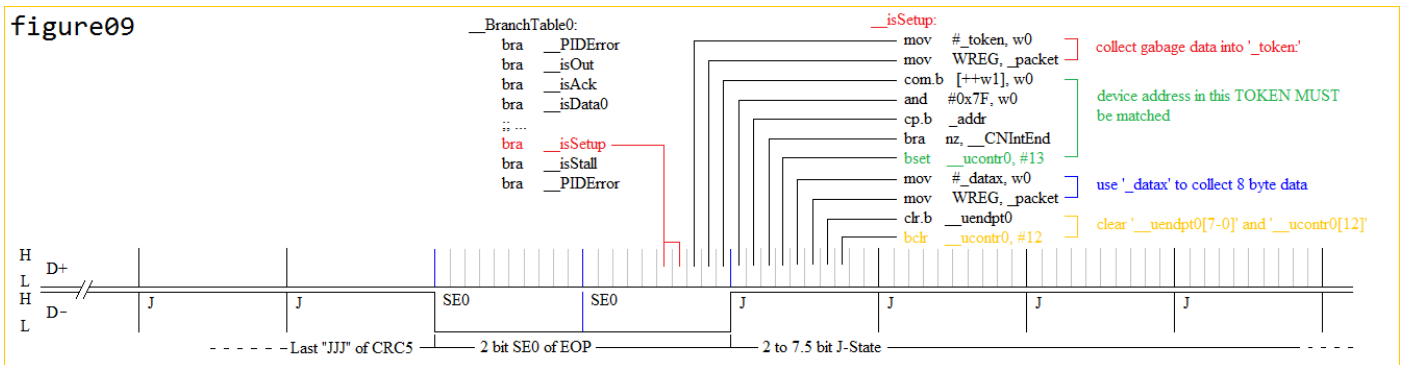


收到 EOP 的第 1 个 SE0 后, 固件使用一个“散转表”分别处理各个 PID, 这有助于获得固定的延迟时间。参见

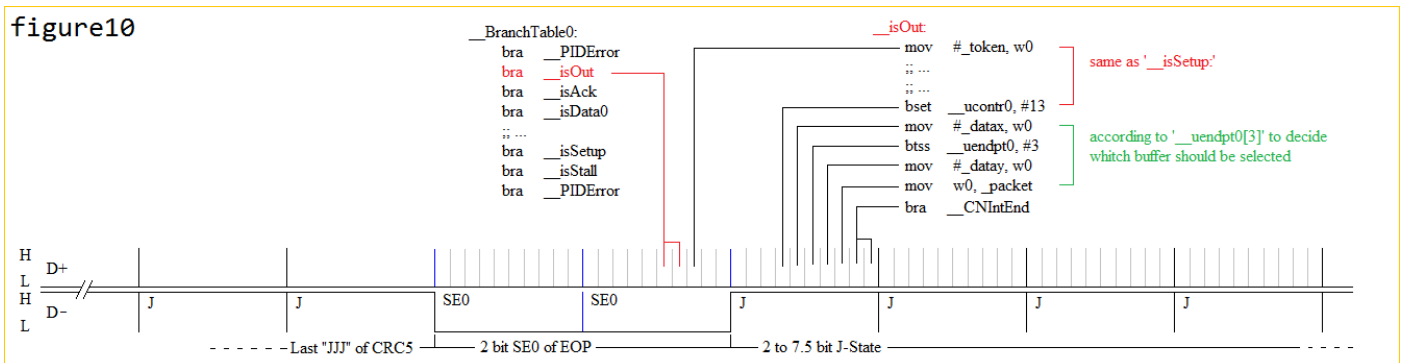
上图 (figure08)。指令 “mov _packet, w1” 将当前缓冲区首地址置入寄存器 w1 中, 指令 “com.b [++w1], w0” 则是将 PID 字节取反后送入寄存器 w0 中。

/*-----*/
/*- <7> 处理 SETUP 和 OUT 令牌 -*/
/*-----不华丽的分割线-----*/

对“SETUP”令牌的处理分为 3 个步骤, 第一步是对令牌中的设备地址进行比对, 当令牌中的地址和变量“_addr:”中保存的当前设备地址匹配时, 固件将变量“__ucontr0”的 bit13 位设为“1”, 表明本次“SETUP”附加的数据包允许接收。若不匹配, 固件就直接退出 CN 中断服务程序了。退出中断后固件会确保之后收到的任何数据都存储在“_token:”这个缓存中, 因为 USB HUB 会把所有数据传输都“广播”给全部下行端口, 因此必须保证不是发给我们的(令牌中的设备地址不匹配)数据一定收入“_token:”缓存中, 不会冲掉“_datax:”和“_datay:”缓存中未被处理的数据。第二步把“_datax:”设置为当前缓存区, “SETUP”令牌附带的 8 字节数据包必然通过“DATA0”发送, 所以一律收入“_datax:”。第三步是初始化变量“__uendpt0”的低字节为“0x00”, 表示一次“控制传输”现在开始了。同时将变量“__ucontr0”的 bit12 位设为“0”, 指明接在本次“SETUP”之后的“IN”或“OUT”须通过“DATA1”传输数据包。下图 (figure09) 显示了“SETUP”令牌的处理过程。



对“OUT”令牌的处理只有两个步骤, 第一步和处理“SETUP”令牌时一样, 比对设备地址。第二步是根据变量“__uendpt0”的 bit3 位确定后附数据是用“_datax:”缓存还是“_datay:”缓存来接收。具体来说, 当“__uendpt0[3]”等于“0”时, “_datax:”已被占用, 此时要使用“_datay:”; 当“__uendpt0[3]”等于“1”时, “_datay:”已被占用, 此时要使用“_datax:”。这里没有对“__uendpt0[7-0]”做任何改变, 只有接收了后续数据包并返回“ACK”时才会修改“__uendpt0[7-0]”。



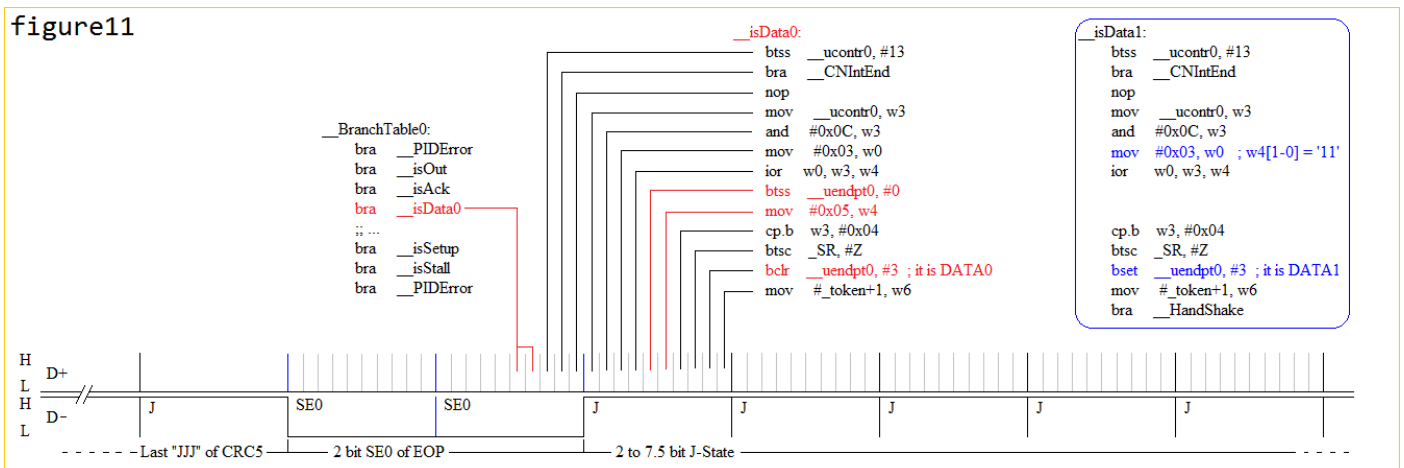
CN 中断服务程序在收取了“SETUP”或“OUT”令牌后就直接退出了, 没有继续收取令牌附加的数据包。这个机制不是一个好的设计, 它导致了 RAM 占用大, “_token:”这个缓存不得不扩大到 12 字节, 以便容纳 HUB 广播来的最长数据包 (SYNC+PID+8bytes+CRC16)。令牌包和后附数据包之间还必须利用变量“__ucontr0[13]”进行关联(令牌包中的设备地址匹配时将“__ucontr0[13]”置“1”)。另一种思路是在收到令牌之后不退出中断, 充分利用 EOP 的 3bit 时间以及下一个 SYNC 字节的前 6bit (KJKJKJ) 还有 EOP 与 SYNC 之间的一组 J 状态 (最多 6.5bit) 尝试进行

CRC5 校验, 然后跳转回 CN 中断入口处的“__waitK:”继续收取数据包。EOP 与 SYNC 之间的间隔时间可以参考 USB2.0 Specification 第 168 页 7.1.18.1 节。

/*-----*/
/*- <8> 处理 DATA0 和 DATA1 令牌 -*/
/*-----不华丽的分割线-----*/

对“DATA0”和“DATA1”这两个数据包的处理不一样,“DATA1”只会附加在“OUT”令牌之后,而“DATA0”可能附加在“OUT”令牌之后,也可能附加在“SETUP”令牌之后,所以固件要区分这两种情况,以便在收取数据之后,正确报告这组数据是附加于“SETUP”令牌还是“OUT”令牌。这也是让令牌包和数据包分别触发 CN 中断所带来的不便。

下图 (figure11) 对比了“DATA0”和“DATA1”的不同之处,如果数据包是附加在“SETUP”令牌后的,w4[1-0]设为“01”,表示成功收取了“SETUP”令牌,w4[3-2]固定设为“01”,表示发送“ACK”;如果数据包是附加在“OUT”令牌后的,w4[1-0]设为“11”,表示成功收取了“OUT”令牌,w4[3-2]来自于“__ucontr0[3-2]”,可能是“ACK”也可能是“NAK”。



判断 DATA0 数据包是附加到 SETUP 令牌还是 OUT 令牌依靠“__uendpt0[0]”位,收到 SETUP 令牌时变量“__uendpt0[7-0]”总是被清除,参考图例 (figure09) 中的指令“clr.b __uendpt0”,所以在图例 (figure11) 中一个指令“btss __uendpt0, #0”就可以完成判断。

/*-----*/
/*- <9> 变量__uendpt0 和__ucontr0 详解 -*/
/*-----不华丽的分割线-----*/

固件中有两个重要的变量,“__uendpt0”和“__ucontr0”,其中“__uendpt0[1-0]”构成一个标识,用于指示出当前收到的令牌是哪一个。当主机或 HUB 发送“SETUP”令牌时,“__uendpt0[1-0]”会被清除,发送了 SETUP 令牌后附数据包时,固件会向主机返回“ACK”,然后设置“__uendpt0[1-0]”为“01”,同时设置“__uendpt0[2]”为“1”,表示“ACK”已经发出了,再设置“__uendpt0[3]”为“0”,表示“SETUP”令牌后附数据包的 PID 是“DATA0”,最后设置“__uendpt0[7-4]”指示从主机端接收的数据包中有效数据长度,通常固定是 8 字节长。

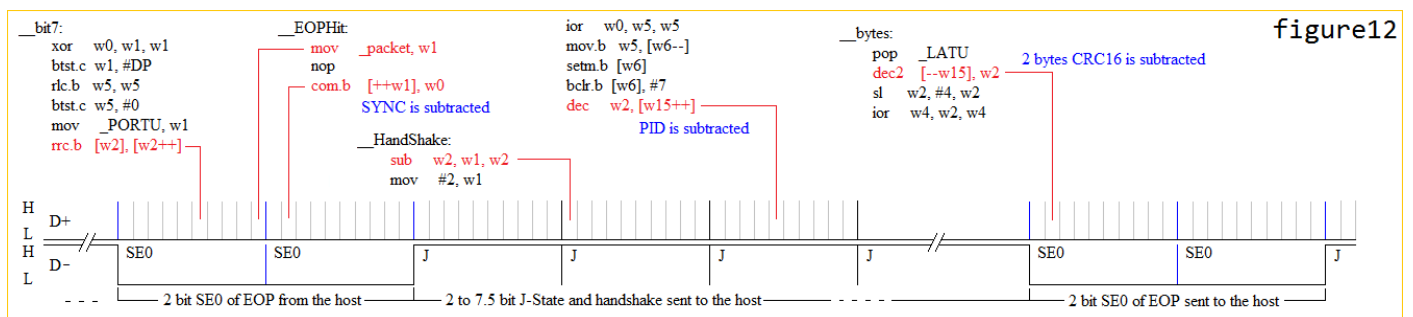
当主机或 HUB 发送“OUT”令牌及其数据包时,固件会根据“__uendpt0[3]”确定后续数据应该收入哪个缓存,收下“OUT”令牌之后“__uendpt0”是不做任何改变的。收下 OUT 令牌后附数据包之后则根据“__ucontr0[3-2]”的指示发送握手包。这两个 bit 在上电复位和“BUS RESET”期间初始化为“10”,默认向主机发送“NAK”,发送 NAK 之后“__uendpt0[7-0]”是不做改变的。固件如果准备好了获取数据包,可设置“__ucontr0[3-2]”为“01”,此后主

机重发“OUT”令牌及其数据包时，固件就会发送“ACK”握手并设置“__uendpt0[2-0]”为“111”，表明“OUT”令牌及数据接收了，并且向主机发送了“ACK”。固件利用“__uendpt0[3]”标识这个“OUT”令牌后附数据使用的PID是“DATA0”还是“DATA1”，如果是“DATA0”，固件设置“__uendpt0[3]”为“0”，是“DATA1”则设置“__uendpt0[3]”为“1”。而“__uendpt0[7-4]”则用于指示从主机端接收的数据包中有效数据长度，为0到8字节之间。向主机发送“ACK”之后，“__ucontr0[3-2]”重置为“10”，后续通信仍然向主机发送“NAK”。

当主机或HUB发送“IN”令牌时，固件会根据“__ucontr0[1-0]”的指示发送握手包。这两个bit在上电复位和“BUS RESET”期间初始化为“10”，默认向主机发送“NAK”，这时“__uendpt0”也是不会改变的。当需要发回给主机的数据准备好时，我们可以设置“__ucontr0[1-0]”为“01”，之后再收到主机发来的“IN”令牌时数据就会被发送。固件会根据“__ucontr0[12]”来决定使用“DATA0”还是“DATA1”。当“__ucontr0[12]”为“0”时使用“DATA1”，为“1”时使用“DATA0”，这与“__uendpt0[3]”相反。注意固件在收到“SETUP”令牌时会把“__ucontr0[12]”初始化为0，因此后续的第一个“IN”是使用“DATA1”的。主机在收到数据后会向设备发送“ACK”或者“NAK”握手，固件收到“ACK”后会翻转“__ucontr0[12]”并且将“__ucontr0[1-0]”重置为“10”，后续通信仍然向主机发送“NAK”。收到主机发来的“NAK”后会重新设置“__ucontr0[1-0]”为“01”，再次发送缓存中的数据。

处理“控制写”传输（SETUP+OUT+OUT.....+OUT+IN）时要依赖“__uendpt0[1-0]”中的令牌类型。具体来说收到“SETUP”令牌后固件设“__uendpt0[1-0]”为“00”。紧接着收到“SETUP”令牌后附的“DATA0”时，固件会依靠“__uendpt0[0]”这1个bit判断当前的令牌是否为“SETUP”。如果“__uendpt0[0]”是“0”，说明当前令牌是“SETUP”，固件向主机发送“ACK”并设置“__uendpt0[1-0]”为“01”。注意这之后固件收到第一个“OUT”令牌，后附数据一定是“DATA1”，固件为这个“DATA1”发送“ACK”之后就会将“__uendpt0[1-0]”进一步设为“11”。也就是说，固件收到“DATA0”这个PID时，“__uendpt0[1-0]”只可能是“00”或者“11”，不会有其它取值，所以可以仅凭“__uendpt0[0]”这一个bit判断令牌是否为“SETUP”。

对“__uendpt0[10,7-0]”的维护比较复杂，在收到“SETUP”令牌后会把“__uendpt0[7-0]”清除（见“__isSetup:”这部分代码）；收到主机发来的“DATA0”或“DATA1”时，如果是回复“ACK”，固件会清除或设置“__uendpt0[3]”（见“__isData1:”和“__siData0:”这部分代码）；在开始发送握手包时，如果是回复“ACK”，固件会清除“__uendpt0[7,4,2-0]”，保持“__uendpt0[3]”不变（见“__Sending:”这部分代码）；握手包发送结束之后，如果是回复了“ACK”，固件会根据实际数据传输情况设置“__uendpt0[10,7,4,2-0]”的值（见“__bytes:”这部分代码）。



上图（figure12）显示了如何计算主机发来的有效数据长度，代码分散在6处，数据接收完毕w2寄存器指向了尾部的CRC16之后（指令rrc.b[w2], [w2++]），缓冲区首部则取入w1寄存器（指令mov_packet, w1）。之后w1寄存器加1（指令com.b[++w1], w0），相当于去掉了SYNC字节，所以w2-w1的结果（指令sub w2, w1, w2）中只多了PID字节和2字节CRC16。固件用指令“dec w2, [w15++]”减去了PID字节并将结果入栈，在发出了握手包之后用指令“dec2 [--w15], w2”减去了2字节CRC16并出栈。至此w2寄存器中为接收到的有效数据长度。

/*-----*/
/*- <10> 数据发送和 bit-stuff 过程 -*/
/*-----不华丽的分割线-----*/

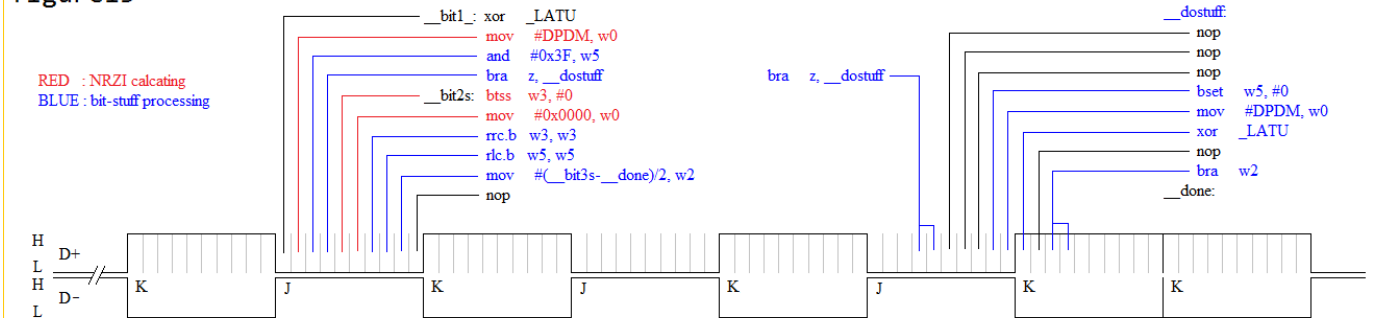
设备向主机发送数据时，有效数据的长度在“__ucontr0[7-4]”中设定，取值范围 0 到 8 字节。“__ucontr0[12]”用于指示本次数据传输使用“DATA0”还是“DATA1”，这一位在每次收到“SETUP”令牌时清 0，每次收到主机发来“ACK”时翻转。“__ucontr0[15-13]”为固件内部使用，“__ucontr0[11-8]”目前没有使用。

设备向主机发送的数据以反码形式保存在缓存“__datay:”之中，每一字节在发送时，bit1 至 bit5 采用的流程相同，在每个 bit 的第 1 机器周期设置 D+/D-的输出电平，之后是进行“bit-stuff”处理以及计算下一个 bit 的 NRZI 编码。固件使用“xor _LATU”指令从 D+/D-引脚发出信号，当 w0 寄存器中对应 D+/D-引脚的两个 bit 都是“1”时，D+/D-引脚的输出电平会发生切换。两 bit 都是“0”时则 D+/D-引脚输出电平不变，所以固件要根据下一个 bit 的值来计算 w0 寄存器的值。

处理“bit-stuff”的思路和接收数据时一样，利用 w5 寄存器追踪 bit 流中是否出现连续 6 个 1，若出现连续 6 个 1 则转到“__dostuff:”，向 w5 寄存器 bit0 位插入一个 0（反码），然后直接翻转 D+/D-。“__dostuff:”使用“bra w2”指令转回主发送流程，这样做有利于缩减代码尺寸，因此要求 w2 寄存器中预先置入正确的返回地址。

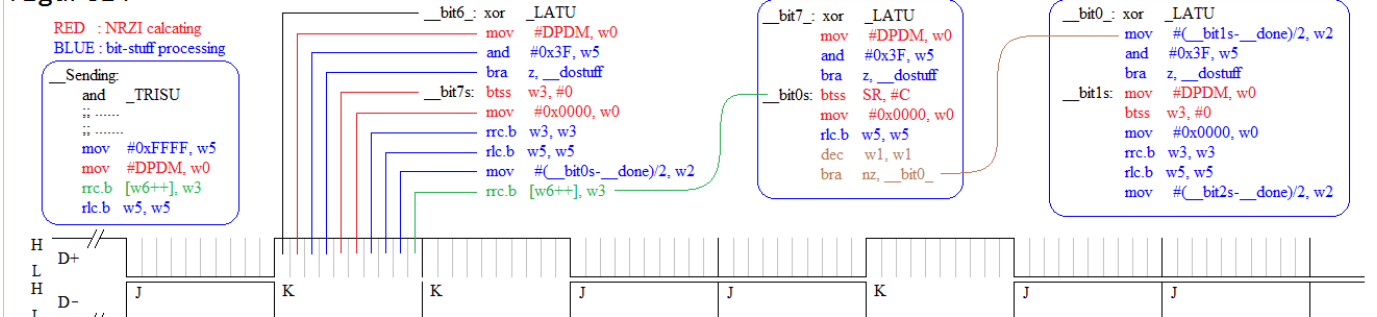
在图例（figure13）中可以看到我们使用指令“xor _LATU”从“_PORTU”口输出 D+/D-信号。寄存器 w0 的取值决定了 D+/D-是“翻转”还是“保持”。

figure13



发送 bit6 的流程与 bit1 至 bit5 相似，只不过利用了最后一个周期从缓存中读取下一个字节到 w3 寄存器中。固件总是使用“rrc.b [w6++], w3”指令读取数据，所以 w3 寄存器中只有 bit1 至 bit7，bit0 在 SR 寄存器的 C 标志位中。正因为此，在发送 bit7 的时候，需根据 SR.C 的值为下一字节的 bit0 准备 NRZI 编码，这要求在指令“rrc.b [w6++], w3”之后执行的所有指令不得破坏 SR 寄存器的 C 标志位，包括“__dostuff:”过程。发送 bit7 时本应该在 w2 寄存器中为下一字节的 bit0 准备“__dostuff:”过程的返回地址，但是固件需要 3 个机器周期构成一个循环，所以这一步没有做，在发送 bit0 时这一步需要补上。

figure14



/*-----*/
 /*- <11> 处理 IN 令牌和主机端握手 -*/
 /*-----不华丽的分割线-----*/

固件处理“IN”令牌的过程简单一些，只需注意在执行“bra __SendBytes”之前有一条指令“dec w1, w2”，w1

寄存器是发送给主机的数据总长，包括“SYNC/PID/CRC16”，w2 寄存器则减去了一个字节。这导致了数据发送完毕后，“__uendpt0[7-4]”会给出刚才发送给主机的有效数据长度。这不在“__uendpt0”这个变量的设计之中，是个额外获得的特性。我无意去除这个指令，你可以将其修改为“mov #0x0003, w2”，这样数据发送完毕后“__uendpt0[7-4]”就是 0 了。

处理主机或 HUB 发来的握手包（ACK/NAK/STALL）有点让人迷惑，CN 中断服务程序在向主机发送了数据包之后就退出了，主机或 HUB 发来的握手包会另外触发新的中断，而 HUB 又是以“广播”的方式发送数据，所以发给 HUB 上其它设备的握手包也会被接收到。由于握手包只中只有一个 PID 字节并无设备地址，所以固件必须确定收到的握手包是不是给本设备的。我采用的方法是判断“__uendpt0[2-0]”是否为“110”，是则说明刚刚有数据包发回给主机，这个握手包需要处理。

```
/*-----*/
/*- <12> sie.s 中的 API 函数 -*/
/*-----不华丽的分割线-----*/
```

使用 C 语言配合 sie.s 编写程序时，可以直接读写“__uendpt0”和“__ucontr0”两个变量，但我更推荐使用 sie.s 中提供的 API 函数。sie.s 共有 7 个 API 函数，以下列出其 C 语言原型并加以说明。

```
- BYTE _usbGetSetup(BYTE * setup); /* 取得主机发来的 SETUP 令牌后附的数据包 */
```

以一个指针变量为参数，该参数会放入 w0 寄存器传送，指向的缓冲区必须大于等于 8 字节。它允许指向奇数地址，允许为 NULL，当它为 NULL 时函数会直接返回一个 0。函数会先判断当前是否收到了“SETUP”令牌及其数据，若已收到就将 8 字节数据拷贝到缓冲区中，同时返回数据长度 8，若未收到则直接返回 0。因此应该在主任务循环中调用这个函数进行轮询。

```
- void _usbLoadData(BYTE * _data, BYTE length); /* 把当前需要返回给主机的数据发送出去 */
```

指针变量“_data”指向待发送的数据缓冲区，该参数会放入 w0 寄存器传送。它允许指向奇数地址，允许为 NULL。字节变量“length”指出需要发送的有效数据长度（字节数），该参数会放入 w1 寄存器传送。它必须小于或等于 8 字节，允许为 0。当“_data”为 NULL 并且“length”为 0 时，本函数会向主机返回一个“空包”，具体是“DATA0”还是“DATA1”由之前的发送序列决定。在收到主机发来的“ACK”握手包之前，本函数不会返回，本函数也没有设计超时退出机制。

```
- BYTE _usbReadData(BYTE * _data, BYTE length); /* 取得主机发来的 OUT 令牌后附的数据包 */
```

指针变量“_data”指向收取数据的缓冲区，该参数会放入 w0 寄存器传送。它允许指向奇数地址，允许为 NULL，但它为 NULL 时字节变量“length”必须是 0，否则函数会直接返回-1（0xFF）。字节变量“length”指出接收缓冲区的长度（字节数），该参数会放入 w1 寄存器传送。当“length”为 0 时，本函数会等待主机发来的一个空包，不论是“DATA0”还是“DATA1”。当“_data”不为 NULL 时，本函数对比主机发来的数据长度和“length”的值，按最小值向缓冲区填充数据，未填充部分会保留原值。

```
- void _usbSendZLP(void); /* 以“DATA1”为 PID 向主机发送一个空包 */
```

这个函数专用于“控制写”传输最后的“Status”阶段。如果不特别指定以“DATA1”为 PID 向主机发送空包，则需调用“_usbLoadData(NULL, 0)”。

```
- void _usbWaitZLP(void); /* 等待主机向设备发送一个空包 */
```

本函数等同于“_usbReadData(NULL, 0)”，可用于“控制读”传输最后的“Status”阶段。需注意本函数并不检查主机发来的空包是否由“DATA1”引导的。

```
- void _usbSetAddress(BYTE a); /* 收到主机分配来的设备地址后调用本函数 */
```

字节参数“a”就是主机发来的新设备地址，该参数会放入 w0 寄存器传送。本函数会先调用“_usbSendZLP()”函数完成“控制写”传输最后的“Status”阶段，然后将新设备地址保存到“_addr:”变量中。

```
- void _usbSetConfig(BYTE c); /* 收到主机发来的 configuration 之后调用本函数 */
```

字节参数“c”就是主机发来的 configuration，该参数会放入 w0 寄存器传送。本函数会将此参数保存于变量“_conf:”，然后调用“_usbSendZLP()”函数完成“控制写”传输最后的“Status”阶段。

```
/*-----*/
/*- <13> CRC5 和 CRC16 算法 -*/
/*-----不华丽的分割线-----*/
```

我没有对 TOKEN 包中的 CRC5 进行校验，但我找到了一个快速的 CRC5 校验算法，你可以试试把这篇论文中的 AVR 汇编代码改成 dsPIC33 的汇编。论文网址：<https://www.michael-joost.de/crc5check.pdf>。另有一个比较常规的算法，http://janaxelson.com/files/usb_crc.c。

CRC16 算法来自于 Microchip 论坛上的一个帖子，网址：<https://www.microchip.com/forums/m603309.aspx>。用于 USB 通信的 CRC16 算法同时也被用于其它一些数据通信协议中，比如贴中提到的“Modbus”。我找到了一个更为快速的 CRC16 算法，https://www.modbustools.com/modbus_crc16.html，但我并没有验证这个源码。

CRC-16 (Modubs) 0x8005 for PIC32

Author
Essentials Only
Full Version
Post

flamewatcher

New Member

★

Total Posts : 8

Reward points : 0

Joined: 5/13/2011

Location: 0

Status: **offline**

Saturday, September 24, 2011 6:21 AM (permalink)

CRC-16 (Modubs) 0x8005 for PIC32

★★★★★
0

I am having of trouble implementing the standard ANSI CRC-16 using the Pic32 DMA crc controller.
The following code works in software:

```

UInt16 ComputeCRC16(const UInt8 * buf, int len)
{
    int j;
    UInt8 i;

    UInt16 crc = 0xffff; //seed
    for ( j = 0; j < len; j++)
    {
        UInt8 b = buf[j];
        for ( i = 0; i < 8; i++)
        {
            crc = ((b ^ (UInt8)crc) & 1) ? ((crc >> 1) ^ 0xA001) : (crc >> 1);
            b >>= 1;
        }
    }
    return crc;
}

```

我把 flamewatcher 的源码做了一些修改（见下方源码），以适应 16bit 的 dsPIC33 芯片，同时将这个源码转换为汇编语言编码。sie.s 中的“__CRC16:”函数在计算 CRC 的时候，会同时将数据拷贝到“__datay:”缓存之中。

```
unsigned short _ComputeCRC16(const unsigned short * buf, int len)
{
```

```

int j;
unsigned char i;

unsigned short crc = 0xffff; //seed

for (j = 0; j < len; j++)
{
    unsigned short b = buf[j];
    for (i = 0; i < 16; i++)
    {
        crc = ((b ^ (unsigned short)crc) & 1) ? ((crc >> 1) ^ 0xA001) : (crc >> 1);
        b >>= 1;
    }
}
return crc;
}

```

```

/*-----*/
/*-  <14>  usb.c 相关提示  -*/
/*-----不华丽的分割线-----*/

```

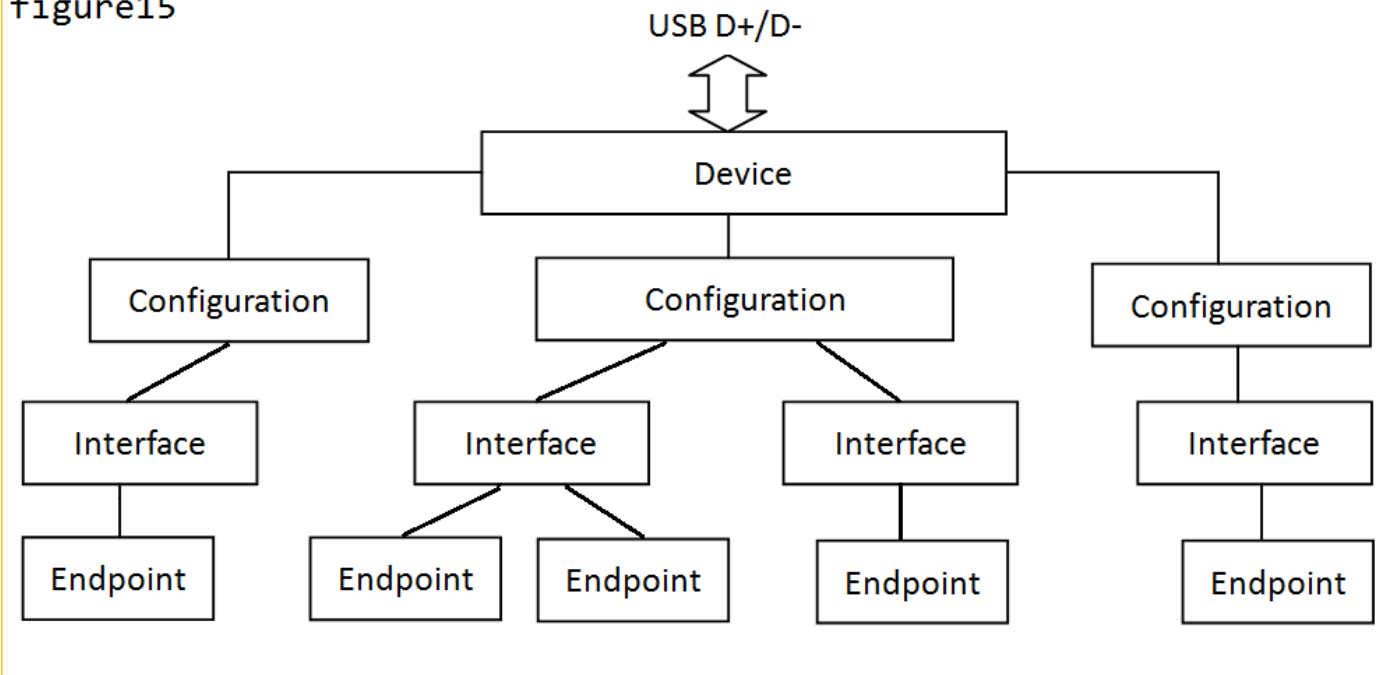
USB 总线可以经由 HUB 同时连接多个设备进行通信，不同类型的设备对通信方式有不同的要求。例如键盘，每次击键传输的数据量非常小，但每次击键都必须传给主机。由于 USB 数据传输总是由主机通过发送一个令牌起始，设备不能主动发起数据传输，所以键盘要求主机定时查询击键事件。再如 USB 声卡，若以 44.1KHz 采样率播放 16bit 立体声音乐，每秒钟需要 44100x2x2 个字节数据，数据量不算大，但必须在每一秒钟内都发送足量数据，亏欠数据会导致音乐出现卡顿。又如 U 盘，它要求相对简单，就是在尽可能短的时间内发送尽可能多的数据。由于 USB 总线数据传输带宽是有限的，所以必须有某种机制为不同类型的设备合理分配带宽。参考 USB2.0 Specification 第 20 页 4.7 节和第 34 页 5.4 节至 5.8 节。

USB2.0 Specification 规定了 4 种类型的数据传输方式，第一为“批量传输”，USB 总线会在一个帧之内安排尽可能多的“批量传输”，U 盘类设备应该支持这种传输方式。第二为“同步传输”，USB 总线确保此类传输在各个帧之间均匀分布，单位时间内的数据量有保证，USB 声卡类设备应该支持这种传输方式。第三为“中断传输”，USB 总线会以一定时间间隔连续产生此种数据传输，USB 键鼠类设备应该支持这种传输方式。第四为“控制传输”，此类传输占用很少量带宽，也没有时间和数据足量的保证，仅在主机需要对设备进行一些管理时应用。参考 USB2.0 Specification 第 20 页 4.7 节和第 221 页 8.5.2 节至 8.5.5 节。

设备通过向主机汇报自身具有的“端点”的方式来声明所支持的数据传输方式。“端点”由一个 4bit 的数字代表，可称之为“端点地址”，而“端点”的本质则是一个数据存储缓冲区。假设一个设备想通过“批量传输”的方式从主机接收数据，它可以开设一个 64 字节长的缓冲区，然后向主机报告一个端点地址，并报告这个端点地址支持“批量传输”，具有 64 字节缓存。主机在向这个设备发送数据时，会把“端点地址”放入令牌中，后附的数据包则容纳最多 64 字节有效数据。需注意任何 USB 设备都至少具有一个地址为 0 的端点，此端点仅支持“控制传输”，其它端点是“可选”的。参考 USB2.0 Specification 第 33 页 5.3.1 节。

下图（figure15）显示出一个设备可以同时复合多个功能，例如一个键盘上同时带有一个触摸板，它就是一个键盘加鼠标的复合设备，它需要有多组“中断端点”并将其分配给键盘和鼠标，如键盘使用端点地址 2 和 3，鼠标使用端点地址 4 和 5，端点 2 和 3 构成了键盘“介面”，端点 4 和 5 构成了鼠标“介面”。多组介面还可以继续分组，每一组称为一个“配置”，主机会在需要的时候激活某个配置，然后通过端点与其下的某个介面进行通信。所有的配置组合在一起形成一个完整的“设备”。

figure15



“设备/配置/介面/端点”都使用名为“描述符”的数据结构加以表达，结构中各成员有明确的意义。一个设备具有一个“设备描述符”，其下包含的各个配置都有对应的“配置描述符”，每个介面都有对应的“介面描述符”，每一个端点也有对应的“端点描述符”。当设备插入主机或 HUB 时，主机端会首先获取“设备描述符”，然后获取“配置+介面+端点”描述符，从而了解该如何与各个介面进行数据传输。

“设备描述符”与“介面描述符”之中有一个名为“类别代码”的成员，这是因为有一部分 USB 设备已经有了明确的功能设计，键盘鼠标，U 盘，音频视频设备，网络通信设备，打印机相机，智能卡读写器等等。设计此类设备时可以在描述符中直接声明设备类别，主机就会根据设备类别明确知道如何与设备进行通信，否则就需要设计者自己为主机端编写通信软件。

需注意“设备描述符”中的类别代码常设为 0，此时要根据“介面描述符”中的类别代码区分设备类，必须在“设备描述符”中指定的设备类不多。设备类代码的定义可以参考 <https://www.usb.org/defined-class-codes>。设备类也有一个描述符，这个类描述符紧接在介面描述符之后，插在端点描述符之前。参考 HID1.11 Specification 第 7 页 4.1 节至 4.3 节。

我这个项目是一个“人机交互设备”，此类设备下包括一些“子类”，隶属于“子类”之下的设备还具有不同的“协议类型”，如鼠标键盘等。但我没有选择这些已经明确定义的子类和协议，因此这是一个自定义的人机交互设备。主机端仅对此类设备提供基本的数据通信支持，但主机并不关心这些数据有什么意义。

HID 设备与主机间通信采用收发“报告”的方式，每条报告中包含一组数据成员，设备定义出数据元的长度（bit 数），以及这个数据元的用途，同时声明这个数据元的取值范围，还要声明一条报告中可以包含几个数据元等。定义及声明这些内容也是采用“描述符”这种数据结构。参考 HID1.11 Specification 第 14 页 5.2 节至 5.6 节和第 23 页 6.2.2 节。

当设备插入主机或 HUB 时，初始的“设备地址”是 0，主机使用初始设备地址通过一组经由“端点 0”的“控制传输”来获取各个描述符。具体来说，主机首先发出一个“SETUP”令牌及 8 字节数据，此 8 字节数据通常是“0x80 0x06 0x00 0x01 0x00 0x00 0x40 0x00”（参考 USB2.0 Specification 第 248 页 9.3 节至 9.4 节。），前 2 字节指出现在需要取描述符，下两个字节指出是取设备描述符，最后两字节是期望读到多少字节数据。示例中最后两字节是“0x40 0x00”，表示数据长度希望是“0x0040”字节。在“SETUP”之后是“IN”令牌，由于“SETUP”令牌所附的 8 字节数据以“DATA0”发出，所以第一个“IN”令牌一定是使用“DATA1”传输。设备根据“端点 0”的缓存尺寸

分块发送描述符，在设备发送完全部数据之前，主机会重复发出多个“IN”令牌，后附数据包以“DATA1/DATA0”交替传输，直至设备将所有数据发完。数据发送完毕后主机会发送一个“OUT”令牌，后附数据包固定使用“DATA1”且有效数据长度为 0，这个“空包”用于告知设备这一次数据交互到此结束。

这里的关键问题是主机如何知道设备已经发送完所有数据了？其策略如下：

- a. 主机给定一个最大的数据长度值，如上所述的“0x0040”；
- b. 如果设备中等待返回的数据长度比主机给定的这个最大值还要长，则设备只能发出主机给定长度那么多的数据，主机在收取了足量数据后会自然结束这次数据交互；
- c. 如果设备中等待返回的数据长度比主机给定的最大值短，则设备发出全部数据。由于要按端点 0 的缓存尺寸分组发送，尾部的最后一组数据长度如果比端点 0 的缓存尺寸短，则这个“短包”会告知主机数据传输结束了。若尾部最后一组数据长度和端点 0 的缓存尺寸相等，则设备需要额外再发出一个“空包”告知主机数据传输结束了。

上述这一组由一个“SETUP”加一组“IN”和一个以“DATA1”传送 0 字节数据的“OUT”所构成的数据传输过程称为“控制读”序列。类似的还有“控制写”序列，即以“SETUP”加一组“OUT”和一个以“DATA1”传送 0 字节数据的“IN”所构成。这个序列很简单，因为主机知道待发送的数据有多长，所以“OUT”令牌的数量是确定的。设备收取了主机发出的所有“OUT”之后，需要通过“DATA1”向主机发送一个“空包”完结这一次数据交互。参考 USB2.0 Specification 第 225 页 8.5.3 节。

需注意通过一系列“IN”或“OUT”令牌传输数据的过程是“可选”的，有些时候并没有数据需要传输。比如当主机为设备分配好一个特定地址之后，主机会使用初始设备地址发送一个“SETUP”令牌，后附 8 字节数据“0x00 0x05 0x03 0x00 0x00 0x00 0x00 0x00”。其中前 2 字节指出现在要为设备分配一个新地址，第 3 个字节“0x03”就是这个地址的数值，最后两字节数据长度是“0x0000”，这是不带数据的，所以设备只需经由“DATA1”返回一个“空包”完结数据交互即可。这里有一个小花招，主机为获得最后这个“空包”要发出一个“IN”令牌，其中的设备地址仍然是初始地址 0，设备必须先收取这个“IN”令牌并返回空包，待主机对这个空包发了“ACK”握手之后才能把设备地址改成新的值。

```
/*-----*/
/*- <15>  usb.c 中的 API 函数                                -*/
/*-----不华丽的分割线-----*/
```

源码文件 usb.c 中包含了一组“描述符”和 4 个函数。HID_ReportDescriptor 中启用了全部 3 个报告，这 3 个报告都是 64 字节长。USB_DeviceDescriptor 指定了“厂商名称”和“设备名称”两个字符串索引，在某些操作系统上当设备连接到主机后会显示出这些名称。

```
- void USB_vInit(void); /* 上电初始化 */
```

此函数目前为空，没有实质性的处理。

```
- BYTE USB_bRxRequest(void* Request); /* 接收 8 字节的 SETUP 数据包，处理部分标准请求 */
```

指针型参数 Request 不允许为 NULL。这个函数仅处理 3 个标准请求：取描述符（Request[1]==0x06），设置设备地址（Request[1]==0x05）和激活配置（Request[0]==0x00 && Request[1]==0x09）。对于 HID 规范制定的两个请求：设置报告（Request[0]==0x21 && Request[1]==0x09）和获取报告（Request[0]==0xA1 && Request[1]==0x01），本函数会返回“USB_REQ_SETUP”，告知调用者自己处理这两个请求。本函数还额外对 HID 的“SET IDLE”做了支持，同样是向调用者返回“USB_REQ_SETUP”，这应该不是必须的，但某些 LINUX 操作系统可能会发出这个请求。

```
- BYTE USB_bGetCtrlData(BYTE * dat, WORD siz, WORD exLength); /* 取得 SETUP 之后的 OUT 数据包 */
```

指针型参数 `dat` 指向数据缓存，不允许为 `NULL`，参数 `siz` 指示 `dat` 缓存所能容纳的数据字节数，参数 `exLength` 是主机在 `SETUP` 数据包中声明的数据长度，即在 `USB_bRxRequest` 函数中收到的“`Request[6]/ Request[7]`”。当 `siz` 的值大于等于 `exLength` 时，本函数会处理所有 `OUT` 数据包并且向主机发回空包结束数据交互。当 `siz` 的值小于 `exLength` 时，本函数不会造成缓存溢出，但不能正确接收所有的 `OUT` 数据包。本函数的返回值为收到的全部数据总字节数。

```
- BYTE USB_bSendCtrlData(BYTE* dat, WORD siz, WORD exLength); /* 向主机发送数据 */
```

指针型参数 `dat` 指向数据缓存，可以为 `NULL`，为 `NULL` 时要求 `siz` 和 `exLength` 必须都为 0。参数 `siz` 指示待发送的数据字节数，可以为 0，为 0 时要求 `exLength` 必须为 0。参数 `exLength` 是主机在 `SETUP` 数据包中声明的数据长度。当 `siz` 和 `exLength` 同时为 0 时，本函数会经由 `DATA1` 向主机发送一个空包。当 `siz` 小于 `exLength` 且恰好是“`MAX_PACKET_SIZE`”的整数倍时，本函数可以自动多发一个空包做为数据尾块。在所有数据发送完毕后，本函数会等待主机经由 `DATA1` 发来的空包。本函数的返回值是数据尾块的字节数。

```
/*-----*/
/*- <16> hid.c 相关提示 -*/
/*-----不华丽的分割线-----*/
```

在 `WINDOWS` 平台上，向 `HID` 设备发送报告有两种方式，其一是调用“`HidD_SetFeature`”函数，这个函数可以通过“控制写”传输方式向 `HID` 设备发送“`Feature Report`”。其二是调用“`HidD_SetOutputReport`”函数，这个函数可以通过“中断传输”向 `HID` 设备发送“`Output Report`”。如果设备没有中断端点，则这个函数改用“控制写”传输方式发送报告。执行“`HidD_SetFeature`”函数之后设备收到的 8 字节 `SETUP` 数据包是“`0x21 0x09 0x00 0x03 0x00 0x00 0x40 0x00`”。调用“`HidD_SetOutputReport`”函数之后设备收到的 8 字节 `SETUP` 数据包是“`0x21 0x09 0x00 0x02 0x00 0x00 0x40 0x00`”。参考 <https://docs.microsoft.com/en-us/windows-hardware/drivers/ddi/hidsdi/>。

从 `HID` 设备收取报告也有两种方式，与“`HidD_SetFeature`”函数对应的是“`HidD_GetFeature`”函数，执行此函数后设备收到的 8 字节 `SETUP` 数据包是“`0xA1 0x01 0x00 0x03 0x00 0x00 0x40 0x00`”。与“`HidD_SetOutputReport`”函数对应的是“`HidD_GetInputReport`”函数，这个函数也是通过“中断传输”从 `HID` 设备读取数据的，在设备没有中断端点时这个函数通过“控制读”方式收取报告，此时设备收到的 8 字节 `SETUP` 数据包是“`0xA1 0x01 0x00 0x01 0x00 0x00 0x40 0x00`”。参考 `HID1.11 Specification` 第 50 页 7.2 节。

```
/*-----*/
/*- <17> hid.c 中的 API 函数 -*/
/*-----不华丽的分割线-----*/
```

如果主机调用“`HidD_SetFeature`”函数给设备发送数据，这些数据会逐字节取反，然后等待主机取回。主机发来的数据前 2 字节还会被当成一个循环计数值，固件在完成这个循环之后才会处理主机调用“`HidD_GetFeature`”函数发来的 `SETUP` 数据。所以当主机向设备发送随机数时，固件会随机延迟一小段时间才会返回结果。这就模拟了“花费一小段时间执行了其它任务，然后继续处理数据通信”这样一个细节。如果主机调用“`HidD_SetOutputReport`”函数给设备发送数据，则这些数据会原样返回，固件也不会返回结果之前做其它工作。

源码 `hid.c` 中包括一个全局变量 `State`，收到主机发来的数据后它的值设为“`COMMAND`”，固件准备好数据待返回时设置这个变量为“`RESPONSE`”。如果在此变量设为“`RESPONSE`”之前主机调用“`HidD_GetFeature`”函数读取数据，固件会直接返回一个空包。当然也可以返回其它一些有特殊意义的数据，这可以协助主机端的应用软件形成一个简单的轮询机制，避免固件执行长耗时任务时造成主机端出现超时。但是很遗憾，由于 `sie.s` 不能在收到主机发来的 `SETUP` 令牌使用某种中断机制与上层软件交互，所以主机端发来的 `SETUP` 请求并不能实时处理。

```
- void HID_vInit(BYTE Mode); /* 上电或复位后初始化 */
```

参数 `Mode` 可以用于指示复位的原因（上电复位/软件复位/看门狗复位等过），目前这个参数没有使用。

```
- BYTE HID_bRxRequest(void *Req, WORD siz); /* 处理 HID1.11 Specification 中定义的部分 SETUP 请求 */
```

指针型参数 `Req` 指向一个数据缓存，参数 `siz` 为缓存尺寸。当 `Req` 不为 `NULL` 且 `siz` 等于 2 时，本函数会将主机发来的前 2 字节放入这个缓存中。返回值为 1 时表示调用者所需的数据已放入 `Req` 缓存中，返回 0 时表示并未通过缓存 `Req` 给调用者返回有意义的数据。

```
- BYTE HID_bTxResult(void *dat, WORD siz); /* 准备将一组数据返回给主机 */
```

指针型参数 `dat` 指向待返回的数据，参数 `siz` 为待返回的数据长度。由于和主机端的数据传输限制在 `hid.c` 之内，本函数仅设置全局变量 `State` 的值并返回了 1。

```
/*-----*/  
/*- <18>  main.c 说明                                -*/  
/*-----不华丽的分割线-----*/
```

源码 `main.c` 非常简单，函数“`setup()`”在上电复位后被 `sie.s` 调用，主任务循环 `loop` 函数中安排了一个 `unsigned short` 型变量 `Req`，“`loop()`”函数调用 `HID_bRxRequest` 函数从主机接收 2 字节数据并赋给此变量，然后将此变量做为计数器执行一个循环。接收数据之前 LED 灯是点亮的，进入循环之后 LED 灯保持熄灭状态，通过 LED 闪烁状态大致可知循环时间长短。

已知的 BUG

当这个设备连接到一个 USB HUB 上时，主机向设备发送数据时会偶尔出错，出错的概率与发送的数据之间有些关联。具体来说，如果主机每次取 64 字节随机数发送给设备，则发送出错的概率就很低；如果主机每次都发送 64 字节取值相同的数据，比如 64 字节都是 0xFF，出错的概率就会比较高。这个 BUG 仅当设备连接到 USB HUB 上出现，设备直接连接到主机上时未观察到这个 BUG。