STM32 USB 设备开发指南

本书正在编写中，有任何问题或建议可以在这里反馈：faq.tusb.org

或是通过电子邮件方式联系作者：admin@xtoolbox.org

目录

1 前言

1.1 本文的由来

ST 公司为 STM32 系列单片机的 USB 模块编写了标准库和 HAL 库，标准库库只支持较早的一些芯片并且不再维护；HAL 库支持全系列芯片，HAL 库可以和 CubeMX 配合，自动生成初始化代码。HAL 库中的 USB 部分，为了适应各种应用场景，对 USB 模块的核心操作进行了层层封装。笔者在一个项目中，能够使用的代码大小受限，因此就在寻找是否有更加简化的 USB 库，功能不一定要很多，但是能够满足需求。

在 github 有两个 STM32 的简化版 USB 库。一个是基于 STM32 的简化版 HID Bootloader 项目，一个是 STM32 的 USB 库项目。前者是做 bootloader 的，侧重于代码空间的优化；后者目标是做一个开源 USB 协议栈，侧重于协议的完整性。这两个项目还有一些缺陷：简化版 Bootloader 项目没有实现更多的 USB 设备类，而后一个项目不能同时支持有多个 USB 模块的芯片。为了解决这两个问题，笔者实现了一个基于 STM32 的简化版协议栈 TeenyUSB。本文主要基于 TeenyUSB 来介绍 USB 设备的实际开发流程。文中部分内容会同时对比分析 ST 官方 HAL 库和 TeenyUSB，让 ST 官方 HAL 库的使用者也能对 HAL 库的实现有所了解。TeenyUSB 的全部源代码、例程和相关工具可以从 www.tusb.org 获得。

1.2 USB 简介

USB 是通用串行总线的缩写，USB 总线具有连接简单，使用范围广的特点。目前市面上的大部分单片机都集成了 USB 模块，可以与上位机方便的进行通讯。这里主要介绍 ST 公司的 STM32 系列单片机的 USB 模块。各个厂家的 USB 模块基本功能都是类似的，熟悉了一种之后也更容易触类旁通。

在嵌入式开发中，USB 一直是较为复杂的内容。一方面，在单片机内部，USB 模块本身就很复杂，涉及到的寄存器很多；另一方面，USB 除了单片机开发外，还涉及到 PC 端主机软件的开发，PC 端的软件由于与硬件联系紧密，调试也比较复杂。TeenyUSB 提供嵌入式协议栈的同时也提供了 PC 端的测试软件，便于在开发过程进行测试。

本文主要参考了 STM32 系列芯片参考手册和《USB2.0 规格书》，在提到一些概念和名词时，为了避免理解上的误差，会分别用中文和参考资料中的英文原文来进行描述。

1.3 阅读指南

后面的 2-5 章是在介绍一些 USB 设备的基础知识，写的比较啰嗦，希望先把代码跑起来的读者，可以直接跳到第 6 章 USB 设备开发，并配合 www.tusb.org 的代码进行实际操作。

第 2 章主要介绍了一些 USB 中的基础概念，介绍时结合了 TeenyUSB 的代码进行说明，这章对于 USB 基础概念熟悉的读者可以跳过。对于熟悉概念却又不知道代码该如何下手的读者可以看一下这章，或许能从中找到概念到代码的转化方法。

第 3 章介绍 USB 标准描述符，介绍时通过描述符生成工具 TeenyDT 进行了详细说明，对于熟悉标准描述的读者可以跳过。对于熟悉描述符但是又觉得描述符编写起来很繁琐的人，可以只看 3.10 描述符工具 TeenyDT 这一节的内容，或许能够找到简化描述符编写工作的办法。

第 4 章介绍 STM32 的 USB FS 模块以及对应的代码实现，USB FS 模块主要使用在 STM32F0，STM32F1 系列，STM32F3 系列芯片中。FS 模块在 ST 官方也叫做 USB 模块，ST 的 USB 模块有两个版本，USB 和 USB+，这两个版本差别不大，合在一起介绍。

第 5 章介绍 STM32 的 OTG 模块设备部分功能，以及对应的代码实现。OTG 模块主要使用在 STM32F2、STM32F4、STM32F7 系列芯片中。OTG 分为 FS 和 HS 两个版本，HS 又分为内置 phy 和外置 phy 的版本。不同 OTG 版本基本操作都类似，合在一起介绍。

第 6 章介绍实际的 USB 设备开发，主要介绍自定义 USB 设备、CDC 串口设备、MSC 大容量存储设备和自定义 HID 设备这四类设备的开发与使用。

第 7 章介绍 STM32 的 OTG 模块主机功能，这部分内容非常简略，只对主机部分的框架进行了梳理，没有深入到一些细节。

第 8 章介绍一个简单的 USB 主机实现，包含最简单的键盘鼠标以及 HUB 功能。主机相关的内容都很简略，介绍主机主要目的是换一个角度来看待设备的实现。

第 9 章介绍一些与 TeenyUSB 项目相关的外部资源，如测试软件用到的 Qt 库和 libusb 库，驱动 INF 文件生成用到的 libwdi 项目，Windows 上的通用 USB 设备驱动 WinUSB。

2 USB 基础概念

在介绍 STM32 的 USB 模块之前，这里先介绍一下 USB 系统中的一些基础概念。这些基本概念有助于我们去理解 USB 模块的设计。本文没有完全覆盖 USB 的方方面面，只介绍一些与 USB 模块操作关联密切的内容。在介绍这些基础概念时，笔者会根据自己的理解以及在代码中的具体实现来对相关概念进行详细说明。完整的 USB 内容可以参阅参考文档《Universal Serial Bus Specification 2.0》，后文中的《USB 2.0 规格书》也是指这份参考资料。

本文不涉及 USB 底层信号，主要从芯片使用的角度来说明 USB 设备的设计。底层信号的处理由芯片 USB 模块内部进行处理，用户层一般不会去操作具体的底层信号。底层信号的详细说明在《USB 2.0 规格书》的《Chapter 8 Protocol Layer》一章。

2.1 物理连接

USB 设备通过 USB 线缆与主机连接，对于低速（LS）、全速（FS）、高速（HS）设备，D+和 D-是数据线，对于 OTG 设备，还有一根 ID 线用来标识目前设备是工作在主机模式还是从机模式。因此对于 STM32 的 USB FS 模块需要 D+和 D-两个信号线。而 OTG 模块除了 D+和 D-信号外，如果需要支持主从一体的双角色设备（Dual Role Device，DRD），还需要一根 ID 线。本文不涉及 Super Speed 的 3.0 设备。

2.1.1 D+/D-信号

D+/D-信号也叫做 DP/DM（Data Plus/Data Minus）信号，USB 主机通过 D+/D-信号上的上来电阻来识别 USB 设备的速度，D+有 1.5K 上拉电阻时，识别为 FS/HS 设备。D-有 1.5K 上拉电阻时，识别为 LS 设备。当没有上拉电阻时，主机认为设备处于断开状态。因此可以通过控制上拉电阻的连接与否来控制设备的连接状态。

有些芯片内置了 1.5K 上拉电阻，可以通过寄存器控制上拉电阻的状态，实现软件控制连接状态。高速（HS）设备在启动时处于全速（FS）状态，保持 D+上拉，完成高速设备枚举配置后，断开上拉电阻，进入电流驱动的高速模式。

有些芯片没有内置 1.5K 上拉电阻，需要外接上拉电阻。如果需要支持软件控制连接状态，还需要通过一个额外的引脚来控制上拉电阻的通断。当关闭了 STM32 的 USB 模块的时钟后，USB 的 D+/D-信号引脚会工作在 GPIO 模式。USB 时钟关闭后，将 D+引脚设置为输出模式，并将 D+引脚设置为低电平。这时 USB 主机端检测到 D+/D-都为低电平，会认为设备断开。可以通过将 D+设置为 IO 输出并置低的方法，在没有内置上拉电阻的芯片上模拟出断开连接的效果，这样可以节省一个控制上拉电阻状态的 IO。

下面是 TeenyUSB 协议栈在 F0 芯片上的 D+引脚处理代码：

```
void tusb_open_device(tusb_device_t* dev){
...
#ifdef INTERNAL_PULLUP
  // USE the interal pull up resistor
  GetUSB(dev)->BCDR|=USB_BCDR_DPPU;
#else
  RCC->AHBENR |= RCC_AHBENR_GPIOAEN;
```

```
  // PA12 = In float
  GPIOA->PUPDR &= ~GPIO_PUPDR_PUPDR12;
  GPIOA->MODER &= ~GPIO_MODER_MODER12;
#endif
  ...
}

void tusb_close_device(tusb_device_t* dev){
#ifdef INTERNAL_PULLUP
  // disable interal pull up resistor
  GetUSB(dev)->BCDR&=~USB_BCDR_DPPU;
#else
  // PA12 = PushPull = 0
  GPIOA->OTYPER |= GPIO_OTYPER_OT_12;
  GPIOA->MODER &= ~GPIO_MODER_MODER12;
  GPIOA->MODER |= GPIO_MODER_MODER12_0;
  GPIOA->OSPEEDR |= GPIO_OSPEEDR_OSPEEDR12;
  GPIOA->BRR = GPIO_BRR_BR_12;
 #endif
 ...
 }
```

如果使用芯片内部的上拉电阻，只需要控制上拉电阻的状态位即可；如果没有使用内置的上拉电阻则通过控制 IO 信号来模拟出通断效果。模拟连接时，将 D+引脚即 PA12 脚设置位输入状态，这时候外置的上拉电阻会将引脚信号拉高；再打开 USB 时钟，这时 USB 模块会接管引脚。模拟断开时，先将 USB 时钟关闭，D+脚恢复为 GPIO 模式；再将 D+配置成输出模式并置为低电平，这时 USB 主机检测到 D+为低电平，会认为设备断开。

2.1.2 ID 信号

在 OTG 设备中多了一个 ID 信号，OTG 设备通过 ID 信号来切换工作模式。当 ID 不连接浮空时，工作在 B 模式，即设备模式。当 ID 接地时，工作在 A 模式，即主机模式。ID 的连接方式可以通过 USB 电缆来确定。当电缆的 ID 接地时，USB 设备工作在 A 模式，因此这种 ID 接地的电缆又叫做 A 电缆。同样地还有 B 电缆。如果使用 type-c 的连接线，ID 信号连接在 type-c 的 CC1/CC2 引脚上，下拉到地时，设备工作在 B 模式，浮空时工作在 A 模式。

2.1.2.1 **主机协商协议** Host Negotiation Protocol （HNP）

接了 A 电缆的 OTG 设备一定工作在主机模式吗？OTG 设备都可以工作在主机和从机模式，如果他们都支持主机协商协议，可以在连接后通过设置特性请求来切换主机。具体做法是 B 设备在枚举阶段上报一个 OTG 描述符给 A 设备。A 设备根据 OTG 描述符中的内容，得知 B 设备是否支持 HNP。如果 B 设备支持，当 A 设备不需要主动控制总线时，发送一个设置特性（Set Feature）请求，切换 B 设备工作模式。A 设备发送完请求后，释放总线，退出主机模式，B 设备获得总线控制权后等待 A 设备以从机的模式连接。A 设备连接后，开始

交换角色后的枚举过程。

对于 OTG 设备而言，A 模式应该说是初始工作在主机的模式，B 模式是初始工作在设备的模式。经过初始状态之后，主机设备可以根据需要互相转换，根据实际需要进行灵活的变换（On The Go）。

2.2 USB 的基本操作

本节介绍一些和芯片 USB 模块相关的 USB 基本操作。

2.2.1 热插拔与复位

当设备插入到 USB 主机的端口时，主机会对通过 USB 总线对设备的 USB 模块进行复位，复位后的设备有以下特点：

1. 响应默认地址上的数据，默认地址即 0 地址
2. 设备处于未配置（Not Configured）状态
3. 设备不能挂起（Not Suspended）

设备收到来自总线的复位信号后，需要将自己的地址设置为 0，清除配置状态，清除挂起状态。配置设备的操作是根据当前的配置信息设置端点，那么清除配置的操作正好是反过来的，将端点配置内容清除并还原为默认状态。如果设备只有一套配置，那么可以在复位时完成所有端点的配置。如果设备有多个配置，那么复位时只能配置端点 0，用于接收枚举和设置地址的请求；等到设备接收到配置请求后，再根据实际配置初始化其他的端点。配置相关内容详见 3,2,3 配置设备。TeenyUSB 只考虑了设备只有一种配置的情况，因此在设备 USB 复位时将所有端点进行了配置。

2.2.2 设置地址 Address Assignment

当主机对设备复位后，主机会设置从设备地址，这个地址在 USB 的树状结构中是唯一的。在设置地址之前，主机会通过默认地址（即 0 地址）发起一次读取设备描述符请求， 通过设备描述符得到设备端点 0 的最大包长。

设置地址是一次没有数据的控制传输，无数据控制传输需要通过两包数据传输来完成（详见 2.3.2.3 无数据控制传输）。一个是 Setup 包，从主机到设备；一个是状态返包，由设备到主机。当设备收到设置地址命令后，此时设备地址还是 0。如果设备在此时立即设置地址并生效，那么后面设备发送的状态包会通过新设置的地址发送出去，这样在主机端看来，这次地址设置没有收到 Status IN，设置地址失败。因此设备需要在状态包发送成功后才能将新地址生效。STM32 的 USB_FS 模块地址设置后立即生效，因此这类 USB 模块需要在设备的状态包发送成功后才能设置地址。而 STM32 的 OTG 模块，由模块内部逻辑控制，在状态发送成功后新地址才生效，因此 OTG 这类模块可以在收到设置地址的 Setup 包后直接设置新地址。下图是 TeenyUSB 与 HAL 库在不同的 USB 模块中，对设置地址请求的处理流程：

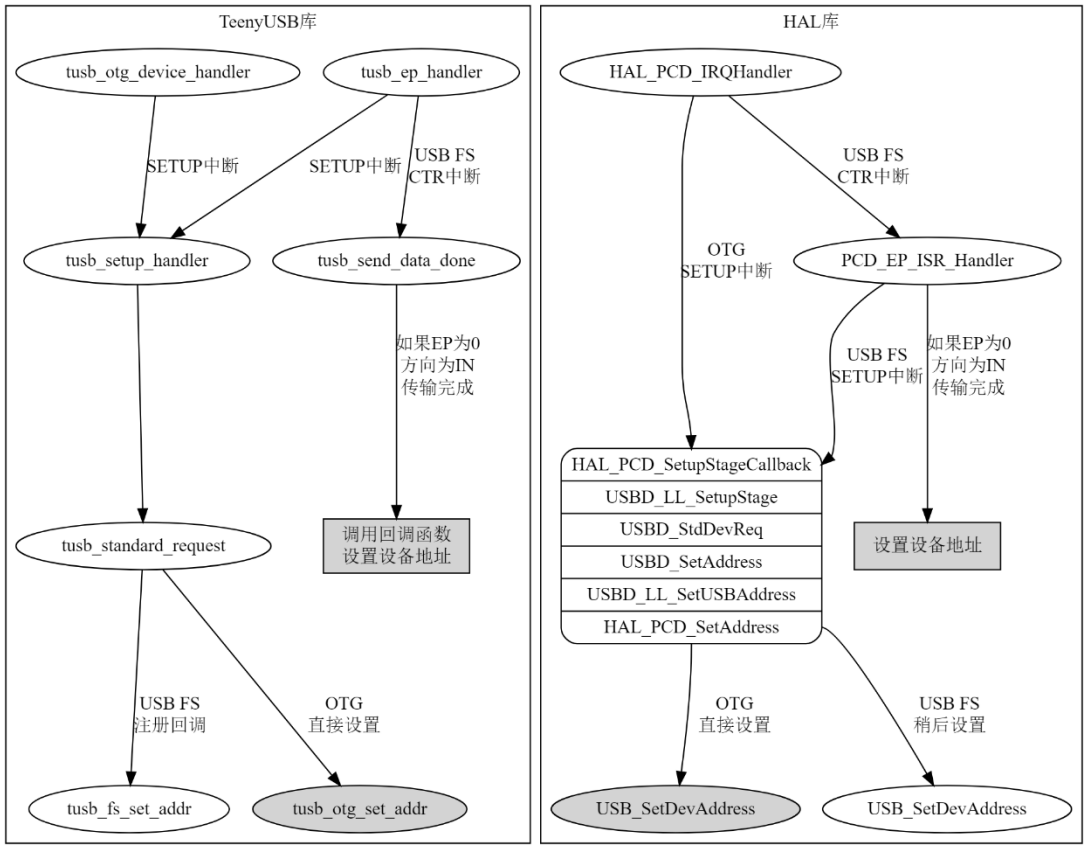


图1 设置地址流程

下面介绍 ST 官方 HAL 库和 TeenyUSB 中处理设置地址请求的流程的源代码。

2.2.2.1 HAL 库 STM32F1 系列设置地址请求

处理设置地址请求的代码如下，调用流程依次从第一函数到最后一个函数：

```
static void USBD_SetAddress(USBD_HandleTypeDef *pdev , USBD_SetupReqTypedef *req){
  ...
    dev_addr = (uint8_t)(req->wValue) & 0x7F;
  ...
      pdev->dev_address = dev_addr;
      USBD_LL_SetUSBAddress(pdev, dev_addr);
  ...
}
USBD_StatusTypeDef USBD_LL_SetUSBAddress(USBD_HandleTypeDef *pdev, uint8_t dev_addr){
  ...
  hal_status = HAL_PCD_SetAddress(pdev->pData, dev_addr);
  ...
}
HAL_StatusTypeDef HAL_PCD_SetAddress(PCD_HandleTypeDef *hpcd, uint8_t address){
  ...
  hpcd->USB_Address = address;
  USB_SetDevAddress(hpcd->Instance, address);
```

```
  ...
}
HAL_StatusTypeDef  USB_SetDevAddress (USB_TypeDef *USBx, uint8_t address){
  if(address == 0) {
   /* set device address and enable function */
   USBx->DADDR = USB_DADDR_EF;
  }
  return HAL_OK;
}
static HAL_StatusTypeDef PCD_EP_ISR_Handler(PCD_HandleTypeDef *hpcd){
  ...
    if (epindex == 0){
  ...
       if((hpcd->USB_Address > 0U)&& ( ep->xfer_len == 0U)){
         hpcd->Instance->DADDR = (hpcd->USB_Address | USB_DADDR_EF);
         hpcd->USB_Address = 0U;
       }
  ...
}
```

通过分析上面的代码，设备在收到设置地址请求后依次调用了 USBD_SetAddress，USBD_LL_SetUSBAddress，HAL_PCD_SetAddress，USB_SetDevAddress 这四个函数。而在最后的 USB_SetDevAddress 函数中，根据其判定条件，在收到主机发来的新地址后并没真正的去设置地址。在之前的 HAL_PCD_SetAddress 函数中，接收到的地址存储在了 hpcd 的 USB_Address 字段中，这个字段在 PCD_EP_ISR_Handler 函数中进行了处理。PCD_EP_ISR_Handler 是端点事件处理函数，当端点数据发送成功或是接收成功时调用。从函数中的判断条件来看，当端点事件为 IN，并且端点号为 0 传输长度为 0 时，会根据 USB_Address 字段中的内容来设置 USB 模块的地址寄存器。端点 0 长度为 0 的 IN 事件，就是控制传输中的 Status IN 发送成功的事件（详见 2.3.2 控制传输）。ST 的 HAL 库的这种处理方式就是为了在状态返回成功后，才将 USB 地址设置到寄存器中生效。

2.2.2.2 HAL 库 STM32F7 系列设置地址请求

HAL 库中处理设置地址请求的代码如下：

```
HAL_StatusTypeDef  USB_SetDevAddress (USB_OTG_GlobalTypeDef *USBx, uint8_t address){
  uint32_t USBx_BASE = (uint32_t)USBx;
  USBx_DEVICE->DCFG &= ~ (USB_OTG_DCFG_DAD);
  USBx_DEVICE->DCFG |= ((uint32_t)address << 4) & USB_OTG_DCFG_DAD;
  return HAL_OK;
}
```

F7 系列的设置地址请求与 F1 系列的类似，上面只列出了不同的部分，其他处理流程都相同。不同的部分出现在 usb_ll 库。在 F7 系列中，USB_SetDevAddress 函数直接将收到的地址设置到了相应的寄存器中。这是因为在 F7 芯片上的 OTG 模块中，地址将会在下一个 Status IN 发送成功后才真正生效，因此代码中不用等到 Stauts IN 发送成功后再去设置。

2.2.2.3 TeenyUSB 设置地址请求

处理设置地址请求的代码如下：

```
static void tusb_standard_request(tusb_device_t* dev, tusb_setup_packet* setup_req){
  ...
  case USB_REQ_SET_ADDRESS:
    dev->addr = LO_BYTE(setup_req->wValue);
#if defined(USB_OTG_FS) || defined(USB_OTG_HS)
    tusb_set_addr(dev);
#else
    dev->ep0_tx_done = tusb_set_addr;
#endif
  ...
}
#if defined(USB_OTG_FS) || defined(USB_OTG_HS)
static void tusb_set_addr (tusb_device_t* dev){
  USB_OTG_GlobalTypeDef *USBx = dev->handle;
  USBx_DEVICE->DCFG &= ~ (USB_OTG_DCFG_DAD);
  USBx_DEVICE->DCFG |= (dev->addr << 4) & USB_OTG_DCFG_DAD ;
}
#else
// Callback function for set address setup
static void tusb_set_addr(tusb_device_t* dev){
  if (dev->addr){
    GetUSB(dev)->DADDR = (dev->addr | USB_DADDR_EF);
    dev->addr = 0;
  }
}
#endif
void tusb_ep_handler(tusb_device_t* dev, uint8_t EPn){
  ...
  if ( (EP & USB_EP_CTR_TX)) { // something transmitted
    if(EPn == 0 && dev->ep0_tx_done){
      // invoke status transmitted call back for ep0
      dev->ep0_tx_done(dev);
      dev->ep0_tx_done = 0;
    }
    TUSB_CLEAR_TX_CTR(GetUSB(dev), EPn, EP);
    tusb_send_data_done(dev, EPn, EP);
  }
  ...
}
```

在 TeenyUSB 的 tusb_standard_request 函数中，收到设置地址请求后，根据 USB 模块

的不同，选择不同的处理方式。如果是 OTG 设备，则直接调用 tusb_set_addr 函数将地址设置到寄存器中；如果是 FS 设备，则将 tusb_set_addr 函数挂在 ep0_tx_done 回调函数上，当端点 0 的数据传输完成时调用注册在 ep0_tx_done 上的 tusb_set_addr 函数，完成地址设置。

2.2.3 配置设备 Configuration

在《USB 规格书》中，设备必须要配置之后才能使用。当设备收到配置设备（Set Configuration）请求时，设备根据当前请求中的配置值来选择配置。一般情况一个设备只有一个配置，所以在收到配置设备请求后，只是简单的将当前配置记录下来，用来给以后的读取配置（Get Configuration）请求返回数据。

在配置确定后，配置中的每个接口（Interface）也可以有多种不同的接口配置，通过设置接口（Set Interface）请求进行选择。在收到设置接口请求后确定设备所用的接口，根据接口描述符（Interface Descriptor）中的端点描述符（Endpoint Descriptor）进行端点的初始化配置。

对于只有一种配置并且接口也只有一种设置的 USB 设备，可以在复位的时候将需要用到的端点一次性配置好。因为后续的 Set Configuration 和 Set Interface 请求都不会改变端点的使用情况。因此，TeenyUSB 库并没有按照 USB 规格书的要求，在收到配置（Set Configuration）请求后才设置端点，而是直接在收到复位事件后，就配置了所有的端点。官方的 HAL 库在 Reset 时只初始化端点 0，在后续的设置配置请求中完成其他端点的初始化。

2.2.4 电源管理

如果是总线供电（BUS Power）的设备，需要在配置描述符中声明所需的电量。如果设备支持远程唤醒（Remote Wake Up），也需要在配置描述符中说明。主机可以通过获取状态（Get Status）请求读取设备当前的电源情况。通过设置特性（Set Feature）请求设置设备是否支持远程唤醒。具体内容见 2.5.4 读取状态请求。

2.2.5 设备请求处理/设备枚举（Request Processing）

设备请求处理，也叫做设备枚举，这个阶段主机会通过一系列的控制传输命令，获取设备信息，对设备进行配置。这个阶段是 USB 设备最关键也最复杂的一个处理阶段，后面会用一个单独的章节（2.5 标准设备请求）来进行详细说明。

2.2.6 接口类型相关的请求

不同的接口类型有一些类型相关的请求。例如 HID 设备的读取报告描述符（Get Report Descriptor）请求，CDC 设备的设置/读取线路参数（Set/Get Line Coding）请求。这些请求处理成功后，设备就可以正常工作了。

2.2.7 基本操作总结

当 USB 设备接上 USB 电缆后，主机先对设备进行复位，这样设备的 USB 模块就工作在了一个已知的状态下。复位后主机先读取设备的设备描述符的前 8 字节，得到设备端点 0 的最大包长，然后设置设备地址，开始发送控制传输命令对设备进行枚举。枚举完毕后主机就能与设备进行通讯了。主机端在枚举过程中会为设备安装相应的驱动程序。

表1 设备基本操作顺序

设备上电
设备复位
获取设备描述符前8字节
设备复位（可选操作）
设置设备地址
获取设备描述符及其他描述符
配置设备
配置接口（可选操作）
配置设备特性（可选操作）
根据设备接口类型设置设备（接口类型相关）
设备正常工作

2.3 USB 数据流

USB 的数据传输由基本事务（Transaction）构成，在基本事务之上，有控制（Control），中断（Interrupt），批量（Bulk），同步（Isochronous）四种数据传输模式，下面分别介绍这四种传输模式。

2.3.1 基本事务

USB 数据的传输事务都是由主机发起的，一个完整的事务包含令牌阶段（Token Stage）、数据阶段（Data Stage）和响应阶段（Acknowledge Stage）。在令牌（Token）阶段由主机指明传输数据的类型，数据阶段传输数据或是状态，数据阶段之后是响应阶段，用来判断数据是否发送成功。如下图所示：

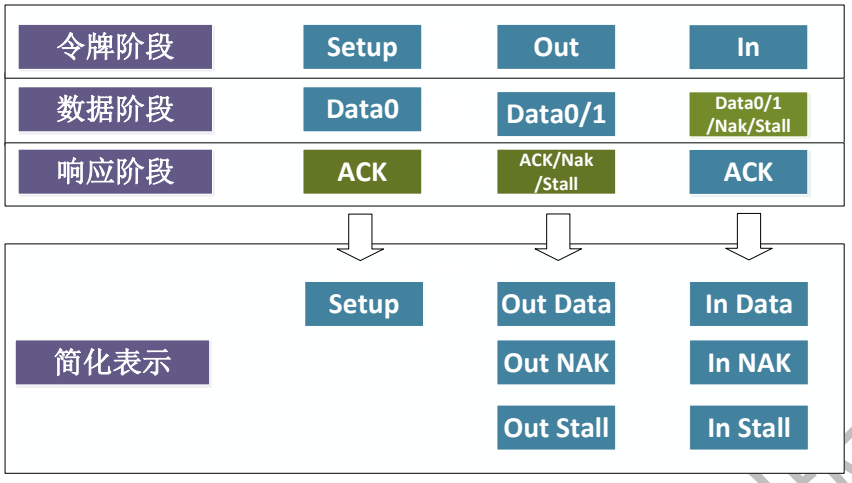


图2 基本数据传输事务（不含同步事务）

设备收到 Setup 事务后只能响应 ACK，因此 Setup 传输只有成功这一种情况。设备收到 Out 事务后，如果能够接收，响应 ACK；如果不能接收，比如缓存未准备就绪，响应 NAK；如果传输的数据不能正确处理，可以响应 STALL。因此 Out 事务有 Out Data，Out NAK，Out Stall 三种状态。IN 事务与 OUT 事务类似，不同的是 IN 事务的状态返回是在数据阶段。

同步传输的 IN 和 OUT 事务没有响应阶段。

在高速设备的批量传输和控制传输中，根据上图 OUT 事务是否成功需要等到响应阶段才知道，如果响应是 NAK，主机在数据阶段发送的内容不会被从机接收，浪费掉了这部分带宽。为了节约总线带宽，除了 IN 和 OUT 事务，高速设备还有 PING 事务，用来查看端点是否准备完毕，如果端点没有准备就绪响应 NYET 状态。在高速设备中应该何时发 PING 包查询状态，何时直接发起 OUT 事务呢？这个问题 USB 协议给了一个不太完美的解决办法，详见 3.6.3 非周期性端点。

在后面介绍不同种类的传输类型时，会采用简化表示来描述一个事务，一个简化表示包含了令牌、数据和响应这三个阶段，一次完整的传输由多个基本事务组成。

令牌、数据和响应这三个阶段还有更底层的数据结构和处理方式，本文不对其做介绍. 更底层的数据结构在参考文档《USB 2.0 规格书》8.4 节 Packet Formats 中有详细介绍。

2.3.2 控制传输 Control transfer

控制传输是 USB 最基本也是最重要的传输，所有的设备都会支持这种传输模式。设备的配置，命令，状态都通过控制传输传递。控制传输都由主机发起，主机先传输一个 Setup 包到设备，Setup 包中包含了这次控制传输的类型和其它参数信息。设备接收到 Setup 包后，先对 Setup 包进行解析，根据 Setup 包的内容进行后续操作。有 3 种控制传输类型，分别为：写数据控制传输（Control Write）、读数据控制传输（Control Read）和无数据控制传输（No-Data Control）。

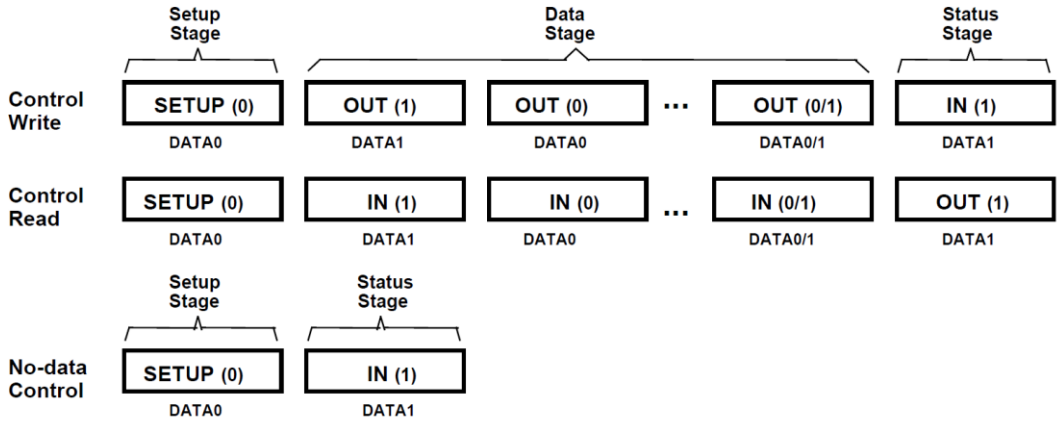


图3　控制传输数据流（来自 USB2.0 规格书）

控制传输会返回三种状态，完成、错误和处理中。见下表：

Status Response	Control Write Transfer (sent during data phase)	Control Read Transfer (sent during handshake phase)
Function completes	Zero-length data packet	ACK handshake
Function has an error	STALL handshake	STALL handshake
Function is busy	NAK handshake	NAK handshake

表2 控制传输的状态返回 （来自 USB2.0 规格书）

在 USB 通讯中，状态都是从设备发向主机的，不同的传输类型对状态的处理有所不同。如果控制传输设备不能处理，设备将端点 0 的 IN 和 OUT 两个方向同时设置为 STALL 状态。由于 Setup 包不受端点状态影响，始终可以发送成功，因此端点 0 设置为 STALL 状态后并不会影响后续的 Setup 包传输。设备在收到新的 Setup 包后开始一次新的控制传输处理例程，并根据此次的处理结果设置端点的状态。

在写控制传输的中，设备成功收到所有数据后，发送一个包长为 0 的包来告诉主机数据传输成功。

在读控制传输成中，设备通过 ACK 响应来告诉主机数据传输成功。这个 ACK 响应指的是：在主机读取到正确的数据后，会发一个长度为 0 的状态包给设备，而设备接收到这个状态包后，响应 ACK 表示传输成功，响应 NAK 表示正在处理中，响应 STALL 表示出错。

2.3.2.1 写数据控制传输 Control Write

主机发起写数据控制传输，先发送一个 Setup 包到设备，然后开读取设备的端点 0，并向设备端点 0 发送后续的数据。此时设备的端点 0 IN 方向还没有数据，主机会收到 NAK 信号，表示正在处理种。设备收到来自主机的 Setup 包后，端点 0 的 OUT 方向一般会由硬件控制会变为 NAK 状态，主机发送的后续数据会阻塞。

设备对接收到的 Setup 包进行解析，获得长度信息。根据 Setup 包中的长度信息准备接收缓存，缓存准备好之后将 OUT0 端点设置为 VALID 状态。此时主机发送的数据能够正确发送到设备。设备接收完毕主机数据后对数据进行处理，处理成功后发送一个数据长度为 0

的包（即状态包）到主机。主机收到状态包后，完成此次写数据控制传输。如果设备不支持或数据有错，将端点设置为 STALL 状态。主机收到 STALL 状态后进行相应的处理。

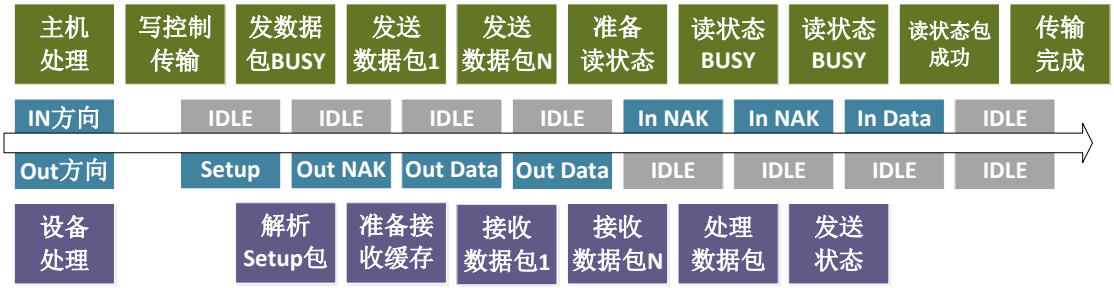


图4 写控制传输流程

上图的解读方式：分析 Out 方向或 Setup 数据时，主机的处理在事务块的前面，设备的处理在事务块的后面；分析 In 方向的数据时，主机的处理在事务块的后面，设备的处理在事务块的前面。灰色的 IDLE 表示主机在这个阶段不会发出令牌包请求数据，分析时可以忽略。上图分解后流程如下所示，顺序从左至右：

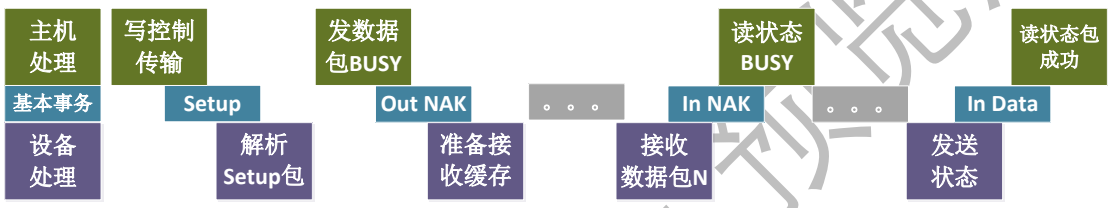


图5 写控制传输流程分解

2.3.2.2 读数据控制传输 Control Read

主机发起读数据控制传输，先发送一个 Setup 包到设备，然后开读取设备的端点 0。此时设备的端点 0 IN 方向还没有数据，主机会收到 NAK 信号，表示数据正在处理中。设备收到来自主机的 Setup 包后，端点 0 的 OUT 方向一般会由硬件控制会变为 NAK 状态，主机发送的后续状态包会阻塞。

设备对接收到的 Setup 包进行解析，获得长度信息。根据 Setup 包中的长度信息和内容通过端点 0 的 IN 方向开始发送数据，主机在 IN 方向上开始接收数据。设备可以在数据发送完毕后或者启动数据发送时，将端点 0 的 OUT 方向设置为 VALID 状态，用来接收主机的状态包。

主机接收到所有的数据包后，发送长度为 0 的状态包，完成此次读数据控制传输。如果设备不支持或数据有错，将端点设置为 STALL 状态。

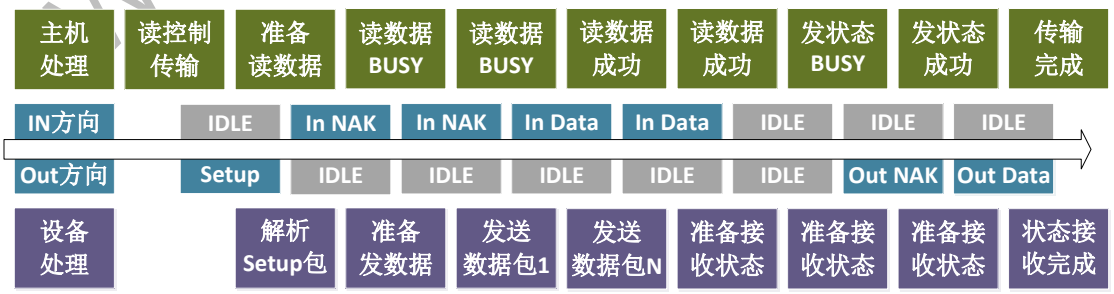


图6 读控制传输流程

2.3.2.3 无数据控制传输 No-data Control

Setup 包中包含了所需的数据，因此不需要后续的数据传输阶段。主机发送完 Setup 包后，设备根据 Setup 内容进行处理，成功发送包长为 0 的状态包，失败将端点设置为 STALL 状态。

图7 无数据控制传输流程

2.3.2.4 控制传输端点设置

USB 设备默认端点 0 为控制传输端点，FS 设备端点 0 最大包长可以为 8、16、32、64 字节，HS 设备最大包长为 64 字节。为了同时兼容 FS 和 HS 设备，TeenyUSB 默认将端点 0 的大小设置为 64 字节。设备接入主机时，主机会发送一个读取设备描述符的 Setup 包，并设置度取数据的长度为 8。设备描述符的前 8 字节包含了端点 0 的最大包长，主机通过解析前 8 字节内容获得端点 0 的最大包长，之后的控制传输以实际的端点 0 最大包长进行传输。

控制传输的 Setup 命令会无视端点的实际状态，传输到设备上，因此控制传输的 Setup 阶段总是会成功。当 Setup 传输成功后，设备端点 0 的 OUT 方向会自动切换成 NAK 状态。在写数据控制传输时，主机后续要发送的数据会被阻塞，直到设备将端点 0 的 OUT 方向设置为 VALID 状态。这样设备有充分的时间去解析 Setup 包内容，为后续的控制传输数据阶段准备缓存。

2.3.3 同步传输 Isochronous transfer

同步传输适用用于数据传输频率固定，能够容忍错误的场合。如音频信号，摄像头的视频流信号。这类数据要求实时性，对错误可以容忍。对于 FS 设备，一个 ISO 包长最大为 1023 字节，对于 HS 设备，最大包长为 1024 字节，在 HS 设备的高带宽端点中，一帧可以传递最多三个最大包长的包，即一帧最多可传输 3072 字节数据。

2.3.4 中断传输 Interrupt transfer

中断传输以固定的频率进行数据传输。USB 主机端根据设备上报的端点描述符（Endpoint Descriptor）信息，为中断传输端点分配带宽。和同步传输一样，在 HS 设备的高带宽端点中，一帧可以最多进行三次传输。同步传输和中断传输，有时又一起叫做周期性传输（Periodic Transfer）。针对周期性传输主机会根据上报的频率信息分配带宽。

2.3.5 批量传输 Bulk transfer

批量传输与中断传输类似，不同的是批量传输会占用总线上所有空余的带宽。因此在批量端点描述符中，没有使用时间间隔字段。在高速设备中，时间间隔字段用来向主机说明一帧中 NAK 的次数。

2.4 传输完成条件 Transfer Complete Condition

在控制传输、中断传输和批量传输中，都有一个传输完成（Transfer Complete）的概念。传输完成表明这一次需要传输的数据已经发送完毕，接收方可以进行后续处理了。控制传输可以看成是有一个 Setup 包作为前导的批量传输。在 USB 中一次传输任务会被分解为多个基本事务，例如一次 IN 传输，由多个 IN 事务组成。USB 协议中并没有要求在传输数据时发送本次需要传输的数据长度，因此 USB 协议通过最大包长和每一次基本事务的数据长度共同来确定一次数据的总长读。

在 USB 设备的端点描述符中包含了端点最大包长（Max Packet Size 即 MPS）信息。USB 的端点一次只能传输最大包长的数据，如果数据超出最大包长，将会分成多包进行传输。多包传输时，前面的数据包长度都是最大包长，只有最后一包数据根据实际长度传输。当传输的数据满足完成条件时，表明一次数据传输过程已经完成。传输完成的条件有两种：

1. 当设备或主机收到的数据不是最大包长时，表示当前传输完成，可以进行后续处理，这种情况又叫做收到短包（Short Packet）；
2. 当设备或主机收到的数据与期望数据长度一致（expect length），表示当前传输完成，可以进行后续处理；

第 1 点比较好理解，当要完成一次传输时，发送一个数据长度不是最大包长的数据即可，有这两种情况：

a. 如果数据长度不能最大包长整除，最后一包自然会小于最大包长，触发传输完成条件；
b. 如果数据长度能被最大包长整除，最一包与最大包长相等，需要再发送一个长度为零的包（Zero Length Packet）来触发传输完成条件。

2.4.1 期望数据长度

关于第 2 点，什么是期望数据长度？

在 PC 端的 USB 主机上，以 libusb 为例，为了接收来自 USB 设备的数据，需要创建一个 transfer，创建 transfer 的时候会分配一块内存，这块内存的大小就是 USB 主机的期望数据长度。libusb 中的 transfer 创建好之后提交给后台进行监听，当一次传输完成时调用回调函数获取数据。

在 USB 设备上，同样也会用一块内存来接收端点数据，这块内存的大小就是 USB 设备端的期望数据长度。当主机发送的数据满足传输完成条件时，USB 设备端的处理程序将收到的数据交给应用层来处理。

无论在主机还是在设备上，期望数据长度通常是最大包长的整数倍。在一些特别的情况下，期望数据长度可以与最大包长相等，这样在收到每一包数据时都会完成一次传输。

2.4.2 传输完成的一些实例

如果发送方在发送数据时，为能被最大包长整除的数据包添加一个 0 包。这种情况下，始终会有一个短包（长度小于最大包长或者长度为 0）。这样无论对方的期望长度是多少，都能正确的触发传输完成条件，如下图：

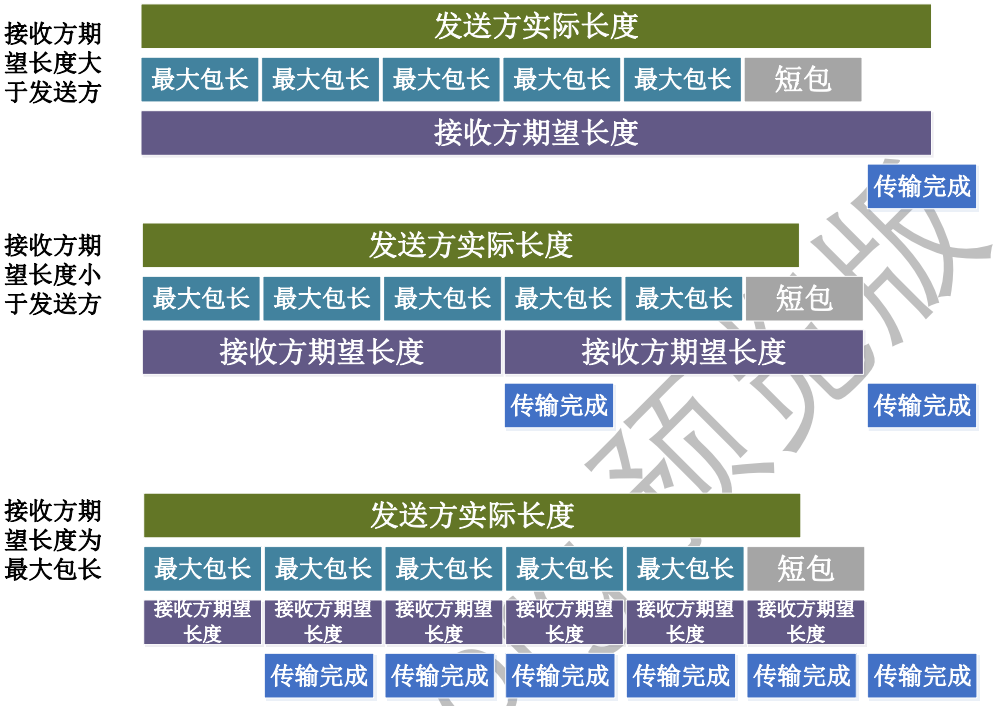


图8 有短包的传输

如果发送方在发送能被最大包长整除的包时，不添加 0 包。在这种情况下，传输有可能会挂起，如下图这种情况：

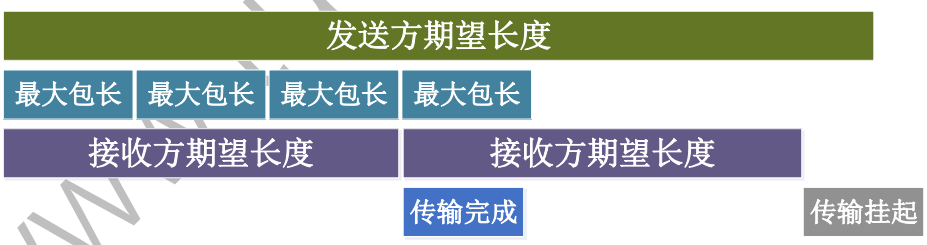


图9 无短包传输-挂起

上图种，发送方发送了一个长度刚好为 4 倍最大包长的数据，后面一段数据小于接收方期望长度，并且接收方没有收到短包，传输没有完成挂起了。这种情况下，有两种处理办法：

一种是通过超时的方式将已经接收到的数据进行处理，然后重新设置接收缓存。比如利用 SOF 中断，周期性检测接收的缓存中是否有数据，如果有，无论是否收到短包都进行处理。

另一种是接收方将期望长度设置为最大包长，这样每包数据都会触发传输完成条件，即使没有短包也能正常完成传输，如下图所示：

图10　无短包传输-正常

无短包传输的这两种处理策略在后续的 CDC 串口例程中会作更详细的说明。

2.5 标准设备请求 Standard Device Request

标准设备请求通过端点 0 的控制传输传递，控制传输都是由主机发起，主机先向设备发送一个 Setup 包，用来通知设备请求的类型和参数。设备根据接收到的 Setup 包内容决定后续的动作。在 TeenyUSB 中，标准设备请求的处理代码在 teeny_usb.c 文件中。当端点接收到 Setup 包后，通用 tusb_setup_handler 函数，根据 Setup 包内容选择处理的方式。

2.5.1 Setup 数据结构

标准请求通过控制传输进行传输，标注设备请求的 Setup 包数据结构定义见下表：

表3 Setup 数据格式 （表格来自 USB2.0 规格书）

Offset	Field	Size	Value	Description
0	*bmRequestType*	1	Bitmap	Characteristics of request: D7: Data transfer direction 0 = Host-to-device 1 = Device-to-host D6...5: Type 0 = Standard 1 = Class 2 = Vendor 3 = Reserved D4...0: Recipient 0 = Device 1 = Interface 2 = Endpoint 3 = Other 4...31 = Reserved
1	*bRequest*	1	Value	Specific request (refer to Table 9-3)
2	*wValue*	2	Value	Word-sized field that varies according to request
4	*wIndex*	2	Index or Offset	Word-sized field that varies according to request; typically used to pass an index or offset
6	*wLength*	2	Count	Number of bytes to transfer if there is a Data stage

Setup 数据在控制传输的 Setup 包中传输，方向都是主机到设备，长度都是 8 个字节。主机和设备的默认通道（端点 0）间在同时只会有一个控制传输，因此在设计设备端的代码时，可以为设备预留一个 8 字节的缓存，用来保存当前端点 0 上的 Setup 包。TeenyUSB 中定义 Setup 包的结构类型的代码：

```
typedef struct _tusb_setup_packet{
  uint8_t  bmRequestType;
  uint8_t  bRequest;
  uint16_t wValue;
  uint16_t wIndex;
  uint16_t wLength;
} tusb_setup_packet;
```

不同的请求类型个字段的含义不同，下表是标准设备请求各字段的含义：

表4 标准设备请求（表格来自 USB2.0 规格书）

bmRequestType	bRequest	wValue	wIndex	wLength	Data
00000000B 00000001B 00000010B	CLEAR_FEATURE	Feature Selector	Zero Interface Endpoint	Zero	None
10000000B	GET_CONFIGURATION	Zero	Zero	One	Configuration Value
10000000B	GET_DESCRIPTOR	Descriptor Type and Descriptor Index	Zero or Language ID	Descriptor Length	Descriptor
10000001B	GET_INTERFACE	Zero	Interface	One	Alternate Interface
10000000B 10000001B 10000010B	GET_STATUS	Zero	Zero Interface Endpoint	Two	Device, Interface, or Endpoint Status
00000000B	SET_ADDRESS	Device Address	Zero	Zero	None
00000000B	SET_CONFIGURATION	Configuration Value	Zero	Zero	None
00000000B	SET_DESCRIPTOR	Descriptor Type and Descriptor Index	Zero or Language ID	Descriptor Length	Descriptor
00000000B 00000001B 00000010B	SET_FEATURE	Feature Selector	Zero Interface Endpoint	Zero	None
00000001B	SET_INTERFACE	Alternate Setting	Interface	Zero	None
10000010B	SYNCH_FRAME	Zero	Endpoint	Two	Frame Number

上表中，有的 bmRequestType 项有多行内容，说明这类请求可以有多种接收者，接收者的定义见表 2 Setup 数据格式。当接收者为接口或端点时，通过 wIndex 字段制定接口号或是端点号。下面是 TeenyUSB 中对 Setup 包的处理主函数：

```
// Setup packet handler
void tusb_setup_handler(tusb_device_t* dev) {
  tusb_setup_packet *setup_req = &dev->setup;
  if( (setup_req->bmRequestType & USB_REQ_TYPE_MASK) == USB_REQ_TYPE_CLASS){
    tusb_class_request(dev, setup_req);
#if defined(HAS_WCID)
  }else if((setup_req->bmRequestType & USB_REQ_TYPE_MASK) == USB_REQ_TYPE_VENDOR){
    tusb_vendor_request(dev, setup_req);
#endif
  }else{
    tusb_standard_request(dev, setup_req);
  }
```

```
}
```

通过解析 bmRequestType 字段中的值，确定当前的 Setup 包的类型，并调用对应接收者的处理函数。当类型是标准请求时，调用 tusb_standard_request 函数作进一步处理。本节主要讲解 tusb_standard_request 函数中的处理方式，函数实现如下：

```
// Standard request process function
static void tusb_standard_request(tusb_device_t* dev, tusb_setup_packet* setup_req) {
  switch (setup_req->bRequest) {
  case USB_REQ_SET_ADDRESS:
    dev->addr = LO_BYTE(setup_req->wValue);
#if defined(USB_OTG_FS) || defined(USB_OTG_HS)
    tusb_otg_set_addr(dev);
#else
    dev->ep0_tx_done = tusb_fs_set_addr;
#endif
    tusb_send_data(dev, 0, 0, 0);
    break;
  case USB_REQ_GET_DESCRIPTOR:
    tusb_get_descriptor(dev, setup_req);
    break;
  case USB_REQ_GET_STATUS:
    tusb_send_data(dev, 0, (uint16_t *) &dev->status, 2);
    break;
  // Only support one configuration, so just save and return the config value
  case USB_REQ_GET_CONFIGURATION:
    tusb_send_data(dev, 0, (uint16_t *) &dev->config, 1);
    break;

  case USB_REQ_SET_CONFIGURATION:
    dev->config = LO_BYTE(setup_req->wValue);
    tusb_send_data(dev, 0, 0, 0);
    break;
  // Only support one alt setting, so just save and return the alt value
  case USB_REQ_GET_INTERFACE:
    tusb_send_data(dev, 0, &dev->alt_cfg, 1);
    break;
  case USB_REQ_SET_INTERFACE:
    dev->alt_cfg = LO_BYTE(setup_req->wValue);
    tusb_send_data(dev, 0, 0, 0);
    break;
  case USB_REQ_SET_FEATURE:
    if(setup_req->wValue == USB_FEATURE_REMOTE_WAKEUP){
      dev->remote_wakeup = 1;
      tusb_send_data(dev, 0, 0, 0);
      break;
```

```
    }
    // otherwise fall to default
  case USB_REQ_CLEAR_FEATURE:
    if(setup_req->wValue == USB_FEATURE_REMOTE_WAKEUP){
      dev->remote_wakeup = 0;
      tusb_send_data(dev, 0, 0, 0);
      break;
    }
    // otherwise fall to default
  default:
    // Error condition, stall ep0
    STALL_EP0(dev);
    break;
  }
}
```

在上面的函数中，除了读取描述符（Get Descriptor）之外，其他的请求都在这个函数内做了简单处理并返回状态。

2.5.2 设置地址 Set Address

设备收到这个请求后，设置设备的地址，这个地址一般设置在 USB 模块的相关寄存器中。不同的芯片设置方式有所不同，在 2.2.2 设置地址一节中有详细说明。

2.5.3 读取描述符 Get Descriptor

根据参数不同，读取设备的各种描述符，描述符是 USB 协议中最核心的内容，设备各种功能与特性都是通过描述符来进行描述的。描述符的第一个字节是描述符长度，然后是描述符类型。一般主机会先发送一个短的读取描述符命令，得到描述符的前面部分内容，根据这部分内容得到整个描述符的长度，然后再发送完整的读取描述符命令，得到完整的描述符内容。

Setup 包中 wValue 字段的高 8 为表示描述符类型，低 8 为表示描述符的索引。如果一个设备有多个配置描述符，通过索引值来获取需要的配置描述符。TeenyUSB 不支持多配置的设备，因此在获取配置描述符时没有检测描述符的索引，直接返回了当前设备的配置描述符。

2.5.4 读取状态 Get Status

读一个 2 字节长的 Status 数据，在标准请求中 Status 数据只定义了 bit0 表示是否自供电，bit1 表示是否支持远程唤醒，其他位都默认位 0。

在 TeenyUSB 协议栈中，Status 数据来自于配置描述符的属性字段，根据属性字段设置 Status 的值。在 teeny_usb_desc.c 文件中，根据配置描述符设置 DEV_STATUS 宏定义的值，如下代码所示：

```
// Power status
#define BULK_DEV_STATUS0      USB_CONFIG_SELF_POWERED
#define BULK_DEV_STATUS1      USB_CONFIG_REMOTE_WAKEUP
#define BULK_DEV_STATUS    ((BULK_DEV_STATUS0) |(BULK_DEV_STATUS1) )
```

在设备初始化函数中，将 DEV_STATUS 的值赋给设备结构体的 status 字段，然后在设备接收到读取状态请求后返回给主机。

```
#define BULK_TUSB_INIT_DEVICE(dev) \
  do{\
    /* Init device features */       \
    memset(dev, 0, TUSB_DEVICE_SIZE);    \
    dev->status = BULK_DEV_STATUS;         \
    dev->descriptors = &BULK_descriptors;         \
  }while(0)
```

USB 设备支持的读取状态请求如下表所示：

表5 读取状态请求

状态	接收者	wValue	数据位	数据位含义
设备状态	设备	0	Bit0	0:总线供电 1:自供电
			Bit1	0:支持远程唤醒 1:不支持远程唤醒
			Bit2-Bit15	保留，默认为0
接口状态	接口	0	Bit0-Bit15	保留，默认为0
端点状态	端点	0	Bit0	1:端点暂停 0:端点正常工作
			Bit1-Bit15	保留，默认为0

2.5.5 读取/设置配置、读取/设置接口

在 TeenyUSB 协议栈中，Get/Set Configuration 和 Get/Set Interface 请求没有做特殊处理，都是在设置参数的请求中将接收到的参数保存下来，在读取参数的请求中将设置的参数返回回去。这是因为 TeenyUSB 只支持一个配置，并且每个接口（Interface）中只支持一种设置。这里没有在收到设置配置请求后进行端点设置，是因为在设备复位的时候，设置端点 0 参数时一起将所有的端点都设置了。

按照标准的做法，设备收到 USB 复位信号时，只配置端点 0 参数，用来收发控制传输数据。设备接收到设置配置请求后，根据参数选择当前配置，并在当前配置中选择接口的默认配置，并根据默认接口配置的参数设置相应端点。设备收到设置接口请求后，检查新的接口设置是否有效，如果有效根据新设置的接口配置端点，返回 ACK；如果新的接口配置无效，返回 STALL。

如果支持多个配置，设备收到 Set Configuration 请求后，根据选择的配置重新设置端点。

如果支持多种接口配置，在收到配置接口的请求时，需要先判断当前的请求是发往哪一个接口的，再根据新的接口配置重新设置接口的端点参数。

2.5.6 设置/清除特性 Set/Clear Feature

设置清除特性，wValue 字段表示要设置/清除的特性内容，TeenyUSB 在标准请求中只支持了远程唤醒（Remote Wakeup）这一个特性，其他的特性请求都会触发错误，返回 STALL 状态。完整的特性设置如下表所示：

表6　设置/清除特性请求

特性	接收者	wValue	备注
停止端点	端点	0	将所有端点设置为STALL模式
远程唤醒	设备	1	打开/关闭设备远程唤醒功能
测试	设备	2	测试时使用
B设备HNP	设备	3	只对OTG设备有效
A设备HNP	设备	4	只对OTG设备有效

2.6 USB 数据流控

在任何通讯协议的设计中，数据的流量控制都是一个很重要的问题。当通讯双方的数据处理速率不一致时，就需要进行流控，避免数据溢出丢失。USB 是一个半双工的通讯协议，完全由主机控制数据的传输。因此主机端的流控比较简单，当数据超出主机处理能力的时候，主机不再发起传输即可。而对于设备端而言，当数据超出设备处理能力时，设备通过 NAK 应答的方式拒绝接收主机的数据。控制传输中的 Setup 包是一个例外，Setup 包不能被设备拒绝，任何情况下都会接收成功，因此设备端代码在设计时会为 Setup 包单独准备一块缓存。而 Setup 包后续的数据包，主机和设备可以根据实际情况进行流控处理。

主机向设备发数据的流控：主机通过 OUT 事务向设备发送数据，当数据超出了设备的处理能力时，设备将 OUT 端点置为 NAK 状态。此后主机的 OUT 事务会收到 NAK 响应，进入等待状态。当设备可以继续接收的时候，再将 OUT 端点置为 VALID 状态。

设备向主机发数据的流控：以 libusb 为例，当主机接收到的数据超出其处理能力时，主机可以暂时不提交(不调用 libusb_submit_transfer)已经完成的 transfer，这样 USB 总线就不会发起读数据命令。设备接收不到读数据的命令，待发送的数据就会一直停留在缓存中。这个过程可以理解为主机主动读取设备数据，主机有足够的处理能力才会发起 IN 事务，因此 IN 事务的响应阶段只有 ACK，没有其他状态。

2.6.1 同步端点的流控

USB 中的同步端点流控与其他类型的传输不同，同步传输没有传输完成的概念，也没有流控的概念。比如声卡的数据流，声音是实时的，不能因为数据超出某一方的处理能力，就将数据停住不处理。因此同步端点只有 Enable 和 Disable 状态，没有 NAK 这样的状态。当端点处于 Enable 状态时，表示通道建立成功，可以发送数据；当端点处于 Disable 状态时，表示通道断开，不能再发送数据。如果同步端点的数据超出处理能力，丢掉当前数据就即可，处理能力恢复后再接着处理后续的数据。如果当前数据没有准备就绪，可以往同步端点上发送长度为 0 的包。同步端点的 IN 事务中，如果设备数据未就绪，发送长度为 0 的数据。同步端点的 OUT 事务中，如果主机数据未就绪，不发起 OUT 事务。

2.6.1.1 TeenyUSB 中同步端点流控方式

在 TeenyUSB 中，IN 方向的 ISO 数据，如果设备未准备就绪，就发送长度为 0 的同步包。因此在 PC 端的代码中采用了下面的处理方式：

```
if(xfer->type == LIBUSB_TRANSFER_TYPE_ISOCHRONOUS){
    if(xfer->actual_length > 0){
        struct libusb_iso_packet_descriptor * desc = xfer->iso_packet_desc;
        // inplace copy make buffer data continue
        unsigned char * buf_actual_data = xfer->buffer;
        unsigned char * buf_packet_data = xfer->buffer;
        for(int i=0; i< xfer->num_iso_packets; i++){
            if(desc->status == LIBUSB_TRANSFER_COMPLETED){
                if(buf_actual_data != buf_packet_data){
                    memcpy(buf_actual_data, buf_packet_data, desc->actual_length);
                }
            }
            buf_actual_data += desc->actual_length;
            buf_packet_data += desc->length;
            desc++;
        }
        int real_length = buf_actual_data - xfer->buffer;
        if(real_length > 0){
            emit ep->m_parent.epDataReady( ep->m_info.bEndpointAddress,
                                           QByteArray((char *)xfer->buffer, real_length));
        }
    }
}
```

上面的代码是基于 libusb 编写的，先检查端点类型和实际数据长度。在同步传输中，这里的实际长度不是接收到的数据长度，而是提交给更底层通讯驱动的缓存大小。真正传输的数据长度在 iso_packet_desc 这个数据结构中。从 iso_packet_desc 中提取出数据的真实长度，并将数据拼在一起。如果最后实际接收的数据长度为 0 则不做处理，如果不为 0，触发 epDataReady 信号。因此当设备数据未就绪时，可以将 ISO 端点发送方向的数据长度设置为 0。下面是设备中 ISO 端点的处理代码：

```
void tusb_send_data_done(tusb_device_t* dev, uint8_t EPn){
  ...
  else if( IS_ISO() ){
    ep->tx_pushed = 0;
    pma = (EP & USB_EP_DTOG_TX) ? PMA_TX0(dev, EPn) : PMA_TX1(dev, EPn);
    pma->cnt = 0;
  }
  ...
  if(ep->tx_size || (EPn == 0 && ep->tx_last_size == GetInMaxPacket(dev, EPn)) ){
    copy_tx(dev, ep, pma, ep->tx_buf, ep->tx_size, GetInMaxPacket(dev, EPn));
```

```
    PCD_SET_EP_TX_STATUS(GetUSB(dev), EPn, USB_EP_TX_VALID);
    return;
  }
  ...
}
```

在端点的数据发送完成中断处理函数中，得到当前应用程序使用的 PMA 描述表地址，立即将发送数据长度设置为 0。后面再根据端点剩余数据的长度，进行后续处理。如果没有更多的数据需要传递，发送数据长度会保持为 0，发送长度为 0 的包。

在这种处理方式下，无论 ISO IN 端点是否有数据，主机和设备都会频繁的进行到数据 IN 传输完成的中断中。实际应用中设计 ISO 端点时，会根据实际传输数据的码率来配置端点最大包长和传输频率，让传输的数据的速率与数据产生的速率一致，尽量不“空转”。

ISO OUT 端点的数据由主机发往设备，有数据需要发送时，主机才会启动传输。设备端一直保持端点可用，等待主机的数据。

2.7 设备端开发调试步骤

根据前面的基本操作流程，这里总结一下开发 USB 设备时调试流程。

2.7.1 USB 模块核心配置

在开发 USB 设备时，当设备接入主机并且能收到主机的复位命令，说明 USB 模块至少已经工作起来了，USB 核心寄存器配置基本正确。如果不能收到复位命令，则说明有可能有这些问题：核心寄存器配置不正确、引脚模式不正确、时钟配置不正确、硬件连线不正确等。

2.7.2 端点 0 OUT 方向配置

更进一步，如果能收到读取设备描述符的控制传输，说明端点 0 的 OUT 方向已经配置正确。由于控制传输的特殊性，端点 0 在任何情况下都必须能正确接收主机的控制传输命令中的 Setup 包，一般芯片会在硬件上对端点 0 的设置进行控制，用户端不可配置。因此这一步通常都会成功，如果失败，应该去检查端点 0 OUT 方向的相关配置是否正确。

紧跟在 Setup 令牌后的数据包含了 Setup 的内容，如果能从端点 0 的接收缓存中读取到正确的 Setup 命令，则说明端点 0 的接收缓存设置成功。这个时候可以根据接收到的 Setup 包内容进行后续操作了。如果只收到 Setup 标志，但是没有正确接收到 Setup 包的内容，说明端点 0 的接收缓存配置有问题，重点检查端点的缓存配置情况。

2.7.3 端点 0 IN 方向配置

第一个控制传输请求一般是获取设备描述符的前 8 字节，收到这个请求后，可以将设备描述符的前 8 字节通过端点 0 IN 发送出去。如果发送成功，主机接下来会发送设置地址的请求。如果失败，主机会复位，再次发送读取设备描述符的请求。失败时，主要检查端点 0 IN 方向的配置是否正确，缓存设置是否正确。如果这一步成功，至此已经实现了端点 0 简

单数据的收发。后续的控制传输会传递多余 8 字节的包，有可能还会传递超出端点 0 最大包长的包，如果对长包的处理有问题，后面在枚举其他信息时也会出问题。

2.7.4 设备枚举

当端点 0 的基本收发都成功后，接下来开始调试设备的枚举过程。枚举过程中调试设备的各种请求是否都进行了正确的处理。如果某一个请求没有正确处理，在设备管理器中可以看到出错的请求的信息。例如，将设备的处理设置地址请求部分的代码注释掉，再接入电脑，可以在设备管理器中看到设置地址失败的信息。如果把发送配置描述符的代码注释掉，可以看到设备管理器中的获取描述符失败的信息。

图11　　获取描述符失败

出现获取描述符失败的错误时，主要检查端点 0 的数据接收和发送是否有问题，缓存配置是否正确。

在 STM32F0、STM32F1 这类设置了地址立即生效的设备中，如果没有在状态发送完成后才设置地址，会出现“设置地址请求失败”的错误，如下图：

图12　　设置地址请求失败

出现设置地址请求失败时，主要检查设置地址的逻辑是否有问题，走到这一步说明上一步的获取设备描述符已经成功。

2.7.5 驱动安装

枚举成功后，操作系统会位设备安装驱动程序。如果是标准的 USB 设备，如键盘鼠标等，系统会自动安装驱动程序；如果是用户自定义的设备，需要安装用户提供的驱动程序。驱动安装成功后用户才能通过应用程序与设备进行通讯。在驱动的安装过程中，驱动也会对设备进行一些配置，例如串口驱动会设置和读取设备的编码方式，如果这些请求没有正确处理，也会出错。

2.7.6 使用设备

如果是标准的 USB 类设备，驱动安装成功后就能用通用的软件进行测试了。例如：CDC 串口类设备，在驱动安装成功后，可以使用串口测试工具进行测试；U 盘类的设备，驱动安装成功后，能够在系统中看到盘符，进行操作。

如果是自定义的设备，在 Windows 上通常通过 SetupAPI 获取到设备的路径，用 CreateFile 函数通过路径打开设备获得设备句柄，最后通过句柄对设备进行操作。在操作的过程如果出错，通过 GetLastError 函数得到具体的错误代码，再分析出错的原因。

对于 libusb 支持的设备，也可以使用 libusb 提供的 API 进行操作。先通过 list_devices 获取挂载在总线上的 USB 设备，然后根据 VID 和 PID 对设备进行筛选，最后用 open 函数打开设备进行操作。

3 USB 标准描述符

USB 描述符是 USB 设备中最基础的内容，设备的所有信息都通过描述符来体现。本章主要介绍各个描述符的作用。

主机与设备间进行用户的业务数据通讯前，先获取设备相关的描述符，根据描述符的内容挂载驱动，确定设备访问的方式。描述符中也包含了设备所用到的所有端点的信息，通过描述符也可以确定 USB 模块的端点使用情况。USB 设备中的描述符结构如下图所示：

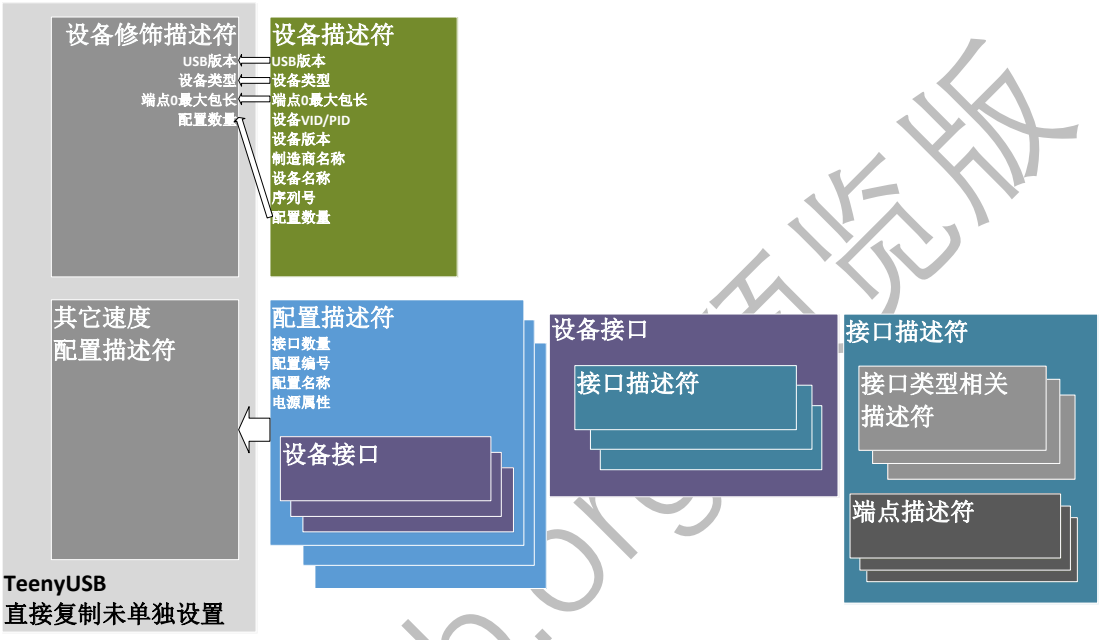


图13　USB 描述符结构

上图设备描述符中的设备类型是由三个字段构成的，分别是 bDeviceClass、bDeviceSubClass 和 bDeviceProtocol。在后文中提到接口类型时，也是由 xxxClass、xxxSubClass 和 xxxProtocl 这三个字段组成的。这个时候简写成“类型为（a，b，c）”的形式，其中 a 表示 Class，b 表示 SubClass，c 表示 Protocol。

本章没有逐字段去介绍各个标准描述符的内容，描述符的各个字段具体含义在《USB2.0 规格书》中 9.6 Standard USB Descriptor Definitions 一节中有详细的描述。本章的最后一节 3.10 描述符工具会介绍如何使用 TeenyUSB 配套的描述工具）（TeenyDT）来生成描述符，在那里会对描述符各字段进行详细介绍。

3.1 设备描述符 Device Descriptor

设备描述符中描述了设备一些基本信息，包括 USB 版本号、设备类型、端点 0 最大包长、设备 VID/PID、设备版本号、制造商名称、设备名称、序列号以及配置描述符的数量。Windows 系统根据设备的 VID 和 PID 进行驱动匹配，如果设备是多接口的，还会根据接口号进行匹配。正式产品中的 VID 需要向 USB 组织去申请，PID 由厂商自己定义。本文主要为讲解 USB 设备的工作原理，生成的设备不会作为正式产品面市，因此在 DEMO 程序中使用了 ST 公司 USB 库中的 VID 0x0483。

3.2 配置描述符 Config Descriptor

一个设备可以有多个配置描述符，配置描述符由两部分构成，前面部分是配置描述符的基本信息，包含了当前配置中的接口数量、配置编号、配置名称和和电源属性信息。后面部分包含了多个功能模块描述符。前面的基本信息和后面的功能描述一起构成了配置描述符。功能不同描述符内容也不同，因此完整配置描述符是一个变长数据结构，完整长度在基本信息的 wTotalLength 字段中。一般情况下主机先读取配置描述符的基本信息，根据基本信息计算出总长度，再读取完整的配置描述符。

3.3 设备接口

配置描述符基本信息之后跟着设备接口相关的描述符，一个配置可以有多个接口。一个接口可以有多个接口描述符，接口中的第一个接口描述符代表了当前接口。当有 IAD 描述符时，IAD 描述符的类型代表了当前接口的类型。

当设备有多个接口时设备描述符中的需要设置为类型（0，0，0），这表示设备的类型由接口来确定。

在 Windows 上，系统会为设备的每一个设备接口安装驱动。进行驱动匹配时：

1. 如果设备只有一个接口，驱动会按照设备来进行匹配，设备路径是 VID_xxxx&PID_yyyy 形式，其中 xxxx 是设备 VID，yyyy 是设备 PID。
2. 如果设备有多个接口，驱动会按照每个接口来进行匹配，此时设备的路径是 VID_xxxx&PID_yyyy&MI_zz 形式，zz 即接口号。

3.4 接口联合描述符 Interface Association Descriptor

如果一个设备功能需要用多个接口来描述，这些接口需要通过 IAD（Interface Association Descriptor）描述符联合起来。但是当一个配置中只有一个功能时，即便这个功能需要多个接口，也不用 IAD 描述符。一旦配置描述符中有 IAD 描述符，设备的类型必须设置为类型（0xEF，0x02，0x01）。IAD 描述符的使用条件如下图所示：

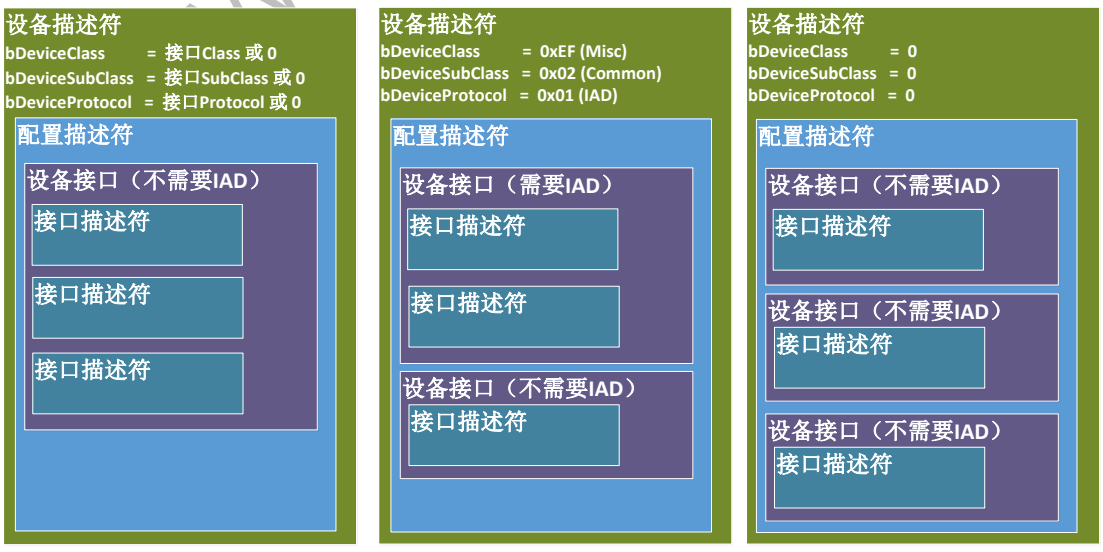


图14 IAD 使用条件

当有 IAD 描述符存在时，设备接口的类型由 IAD 描述符的类型来确定。

3.5 接口描述符 Interface Descriptor

一个设备接口由一个或多个接口描述符构成，设备接口中第一个接口描述符的类型代表了当前接口的类型。接口描述符包含了接口号、接口类型、端点数量、接口名称信息，如果接口支持多种配置，不同接口配置通过 bAlternateSetting 字段进行区分。根据接口描述符的类型不同，接口描述符后会有类型相关的描述符。如 HID 设备的 HID 描述符，CDC 设备的功能描述符（Function Descriptor）。

3.6 端点描述符 Endpoint Descriptor

在 USB 设备中，端点是主机和设备之间进行通讯的基本单元。一个 USB 设备无论多复杂，有多少的接口，最终与主机进行通讯的都是端点。配置设备和设置接口请求最终目标都是为了确定当前设备所用到的端点配置情况。

在 USB 总线上，通过设备地址和端点地址就能唯一确定一条数据的来源。设备地址在设备连接时由主机分配，设备内的端点地址由设备自身在设计时决定。这有点像是 IP 网络中的 IP 地址和端口号，IP 地址由服务商分配，而端口号由服务内容决定。通过 IP 地址和端口号，就能获取到需要的服务。

端点号由 8 位数据组成，最高位 bit7 表示数据的方向，最高位为 1 时，表示这是一个 IN 端点；最高位为 0 时，表示这是一个 OUT 端点。在 USB 中讨论数据方向时都是站在主机的角度来描述的。IN 表示设备发往主机的数据，OUT 表示主机发往设备的数据。

USB 设备通过端点描述符上报端点信息，这些信息包括：

1. 端点号及数据传输方向
2. 传输类型
3. 传输频率
4. 最大包长

3.6.1 高速高带宽端点 High speed high bandwidth endpoint

在 HS 设备中，同步端点和中断端点可以设置为高速高带宽端点，这样在一帧中可以传递多包数据，端点描述符 wMaxPacketSize 字段中的 Bit11 和 Bit12 表示一帧传输的包数。0 表示普通端点，只传一包；1 表示一帧 2 包，2 表示一帧 3 包。更多内容见参考文档 1 的 5.9 节。当一帧只传输 1 次时，最大包长可以为小于等 1024 的任意值；一帧传输 2 次时，最大包长为 513 至 1024 字节；一帧传输 3 次时，最大包长为 683 至 1024 字节。

为什么批量端点不用设置为高带宽模式？因为批量端点本身就会占用一帧剩余的所有带宽。而中断端点和同步端点这类周期性的端点（Periodic Endpoint），在一帧中只会发起一次传输，通过高带宽相关的描述符，可以突破这个限制，增加一帧中的数据传输数量。

3.6.2 周期性端点 Periodic Endpoint

USB 中的周期性端点指的是类型是中断传输或同步传输的端点，这类端点最快调度周期就是一帧传输一次。USB 通讯是一帧一帧进行的，在一帧中可以进行多中类型的传输。对 FS 设备而言，一帧是以 1ms 为周期；对高速设备而言，一帧以 125us 为周期。中断传输和同步传输的最大传输频率是一帧一次。

对于 FS 设备而言，传输频率 bInterval 字段表示传输频率，单位为 1ms，bInterval=4 时表示每 4ms 传输一次。

对于 HS 设备而言，$2^{bInterval-1}$ 表示传输频率，单位为 125us，bInterval=4 表示 2^{4-1}x125us=1ms，每 1ms 传输一次，当 bInterval=1 时，每 125us 传输一次。

3.6.3 非周期性端点 Non-Periodic Endpoint

与周期性端点相对应的，是非周期性端点（Non-Periodic Endpoint）。控制端点和批量端点都是非周期性端点。非周期性端点的传输会占用总线上剩余的所有带宽。在 FS 设备中，非周期性端点的 bInterval 字段没有含义。

在 HS 设备中的非周期性端点中，bInterval 表示一帧中的 NAK 频率，当 bInterval 为 0 时，表示永不 NAK。其实这个 bInterval 在非周期性端点中并没有什么用处。试想一下，如果一个高速设备的 BULK OUT 端点 bInterval 设置为 0，根据规范说明这个端点永不 NAK。如果此时这个设备确实不能继续接收数据了，还是会 NAK 掉主机的 OUT 事务。笔者的理解是，这里的 NAK 频率更像是一个承诺，在高速设备中有 PING 机制，当主机看到这个承诺后，认为设备在 NAK 一定的次数后，就会 ACK 后续的包。这样主机接收到一定数量的 NAK 后就不会再发 PING 包，而是直接发起 OUT 事务包。例如：当主机看到设备承诺一帧最多 NAK 10 次时，当主机 PING 10 次都被 NAK 后，第 11 次主机就会直接发 OUT 包了。在实际使用中，设备真的能保证第 11 次就能成功吗，如果不能保证，主机的 OUT 事务阶段的带宽还是会被浪费掉，不如继续发 PING 包合算。因此笔者觉得这个针对高速设备非周期端点的 bInterval 设计用处不大，不如一直用 PING 包探测合算。

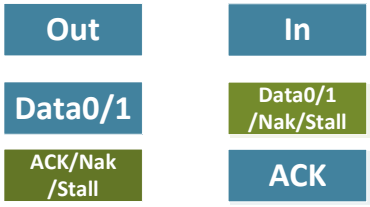


图15　　OUT 事务包与 IN 事务包

如上图所示，OUT 事务中，主机会先发 OUT 令牌和数据，再等设备响应。如果设备没有准备就绪，数据阶段就会浪费带宽，因此引入了 PING 机制来探测设备是否就绪，就绪后才开始 OUT 事务，发送数据。而 IN 事务令牌包后是设备的数据或设备的响应，如果设备没有准备就绪，在数据阶段就可以返回 ACK，不会浪费带宽。因此 PING 机制只用在非周期性的 OUT 传输中。

3.7 设备修饰描述符 Device Qualifier Descriptor

设备修饰描述符用在当一个设备能够工作在不同的速度下时，会获取设备修饰描述符。比如有一个高速的设备和一个全速的设备，他们的 VID、PID 以及设备版本号都一样，先接入高速的设备，系统会“记住”他的速度是高速的。拔掉后再接入一个全速的设备，由于他们的标识都一样，系统会认为这是个相同的设备工作在不同的速度下，会请求设备修饰描述符。这里其实虽然用了两个不同速度的设备来举例，但是在主机端看来，这和一个设备工作在两种速率是一样的。反过来如果先接入全速再接入高速，也会这样。

TeenyUSB 中没有单独的设备修饰描述符，而是通过设备描述符自动生成的。

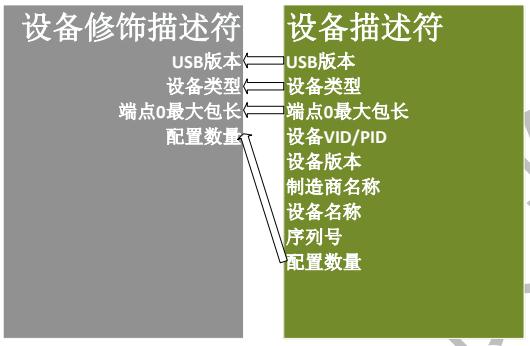


图16　　设备描述符与设备修饰描述符

3.8 其他速度配置描述符 Other Speed Configuration

其他速度配置描述符与设备修饰描述符配合使用，当工作在其他速度下时使用，除了描述符类型，其他字段含义格式与配置描述符完全一致。TeenyUSB 没有单独的其他速度配置描述符，直接根据设备描述符生成的其他速度配置描述符。

TeenyUSB 中通过 SUPPORT_OTHER_SPEED 宏启用其他速度配置功能，启用后会开辟一块描述符缓存，用来临时存放根据配置描述符生成的其他速度配置描述符。

3.9 字符描述符 String Descriptor

USB 中的字符采用 Unicode 格式，一个字符占用 2 字节。对于 ASCII 字符，低 8 位是 ASCII 值，高 8 位为 0。字符描述符不是以 0 结尾，而是在字符描述符的第 1 个字节描述字符的总长度。在设备、配置、接口描述符中，都含有描述名称的字段，这个字段是一个索引值。0 表示没有字符，其他值表示字符索引。索引为 0 的字符描述符表示字符的语言 ID（language ID）列表，一个设备可以支持多个语言 ID。读取字符描述符时，请求的 wIndex 字段表示语言 ID，根据语言 ID 返回相应的字符描述符。TeenyUSB 不支持多语言，因此在获取字符时没有对 wIndex 进行处理。

3.9.1 WCID 设备中的特殊字符描述符

在 Windows 系统中，有一种叫做 WCID 的 USB 设备，这类设备的驱动不是以 VID/PID

进行匹配的，而是根据 Compatible ID 来进行匹配的。这样做的好处是，只要是 WCID 相同的设备，即使是不同厂家生产的，也可以使用相同的驱动。为了支持 WCID，设备需要响应一个特殊的字符描述符请求，并返回一个特殊的字符描述符。Windows 系统会根据这个字符描述符中的参数，发起厂商自定义请求来获取设备的 WCID。获取到 WCID 后，根据 WCID 进行驱动匹配与安装。WCID 设备详细介绍见 6.2.8 WCID 设备这一节中的内容。

3.10 描述符工具 TeenyDT

本章前面介绍了几个标准的描述符，本节将介绍标准描述符中各字段具体含义，同时介绍如何使用描述符工具 TeenyDT 来生成描述符。描述符工具有生成描述符、生成端点初始化代码、生成驱动程序三个功能，本节只介绍生成描述符的功能。TeenyDT 可以用两种方式来生成描述符，一种是命令行方式，一种是图形界面（GUI）方式。命令行方式通过一个 lua 格式的输入文件来定义描述符，TeenyDT 根据输入文件生成 C 语言版的描述符。自动生成的描述符是 C 语言格式的，可以在 TeenyUSB 或其他 USB 协议栈中使用。

3.10.1 TeenyDT 命令行模式

TeenyDT 命令行模式下的输入文件采用 lua 格式编写，命令行模式需要 lua5.3 的可执行程序来运行脚本。将 lua 的可执行程序所在目录放入系统环境变量，然后进入 TeenyDT 目录，执行 lua gen_descriptor.lua 命令将会出现下面的结果：

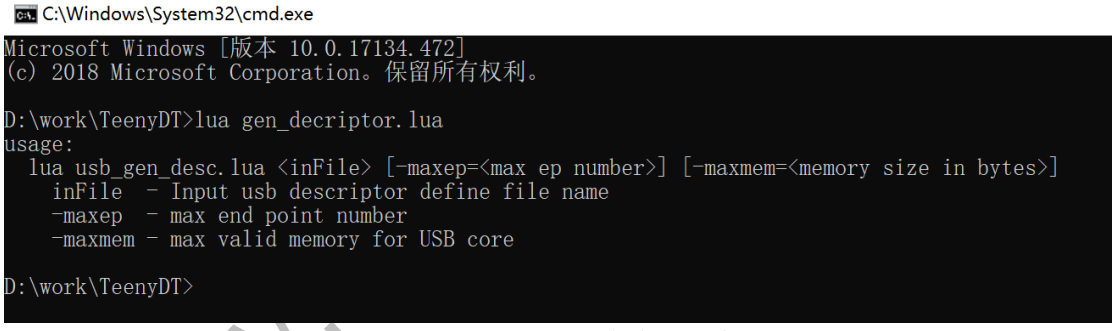


图17 TeenyDT 命令行信息

上面会提示必须要有一个输入文件，以及可选的端点数目和内存大小。这里的内存大小是指 USB_FS 模块的专用内存大小。OTG 模块不需要设置 USB 专用内存大小，会自动生成 1280 字节的 OTG_FS 模块初始化代码，以及 4096 字节的 OTG_HS 模块初始化代码。对于不同的芯片型号，端点数和内存大小不同。端点数目默认是 7，内存大小默认是 1024。关于端点数目和内存大小的作用，在后面介绍 USB 设备时会做详细说明。

3.10.2 设备描述符

设备描述符各字段含义见下表：

表7 设备描述符

字段	含义	TeenyDT处理方式
bLength	描述符长度	自动生成
bDescriptorType	描述符类型，固定值	自动生成
bcdUSB	USB版本	默认为0x200，2.0
bDeviceClass	设备类型	默认为0，由接口制定
bDeviceSubClass	设备子类型	默认为0，由接口制定
bDevicePrototol	设备协议	默认为0，由接口制定
bMaxPacketSize	端点0最大包长	默认为64
idVendor	制造商ID	默认为0x0483
idProduct	设备ID	默认为0x1234
bcdDevice	设备版本	默认为0x0100，1.0
iManufacture	制造商名索引	默认为0，无制造商名
iProduct	设备名索引	默认为0，无设备名
iSerial	序列号索引	默认为0，无序列号
bNumOfConfiguration	配置数量	根据配置数量自动生成

上表中，TeenyDT 为描述符中的大部分内容都预定义了默认值，描述符一些能够根据上下文计算出的值也不用指定，因此 TeenyDT 的输入文件在编写时可以省略很多内容。下图是生成的 C 语言的设备描述符与 TeenyDT 的设备描述符的比较：

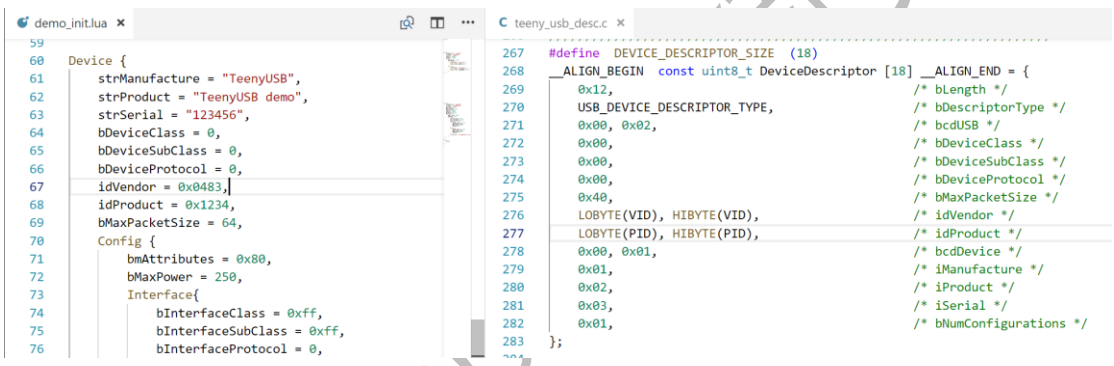


图18　TeenyDT 与 C 语言设备描述符

上图中右侧是生成的 C 语言版本的设备描述符，左侧是 TeenyDT 格式的设备描述符。在上面示例中的 TeenyDT 版本描述符并没有为 bcdUSB 和 bcdDevice 指定内容。默认的 bcdUSB 是 0x200，bcdDevice 是 0x100，右侧生成的 C 代码中 bcdUSB 和 bcdDevice 采用了默认值，分别为 0x200 和 0x100。TeenyDT 中设备描述符的所有字段都可以自动生成或是有默认值，因此一个最简单的 TeenyDT 版本设备描述符中可以什么都不写。

上图的中的 C 语言版的 iManufacture、iProduct 和 iSerial 字段是根据左侧的 strManufacture、strProducr 和 strSerial 自动生成的，字符串在 TeenyDT 中的处理方式见下一节 3.10.3 字符描述符。

3.10.3 字符描述符

设备描述符中通常都会加上制造商名、产品名和序列号。在 TeenyDT 中字符描述符是根据需要的字符串自动生成的。具体做法是在 TeenyDT 版本的描述符中，将需要字符描述符的地方，由 iXXX 改为 strXXX，并附上字符的实际内容。如上面的示例中所示，iManufacture、iProduct 和 iSerial 分别替换成了 strManufacture、strProduct 和 strSerial，生成的描述符中会自动生成相应的字符描述符，并将 iManufacture、iProduct 和 iSerial 设置为对应的字符描述符索引。下面是一个含有字符串的设备描述符，其余各字段都是默认值：

```
return Device {
```

```
    strManufacture = "TeenyUSB",
    strProduct = "TeenyUSB demo",
    strSerial = "123456",
}
```

自动生成的字符描述符内容如下，省略了字符描述符中间的字符内容：

```
// Strings
#define STRING_DESCRIPTOR0_STR          "\x09\x04"
#define STRING_DESCRIPTOR0_SIZE          (4)
WEAK __ALIGN_BEGIN const uint8_t  StringDescriptor0 [4] __ALIGN_END = {
    0x04,                                       /* bLength */
    USB_STRING_DESCRIPTOR_TYPE,                 /* bDescriptorType */
    0x09, 0x04,                                 /* wLangID0 */
};
#define STRING_DESCRIPTOR1_STR          "TeenyUSB"
#define STRING_DESCRIPTOR1_SIZE          (18)
WEAK __ALIGN_BEGIN const uint8_t  StringDescriptor1 [18] __ALIGN_END = {
    0x12,                                       /* bLength */
    USB_STRING_DESCRIPTOR_TYPE,                 /* bDescriptorType */
    'T', 0x00,                                  /* wChar0 */
    ...
    'B', 0x00,                                  /* wChar7 */
};
#define STRING_DESCRIPTOR2_STR          "TeenyUSB demo"
#define STRING_DESCRIPTOR2_SIZE          (28)
WEAK __ALIGN_BEGIN const uint8_t  StringDescriptor2 [28] __ALIGN_END = {
    0x1c,                                       /* bLength */
    USB_STRING_DESCRIPTOR_TYPE,                 /* bDescriptorType */
    'T', 0x00,                                  /* wChar0 */
    ...
    'o', 0x00,                                  /* wChar12 */
};
#define STRING_DESCRIPTOR3_STR          "123456"
#define STRING_DESCRIPTOR3_SIZE          (14)
WEAK __ALIGN_BEGIN const uint8_t  StringDescriptor3 [14] __ALIGN_END = {
    0x0e,                                       /* bLength */
    USB_STRING_DESCRIPTOR_TYPE,                 /* bDescriptorType */
    '1', 0x00,                                  /* wChar0 */
    ...
    '6', 0x00,                                  /* wChar5 */
};
#define STRING_DESCRIPTOR4_STR          "TeenyUSB demo interface"
#define STRING_DESCRIPTOR4_SIZE          (48)
WEAK __ALIGN_BEGIN const uint8_t  StringDescriptor4 [48] __ALIGN_END = {
```

```
    0x30,                                   /* bLength */
    USB_STRING_DESCRIPTOR_TYPE,             /* bDescriptorType */
    'T', 0x00,                              /* wChar0 */
    ...
    'e', 0x00,                              /* wChar22 */
};
const uint8_t* const StringDescriptors[STRING_COUNT] = {
StringDescriptor0,
StringDescriptor1,
StringDescriptor2,
StringDescriptor3,
StringDescriptor4,
};
```

所有的字符描述符都打包到了一个名为 StringDescriptors 的数组中。索引为 0 的字符描述符是语言 ID 列表（Language IDs），表示此设备字符描述符所用的语言。通过 Setup 中的 wIndex 字段来选择不同语言 ID 的字符描述符。TeenyUSB 只支持一种语言，因此这里生成的 LangID 只有一个语言 ID。后面自动为不同的字符内容生成了不同索引字符描述符，这些字符描述符按顺序被合并在了一个名为 StringDescriptors 的数组中。

3.10.3.1 中文字符描述符

中文字符描述符与英文一样，因为 USB 的字符描述符采用的是 Unicode 编码格式，只要保证输入文件的编码格式为 Unicode 就能生成正确的含有中文字符的字符描述符。下图是一个含有中文的字符描述符示例，需要注意左侧文件的编码格式：

demo_chn.lua

```
return Device {
    strManufacture = "微型USB栈",
    strProduct = "Teeny示例",
    strSerial = "123456",
    idVendor = 0x0483,
    idProduct = 0x1234,
    Config{
        RemoteWakeup = true,
        Interface{
            strInterface = "Interface 0",
            EndPoint(OUT(1), Bulk, 64, 1),
            EndPoint(IN(1), Bulk, 64, 1),
        },
        Interface{
            strInterface = "Interface 1",
            EndPoint(OUT(2), BulkDouble, 64, 1),
            EndPoint(IN(3), BulkDouble, 64, 1),
        },
    }
}
```

teeny_usb_desc.c

```
#define STRING_DESCRIPTOR1_STR          "\xce\xa2\xd0\xcdUSB\xd5\xbb"
#define STRING_DESCRIPTOR1_SIZE         (14)
WEAK __ALIGN_BEGIN const uint8_t  StringDescriptor1 [14] __ALIGN_END = {
    0x0e,                                   /* bLength */
    USB_STRING_DESCRIPTOR_TYPE,             /* bDescriptorType */
    0xce, 0xa2,                             /* wChar0 */
    0xd0, 0xcd,                             /* wChar2 */
    'U', 0x00,                              /* wChar4 */
    'S', 0x00,                              /* wChar5 */
    'B', 0x00,                              /* wChar6 */
    0xd5, 0xbb,                             /* wChar7 */
};
#define STRING_DESCRIPTOR2_STR          "Teeny\xca\xbe\xc0\xfd"
#define STRING_DESCRIPTOR2_SIZE         (16)
WEAK __ALIGN_BEGIN const uint8_t  StringDescriptor2 [16] __ALIGN_END = {
    0x10,                                   /* bLength */
    USB_STRING_DESCRIPTOR_TYPE,             /* bDescriptorType */
    'T', 0x00,                              /* wChar0 */
    'e', 0x00,                              /* wChar1 */
    'e', 0x00,                              /* wChar2 */
    'n', 0x00,                              /* wChar3 */
    'y', 0x00,                              /* wChar4 */
    0xca, 0xbe,                             /* wChar5 */
    0xc0, 0xfd,                             /* wChar7 */
};
```

图19　　中文字符描述符

3.10.4配置描述符

配置描述符基本内容中各字段含义如下：

表8 配置描述符基本内容

字段	含义	TeenyDT处理方式
bLength	描述符长度	自动生成
bDescriptorType	描述符类型，固定值	自动生成
wTotalLength	配置描述符总长度	根据接口内容自动生成
bNumInterface	接口数量	根据接口实际情况自动生成
bConfigurationValue	当前配置值	根据当前配置位置自动生成
iConfiguration	配置名索引	默认为0
bmAttributes	电源状态 bit7 始终为1 bit6 1-自供电；0-总线供电 bit5 1-远程唤醒；0-不能远程唤醒 bit0-4 始终为0	默认为0x80，总线供电，不能远程唤醒
bMaxPower	最大消耗电流，总线供电时使用	默认为100，从总线获取最大200mA电流

为了方便编写配置描述符的 bmAttributes 字段，TeenyDT 中为配置描述符定义了 SelfPower 和 RemoteWakeup 两个辅助字段，如果要支持自供电，就将 SelfPower 设置为 true，如果要支持远程唤醒，就将 Remotewakeup 设置为 true。一个总线供电，最大电流 100mA，支持远程唤醒的配置描述符在 TeenyDT 中描述如下：

```
Config {
    SelfPower = false,
    RemoteWakeup = true,
    bMaxPower = 50,
}
```

生成的 C 语言内容如下：

```
// Configs
#define  CONFIG_DESCRIPTOR_SIZE  (9)
__ALIGN_BEGIN const uint8_t ConfigDescriptor [9] __ALIGN_END = {
    0x09,                                   /* bLength */
    USB_CONFIGURATION_DESCRIPTOR_TYPE,      /* bDescriptorType */
    0x09, 0x00,                             /* wTotalLength */
    0x00,                                   /* bNumInterfaces */
    0x01,                                   /* bConfigurationValue */
    0x00,                                   /* iConfiguration */
    0xa0,                                   /* bmAttributes */
    0x32,                                   /* bMaxPower */
};
```

3.10.5接口描述符

接口描述符紧跟在配置描述符的基础部分之后，其内容定义如下：

表9 接口描述符

字段	含义	TeenyDT处理方式
bLength	描述符长度	自动生成
bDescriptorType	描述符类型，固定值	自动生成
bInterfaceNumber	接口ID，以0为起点	根据接口数量自动生成
bAlternateSetting	当前接口值	默认为0
bNumEndpoints	接口端点数量	根据实际端点配置自动生成
bInterfaceClass	接口类型	默认为0xFF，用户自定义类型
bInterfaceSubClass	接口子类型	默认为0xFF，用户自定义子类型
bInterfaceProtocol	接口协议	默认为0
iInterface	接口名索引	默认为0，没有名字

TeenyDT 中的接口描述符所有字段都有默认值或是自动生成，一个自定义的接口类型可以什么都不用写。在结构上接口描述符属于配置描述符，一个配置描述符中可以包含多个接口描述符。接口描述符之后是类型相关的描述符，一般为功能描述符（Function Descriptor），然后是端点描述符，用来描述当前接口需要用到的通讯端点。

接口的类型、子类型以及协议字段定义了接口的功能，USB 组织预定义了一些设备的接口类型。不同类型的接口描述符会在介绍特定设备类型时做详细说明。

3.10.6端点描述符

端点描述符在接口描述符的最后面，其数量在接口描述符中指定，内容如下：

表10　　端点描述符

字段	含义	TeenyDT处理方式
bLength	描述符长度	自动生成
bDescriptorType	描述符类型，固定值	自动生成
bEndpointAddress	端点地址 bit0-3　端点号 bit4-6　保留，始终为0 bit7 端点方向，1-IN 0-OUT	根据用户配置生成
bmAttributes	端点类型	根据用户配置生成
wMaxPacketSize	最大包长 bit0-10 最大包长 bit11-12 一帧传输次数定义 bit13-15 保留，始终为0	默认为64，一帧传输一次
bInterval	上报频率	默认为1，最快

TeenyDT 中有两种定义接口描述符的方式，一种是采用“字段=值”的形式，与接口描述符和配置描述符方式相同；还有一种是初始化参数的形式，将配置参数像函数参数一样传给端点描述符生成器。参数方式的端点定义看起来比较简洁，如下图所示：

demo.lua

```
return  Device{
    strManufacture = "TeenyUSB",
    strProduct = "Teeny DEMO",
    strSerial = "123456",
    Config{
        Interface{
            EndPoint{
                bEndpointAddress = OUT(1),
                bmAttributes = Bulk,
                wMaxPacketSize = 64,
                bInterval = 1,
            },
            EndPoint(IN(1), Bulk, 64, 1),
        }
    }
}
```

teeny_usb_desc.c

```
    0xff,                           /* bInterfaceSubClass */
    0x00,                           /* bInterfaceProtocol */
    0x00,                           /* iInterface */
    /* EndPoint descriptor */
    0x07,                           /* bLength */
    USB_ENDPOINT_DESCRIPTOR_TYPE,   /* bDescriptorType */
    0x01,                           /* bEndpointAddress */
    0x02,                           /* bmAttributes */
    0x40, 0x00,                     /* wMaxPacketSize */
    0x01,                           /* bInterval */
    /* EndPoint descriptor */
    0x07,                           /* bLength */
    USB_ENDPOINT_DESCRIPTOR_TYPE,   /* bDescriptorType */
    0x81,                           /* bEndpointAddress */
    0x02,                           /* bmAttributes */
    0x40, 0x00,                     /* wMaxPacketSize */
    0x01,                           /* bInterval */
};
```

图20　两种不同风格的端点定义方式

3.10.7 接口联合描述符 IAD

当一个接口需要多个接口描述符时，需要使用接口联合描述符将多个接口联合起来。

表11　IAD 描述符

字段	含义	TeenyDT处理方式
bLength	描述符长度	自动生成
bDescriptorType	描述符类型，固定值	自动生成
bFirstInterface	第一个接口ID	根据IAD中的第一个接口自动生成
bInterfaceCount	接口数量	根据IAD中的接口数量自动生成
bFunctionClass	功能类型	根据第一个接口自动生成
bFunctionSubClass	功能子类型	根据第一个接口自动生成
bFunctionProtocol	功能协议	根据第一个接口自动生成
iFunction	功能名索引	默认为0，没有名字

TeenyDT 会根据设备接口的实际情况，生成 IAD 描述符。如果设备只有一个接口，这个接口即使需要多个接口描述符，也不会生成 IAD 描述符。如果设备有多个接口，并且至少有一个接口需要 IAD 描述符，那么 TeenyDT 会将设备类型设置为（0xEF,0x02,0x01），并生成 IAD 描述符。

3.10.8 功能描述符

一些特殊的接口类型中需要指定功能描述符，用来描述接口的一些特殊功能。如 CDC 串口需要功能描述符来描述其能够处理的操作。

表12　功能描述符

字段	含义	TeenyDT处理方式
bLength	描述符长度	自动生成
bDescriptorType	描述符类型，固定值	自动生成
bDescriptorSubtype	描述符子类型，功能相关	使用时指定
bFuncData0	功能相关数据	使用时指定
bFuncData1	功能相关数据	使用时指定
...	...	...

TeenyDT 为了方便使用功能描述符，定义了 alias 和 varData 两个辅助字段。alias 可以为功能描述符起一个别名在生成的代码注释中。varData 是一些键值对（Key-Value），生成功能相关数据时，数据会以 key 为注释。一个功能描述符定义如下：

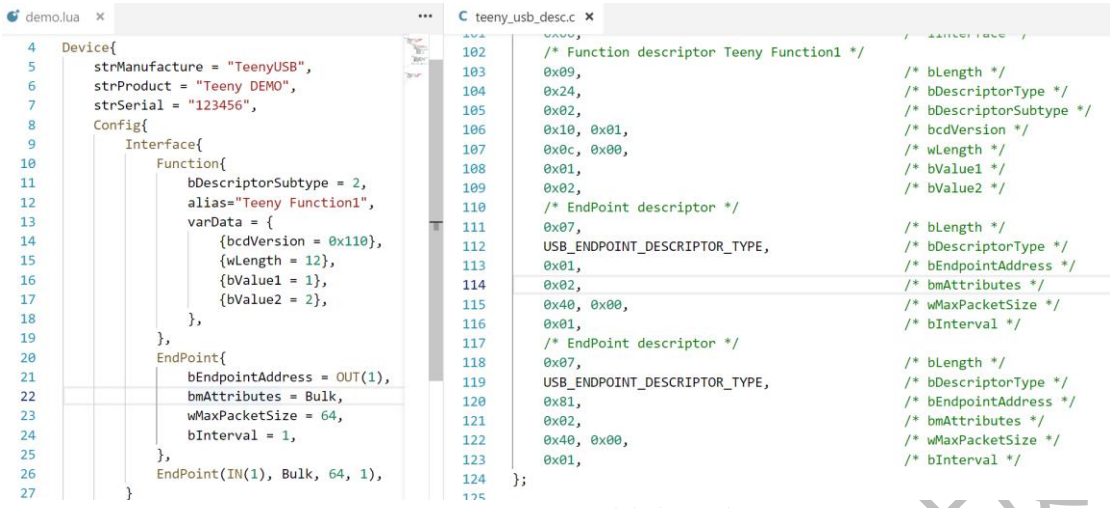


图21　　TeenyDT 功能描述符

由于 lua 中的 KV 表没有顺序，因此在生成 key-value 对时，又用了一个表来包装 KV 对，再以数组的方式存入 varData 表中，保证了数据的先后顺序。

3.10.9 设备描述符举例

TeenyDT 中一个完整的设备描述符如下面代码所示：

```
return Device {
    strManufacture = "TeenyUSB",
    strProduct = "Teeny Demo",
    strSerial = "123456",
    idVendor = 0x0483,
    idProduct = 0x1234,
    Config{
        RemoteWakeup = true,
        Interface{
            strInterface = "Interface 0",
            EndPoint(OUT(1), Bulk, 64, 1),
            EndPoint(IN(1), Bulk, 64, 1),
        }
        Interface{
            strInterface = "Interface 1",
            EndPoint(OUT(2), DoubleBulk, 64, 1),
            EndPoint(IN(3), DoubleBulk, 64, 1),
        }
    }
}
```

上面的代码定义了一个 VID 为 0x0483，PID 为 0x1234 的 USB 设备，厂商名为“TeenyUSB”，产品名为“Teeny Demo”，序列号为“123456”。这个设备有一个配置，支持远程唤醒。有两个接口，接口名称分别为“Interface 0”和“Interface 1”。第一个接口有两个 Bulk 端点，端点地址都为 1。第二个接口也有两个 Bulk 端点，地址分别为 2 和 3，这两个端点都支持双

缓冲模式。在 STM32 芯片中，双缓冲模式在 USB FS 模块中有效，在 OTG 模块中，没有双缓冲模式。

TeenyDT 的输入文件结构与 USB 描述符的结构一致，最外层是设备描述符，设备描述符中可以有多个配置描述符（目前只支持一个配置）。配置描述符中可以有多个接口，每个接口可以由一个或多个接口描述符来描述。这里每个接口只有一个接口描述符，因此没有使用 IAD 描述符。接口描述符中可以有多个端点描述符，这里使用的是自定义接口类型（TeenyDT 默认值），因此接口中没有功能描述符，只有端点描述符。

3.10.10 TeenyDT 图形界面模式

TeenyDT 也支持图形界面模式，在图形界面模式下，除了可以生成描述符和端点初始化代码之外，还可以根据描述符内容生成带签名的驱动程序。

TeenyDT 的图形界面是一个绿色的免安装程序，配合初始化脚本 xtool_init.lua 实现生成 USB 描述符的功能。TeenyDT 图形界面程序基于 XToolbox 开发，有关 XToolbox 的详细说明见 xtoolbox.org。执行 TeenyDT 目录中的 xtoolbox.exe 后出现下面的界面：

图22 TeenyDT 图形界面

在这里编辑设备 VID、PID、厂商名称等信息。

3.10.10.1 添加接口

增加的接口如下图所示，不同接口需要配置不同的参数，对于通用接口，可以添加更多的端点。对于 HID 设备，可以定义其报告描述述的内容。

图23　　TeenyDT 增加接口

3.10.10.2 生成描述符和端点初始化定义

设备及接口定义好后，预览生成的描述符和端点定义文件，如下图：

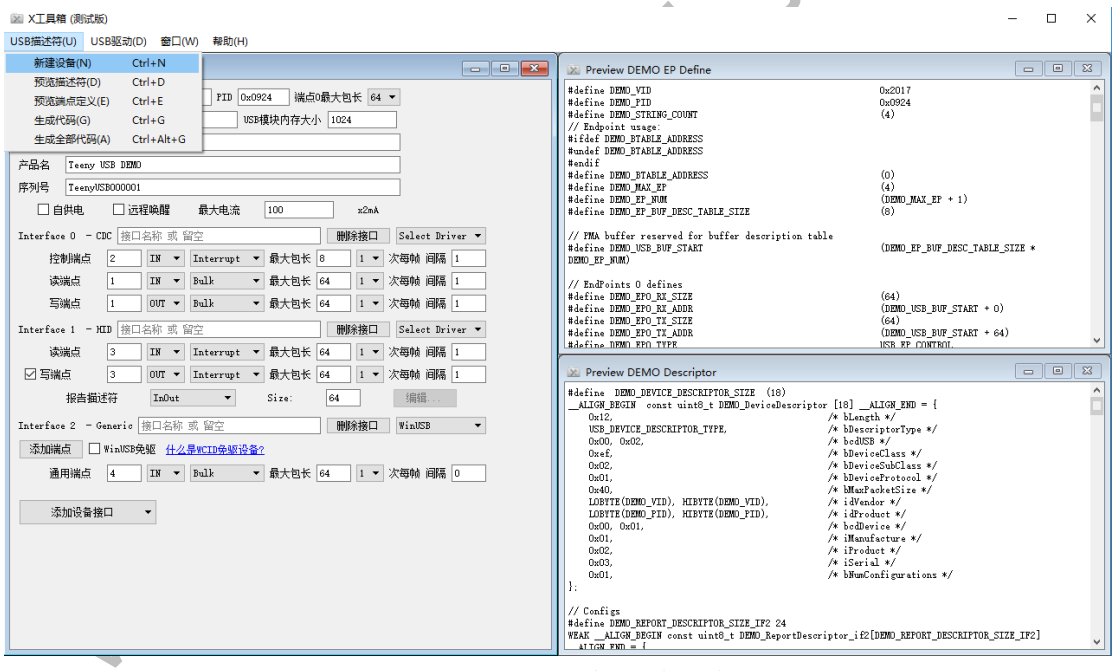


图24　　TeenyDT 预览描述符及端点定义

USB 描述符菜单功能说明见下表：

表13　　TeenyDT 描述符菜单

菜单名称	功能
创建设备	生成一个新的USB设备
预览描述符	查看当前设备的描述符
预览端点定义	查看当前设备的端点定义
生成代码	生成当前设备的描述符和端点定义文件
生成全部代码	将所有设备的描述符和端点定义生成到同一个文件中

3.10.10.3 生成驱动文件

TeenyDT 能够根据描述符生成对用的驱动文件，方式如下图所示：

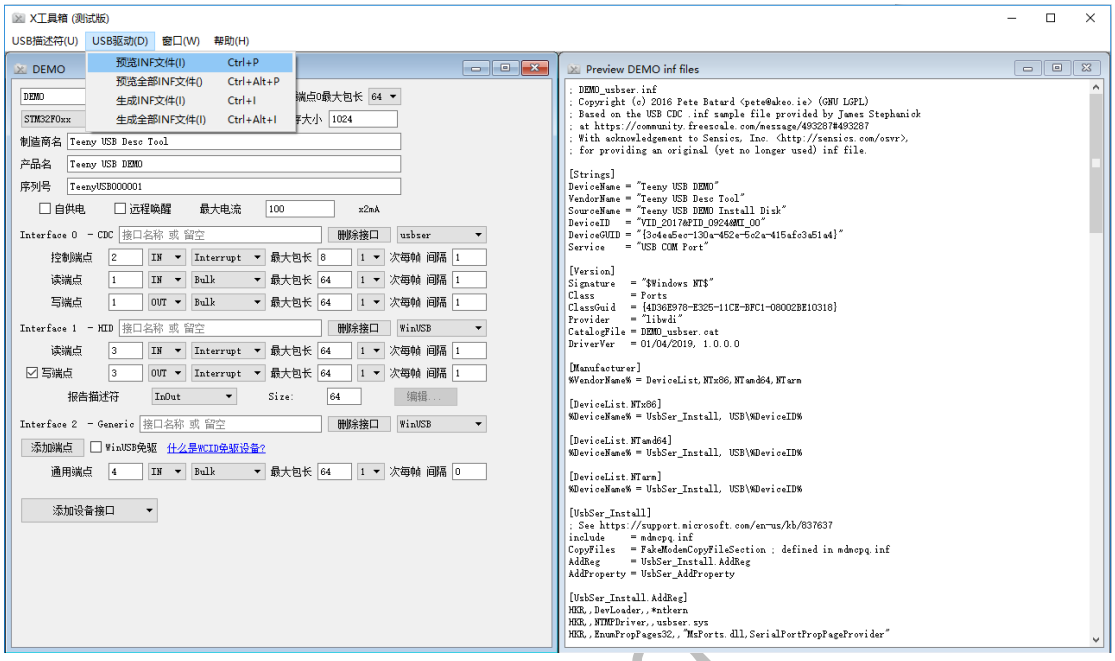


图25 TeenyDT 生成 INF 文件

USB 驱动菜单功能见下表：

表14 TeenyDT 驱动菜单

菜单名称	功能
预览INF文件	查看当前设备的INF文件，如果一个设备的多个接口配置了不同类型的驱动，则会为每种不同类型的驱动生成一个单独的INF文件
预览全部INF文件	查看所有设备的INF
生成INF文件	生成当前设备的INF文件，并生成对应的自签名cat文件
生成全部INF文件	生成全部设备的INF文件，并生成对应的自签名cat文件

3.11 描述符小结

本章介绍了 USB 设备的基本描述符，一些与设备类型相关的描述符会在相关设备例程中做简要介绍。可以用描述符工具 TeenyDT 来简化描述符的编写过程，后面例程中的描述符都是采用 TeenyDT 生成的。

4 STM32 USB FS 模块

USB FS 模块主要应用在 STM32F0，STM32F103，STM32F3 系列芯片上，不同的芯片上 FS 模块的版本有所不同，在操作上有少许差异。

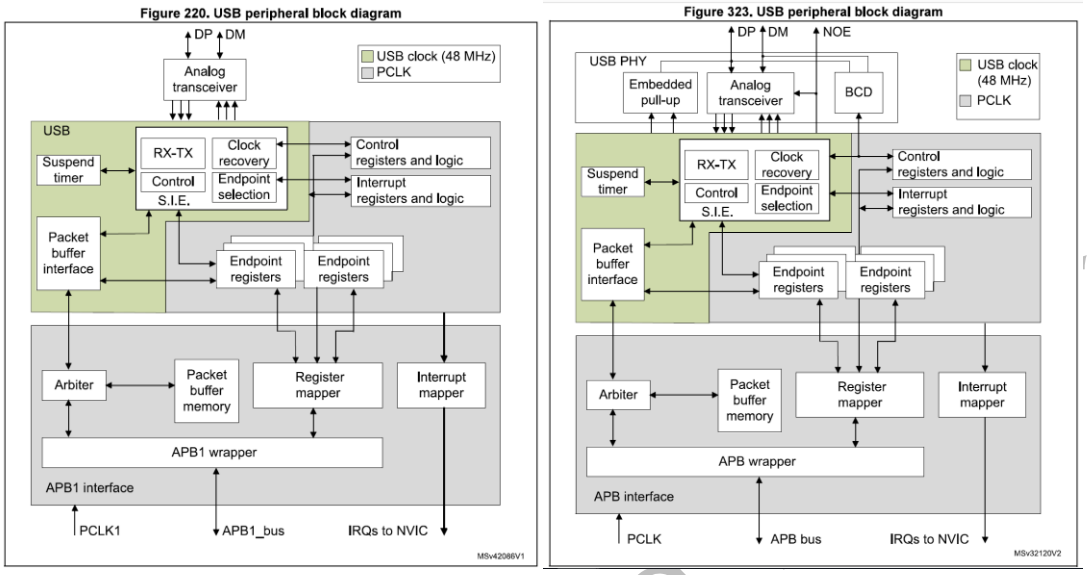


图26　　F1 与 F0 系列的 FS 模块对比图（来自 STM32 参考手册）

从上图中可以看到，应用程序软件通过 APB（本文将 APB1 与 APB 都统称为 APB）总线与 USB 模块通讯，寄存器和包缓存（Packet buffer memory）都通过 APB wrapper 来进行访问。当 APB 和 USB 模块都要访问包缓存时，由仲裁器（Arbiter）来决定谁能访问。在 ST 官方的固件库中，packet buffer memory 又叫做 Packet Memory Area，即 PMA，后续都用 PMA 来指代这块区域。

4.1 USB 核心初始化操作

STM32 的 USB FS 模块在使用前需要做一些初始化配置，本节介绍这部分内容。USB 功能初始化部分的代码在 TeenyUSB 协议栈 stm32f0_init.c 和 stm32f1_init.c 文件中实现。时钟初始化部分的代码在 system_stm32f0xx.c 和 system_stm32f10x.c 文件中实现。

4.1.1 时钟设置

在官方的 HAL 库中，时钟的配置的代码在 SystemClock_Config 函数中，由 CubeMX 自动生成。TeenyUSB 采用了老版本库的做法，使用 system_init.c 文件，在 SystemInit 函数中对时钟进行配置。HAL 库中对时钟的配置带超时检测机制，需要 systick 配合使用，因此 HAL 库在调用其他库函数前需要对 systick 进行配置。TeenyUSB 的时钟配置代码也是来自于 CubeMX 自动生成的代码，去掉了超时等待部分内容。

4.1.1.1 F1 系列芯片时钟设置

USB FS 模块需要 48MHz 的时钟来工作，下图是 F1 系列芯片中的 USB 时钟关系树。

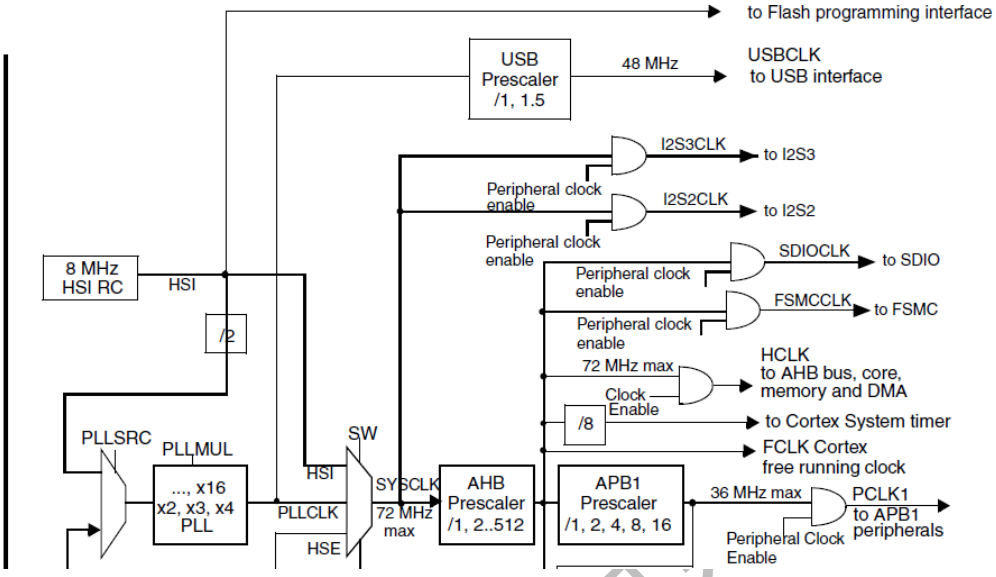


图27　　F1 系列 USB 时钟（来自 STM32 参考手册）

上图中可以看到，USB 时钟来自 PLLCLK，并且只能对 PLLCLK 进行 1 分频或 1.5 分频，这就要求 PLLCLK 只能是 72MHz 或是 48MHz。在 F1 系列芯片上使用 USB 功能，需要打开 PLL 功能，根据外部晶振的频率设置 PLL 参数，使得 PLLCLK 为 72MHz 或是 48MHz。然后打开 USB 时钟和相应的 IO 时钟。SystemInit 函数中将主时钟配置为 PLL，72MHz，TeenyUSB 的 stm32f1_init.c 文件初始化代码中打开 USB 时钟。分频默认为 1.5，因此这里没有配置分频系数。如果 PLL 时钟为 48MHz，需要配置分频系数为 1。

4.1.1.2 F0 系列芯片时钟设置

下图是 F0 系列芯片中的 USB 时钟关系树：

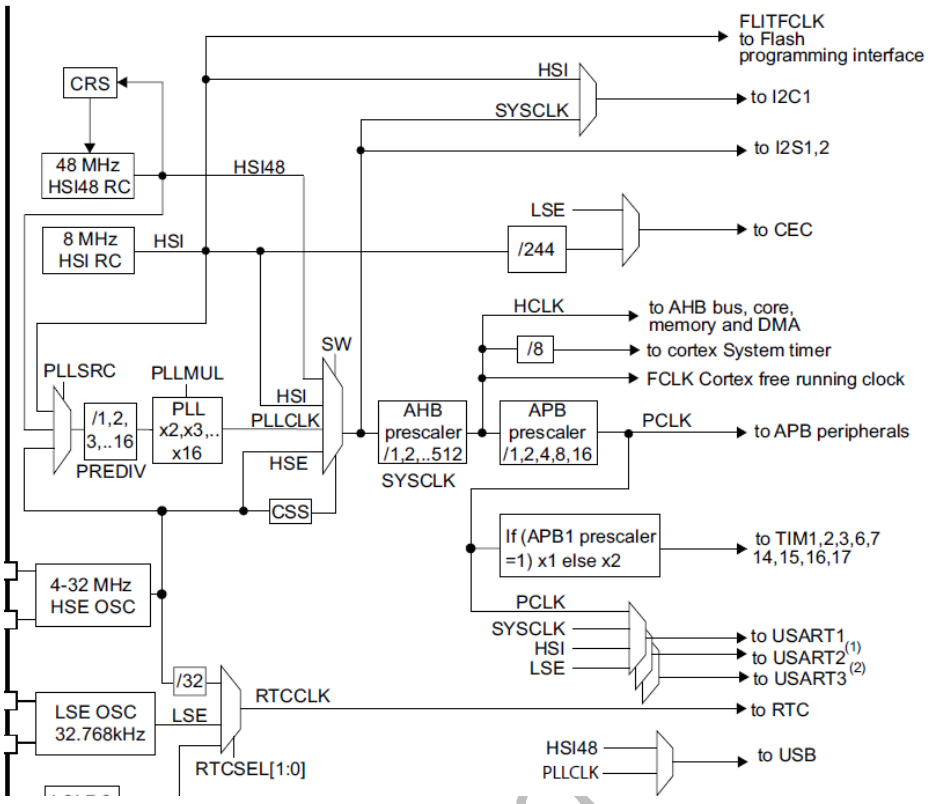


图28　　F0 系列 USB 时钟（来自 STM32 参考手册）

在 F0 芯片中 USB 时钟有两个来源，一个是 PLLCLK，一个是 HSI48。如果采用 PLLCLK，那么 PLL 时钟只能配置为 48MHz。如果采用 HSI48，需要将 HSI48 的 CRS 配置为使用 USB 的 SOF 包来校准。TeenyUSB 中 F0 芯片 USB 时钟配置如下：

```
#if defined USB_CLOCK_SOURCE_CRS
  RCC->CFGR3 &= ~RCC_CFGR3_USBSW;
  RCC->APB1ENR |= RCC_APB1ENR_CRSEN;
  CRS->CFGR &= ~CRS_CFGR_SYNCSRC;
  CRS->CFGR |= RCC_CRS_SYNC_SOURCE_USB;
  CRS->CR |= (CRS_CR_AUTOTRIMEN | CRS_CR_CEN);
#else
  // otherwise use pll clk
  RCC->CFGR3 |= RCC_CFGR3_USBSW_PLLCLK;
#endif
```

根据定义选择使用 PLL 时钟还是 HSI48 时钟，如果使用 HSI48 的时钟，打开 CRS 功能，并将 CRS 来源设置为 USB。当使用 HSI48 时，芯片不用外接晶振，能够减少器件成本。

4.1.2 IO 设置

USB FS 模块通讯需要 D+和 D-两个信号，在 F1 和 F0 系列芯片中，当 USB 时钟打开后，D+和 D-对应的 IO 会自动被 USB 模块接管。F1 芯片中的 D+没有内部上拉功能，因此需要外接一个 1.5K 的上拉电阻到 3.3V。F0 芯片中的 D+包含一个内部上拉电阻，并且可以通过寄存器进行设置。在 F1 芯片中，TeenyUSB 没有为 D+增加额外的上拉电阻控制引脚，而是

通过将 D+引脚设置为低电平来模拟断开。代码如下：

```
  // PA_12 output mode: OD = 0
  GPIOA->CRH |= GPIO_CRH_CNF12_0;
  GPIOA->CRH &= (~GPIO_CRH_CNF12_1);
  GPIOA->CRH |= GPIO_CRH_MODE12;// PA_12 set as: Output mode, max speed 50 MHz.
  GPIOA->BRR = GPIO_BRR_BR12;
```

在 F0 芯片中，TeenyUSB 根据配置，选择使用内置的上拉电阻还是 IO 模拟来实现断开功能。代码如下：

```
#ifdef USB_FS_INTERNAL_PULLUP
  // disable interal pull up resistor
  GetUSB(dev)->BCDR&=~USB_BCDR_DPPU;
#else
  // PA12 = PushPull = 0
  GPIOA->OTYPER |= GPIO_OTYPER_OT_12;
  GPIOA->MODER &= ~GPIO_MODER_MODER12;
  GPIOA->MODER |= GPIO_MODER_MODER12_0;
  GPIOA->OSPEEDR |= GPIO_OSPEEDR_OSPEEDR12;
           GPIOA->BRR = GPIO_BRR_BR_12;
 #endif
```

4.1.3 中断设置

为了在中断中响应 USB 的事件，需要开启 USB 相关的中断服务。在 F0 系列的芯片中，只有一个 USB_IRQn，打开即可。在 F1 系列芯片中，USB 的中断分为 LP 和 HP 两种。LP 是 Low Priority（低优先级）的简写，HP 是 High Priority（高优先级）的简写。在 LP 中断中，处理 USB 的所有中断事件，在 HP 中断中，处理端点事件。F1 中 USB 中断设置和处理的代码如下：

```
#if defined(HAS_DOUBLE_BUFFER)
  NVIC_EnableIRQ(USB_HP_CAN1_TX_IRQn);
#endif
  NVIC_EnableIRQ(USB_LP_CAN1_RX0_IRQn);

void USB_HP_CAN1_TX_IRQHandler(void)
{
  uint16_t wIstr;
  tusb_device_t* dev = &tusb_dev;
  while ((wIstr = GetUSB(dev)->ISTR ) & USB_ISTR_CTR){
     GetUSB(dev)->ISTR = (uint16_t)(USB_CLR_CTR);
     tusb_ep_handler(dev, wIstr & USB_ISTR_EP_ID);
  }
}

void USB_LP_CAN1_RX0_IRQHandler(void)
{
```

```
  tusb_device_t* dev = &tusb_dev;
  uint16_t wIstr;
  while ((wIstr = GetUSB(dev)->ISTR ) & USB_ISTR_CTR){
     GetUSB(dev)->ISTR = (uint16_t)(USB_CLR_CTR);
     tusb_ep_handler(dev, wIstr & USB_ISTR_EP_ID);
  }

  if (wIstr & USB_ISTR_RESET) {
    GetUSB(dev)->ISTR = (USB_CLR_RESET);
    GetUSB(dev)->BTABLE = (BTABLE_ADDRESS);
    tusb_reconfig(dev);
    GetUSB(dev)->DADDR = (0 | USB_DADDR_EF);
  }
  ...
}
```

在 F1 系列芯片中，如果使用了双缓存功能，那么就打开 USB_HP 中断，并在 HP 中断中处理端点事件。如果没有使用双缓存，就只使用 USB_LP 中断处理所有事件。

4.1.4 USB 模块设置

USB FS 模块比较简单，USB 核心寄存器只有 5 个。通过 CNTR 对 USB 模块进行复位，通过 ISTR 查询中断标志，判断中断来源。通过 BTABLE 寄存器设置端点的数据缓存区地址。DADDR 寄存器不在初始化时配置，在收到 RESET 信号和设置地址请求时配置 DADDR 寄存器。F1 与 F0 的 USB 模块基本功能设置代码相同，代码如下：

```
  RCC->APB1ENR |= RCC_APB1ENR_USBEN;

  GetUSB(dev)->CNTR = (USB_CNTR_FRES);
  GetUSB(dev)->CNTR = (0);

  // wait reset finish
  while (!((GetUSB(dev)->ISTR) & USB_ISTR_RESET));

  GetUSB(dev)->ISTR = (0);
  GetUSB(dev)->BTABLE = (BTABLE_ADDRESS);
  GetUSB(dev)->CNTR = (IMR_MSK);
```

处理流程：打开 USB 模块时钟，复位 USB 模块，等待复位完成，清除所有中断标志，设置内存描述表的起始地址，打开需要处理的中断标志。

4.1.5 TeenyUSB 中的核心初始化

F0、F1 系列芯片中 USB 核心完整的初始化代码(tusb_open_device)如下：

```
void tusb_open_device(tusb_device_t* dev){
```

```
  tusb_disconnect(dev);
  tusb_delay_ms(20);
#if defined(STM32F0)
#if defined USB_FS_CLOCK_SOURCE_CRS
  RCC->CFGR3 &= ~RCC_CFGR3_USBSW;
  RCC->APB1ENR |= RCC_APB1ENR_CRSEN;
  CRS->CFGR &= ~CRS_CFGR_SYNCSRC;
  CRS->CFGR |= RCC_CRS_SYNC_SOURCE_USB;
  CRS->CR |= (CRS_CR_AUTOTRIMEN | CRS_CR_CEN);
#else
  // otherwise use pll clk
  RCC->CFGR3 |= RCC_CFGR3_USBSW_PLLCLK;
#endif
#endif
  RCC->APB1ENR |= RCC_APB1ENR_USBEN;
  GetUSB(dev)->CNTR = (USB_CNTR_FRES);
  GetUSB(dev)->CNTR = (0);
  // wait reset finish
  while (!((GetUSB(dev)->ISTR) & USB_ISTR_RESET));
  GetUSB(dev)->ISTR = (0);
  GetUSB(dev)->BTABLE = (BTABLE_ADDRESS);
  GetUSB(dev)->CNTR = (IMR_MSK);
#if defined(STM32F1)
  // PA12 = Input
  RCC->APB2ENR |= RCC_APB2ENR_IOPAEN;
  GPIOA->CRH |= GPIO_CRH_CNF12_0;
  GPIOA->CRH &= ~GPIO_CRH_CNF12_1;
  GPIOA->CRH &= ~GPIO_CRH_MODE12;
#if defined(HAS_DOUBLE_BUFFER)
  NVIC_EnableIRQ(USB_HP_CAN1_TX_IRQn);
#endif
  NVIC_EnableIRQ(USB_LP_CAN1_RX0_IRQn);
#endif
#if defined(STM32F0)
#ifdef USB_FS_INTERNAL_PULLUP
  // USE the interal pull up resistor
  GetUSB(dev)->BCDR|=USB_BCDR_DPPU;
#else
  RCC->AHBENR |= RCC_AHBENR_GPIOAEN;
  // PA12 = In float
  GPIOA->PUPDR &= ~GPIO_PUPDR_PUPDR12;
  GPIOA->MODER &= ~GPIO_MODER_MODER12;
#endif
  NVIC_EnableIRQ(USB_IRQn);
```

```
#endif
}
```

上面代码主要功能是设置 IO，复位 USB 模块，打开对应的中断。不同的是在 F1 系列的代码中，会根据是否有双缓存端点来选择是否打开 USB_HP 中断。在 F0 系列的代码中会根据时钟配置来选择不同的 USB 时钟源。在 F0 中会根据上拉电阻的配置选择使用内部上拉电阻还是外部的。在启动 USB 之前会先调用 tusb_disconnect 函数断开 USB 端口，这是为了在 USB 接口产生一次重新连接事件，让主机对设备重新进行枚举。

完成初始化之后，USB 设备插上主机后产生 USB Reset 中断，在 Reset 中断中完成端点的配置，启用端点数据处理功能。下面介绍端点配置相关内容。

4.1.6 HAL 库中的核心初始化

下面是 HAL 库中 USB 模块核心初始化的代码：

```
HAL_StatusTypeDef USB_DevInit (USB_TypeDef *USBx, USB_CfgTypeDef cfg){
  /* Prevent unused argument(s) compilation warning */
  UNUSED(cfg);
  /* Init Device */
  /*CNTR_FRES = 1*/
  USBx->CNTR = USB_CNTR_FRES;
  /*CNTR_FRES = 0*/
  USBx->CNTR = 0;
  /*Clear pending interrupts*/
  USBx->ISTR = 0;
  /*Set Btable Address*/
  USBx->BTABLE = BTABLE_ADDRESS;
  /* Enable USB Device Interrupt mask */
  USB_EnableGlobalInt(USBx);
  return HAL_OK;
}
```

4.2 USB 端点寄存器操作

USB FS 模块中与端点相关的寄存器只有一个 EPnR，寄存器内容见下图：

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
CTR_RX	DTOG_RX	STAT_RX[1:0]		SETUP	EP TYPE[1:0]		EP_KIND	CTR_TX	DTOG_TX	STAT_TX[1:0]		EA[3:0]			
rc_w0	t	t	t	r	rw	rw	rw	rc_w0	t	t	t	rw	rw	rw	rw

图29　　EPnR 寄存器（来自 STM32 参考手册）

在 STM32 的参考手册中对个字段的值有详细的说明，这里只介绍一下这个寄存器比较特殊的一些操作，便于理解代码。这个寄存器中有 4 类标志，分别是只能清零（rc_w0），写 1 翻转（t），只读（r），读写（rw）。

4.2.1 只能清零（rc_w0）

只能清零的字段，写 1 不会改变其内容，写 0 会清除。因此当我们不希望改变这个字段的内容时，可以将这个字段的值或上 1。

4.2.2 写 1 翻转（t）

对于写 1 翻转的字段，如果我们希望将其状态改变，那就写 1；如果不希望改变，就写 0。异或操作正好可以实现这样的功能，先将寄存器中当前的值读出，与我们希望设置的值进行异或。异或运算相同的值结果为 0，不同的值结果为 1。不同的值即为需要翻转的字段，将异或的结果写入寄存器中，正好将需要翻转的字段设置为期望的值。

HAL 库中对写 1 翻转字段的处理方式是逐位进行的，代码如下：

```
#define PCD_SET_EP_RX_STATUS(USBx, bEpNum,wState) {\
    register uint16_t _wRegVal;   \
    \
    _wRegVal = PCD_GET_ENDPOINT((USBx), (bEpNum)) & USB_EPRX_DTOGMASK;\
    /* toggle first bit ? */  \
    if((USB_EPRX_DTOG1 & (wState))!= 0U) \
    {                                                                   \
      _wRegVal ^= USB_EPRX_DTOG1;  \
    }                                                                   \
    /* toggle second bit ? */  \
    if((USB_EPRX_DTOG2 & (wState))!= 0U) \
    {                                                                   \
      _wRegVal ^= USB_EPRX_DTOG2;  \
    }                                                                   \
    PCD_SET_ENDPOINT((USBx), (bEpNum), (_wRegVal | USB_EP_CTR_RX|USB_EP_CTR_TX)); \
  } /* PCD_SET_EP_RX_STATUS */
```

在上面的代码中逻辑表如下：

表15　　HAL 库翻转逻辑

新值	原始值	写入的值	翻转后的值
1	0	1	1
1	1	0	1
0	0	0	0
0	1	1	0

当新设置的值为 1 时，将原始值与 1 异或后写入寄存器中，翻转后的值为 1。当新设置的值为 0 时，原始值不变，再次写入寄存器中，这样翻转后的值就为 0 了。

根据上表，写入的值 = 新值 xor 原始值，这个也是 TeenyUSB 中对写 1 翻转字段的操作的方式。采用异或的方式可以一次性将所有需要的值写入寄存器中，不用逐位判断。

4.2.3 只读和读写（r，rw）

只读的字段位写任何值都不会改变其状态，在操作时可以忽略。读写的字段直接将需要的值写入即可。

4.2.4 TeenyUSB 中 USB 端点寄存器配置

在 TeenyUSB 中，通过一个宏对 USB 端点寄存器的所有字段一次性进行配置，其代码如下：

```
// Macro used to build ep setting, Tx/Rx CTR not changed, Tx/Rx TOG bits forced to 0
#define  BUILD_EP_SETTING(dev, bEpNum, type, rx_state, tx_state, kind)\
  /* Get end point current value */  \
 (( PCD_GET_ENDPOINT(GetUSB(dev), bEpNum)      \
  /* TOG bits is write 1 to toggle, mask them with current value */\
  /* then write them back will set the TOG bits to 0*/\
   & (USB_EP_DTOG_RX | USB_EP_DTOG_TX | USB_EPTX_STAT | USB_EPRX_STAT) \
  /* tx/rx state is wirte 1 to toggle, xor them with current value */ \
  /* then write them back, will set them to desired value */ \
   ^ ((tx_state) | (rx_state)) ) \
  /* Write 1 to CTR has no effect, ored them with 1 makes no change */ \
  | (USB_EP_CTR_RX|USB_EP_CTR_TX)     \
  /* Write ep num, tpye and kind to the register */ \
  | (bEpNum) | (type) | (kind) )
```

上面的代码中，将 DTOG_TX 和 DTOG_RX 字段设置为 0，将 TX_STAT 和 RX_STAT 设置为宏参数指定的值。DTOG 和 STAT 字段都是写 1 翻转的，因此先用掩码与原来的值进行与操作，得到原来的 DTOG 和 STAT 字段的值。再用新设置的值与原来寄存器中的值进行异或得到需要翻转的值，最后将需要翻转的值写入寄存器中。

CTR_RX 和 CTR_TX 字段属性是写 0 清零，在这里没有对其进行修改，因此或上了 CTR_RX 和 CTR_TX 的掩码，让其值为 1，不对寄存器这两个字段的内容做修改。

EA，EP_TYPE，EP_KIND 字段属性是读写，这里直接写入了新的值。

为了简化操作，TeenyUSB 对 BUILD_EP_SETTING 宏做了进一步封装：

```
// Init ep in tx/tx mode, and set tx to NAK, set rx to VALID
#define  INIT_EP_BiDirection(dev, bEpNum, type) \
  PCD_SET_ENDPOINT(GetUSB(dev), bEpNum,  BUILD_EP_SETTING(dev, bEpNum, type, USB_EP_RX_VALID,
USB_EP_TX_NAK, 0) )

// Init ep in tx mode, and set tx to NAK state
#define  INIT_EP_TxOnly(dev, bEpNum, type) \
  PCD_SET_ENDPOINT(GetUSB(dev), bEpNum,  BUILD_EP_SETTING(dev, bEpNum, type, USB_EP_RX_DIS,
USB_EP_TX_NAK, 0) )

// Init ep in rx mode, and set rx to VALID state
#define  INIT_EP_RxOnly(dev, bEpNum, type) \
```

```
  PCD_SET_ENDPOINT(GetUSB(dev), bEpNum,  BUILD_EP_SETTING(dev, bEpNum, type, USB_EP_RX_VALID,
USB_EP_TX_DIS, 0) )

// Init ep in tx double buffer mode, and set tx to NAK state
#define  INIT_EP_TxDouble(dev, bEpNum, type) \
  /* Keep to Rx/Tx TOG bit to 0, the double buffer tx ep is NAK even tx state is VALID */ \
  PCD_SET_ENDPOINT(GetUSB(dev), bEpNum,  BUILD_EP_SETTING(dev, bEpNum, type, USB_EP_RX_DIS,
USB_EP_TX_VALID, USB_EP_KIND) )

// Init ep in rx double buffer mode, and set rx to VALID state
#define  INIT_EP_RxDouble(dev, bEpNum, type) \
  PCD_SET_ENDPOINT(GetUSB(dev), bEpNum,  BUILD_EP_SETTING(dev, bEpNum, type, USB_EP_RX_VALID,
USB_EP_TX_DIS, USB_EP_KIND) \
  /* Toggle the SW_BUF bits, to make sure the rx state is real valid */ \
  ^ USB_EP_DTOG_TX )
```

上面的代码中，对于 OUT（RX）端点，初始化为 Valid 状态，对于 IN（TX）端点，初始化为 NAK 状态。如果端点只有 IN（TX）方向，OUT（RX）方向设置为 Disable 状态；如果端点只有 OUT（RX）方向，IN（TX）方向设置为 Disable 状态。端点配置后还不能正常工作，还需要在端点的 PMA 描述表中设置端点的缓存地址，最大包长和收发数据长度。关于 PMA 的详细内容在下节介绍。

同步端点（Isochronous）只能工作在双缓存模式，一个方向独占一个端点寄存器。同步端点只有 Enable 和 Disable 的区别，NAK 状态的 IN（TX）端点依然会发送数据。同步 IN（TX）端点通过实际数据长度来区分有效数据和无效数据，详细内容见同步端点的流控。

4.2.5 HAL 库中的端点寄存器配置

HAL 库中的端点配置在 USB_ActivateEndpoint 函数中实现，采用了多个宏对端点寄存器配置进行封装，见下表：

表16　　HAL 库端点配置宏

端点设置宏	功能
PCD_SET_EPTYPE	设置EP类型
PCD_SET_EP_ADDRESS	设置EP地址
PCD_CLEAR_TX_DTOG	清除EP DTOG_TX标志
PCD_CLEAR_RX_DTOG	清除EP DTOG_RX标志
PCD_SET_EP_TX_STATUS	设置EP TX状态
PCD_SET_EP_RX_STATUS	设置EP RX状态
PCD_SET_EP_DBUF	设置EP为双缓冲模式

4.3 PMA 操作

USB FS 模块与芯片内核通过一片叫做 PMA 的内存区域进行数据交互，端点在收发数据前需要配置 PMA。当需要发送数据到 USB 总线上时，先将数据复制到 PMA 的内存区域，然后设置相关的标志位，USB 模块接收到响应的指令后开始发送数据。发送完成后，通过中断通知应用程序。当需要从 USB 总线上接收数据时，配置接收数据的地址和相关的标志位。

USB 模块在收到数据后自动将数据复制到 PMA 指定区域并通知应用程序。应用程序收到通知后，将数据从 PMA 内存区域中读出。

为了更好的说明 USB 模块和应用程序与 PMA 内存的交互方式，这里将分为两部来介绍 PMA 内存。先从 USB 模块的角度来介绍 PMA 内存的访问，再从应用程序的角度来介绍。不同型号的芯片中 PMA 内存大小不同，从 512 至 1024 字节不等。

4.3.1 USB 模块访问 PMA

下图是 PMA 结构图：

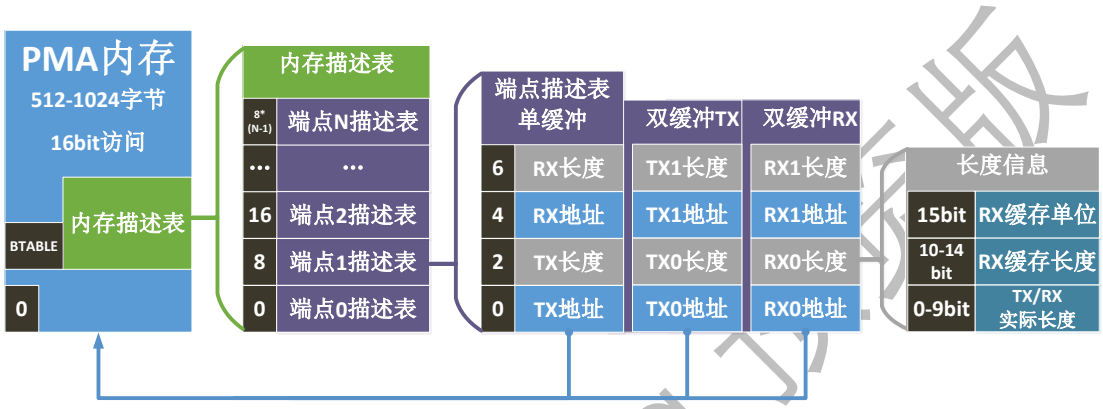


图30　　USB FS 设备 PMA 结构

USB FS 模块以 16 位方式对 PMA 进行访问，USB 模块访问时 PMA 时，地址从 0 开始。先通过 BTABLE 寄存器在 PMA 内存中找到内存描述表的位置。BTABLE 寄存器的值表示内存描述表的起始地址，这个地址的范围必须在 PMA 内存中并且 8 字节对齐。内存描述表的大小是动态的，由使用的最大端点号决定。如果只用了 0 和 1 端点，内存描述表只需要 16 字节。如果只用了 0 和 7 端点，那么内存描述表需要 64 字节。虽然 1-6 端点没有用到，但是 7 号端点的描述表是从 56 字节开始的，会占用 64 字节的空间。

根据端点类型的不同，端点描述表有三种类型，分别是单缓冲、双缓冲发送（TX）、双缓冲接收（RX）。当端点配置为 Bulk 双缓冲或是同步模式时，端点描述符工作在双缓冲模式。由于双缓冲的一个方向会占用全部 8 字节的端点描述表，因此双缓冲端点和同步端点只能是单向的。

当 USB 模块要发送数据时，通过端点号和 BTABLE 获取到当前端点描述表。再根据描述表中 TX 地址和 TX 实际长度，从 PMA 内存中取出数据进行发送。

当 USB 模块接收到数据时，通过端点号和 BTABLE 获取到当前端点描述表。再根据描述表中 RX 地址和 RX 缓存长度进行数据接收，并将此次接收数据的实际长度放到 RX 实际长度字段中。

4.3.2 应用程序访问 PMA

STM32 上的应用程序通过 APB 总线访问 PMA，PMA 映射在 0x40006000 地址处。不同的芯片，PMA 内存的挂载方式有所不同。F0 这类的芯片，从 APB 访问 PMA 的方式与 USB 模块访问方式相同，地址空间除了起始地址不同，其它都一样。F1 这类芯片，从 APB 访问 PMA 需要按照 32 位对齐，16 位读写方式访问。

什么叫 32 位对齐 16 位访问？又是 32 位又是 16 位的，那么到底是 32 位还 16 位呢。

16 位访问的意思是，一次只能读或写 16 位的数据，不能是 8 位也不能是 32 位。用汇编的话来讲叫做只能用 LDRH 和 STRH 这样的指令来操作，用 C 语言的话来讲叫做只能用 uint16_t 这类 16 位的类型来操作，比如`*(uint16_t*)addr = xxx;`。32 位对齐的意思是，操作数据的地址必须是 32 位对齐的，即前面代码中的 addr 必须是 32 位对齐的。所以 32 位对齐是对地址值的限制，16 位访问是对这个地址上操作的限制。

下图是两种不同芯片上的 PMA 在 APB 总线上的映射方式：

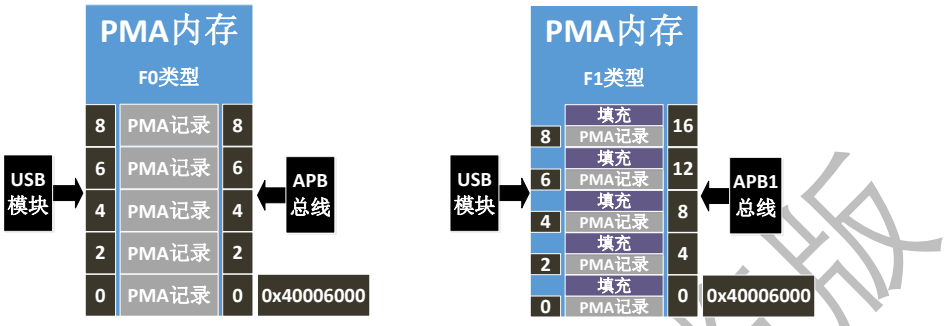


图31　　不同芯片中的 PMA 映射方式对比

4.3.3 USB FS 的 PMA 操作

PMA 配置的内容是将地址信息和长度信息写入到对应的端点描述表中。对于 IN 端点，需要配置缓存的地址，在发送数据前，将数据复制到 PMA 中，设置实际长度，然后将对应的端点设置为 Valid 状态，数据发送完成后触发 CTR_TX 中断。

对于 OUT 端点，需要配置缓存的地址，配置接收缓存长度，然后将对应的端点设置位 Valid 状态。接收到数据后，触发 CTR_RX 中断，在中端处理函数中检查对应端点的实际长度，将数据从 PMA 缓存中复制出来。一般情况下端点的 PMA 地址信息和缓存长度信息在初始化后不会改变。因此在 TeenyUSB 中，PMA 的地址设置和缓存长度设置与端点配置一起进行设置。下面先介绍 PMA 的配置，然后再介绍 PMA 中数据的读写。

4.3.3.1 TeenyUSB 对 PMA 的定义

TeenyUSB 中对 PMA 内存结构的定义如下：

```
#if defined(STM32F0)
// Define PMA buffer layout for STM32F0xx
typedef struct _pma_data{
  uint16_t data;
}pma_data;
typedef struct _pma_record{
  uint16_t    addr;       // TX/RX address
  uint16_t    cnt:10;     // TX/RX data count
  uint16_t    block:6;    // RX buffer size
}pma_record;

#elif defined(STM32F1)
// Define PMA buffer layout for STM32F1xx
```

```
typedef struct _pma_data{
  uint16_t data;
  uint16_t padding;
}pma_data;
typedef struct _pma_record{
  uint16_t    addr;       // TX/RX address
  uint16_t padding1;
  uint16_t    cnt:10;     // TX/RX data count
  uint16_t    block:6;    // RX buffer size
  uint16_t padding2;
}pma_record;
#endif
typedef union _pma_ep_desc {
  struct _normal{
    pma_record     tx;
    pma_record     rx;
  }normal;
  struct _dbl_tx{
    pma_record     tx0;
    pma_record     tx1;
  }dbl_tx;
  struct _dlb_rx{
    pma_record     rx0;
    pma_record     rx1;
  }dbl_rx;
}pma_ep_desc;
```

在 F0 系列芯片中，pma_data 和 pma_record 都是按照 USB 模块的访问模式来定义的，而在 F1 系列芯片中，pma_data 和 pma_record 中定义了 padding 字段，用来填充 32 位对齐 16 位访问模式下的“空隙”。地址信息和长度信息合成了一个 pma_record，两个 pma_record 组成一个完整端点信息 pma_ep_desc。由于端点可能工作在普通模式和双缓冲模式，所以端点描述表 pma_ep_desc 结构体分别由 normal、dbl_tx、dbl_rx 这三种端点类型组成。通过 PMA_DESC 宏或 EPT 宏可以得到端点描述表，实现如下：

```
#define GetUSB(dev)     (USB)
#define PMATable(dev)   ( (pma_data*) ((uint32_t)GetUSB(dev) + 0x400U) )
#define GetBTABLE(dev)  (BTABLE_ADDRESS)
#define PMA_DESC(dev)   ((pma_ep_desc*) (GetBTABLE(dev) + PMATable(dev)) )
#define EPT(dev, bEpNum)   (PMA_DESC(dev)[bEpNum])
```

通过 GetUSB 宏得到芯片中的 USB 模块地址，由于在 STM32F1 和 F0 系列中，只有一个 USB 模块，所以 GetUSB 宏直接返回了预定义的 USB 宏。再通过 PMATable 宏计算出 PMA 的映射地址。GetBTABLE 宏，得到内存描述表的偏移地址。通过 PMA 映射地址和内存描述表偏移地址计算出端点描述表的地址。一般情况下设置的 BTABLE 值不会变化，所以 GetBTABLE 直接返回了一个常量值，没有从 BTABLE 寄存器中去读取设置的值。这样整个 PMA_DESC 宏都是常量计算，在编译时就能确定。通过 PMA_DESC 宏可以获取端点的描述表，例如 PMA_DESC(dev)[1]这样得到端点 1 对应的描述表。EPT 宏对 PMA_DESC 做了进一

步封装，通过 EPT(dev,1)能方便的获得对应的端点描述表。

4.3.4 PMA 地址与缓存长度配置

4.3.4.1 TeenyUSB 处理方式

当端点为普通端点时，通过 EPT(dev, 1).normal.tx.addr 可以读取和设置端点 1 的发送缓存地址。如果是双缓存 OUT 端点，通过 EPT(dev, 1).dbl_tx.tx0.addr 可以读取和设置缓存 0 的发送地址，通过 EPT(dev, 1).dbl_tx.tx1.addr 可以读取和设置缓存 1 的发送地址。为了简化 dbl_tx.tx1.addr 这一串内容，定义了下面这些宏：

```
#define  tx_addr   normal.tx.addr
#define  tx_cnt    normal.tx.cnt
#define  rx_addr   normal.rx.addr
#define  rx_cnt    normal.rx.cnt
#define  rx_block  normal.rx.block

#define  tx0_addr   dbl_tx.tx0.addr
#define  tx0_cnt    dbl_tx.tx0.cnt
#define  tx1_addr   dbl_tx.tx1.addr
#define  tx1_cnt    dbl_tx.tx1.cnt

#define  rx0_addr   dbl_rx.rx0.addr
#define  rx0_cnt    dbl_rx.rx0.cnt
#define  rx0_block  dbl_rx.rx0.block

#define  rx1_addr   dbl_rx.rx1.addr
#define  rx1_cnt    dbl_rx.rx1.cnt
#define  rx1_block  dbl_rx.rx1.block
```

通过上面的宏，要设置普通端点的缓存地址时，使用 EPT(dev, bEpNum).tx_addr = addr。如果要配置双缓冲 RX 的地址，使用 EPT(dev, bEpNum).rx0_addr = addr0; EPT(dev, bEpNum) .rx1_addr = addr1;。为了进一步简化地址，长度的设置，定义了下面的这些宏：

```
#define  SET_TX_ADDR(dev, bEpNum, addr)  do { EPT(dev, bEpNum).tx_addr = (addr); }while(0)
#define  SET_RX_ADDR(dev, bEpNum, addr)  do { EPT(dev, bEpNum).rx_addr = (addr); }while(0)
#define  SET_DOUBLE_ADDR(dev, bEpNum, addr0, addr1) \
  do{\
    EPT(dev, bEpNum).tx0_addr = addr0;\
    EPT(dev, bEpNum).tx1_addr = addr1;\
  }while(0)
#define  SET_RX_CNT(dev, bEpNum, cnt)\
  do{\
      uint16_t block = (cnt)>62 ? ((((cnt)+31)/32) | (1<<5)): (((cnt)+1)/2);\
      EPT(dev, bEpNum).rx_cnt = 0;\
```

```
      EPT(dev, bEpNum).rx_block = block;\
  }while(0)
#define SET_TX_CNT(dev, bEpNum, cnt) \
  do{  EPT(dev, bEpNum).tx_cnt = (cnt); }while(0)
#define SET_DBL_TX_CNT(dev, bEpNum, cnt)\
  do{\
    EPT(dev, bEpNum).tx0_cnt = (cnt);\
    EPT(dev, bEpNum).tx1_cnt = (cnt);\
  }while(0)
#define SET_DBL_RX_CNT(dev, bEpNum, cnt)\
  do{\
    uint16_t block = (cnt)>62 ? ((((cnt)+31)/32) | (1<<5)): (((cnt)+1)/2);\
    EPT(dev, bEpNum).rx0_cnt = 0;\
    EPT(dev, bEpNum).rx0_block = block;\
    EPT(dev, bEpNum).rx1_cnt = 0;\
    EPT(dev, bEpNum).rx1_block = block;\
  }while(0)
```

在设置普通端点的发送缓存地址时，使用 SET_TX_ADDR(dev, 1, 0x100)，设置双缓存端点（OUT 端点）的地址时，使用 SET_DOUBLE_ADDR(dev, 1, 0x100, 0x140)。对于接收（OUT）端点，还需要设置接收端点的最大包长，这个值与端点描述符中的一致。接收包长采用的是 block 数量的方式来设置。block 字段一共 6 位，block 的最高位为 0 时，一个 block 2 字节；block 的最高位为 1 时，一个 block 32 字节。Block 的低五位表示实际的 block 数量，最大为 31。因此当最大包长大于 62 字节时，block 最高位置为 1，表示一个 block 32 字节。然后根据最大包长设置 block 数量。

4.3.4.2 HAL 库处理方式

HAL 库中的 PMA 配置在 USB_ActivateEndpoint 函数中实现，通过 PCD_xx_EP_xxx 相关的宏对端点的 PMA 进行配置。HAL 库与 TeenyUSB 中对端点 PMA 配置的宏见下表：

表17　　PMA 配置宏

功能	TeenyUSB使用的宏	HAL库使用的宏
设置TX地址	SET_TX_ADDR	PCD_SET_EP_TX_ADDRESS
设置RX地址	SET_RX_ADDR	PCD_SET_EP_RX_ADDRESS
设置TX大小	SET_TX_CNT	PCD_SET_EP_TX_CNT
设置RX大小	SET_RX_CNT	PCD_SET_EP_RX_CNT
设置双缓冲地址	SET_DOUBLE_ADDR	PCD_SET_EP_DBUF_ADDR
设置双缓冲TX大小	SET_DBL_TX_CNT	PCD_SET_EP_DBUF_CNT
设置双缓冲RX大小	SET_DBL_RX_CNT	PCD_SET_EP_DBUF_CNT

4.3.5 PMA 数据读写

PMA 数据的读写在 TeenyUSB 中分为了两步，第一步先获取端点的 PMA 记录，第二步根据记录来读写 PMA 中的数据。

4.3.5.1 获取 PMA 记录

应用程序要读写端点 PMA 中的数据，先根据端点号得到端点的描述表，然后根据端点的属性，从描述表中得到 PMA 记录。一个 PMA 记录包含了 PMA 地址信息和长度信息。如果是普通端点，根据端点方向获取 PMA 记录。如果是双缓冲端点，根据当前应用程序使用的 PMA ID 来获取 PMA 记录。为了简化 PMA 记录的获得，TeenyUSB 中定义了下面这些宏：

```
#define  PMA_TX(dev, bEpNum)   (&(PMA_DESC(dev)[bEpNum].pma_tx))
#define  PMA_RX(dev, bEpNum)   (&(PMA_DESC(dev)[bEpNum].pma_rx))
#define  PMA_TX0(dev, bEpNum)  (&(PMA_DESC(dev)[bEpNum].pma_tx0))
#define  PMA_TX1(dev, bEpNum)  (&(PMA_DESC(dev)[bEpNum].pma_tx1))
#define  PMA_RX0(dev, bEpNum)  (&(PMA_DESC(dev)[bEpNum].pma_rx0))
#define  PMA_RX1(dev, bEpNum)  (&(PMA_DESC(dev)[bEpNum].pma_rx1))
```

TeenyUSB 中，CTR_TX 中断处理函数 tusb_send_data_done 中获取 PMA 记录的代码如下：

```
if(IS_DOUBLE()){
  // double buffer bulk end point, toggle first then copy data
  if(ep->tx_pushed){ep->tx_pushed--;}
  if(ep->tx_pushed){
    // toggle it first, then do some copy
    TUSB_RX_DTOG(GetUSB(dev), EPn, EP);
  }
  pma = (EP & USB_EP_DTOG_RX) ? PMA_TX0(dev, EPn) : PMA_TX1(dev, EPn);
}else if( IS_ISO() ){
  ep->tx_pushed = 0;
  pma = (EP & USB_EP_DTOG_TX) ? PMA_TX0(dev, EPn) : PMA_TX1(dev, EPn);
  pma->cnt = 0;
}else{
  ep->tx_pushed = 0;
  pma = PMA_TX(dev, EPn);
}
```

上面的代码在 CTR_TX 的处理函数中，即 IN 端点数据发送完成的处理函数。

对于双缓冲 IN 端点，DTOG_RX 字段指示应用当前使用的缓存。0 表示使用 TX0，1 表示使用 TX1。这里应为前面的代码做了一次寄存器翻转操作，但是判定的代码还是以前的值，所以逻辑是反着的。这样做可以减少一次读寄存器的操作。

对于同步 IN 端点，DTOG_TX 字段指示 USB 内核当前使用的缓存。0 表示内核使用 TX0，应用程序使用 TX1。1 表示内核使用 TX1，应用程序使用 TX0。同步端点强制使用双缓存，但是只使用一个 DTOG 标志。

对于普通端点，将对应端点的 PMA 记录赋值给 pma 变量，给后续的代码使用。TeenyUSB 中， CTR_RX 中断处理函数 tusb_recv_data 中获取 PMA 记录的代码如下：

```
if(DOUBLE_BUFF && (EP & (USB_EP_TYPE_MASK | USB_EP_KIND)) == (USB_EP_BULK | USB_EP_KIND)){
  // double buffer bulk end point
  // If rx count is not valid, freeze the DTOG, this will cause ep NAK
  if(ep->rx_count < ep->rx_size){
    TUSB_TX_DTOG(GetUSB(dev), EPn, EP);
```

```
      // If App last used buffer is 1, dev will fill data in 0
      // If App last used buufer is 0, dev will fill data in 1
      // We read the data filled by device
      pma = (EP & USB_EP_DTOG_TX) ? PMA_RX0(dev, EPn) :  PMA_RX1(dev, EPn);
    }
  }else if( ISO_EP && ((EP & USB_EP_TYPE_MASK) == USB_EP_ISOCHRONOUS ) ){
    // ISO out endpoint will always receive the new packet
    // if rx count is not valid, current packet in PMA buffer will be dropped
    if(ep->rx_count < ep->rx_size){
      pma = (EP & USB_EP_DTOG_RX) ? PMA_TX0(dev, EPn) : PMA_TX1(dev, EPn);
    }
  }else{
    pma = PMA_RX(dev, EPn);
  }
```

处理逻辑与 IN 端点的处理方式类似。最终得到端点对应的 PMA 记录。

4.3.5.2 读写 PMA 数据

TeenyUSB 中读写 PMA 数据的代码如下：

```
static void tusb_pma_tx(tusb_device_t* dev, pma_record* pma, const void* data, uint32_t len){
  uint32_t count = (len + 1) / 2;
  pma_data *dest = GetPMAAddr(dev, pma->addr);
  pma->cnt = len;
#ifdef ALIGNED
  const uint16_t *src = ( const uint16_t *)data;
  for (uint8_t i = 0; i < count; i++) {
    dest->data = *src;
    dest++;
    src++;
  }
#else
  const uint8_t *src = ( const uint8_t *)data;
  for (uint8_t i = 0; i < count; i++) {
    uint16_t v = src[0] | (src[1]<<8);
    src+=2;
    dest->data = v;
    dest++;
  }
#endif
}
// Copy data from PMA buffer, if memory is aligned, copy two by two
static uint32_t tusb_pma_rx(tusb_device_t* dev, pma_record* pma, void* data){
  uint32_t len = pma->cnt;
  pma_data *src = GetPMAAddr(dev, pma->addr);
```

```
  uint32_t count = (len+1)/2;
#ifdef ALIGNED
  uint16_t *dest = (uint16_t *) data;
  for (uint32_t i = 0; i < count; i++) {
    *dest = src->data;
    dest++;
    src++;
  }
#else
  uint8_t *dest = (uint8_t *) data;
  for (uint32_t i = 0; i < count; i++) {
    uint16_t v = src->data;
    *dest = (uint8_t)v;
    dest++;
    v>>=8;
    *dest = (uint8_t)v;
    dest++;
    src++;
  }
#endif
  return len;
}
```

如果能够保证数据都是 2 字节对齐的，会使用 2 字节同时复制的代码，如果不能保证，使用普通的处理代码，将 2 次读写的内容合并成 2 字节与 PMA 进行交互。GetPMAAddr 宏定义如下：

```
#define  PMATable(dev)     ( (pma_data*) ((uint32_t)GetUSB(dev) + 0x400U) )
#define  GetPMAAddr(dev, offset)    \
  (PMATable(dev) + ( (offset) / sizeof(((pma_data*)0)->data)) )
```

根据 USB 模块的 PMA 地址和偏移值计算出 PMA 的实际地址，然后将其转成 pma_data 格式。通过 pma_data 类型的 data 字段进行读写操作。

4.4 端点初始化

完整的端点初始化包含端点配置和端点 PMA 配置两部分内容，在 TeenyUSB 中，端点的初始化是在接收到 Reset 事件后一次性完成的。在 HAL 库中，在 USB 设备初始化时配置了端点的 PMA 参数，在 Reset 事件中初始化了端点 0，在设置配置（Set Configuration）请求中，初始化了设备类相关的端点。

4.4.1 TeenyUSB 端点初始化

前面介绍的 TeenyDT 工具除了能够生成描述符外，还可以生成端点初始化代码。USB FS 模块的端点初始化需要完成端点类型配置，端点 PMA 缓存配置。TeenyDT 会根据 USB 设备中的描述符分析端点使用情况，自动计算出端点的初始化参数。

计算方式是，先统计端点使用的总数，计算内存描述表占用的空间，然后以此为基础添加各端点的内存起始地址。TeenyDT 生成的端点初始化代码如下：

```
//////////////////////////////////////////////////////////////////////
//// EndPoint for device1 define begin
//////////////////////////////////////////////////////////////////////
#define BULK_VID                                        0x0483
#define BULK_PID                                        0x1234
#define BULK_STRING_COUNT                               (5)
// Endpoint usage:
#ifdef BULK_BTABLE_ADDRESS
#undef BULK_BTABLE_ADDRESS
#endif
#define BULK_BTABLE_ADDRESS                             (0)
#define BULK_MAX_EP                                     (2)
#define BULK_EP_NUM                                     (BULK_MAX_EP + 1)
#define BULK_EP_BUF_DESC_TABLE_SIZE                     (8)

// PMA buffer reserved for buffer description table
#define BULK_USB_BUF_START                      (BULK_EP_BUF_DESC_TABLE_SIZE * BULK_EP_NUM)

// EndPoints 0 defines
#define BULK_EP0_RX_SIZE                                (64)
#define BULK_EP0_RX_ADDR                                (BULK_USB_BUF_START + 0)
#define BULK_EP0_TX_SIZE                                (64)
#define BULK_EP0_TX_ADDR                                (BULK_USB_BUF_START + 64)
#define BULK_EP0_TYPE                                   USB_EP_CONTROL

// EndPoints 1 defines
#define BULK_EP1_TX_SIZE                                (64)
#define BULK_EP1_TX0_ADDR                               (BULK_USB_BUF_START + 128)
#define BULK_EP1_TX1_ADDR                               (BULK_USB_BUF_START + 192)
#define BULK_EP1_TYPE                                   USB_EP_BULK

// EndPoints 2 defines
#define BULK_EP2_RX_SIZE                                (64)
#define BULK_EP2_RX0_ADDR                               (BULK_USB_BUF_START + 256)
#define BULK_EP2_RX1_ADDR                               (BULK_USB_BUF_START + 320)
#define BULK_EP2_TYPE                                   USB_EP_BULK

#define BULK_TUSB_INIT_EP_FS(dev) \
  do{\
    /* Init ep0 */ \
    INIT_EP_BiDirection(dev, PCD_ENDP0, BULK_EP0_TYPE);  \
    SET_TX_ADDR(dev, PCD_ENDP0, BULK_EP0_TX_ADDR);  \
```

```
    SET_RX_ADDR(dev, PCD_ENDP0, BULK_EP0_RX_ADDR);  \
    SET_RX_CNT(dev, PCD_ENDP0, BULK_EP0_RX_SIZE);  \
    /* Init ep1 */ \
    INIT_EP_TxDouble(dev, PCD_ENDP1, BULK_EP1_TYPE);     \
    SET_DOUBLE_ADDR(dev, PCD_ENDP1, BULK_EP1_TX0_ADDR, BULK_EP1_TX1_ADDR);  \
    SET_DBL_TX_CNT(dev, PCD_ENDP1, 0);     \
    /* Init ep2 */ \
    INIT_EP_RxDouble(dev, PCD_ENDP2, BULK_EP2_TYPE);     \
    SET_DOUBLE_ADDR(dev, PCD_ENDP2, BULK_EP2_RX0_ADDR, BULK_EP2_RX1_ADDR);     \
    SET_DBL_RX_CNT(dev, PCD_ENDP2, BULK_EP2_RX_SIZE);     \
  }while(0)
```

在 TeenyUSB 的代码中调用 BULK_TUSB_INIT_EP_FS 宏完成 USB FS 模块的端点初始化。初始化之后，IN（TX）端点处于 NAK 状态，OUT（RX）端点处于 VALID 状态，可以接收数据。上面的代码完成了端点 0、端点 1 和端点 2 的配置和 PMA 描述表初始化，其中端点 0 是一个普通的双向端点，端点 1 是一个双缓存 IN（TX）端点，端点 2 是一个双缓存 OUT（RX）端点。TeenyDT 同时也会生成 OTG 模块版本的端点初始化代码，这部分内容在后面介绍 OTG 模块时详细说明。

4.4.2 HAL 库端点初始化

这里只介绍 HAL 库中端点初始化函数的实现与调用方式，具体设备端点初始化的完整流程在 HAL 库 Reset 事件处理与端点初始化这一节中详细说明。HAL 库中的端点初始化主要通过两个函数来完成，一个是 USBD_LL_OpenEP，另一个是 HAL_PCDEx_PMAConfig。

HAL_PCDEx_PMAConfig 函数中，将 PMA 地址和最大包长信息写入到端点的 PCD_EPTypeDef 数据结构中。在 USBD_LL_OpenEP 函数中最后会调用 USB_ActivateEndpoint 函数，对端点寄存器和 PMA 进行初始化。

4.5 响应 Reset 事件与初始化端点

TeenyUSB 协议栈收到 Reset 事件后，将设备地址设置为 0，并且重新初始化所有端点。按照 USB 规范，应当只初始化端点 0，在设置配置请求（Set Configuration Request）中设置其他端点。TeenyUSB 采用了简化的处理方式，配置了所有的端点，原因见配置设备这一节中的内容。Reset 后设备就能与主机通过默认地址 0 进行通讯了。基于前面的原因，初始化端点与 Reset 处理放到了一起来进行介绍。

TeenyUSB 与 HAL 库处理 Reset 事件和配置端点的流程如下图所示：

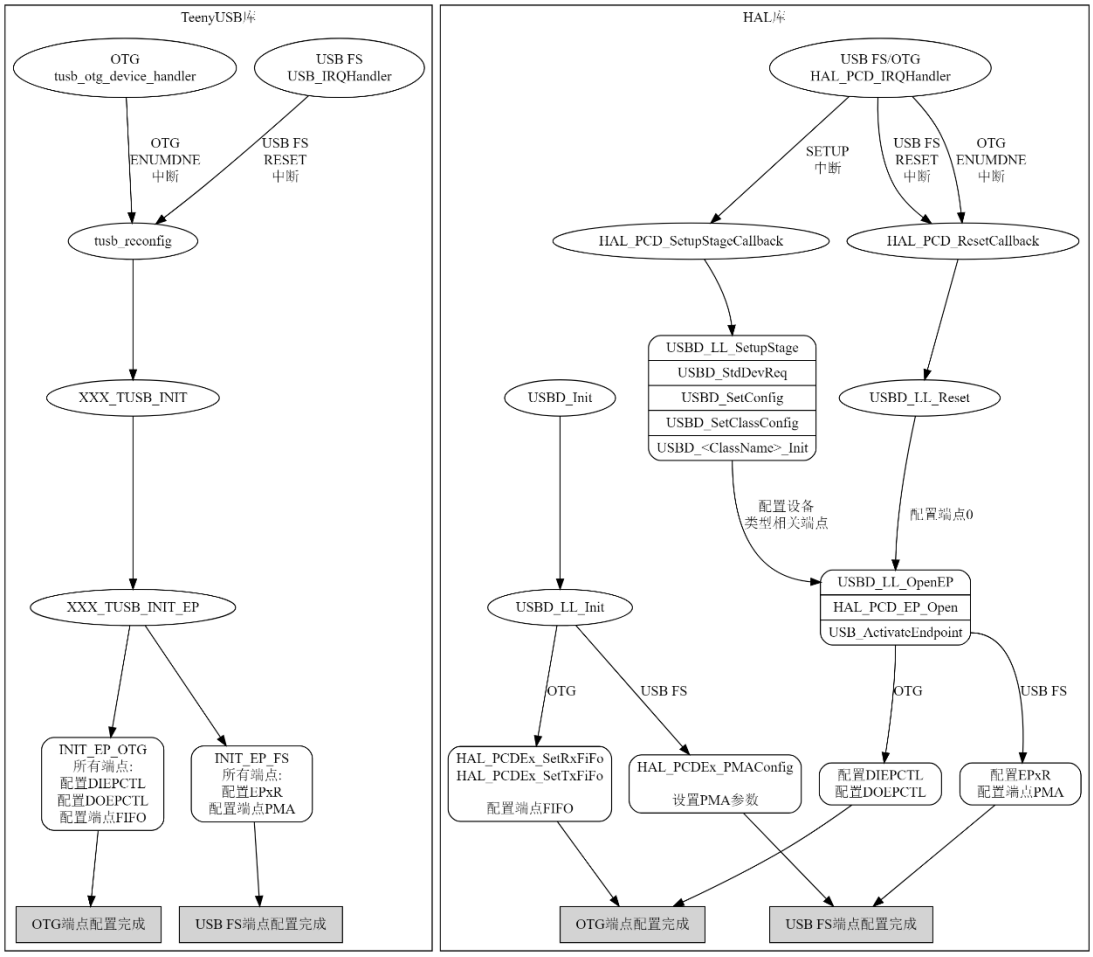


图32　端点配置

上图中包含了 OTG 模块的端点初始化流程，下面会分别介绍 TeenyUSB 和 HAL 库中初始化端点的代码，对代码熟悉的读者可以跳过本节后续内容。

4.5.1 TeenyUSB 库 Reset 事件处理与端点初始化

Reset 事件在 USB 中断处理函数中进行处理，代码如下：

```
void USB_IRQHandler(void){
...
  if (wIstr & USB_ISTR_RESET) {
    GetUSB(dev)->ISTR = (USB_CLR_RESET);
    GetUSB(dev)->BTABLE = (BTABLE_ADDRESS);
    tusb_reconfig(&tusb_dev);
    GetUSB(dev)->DADDR = (0 | USB_DADDR_EF);
  }
...
}
void tusb_reconfig(tusb_device_t* dev){
  // call the BULK device init macro
  BULK_TUSB_INIT(dev);
```

```
  // setup recv buffer for rx end point
  tusb_set_recv_buffer(dev, RX_EP, buf, sizeof(buf));
  // enable rx ep after buffer set
  tusb_set_rx_valid(dev, RX_EP);
}
```

当接收到 Reset 事件时，重新设置内存描述表地址和设备地址，并调用 tusb_reconfig 回调函数对端点进行重新配置。tusb_reconfig 函数中先调用 BULK_TUSB_INIT 宏，对所有端点进行配置，这个宏由 TeenyDT 自动生成，详细说明见端点初始化。然后设置 OUT（RX）端点的接收缓存并设置端点状态为 Valid，这时端点已经可以正常接收数据。这里后面的 tusb_set_rx_valid 函数可以不用调用，因为在前面的端点初始化宏中，会将 OUT（RX）端点的状态设置为 Valid，这里这样写是为了和 OTG 版本的驱动兼容。

在 TeenyUSB 中，设备所有端点的初始化都在 Reset 事件中进行处理，初始化之后主机可以与设备进行通讯。端点的初始化包含两部分内容，一部分是端点寄存器设置，一部分是端点 PMA 缓存设置。

4.5.2 HAL 库 Reset 事件处理与端点初始化

在 HAL 库中，端点的初始化被分为了 3 个部分，第 1 个部分是 USB 设备初始化过程中的端点 PMA 缓存配置，第 2 个部分是 Reset 事件中的端点 0 配置，第 3 个部分是接收到设置配置（Set Configuration）请求后特定设备类型相关端点的配置。

4.5.2.1 Reset 事件中的端点配置

HAL 库中对 Reset 事件的处理主要代码如下：

```
void HAL_PCD_IRQHandler(PCD_HandleTypeDef *hpcd){
...
  if (__HAL_PCD_GET_FLAG (hpcd, USB_ISTR_RESET)){
    __HAL_PCD_CLEAR_FLAG(hpcd, USB_ISTR_RESET);
    HAL_PCD_ResetCallback(hpcd);
    HAL_PCD_SetAddress(hpcd, 0U);
  }
...
}
void HAL_PCD_ResetCallback(PCD_HandleTypeDef *hpcd){
...
  USBD_LL_Reset((USBD_HandleTypeDef*)hpcd->pData);
...
}
USBD_StatusTypeDef USBD_LL_Reset(USBD_HandleTypeDef  *pdev){
  /* Open EP0 OUT */
  USBD_LL_OpenEP(pdev,
                 0x00,
                 USBD_EP_TYPE_CTRL,
```

```
              USB_MAX_EP0_SIZE);
  pdev->ep_out[0].maxpacket = USB_MAX_EP0_SIZE;
  /* Open EP0 IN */
  USBD_LL_OpenEP(pdev,
              0x80,
              USBD_EP_TYPE_CTRL,
              USB_MAX_EP0_SIZE);
  pdev->ep_in[0].maxpacket = USB_MAX_EP0_SIZE;
  /* Upon Reset call user call back */
  pdev->dev_state = USBD_STATE_DEFAULT;
  if (pdev->pClassData)
    pdev->pClass->DeInit(pdev, pdev->dev_config);
  return USBD_OK;
}
```

在 HAL_PCD_IRQHandler 函数中检测到 Reset 事件后，调用 HAL_PCD_ResetCallback 并设置设备地址为 0。在 HAL_PCD_ResetCallback 函数中调用 USBD_LL_Reset 函数。在 USBD_LL_Reset 函数中只对端点 0 进行了初始化。

4.5.2.2 设备类型相关端点配置

HAL 库对设备类型相关端点的初始化，是放在设备类的 Init 函数中进行的。这个类的 Init 函数会在收到设置配置请求后调用。设备类端点初始化相关代码如下：

```
USBD_StatusTypeDef  USBD_StdDevReq (USBD_HandleTypeDef *pdev , USBD_SetupReqTypedef  *req){
  case USB_REQ_SET_CONFIGURATION:
    USBD_SetConfig (pdev , req);
    break;
...
}
static void USBD_SetConfig(USBD_HandleTypeDef *pdev ,
                           USBD_SetupReqTypedef *req){
...
    case USBD_STATE_ADDRESSED:
...
        if(USBD_SetClassConfig(pdev , cfgidx) == USBD_FAIL)
...
}
USBD_StatusTypeDef USBD_SetClassConfig(USBD_HandleTypeDef  *pdev, uint8_t cfgidx){
...
    /* Set configuration  and Start the Class*/
    if(pdev->pClass->Init(pdev, cfgidx) == 0){
...
}
static uint8_t  USBD_CUSTOM_HID_Init (USBD_HandleTypeDef *pdev, uint8_t cfgidx){
...
```

```
  /* Open EP IN */
  USBD_LL_OpenEP(pdev,
                 CUSTOM_HID_EPIN_ADDR,
                 USBD_EP_TYPE_INTR,
                 CUSTOM_HID_EPIN_SIZE);
  /* Open EP OUT */
  USBD_LL_OpenEP(pdev,
                 CUSTOM_HID_EPOUT_ADDR,
                 USBD_EP_TYPE_INTR,
                 CUSTOM_HID_EPOUT_SIZE);
...
    USBD_LL_PrepareReceive(pdev, CUSTOM_HID_EPOUT_ADDR, hhid->Report_buf,
                           USBD_CUSTOMHID_OUTREPORT_BUF_SIZE);
...
}
```

在标准请求处理函数 USBD_StdDevReq 中，收到设置配置请求时，调用 USBD_SetConfig 函数。然后根据设备状态调用 USBD_SetClassConfig 函数对设备当前类型进行配置。设备类型是动态注册在 USBD_HandleTypeDef 结构上的，调用具体设备类的初始化函数。这里以自定义 HID 设备为例，在 HID 设备的初始化函数中，完成了 HID 类用到的端点配置。

4.5.2.3 端点 PMA 缓存配置

HAL 库中的 USBD_LL_OpenEP 函数只对端点寄存器进行了配置，端点 PMA 缓存的配置是在 USB 设备初始化中完成的，代码如下：

```
USBD_StatusTypeDef USBD_Init(USBD_HandleTypeDef *pdev, USBD_DescriptorsTypeDef *pdesc, uint8_t id){
...
  USBD_LL_Init(pdev);
...
}
USBD_StatusTypeDef USBD_LL_Init(USBD_HandleTypeDef *pdev){
...
  HAL_PCDEx_PMAConfig((PCD_HandleTypeDef*)pdev->pData , 0x00 , PCD_SNG_BUF, 0x18);
  HAL_PCDEx_PMAConfig((PCD_HandleTypeDef*)pdev->pData , 0x80 , PCD_SNG_BUF, 0x58);
  HAL_PCDEx_PMAConfig((PCD_HandleTypeDef*)pdev->pData , CUSTOM_HID_EPIN_ADDR , PCD_SNG_BUF, 0x98);
  HAL_PCDEx_PMAConfig((PCD_HandleTypeDef*)pdev->pData , CUSTOM_HID_EPOUT_ADDR , PCD_SNG_BUF, 0xD8);
...
}
```

USBD_Init 是 USB 设备的初始化函数，在其中调用了定义在 usbd_conf.c 文件中的 USBD_LL_Init 函数。USBD_LL_Init 函数是 CubeMX 自动生成的，会根据端点的使用情况初始化端点的 PMA 缓存。

4.6 端点数据处理

主机与设备通过端点进行数据交互，完成设备枚举和通讯。本节将介绍在 TeenyUSB 中如何处理来自主机的各种数据。STM32 的 USB FS 模块，当端点有数据传输成功时会触发正确传输（Correct Transfer CTR）中断事件。代码检测到 CTR 标志时，进入端点数据处理函数。发生中断事件的端点 ID 会被写入 ISTR 寄存器的 EP_ID 字段中，代码通过读取 EP_ID 字段中的内容，得到产生中断的端点 ID 号，TeenyUSB 中代码如下：

```
void USB_IRQHandler(void){
...
  while ((wIstr = GetUSB(dev)->ISTR ) & USB_ISTR_CTR){
     GetUSB(dev)->ISTR = (uint16_t)(USB_CLR_CTR);
     tusb_ep_handler(dev, wIstr & USB_ISTR_EP_ID);
  }
...
}
```

中断中检测到 CTR 事件后会循环处理，直到所有端点数据都处理完成。在介绍端点数据处理前，这里先回顾一下 USB 的几种基本数据传输事务（Transaction），详细内容见 2.3.1 基本事务。

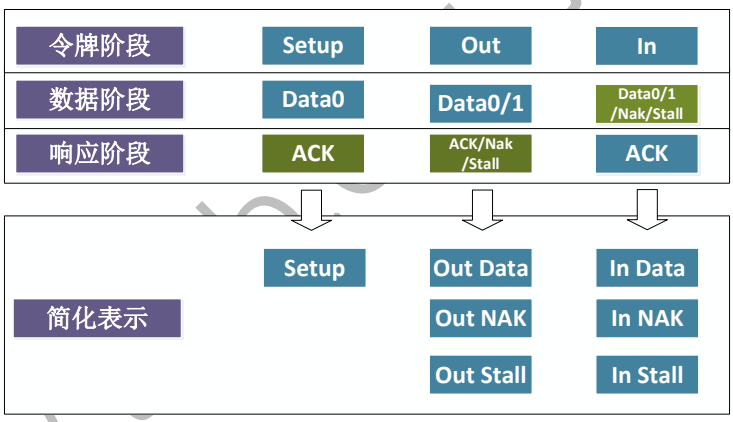


图33　基本数据传输事务

这里有三种基本事务，Setup、Out 和 In，同步端点的事务没有响应阶段。在 STM32 的 USB FS 模块中，正确传输（CTR）中断会在响应阶段收到 ACK 事件后触发。Setup 或者 Out 事务，设备响应 ACK 后触发 CTR_RX 中断。In 事务，设备收到来自主机的 ACK 响应后，触发 CTR_TX 中断。对于同步端点，只要端点为使能（Enable）状态，主机进行 OUT 或 IN 传输，都会触发正确传输（CTR）中断。

用简化表示来描述就是，当收到 Setup，Out Data 和 In Data 这三种事务时，会触发端点的 CTR 中断。Setup 与 Out Data 同属于 CTR_RX 事件，在 STM32 的 USB FS 模块中，通过 EPxR 寄存器的 SETUP 字段来区分 Setup 和 Out Data 事务。

4.6.1 基本事务与数据传输

在《USB2.0 规格书》中，事务即 transaction，传输即 transfer。一次数据传输由多个事务组成，通过一些特定的条件来完成一次数据传输，详细内容见传输完成条件。事务是 USB 设备底层需要考虑的事情，传输是应用程序要考虑的事情。前面提到 CTR 中断表示一次基

本事务完成，这里将介绍 TeenyUSB 中如何将基本事务组合成数据传输，供应用层来使用。设备的描述符获取，配置的设置和读取，都是通过数据传输来完成的。

4.6.2 数据传输处理流程

TeenyUSB 中的数据传输处理流程见下图：

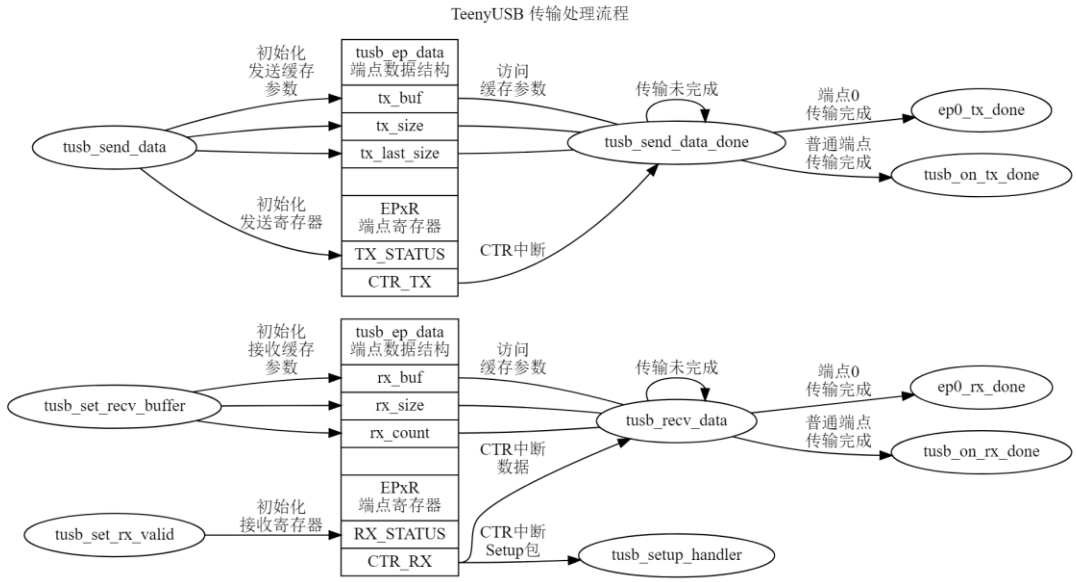


图34　　数据传输处理流程

上图是 IN、OUT 和 SETUP 的处理流程，下面对三种数据处理流程的代码进行详细介绍。

4.6.3 IN 传输处理

TeenyUSB 中，进行 IN 数据传输时，处理流程如下：

1. 调用 tusb_send_data 函数对发送缓存和端点寄存器进行初始化。对于普通端点，将第一包需要发送的数据写入 PMA 中，然后配置端点寄存器使能发送。对于双缓冲端点的处理，将会在后面单独介绍。
2. 一包数据发送完成，会触发 CTR_TX 中端，CTR_TX 中端在函数 tusb_send_data_done 中处理。根据缓存的总大小和实际处理的数据长度判断数据是否传输完成。如果传输完成，普通端点调用 tusb_on_tx_done 函数通知应用程序，端点 0 调用注册在 ep0_tx_done 上的回调函数进行处理。如果传输未完成，则将下一包数据复制到端点对应的 PMA 中，等待下一次的 CTR_TX 中断。

4.6.3.1 IN 传输完成判断

对于 IN 数据传输完成的判断代码如下：

```
if(ep->tx_remain_size || ((EPn == 0 || ep->tx_need_zlp) && ep->tx_last_size == GetInMaxPacket(dev,
EPn) )) {
    copy_tx(dev, ep, pma, ep->tx_buf, ep->tx_remain_size, GetInMaxPacket(dev, EPn));
```

```
    PCD_SET_EP_TX_STATUS(GetUSB(dev), EPn, USB_EP_TX_VALID);
    return;
  }
  if(ep->tx_pushed == 0){
    if( IS_ISO() ){
      if(ep->tx_buf){
        tusb_on_tx_done(dev, EPn);
        ep->tx_buf = 0;
      }
    }else{
      if(EPn == 0){
        if(dev->ep0_tx_done){
          // invoke status transmitted call back for ep0
          dev->ep0_tx_done(dev);
          dev->ep0_tx_done = 0;
        }
      }else{
        tusb_on_tx_done(dev, EPn);
      }
    }
  }
```

根据端点缓存中的 tx_remain_size 字段查看是否还有数据需要发送。

如果 tx_remain_size 不为 0，说明还有数据要发送，将剩余的数据复制到 PMA 中继续发送。

当 tx_remain_size 为 0 时，查看最后一包数据的长度。如果最后一包数据是最大包长，并且 tx_need_zlp 标志为 1 或者端点号为 0，发送一个 0 包结束当前传输。关于传输完成条件在 2.4 传输完成条件有详细介绍。对于非 0 端点，应用程序可以通过 TUSB_TXF_ZLP 标志位来控制是否需要发送 0 长度的包来完成传输。这是因为有些应用需要发送 0 包来结束传输，比如 CDC 串口。而有些应用不需要发送 0 包，比如 MSC 大容量存储设备。

如果 tx_pushed 为 0，说明没有需要发送的数据了，进入后面的数据传输完成处理过程，端点 0 调用注册在 ep0_tx_done 上的回调函数处理，其它端点调用 tusb_on_tx_done 回调函数处理。对于同步端点，通过发送缓存是否为空，来判断是否需要通知应用程序传输完成。这是因为同步 IN 端点的一旦使能后，会一直触发 CTR 中断。如果没有数据发送，Teeny USB 将同步端点的数据长度设置为 0，并且清掉发送缓存。

4.6.4 OUT 传输处理

进行 OUT 数据传输时，处理流程如下：

1. 调用 tusb_set_recv_buffer 函数，对接收缓存进行初始化；
2. 调用 tusb_set_rx_valid 函数，对接收寄存器进行设置；
3. 在接收到一包 Out 数据时，触发 CTR_RX 中断。在 tusb_recv_data 函数中，把 PMA 中接收到的数据复制到应用程序的缓存中。根据缓存总长度和当前接收到的数据长度，判断传输是否完成。如果传输完成，根据端点选择不同的处理函数，普通端点

调用 tusb_on_rx_done，端点 0 调用注册在 ep0_rx_done 上的回调函数。如果传输未完成，重新设置端点为 Valid 状态，等待下一包数据。

4.6.4.1 OUT 端点的数据流控

当数据超过设备的处理能力时，设备需要通知主机不要继续发送数据。在 USB 中，设备将 OUT 端点设置为 NAK 状态，阻塞主机的数据。在 TeenyUSB 中，当一次数据传输完成后，调用 tusb_on_rx_done 回调，此函数返回 0 时表示缓存中数据处理完毕，可以开始下一次传输。返回其他值时，表示数据未处理，暂停接收数据。当数据处理完成后调用 tusb_set_rx_valid，重新启动接收。

4.6.4.2 OUT 传输完成判断

对 OUT 数据传输完成的判断代码如下：

```
void tusb_recv_data(tusb_device_t* dev, uint8_t EPn){
...
  if(ep->rx_buf && pma){
    uint32_t len = tusb_pma_rx(dev, pma, ep->rx_buf + ep->rx_count);
    pma->cnt = 0;
    ep->rx_count += len;
    if(len != GetOutMaxPacket(dev, EPn) || ep->rx_count >= ep->rx_size){
      if(EPn == 0){
        if(dev->ep0_rx_done){
          dev->ep0_rx_done(dev);
          dev->ep0_rx_done = 0;
        }
        ep->rx_buf = 0;
      }else{
        if(tusb_on_rx_done(dev, EPn, ep->rx_buf, ep->rx_count) == 0){
          ep->rx_count = 0;
        }else{
          // of rx done not return success, change rx_count to rx_size, this will block
          // the data recieve
          ep->rx_count = ep->rx_size;
        }
      }
    }
  }
  if(ep->rx_count<ep->rx_size){
    TUSB_SET_RX_STATUS(GetUSB(dev), EPn, EP, USB_EP_RX_VALID);
  }
}
```

当接收到的数据小于最大包长，或者数据总长度大于缓存长度时，表示 OUT 传输完成。OUT 传输完成时，端点 0 调用注册在 ep0_rx_done 上的回调函数，其它端点调用 tusb_on_rx_done。TeenyUSB 通过 tusb_on_rx_done 的返回值判断缓存是否能够继续使用。

4.6.4.3 最大包长信息

在 IN 和 OUT 传输的处理流程中，都会用到端点的最大包长信息。在发送数据时，通过判断最后一包数据是否为最大包长，来确定是否要发送 0 长度的包。在接收数据时，通过判断数据是否小于最大包长，来确定当前传输是否完成。在 USB FS 设备中，最大包长信息保存在设备结构体 tusb_device_t 的 rx_max_size 和 tx_max_size 字段中，在设备初始化时进行设置。代码如下：

```
#define BULK_TUSB_INIT_DEVICE(dev) \
  do{\
    /* Init device features */       \
    memset(&dev->addr, 0, TUSB_DEVICE_SIZE);    \
    dev->status = BULK_DEV_STATUS;         \
    dev->rx_max_size = BULK_rxEpMaxSize;         \
    dev->tx_max_size = BULK_txEpMaxSize;         \
    dev->descriptors = &BULK_descriptors;         \
  }while(0)
```

这里的 BULK_rxEpMaxSize 和 BULK_txEpMaxSize 是由 TeenyDT 根据描述符中的内容自动生成的。在 OTG 设备上最大包长信息会写入端点相关的寄存器中，所以在 OTG 的设备结构体中没有 rx_max_size 和 tx_max_size 字段。

4.6.5 SETUP 处理

Setup 数据的传输比较特殊，Setup 不受端点状态影响，始终能够传输成功。Setup 数据会触发 CTR_RX 中断，同时将端点寄存器中的 SETUP 字段设置为 1。Setup 包数据会缓存在端点 0 的 PMA 中，因此在 TeenyUSB 中，检测到 Setup 时，先从端点 0 的 PMA 中读取出 Setup 内容到设备的 setup 字段中。然后调用 tusb_setup_handler 对 Setup 包进行处理。

Setup 有三种传输类型，分别为读数据、写数据和无数据。关于这三种控制传输类型的详细介绍在 2.3.2 控制传输这一节中，这里介绍在 TeenyUSB 中的处理方式。

4.6.5.1 读数据传输

Setup 包的内容为读数据时，根据 Setup 的 wLength 字段获取到要读取的数据总长度，然后调用 tusb_send_data 向端点 0 发送实际数据。如果实际数据长度小于 wLength，发送数据长度为实际数据长度。如果实际数据长度大于 wLength，发送数据长度为 wLength。例如读取配置描述符时，主机会先读取 9 字节的头部数据，这时只用发送前 9 字节。然后主机根据配置描述符中的 wTotalLength 字段，读取全部的配置描述符，这时设备发送全部的数据内容。

数据发送完成后，主机会发送一个长度为 0 状态包到设备，为了能够正确处理状态包，

设备需要将端点 0 的 OUT 方向设置为 Valid 状态。

4.6.5.2 写数据传输

Setup 包的内容为写数据时，根据 Setup 的 wLength 字段准备接收缓存。设备在接收到 Setup 包后，会将端点 0 的 OUT 方向设置为 NAK 状态。当接收 Setup 包的数据缓存准备就绪后，将端点设置为 Valid 状态，接收后续的数据。在 TeenyUSB 中，Setup 数据缓存的准备在 tusb_setup_handler 函数中完成，端点状态的设置在 tusb_ep_handler 函数中完成。

Setup 数据传输完毕后，调用注册在 ep0_rx_done 上的回调函数进行处理。处理成功后向主机发送一个长度为 0 的状态包。

4.6.5.3 无数据传输

Setup 包的内容为无数据时，直接对 Setup 中的内容进行处理，处理成功后发送一个长度为 0 的状态包。

对于控制传输，如果不发送状态包，主机会认为处理超时，会再次发起控制传输，超过重试次数后失败。如果控制传输内容不正确，例如设备不支持，可以将端点 0 设置为 STALL 状态通知主机。端点 0 设置为 STALL 状态后并不会影响后续的控制传输，

4.6.6 双缓冲端点

为了能够提高数据传输带宽，USB FS 模块为 BULK 和同步端点提供了双缓冲功能。双缓冲的工作方式是，当应用程序操作一个缓存时，USB 模块可以操作另外一个缓存。当数据吞吐量很大时，不会因为应用程序复制数据而阻塞 USB 模块操作。对于普通端点，由端点寄存器中的 STATUS 字段来指定端点状态。对于 BULK 双缓冲端点，当端点寄存器设置为 Valid 状态时，由 DTOG_RX 和 DTOG_TX 字段共同决定端点实际状态，端点实际状态与 DTOG 字段关系见下表：

表18　　Bulk 双缓存 DTOG 位与端点状态

端点类型	DTOG_TX	USB模块缓存	DTOG_RX	应用程序缓存	端点状态
BULK IN	0	TX0	0	TX0	NAK
	0	TX0	1	TX1	VALID
	1	TX1	0	TX0	VALID
	1	TX1	1	TX1	NAK
端点类型	DTOG_RX	USB模块缓存	DTOG_TX	应用程序缓存	端点状态
BULK OUT	0	RX0	0	RX0	NAK
	0	RX0	1	RX1	VALID
	1	RX1	0	RX0	VALID
	1	RX1	1	RX1	NAK

上表中，当 USB 模块与应用程序使用同一块缓存时，端点会被设置位 NAK 状态，否则为 VALID 状态。当端点为 IN 时，DTOG_TX 指示 USB 模块使用的缓存，DTOG_RX 指示应用程序使用的缓存。当端点为 OUT 时，DTOG_RX 指示 USB 模块使用的缓存，DTOG_TX 指示应用程序使用的缓存。

同步端点没有数据流控，端点只有 Enable 和 Disable 两种状态，因此同步端点的 DTOG 字段只表示缓存使用情况，不能控制端点状态。

4.6.6.1 BULK IN 双缓存

TeenyUSB 中，BULK IN 双缓存的处理流程如下：

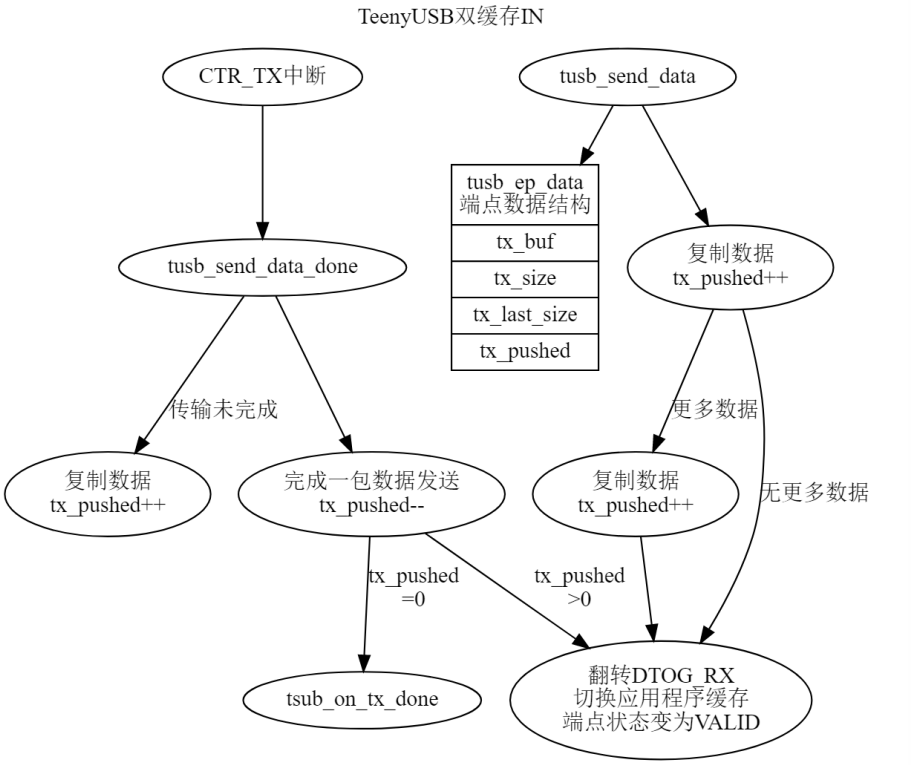


图35　BULK IN 双缓存处理流程

BULK IN 双缓存端点初始化时设置为 VALID 状态，但是 DTOG_TX 和 DTOG_RX 都设置为 0，此时端点实际状态为 NAK。要发送数据时，先将数据复制到相应的 PMA 中，再翻转 DTOG_RX，进行数据发送。根据数据实际的长度，在开始发送时会复制 1 包或者 2 包数据到 PMA 中。在 CTR_TX 中断中，为了提高数据吞吐能力，TeenyUSB 先判断缓存中是否还有数据，如果有，立刻翻转 DTOG_RX，这样 USB 模块可以先对已经复制到 PMA 中的数据进行处理。然后再根据 tx_pushed 计数判断数据是否传输完成。如果未完成，继续复制数据到 PMA 中，并对 tx_pushed+1。当 tx_pushed 减为 0 时，数据发送完成，调用 tusb_on_tx_done 函数，通知应用程序。

4.6.6.2 BULK OUT 双缓存

TeenyUSB 中，BULK OUT 双缓存的处理流程如下：

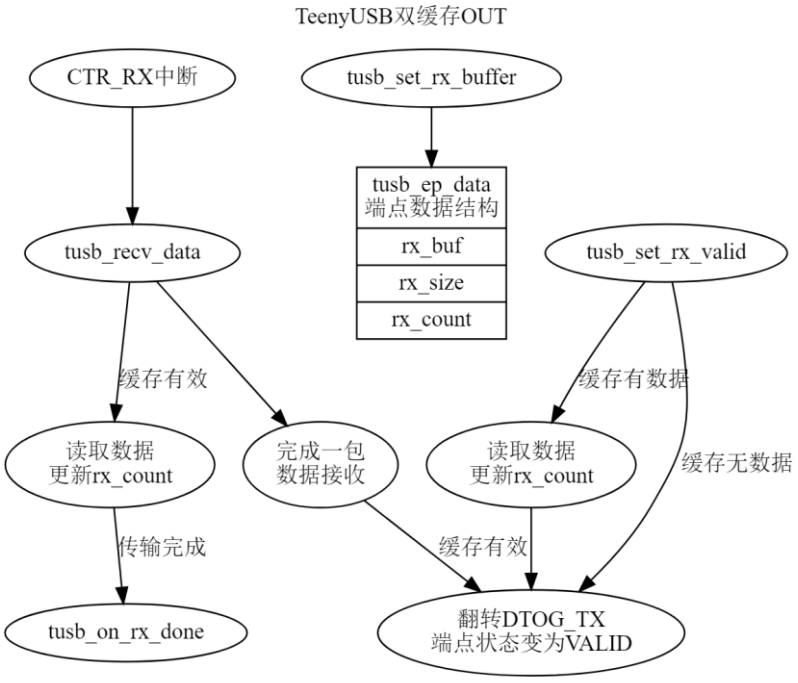


图36　　BULK OUT 双缓存处理流程

BULK OUT 双缓存端点初始化时设置为 VALID 状态，与 IN 不同的是，DTOG_RX 设置为 0，DTOG_TX 设置为 1，这时端点实际状态为 VALID，能够接收主机的 OUT 数据。如果此时主机开始发送数据到设备，因为设备的接收缓存还未准备就绪，数据会存放在 PMA 中。在 tusb_recv_data 函数中，由于缓存未就绪，不会对 TX_DTOG 进行翻转，端点会处于 NAK 状态，阻塞后面的数据。当通过 tusb_set_recv_buffer 函数设置端点缓存后，再调用 tusb_set_rx_valid 函数。在调用 tusb_set_rx_valid 函数前，会先查看 PMA 中是否有数据未处理，如果 PMA 中已经有数据，先将数据复制到应用程序提供的缓存中，然后翻转 TX_DTOG，将端点设置为 VALID 状态，接收后续的数据。

4.6.6.3 同步端点的双缓存

同步端点的双缓存比较简单，DTOG 状态定义见下表：

表19　　同步端点 DTOG 位

端点类型	DTOG_TX	USB模块缓存	应用程序缓存
ISO IN	0	TX0	TX1
	1	TX1	TX0
端点类型	DTOG_RX	USB模块缓存	应用程序缓存
ISO OUT	0	RX0	RX1
	1	RX1	RX0

同步端点一旦设置为 Enable 状态，会一直发送/接收数据。如果应用程序将数据复制到同步 IN 端点的 PMA 中后，不做任何干预，缓存 0 和缓存 1 中的数据会交替发送出去，一直发送不会停止。为了表示数据是否有效，TeenyUSB 在 ISO 数据传输成功后将缓存长度设置为 0，主机收到长度为 0 的数据就会认为是无效数据。详细内容见 2.6.1 同步端点的流控。对于同步 OUT 端点，数据会直接接收到 PMA 中，应用程序需要保证及时将数据读取出，避

免被后面的数据覆盖。

4.6.7 BULK 双缓存与 FIFO

BULK 双缓存的工作机制还可以看成是只有两级深度的 FIFO。

4.6.7.1 数据发送流程

当端点方向为 IN 时，发送流程为：

1. 根据待发送的数据长度向 FIFO 中写入数据，更新 tx_pushed 值，然后启动发送；
2. 接收到 CTR_TX 中断时，tx_pushed 值减 1，如果 tx_pushed 不为 0，说明 FIFO 中还有数据，立刻启动发送；
3. 根据剩余发送数据长度，如果传输未完成继续向 FIFO 中写入数据，如果传输完成，调用 tusb_on_tx_done 回调函数通知应用程序。如果传输未完成，tx_pushed 值加 1，复制数据到 PMA 中。

4.6.7.2 数据接收流程

数据接收时，无论应用程序是否提供缓存，都会将端点设置为 Valid 状态，这样如果有 OUT 数据，会先存入 FIFO 中，等到应用程序提供缓存后将 FIFO 中的数据读出。当端点方向为 OUT 时，接收流程为：

1. 配置接收缓存
2. 启动端点接收，启动时先查看接收 FIFO 中是否有数据，如果有则读取到应用程序设置的缓存中；
3. 在 CTR_RX 中断中，如果缓存有足够空间，再次启动接收，这时 FIFO 中已经有一包数据，还未读出，启动后新的数据会写入 FIFO 的另一个区域中，提高数据吞吐。如果缓存没有空间，停止接收。没有空间时，新的包会被接收到 FIFO 中，但是应用程序不会将数据读取出，在下次调用 tusb_set_rx_valid 函数时读取到应用程序的缓存中；
4. 将从 FIFO 中读取数据到应用程序配置的缓存中，调整 rx_count；
5. 根据接收到的数据包长，判断传输是否完成，如果完成调用 tusb_on_rx_done 回调函数通知应用程序。

4.7 USB FS 使用小结

TeenyUSB 中 USB FS 设备的使用流程如下图所示：

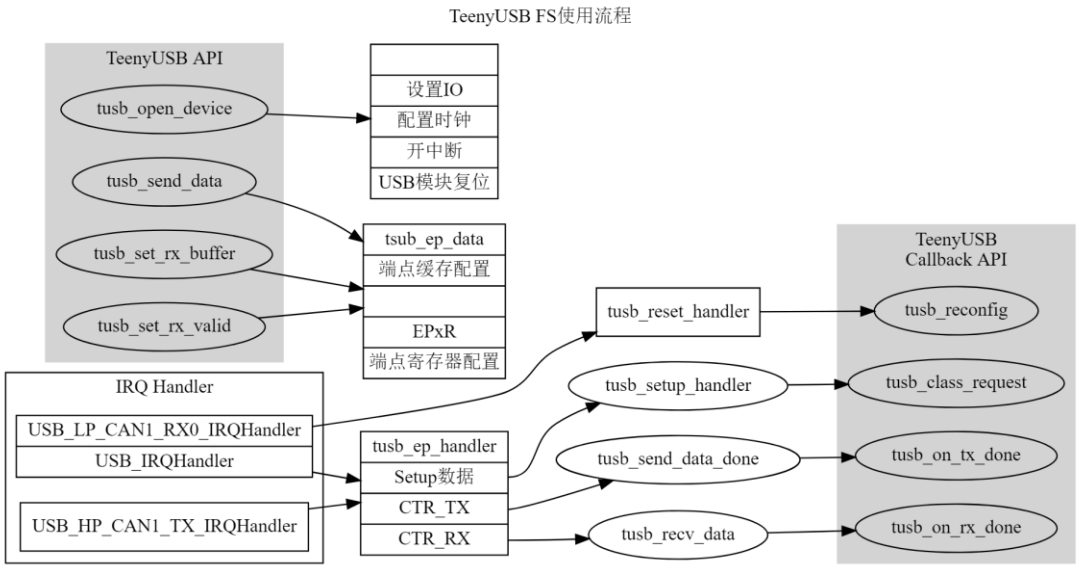


图37　　USB FS 使用流程

上图中灰色部分是 TeenyUSB 提供的 API，有两类 API，一类是普通函数，一类是回调函数。TeenyUSB 中，USB 设备的开发都是通过上述的普通函数和回调函数来完成的，具体的 USB 设备开发过程见第 6 章 USB 设备开发。

4.7.1 普通函数

普通函数由应用程序主动调用，定义见下表：

表20　　TeenyUSB 普通函数

普通函数	说明
tusb_get_device	获得设备，在开始使用USB设备前，调用此函数获得设备对象，供后面的函数使用，上图中未画出
tusb_open_device	打开USB模块，将设备设置为连接状态
tusb_close_device	关闭USB模块，将设备设置为断开状态，上图中未画出
tusb_send_data	发送数据
tusb_set_rx_buffer	设置接收缓存，启动端点发送功能
tusb_set_rx_valid	启动端点接收功能，如果是双缓存端点，此函数还会获取接收端点PMA中的数据

4.7.2 回调函数

当特定事件发生时调用回调函数，TeenyUSB 中回调函数为 weak 类型，应用程序中实现后调用应用程序中的版本，如果应用程序未实现则调用默认版本。回调函数定义见下表：

表21　　TeenyUSB 回调函数

回调函数	说明
tusb_reconfig	接收到Reset中断时调用，在这个函数中初始化所有端点，初始化设备的所有描述符
tusb_class_request	接收到设备类请求时调试，在这个函数中处理特定设备类型相关的请求.返回0说明请求需要协议栈继续处理。返回1表示请求已经处理，不需要协议栈处理。
tusb_on_rx_done	OUT传输完成时调用，此函数返回0表示缓存中数据处理完成，缓存可以接收后续的包。此函数返回非0，表述缓存数据未处理，接收挂起，当数据处理完毕后调用tusb_set_rx_valid重新使能端点接收
tusb_on_tx_done	IN端点传输完成时调用，表示数据传输完成
tusb_delay_ms	以毫秒为单位的延时函数，由应用程序根据平台实现

5 STM32 OTG 模块设备模式

STM32 的 OTG 模块有 FS 和 HS 两个版本，其中 HS 又分为内置 phy 和外置 phy 的版本。这些不同版本的 OTG 模块操作方式类似。OTG 模块框图如下：

Figure 303. OTG full-speed block diagram

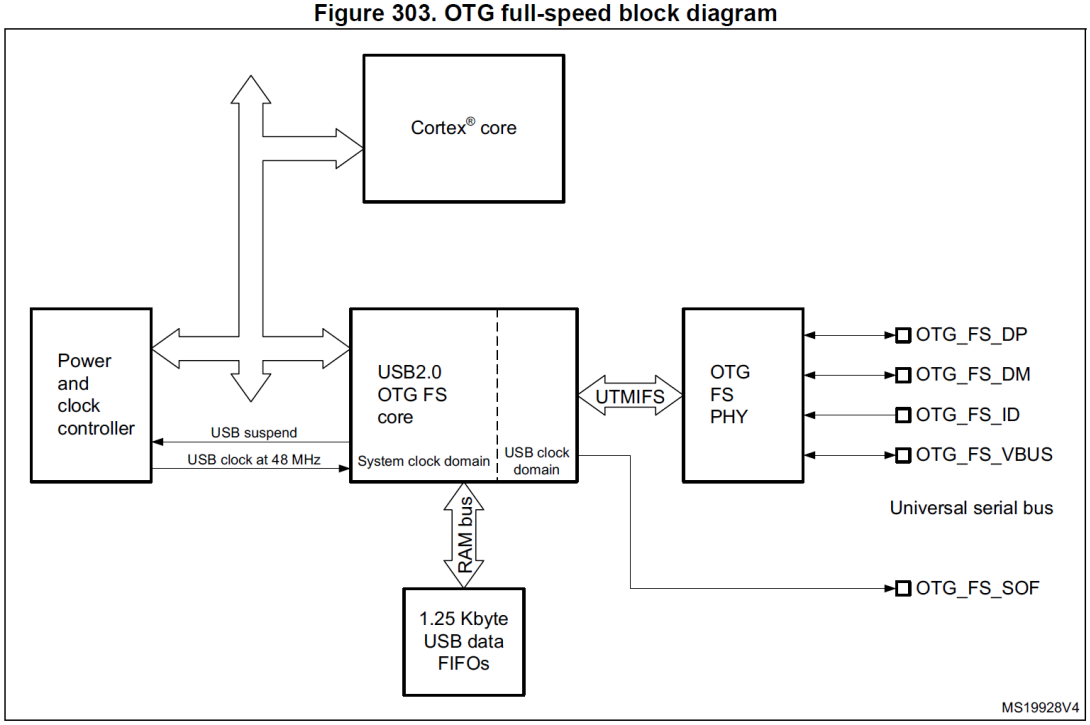


图38　　OTG FS 模块框图

Figure 427. OTG high-speed block diagram for STM32F7x2xx

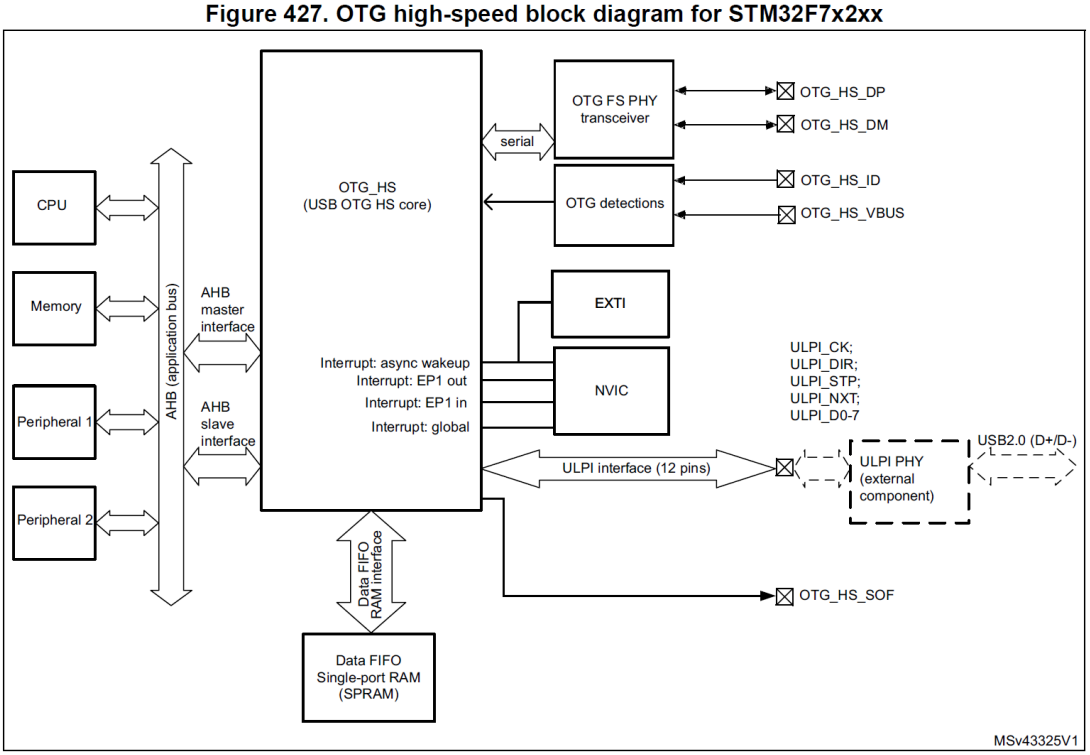


图39　　OTG HS 模块框图

从上图中可以看到，OTG FS 和 OTG HS 模块都有 FIFO 对 USB 数据进行缓存。而 OTG HS 模块同时具备 AHB 主接口和 AHB 从接口，这是因为 OTG HS 模块具备 DMA 的能力，能够通过 PHB 总线在 USB FIFO 和 Memory 之间搬移数据。USB OTG 的实际含义是能够在使用过程中切换主机和设备的角色。在 STM32 的 OTG 模块中，当工作在设备模式时，使用一套寄存器；当工作在主机模式时，使用另一套寄存器。本章只介绍工作在设备模式下 OTG 模块的操作，本章中后续提到的 USB 设备即工作在设备模式下的 OTG 模块。

5.1 USB 核心初始化操作

不同芯片的 USB 设备初始化操作在 stm32fxxx_init.c 文件中实现，OTG 设备的中断处理在 teeny_usb_stm32_otg_device.c 文件中实现。本节介绍如何初始化工作在设备模式的 OTG 模块。初始化完成后，设备能够接收到 USB 总线上的复位信号。

5.1.1 时钟设置

OTG 模块相关时钟如下图所示：

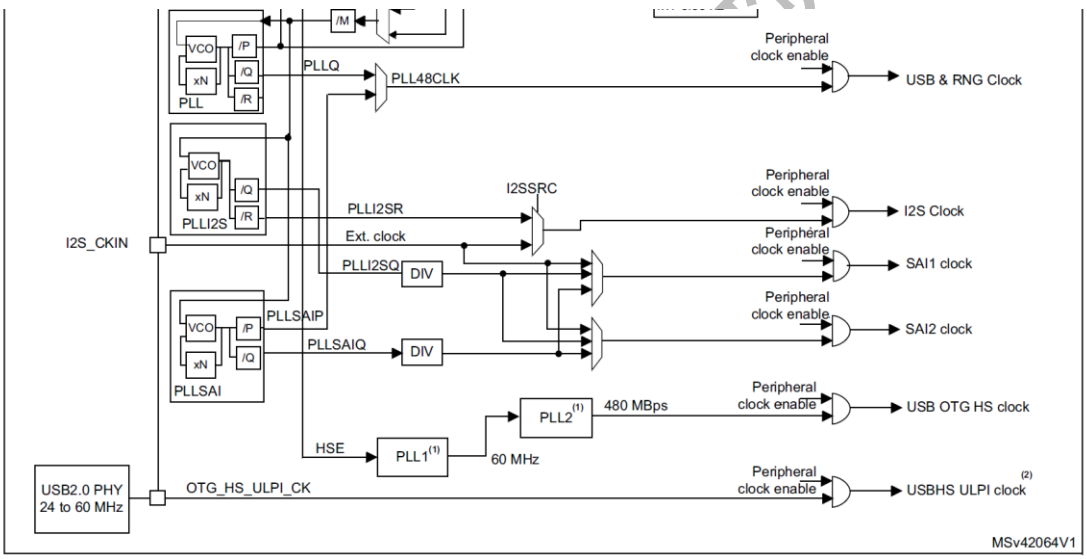


图40　　STM32 部分时钟树

上图中有三种与 USB 相关的时钟，USB&RNG clock，USB OTG HS clock，USB HS ULPI clock。

USB&RNG clock 是给 OTG FS 模块的时钟，来自于 PLLQ 或者 PLLSAI。

USB OTG HS clock 是给内置 OTG 高速 phy 的时钟，来自于内置高速 phy 专用的 PLL1 和 PLL2，PLL1 又来自于 HSE。因此要使用内置的高速 phy 时，需要选择一个合适的 HSE 时钟。

USB HS ULPI clock 是来自于外置 phy 的时钟，这个时钟由外置的 phy 提供给芯片内部的 ULPI 接口使用。

当采用不同的 phy 时配置不同的时钟，时钟相关配置代码如下：

```
static void tusb_otg_core_init(tusb_core_t* core){
  USB_OTG_GlobalTypeDef* USBx = GetUSB(core);
  if(GetUSB(core) == USB_OTG_FS){
```

```
#if defined(OTG_FS_EMBEDDED_PHY)
    __HAL_RCC_USB_OTG_FS_CLK_ENABLE();
#endif
  }else if(GetUSB(core) == USB_OTG_HS){
#if defined(USB_HS_PHYC)
    __HAL_RCC_OTGPHYC_CLK_ENABLE();
#endif
    __HAL_RCC_USB_OTG_HS_CLK_ENABLE();
    __HAL_RCC_USB_OTG_HS_ULPI_CLK_ENABLE();
...
  }
...
}
```

根据不同的模块，选择打开不同的时钟。

5.1.2 IO 设置

不同的模块使用的 IO 也有所不同，代码如下：

```
static void tusb_setup_otg_fs_io(void){
  /**USB_OTG_FS GPIO Configuration
  PA8     ------> USB_OTG_FS_SOF
  PA9     ------> USB_OTG_FS_VBUS
  PA10     ------> USB_OTG_FS_ID
  PA11     ------> USB_OTG_FS_DM
  PA12     ------> USB_OTG_FS_DP
  */
  __HAL_RCC_GPIOA_CLK_ENABLE();
  set_io_af(GPIOA, 11, GPIO_AF10_OTG_FS);
  set_io_af(GPIOA, 12, GPIO_AF10_OTG_FS);
}

#if defined(USB_HS_PHYC)
// Setup IO for OTG_HS core with embedded HS phy
static void tusb_setup_otg_hs_io(void){
  __HAL_RCC_GPIOB_CLK_ENABLE();
  set_io_af(GPIOB, 14, GPIO_AF12_OTG_HS_FS);
  set_io_af(GPIOB, 15, GPIO_AF12_OTG_HS_FS);
}
#endif
...
#elif defined(OTG_HS_EXTERNAL_PHY)
    OTG_HS_ULPI_IO_CLK_ENABLE();
    set_io_af_mode( OTG_HS_ULPI_D0 );
    set_io_af_mode( OTG_HS_ULPI_D1 );
```

```
    set_io_af_mode( OTG_HS_ULPI_D2 );
    set_io_af_mode( OTG_HS_ULPI_D3 );
    set_io_af_mode( OTG_HS_ULPI_D4 );
    set_io_af_mode( OTG_HS_ULPI_D5 );
    set_io_af_mode( OTG_HS_ULPI_D6 );
    set_io_af_mode( OTG_HS_ULPI_D7 );
    set_io_af_mode( OTG_HS_ULPI_DIR );
    set_io_af_mode( OTG_HS_ULPI_STP );
    set_io_af_mode( OTG_HS_ULPI_NXT );
    set_io_af_mode( OTG_HS_ULPI_CK );
#endif
```

上面的 ULPI 部分 IO 的初始化，在不同的板子上 ULPI 会采用不同的 IO，需要根据板子的实际情况 对这些宏进行定义。

5.1.3 中断设置

中断设置比较简单，根据不同的模块打开对应中断即可，代码如下：

```
static void tusb_otg_core_init(tusb_core_t* core){
  USB_OTG_GlobalTypeDef* USBx = GetUSB(core);
  if(GetUSB(core) == USB_OTG_FS){
...
    NVIC_SetPriority(OTG_FS_IRQn, 0);
    NVIC_EnableIRQ(OTG_FS_IRQn);
...
  }else if(GetUSB(core) == USB_OTG_HS){
...
    NVIC_SetPriority(OTG_HS_IRQn, 0);
    NVIC_EnableIRQ(OTG_HS_IRQn);
...
  }
}
```

TeenyUSB 中没有使用高速 OTG 模块的端点 1 专用中断，这里没有做相应的初始化。

5.1.4 USB 模块设置

STM32 的 OTG 模块的配置包含两部分的内容，一部分是 OTG Core 相关的设置，无论是设备模式还是主机模式都需要配置，另一部分根据 OTG 模块的实际工作情况进行配置。TeenyUSB 中，OTG 模块内核部分的初始化工作在 tusb_otg_core_init 函数中完成，设备部分的初始化工作在 tusb_init_otg_device 函数中完成，代码如下：

```
void tusb_open_device(tusb_device_t* dev){
  USB_OTG_GlobalTypeDef* USBx = GetUSB(dev);
```

```
  tusb_otg_core_init((tusb_core_t*) dev);
  USBx->GUSBCFG &= ~(USB_OTG_GUSBCFG_FHMOD | USB_OTG_GUSBCFG_FDMOD);
  USBx->GUSBCFG |= USB_OTG_GUSBCFG_FDMOD;
  tusb_init_otg_device(dev);
}
```

如果是打开为设备模式，先初始化内核部分内容，然后将 OTG 模块置为设备模式，再调用 tusb_init_otg_device 函数完成设备功能的初始化。

5.1.5 核心初始化操作小结

当完成设备模式的核心初始化之后，当设备的 USB 接口接入主机时，能够触发 OTG 模块的中断。后面的操作都在 OTG 模块的中断函数中进行处理。

5.2 USB 端点寄存器操作

在 STM32 的 OTG 模块中，USB 的 IN 端点和 OUT 端点采用了不同的寄存器组，HAL 库中对 OTG 设备端点寄存器定义如下：

```
/**
  * @brief USB_OTG_IN_Endpoint-Specific_Register
  */
typedef struct{
  __IO uint32_t DIEPCTL;         /*!< dev IN Endpoint Control Reg    900h + (ep_num * 20h) + 00h */
  uint32_t Reserved04;           /*!< Reserved                       900h + (ep_num * 20h) + 04h */
  __IO uint32_t DIEPINT;         /*!< dev IN Endpoint Itr Reg        900h + (ep_num * 20h) + 08h */
  uint32_t Reserved0C;           /*!< Reserved                       900h + (ep_num * 20h) + 0Ch */
  __IO uint32_t DIEPTSIZ;        /*!< IN Endpoint Txfer Size         900h + (ep_num * 20h) + 10h */
  __IO uint32_t DIEPDMA;         /*!< IN Endpoint DMA Address Reg    900h + (ep_num * 20h) + 14h */
  __IO uint32_t DTXFSTS;         /*!< IN Endpoint Tx FIFO Status Reg 900h + (ep_num * 20h) + 18h */
  uint32_t Reserved18;           /*!< Reserved  900h+(ep_num*20h)+1Ch-900h+ (ep_num * 20h) + 1Ch */
} USB_OTG_INEndpointTypeDef;
/**
  * @brief USB_OTG_OUT_Endpoint-Specific_Registers
  */
typedef struct{
  __IO uint32_t DOEPCTL;  /*!< dev OUT Endpoint Control Reg           B00h + (ep_num * 20h) + 00h */
  uint32_t Reserved04;    /*!< Reserved                               B00h + (ep_num * 20h) + 04h */
  __IO uint32_t DOEPINT;  /*!< dev OUT Endpoint Itr Reg               B00h + (ep_num * 20h) + 08h */
  uint32_t Reserved0C;    /*!< Reserved                               B00h + (ep_num * 20h) + 0Ch */
  __IO uint32_t DOEPTSIZ; /*!< dev OUT Endpoint Txfer Size            B00h + (ep_num * 20h) + 10h */
  __IO uint32_t DOEPDMA;  /*!< dev OUT Endpoint DMA Address           B00h + (ep_num * 20h) + 14h */
  uint32_t Reserved18[2]; /*!< Reserved B00h + (ep_num * 20h) + 18h - B00h + (ep_num * 20h) + 1Ch */
} USB_OTG_OUTEndpointTypeDef;
```

通过 HAL 库中的 USBx_INEP 和 USBx_OUTEP 宏可以得到对应的端点寄存器地址。初始

化端点时，通过 DxEPCTL 寄存器配置端点的类型，最大包长，FIFO 号等信息。使用端点传输数据时，通过 DxEPTSIZ 寄存器配置需要发送数据的大小，包数量等信息，然后配置 DxEPCTL 启动数据传输。关于端点初始化和数据发送在下面会有详细介绍。

5.3 FIFO 操作

在 STM32 的 OTG 模块中，端点数据通过 FIFO 在应用程序和 OTG 模块之间进行交互。设备模式下的 FIFO 访问见下图：

Figure 434. Device-mode FIFO address mapping and AHB FIFO access mapping

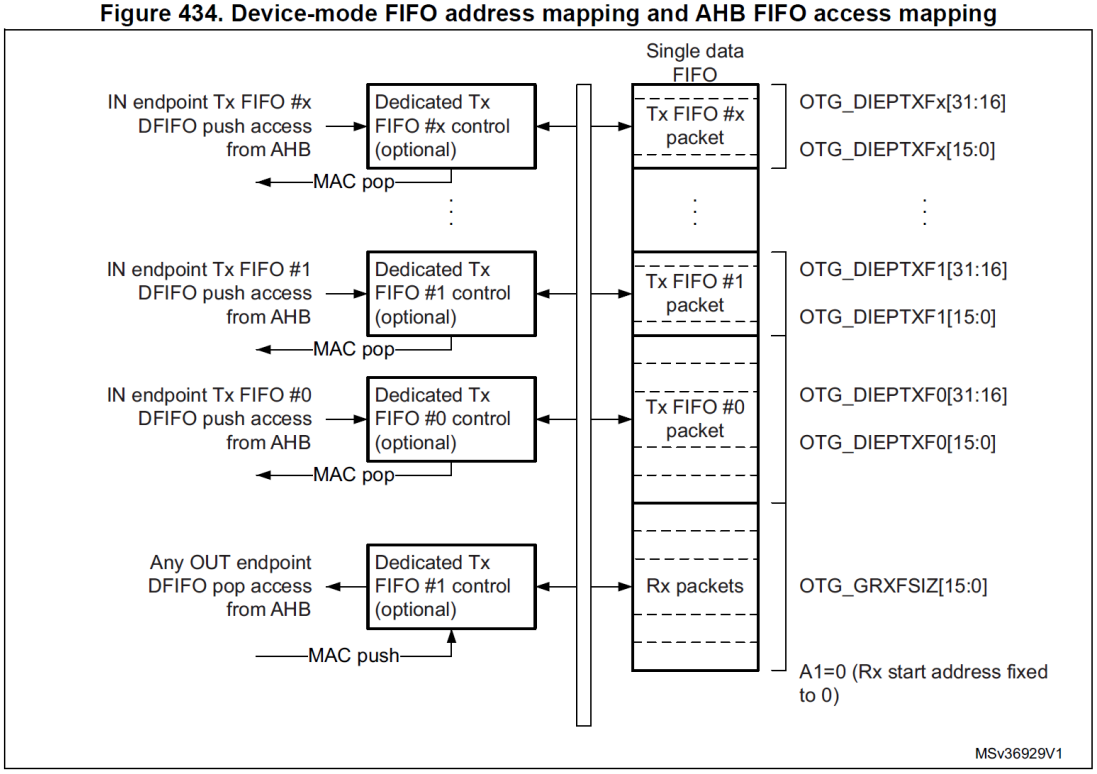


图41　OTG FIFO 框图（来自 STM32 规格书）

上图中可以看到，每个 IN（TX）端点都有一个独立的 FIFO，所有的 OUT（RX）端点共用一个 FIFO。因此在初始化 IN 端点时，需要为 IN 端点配置 FIFO 号。

5.3.1 FIFO 初始化

FIFO 使用前需要配置其大小和起始地址，对于 RX FIFO，起始地址固定为 0。TeenyUSB 中 FIFO 配置代码如下：

```
#define  SET_TX_FIFO(dev, bEpNum, addr, size) \
do{\
  if(bEpNum == 0){\
    GetUSB(dev)->DIEPTXF0_HNPTXFSIZ = (uint32_t)(((uint32_t)( (size)/4) << 16) | ( (addr)/4));\
  }else{\
                      /*avoid the compiler warning */\
```

```
    GetUSB(dev)->DIEPTXF[bEpNum==0?0: bEpNum- 1] = (uint32_t)(((uint32_t)( (size)/4) << 16) |
( (addr)/4));\
  }\
}while(0)

#define  SET_RX_FIFO(dev, addr, size) \
  do{\
    GetUSB(dev)->GRXFSIZ = (size/4);\
  }while(0)
```

端点 0 的 TX FIFO 在一个固定位置，其它端点的 TX FIFO 寄存器在 DIEPTXF 这个数组中。RX FIFO 的地址是固定的，因此只配置的大小。FIFO 寄存器中的大小单位是 4 字节，这里对 size 除以 4 后再写入寄存器中。TeenyDT 可以根据描述符的内容生成 OTG 模块的 FIFO 地址及大小，如下：

```
// Endpoint define for OTG core
#define BULK_OTG_MAX_OUT_SIZE                           (64)
#define BULK_OTG_CONTROL_EP_NUM                         (1)
#define BULK_OTG_OUT_EP_NUM                             (1)
// RX FIFO size / 4 > (CONTROL_EP_NUM * 5 + 8) +  (MAX_OUT_SIZE / 4 + 1) + (OUT_EP_NUM*2) + 1 = 33
#define BULK_OTG_RX_FIFO_SIZE                           (256)
#define BULK_OTG_RX_FIFO_ADDR                           (0)
// Sum of IN ep max packet size is 128
// Remain Fifo size is 768 in bytes, Rx Used 256 bytes
#define BULK_EP0_TX_FIFO_ADDR                           (256)
#define BULK_EP0_TX_FIFO_SIZE                           (BULK_EP0_TX_SIZE * 6)
#define BULK_EP1_TX_FIFO_ADDR                           (640)
#define BULK_EP1_TX_FIFO_SIZE                           (BULK_EP1_TX_SIZE * 6)
```

上面根据描述符的内容，自动生成了 FIFO 的地址和大小，将这些值通过 SET_TX_FIFO 和 SET_RX_FIFO 宏配置到 FIFO 寄存器中，完成 FIFO 配置。

为了避免 FIFO 中数据的状态不确定，在使用 FIFO 前会对 FIFO 进行 Flush 操作。代码如下：

```
void flush_rx(USB_OTG_GlobalTypeDef *USBx){
  USBx->GRSTCTL = USB_OTG_GRSTCTL_RXFFLSH;
  while ((USBx->GRSTCTL & USB_OTG_GRSTCTL_RXFFLSH) == USB_OTG_GRSTCTL_RXFFLSH);
}

void flush_tx(USB_OTG_GlobalTypeDef *USBx, uint32_t num){
  USBx->GRSTCTL = ( USB_OTG_GRSTCTL_TXFFLSH |(uint32_t)( num << 6));
  while ((USBx->GRSTCTL & USB_OTG_GRSTCTL_TXFFLSH) == USB_OTG_GRSTCTL_TXFFLSH);
}
```

当 TX FIFO 号为 0x10 时，会对所有的 TX FIFO 都进行 Flush。

5.3.2 读取 FIFO 数据

所有的 OUT（RX）端点共用一个 FIFO，当 RX FIFO 中有数据时，会触发 RXFLVL 中断，在这个中断中处理 FIFO 中的数据。代码如下：

```
void tusb_otg_device_handler(tusb_device_t* dev){
  ...
    if(INTR() & USB_OTG_GINTSTS_RXFLVL){
      USB_MASK_INTERRUPT(USBx, USB_OTG_GINTSTS_RXFLVL);
      {
        uint32_t sts = USBx->GRXSTSP;
        uint8_t EPn = sts & USB_OTG_GRXSTSP_EPNUM;
        uint32_t len = (sts & USB_OTG_GRXSTSP_BCNT) >> 4;
        if(((sts & USB_OTG_GRXSTSP_PKTSTS) >> 17) ==  STS_DATA_UPDT){
          tusb_ep_data* ep = &dev->Ep[EPn];
          if(ep->rx_count<ep->rx_size && ep->rx_buf){
            // copy data packet
            tusb_otg_read_data(USBx, ep->rx_buf + ep->rx_count, len);
            ep->rx_count += len;
          }else{
            // drop the data because no memory to handle them
            tusb_otg_read_data(USBx,0, len);
          }
        }else if(((sts & USB_OTG_GRXSTSP_PKTSTS) >> 17) ==  STS_SETUP_UPDT){
          // copy setup packet
          tusb_otg_read_data(USBx, &dev->setup, len);
        }
      }
      USB_UNMASK_INTERRUPT(USBx, USB_OTG_GINTSTS_RXFLVL);
    }
  ...
}
```

检测到 RX FIFO 中断后，先通过 GRXSTSP 寄存器得到 FIFO 中数据类型，如数据端点号，是否是 SETUP 包。然后通过 tusb_otg_read_data 将 FIFO 中的数据读取出。当读取 FIFO 中的数据时，对应端点的寄存器内容也会随之变化。如果 FIFO 中的数据满足完成条件，也会触发数据传输完成中断。例如读取的数据小于最大包长，或是数据已经达到传输总长度。关于传输完成条件的详细说明，见 2.4 传输完成条件这一节的内容。

5.3.3 写入 FIFO 数据

IN（TX）端点都有各自的 TX FIFO，通过 DIEPCTL 寄存器中的 TXFNUM 字段配置 FIFO 号。TeenyUSB 将 FIFO 号与端点号设置为一样的，方便后续的操作。当 TX FIFO 中没有数据时，会触发端点的 TXFE 中断。在 TXFE 中断将需要发送的数据写入到端点对应的 FIFO 中，代码如下：

```
void tusb_otg_device_handler(tusb_device_t* dev){
  ...
          if( ((epint & USB_OTG_DIEPINT_TXFE) == USB_OTG_DIEPINT_TXFE) && (USBx_DEVICE->DIEPEMPMSK &
(1 << ep)) ){
            tusb_fifo_empty(dev, ep);
          }
  ...
}
// when fifo empty call this function
void tusb_fifo_empty(tusb_device_t* dev, uint8_t EPn){
  tusb_ep_data* ep = &dev->Ep[EPn];
  PCD_TypeDef* USBx = GetUSB(dev);
  USB_OTG_INEndpointTypeDef* epin = USBx_INEP(EPn);
  uint32_t xfer_size = epin->DIEPTSIZ & USB_OTG_DIEPTSIZ_XFRSIZ;
  uint32_t maxpacket = GetInMaxPacket(dev, EPn);
  uint32_t fifo_len = epin->DTXFSTS & USB_OTG_DTXFSTS_INEPTFSAV;
  const uint8_t* src = (const uint8_t*)ep->tx_buf;
  uint32_t len32b;
  // round transfer size to max packet boundary
  if(xfer_size > fifo_len*4){
    xfer_size = (fifo_len*4 / maxpacket) * maxpacket;
  }
  len32b = (xfer_size+3) /4;
  // push data to fifo
  while(len32b){
    USBx_DFIFO(EPn) = *((__packed uint32_t *)src);
    src+=4;
    len32b--;
  }
  // adjust the dat buffer
  ep->tx_buf = src;
  //calculate last packet size
  ep->tx_last_size = xfer_size ? (xfer_size-1)%maxpacket+1 : 0;
  //if( xfer_size == 0){
  if( (epin->DIEPTSIZ & USB_OTG_DIEPTSIZ_XFRSIZ) == 0 ){
    // this xfer complete, so no need the fifo empty interrupt
    USBx_DEVICE->DIEPEMPMSK &= ~(1ul<<EPn);
  }
  return;
}
```

当检测到 TXFE 中断后，调用 tusb_fifo_empty 函数进行处理。根据端点需要发送的数据长度和 FIFO 空余空间，计算出当前能够传输到 FIFO 中的数据长度。然后将这些数据写入 TX FIFO 中。写入 TX FIFO 数据时， OTG 模块会减去对应端点的 DIEPTSIZ 寄存器中的值。如果 DIEPTSIZ 为 0，说明没有数据要发送了，屏蔽 TXFE 中断。当 FIFO 中的数据通过 USB

传输完成时，会触发 IN 端点的传输完成中断。

5.3.4 OTG_FS 与 OTG_HS 的 FIFO 初始化

STM32 的 OTG 模块有 FS 和 HS 两种，其中 HS 模块支持 DMA 功能，当启用 DMA 时对 FIFO 的设置有一些要求。为了应对这两种不同的设备的 FIFO 设置，TeenyUSB 会分别为 FS 和 HS 模块生成不同的 FIFO 初始化代码。

5.3.5 FIFO 小结

通过 FIFO，OTG 模块屏蔽掉了端点的一些细节。对于 OUT 端点，程序只需要读取 RX FIFO 中的数据，并写入对应的端点缓存中。对于 IN 端点，程序只需要在 TX FIFO 空中断中，将端点缓存中的数据写入 FIFO 中。

5.4 端点初始化

端点在使用前需要配置类型，最大包长，FIFO 号等内容，TeenyUSB 中的 OTG 模块的端点初始化代码如下：

```
#define init_ep_tx(dev, EPn, type, mps)                                          \
do{                                                                              \
  PCD_TypeDef *USBx = GetUSB(dev);                                               \
  uint32_t maxpacket = mps;                                                      \
  USB_OTG_INEndpointTypeDef* INEp = USBx_INEP(EPn);                              \
  if(USBx == USB_OTG_FS && EPn == 0){                                            \
    maxpacket = __CLZ(maxpacket) - 25;                                           \
  }                                                                              \
  USBx_DEVICE->DAINTMSK |=  ( (USB_OTG_DAINTMSK_IEPM) & ( (1 << (EPn))) );       \
  INEp->DIEPCTL |= ((maxpacket & USB_OTG_DIEPCTL_MPSIZ ) | (type << 18 ) |       \
    ((EPn) << 22 ) | (USB_OTG_DIEPCTL_SD0PID_SEVNFRM) | (USB_OTG_DIEPCTL_USBAEP));\
}while(0)

#define init_ep_rx(dev, EPn, type, mps)                                          \
do{                                                                              \
  PCD_TypeDef *USBx = GetUSB(dev);                                               \
  uint32_t maxpacket = mps;                                                      \
  if(EPn == 0){                                                                  \
    maxpacket = __CLZ(maxpacket) - 25;                                           \
  }                                                                              \
  USBx_DEVICE->DAINTMSK |=  ( (USB_OTG_DAINTMSK_OEPM) & ( (1 << (EPn))<<16 ) );  \
  USBx_OUTEP(EPn)->DOEPCTL |= ((maxpacket & USB_OTG_DOEPCTL_MPSIZ ) | (type << 18 ) |\
    (USB_OTG_DOEPCTL_SD0PID_SEVNFRM) | (USB_OTG_DIEPCTL_USBAEP));               \
}while(0)
```

对于端点 0，最大包长的初始化方式有所不同，其余的都一样。在初始化端点时，同时也打开了对应端点的中断掩码位。TeenyDT 会根据描述符内容生成端点初始化的代码，OTG 模块的端点初始化代码如下：

```
// PMA buffer reserved for buffer description table
#define BULK_USB_BUF_START                              (BULK_EP_BUF_DESC_TABLE_SIZE *
BULK_EP_NUM)

// EndPoints 0 defines
#define BULK_EP0_RX_SIZE                                (64)
#define BULK_EP0_RX_ADDR                                (BULK_USB_BUF_START + 0)
#define BULK_EP0_TX_SIZE                                (64)
#define BULK_EP0_TX_ADDR                                (BULK_USB_BUF_START + 64)
#define BULK_EP0_TYPE                                   USB_EP_CONTROL

// EndPoints 1 defines
#define BULK_EP1_TX_SIZE                                (64)
#define BULK_EP1_TX0_ADDR                               (BULK_USB_BUF_START + 128)
#define BULK_EP1_TX1_ADDR                               (BULK_USB_BUF_START + 192)
#define BULK_EP1_TYPE                                   USB_EP_BULK

// EndPoints 2 defines
#define BULK_EP2_RX_SIZE                                (64)
#define BULK_EP2_RX0_ADDR                               (BULK_USB_BUF_START + 256)
#define BULK_EP2_RX1_ADDR                               (BULK_USB_BUF_START + 320)
#define BULK_EP2_TYPE                                   USB_EP_BULK

// Endpoint define for OTG core
#define BULK_OTG_MAX_OUT_SIZE                           (64)
#define BULK_OTG_CONTROL_EP_NUM                         (1)
#define BULK_OTG_OUT_EP_NUM                             (1)
// RX FIFO size / 4 > (CONTROL_EP_NUM * 5 + 8) +  (MAX_OUT_SIZE / 4 + 1) + (OUT_EP_NUM*2) + 1 = 33
#define BULK_OTG_RX_FIFO_SIZE                           (256)
#define BULK_OTG_RX_FIFO_ADDR                           (0)
// Sum of IN ep max packet size is 128
// Remain Fifo size is 768 in bytes, Rx Used 256 bytes
#define BULK_EP0_TX_FIFO_ADDR                           (256)
#define BULK_EP0_TX_FIFO_SIZE                           (BULK_EP0_TX_SIZE * 6)
#define BULK_EP1_TX_FIFO_ADDR                           (640)
#define BULK_EP1_TX_FIFO_SIZE                           (BULK_EP1_TX_SIZE * 6)

// EndPoints init function for USB OTG core
#define BULK_TUSB_INIT_EP_OTG(dev) \
  do{\
    SET_RX_FIFO(dev, BULK_OTG_RX_FIFO_ADDR, BULK_OTG_RX_FIFO_SIZE);  \
```

```
  /* Init ep0 */ \
  INIT_EP_Tx(dev, PCD_ENDP0, BULK_EP0_TYPE, BULK_EP0_TX_SIZE);  \
  SET_TX_FIFO(dev, PCD_ENDP0, BULK_EP0_TX_FIFO_ADDR, BULK_EP0_TX_FIFO_SIZE);  \
  INIT_EP_Rx(dev, PCD_ENDP0, BULK_EP0_TYPE, BULK_EP0_RX_SIZE);  \
  /* Init ep1 */ \
  INIT_EP_Tx(dev, PCD_ENDP1, BULK_EP1_TYPE, BULK_EP1_TX_SIZE);  \
  SET_TX_FIFO(dev, PCD_ENDP1, BULK_EP1_TX_FIFO_ADDR, BULK_EP1_TX_FIFO_SIZE);  \
  /* Init ep2 */ \
  INIT_EP_Rx(dev, PCD_ENDP2, BULK_EP2_TYPE, BULK_EP2_RX_SIZE);  \
}while(0)
```

5.5 响应 Reset 事件与初始化端点

TeenyUSB 协议栈收到 Reset 事件后，将设备地址设置为 0，并且重新初始化所有端点。按照 USB 规范，应当只初始化端点 0，在设置配置请求（Set Configuration Request）中设置其他端点。TeenyUSB 采用了简化的处理方式，配置了所有的端点，原因见配置设备这一节中的内容。Reset 后设备就能与主机通过默认地址 0 进行通讯了。基于前面的原因，初始化端点与 Reset 处理放到了一起来进行介绍。

TeenyUSB 与 HAL 库处理 Reset 事件和配置端点的流程如下图所示：

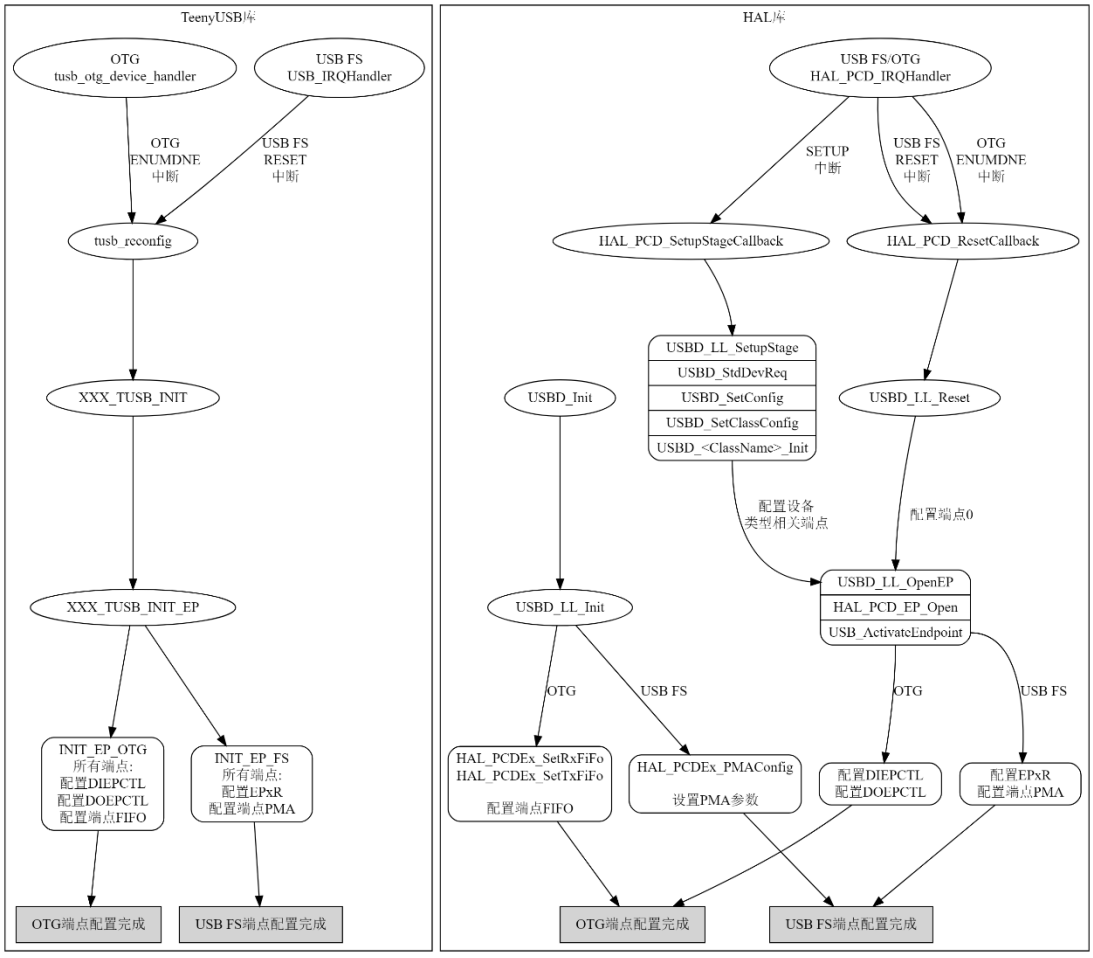


图42　端点配置

上图中包含了 USB FS 模块的端点初始化流程，下面介绍 TeenyUSB 中初始化端点的代码，对代码熟悉的读者可以跳过本节后续内容。

5.5.1 TeenyUSB 库 Reset 事件处理与端点初始化

在 OTG 模块中，Reset 事件分成了两部分来处理，一部分在 USBRST 中断中处理，一部分在 ENUMDNE 中断中处理。代码如下：

```
void tusb_otg_device_handler(tusb_device_t* dev){
  ...
    if(INTR() & USB_OTG_GINTSTS_USBRST ){
      uint32_t i;
      USBx_DEVICE->DCTL &= ~USB_OTG_DCTL_RWUSIG;
      flush_tx(USBx, 0x10);
      for (i = 0; i < MAX_EP_NUM ; i++){
        USBx_INEP(i)->DIEPINT = 0xFF;
        USBx_INEP(i)->DIEPCTL &= ~USB_OTG_DIEPCTL_STALL;
        USBx_INEP(i)->DIEPCTL |= USB_OTG_DIEPCTL_EPDIS;
        USBx_OUTEP(i)->DOEPINT = 0xFF;
        USBx_OUTEP(i)->DOEPCTL &= ~USB_OTG_DOEPCTL_STALL;
        USBx_OUTEP(i)->DOEPCTL |= USB_OTG_DOEPCTL_EPDIS;
      }
      USBx_DEVICE->DAINT = 0xFFFFFFFF;
      USBx_DEVICE->DAINTMSK |= 0x10001;
      {
        USBx_DEVICE->DOEPMSK |= (USB_OTG_DOEPMSK_STUPM | USB_OTG_DOEPMSK_XFRCM |
USB_OTG_DOEPMSK_EPDM);
        USBx_DEVICE->DIEPMSK |= (USB_OTG_DIEPMSK_TOM | USB_OTG_DIEPMSK_XFRCM |
USB_OTG_DIEPMSK_EPDM);
      }
      /* Set Default Address to 0 */
      USBx_DEVICE->DCFG &= ~USB_OTG_DCFG_DAD;
      USBx->GINTSTS = USB_OTG_GINTSTS_USBRST;
    }
    /* Handle Enumeration done Interrupt */
    if(INTR() & USB_OTG_GINTSTS_ENUMDNE ){
      tusb_reconfig(dev);
      USBx->GUSBCFG &= ~USB_OTG_GUSBCFG_TRDT;
      switch(USBx_DEVICE->DSTS & USB_OTG_DSTS_ENUMSPD){
        case DSTS_ENUMSPD_HS_PHY_30MHZ_OR_60MHZ:
          USBx->GUSBCFG |= (uint32_t)((USBD_HS_TRDT_VALUE << 10) & USB_OTG_GUSBCFG_TRDT);
          break;
        case DSTS_ENUMSPD_LS_PHY_6MHZ:
          USBx_INEP(0)->DIEPCTL |= 3; // force ep0 packet size to 8 when in LS mode
        case DSTS_ENUMSPD_FS_PHY_30MHZ_OR_60MHZ:
```

```
        case DSTS_ENUMSPD_FS_PHY_48MHZ:
          USBx->GUSBCFG |= (uint32_t)((0x6 << 10) & USB_OTG_GUSBCFG_TRDT);
          break;
      }
      /* setup EP0 to receive SETUP packets */
      tusb_otg_device_prepare_setup(dev);
      USBx_DEVICE->DCTL |= USB_OTG_DCTL_CGINAK;
      USBx->GINTSTS = USB_OTG_GINTSTS_ENUMDNE;
    }
  ...
}
```

在 RESET 中断中，重新初始化了所有的端点，并将设备地址设置位默认地址 0。

在 ENUMDNE 中断中，调用 tusb_reconfig 回调函数，对端点和 FIFO 进行配置。启动端点 0 上的 Setup 包接收。

5.6 端点数据处理

与 USB FS 模块不同，OTG 模块中的数据处理是以传输为基础。在 USB FS 模块中，一次事务就会触发一次 CTR 中断。而在 OTG 模块中，一次完整的传输触发 XFRC 中断，在 XFRC 中断中进行端点的数据处理。当传输的数据有短包或是传输的数据长度与预设一样时，触发传输完成条件。详细传输完成条件见 2.4 传输完成条件。

OTG 模块中的数据传输通过 FIFO 进行，在 RXFLVL 中断中接收数据，在 TXFE 中断中发送数据。由于 FIFO 的存在，端点传输事件的处理和数据的搬移操作被分开了。OTG 模块的端点数据处理流程如下图所示：

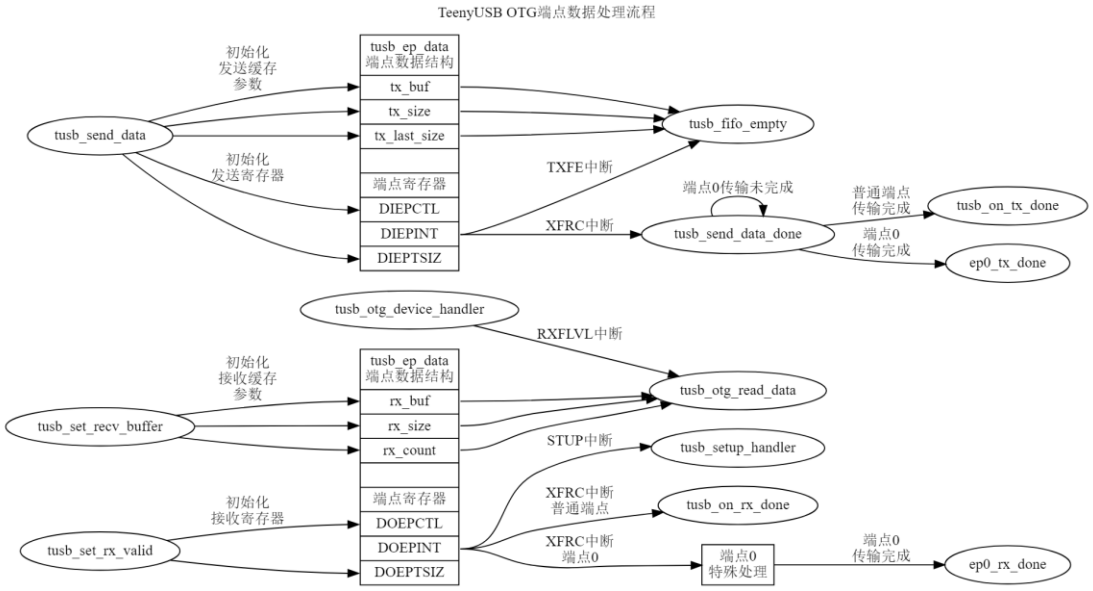


图43　　TeenyUSB OTG 设备模式端点数据处理

从上图中可以看到，数据的处理分别在 tusb_otg_read_data 和 tusb_fifo_empty 函数中，而传输完成的处理与数据处理独立开了。

上图中还可以看到，端点 0 是个例外，有特殊的处理流程。这是因为端点 0 的传输数据寄存器与其它不同，能够处理的数据有限。在这里将端点 0 的传输长度设置成了最大包长，这样没发送完一包数据会触发一次完成中断。在完成中断中再根据剩余数据的大小，决定是否需要继续传输数据。

5.6.1 OUT 端点数据处理

OUT 端点数据处理代码如下：

```
if(INTR() & USB_OTG_GINTSTS_OEPINT){
  uint32_t ep_intr = ((USBx_DEVICE->DAINT & USBx_DEVICE->DAINTMSK)) >> 16;
  uint8_t EPn = 0;
  while(ep_intr){
    if(EPn >= MAX_EP_NUM){
      break;
    }
    if (ep_intr & 0x1){
      uint32_t epint = USBx_OUTEP(EPn)->DOEPINT;
      tusb_ep_data* ep = &dev->Ep[EPn];
      if(( epint & USB_OTG_DOEPINT_XFRC) == USB_OTG_DOEPINT_XFRC){
        uint32_t maxpacket = GetOutMaxPacket(dev, EPn);
        if(EPn == 0){
          if(ep->rx_count == 0 || ep->rx_count >= ep->rx_size || ep->rx_count % maxpacket){
            if(dev->ep0_rx_done){
              dev->ep0_rx_done(dev);
              dev->ep0_rx_done = 0;
            }
            ep->rx_buf = 0;
            // prepare ep0 to recv next setup packet
            tusb_otg_device_prepare_setup(dev);
          }else{
            ep->rx_buf += ep->rx_count;
            ep->rx_size -= ep->rx_count;
            tusb_set_rx_valid(dev, EPn);
          }
        }else{
          if(tusb_on_rx_done(dev, EPn, ep->rx_buf, ep->rx_count) == 0){
            ep->rx_count = 0;
            tusb_set_rx_valid(dev, EPn);
          }else{
            ep->rx_count = ep->rx_size;
          }
        }
      }
      // Read the DOEPINT again, to make sure the SETUP flag is set
```

```
        epint = USBx_OUTEP(EPn)->DOEPINT;
        if(( epint & USB_OTG_DOEPINT_STUP) == USB_OTG_DOEPINT_STUP){
          ep->rx_buf = 0;
          tusb_setup_handler(dev);
          if(ep->rx_buf){
            // rx_buf is not null, means setup need write some data
            tusb_set_rx_valid(dev, EPn);
          }else{
            // otherwise prepare recv setup packet again
            tusb_otg_device_prepare_setup(dev);
          }

        }
        // clear all interrupt flags
        USBx_OUTEP(EPn)->DOEPINT = epint;
      }
      ep_intr>>=1;
      EPn+=1;
    }
  }
```

最外层是下面的一个循环结构：

```
  if(INTR() & USB_OTG_GINTSTS_OEPINT){
    uint32_t ep_intr = ((USBx_DEVICE->DAINT & USBx_DEVICE->DAINTMSK)) >> 16;
    uint8_t EPn = 0;
    while(ep_intr){
      if(EPn >= MAX_EP_NUM){
        break;
      }
...
      ep_intr>>=1;
      EPn+=1;
    }
```

在 OTG 模块中，所有端点的中断标志都合成在了一个寄存器中，通过查询这个寄存器中的标志位，就能知道第几号端点有中断发生。然后再根据端点的对应寄存器，得到中断产生的具体原因。

端点 0 处理方式，如果接收缓存已满或者当前包为短包，说明传输完成，执行注册在 ep0_rx_done 上的函数，并且重新开始接收 SETUP 包。否则调整缓存位置，继续接收下一包数据。如果是其它端点，则直接调用 tusb_on_rx_done 回调函数。如果是 SETUP 包，则执行 tusb_setup_handler，执行 SETUP 处理流程。

OUT 端点的数据搬移是在 FIFO 相关的处理函数中进行的，见 5.3.2 读取 FIFO 数据。

在 TeenyUSB 中，通过 tusb_set_rx_valid 函数使能 OUT 端点，函数实现代码如下：

```
void tusb_set_rx_valid(tusb_device_t* dev, uint8_t EPn){
  PCD_TypeDef* USBx = GetUSB(dev);
  USB_OTG_OUTEndpointTypeDef* epout = USBx_OUTEP(EPn);
```

```
  tusb_ep_data* ep = &dev->Ep[EPn];
  uint32_t maxpacket = GetOutMaxPacket(dev, EPn);
  uint32_t pktCnt;
  uint32_t len = ep->rx_size;
  ep->rx_count = 0;
  if(EPn == 0){
    // EP0 always recv 1 packet
    if(len > maxpacket){
      len = maxpacket;
    }
  }
  pktCnt = (((len + maxpacket - 1) / maxpacket));
  if(pktCnt == 0){
    // avoid zero packet count, used to send ZLP(zero length packet)
    pktCnt = 1;
  }
  if(len) len = pktCnt * maxpacket;
  // clear and set the EPT size field
  epout->DOEPTSIZ =  (pktCnt<<19) | len;
  if(USBx->GAHBCFG & USB_OTG_GAHBCFG_DMAEN){
    epout->DOEPDMA = (uint32_t)ep->rx_buf;
  }
  if(ISO_EP && ((epout->DOEPCTL & USB_OTG_DOEPCTL_EPTYP)  ==
((USB_EP_ISOCHRONOUS)<<(USB_OTG_DOEPCTL_EPTYP_Pos))) ){
    if ((USBx_DEVICE->DSTS & ( 1 << 8 )) == 0){
      epout->DOEPCTL |= USB_OTG_DOEPCTL_SODDFRM;
    }else{
      epout->DOEPCTL |= USB_OTG_DOEPCTL_SD0PID_SEVNFRM;
    }
  }
  epout->DOEPCTL |= (USB_OTG_DOEPCTL_CNAK | USB_OTG_DOEPCTL_EPENA);
}
```

与 USB FS 模块中的 tusb_set_rx_valid 函数相比，这个函数复杂了很多。这是因为 OTG 模块会自动根据传输总长度和端点的最大包长来判定传输是否完成，所以在初始化时，要事先配置缓存的长度。

5.6.2 IN 端点数据处理

IN 端点中断数据处理代码如下：

```
        USB_OTG_INEndpointTypeDef* epin = USBx_INEP(ep);
        uint32_t epint = epin->DIEPINT;
        // Xfer complete interrupt handler
        if(epint & USB_OTG_DIEPINT_XFRC){
          USBx_DEVICE->DIEPEMPMSK &= ~(0x1ul << ep);
```

```
            epin->DIEPINT = USB_OTG_DIEPINT_XFRC;
            tusb_send_data_done(dev, ep);
          }
          // FIFO empty interrupt handler
          if( ((epint & USB_OTG_DIEPINT_TXFE) == USB_OTG_DIEPINT_TXFE) && (USBx_DEVICE->DIEPEMPMSK &
(1 << ep)) ){
            tusb_fifo_empty(dev, ep);
          }
```

这里没有写出外层循环的代码。如果是传输完成，则调用 tusb_send_data_done 函数。由于每个 IN 端点都有独立的 FIFO，因此还需要在每个 IN 端点中处理 FIFO 相关的事件。TX FIFO 对于应用程序而言只能写数据，因此在 IN 端点需要发送数据时，打开 TX FIFO 的 EMPTY 中断，这样产生中断后就能向 FIFO 中写数据了。TX FIFO 处理函数详细内容见 5.3.3 写入 FIFO 数据。IN 端点数据传输完成处理函数如下：

```
// called by the ep data interrupt handler when last packet tranfer done
void tusb_send_data_done(tusb_device_t* dev, uint8_t EPn){
  PCD_TypeDef* USBx = GetUSB(dev);
  tusb_ep_data* ep = &dev->Ep[EPn];
  uint32_t maxpacket = get_max_in_packet_size(USBx, EPn);
  //track(EPn, ep->tx_size, 3, ep->tx_last_size);
  if(EPn == 0){
    if(ep->tx_size){
      tusb_send_data(dev, EPn, ep->tx_buf, ep->tx_size);
    }else if(ep->tx_last_size == maxpacket){
      // Send a ZLP
      tusb_send_data(dev, EPn, ep->tx_buf, 0);
    }else if(dev->ep0_tx_done){
      // invoke status transmitted call back for ep0
      dev->ep0_tx_done(dev);
      dev->ep0_tx_done = 0;
    }
  }else{
    // clear the fifo empty mask
    tusb_on_tx_done(dev, EPn);
  }
}
```

如果是端点 0，根据缓存中剩余数据大小或是最后一包数据的包长再次调用 tusb_send_data 发送数据。如果是其他端点，直接调用 tusb_on_tx_done 回调函数。

对于非 0 端点而言，OTG 模块中的传输完成处理非常简单，根据中断信号调用相应的回调函数即可。

IN 端点的数据发送函数代码如下：

```
int tusb_send_data(tusb_device_t* dev, uint8_t EPn, const void* data, uint16_t len){
  PCD_TypeDef* USBx = GetUSB(dev);
  tusb_ep_data* ep = &dev->Ep[EPn];
  uint32_t maxpacket = GetInMaxPacket(dev, EPn);
```

```
  uint32_t pktCnt;
  uint32_t total_len = len;
  USB_OTG_INEndpointTypeDef* epin = USBx_INEP(EPn);
  ep->tx_buf = (const uint8_t*)data;
  if(epin->DIEPCTL & USB_OTG_DIEPCTL_EPENA){
    return -1;
  }
  if(EPn == 0){
    // EP0 always send 1 packet
    if(len > maxpacket){
      len = maxpacket;
    }
  }
  // set remain size of tx buffer, current tx size is saved in the DIEPTSIZ register
  ep->tx_size = total_len - len;
  // calculate last packet size
  ep->tx_last_size = len ? (len-1) % maxpacket + 1 : 0;

  pktCnt = (((len + maxpacket - 1) / maxpacket)<<19);
  if(pktCnt == 0){
    // avoid zero packet count, used to send ZLP(zero length packet)
    pktCnt = 1<<19;
  }
  // clear and set the EPT size field
  epin->DIEPTSIZ =  pktCnt| len;
  if(USBx->GAHBCFG & USB_OTG_GAHBCFG_DMAEN){
    epin->DIEPDMA = (uint32_t)ep->tx_buf;
  }else{
    if(len > 0){
      USBx_DEVICE->DIEPEMPMSK |= (1 << EPn);
    }
  }
  // do not fill data here, data will be filled in the empty interrupt
  //copy_data_to_fifo(dev, ep, EPn, data, len, total_len);
  epin->DIEPCTL |= (USB_OTG_DIEPCTL_CNAK | USB_OTG_DIEPCTL_EPENA);
  return 0;
}
```

这个函数中主要是进行数据包长度的计算，设置包数量和数据长度，然后启动端点。对于端点 0，一次最多发送最大包长的数据量。如果没有开启 DMA，并且包长不为 0，开启端点的 FIFO EMPTY 中断，在 EMPTY 中断中搬移数据到 FIFO 中。这里也可以进行数据搬移，如果 FIFO 剩余空间足够大，并且数据全部搬移成功，也可以不打开 FIFO EMPTY 中断。在 HAL 库的主机端代码就是发送函数中搬移数据的，不过 HAL 库主机部分的代码又走向了另一个极端，他只在发送函数中进行数据搬移，没有启动 EMPTY 中断来搬移剩余的数据。

5.7 DMA 操作

高速的 OTG 模块具备 DMA 的功能，数据的搬移可以由 OTG 模块的 DMA 控制器来完成。有些芯片的 OTG DMA 不能从 Flash 中搬移数据，对于描述符这样的静态数据而言，需要先搬移到 RAM 中再进行发送。HAL 库中将所有的描述符都定义成为了 RAM 数据，因此不存在这个问题。

5.7.1 OUT 端点 DMA

未开启 DMA 时，OUT 端点的数据接收在 RXFLVL 中断中完成，开启 DMA 后，不需要 RXFLVL 中断来搬移数据，代码如下：

```
  if(USBx->GAHBCFG & USB_OTG_GAHBCFG_DMAEN){
    // If DMA enabled, setup the threshold value
    USBx_DEVICE->DTHRCTL = (USB_OTG_DTHRCTL_TXTHRLEN_6 | USB_OTG_DTHRCTL_RXTHRLEN_6);
    USBx_DEVICE->DTHRCTL |= (USB_OTG_DTHRCTL_RXTHREN | USB_OTG_DTHRCTL_ISOTHREN |
USB_OTG_DTHRCTL_NONISOTHREN);
  }else{
    USBx->GINTMSK |= USB_OTG_GINTMSK_RXFLVLM;
  }
```

未开启 DMA 时，接收数据的实际长度在 RXFLVL 中断中更新，开启 DMA 后，直到数据传输完成才会通知应用程序，因此在传输完成中断中计算实际接收到的数据长度。代码如下：

```
        uint32_t maxpacket = GetOutMaxPacket(dev, EPn);
        if(USBx->GAHBCFG & USB_OTG_GAHBCFG_DMAEN){
          // Calculate recv data length from the XFRSIZ field
          uint32_t total_xfer_size;
          if(EPn == 0){
            total_xfer_size = maxpacket;
          }else{
            total_xfer_size = ((ep->rx_size + maxpacket - 1) / maxpacket) * maxpacket;
          }
          // DMA enabled, recv data count is total_xfer_size minus transfer remain length
          ep->rx_count += total_xfer_size - (USBx_OUTEP(EPn)->DOEPTSIZ & USB_OTG_DOEPTSIZ_XFRSIZ);
        }
```

对于端点 0，需要传输数据的总长度为最大包长，对于其他端点，需要传输数据的总长度为缓存长度圆整到最大包长的值。DOEPTSIZ 中的 XFRSIZ 字段在初始化时设置为需要传输的数据总长度，当接收到数据后，会自动递减。到这里时 XFRSIZ 字段中表示总长度还剩余的字节数。实际接收的数据长度=总长度-剩余长度。

这里有还一点问题，当收到 SETUP 包时，SETUP 包只有 8 字节，但是这里是用最大包长去计算的，因此 SETUP 包计算出的实际长度会不正确。不过这里没有关系，SETUP 包始终为 8 字节，不会用这里计算出的长度。

5.7.2 SETUP 包的 DMA 处理

在没有启动 DMA 的时候，SETUP 包在 RXFLVL 中断中进行处理，根据 FIFO 中的数据标志，将 SETUP 包复制到设备的 SETUP 缓存中。而端点无论状态如何，SETUP 传输总会成功。因此在没有启动 DMA 时，不用去关心 SETUP 包的接收问题。

而启用 DMA 后，SETUP 包会放在端点 0 的 DOEPDMA 寄存器指向的地址中，DMA 传输后会自动更新 DOEPDMA 的值，因此需要在适当的时候更新 DOEPDMA 中的值，确保 SETUP 包的正确接收。TeenyUSB 通过 tusb_otg_device_prepare_setup 函数设置 SETUP 接收缓存的地址。

5.7.3 IN 端点 DMA

IN 端点的 DMA 数据处理比较简单，如果启动了 DMA，只需要将缓存地址写入 DMA 寄存器即可，代码如下：

```
int tusb_send_data(tusb_device_t* dev, uint8_t EPn, const void* data, uint16_t len){
...
  if(USBx->GAHBCFG & USB_OTG_GAHBCFG_DMAEN){
    epin->DIEPDMA = (uint32_t)ep->tx_buf;
  }else{
    if(len > 0){
      USBx_DEVICE->DIEPEMPMSK |= (1 << EPn);
    }
  }
...
}
```

5.8 USB OTG 使用小结

TeenyUSB 中 USB OTG 设备的使用流程如下图所示：

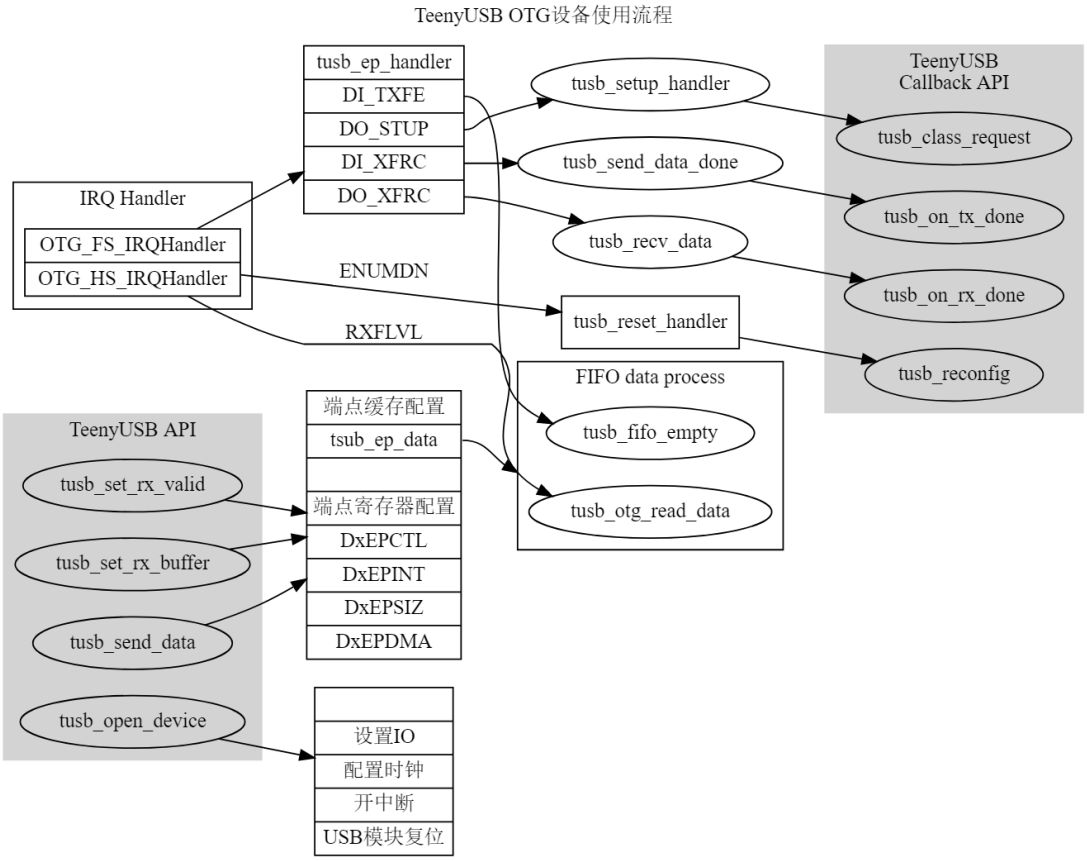


图44　　USB OTG 设备模式使用流程

上图中灰色部分是 TeenyUSB 提供的 API，有两类 API，一类是普通函数，一类是回调函数。TeenyUSB 中，USB 设备的开发都是通过上述的普通函数和回调函数来完成的，具体的 USB 设备开发过程见第 6 章 USB 设备开发。

5.8.1 普通函数

普通函数由应用程序主动调用，定义见下表：

表22　　TeenyUSB 普通函数

普通函数	说明
tusb_get_device	获得设备，在开始使用USB设备前，调用此函数获得设备对象，供后面的函数使用，上图中未画出
tusb_open_device	打开USB模块，将设备设置为连接状态
tusb_close_device	关闭USB模块，将设备设置为断开状态，上图中未画出
tusb_send_data	发送数据
tusb_set_rx_buffer	设置接收缓存，启动端点发送功能
tusb_set_rx_valid	启动端点接收功能，根据tusb_set_rx_buffer中设置的内容对端点进行初始化

5.8.2 回调函数

当特定事件发生时调用回调函数，TeenyUSB 中回调函数为 weak 类型，应用程序中实

现后调用应用程序中的版本，如果应用程序未实现则调用默认版本。回调函数定义见下表：

表23　　TeenyUSB 回调函数

回调函数	说明
tusb_reconfig	接收到Reset中断时调用，在这个函数中初始化所有端点，初始化设备的所有描述符
tusb_class_request	接收到设备类请求时调试，在这个函数中处理特定设备类型相关的请求.返回0说明请求需要协议栈继续处理。返回1表示请求已经处理，不需要协议栈处理。
tusb_on_rx_done	OUT传输完成时调用，此函数返回0表示缓存中数据处理完成，缓存可以接收后续的包。此函数返回非0，表述缓存数据未处理，接收挂起，当数据处理完毕后调用tusb_set_rx_valid重新使能端点接收
tusb_on_tx_done	IN端点传输完成时调用，表示数据传输完成
tusb_delay_ms	以毫秒为单位的延时函数，由应用程序根据平台实现

5.8.3 USB FS 与 OTG 设备对比

5.8.3.1 端点数据处理

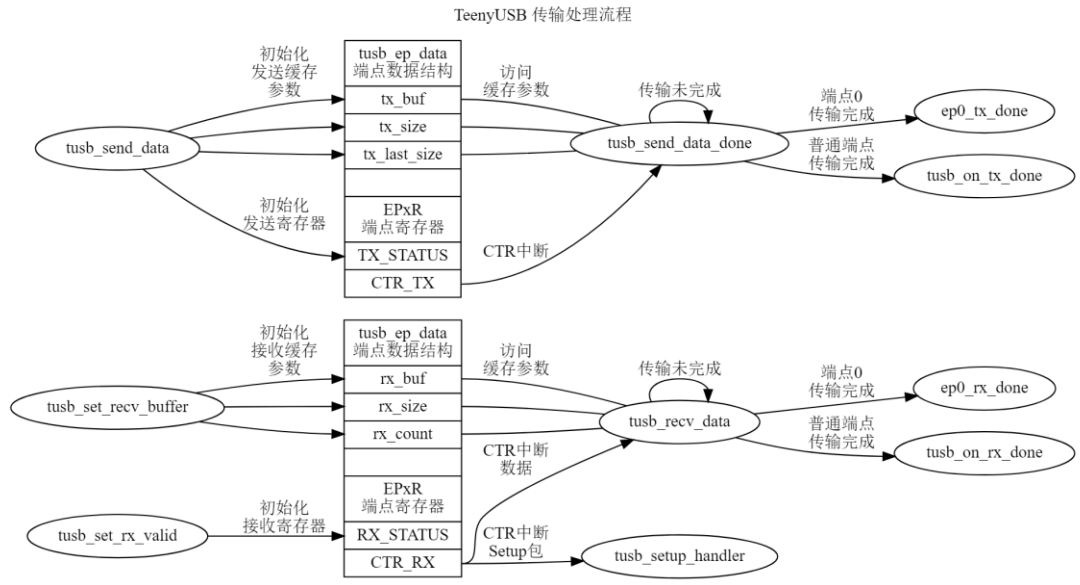


图45　　USB FS 模块端点数据处理

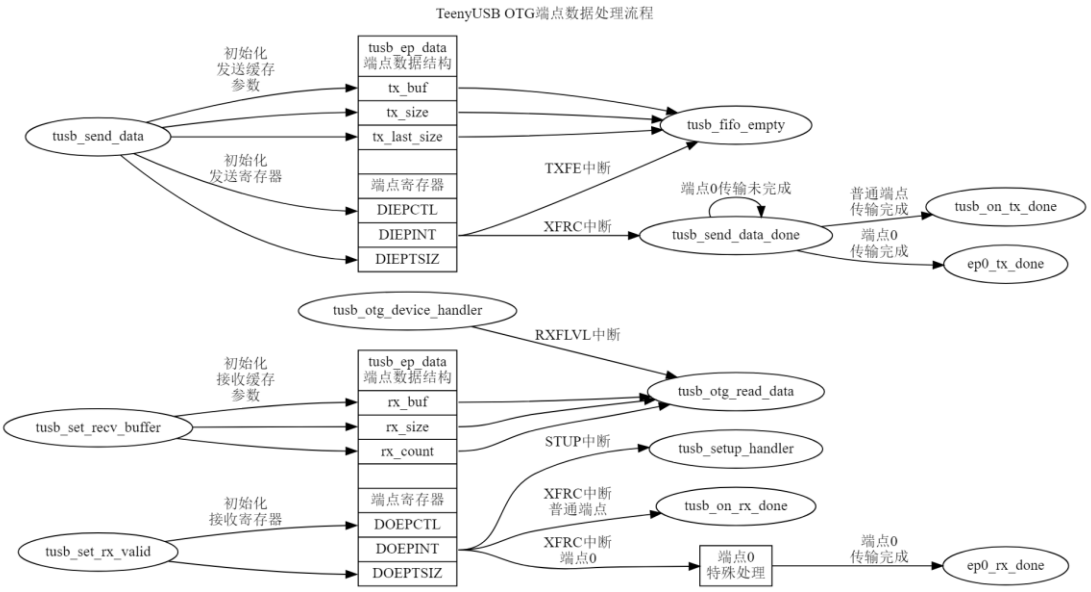


图46　　OTG 模块端点数据处理

上面两图中可以看到 USB FS 与 OTG 处理流程基本类似。不同的是，在 USB FS 的 tusb_send_data_done 函数和 tusb_recv_data 函数中，不但要判断传输是否完成，还要进行数据搬移。而 OTG 模块中，数据搬移分别在 tusb_fifo_empty 和 tusb_org_read_data 中处理，这两个函数不会判断传输是否完成，只做数据搬移。当高速的 OTG 模块启用了 DMA 后，tusb_fifo_empty 和 tusb_org_read_data 这两个函数也可以不用了。

6 USB 设备实例

6.1 TeenyUSB 协议栈使用说明

在设计具体的 USB 设备时，一般流程是：先规划设备的功能，确定设备描述符，实现设备类型相关的请求处理，处理设备业务数据。当使用 TeenyUSB 时，可以采用下面的设计流程：

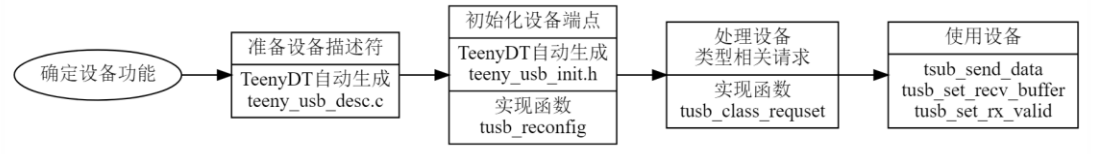


图47　　USB 设备开发流程

6.1.1 TeenyUSB 文件结构

TeenyUSB 协议栈代码在 usb_stack 目录下，其中各文件功能说明如下表：

表24　　TeenyUSB 文件结构及功能

目录	文件名	说明
usb_stack/inc	teeny_usb.h	TeenyUSB协议栈的数据结构定义和API声明文件，使用TeenyUSB协议栈前包含此文件
usb_stack/inc	usbd_conf.h	HAL库USB协议栈配置文件，在这里包含teeny_usb_init.h文件
usb_stack/inc	teeny_usb_platform.h	芯片平台定义选择文件，当需要新增加支持的芯片时，修改此文件，增加芯片平台定义
usb_stack/inc	stm32_otg_platform.h	OTG平台定义文件，在此文件中定义对芯片寄存器的相关操作，如设置端点类型，设置数据长度等。适用于STM32F105/107/2xx/4xx/7xx等具有OTG模块的芯片
usb_stack/inc	stm32_fs_platform.h	FS平台定义文件，在此文件中定义对芯片寄存器的相关操作，如设置端点类型，设置数据长度等。适用于STM32F/0xx/103/3xx等具有USB FS模块的芯片
usb_stack/src	teeny_usb.c	USB设备标准请求实现文件
usb_stack/src	teeny_usb_stm32_fs_device.c	STM32 USB FS模块操作实现文件
usb_stack/src	teeny_usb_stm32_otg_device.c	STM32 OTG模块设备模式基本操作实现文件
usb_stack/src	stm32f_otg_init.c	STM32芯片OTG模块初始化相关操作实现文件，在此文件中实现IO初始化，USB模块初始化，中断入口处理等
usb_stack/src	stm32f_fs_init.c	STM32芯片FS模块初始化相关操作实现文件，在此文件中实现IO初始化，USB模块初始化，中断入口处理等
开发板相关目录	startup_stm32f<xxx>.s	芯片相关启动文件
开发板相关目录	system_stm32f<xxx>.c	芯片相关时钟配置文件
开发板相关目录	stm32f<xx>_hal_conf.h	芯片相关HAL库功能定义文件
应用相关目录	teeny_usb_init.h	TeenyDT生成的端点初始化头文件，在usbd_conf.h中包含
应用相关目录	teeny_usb_desc.c	TeenyDT生成的描述符文件
应用相关目录	<xxx>_desc.lua	TeenyDT格式的描述符定义文件
应用相关目录	<xxx>_class.c	设备类型相关请求的实现文件

TeenyUSB 寄存器定义部分采用 HAL 库的定义，因此 TeenyUSB 工程需要依赖 HAL 库以及 STM32 的 USB 设备库（STM32_USB_Device_Library）。在使用 HAL 库时，需要用 stm32fxxx_hal_conf.h 文件来定义 HAL 库的功能。使用 HAL 版本的 USB 库时，需要 usbd_confg.h 文件定义 USB 库的功能。在 TeenyUSB 中，端点相关的定义由 TeenyDT 自动生成在 teeny_usb_init.h 文件中，因此在 usbd_confg.h 文件中增加包含 teeny_usb_init.h。在工程路径中增加 HAL 库 USB 库的相关路径。

6.2 自定义 USB 设备

用户自定义设备是最简单的一种设备类型，也是最复杂的一种设备类型。简单，是对设备端而言，自定义设备类型用到的描述符最简单，设备结构也简单，不需要处理设备相关的特殊请求。复杂，是对 PC 端而言，需要为自定义的设备开发驱动程序，驱动的开发很是繁琐。不过既然我们从自定义设备开始，就要先化繁为简，在本节后面我们会想办法省掉驱动的编写过程，最好是连 INF 文件都不用写，真正做到即插即用。自定义 USB 设备的完整代码在 demo/custom_bulk 目录中。自定义 USB 设备工程结构如下图：

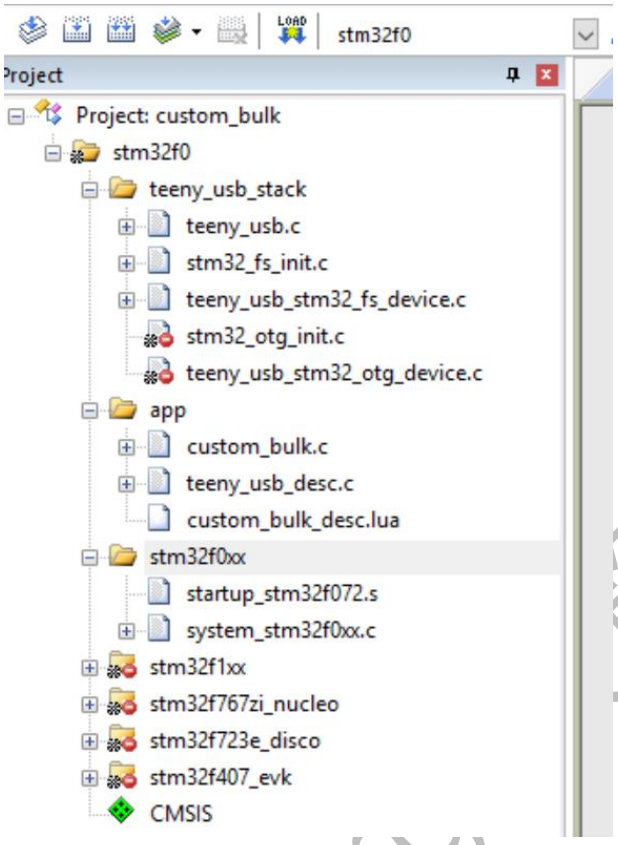


图48　　自定义 USB 设备工程结构

teeny_usb_stack 目录中是协议栈相关的文件，其中 teeny_usb.c 文件中实现了 USB 的标准请求。teeny_usb_stm32_fs_device.c 和 teeny_usb_stm32_otg_device.c 分别是 USB FS 模块和 OTG 模块设备模式的内核相关代码。stm32_fs_init.c 和 stm32_otg_init.c 是 TeenyUSB 协议栈初始化相关代码，在这里实现了 USB 模块时钟配置，IO 初始化，中断初始化，以及中断回调处理。

app 目录中是应用相关的代码，custom_bulk.c 是本例程的主程序代码，teeny_usb_desc.c 是 TeenyDT 自动生成的描述符文件。custom_bulk_desc.lua 是 TeenyDT 格式的描述符文件，用来生成 C 语言格式的描述符。

stm32fxxx 目录下是开发板相关的代码， startup_xxx.s 是芯片相关的启动文件。system_stm32fxxx.c 是开发板相关的全局时钟配置代码，在这里根据 HSE 的频率配置芯片的时钟。

下图是工程相关的宏定义：

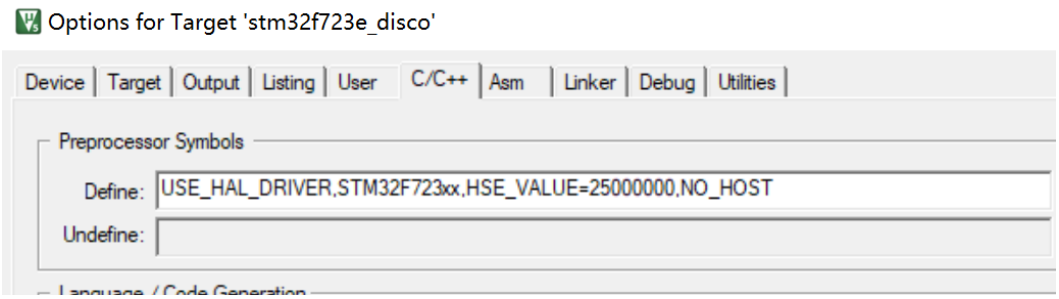


图49　　示例工程宏定义

USE_HAL_DRIVER 和 STM32F723xx 是 HAL 库需要的宏定义，HSE_VALUE=25000000 宏配置

芯片的外置晶振频率，HAL 库根据 HSE_VALUE 完成时钟初始化。NO_HOST 是 TeenyUSB 协议栈的宏，表示不需要主机相关代码。NO_DEVICE 表示不需要设备相关代码。如果没有定义 NO_HOST 宏，需要将主机相关的代码包含到工程中，否则会报错。同样，如果没有定义 NO_DEVICE 宏，需要将设备相关的代码包含到工程中，否则会报错。这个示例只使用了设备相关的功能，因此使用 NO_HOST 去掉主机相关代码，减少最终代码的体积。

6.2.1 确定设备功能

自定义设备只有一个接口，这个接口上有一个 BULK IN 端点和一个 BULK OUT 端点分别用来做数据输入和输出。BULK IN 使用端点 1，BULK OUT 使用端点 2，IN 和 OUT 分为两个端点是为了可以在 USB FS 模块上使用双缓存功能。

6.2.2 描述符设计

根据前面的功能描述，这个设备有一个设备描述符，一个配置描述数，配置描述符中有一个接口描述符，接口描述符中有两个端点描述符。

6.2.2.1 TeenyDT 命令行模式

TeenyDT 格式的描述符如下：

```
return Device {
    strManufacture = "TeenyUSB",
    strProduct = "TeenyUSB Custom Bulk ",
    strSerial = "TUSB123456",
    idVendor = 0x0483,
    idProduct = 0x0001,
    prefix = "BULK",
    Config {
        Interface{
            EndPoint(IN(1),  BulkDouble, 64),
            EndPoint(OUT(2), BulkDouble, 64),
   }
}
```

根据 3.10 描述符工具 TeenyDT 这一节中的内容，设备默认为总线供电 200mA 电流，不支持远程唤醒。上面的内容在代码的 usb_stack\demo\custom_bulk\custom_bulk_desc.lua 中。将 lua5.3 可执行程序所在目录配置到 Windwos 系统的路径（PATH）环境变量中，在 TeenyDT 目录中启动命令行，执行下面的命令：

```
lua gen_decriptor.lua ..\demo\custom_bulk\custom_bulk_desc.lua
```

在 demo\custom_bulk 目录中会生成 teeny_usb_desc.c 和 teeny_usb_init.h 文件。其中 teeny_usb_desc.c 是描述符定义文件，teeny_usb_init.h 是端点初始化定义文件。

为了方便使用 TeenyDT 命令行模式生成描述符文件，在 custom_bulk 目录下有一个名为 gen_desc.bat 批处理文件，在命令行中执行 gen_desc.bat 会调用 lua 执行生成描述符的

命令。如果 gen_desc.bat 后面不带参数，会默认为 custom_bulk_desc.lua 生成描述符。如果 gen_desc.bat 后面有参数，会以第一个参数为输入文件生成描述符。可以在 Keil 的 build 事件中使用 gen_desc.bat，在编译时根据输入文件自动生成描述符和初始化文件。参数中还能为不同的芯片设置不同的最大端点数和 USB 专用缓存大小。设置如下图所示：

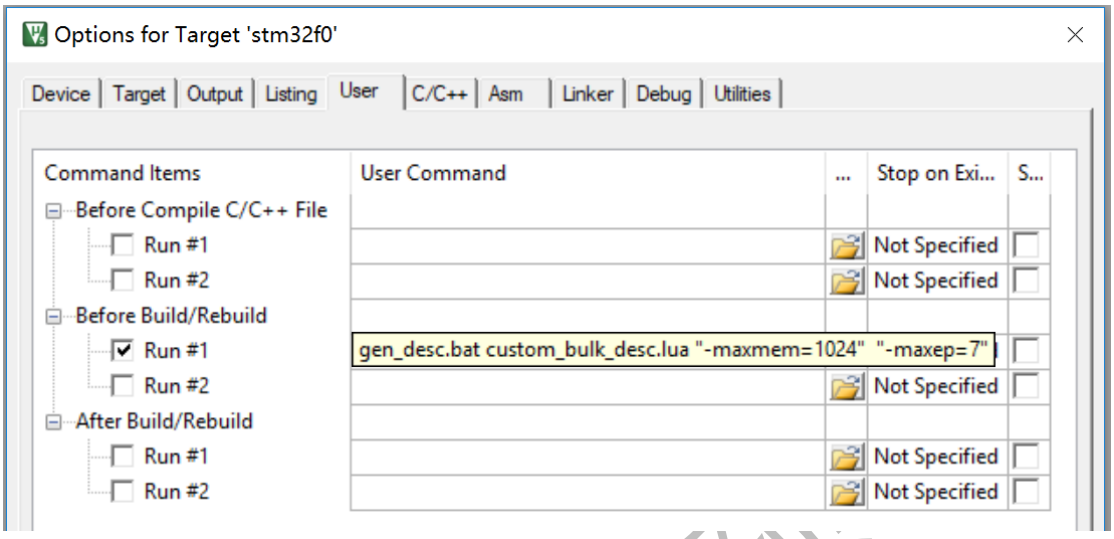


图50　　Keil 中的 Build 事件设置

6.2.2.2 TeenyDT 图形界面模式

图形界面下描述符配置如下图所示：

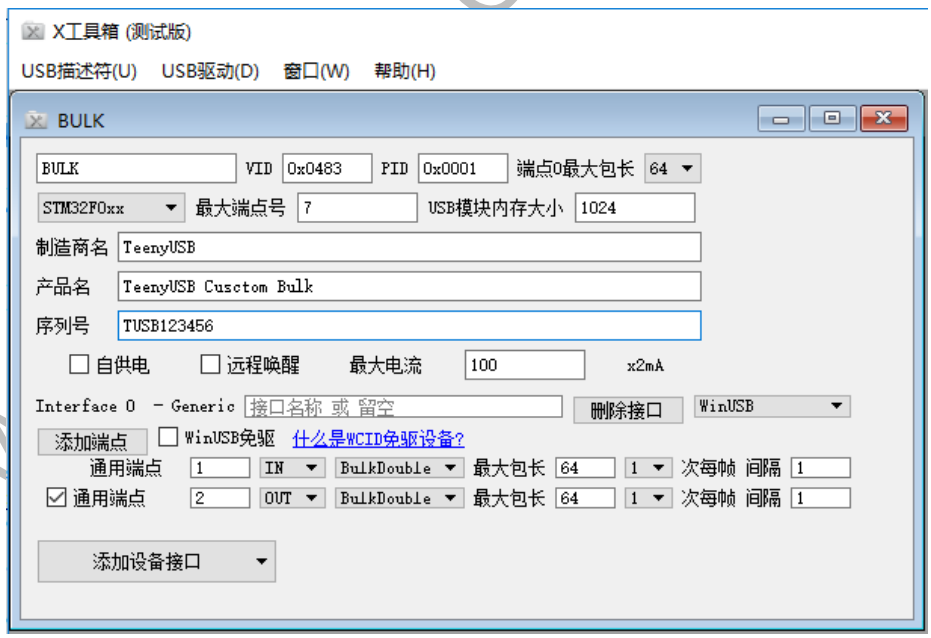


图51　　图形界面生成 Bulk 描述符

在菜单【USB 描述符】->【生成代码】中选择生成代码的路径，生成代码。或者使用快捷键 Ctrl+G 生成代码。图形界面模式生成的代码与命令行方式生成的代码相同。

6.2.3 初始化设备端点

设备的端点 0 初始化为控制端点，最大包长为 64 字节。端点 1 初始化为双缓冲 Bulk IN，端点 2 初始化为双缓冲 Bulk OUT。如果是 OTG 设备，没有双缓冲，初始化为普通的 Bulk 端点。在描述符设计时自动生成的代码中包含了端点初始化的代码，在 teeny_usb_init.h 文件中。初始化代码如下：

```
#define BULK_TUSB_INIT_EP_FS(dev) \
  do{\
    /* Init ep0 */ \
    INIT_EP_BiDirection(dev, PCD_ENDP0, BULK_EP0_TYPE);  \
    SET_TX_ADDR(dev, PCD_ENDP0, BULK_EP0_TX_ADDR);  \
    SET_RX_ADDR(dev, PCD_ENDP0, BULK_EP0_RX_ADDR);  \
    SET_RX_CNT(dev, PCD_ENDP0, BULK_EP0_RX_SIZE);  \
    /* Init ep1 */ \
    INIT_EP_TxDouble(dev, PCD_ENDP1, BULK_EP1_TYPE);     \
    SET_DOUBLE_ADDR(dev, PCD_ENDP1, BULK_EP1_TX0_ADDR, BULK_EP1_TX1_ADDR);  \
    SET_DBL_TX_CNT(dev, PCD_ENDP1, 0);     \
    /* Init ep2 */ \
    INIT_EP_RxDouble(dev, PCD_ENDP2, BULK_EP2_TYPE);     \
    SET_DOUBLE_ADDR(dev, PCD_ENDP2, BULK_EP2_RX0_ADDR, BULK_EP2_RX1_ADDR);     \
    SET_DBL_RX_CNT(dev, PCD_ENDP2, BULK_EP2_RX_SIZE);     \
  }while(0)
// EndPoints init function for USB OTG core
#define BULK_TUSB_INIT_EP_OTG(dev) \
  do{\
    SET_RX_FIFO(dev, BULK_OTG_RX_FIFO_ADDR, BULK_OTG_RX_FIFO_SIZE);  \
    /* Init ep0 */ \
    INIT_EP_Tx(dev, PCD_ENDP0, BULK_EP0_TYPE, BULK_EP0_TX_SIZE);  \
    SET_TX_FIFO(dev, PCD_ENDP0, BULK_EP0_TX_FIFO_ADDR, BULK_EP0_TX_FIFO_SIZE);  \
    INIT_EP_Rx(dev, PCD_ENDP0, BULK_EP0_TYPE, BULK_EP0_RX_SIZE);  \
    /* Init ep1 */ \
    INIT_EP_Tx(dev, PCD_ENDP1, BULK_EP1_TYPE, BULK_EP1_TX_SIZE);  \
    SET_TX_FIFO(dev, PCD_ENDP1, BULK_EP1_TX_FIFO_ADDR, BULK_EP1_TX_FIFO_SIZE);  \
    /* Init ep2 */ \
    INIT_EP_Rx(dev, PCD_ENDP2, BULK_EP2_TYPE, BULK_EP2_RX_SIZE);  \
  }while(0)
```

根据 USB 模块类型的不同，选择不同的端点初始化宏，并生成最终的设备初始化宏，代码如下：

```
#if defined(USB)
#define BULK_TUSB_INIT_EP(dev) BULK_TUSB_INIT_EP_FS(dev)
// Teeny USB device init function for FS core
#define BULK_TUSB_INIT_DEVICE(dev) \
  do{\
```

```
    /* Init device features */       \
    memset(&dev->addr, 0, TUSB_DEVICE_SIZE);    \
    dev->status = BULK_DEV_STATUS;         \
    dev->rx_max_size = BULK_rxEpMaxSize;         \
    dev->tx_max_size = BULK_txEpMaxSize;         \
    dev->descriptors = &BULK_descriptors;         \
  }while(0)
#endif
#if defined(USB_OTG_FS) || defined(USB_OTG_HS)
#define BULK_TUSB_INIT_EP(dev) BULK_TUSB_INIT_EP_OTG(dev)
// Teeny USB device init function for OTG core
#define BULK_TUSB_INIT_DEVICE(dev) \
  do{\
    /* Init device features */       \
    memset(&dev->addr, 0, TUSB_DEVICE_SIZE);    \
    dev->status = BULK_DEV_STATUS;         \
    dev->descriptors = &BULK_descriptors;         \
  }while(0)
#endif
#define BULK_TUSB_INIT(dev) \
  do{\
    BULK_TUSB_INIT_EP(dev);   \
    BULK_TUSB_INIT_DEVICE(dev);   \
  }while(0)
```

在 tusb_reconfig 回调函数中调用 BULK_TUSB_INIT 宏对端点和设备上下文结构体 dev 进行初始化。

6.2.4 设备类型相关请求处理

自定义设备没有设备类型相关的请求。

6.2.5 设备功能实现

自定义设备只有两个端点，端点 1 类型为 IN，使用端点 1 发送数据。端点 2 类型为 OUT，使用端点 2 接收数据。这个自定义设备的功能是将收到的所有数据逐字节加 1 后返回给主机，做一个回环测试。代码处理流程见下图：

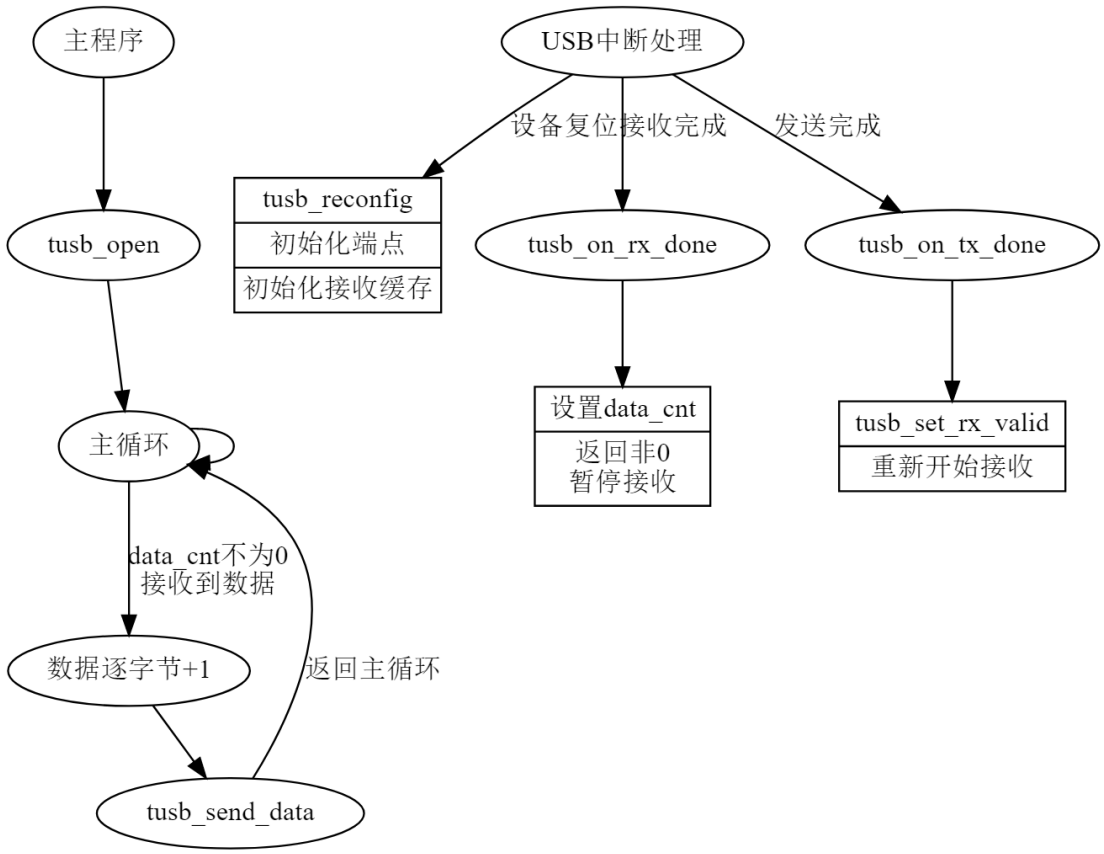


图52　　自定义设备处理流程

自定义设备使用固定缓存接收数据，接收数据成功后在 tusb_on_rx_done 回调中设置 data_cnt 并返回-1，通知 TeenyUSB 协议栈缓存数据未处理完毕，暂停后续数据的接收。

在主循环中检测到 data_cnt 不为 0 时，将缓存中的数据逐字节+1 后发送给主机。发送完成后调用 tusb_set_rx_valid 重新开始接收数据。

自定义设备主程序完整代码如下：

```
#define  TX_EP   PCD_ENDP1
#define  RX_EP   PCD_ENDP2
uint8_t buf[4096];
__IO uint32_t data_cnt = 0;
// if data tx done, set rx valid again
void tusb_on_tx_done(tusb_device_t* dev, uint8_t EPn){
  if(EPn == TX_EP){
    tusb_set_rx_valid(dev, RX_EP);
  }
}
int tusb_on_rx_done(tusb_device_t* dev, uint8_t EPn, const void* data, uint16_t len){
  if(EPn == RX_EP){
    data_cnt = len;
    return len;
  }
  return 0;
}
```

```
void tusb_reconfig(tusb_device_t* dev){
  // call the BULK device init macro
  BULK_TUSB_INIT(dev);
  // setup recv buffer for rx end point
  tusb_set_recv_buffer(dev, RX_EP, buf, sizeof(buf));
  // enable rx ep after buffer set
  tusb_set_rx_valid(dev, RX_EP);
}
void delay_ms(uint32_t ms){
  uint32_t i,j;
  for(i=0;i<ms;++i)
    for(j=0;j<20;++j);
}
int main(void){
#if defined(STM32F723xx)
  tusb_device_t* dev = tusb_get_device(1);
#else
  tusb_device_t* dev = tusb_get_device(0);
#endif
  tusb_close_device(dev);
  delay_ms(100);
  tusb_open_device(dev);
  while(1){
    if(data_cnt){
      // every data plus 1 and echo back
      for(int i=0;i<data_cnt;i++){
        buf[i]++;
      }
      tusb_send_data(dev, TX_EP, buf, data_cnt);
      data_cnt = 0;
    }
  }
}
```

自定义设备的完整工程在demo\custom_bulk目录中，根据实际芯片型号选择工程配置，编译后将代码下载到开发板中。连接电脑和开发板的 USB 接口，在设备管理器中看到下图的内容：

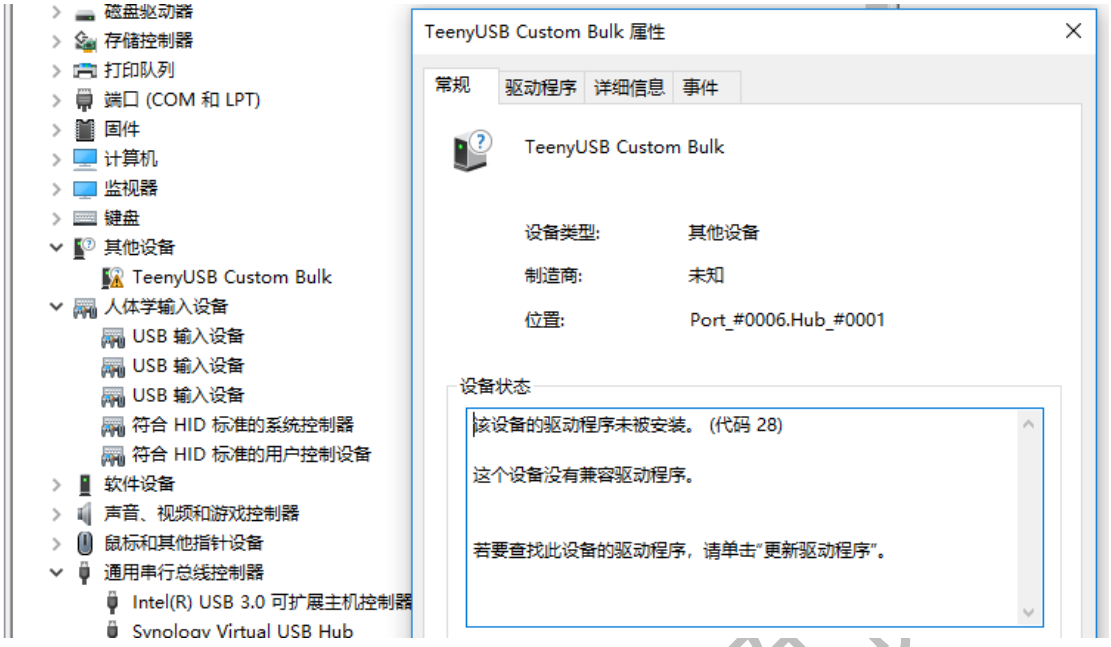


图53　　Custom Bulk 未安装驱动

在任务管理器中看到一个名为“TeenyUSB Custom Bulk”设备，说明设备枚举成功，有一个问号是因为设备未安装驱动，下一节将介绍如何安装设备驱动。

6.2.6 设备驱动

自定义设备没有驱动，在设备管理器中会显示为“该设备的驱动未安装”。本节开头提到过省掉驱动编写的过程，所以这里不会介绍怎么编写驱动，而是用通用 USB 设备驱动来操作这个自定设备。

社区和 Windows 官方针对 USB 设备，提供了通用的驱动程序。通用驱动程序提供最基本操作功能，例如发起控制传输，读写端点等。有了这些基本操作，PC 上的应用程序就可以和 USB 设备进行通讯了。Windows 提供的通用驱动名为 WinUSB，在 Win7 之后，Windows 系统中都自带了这个驱动程序，所以用 WinUSB 在 Windows 上是一个比较方便的选择。除了 WinUSB 外，社区还有还有 libusb 通用驱动。本文后续只介绍如何使用 WinUSB，不对 libusb 做介绍。TeenyDT 可以生成 libusb 版本驱动的 inf 文件，驱动安装后操作方式与 WinUSB 基本类似。

为了使用 WinUSB 驱动，还需要编写 inf 文件，在 inf 文件中将设备的 VID、PID、接口号等与 WinUSB 驱动关联起来。一个名为 libwdi 的开源项目为 USB 设备提供自动安装 WinUSB 驱动的功能，TeenyDT 中使用了 libwdi 中的驱动模板来为设备生成 inf 文件。在 Win10 系统上驱动文件必须要有签名才能安装，因此 TeenyDT 对生成的 inf 文件进行了签名。笔者没有 Windows 授权的证书，签名采用的是自签名形式，在安装驱动时会提示是否信任并安装。签名部分的功能也是采用了 libwdi 中的代码。使用 TeenyDT 生成驱动的方式如下图所示：

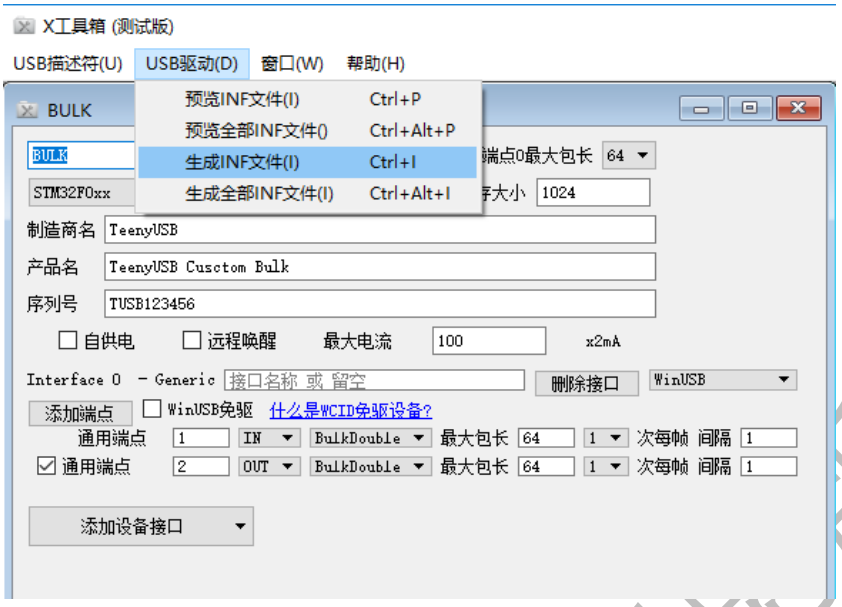


图54　　Custom Bulk 生成驱动

在上图中，为接口选择驱动类型为 WinUSB，然后在菜单中选择【生成 INF 文件】或用快捷键 Ctrl+I。驱动生成后放在 TeenyDT 程序目录下的 out 目录中。在设备管理其中选择“TeenyUSB Custom Bulk”设备安装驱动，如下图所示：

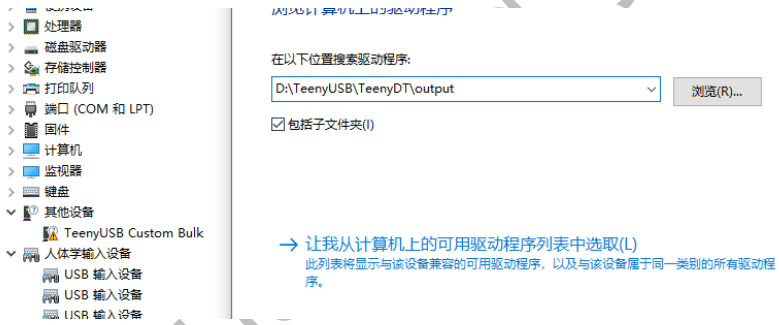


图55　　Custom Bulk 选择驱动

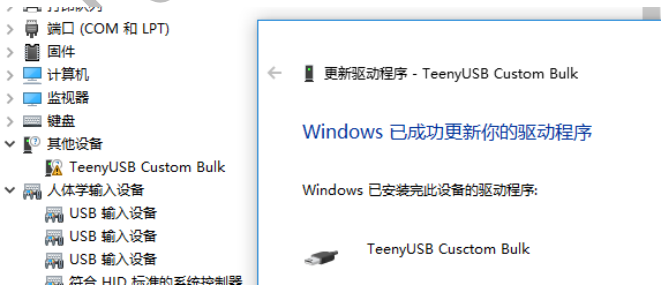


图56　　Custom Bulk 驱动安装成功

也可以在生成的 inf 文件上右键选择【安装】来安装驱动，如下图所示：

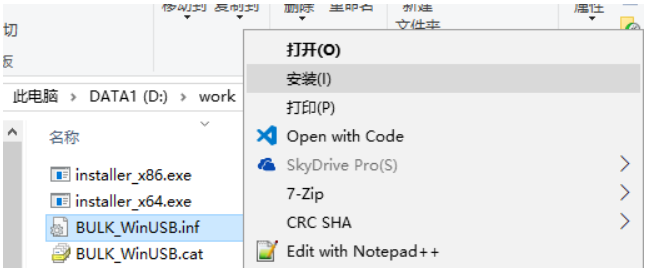


图57　　Custom Bulk 安装 INF 文件

驱动安装后设备管理器中看到自定义的 USB 设备可以正常使用了，在设备管理器中设备显示正常，如下图：

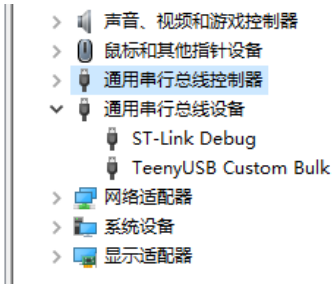


图58　　Custom Bulk 驱动安装完成

6.2.7 功能测试

通用的 USB 设备测试工具在 pc_test_tool 目录中，工具采用 lua Qt 框架编写，关于这个框架的说明在 xtoolbox.org。运行 xtoolbox.exe 后可以看到如下的界面：

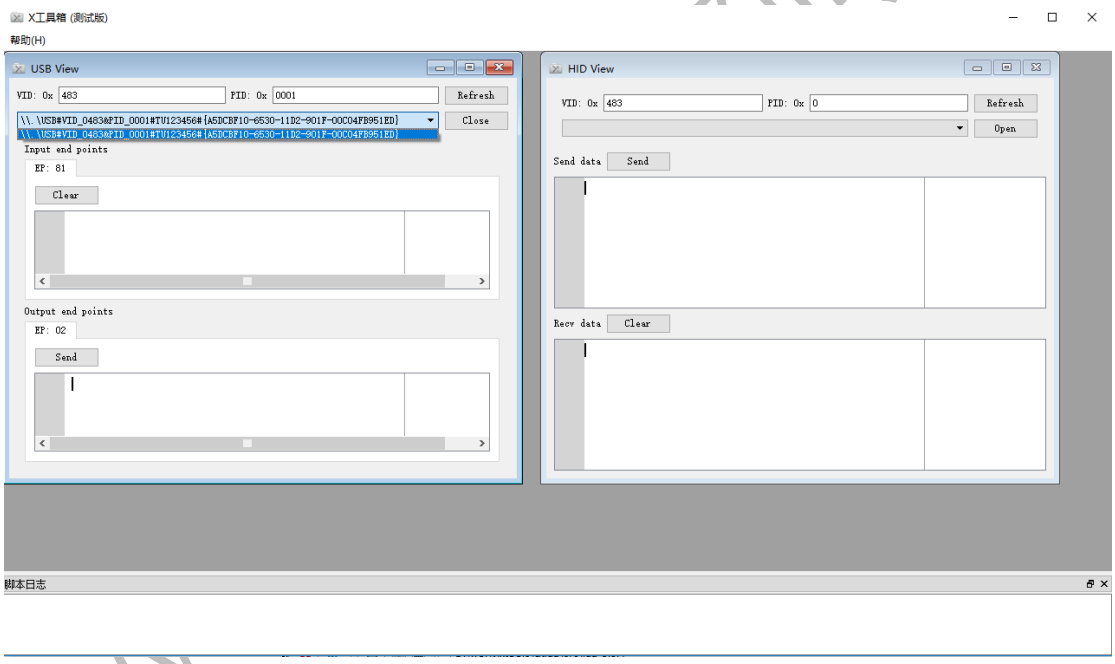


图59　　Custom Bulk USB View

在左侧的 USB View 窗口中，输入自定义设备的 VID 0x0483 和 PID 0x0001，点击 Refresh 后可以看到设备路径，如果有多个相同 VID，PID 的设备，路径中会有多个设备路径。选择路径后点击 Open 按钮打开设备。设备打开后会看到设备中所有的 IN 端点和 OUT 端点。

自定义设备会将 OUT 端点收到的数据全部+1 后通过 IN 端点发回，在 OUT 端点中随便写入一些数据，然后点击 Send。可以看到在 IN 端点中这些数据被+1 后返回了，并且长度与 OUT 中的相同，如下图所示：

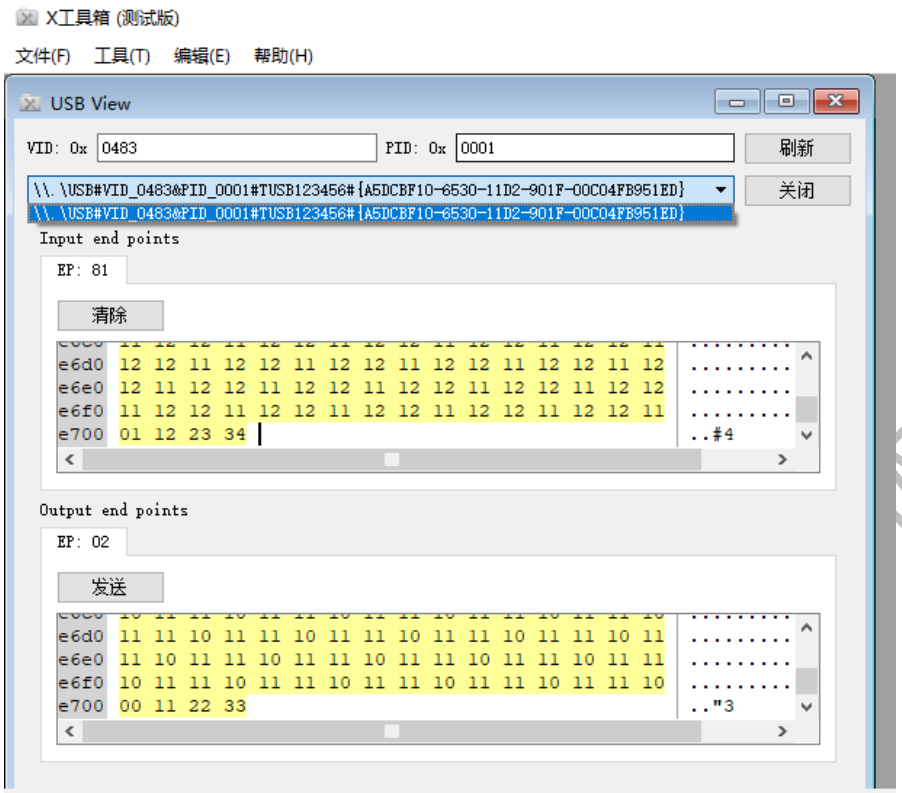


图60　　Custom Bulk 测试

至此，一个完整的自定义设备就完成了。不过本节开头说过自定义设备很简单，连 INF 文件都不用写，前面虽然没有写驱动，但是还得带个 INF 文件，还得签名。既然挖了一个不要驱动的坑，下面就来填这个坑，介绍如何做一个连 INF 文件都不用的自定义设备。

6.2.8 WCID 设备

在 Windows 上可以定义一种名为 WCID 的设备，这类设备的驱动不是通过 VID、PID 来进行匹配的，而是通过兼容 ID（Windows Compatible ID）来匹配。系统中预装的 WinUSB 支持 WCID 设备，他的兼容 ID 是“WINUSB”，只要我们设备的 WCID 能被识别为“WINUSB”就能自动安装上 WinUSB 驱动，不需要编写 INF 文件。

如何让设备被 Windows 系统识别为 WCID 设备呢？在设备枚举阶段，系统会发送一个 ID 为 0xEE 的字符串请求，设备收到这个请求后，返回字符串内容为“MSFT100\x17\x00”。这里的\x17 也可以是其他值，后面在响应厂商请求时会用到，这里以 0x17 为例。然后 Windows 系统会发送一个请求号为 0x17，索引为 4 的厂商请求，设备返回如下的内容：

```
WEAK __ALIGN_BEGIN const uint8_t BULK_WCIDDescriptor [40] __ALIGN_END = {
    0x28, 0x00, 0x00, 0x00,                 /* dwLength */
    0x00, 0x01,                             /* bcdVersion */
    0x04, 0x00,                             /* wIndex */
    0x01,                                   /* bCount */
    0,0,0,0,0,0,0,                          /* Reserved */
    /* WCID Function  */
    0x00,                                   /* bFirstInterfaceNumber */
    0x01,                                   /* bReserved */
```

```
    /* CID */
    'W', 'I', 'N', 'U', 'S', 'B', 0x00, 0x00,
    /* sub CID */
    0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
    0,0,0,0,0,0,                                    /* Reserved */
};
```

这样 Windows 就能将自定义设备识别为 WCID 是“WINUSB”的设备了。

TeenyUSB 代码中已经包含了对 WCID 设备的支持，需要打开 HAS_WCID 宏开关。TeenyDT 也支持生成 WCID 相关的描述符，需要在接口中指定 WCID 的类型，目前 TeenyDT 只支持 WinUSB 类型。启用了 WCID 的 TeenyDT 格式描述符内容如下：

```
return Device {
    strManufacture = "TeenyUSB",
    strProduct = "TeenyUSB Custom Bulk",
    strSerial = "TUSB123456",
    idVendor = 0x0483,
    idProduct = 0x0001,
    prefix = "BULK",
    Config {
        Interface{
            WCID=WinUSB,
            EndPoint(IN(1),  BulkDouble, 64),
            EndPoint(OUT(2), BulkDouble, 64),
        },
   }
}
```

在需要增加 WCID 支持的接口中加入 WCID=WinUSB。

图形化界面中操作如下：

图61　　Custom Bulk WCID

在接口中勾选上【WinUSB 免驱】选项，生成的描述符和初始化文件中会包含 WCID 的

内容。

如果将自定义设备的 VID 或者 PID 修改后，再加入 WCID 支持，可以不做下面的卸载操作。如果还要用原来的 VID 和 PID，需要卸载掉之前的驱动。因为 WCID 的存在，卸载时还需要将设备的 WCID 属性从注册表中删掉，这样系统会重新从设备读取 WCID 属性。

6.2.8.1 卸载驱动

在设备管理器中卸载掉之前的驱动，勾选上【删除此设备的驱动程序软件】，

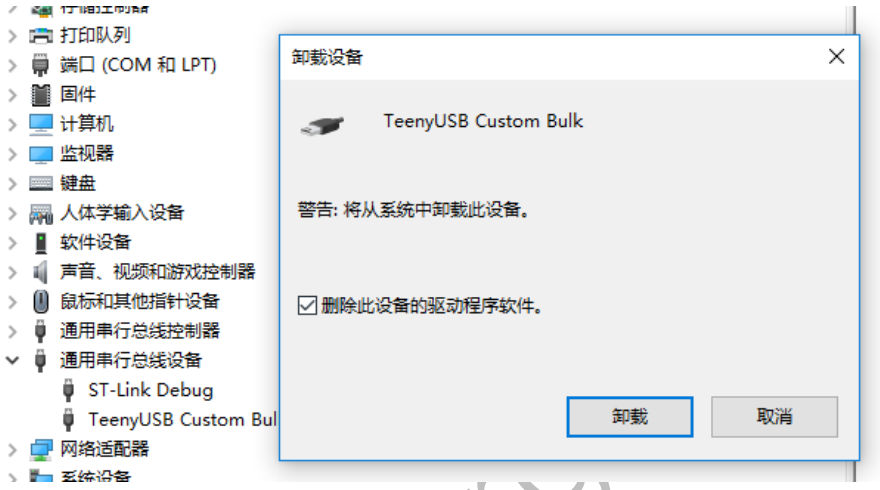


图62　　Custom Bulk 卸载驱动

然后在注册表的[HKEY_LOCAL_MACHINE\SYSTEM\ControlSet001\Control\usbflags]键中，找到 048300010100 这一项，全部删除。048300010100 拆开就是 VID=0483，PID=0001，bcdDevice=0100。如果 VID 和 PID 以及设备版本（bcdDevice）是其他值，则删除其对应的键值。

重新生成描述符和初始化文件，编译新的包含了 WCID 支持的代码，下载到开发板中，然后连接上电脑。可以看到在设备管理器中为设备自动安装了 WinUSB 驱动，可以看到驱动的签名者是 Microsoft Windows，说明驱动安装成功。

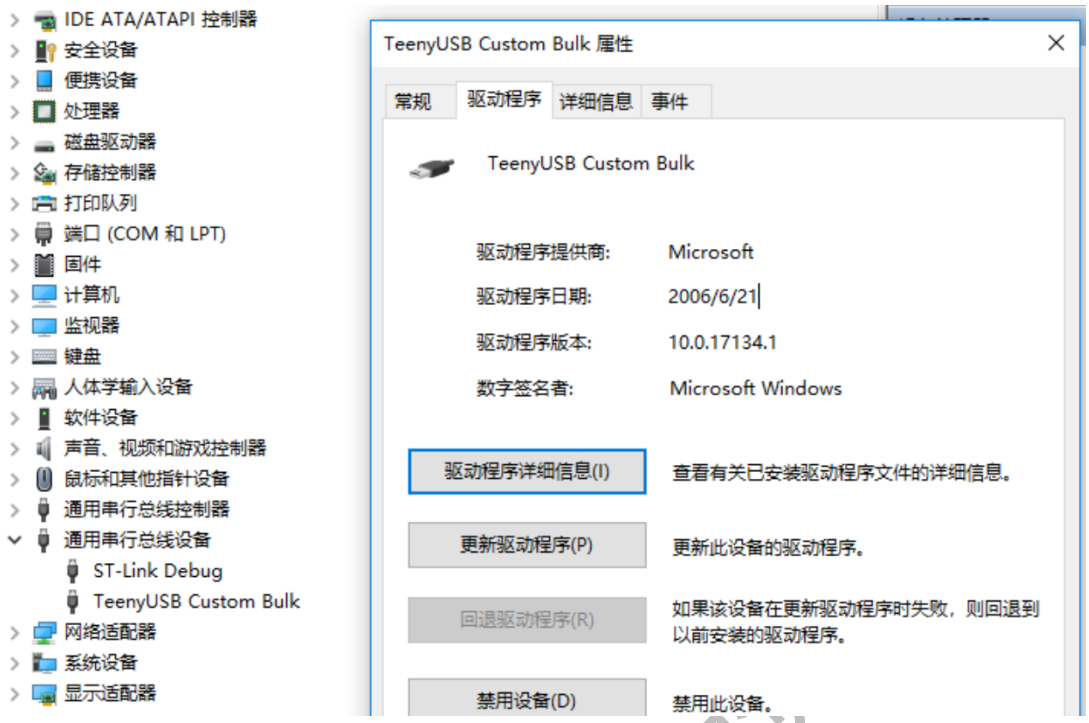


图63　　Custom Bulk WCID WinUSB

WCID 安装的驱动和之前用生成的 INF 安装的驱动是一样的，都是 WinUSB，不同的只是 inf 文件，因此对设备的操作也是一样的。前面的功能测试流程同样适用于 WCID 自动安装的 WinUSB 驱动。

实际上自定义驱动程序也可以具备这样的能力，不匹配设备 VID 和 VID，而是匹配设备的 WCID。在 WinUSB 的 inf 文件中有这下面的字段：

```
[Generic.Section.NTamd64]
%USB\MS_COMP_WINUSB.DeviceDesc%=WINUSB,USB\MS_COMP_WINUSB
```

其中 MS_COMP_前缀表示此驱动可以匹配 WCID，如果要为自定义的驱动程序增加支持 WCID 设备的功能，可以对照着 WinUSB 的 inf 文件，增加相应的内容。

有关 WCID 设备更详细的说明，参阅什么是 WCID 免驱设备。

6.3 CDC 串口设备

USB 转串口是用得比较多的设备，PC 上有大量的通过串口进行通讯的软件，本节介绍如何制作 CDC 串口设备。CDC 串口的完整代码在 usb_stack/demo/cdc 目录中。

6.3.1 确定设备功能

这里只实现一个只有一个串口功能的设备，因为只是做 USB 功能的测试，同前面的自定义设备一样，将接收到的数据+1 后再发送回主机。

6.3.2 描述符设计

CDC 串口采用的是 CDC 设备下面的 ACM（Abstract Control Model）类来实现，关于这个类的文档在这里。CDC 全称是 Communication Device Class，通讯设备类。CDC 串口设备的描述符设计参照已有的例程来实现。相较于前面的自定义设备而言，在 Interface 接口描述中，增加了 CDC 功能描述符相关的内容。需要下面四个功能描述符，这些功能描述符详细内容见下表：

Table 15: Class-Specific Descriptor Header Format

Offset	Field	Size	Value	Description
0	*bFunctionLength*	1	Number	Size of this descriptor in bytes.
1	*bDescriptorType*	1	Constant	CS_INTERFACE descriptor type.
2	*bDescriptorSubtype*	1	Constant	Header functional descriptor subtype as defined in Table 13.
3	*bcdCDC*	2	Number	USB Class Definitions for Communications Devices Specification release number in binary-coded decimal.

表25　头部功能描述符（来自 USB CDC 类规格书）

Table 3: Call Management Functional Descriptor

Offset	Field	Size	Value	Description
0	*bFunctionLength*	1	Number	Size of this functional descriptor, in bytes.
1	*bDescriptorType*	1	Constant	CS_INTERFACE
2	*bDescriptorSubtype*	1	Constant	Call Management functional descriptor subtype, as defined in [USBCDC1.2].
3	*bmCapabilities*	1	Bitmap	The capabilities that this configuration supports: D7..D2: RESERVED (Reset to zero) D1: 0 - Device sends/receives call management information only over the Communications Class interface. 1 - Device can send/receive call management information over a Data Class interface. D0: 0 - Device does not handle call management itself. 1 - Device handles call management itself. The previous bits, in combination, identify which call management scenario is used. If bit D0 is reset to 0, then the value of bit D1 is ignored. In this case, bit D1 is reset to zero for future compatibility.
4	*bDataInterface*	1	Number	Interface number of Data Class interface optionally used for call management. *

表26　　呼叫管理功能描述符（来自 USB PSTN 类规格书）

Table 4: Abstract Control Management Functional Descriptor

Offset	Field	Size	Value	Description
0	*bFunctionLength*	1	Number	Size of this functional descriptor, in bytes.
1	*bDescriptorType*	1	Constant	CS_INTERFACE
2	*bDescriptorSubtype*	1	Constant	Abstract Control Management functional descriptor subtype as defined in [USBCDC1.2].
3	*bmCapabilities*	1	Bitmap	The capabilities that this configuration supports. (A bit value of zero means that the request is not supported.) D7..D4: RESERVED (Reset to zero) D3: 1 - Device supports the notification Network_Connection. D2: 1 - Device supports the request Send_Break D1: 1 - Device supports the request combination of Set_Line_Coding, Set_Control_Line_State, Get_Line_Coding, and the notification Serial_State. D0: 1 - Device supports the request combination of Set_Comm_Feature, Clear_Comm_Feature, and Get_Comm_Feature. The previous bits, in combination, identify which requests/notifications are supported by a CommunicationsClass interface with the SubClass code of Abstract Control Model.

表27　　ACM 管理功能描述符（来自 USB PSTN 类规格书）

Table 16: Union Interface Functional Descriptor

Offset	Field	Size	Value	Description
0	*bFunctionLength*	1	Number	Size of this functional descriptor, in bytes.
1	*bDescriptorType*	1	Constant	CS_INTERFACE
2	*bDescriptorSubtype*	1	Constant	Union Functional Descriptor SubType as defined in Table 13.
3	*bControlInterface*	1	Constant	The interface number of the Communications or Data Class interface, designated as the controlling interface for the union.*
4	*bSubordinateInterface0*	1	Number	Interface number of first subordinate interface in the union. *
…	…	…	…	
N+3	*bSubordinateInterfaceN-1*	1	Number	Interface number of N-1 subordinate interface in the union. *

表28　　联合接口功能描述符（来自 USB CDC 类规格书）

CDC 中部分 bDescriptorSubtype 的值定义见下表：

Table 13: bDescriptor SubType in Communications Class Functional Descriptors

Descriptor subtype	Functional description
00h	Header Functional Descriptor, which marks the beginning of the concatenated set of functional descriptors for the interface.
01h	Call Management Functional Descriptor.
02h	Abstract Control Management Functional Descriptor.
03h	Direct Line Management Functional Descriptor.
04h	Telephone Ringer Functional Descriptor.
05h	Telephone Call and Line State Reporting Capabilities Functional Descriptor.
06h	Union Functional Descriptor

表29　　CDC 功能描述符定义部分（来自 USB CDC 类规格书）

CDC 串口设备需要两个接口来实现功能，一个是通讯接口，用来描述设备功能，一个是数据接口，用来传输数据。TeenyDT 中串口相关的接口描述符实现如下：

```
function CDC_ACM(param)
    Ctrl = Interface{
        bInterfaceClass = 2,
        bInterfaceSubClass = 2,
        bInterfaceProtocol = 1,
        Function{   bDescriptorSubtype = 0,
                    alias = "Header",
                    varData = {
                        {bcdCDC = 0x110},
                    }
                },
        Function{   bDescriptorSubtype = 1,
                    alias = "Call Management",
```

```
                    varData = {
                        {bmCapabilities = 0x00},
                        {bDataInterface = 0x01},
                    }
                },
        Function{   bDescriptorSubtype = 2,
                    alias = "ACM (Abstract Control Management)",
                    varData = {
                        {bmCapabilities = 0x02},
                    }
                },
        Function{   bDescriptorSubtype = 6,
                    alias = "Union",
                    varData = {
                        {bMasterInterface = 0x00},
                        {bSlaveInterface0 = 0x01},
                    }
                },
        EndPoint(IN(3),  Interrupt,  64),
    }
    Data = Interface{
        bInterfaceClass = 0x0a,
        bInterfaceSubClass = 0,
        bInterfaceProtocol = 0,
        EndPoint(IN(1),  BulkDouble,  64),
        EndPoint(OUT(2),  BulkDouble,  64),
    }
    return IAD{Ctrl, Data}
end
```

上面的代码分别定义了两个接口，一个 Ctrl 和一个 Data。Ctrl 是通讯控制接口，里面包含了前文中提到的 4 个功能描述符和 1 个端点描述符。Ctrl 接口的类型为（2，2，1），接口类型值在 USB CDC 规格书中有详细定义。

Data 是数据接口，包含两个端点。Ctrl 和 Data 接口通过 IAD 描述符联合在一起。TeenyDT 为了简化串口描述符的编写，将上面的内容封装成了串口设备的描述符定义接口 CDC_ACM。

完整的串口设备描述符如下：

```
return Device {
    strManufacture = "TeenyUSB",
    strProduct = "TeenyUSB CDC",
    strSerial = "TUSB123456",
    idVendor = 0x0483,
    idProduct = 0x0002,
    prefix = "CDC",
    Config {
```

```
        CDC_ACM{
            EndPoint(IN(3),  Interrupt,  64),
            EndPoint(IN(1),  BulkDouble, 64),
            EndPoint(OUT(2), BulkDouble, 64),
        },
    }
}
```

TeenyDT 格式描述符的详细介绍在 3.10 描述符工具 TeenyDT 这一节中。

6.3.3 初始化设备端点

下面的设备端点初始化代码由 TeenyDT 自动生成：

```
// EndPoints init function for USB FS core
#define CDC_TUSB_INIT_EP_FS(dev) \
  do{\
    /* Init ep0 */ \
    INIT_EP_BiDirection(dev, PCD_ENDP0, CDC_EP0_TYPE);  \
    SET_TX_ADDR(dev, PCD_ENDP0, CDC_EP0_TX_ADDR);  \
    SET_RX_ADDR(dev, PCD_ENDP0, CDC_EP0_RX_ADDR);  \
    SET_RX_CNT(dev, PCD_ENDP0, CDC_EP0_RX_SIZE);  \
    /* Init ep1 */ \
    INIT_EP_TxDouble(dev, PCD_ENDP1, CDC_EP1_TYPE);     \
    SET_DOUBLE_ADDR(dev, PCD_ENDP1, CDC_EP1_TX0_ADDR, CDC_EP1_TX1_ADDR);  \
    SET_DBL_TX_CNT(dev, PCD_ENDP1, 0);     \
    /* Init ep2 */ \
    INIT_EP_RxDouble(dev, PCD_ENDP2, CDC_EP2_TYPE);     \
    SET_DOUBLE_ADDR(dev, PCD_ENDP2, CDC_EP2_RX0_ADDR, CDC_EP2_RX1_ADDR);     \
    SET_DBL_RX_CNT(dev, PCD_ENDP2, CDC_EP2_RX_SIZE);     \
    /* Init ep3 */ \
    INIT_EP_TxOnly(dev, PCD_ENDP3, CDC_EP3_TYPE);       \
    SET_TX_ADDR(dev, PCD_ENDP3, CDC_EP3_TX_ADDR);  \
  }while(0)


// EndPoints init function for USB OTG core
#define CDC_TUSB_INIT_EP_OTG(dev) \
  do{\
    SET_RX_FIFO(dev, CDC_OTG_RX_FIFO_ADDR, CDC_OTG_RX_FIFO_SIZE);  \
    /* Init ep0 */ \
    INIT_EP_Tx(dev, PCD_ENDP0, CDC_EP0_TYPE, CDC_EP0_TX_SIZE);  \
    SET_TX_FIFO(dev, PCD_ENDP0, CDC_EP0_TX_FIFO_ADDR, CDC_EP0_TX_FIFO_SIZE);  \
    INIT_EP_Rx(dev, PCD_ENDP0, CDC_EP0_TYPE, CDC_EP0_RX_SIZE);  \
    /* Init ep1 */ \
    INIT_EP_Tx(dev, PCD_ENDP1, CDC_EP1_TYPE, CDC_EP1_TX_SIZE);  \
```

```
    SET_TX_FIFO(dev, PCD_ENDP1, CDC_EP1_TX_FIFO_ADDR, CDC_EP1_TX_FIFO_SIZE);  \
    /* Init ep2 */ \
    INIT_EP_Rx(dev, PCD_ENDP2, CDC_EP2_TYPE, CDC_EP2_RX_SIZE);  \
    /* Init ep3 */ \
    INIT_EP_Tx(dev, PCD_ENDP3, CDC_EP3_TYPE, CDC_EP3_TX_SIZE);  \
    SET_TX_FIFO(dev, PCD_ENDP3, CDC_EP3_TX_FIFO_ADDR, CDC_EP3_TX_FIFO_SIZE);  \
  }while(0)
```

与前面的自定义设备相比，多了一个端点 3 的初始化。

6.3.4 设备类型相关请求处理

CDC 需要处理串口设置相关的请求。在 ACM 功能描述符中，bmCapabilities 的值为 2，根据 ACM 功能描述符的定义，表示设备只支持 Set Linecoding 和 Get Linecoding 的请求。CDC 设备相关的 USB 请求处理内容如下：

```
static uint16_t VCP_Ctrl (uint32_t Cmd, uint8_t* Buf, uint32_t Len){
  switch (Cmd){
  case SEND_ENCAPSULATED_COMMAND:
    /* Not  needed for this driver */
    break;
  case GET_ENCAPSULATED_RESPONSE:
    /* Not  needed for this driver */
    break;
  case SET_COMM_FEATURE:
    /* Not  needed for this driver */
    break;
  case GET_COMM_FEATURE:
    /* Not  needed for this driver */
    break;
  case CLEAR_COMM_FEATURE:
    /* Not  needed for this driver */
    break;
  case SET_LINE_CODING:
    linecoding.bitrate = (uint32_t)(Buf[0] | (Buf[1] << 8) | (Buf[2] << 16) | (Buf[3] << 24));
    linecoding.format = Buf[4];
    linecoding.paritytype = Buf[5];
    linecoding.datatype = Buf[6];
    break;
  case GET_LINE_CODING:
    Buf[0] = (uint8_t)(linecoding.bitrate);
    Buf[1] = (uint8_t)(linecoding.bitrate >> 8);
    Buf[2] = (uint8_t)(linecoding.bitrate >> 16);
    Buf[3] = (uint8_t)(linecoding.bitrate >> 24);
    Buf[4] = linecoding.format;
    Buf[5] = linecoding.paritytype;
```

```
    Buf[6] = linecoding.datatype;
    break;

  case SET_CONTROL_LINE_STATE:
    /* Not  needed for this driver */
    break;
  case SEND_BREAK:
    /* Not  needed for this driver */
    break;
  default:
    break;
  }
  return 0;
}
__ALIGN_BEGIN static uint8_t cdc_cmd[8] __ALIGN_END;
void CDC_DataoutRequest(tusb_device_t* dev)
{
  uint32_t len = dev->Ep[0].rx_buf - cdc_cmd;
  VCP_Ctrl(dev->setup.bRequest, cdc_cmd, len);
  tusb_send_status(dev);
}

int tusb_class_request(tusb_device_t* dev, tusb_setup_packet* setup_req){
  if( (setup_req->bRequest & USB_REQ_TYPE_MASK) == USB_REQ_TYPE_CLASS){
    if(setup_req->wLength > 0){
      if (setup_req->bmRequestType & 0x80){
        VCP_Ctrl(setup_req->bRequest, cdc_cmd, setup_req->wLength);
        tusb_control_send(dev, (uint16_t*)cdc_cmd, setup_req->wLength);
        return 1;
      }else{
        tusb_set_recv_buffer(dev, 0, cdc_cmd, setup_req->wLength);
        dev->ep0_rx_done = CDC_DataoutRequest;
        return 1;
      }
    }else{
      VCP_Ctrl(setup_req->bRequest, NULL, 0);
      tusb_send_status(dev);
      return 1;
    }
  }
  return 0;
}
```

上面代码中的 VCP_Ctrl 函数来自于 ST 的 HAL 库，VCP_Ctrl 只处理了 Get/Set Linecoding 请求，这是因为在描述符中只定义了这个功能。tusb_class_request 是 TeenyUSB 的设备类型

相关请求处理的回调函数，当请求方向是读数据时，调用 VCP_Ctrl 获取当前的 Linecoding，发送给主机。当请求方向时写数据时，将 CDC_DataoutRequest 函数注册在 ep0_rx_done 上，在写请求的数据传输完成时调用。在 CDC_DataoutRequest 再调用 VCP_Ctrl 设置设备的 Linecoding。

6.3.5 设备功能实现

CDC 串口设备主程序的处理流程与自定义设备是一样的，当接收到数据后逐字节+1 后发送回主机。

6.3.6 设备驱动

当设备插入电脑后，设备管理器会显示出设备的串口号，但是因为没有驱动而无法使用，如下图所示：

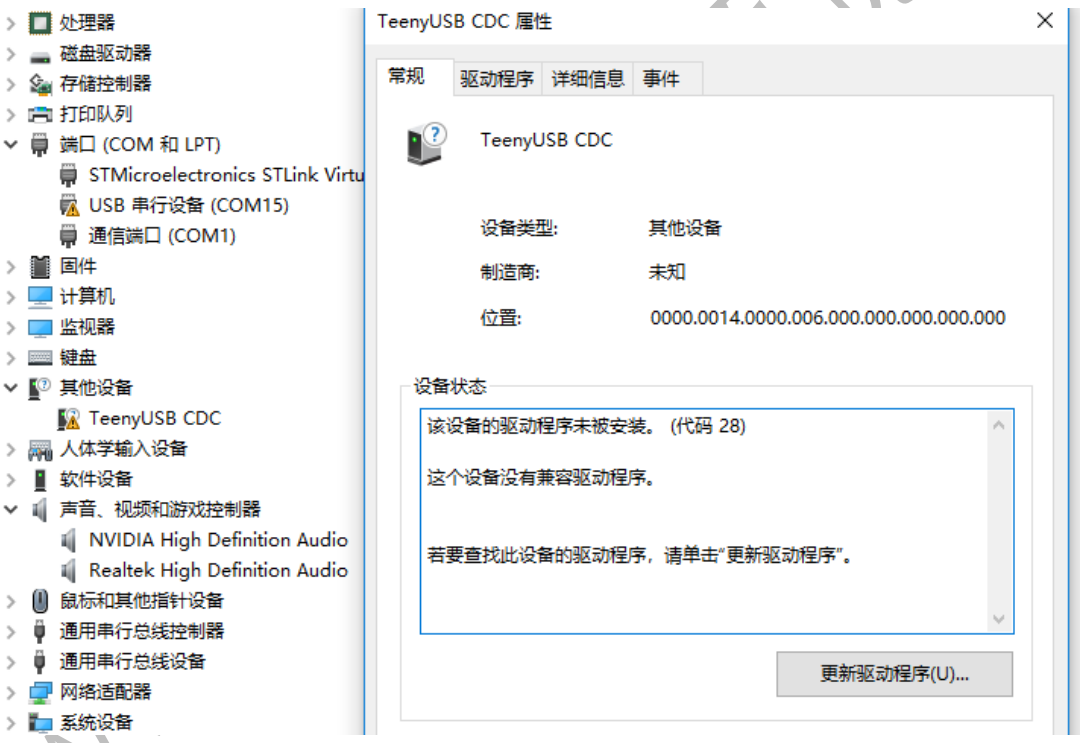


图64　　CDC 设备无驱动

通过 TeenyDT 的图形化界面方式生成串口驱动，如下图所示：

图65　　CDC 驱动生成

CDC 驱动与自定义设备的驱动类似，只需要一个 INF 文件，系统提供了串口设备的通用驱动 usbser.sys。这里需要注意的时，当 CDC 设备接上电脑时，设备管理器会出现两个不能使用的设备，这是因为串口设备有两个接口。由于需要对两个接口一起去安装驱动，可以在生成的 INF 文件上点击右键安装驱动。在设备管理器中，一次只能选择一个接口，因此在设备管理器中安装驱动时会发生错误。 驱动程序安装成功后，设备管理器如下图：

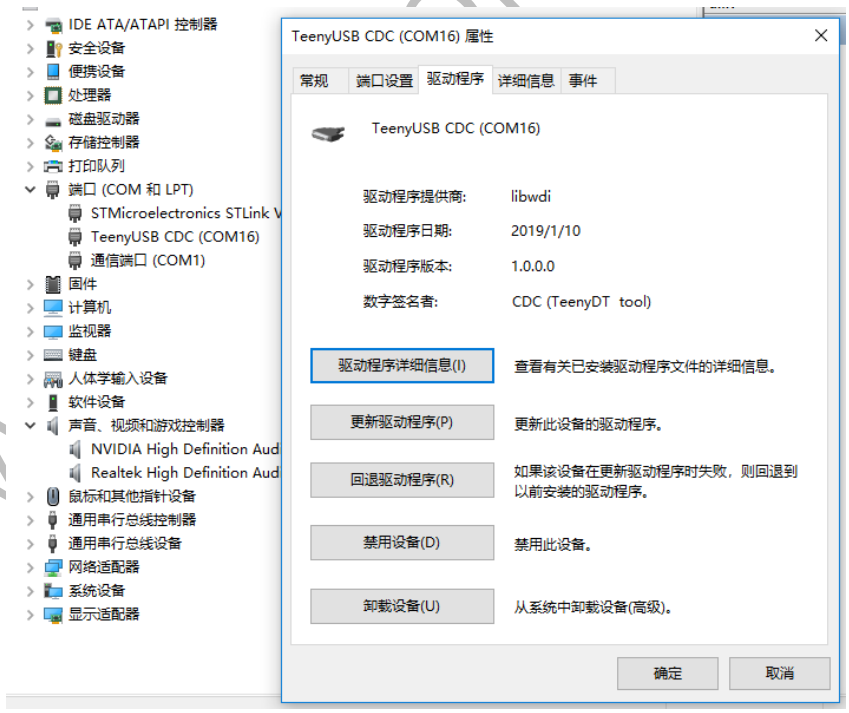


图66　　CDC 驱动安装成功

6.3.7 功能测试

打开测试工具，在【Test Tool】菜单中选择【CDC View】或是 Ctrl+D 打开串口测试串口，选择要测试的串口并打开。在【Send Data】输入框中输入一些数据，点击【Send】后可以看到这些数据都+1 后返回到了【Recv Data】的数据框中中。

6.3.8 多串口复合设备

一个 USB 设备中可以有多个不同的接口，这个设备叫做复合设备，这里以多个串口为例。多串口符合设备的完整代码在 demo/cdc5 目录下。

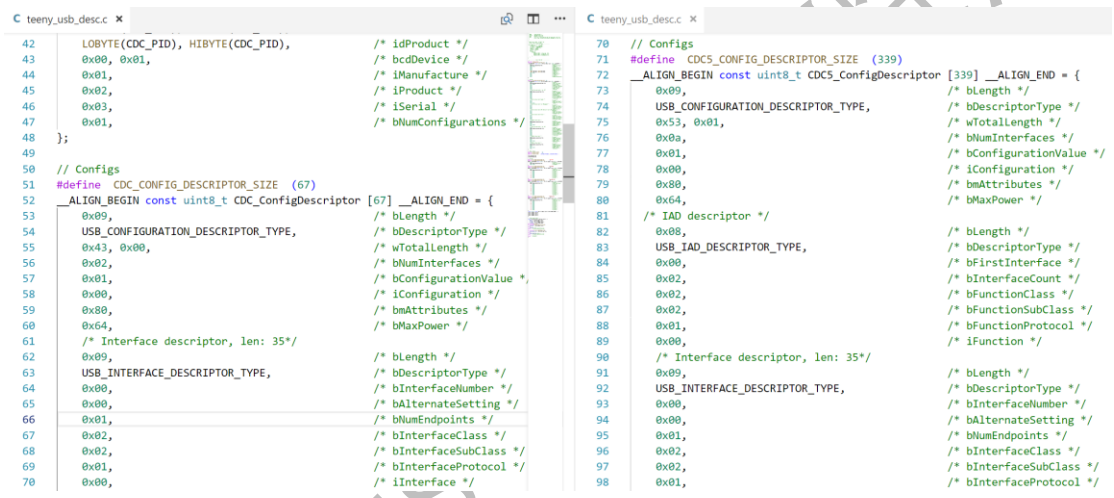
TeenyDT 格式的设备描述符如下：

```
return Device {
    strManufacture = "TeenyUSB",
    strProduct = "TeenyUSB CDC",
    strSerial = "TUSB123456",
    idVendor = 0x0483,
    idProduct = 0x0005,
    prefix = "CDC5",
    Config {
        CDC_ACM{
            EndPoint(IN(8),  Interrupt,  8),
            EndPoint(IN(1),  BulkDouble, 64),
            EndPoint(OUT(1), BulkDouble, 64),
        },
        CDC_ACM{
            EndPoint(IN(9),  Interrupt,  8),
            EndPoint(IN(2),  BulkDouble, 32),
            EndPoint(OUT(2), BulkDouble, 32),
        },
        CDC_ACM{
            EndPoint(IN(10),  Interrupt,  8),
            EndPoint(IN(3),  BulkDouble, 32),
            EndPoint(OUT(3), BulkDouble, 32),
        },
        CDC_ACM{
            EndPoint(IN(11),  Interrupt,  8),
            EndPoint(IN(4),  BulkDouble, 16),
            EndPoint(OUT(4), BulkDouble, 16),
        },
        CDC_ACM{
            EndPoint(IN(12),  Interrupt,  8),
            EndPoint(IN(5),  BulkDouble, 16),
            EndPoint(OUT(5), BulkDouble, 16),
```

```
        },
    }
}
```

上面定义了一个有 5 个串口的复合设备。5 个串口需要 10 个额外的端点，而 STM32 系列芯片最多只有 9 个端点，理论上做不出 5 个串口。这里采用了一点小手段来骗过系统。数据端点采用真实的端点号，而控制端点采用大于实际端点数的端点号。TeenyDT 在解析到超出设备最大端点数的端点号时，会忽略这个端点，不生成代码。描述符中类型为 BulkDouble 的端点，会根据实际情况来决定是否采用双缓冲。上面的例子中，由于端点的 IN 和 OUT 方向都被占用，不会采用双缓冲。由于 STM32F1 系列芯片的 USB 专用缓存只有 512 字节，因此端点的最大包长不能全部都设置为 64 字节。这里端点最大包长逐渐减少，让 USB 缓存使用不超过 512 字节。使用 TeenyDT 生成描述符时，如果 USB 缓存溢出，会进行错误提示。

如果观察单串口和多串口生成的描述符 C 代码，会发现单串口的代码中没有 IAD 描述符，而多串口的描述符中有 IAD 描述符。关于 IAD 描述符的使用条件，在 3.4 接口联合描述符中有详细介绍。



```
    LOBYTE(CDC_PID), HIBYTE(CDC_PID),           /* idProduct */
    0x00, 0x01,                                 /* bcdDevice */
    0x01,                                       /* iManufacture */
    0x02,                                       /* iProduct */
    0x03,                                       /* iSerial */
    0x01,                                       /* bNumConfigurations */
};

// Configs
#define  CDC_CONFIG_DESCRIPTOR_SIZE  (67)
__ALIGN_BEGIN const uint8_t CDC_ConfigDescriptor [67] __ALIGN_END = {
    0x09,                                       /* bLength */
    USB_CONFIGURATION_DESCRIPTOR_TYPE,          /* bDescriptorType */
    0x43, 0x00,                                 /* wTotalLength */
    0x02,                                       /* bNumInterfaces */
    0x01,                                       /* bConfigurationValue */
    0x00,                                       /* iConfiguration */
    0x80,                                       /* bmAttributes */
    0x64,                                       /* bMaxPower */
    /* Interface descriptor, len: 35*/
    0x09,                                       /* bLength */
    USB_INTERFACE_DESCRIPTOR_TYPE,              /* bDescriptorType */
    0x00,                                       /* bInterfaceNumber */
    0x00,                                       /* bAlternateSetting */
    0x01,                                       /* bNumEndpoints */
    0x02,                                       /* bInterfaceClass */
    0x02,                                       /* bInterfaceSubClass */
    0x01,                                       /* bInterfaceProtocol */
    0x00,                                       /* iInterface */
```

```
// Configs
#define  CDC5_CONFIG_DESCRIPTOR_SIZE  (339)
__ALIGN_BEGIN const uint8_t CDC5_ConfigDescriptor [339] __ALIGN_END = {
    0x09,                                       /* bLength */
    USB_CONFIGURATION_DESCRIPTOR_TYPE,          /* bDescriptorType */
    0x53, 0x01,                                 /* wTotalLength */
    0x0a,                                       /* bNumInterfaces */
    0x01,                                       /* bConfigurationValue */
    0x00,                                       /* iConfiguration */
    0x80,                                       /* bmAttributes */
    0x64,                                       /* bMaxPower */
  /* IAD descriptor */
    0x08,                                       /* bLength */
    USB_IAD_DESCRIPTOR_TYPE,                    /* bDescriptorType */
    0x00,                                       /* bFirstInterface */
    0x02,                                       /* bInterfaceCount */
    0x02,                                       /* bFunctionClass */
    0x02,                                       /* bFunctionSubClass */
    0x01,                                       /* bFunctionProtocol */
    0x00,                                       /* iFunction */
    /* Interface descriptor, len: 35*/
    0x09,                                       /* bLength */
    USB_INTERFACE_DESCRIPTOR_TYPE,              /* bDescriptorType */
    0x00,                                       /* bInterfaceNumber */
    0x00,                                       /* bAlternateSetting */
    0x01,                                       /* bNumEndpoints */
    0x02,                                       /* bInterfaceClass */
    0x02,                                       /* bInterfaceSubClass */
    0x01,                                       /* bInterfaceProtocol */
```

图67　单串口与多串口描述符对比

在 CDC5 示例中，接收到数据后根据接口编号对数据逐字节操作，第 1 个接口数据+1，第 2 个接口数据+2，以此类推。代码如下：

```
#define  MAX_EP  5
typedef struct _cdc{
  uint8_t   buf[256];
  uint32_t  data_cnt;
}cdc_t;
cdc_t cdc[MAX_EP];
// if data tx done, set rx valid again
void tusb_on_tx_done(tusb_device_t* dev, uint8_t EPn){
  if(EPn>0 && EPn<=MAX_EP){
    tusb_set_rx_valid(dev, EPn);
  }
}
int tusb_on_rx_done(tusb_device_t* dev, uint8_t EPn, const void* data, uint16_t len){
  if(EPn>0 && EPn<=MAX_EP){
    uint8_t idx = EPn-1;
```

```
    cdc[idx].data_cnt = len;
    return len;
  }
  return 0;
}
void tusb_reconfig(tusb_device_t* dev){
  uint32_t i;
  // call the BULK device init macro
  CDC5_TUSB_INIT(dev);
  for(i=1; i<= MAX_EP;i++){
    tusb_set_recv_buffer(dev, i, cdc[i-1].buf, sizeof(cdc[i-1].buf));
    cdc[i-1].data_cnt = 0;
    tusb_set_rx_valid(dev, i);
  }
}
void delay_ms(uint32_t ms){
  uint32_t i,j;
  for(i=0;i<ms;++i)
    for(j=0;j<20;++j);
}
int main(void){
#if defined(STM32F723xx)
  tusb_device_t* dev = tusb_get_device(1);
#else
  tusb_device_t* dev = tusb_get_device(0);
#endif
  tusb_close_device(dev);
  delay_ms(100);
  tusb_open_device(dev);
  while(1){
    uint32_t idx;
    for(idx=0;idx<MAX_EP;idx++){
      cdc_t* pc = &cdc[idx];
      if(pc->data_cnt){
        for(int i=0;i<pc->data_cnt;i++){
          pc->buf[i] += (idx+1);
        }
        tusb_send_data(dev, idx+1, pc->buf, pc->data_cnt);
        pc->data_cnt = 0;
      }
    }
  }
}
```

由于这个只是做一个数据的回环测是，没有连接设备上的真实串口，因此在处理 Set/Get

Linecoding 请求时，没有根据接口号来进行区分。

多串口复合设备接入电脑时设备管理器如下：

图68　　CDC5 接入电脑

这里没有为多串口设备生成驱动，在 Win10 系统上可以正常使用。使用测试工具进行测试，如下图所示：

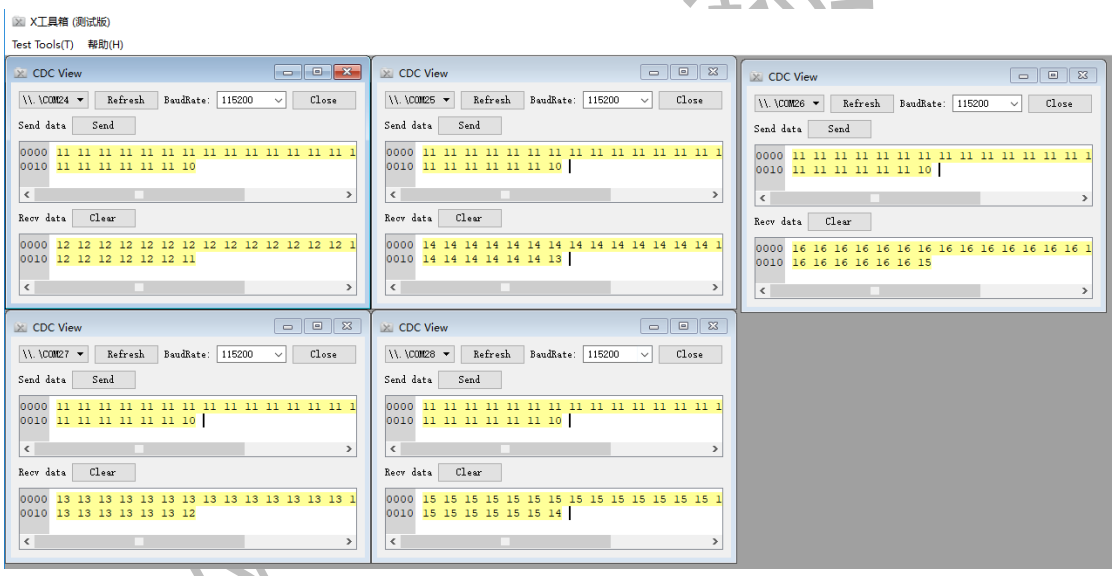


图69　　CDC5 测试

在测试工具的【Test Tool】菜单下选择【CDC View】或是用快捷键 Ctrl+D 打开 CDC 测试窗口。不同的串口对输入数据加上不同值后再发回 PC。

6.4 大容量存储设备 MSC

MSC 是 Mass Storage Class 的缩写，是大容量存储类的意思，通常叫做 U 盘。U 盘只提供最基本的存储区读写操作功能，文件系统这类内容都是由操作系统来完成的。因此一个 U 盘可以被格式化成各种不同的文件系统。大容量存储设备的完整代码在 demo/msc 目录中，这个项目依赖 ST HAL 库中的 USB MSC 设备类，MSC 设备类的代码路径为：
Drivers\Middlewares\ST\STM32_USB_Device_Library\Class\MSC
STM32F1 和 F0 的测试程序，还依赖 HAL 库中的 Flash 部分内容。

6.4.1 确定设备功能

实现一个 U 盘功能，对于 RAM 小的设备，使用芯片内部 Flash 的空余空间来模拟存储区。对于 RAM 较大的设备，使用芯片的 RAM 来模拟 U 盘，断电后数据丢失。这里将使用 ST 的 HAL 库来实现 U 盘相关的代码。

6.4.2 描述符设计

ST 的 HAL 库实现了 MSC 的 BOT 协议，这里也采用 MSC 的 BOT 协议，描述符内容如下：

```
return Device {
    strManufacture = "TeenyUSB",
    strProduct = "TeenyUSB MSC",
    strSerial = "TUSB123456",
    idVendor = 0x0483,
    idProduct = 0x0003,
    prefix = "MSC",
    Config {
        Interface{
            bInterfaceClass = 0x08,        -- MSC
            bInterfaceSubClass = 0x06,     -- SCSI
            bInterfaceProtocol = 0x50,     -- BOT
            EndPoint(IN(1),  BulkDouble, 64),
            EndPoint(OUT(1), BulkDouble, 64),
        },
   }
}
```

与自定义设备相比，将接口的 bInterfaceClass、bInterfaceSubClass 和 bInterfaceProtocol 分别设置为 0x08、0x06、0x50。为了与 HAL 库保持一致，只使用了端点 1 来收发数据。这里虽然使用的是 BulkDouble 类型，但是双缓冲只能用于不同编号的端点，因此在 F1/F0 设备上，双缓冲会自动停用。在生成描述符时会有警告信息提醒。

TeenyDT 格式描述符的详细介绍在 3.10 描述符工具 TeenyDT 这一节中。

6.4.3 初始化设备端点

TeenyDT 自动生成的端点初始化代码如下：

```
// EndPoints init function for USB FS core
#define MSC_TUSB_INIT_EP_FS(dev) \
  do{\
    /* Init ep0 */ \
    INIT_EP_BiDirection(dev, PCD_ENDP0, MSC_EP0_TYPE);  \
    SET_TX_ADDR(dev, PCD_ENDP0, MSC_EP0_TX_ADDR);  \
    SET_RX_ADDR(dev, PCD_ENDP0, MSC_EP0_RX_ADDR);  \
    SET_RX_CNT(dev, PCD_ENDP0, MSC_EP0_RX_SIZE);  \
    /* Init ep1 */ \
    INIT_EP_BiDirection(dev, PCD_ENDP1, MSC_EP1_TYPE);  \
    SET_TX_ADDR(dev, PCD_ENDP1, MSC_EP1_TX_ADDR);  \
    SET_RX_ADDR(dev, PCD_ENDP1, MSC_EP1_RX_ADDR);  \
    SET_RX_CNT(dev, PCD_ENDP1, MSC_EP1_RX_SIZE);  \
  }while(0)
// EndPoints init function for USB OTG core
#define MSC_TUSB_INIT_EP_OTG(dev) \
  do{\
    SET_RX_FIFO(dev, MSC_OTG_RX_FIFO_ADDR, MSC_OTG_RX_FIFO_SIZE);  \
    /* Init ep0 */ \
    INIT_EP_Tx(dev, PCD_ENDP0, MSC_EP0_TYPE, MSC_EP0_TX_SIZE);  \
    SET_TX_FIFO(dev, PCD_ENDP0, MSC_EP0_TX_FIFO_ADDR, MSC_EP0_TX_FIFO_SIZE);  \
    INIT_EP_Rx(dev, PCD_ENDP0, MSC_EP0_TYPE, MSC_EP0_RX_SIZE);  \
    /* Init ep1 */ \
    INIT_EP_Tx(dev, PCD_ENDP1, MSC_EP1_TYPE, MSC_EP1_TX_SIZE);  \
    SET_TX_FIFO(dev, PCD_ENDP1, MSC_EP1_TX_FIFO_ADDR, MSC_EP1_TX_FIFO_SIZE);  \
    INIT_EP_Rx(dev, PCD_ENDP1, MSC_EP1_TYPE, MSC_EP1_RX_SIZE);  \
  }while(0)
```

6.4.4 设备类型相关请求处理

MSC 设备类型相关请求处理代码如下：

```
__ALIGN_BEGIN uint32_t cmd_buf __ALIGN_END;
int tusb_class_request(tusb_device_t* dev, tusb_setup_packet* setup_req)
{
  USBD_HandleTypeDef* pdev = &hDev;
  if( (setup_req->bmRequestType & USB_REQ_TYPE_MASK) == USB_REQ_TYPE_CLASS){
    switch (setup_req->bRequest){
    case BOT_GET_MAX_LUN :
      if((setup_req->wValue  == 0) &&
        (setup_req->wLength == 1) &&
```

```
      ((setup_req->bmRequestType & 0x80) == 0x80)){
        MSCData.max_lun = ((USBD_StorageTypeDef *)pdev->pUserData)->GetMaxLun();
        cmd_buf = MSCData.max_lun;
        tusb_control_send(dev, &cmd_buf,1);
      }
      return 1;
    case BOT_RESET :
      if((setup_req->wValue  == 0) &&
        (setup_req->wLength == 0) &&
      ((setup_req->bmRequestType & 0x80) != 0x80)){
         MSC_BOT_Reset(pdev);
      }
      tusb_send_status(dev);
      return 1;
    }
  }
  return 0;
}
```

上面的代码只处理了 BOT_GET_MAX_LUN 和 BOT_RESET 请求。

6.4.5 设备功能实现

ST 的 HAL 库中，MSC 协议栈数据处理流程如下图所示：

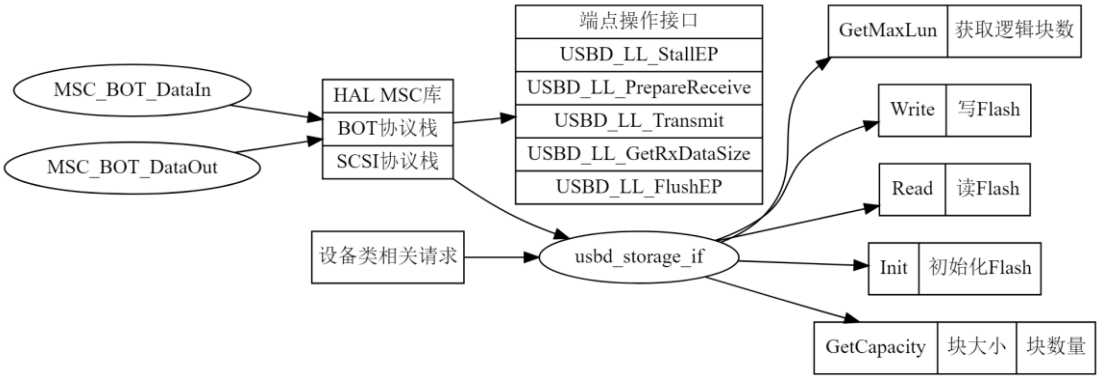


图70　　HAL 库 MSC 协议栈

MSC_BOT_DataIn 和 MSC_BOT_DataOut 函数是 MSC 协议栈数据处理的总入口，数据经过协议栈处理后，访问 usbd_storage_if 接口进行数据的读写等操作。这里需要实现端点操作接口，usbd_storage_if，在端点的 IN 和 OUT 传输完成事件中调用 MSC_BOT_DataIn 和 MSC_BOT_DataOut。传输完成事件的处理代码如下：

```
uint8_t msc_in = 0;
uint8_t msc_out = 0;
void tusb_on_tx_done(tusb_device_t* dev, uint8_t EPn){
  if(EPn == (MSC_EPIN_ADDR & 0x7f) ){
    msc_in = 1;
  }
```

```
}
int tusb_on_rx_done(tusb_device_t* dev, uint8_t EPn, const void* data, uint16_t len){
  if(EPn == (MSC_EPOUT_ADDR & 0x7f) ){
    msc_out = 1;
    return 1;
  }
  return 0;
}
void tusb_reconfig(tusb_device_t* dev){
  // call the MSC device init macro
  MSC_TUSB_INIT(dev);
  // Init the HAL MSC class
  init_hal_msc_class(dev);
}
int main(void){
...
  while(1){
    if(msc_in){
      msc_in_data();
      msc_in = 0;
    }
    if(msc_out){
      msc_out_data();
      msc_out = 0;
    }
  }
}
```

在传输完成的回调函数中设置相应的标志，在主循环中根据标志调用 MSC 的数据处理函数。在 tusb_reconfig 回调函数中，初始化端点，并调用 init_hal_msc_class 对 HAL 库的 MSC 协议栈进行初始化。

TeenyUSB 与 HAL 库对 USB 设备的操作不同，对 MSC 协议栈中用到的 USBD_LL_xxx 函数进行了重新封装，代码如下：

```
USBD_HandleTypeDef  hDev;
extern USBD_StorageTypeDef USBD_Storage_Interface_fops_FS;
void msc_in_data(void){
  MSC_BOT_DataIn(&hDev, MSC_EPIN_ADDR);
}
void msc_out_data(void){
  MSC_BOT_DataOut(&hDev, MSC_EPOUT_ADDR);
}
USBD_MSC_BOT_HandleTypeDef  MSCData;
void init_hal_msc_class(tusb_device_t* dev){
  hDev.pUserData = &USBD_Storage_Interface_fops_FS;
  hDev.pClassData = &MSCData;
```

```
  hDev.pData = dev;
  {
    USBD_HandleTypeDef *pdev = &hDev;
    USBD_MSC_BOT_HandleTypeDef  *hmsc = (USBD_MSC_BOT_HandleTypeDef*)pdev->pClassData;
    hmsc->bot_state  = USBD_BOT_IDLE;
    hmsc->bot_status = USBD_BOT_STATUS_NORMAL;
    hmsc->scsi_sense_tail = 0;
    hmsc->scsi_sense_head = 0;
    ((USBD_StorageTypeDef *)pdev->pUserData)->Init(0);
    USBD_LL_FlushEP(pdev, MSC_EPOUT_ADDR);
    USBD_LL_FlushEP(pdev, MSC_EPIN_ADDR);

    /* Prapare EP to Receive First BOT Cmd */
    USBD_LL_PrepareReceive (pdev,
                            MSC_EPOUT_ADDR,
                            (uint8_t *)&hmsc->cbw,
                            USBD_BOT_CBW_LENGTH);
  }
}
#if defined(USB_OTG_FS) || defined(USB_OTG_HS)
void flush_rx(USB_OTG_GlobalTypeDef *USBx);
void flush_tx(USB_OTG_GlobalTypeDef *USBx, uint32_t num);
#endif
USBD_StatusTypeDef USBD_LL_FlushEP(USBD_HandleTypeDef *pdev, uint8_t ep_addr){
#if defined(USB_OTG_FS) || defined(USB_OTG_HS)
  tusb_device_t* dev = (tusb_device_t*)pdev->pData;
  if(ep_addr&0x80){
    flush_tx(GetUSB(dev), ep_addr & 0x7f);
  }else{
    flush_rx(GetUSB(dev));
  }
#endif
  return USBD_OK;
}
uint32_t USBD_LL_GetRxDataSize(USBD_HandleTypeDef *pdev, uint8_t ep_addr){
  return USBD_BOT_CBW_LENGTH;
}
USBD_StatusTypeDef USBD_LL_PrepareReceive(USBD_HandleTypeDef *pdev, uint8_t ep_addr, uint8_t *pbuf,
uint16_t size){
  tusb_device_t* dev = (tusb_device_t*)pdev->pData;
  tusb_set_recv_buffer(dev, ep_addr&0x7f, pbuf, size);
  tusb_set_rx_valid(dev, ep_addr&0x7f);
  return USBD_OK;
}
```

```
USBD_StatusTypeDef USBD_LL_StallEP(USBD_HandleTypeDef  *pdev, uint8_t ep){
  tusb_device_t* dev = (tusb_device_t*)pdev->pData;
#if defined(USB)
  (void)dev;
  if( (ep & 0x7f) == 0){
    PCD_SET_EP_TXRX_STATUS(GetUSB(dev), 0, USB_EP_RX_STALL, USB_EP_TX_STALL);
  }else{
    if(ep & 0x80){
      PCD_SET_EP_TX_STATUS(GetUSB(dev), ep &0x7f , USB_EP_TX_STALL);
    }else{
      PCD_SET_EP_RX_STATUS(GetUSB(dev), ep &0x7f , USB_EP_RX_STALL);
    }
  }
#endif
#if defined(USB_OTG_FS) || defined(USB_OTG_HS)
  PCD_TypeDef* USBx =  GetUSB(dev);
  if( (ep & 0x7f) == 0){
    USBx_INEP(0)->DIEPCTL |= USB_OTG_DIEPCTL_STALL;
    USBx_OUTEP(0)->DOEPCTL |= USB_OTG_DOEPCTL_STALL;
  }else{
    if(ep & 0x80){
      USBx_INEP(0)->DIEPCTL |= USB_OTG_DIEPCTL_STALL;
    }else{
      USBx_OUTEP(0)->DOEPCTL |= USB_OTG_DOEPCTL_STALL;
    }
  }
#endif
  return USBD_OK;
}
USBD_StatusTypeDef USBD_LL_Transmit(USBD_HandleTypeDef *pdev, uint8_t ep_addr, uint8_t *pbuf,
uint16_t size){
  tusb_device_t* dev = (tusb_device_t*)pdev->pData;
  tusb_send_data(dev, ep_addr&0x7f, pbuf, size);
  return USBD_OK;
}
```

usbd_storage_if 中主要实现的函数如下：

```
#if defined(FLASH_SIM_MSC)
void erase_page(uint32_t addr){
  uint32_t PAGEError;
  FLASH_EraseInitTypeDef EraseInitStruct;
  EraseInitStruct.TypeErase   = FLASH_TYPEERASE_PAGES;
  EraseInitStruct.PageAddress = addr;
  EraseInitStruct.NbPages     = 1;
  if(HAL_FLASHEx_Erase(&EraseInitStruct, &PAGEError) != HAL_OK){
```

```
    while(1);
  }
}
void write_buf(uint32_t addr, uint8_t* buf, uint32_t size){
  uint32_t endAddr = addr+size;
  uint32_t* pData = (uint32_t*)buf;
  while(addr<endAddr){
    HAL_FLASH_Program(FLASH_TYPEPROGRAM_WORD, addr, *pData);
    addr+=4;
    pData++;
  }
}
void write_data(uint32_t addr, uint8_t* buf, uint32_t size){
  addr += MSC_START_ADDR;
  while(size >= PAGE_SIZE){
    erase_page(addr);
    write_buf(addr,buf, PAGE_SIZE);
    size-=PAGE_SIZE;
    addr+=PAGE_SIZE;
  };
  return;
}
void read_data(uint32_t addr, uint8_t* buf, uint32_t size){
  memcpy(buf, (uint8_t*)(addr+MSC_START_ADDR), size);
}
void init_data(void){
  HAL_FLASH_Unlock();
}
#elif defined(RAM_SIM_MSC)
void init_data(void){}
void read_data(uint32_t addr, uint8_t* buf, uint32_t size){
  memcpy(buf, (uint8_t*)(addr+RAM_START_ADDR), size);
}
void write_data(uint32_t addr, uint8_t* buf, uint32_t size){
  memcpy((uint8_t*)(addr+RAM_START_ADDR), buf, size);
}
#else
#error Unknown MSC media
#endif
int8_t STORAGE_Init_FS(uint8_t lun){
  init_data();
  return (USBD_OK);
}
int8_t STORAGE_Read_FS(uint8_t lun, uint8_t *buf, uint32_t blk_addr, uint16_t blk_len){
```

```
  uint32_t  addr = (blk_addr*STORAGE_BLK_SIZ);
  read_data(addr, buf, blk_len*STORAGE_BLK_SIZ);
  return (USBD_OK);
}
int8_t STORAGE_Write_FS(uint8_t lun, uint8_t *buf, uint32_t blk_addr, uint16_t blk_len){
  write_data((blk_addr * STORAGE_BLK_SIZ),  buf, blk_len*STORAGE_BLK_SIZ);
  return (USBD_OK);
}
```

如果是用 Flash 模拟存储，调用 Flash 相关的读写函数来实现 STORAGE_Read_FS 和 STORAGE_Write_FS 函数。如果是用 RAM 模拟存储，直接对数据进行复制。

6.4.6 设备驱动

大容量存储设备系统支持，不需要编写驱动。

6.4.7 功能测试

设备接入电脑后会提示有 U 盘插入，系统会为新插入的 U 盘生成盘符。并根据 U 盘中的内容提示是否需要格式化。格式化成功后就可以像普通 U 盘那样操作了。

6.5 自定义 HID 设备

自定义 HID 设备也是常见的一种设备类型，这类设备在所有的 Windows 系统上都是免驱的，即插即用。HID 是 Human Input Device 的简称，键盘、鼠标、游戏摇杆都属于这类设备。

6.5.1 确定设备功能

自定义 HID 设备使用一个 IN 和一个 OUT 端点来传输，端点类型都是中断传输，最大包长为 64 字节。与之前的测试设备一样，接收到的数据逐字节+1 后发回主机。

6.5.2 描述符设计

HID 设备的接口描述符中需要增加一个 HID 描述符，在 HID 描述符中指定报告描述符的长度。HID 设备需要相应获取报告描述符的请求。报告描述符是 HID 设备特有的一种描述符，其详细介绍在 HID 设备规格书中，在 https://www.usb.org/documents 可以下载。USB 组织为了简化报告描述符的编写，提供了报告描述符工具，在下面这个地址可以下载：
https://www.usb.org/document-library/hid-descriptor-tool

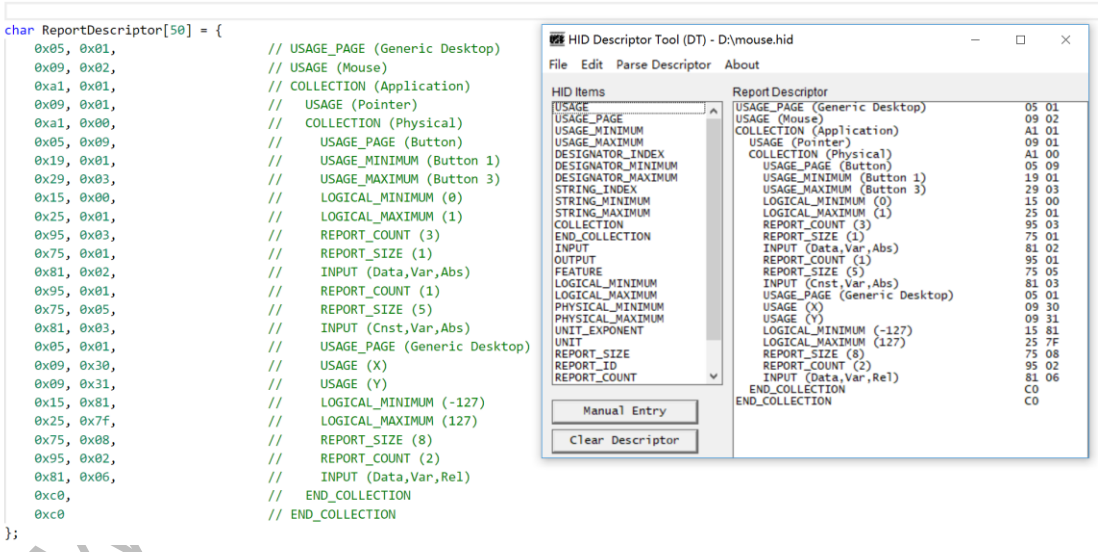


图71　　HID 报告描述符工具

HID 报告描述符工具可以生成 C 语言格式的报告描述符，C 语言格式的报告描述符也可以在 TeenyDT 中使用。完整的自定义 HID 设备的描述符如下：

```
return Device {
    strManufacture = "TeenyUSB",
    strProduct = "TeenyUSB Custom HID",
    strSerial = "TUSB123456",
    idVendor = 0x0483,
    idProduct = 0x0004,
    prefix = "HID",
```

```
    Config {
        USB_HID{
            ReadEp = EndPoint(IN(1),  Interrupt, 64),
            WriteEp = EndPoint(OUT(1), Interrupt, 64),
            report = HID_BuildReport([[
  // report descriptor for general input/output
  0x06, 0x00, 0xFF,  // Usage Page (Vendor Defined 0xFF00)
  0x09, 0x01,        // Usage (0x01)
  0xA1, 0x01,        // Collection (Application)
  0x09, 0x02,        //   Usage (0x02)
  0x15, 0x00,        //   Logical Minimum (0)
  0x25, 0xFF,        //   Logical Maximum (255)
  0x75, 0x08,        //   Report Size (8)
  0x95, 0x40,        //   Report Count 64
  0x81, 0x02,        //   Input (Data,Var,Abs,No Wrap,Linear,Preferred State,No Null Position)
  0x09, 0x03,        //   Usage (0x03)
  0x91, 0x02,        //   Output (Data,Var,Abs,No Wrap,Linear,Preferred State,No Null Position,Non-
volatile)
  0xC0               // End Collection
]]),
        },
    }
}
```

接口使用的是 USB_HID 来生成，USB_HID 与之前的 CDC_ACM 类似，都是将特定类型相关的描述符进行了封装。USB_HID 的实现如下：

```
function USB_HID(param)
    local Hid
    local ReadEp = param.ReadEp or param[1]
    local WriteEp = param.WriteEp or param[2]

    Hid = Interface{
        bInterfaceClass = 3,
        bInterfaceSubClass = param.isBoot and 1 or 0,
        bInterfaceProtocol = param.isKey and 1 or (param.isMouse and 2 or 0),
        HIDDescriptor{
            param.report,
            param.physical,
        },
        ReadEp,
        WriteEp,
    }
    Hid.report = param.report
    Hid.physical = param.physical
    Hid.extraDesc = outputReportDescriptor
```

```
    Hid.hasReport = true
    return Hid
end
```

在普通的接口描述符中，增加了 HIDDescriptor，HIDDescriptor 需要报告描述符和物理描述符来初始化。其中报告描述符是必须的，物理描述符是可选的。报告描述可以写成下面的形式，生成的代码可读性比较差：

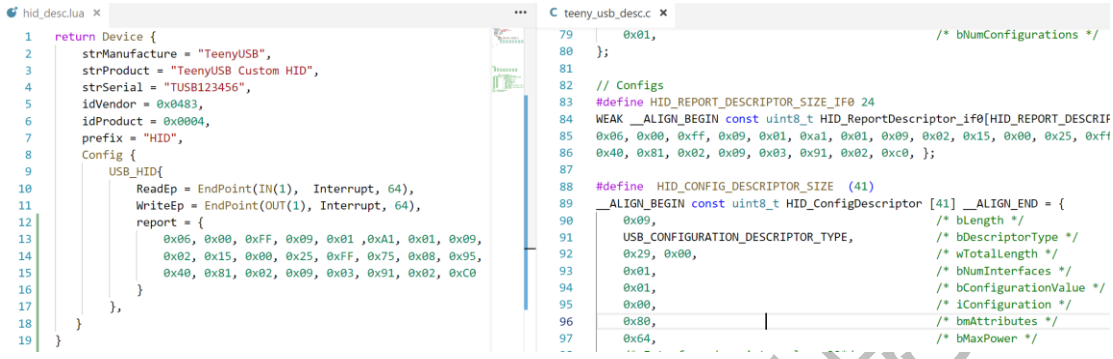


图72　　数组格式报告描述符

也可以用 HID_BuildReport 直接将 HID 报告描述工具生成的 C 格式描述符转成 TeenyDT 格式，这样生成的描述符可读性较好，如下图：

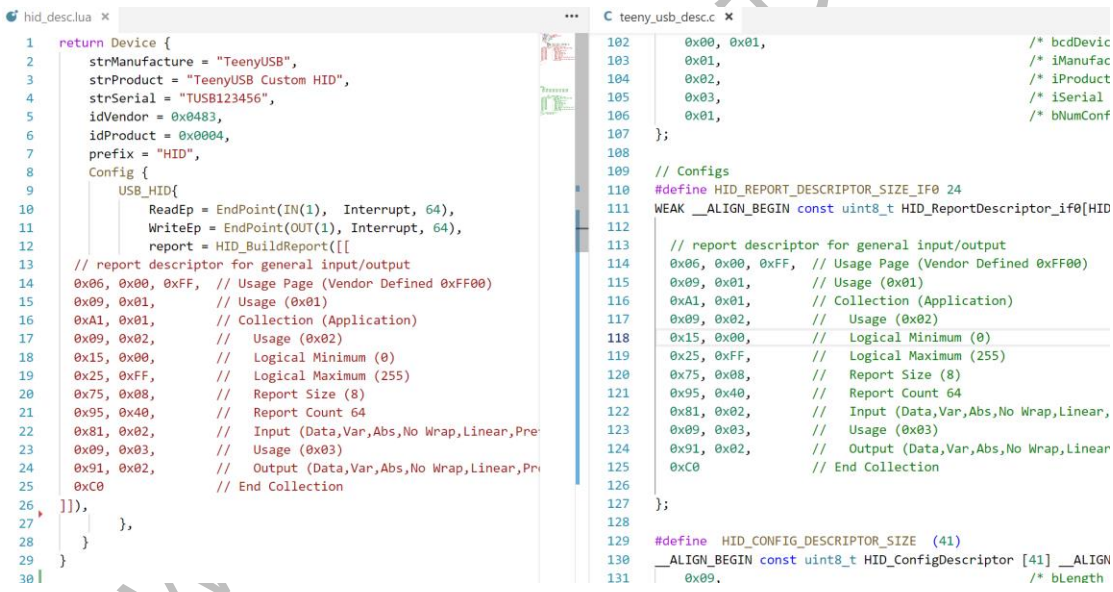


图73　　C 格式报告描述符

图形化工具也可以生成 HID 设备的描述符，如下图：

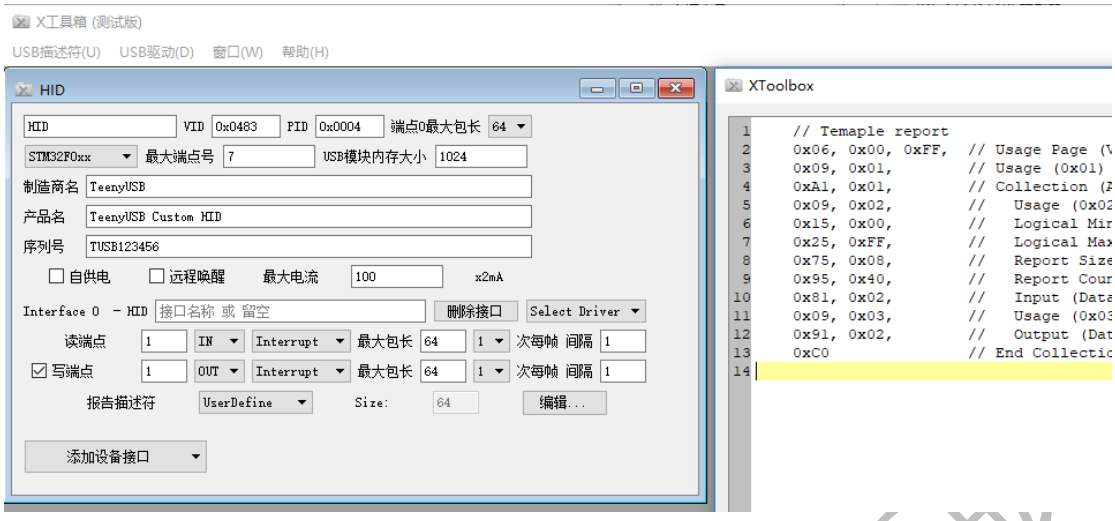


图74 HID 设备描述符图形化方式

图形化方式提供三种预定义的报告描述符，分别是：

InOut 基本的数据输入输出功能，只需要指定数据的大小

BootMouse Boot 协议的鼠标

BootKeyBoard Boot 协议的键盘

用户也可以自定义报告描述符，报告描述符类型选择【UserDefine】，然后点击【编辑】按钮，在弹出的对话框中可以修改自定义的报告描述符，兼容 HID 报告描述工具生成的 C 语言格式。

6.5.3 初始化设备端点

端点初始化代码由 TeenyDT 自动生成，在 tusb_reconfig 回调函数中调用。

6.5.4 设备类型相关请求处理

HID 设备类型相关请求处理代码如下：

```
__ALIGN_BEGIN static uint8_t hid_cmd[64] __ALIGN_END;
// make sure data is 32bit aligned
__ALIGN_BEGIN static uint32_t cmd_buff __ALIGN_END;
static uint8_t USBD_HID_IdleState;
static uint8_t USBD_HID_Protocol;
void HID_DataoutRequest(tusb_device_t* dev)
{
  uint8_t reportID = LO_BYTE(dev->setup.wValue);
  (void)reportID;
  // got output report
}

int tusb_class_request(tusb_device_t* dev, tusb_setup_packet* setup_req)
{
```

```
  // Get interface report descriptor
  if( (setup_req->bmRequestType & USB_REQ_RECIPIENT_MASK) == USB_REQ_RECIPIENT_INTERFACE ){
    if(setup_req->bRequest == USB_REQ_GET_DESCRIPTOR){
      if(HI_BYTE(setup_req->wValue) == USB_DESC_TYPE_REPORT){
        uint16_t ifn = setup_req->wIndex;
        if(ifn < 1){
          uint16_t len = HID_REPORT_DESCRIPTOR_SIZE_IF0;
          const uint8_t* desc = HID_ReportDescriptor_if0;
          len = setup_req->wLength > len ? len : setup_req->wLength;
#if defined(DESCRIPTOR_BUFFER_SIZE) && DESCRIPTOR_BUFFER_SIZE > 0
          if(dev->desc_buffer != desc){
            memcpy(dev->desc_buffer, desc, len);
            desc = dev->desc_buffer;
          }
#endif
          tusb_control_send(dev, desc, len);
          return 1;
        }
      }
    }
  }
  // handle class request
  if( (setup_req->bmRequestType & USB_REQ_TYPE_MASK) == USB_REQ_TYPE_CLASS ){
    switch(setup_req->bRequest){
      case HID_REQ_SET_IDLE:
        USBD_HID_IdleState = HI_BYTE(setup_req->wValue);
        tusb_send_status(dev);
        return 1;
      case HID_REQ_GET_IDLE:
        cmd_buff = USBD_HID_IdleState;
        tusb_control_send(dev, &cmd_buff, 1);
        return 1;
      case HID_REQ_SET_PROTOCOL:
        USBD_HID_Protocol = LO_BYTE(setup_req->wValue);
        tusb_send_status(dev);
        return 1;
      case HID_REQ_GET_PROTOCOL:
        cmd_buff = USBD_HID_Protocol;
        tusb_control_send(dev,  &cmd_buff, 1);
        return 1;
      case HID_REQ_SET_REPORT:
        tusb_set_recv_buffer(dev, 0, hid_cmd, setup_req->wLength);
        dev->ep0_rx_done = HID_DataoutRequest;
        return 1;
```

```
      case HID_REQ_GET_REPORT:
      {
        uint8_t reportID = LO_BYTE(dev->setup.wValue);
        (void)reportID;
        tusb_control_send(dev,  &hid_cmd, setup_req->wLength);
        return 1;
      }
      default:
        return 0;
    }
  }
  return 0;
}
```

自定义 HID 设备需要返回接口的报告描述符，如果有多个接口，需要根据请求中指定的接口号来返回报告描述符。处理了 Report/Protocol/Idle 这三个请求：

Report 请求，HID 设备定义了三种 Report 请求，分别是 Input，Output，Feature。当设备没有定义 Out 中断传输端点时，可以通过 Set Output Report 请求向设备发送数据。例如标准的 USB 键盘的 LED 灯状态，没有 Out 端点，通过 Set Output Report 请求进行设置。

Protocol 请求设置设备的工作模式，对于键盘和鼠标可以有 Boot Protocol，如果主机不能解析报告描述符，可以通过将键盘鼠标设置为 Boot 模式来使用。设置时 Setup 包的 wVlaue 字段表示 Protocol 值，0 表示 Boot Protocol，1 表示 Report Protocol。读取时通过数据包返回。

Idle 请求设置设备上报间隔，当设置为 0 时，表示只有数据发生变化时才上报。如果不为 0，则无论是否有数据变化，按设置的时间间隔上报。以键盘为例，当 Idle 值为 100ms 时，即使没有按键，也会在 100ms 时上报一个全零的报告。当 Idle 值为 0 时，只有按键变化时才上报。设置时 Setup 包的 wVlaue 字段高 8 为表示 Idle 的时间间隔，以 4ms 为单位，最长为 1020ms。读取时通过数据包返回。

6.5.5 设备功能实现

功能与自定义 USB 设备类似，只是将接收缓存大写设置为了最大包长，这样在收到每一包数据时都会产生传输完成事件。

6.5.6 设备驱动

自定义 HID 设备不需要编写驱动，系统已经提供。

6.5.7 功能测试

测试工具在 pc_test_tool 目录下，HID 设备提供了两种访问方式。一种是 libusb 风格的访问方式，一种是 Windows 的访问方式。

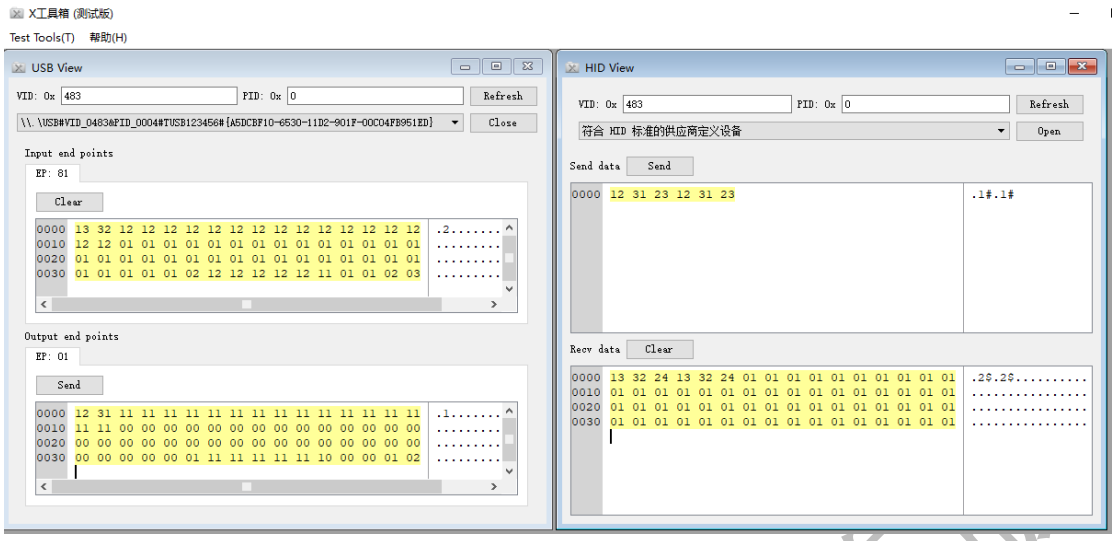


图75　　自定 HID 设备测试

使用 USB View 测试时，用的是 libusb 风格的访问方式。这种方式下要求测试数据必须与报告描述符中的长度一致。如果有报告 ID，第一个字节为报告 ID。

使用 HID View 测试时，用的是 Windows 风格的访问方式。这种方式下数据长度可以是任意值，当小于报告长度时，后面自动补 0。大于报告长度时，截断为报告长度。其实 libusb 对 HID 设备访问的底层实现也是调用 Windows 风格的 API 来完成的。

7 STM32 OTG 模块主机模式

STM32 的 OTG 模块还可以工作在主机模式下，工作在主机模式下时采用了一套与设备不同的寄存器来进行操作。与设备上的端点不同，主机引入了通道的概念。每一个通道都是等价的，根据需要配置成 IN 或 OUT，在不需要的时候还可以将通道回收。本文仅对主机模式做一个简要的介绍。

7.1 主机模式的初始化

核心部分初始化与设备模式相同，时钟，中断，IO 设置与设备模式相同。不同在于内核复位成功后，调用主机模式的初始化代码，代码如下：

```
void tusb_open_host(tusb_host_t* host){
  USB_OTG_GlobalTypeDef* USBx = GetUSB(host);
  tusb_otg_core_init((tusb_core_t*) host);
  // Force to host mode
  USBx->GUSBCFG &= ~(USB_OTG_GUSBCFG_FHMOD | USB_OTG_GUSBCFG_FDMOD);
  USBx->GUSBCFG |= USB_OTG_GUSBCFG_FHMOD;
  tusb_init_otg_host(host);
}
```

主机和设备的初始化都有一个 force to xxx mode 的代码，这里其实也可以不用强制设置为特定模式，由 ID 来区分。当检测到 ID 变化时，根据设备新的角色进行初始化。

7.2 主机的端口操作

主机内核初始化完成后，打开 PORT，等待设备连接。在 HAL 库中，PORT 定义为了 Port0，猜测这个内核可能支持多个 Port，类似于一个 Hub。设备连接时端口的状态变化如下图所示：

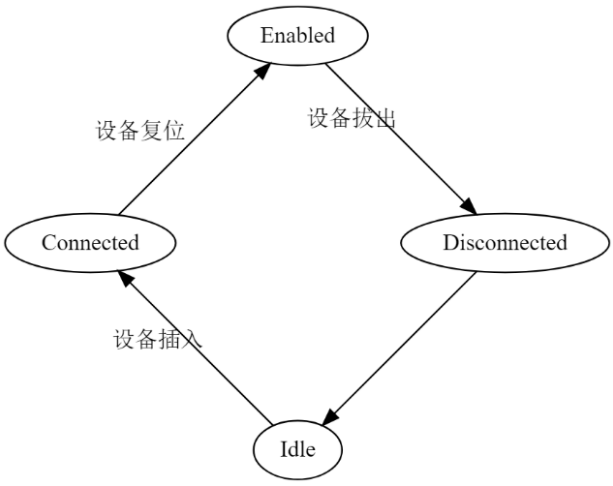


图76　　主机端口状态迁移图

当端口处于 Enabled 状态时，可以操作设备。此时设备完成了复位，复位后设备地址是

0，主机通过 0 地址访问设备，先设置地址，然后用新地址对设备进行枚举。

7.3 主机模式的 FIFO

主机 FIFO 结构见下图：

Figure 435. Host-mode FIFO address mapping and AHB FIFO access mapping

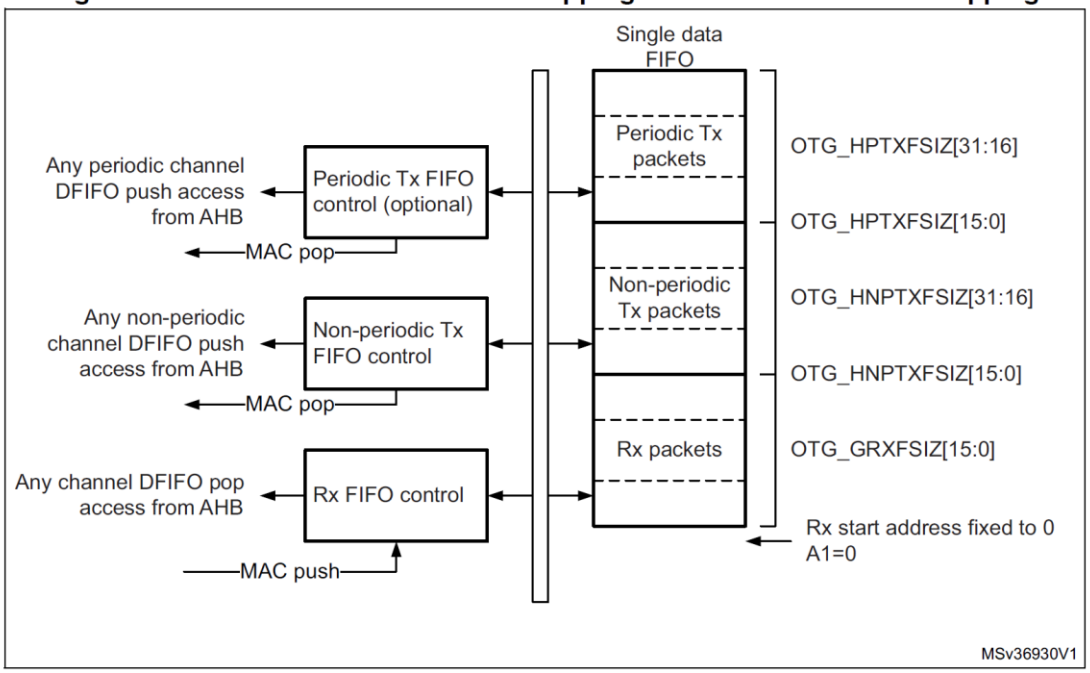


图77　　OTG 主机 FIFO 结构（图片来自 STM32 规格书）

主机模式下一共有三个 FIFO，一个 Rx FIFO 用于所有的 IN 通道，一个周期性 Tx FIFO 用于类型为中断和同步的 OUT 通道，一个非周期性 Tx FIFO，用于类型为控制和 Bulk 的 OUT 通道。

由于主机的 FIFO 模式是固定，因此在初始化时完成 FIFO 空间和地址的分配，需要注意的是，在内核复位后，FIFO 的地址分配需要等待一段时间才能设置成功，这个时间参考的是 HAL 库代码的实现，具体原因不详。

7.4 主机中的请求队列 Request Queue

根据芯片手册，STM32 的 OTG 主机除了数据 FIFO 之外还有一个叫做 Reqeust Queue 的请求队列，这个队列的深度是 8，周期性通道和非周期性通道各有一个。在向 FIFO 中发送数据前，需要检查 FIFO 是否有空余的空间，同时也要检查请求队列是否有空余空间。一个基本事务会占用一个请求队列。对于 IN/OUT 传输而言，一次传输是由多个事务组成的。关于传输于事务的关系，见 2.4 传输完成条件这一节的内容。

例如一次 1024 字节的传输，在最大包长为 64 字节的端点上，至少需要 16 次事务。即使 FIFO 有 1024 字节的空余空间，也不能完全传输。因为 STM32 的 OTG 模块最多只能处理 8 个请求队列。这时需要根据 FIFO 空间和队列空间计算出能够传输的数据数量，写入 FIFO 中，剩余的数据在 NPTXFE 或是 PTXFE 中进行处理。如果启用了 DMA，这部分处理会由 OTG 模块自动完成。

7.5 主机的通道

主机的通道特性寄存器定义如下：

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
CHENA	CHDIS	ODD FRM	DAD[6:0]							MCNT[1:0]		EPTYP[1:0]		LSDEV	Res.
rs	rs	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
EPDIR	EPNUM[3:0]				MPSIZ[10:0]										
rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw

图78　　STM32 通道特性寄存器（图片来自 STM32 规格书）

可以看到，通道可以定义设备地址，通道类型，端点号，端点方向，最大包长等信息。在设备模式下，不能为每个端点配置设备地址，这是因为设备只有一个地址。而在主机模式下，可以为每个通道配置不同的设备地址，这是因为主机模式的通道可以通过 HUB 去访问不同的设备。

上图中的 LSDEV 字段，表示通道是否为低速通道。这个低速不是只当前 STM32 主机端口的速度，而是指目标设备的接口速度。比如 STM32 的主机接口连接了一个全速的 HUB，全速的 HUB 上连了一个低速的键盘。如果要向这个低速的键盘传输数据，需要将通道特性寄存器的速度设置为低速。

7.5.1 通道数据收发

主机通道数据处理流程如下图所示：

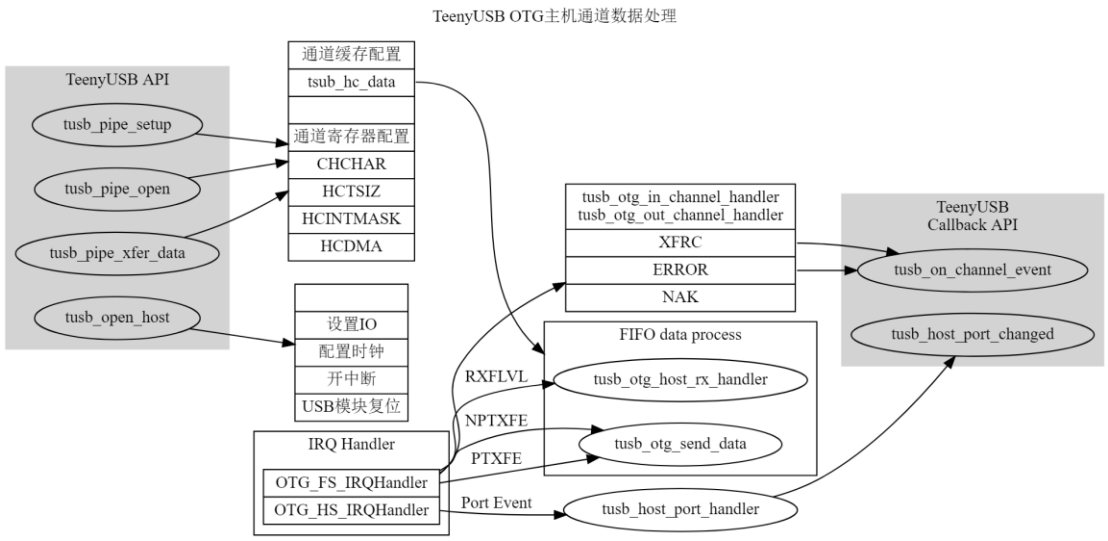


图79　　主机通道数据处理流程

TeenyUSB 提供的 API 说明见下表：

表30　　TeenyUSB 主机模式接口说明

普通函数	说明
tusb_get_host	获得主机，在开始使用USB主机前，调用此函数获得主机对象，供后面的函数使用，上图中未画出
tusb_open_host	打开USB主机，将主机设置为就绪状态
tusb_close_host	关闭USB模块，将主机设置为关闭状态，上图中未画出
tusb_host_port_reset	复位端口
tusb_pipe_open	打开一个通道
tusb_pipe_close	关闭一个通道
tusb_pipe_setup	使用通道发送SETUP包
tusb_pipe_xfer_data	使用通道传输数据
tusb_pipe_wait	等待通道数据传输，可以设置超时时间，如果超时时间为0，则查看通道状态后立即返回
回调函数	**说明**
tusb_host_port_changed	当端口状态改变时调用此回调函数
tusb_on_channel_event	当通道状态改变时调用此回调函数

8 USB 主机开发

主机的基本数据传输很简单，主机的主要难度在协议栈的开发和对各种设备的兼容性处理。下面将用两个示例来说明主机端的主要开发方式。这些示例都只实现了设备的部分功能，只能够演示主机是如何工作的，不能在实际项目中使用。对设备的兼容性也很差，可能只能在一些特定的设备上能工作。代码中也没有太多的容错机制，主机协议栈为了保证较好的兼容性，需要做大量的容错处理。

8.1 无协议栈的主机

将前面的自定义 USB 设备作为从机，然后通过主机直接进行控制。无协议栈的主机工作流程非常简单，先用控制通道获取设备描述符，然后通过 OUT 通道发送数据，再用 IN 通道接收设备返回的数据。完整的代码在 demp/host_raw 目录中。

主函数处理代码如下：

```
int main(void){
  uint8_t speed;
  tusb_host_t* host = tusb_get_host(TEST_APP_USB_CORE);
  HOST_PORT_POWER_ON();
  tusb_open_host(host);
  while(1){
    if(state == TUSB_HOST_PORT_CONNECTED){
      state = TUSB_HOST_PORT_DUMMY;
      // reset port0
      tusb_port_set_reset(host, 0, 1);
      tusb_delay_ms(100);
      // release port0
      tusb_port_set_reset(host, 0, 0);
      tusb_delay_ms(100);
      speed = tusb_port_get_speed(host, 0);
      // Open Control pipe
      tusb_pipe_open(host, &pipe_ctrl_in, 0, 0x80, EP_TYPE_CTRL, EP_MPS, speed);
      tusb_pipe_open(host, &pipe_ctrl_out, 0, 0x00, EP_TYPE_CTRL, EP_MPS, speed);

      // Get device descriptor
      tusb_pipe_setup(&pipe_ctrl_out, &setup);
      tusb_pipe_wait(&pipe_ctrl_out, 0xffffffff);

      // Prepare buffer to recv device descriptor
      tusb_pipe_xfer_data(&pipe_ctrl_in, device_descriptor, sizeof(device_descriptor));
      tusb_pipe_wait(&pipe_ctrl_in, 0xffffffff);

      // Send status packet
```

```
      tusb_pipe_xfer_data(&pipe_ctrl_out, 0, 0);
      tusb_pipe_wait(&pipe_ctrl_out, 0xffffffff);

      // Send test data
      tusb_pipe_open(host, &pipe_write, 0, WRITE_EP, EP_TYPE_BULK, EP_MPS, speed);
      tusb_pipe_xfer_data(&pipe_write, test_data, sizeof(test_data));
      tusb_pipe_wait(&pipe_write, 0xffffffff);

      // Recv test data
      tusb_pipe_open(host, &pipe_read, 0, READ_EP, EP_TYPE_BULK, EP_MPS, speed);
      tusb_pipe_xfer_data(&pipe_read, buf, sizeof(buf));
      tusb_pipe_wait(&pipe_read, 0xffffffff);

    }else if(state == TUSB_HOST_PORT_DISCONNECTED){
      state = TUSB_HOST_PORT_DUMMY;
      // Close all pipe when device disconnected
      tusb_pipe_close(&pipe_ctrl_in);
      tusb_pipe_close(&pipe_ctrl_out);
      tusb_pipe_close(&pipe_write);
      tusb_pipe_close(&pipe_read);
    }
  }
}
```

当设备连接成功后，对设备进行复位，正式做法应该等待端口成为 Enabled 状态，这里只是简单等待了一下。然后打开一个地址为 0，端点分别为 0x80 和 0x00 的控制通道。这两个通道对应的是设备的控制端点，由于设备复位后地址为 0，所以使用了地址 0。

然后通过 OUT 通道发出一个获取设备描述符的 SETUP 包，这是一个读数据控制传输。因此后面跟着一个 IN 传输，用来接收设备描述符。根据读数据控制传输的处理流程，接收成功后，再通过 OUT 传输发送一个包长为 0 的状态包。

接下来打开一个 BULK OUT 通道，端点号为 WRITE_EP，向其写入测试数据。然后再打开一个 BULK IN 通道，端点号为 READ_EP，向其读回测试数据。至此完成测试。

这样的主机虽然看起来简单，但确实是可以和设备进行通讯的。这里是对设备请求处理和数据处理的一个演示，实际的设备会比这个复杂很多。标准的流程如下：

1. 需要设置设备地址，这里没有做，因为我们始终使用 0 地址通讯。
2. 读取配置，得到端点使用信息。这里没有做，因为设备是我们自己开发的，使用的什么端点参数是什么我们都知道，所以没有读配置描述符也能使用。
3. 配置设备，这里也没有做。因为我们的设备在复位的时候会将所有端点一起初始化，所以不配置也能使用。
4. 类型相关请求处理，这里也没有，如果是别的设备，需要进行类型相关请求处理。
5. 操作设备，正常情况下设备如果有 IN 端点，主机应该根据端点描述符的内容，为设备开启一个通道，然后保持这个通道。这里我们接收到数据后没有将 IN 通道再次使能，正常的主机应当保持 IN 通道有效。在传输完成后再次使能通道。

8.2 支持键盘、鼠标与 HUB 的主机

本节将简单介绍键盘、鼠标以及 HUB 工作方式。完整的代码在 demo/host_input 目录中。本例程实现的功能是当键盘的 Caps Lock 键按下时，更新 Caps Lock 灯的状态；当 Num Lock 按下时，更新 Num Lock 灯的状态。如果键盘和鼠标通过 HUB 一起接在开发板上，除了键盘可以控制键盘上的 LED 灯外，鼠标也可以控制键盘 LED 灯的状态。当鼠标左键点下时，更新 Num Lock 的状态；当鼠标右键点下时，更新 Caps Lock 状态。

8.2.1 通用设备的枚举过程

设备的枚举过程如下表：

表31　　设备枚举过程

设备上电
设备复位
获取设备描述符前8字节
设备复位（可选操作）
设置设备地址
获取设备描述符及其他描述符
配置设备
配置接口（可选操作）
配置设备特性（可选操作）
根据设备接口类型设置设备（接口类型相关）
设备正常工作

关于设备枚举过程的详细说明见 2.2 USB 的基本操作。在示例的 host_loop 函数中，检测端口状态，当端口为 Enabled 状态时，调用 enum_device 函数开始设备枚举，代码如下：

```
void host_loop(tusb_host_t* host){
  if( host->state == TUSB_HOST_PORT_CONNECTED){
    // reset port
    tusb_host_port_reset(host, 0, 1);
    delay_ms(50);
    tusb_host_port_reset(host, 0, 0);
    delay_ms(100);
  }else if( host->state ==  TUSB_HOST_PORT_ENABLED ){
    if(!root){
      root = new_device();
      root->is_root = 1;
      if(enum_device(host, root) != 0){
        free_device(root);
        root = 0;
      }
    }
  }else if( host->state == TUSB_HOST_PORT_DISCONNECTED ){
    if(root){
```

```
      free_device(root);
      root = 0;
    }
  }
}
```

在 enum_device 函数中根据上面表格中的流程枚举设备，部分代码如下：

```
int enum_device(tusb_host_t* host, usb_device_t* device){
  static uint8_t init = 0;
  static uint8_t addr = 1;
  uint32_t retry = 0;
  device->host = host;
  if(!init){
    if( tusb_pipe_open(host, &def_ctrl.ctrl_out, 0, 0x00, EP_TYPE_CTRL, 8) == 0
     && tusb_pipe_open(host, &def_ctrl.ctrl_in, 0, 0x80, EP_TYPE_CTRL, 8) == 0){
       init = 1;
     }
  }
  retry = GetDeviceDesc(&def_ctrl, device->device_desc, 8) == 0;
  device->addr = addr;
  addr++;
  if(SetAddress(&def_ctrl, device->addr) != 0){
    return -1;
  }
  if( tusb_pipe_open(host, &device->ctrl_pipe.ctrl_in, device->addr, 0x80, EP_TYPE_CTRL,
device->device_desc[7]) == 0
   && tusb_pipe_open(host, &device->ctrl_pipe.ctrl_out, device->addr, 0x00, EP_TYPE_CTRL,
device->device_desc[7]) == 0){
  }else{
    // fail to open control pipe
    goto dev_fail;
  }
  // Get config descriptor header
  if(GetConfigDesc(&device->ctrl_pipe, device->config_desc, 9) != 0){
    goto dev_fail;
  }
  device->config_len = *(uint16_t*)(device->config_desc+2);
  // get total config descriptor
  retry = GetConfigDesc(&device->ctrl_pipe, device->config_desc, device->config_len) == 0;
  // Set config to 0
  if(SetConfig(&device->ctrl_pipe) != 0){
    goto dev_fail;
  }
  if(init_interface(device) != 0){
    goto dev_fail;
```

```
  }
  return 0;
}
```

先使用默认地址上的通道获取设备描述符前 8 字节和设置地址，设置地址成功后，在设备上打开两个新的控制管道，后续的枚举过程用新的控制通道来完成。

然后获取配置描述符，设置配置，成功后进入接口类型相关的处理函数中，类型相关处理代码如下：

```
int init_interface(usb_device_t* device){
  USBH_CfgDescTypeDef* cfg = (USBH_CfgDescTypeDef*)device->config_desc;
  if(cfg->bNumInterfaces>0){
    if(cfg->Itf_Desc[0].bInterfaceClass == USB_HUB_CLASS){
      return init_hub(device);
    }else if(cfg->Itf_Desc[0].bInterfaceClass == USB_HID_CLASS){
      return init_hid(device);
    }else{
      // un-supported class
      goto itf_fail;
    }
  }
  return 0;
itf_fail:
  return -1;
}
```

这个示例中只支持了 HID 和 HUB 两种类型的设备。

8.2.2 键盘与鼠标

键盘与鼠标都属于 HID 设备，HID 设备的数据格式是由报告描述符来描述的，这里不打算对报告描述符进行解析和处理。因为键盘和鼠标有一个特殊的协议叫做 Boot Protocol，这个协议就是为了让键盘鼠标能够在资源十分有限的主机上运行，比如电脑的 BIOS 系统中。在这种情况下解析报告描述符是很浪费的，因此 USB 组织设计了键盘鼠标的 Boot 协议，在此协议下报告描述符是固定的，不用解析。

Boot 协议下，键盘的报告描述符：

```
    0x05, 0x01,     // USAGE_PAGE (Generic Desktop)
    0x09, 0x06,     // USAGE (Keyboard)
    0xa1, 0x01,     // COLLECTION (Application)
    0x05, 0x07,     //   USAGE_PAGE (Keyboard)
    0x19, 0xe0,     //   USAGE_MINIMUM (Keyboard LeftControl)
    0x29, 0xe7,     //   USAGE_MAXIMUM (Keyboard Right GUI)
    0x15, 0x00,     //   LOGICAL_MINIMUM (0)
    0x25, 0x01,     //   LOGICAL_MAXIMUM (1)
    0x75, 0x01,     //   REPORT_SIZE (1)
    0x95, 0x08,     //   REPORT_COUNT (8)
    0x81, 0x02,     //   INPUT (Data,Var,Abs)
```

```
0x95, 0x01,     //   REPORT_COUNT (1)
0x75, 0x08,     //   REPORT_SIZE (8)
0x81, 0x03,     //   INPUT (Cnst,Var,Abs)
0x95, 0x05,     //   REPORT_COUNT (5)
0x75, 0x01,     //   REPORT_SIZE (1)
0x05, 0x08,     //   USAGE_PAGE (LEDs)
0x19, 0x01,     //   USAGE_MINIMUM (Num Lock)
0x29, 0x05,     //   USAGE_MAXIMUM (Kana)
0x91, 0x02,     //   OUTPUT (Data,Var,Abs)
0x95, 0x01,     //   REPORT_COUNT (1)
0x75, 0x03,     //   REPORT_SIZE (3)
0x91, 0x03,     //   OUTPUT (Cnst,Var,Abs)
0x95, 0x06,     //   REPORT_COUNT (6)
0x75, 0x08,     //   REPORT_SIZE (8)
0x15, 0x00,     //   LOGICAL_MINIMUM (0)
0x25, 0x65,     //   LOGICAL_MAXIMUM (101)
0x05, 0x07,     //   USAGE_PAGE (Keyboard)
0x19, 0x00,     //   USAGE_MINIMUM (Reserved (no event indicated))
0x29, 0x65,     //   USAGE_MAXIMUM (Keyboard Application)
0x81, 0x00,     //   INPUT (Data,Ary,Abs)
0xc0            // END_COLLECTION
```

根据上面的描述符，键盘的输入数据格式是，第 1 字节低 6 位是修饰键，逐位表示，第 2 字节是占位用的。第 3 至第 8 字节是按键码，每一个字节代表一个键。这里还有一个输出数据，只有一个字节，低 5 位是键盘 LED 灯的状态，高 3 位是占位用的。

BOOT 协议下，鼠标的报告描述符：

```
0x05, 0x01,     // USAGE_PAGE (Generic Desktop)
0x09, 0x02,     // USAGE (Mouse)
0xa1, 0x01,     // COLLECTION (Application)
0x09, 0x01,     //   USAGE (Pointer)
0xa1, 0x00,     //   COLLECTION (Physical)
0x05, 0x09,     //     USAGE_PAGE (Button)
0x19, 0x01,     //     USAGE_MINIMUM (Button 1)
0x29, 0x03,     //     USAGE_MAXIMUM (Button 3)
0x15, 0x00,     //     LOGICAL_MINIMUM (0)
0x25, 0x01,     //     LOGICAL_MAXIMUM (1)
0x95, 0x03,     //     REPORT_COUNT (3)
0x75, 0x01,     //     REPORT_SIZE (1)
0x81, 0x02,     //     INPUT (Data,Var,Abs)
0x95, 0x01,     //     REPORT_COUNT (1)
0x75, 0x05,     //     REPORT_SIZE (5)
0x81, 0x03,     //     INPUT (Cnst,Var,Abs)
0x05, 0x01,     //     USAGE_PAGE (Generic Desktop)
0x09, 0x30,     //     USAGE (X)
0x09, 0x31,     //     USAGE (Y)
```

```
        0x15, 0x81,     //     LOGICAL_MINIMUM (-127)
        0x25, 0x7f,     //     LOGICAL_MAXIMUM (127)
        0x75, 0x08,     //     REPORT_SIZE (8)
        0x95, 0x02,     //     REPORT_COUNT (2)
        0x81, 0x06,     //     INPUT (Data,Var,Rel)
        0xc0,           //   END_COLLECTION
        0xc0            // END_COLLECTION
```

根据上面的描述符，鼠标的数据格式：第 1 字节低 3 位是按键，高 5 位占位用；第 2 字节是 x 偏移量；第 3 字节是 y 偏移量。

对于键盘和鼠标，都有一个 IN 端点用来向主机发送数据，键盘还有一个 Out 数据通过 HID 设备的 Set Output Report 请求来实现，HID 设备类初始化处理代码如下：

```
int init_hid(usb_device_t* device){
  USBH_CfgDescTypeDef* cfg = (USBH_CfgDescTypeDef*)device->config_desc;
  device->itf.handle = 0;
  if(cfg->Itf_Desc[0].bInterfaceProtocol == HID_KEYBRD_BOOT_CODE){
    device->itf.handle = kbd_handle;
  }else if(cfg->Itf_Desc[0].bInterfaceProtocol == HID_MOUSE_BOOT_CODE){
    device->itf.handle = mouse_handle;
    device->itf.in_len = 4; // boot mouse report length is
  }
  // TODO: parse the interfce descriptor to get the endpoint descritptor
  memcpy(cfg->Itf_Desc[0].Ep_Desc, (uint8_t*)cfg->Itf_Desc[0].Ep_Desc + 9, 14);
  if(device->itf.handle){
    if(open_ep(device) != 0){
      goto hid_fail;
    }
    device->itf.deinit = close_ep;
    if(SetBootProtocol(&device->ctrl_pipe) != 0){
      goto hid_fail;
    }
    tusb_pipe_xfer_data(&device->itf.data_in, device->itf.in_buf, device->itf.in_len);
  }
  return 0;
hid_fail:
  return -1;
}
```

根据协议设置处理函数，解析描述符，得到端点描述符。这里没有解析，直接将端点描述符复制到了前面，跳过了 HID 描述符。正式的做法是解析描述符，然后打开设备端点。这里没有考虑设备有多接口的情况，也没有考虑端点数量大于 2 的情况。打开端点成功后，发送 SetBootProtocol 请求，让设备工作在 Boot 协议下。

键盘和鼠标的数据处理代码也很简单，代码如下：

```
void mouse_handle(usb_device_t* device){
  static __IO uint32_t tick = 0;
  channel_state_t state = tusb_pipe_wait(&device->itf.data_in, 0);
```

```
  if(state == TUSB_CS_TRANSFER_COMPLETE) {
    // get mouse data
    if(device->itf.in_buf[0] & 1){
      // left key down, toggle num lock
      kbd_led ^= LED_NUM_LOCK;
    }
    if(device->itf.in_buf[0] & 2){
      // right key down, toggle caps lock
      kbd_led ^= LED_CAPS_LOCK;
    }
    tusb_pipe_xfer_data(&device->itf.data_in, device->itf.in_buf, device->itf.in_len);
  }else if(state != TUSB_CS_XFER_ONGOING){
    tick++;
    if(tick>2000){
      tick = 0;
      tusb_pipe_xfer_data(&device->itf.data_in, device->itf.in_buf, device->itf.in_len);
    }
  }
}
void kbd_handle(usb_device_t* device){
  static __IO uint32_t tick = 0;
  channel_state_t state = tusb_pipe_wait(&device->itf.data_in, 0);
  if(state == TUSB_CS_TRANSFER_COMPLETE) {
    // get key board data
    if(device->itf.in_buf[2] == KEY_NUM_LOCK){
      // Num Lock
      kbd_led ^= LED_NUM_LOCK;
    }
    if(device->itf.in_buf[2] == KEY_CAPS_LOCK){
      // Caps lock
      kbd_led ^= LED_CAPS_LOCK;
    }
    tusb_pipe_xfer_data(&device->itf.data_in, device->itf.in_buf, device->itf.in_len);
  }else if(state != TUSB_CS_XFER_ONGOING){
    tick++;
    if(tick>2000){
      tick = 0;
      tusb_pipe_xfer_data(&device->itf.data_in, device->itf.in_buf, device->itf.in_len);
    }
  }
}
```

鼠标收到有效数据后，根据左右键设置按键 LED 状态。键盘收到有效数据后，根据按键数据中第一个按键值设置按键 LED 状态。键盘和鼠标都有一个 tick 计数，这里是让传输等一

段时间再发起。正式的做法应该是根据端点描述符中 bInterval 的值，确定传输发起的间隔。

8.2.3 HUB 处理

HUB 也是一种 USB 设备，有地址，能传输数据，有设备描述符，有配置描述符，还有 HUB 描述符。HUB 有一个中断 IN 端点，用来通知主机端口发生变化。主机根据 HUB 上报的数据，确定是哪个端口发生的变化，然后针对这个端口，进行操作。主机要通过 HUB 与设备通讯，流程如下：

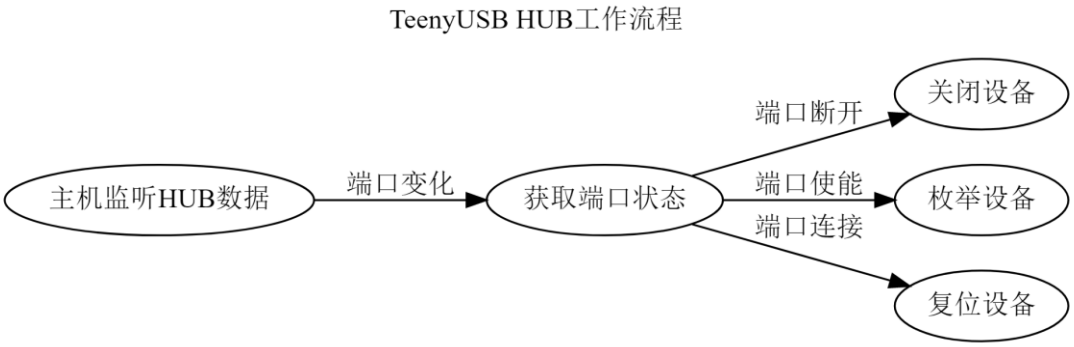


图80　　HUB 工作流程

当主机完成 HUB 枚举和设置后，监听 HUB 数据。当端口发生变化时，HUB 会向主机上报数据。主机根据数据内容确定哪个端口有变化。通过 GetStatus 请求获取到端口的当前状态。如果端口状态是 CONNECT，发送复位请求到指定端口。复位完成后，端口状态变成 ENABLE，这时开始枚举流程。有 HUB 和没有 HUB 的枚举流程是一样的，需要注意的是，HUB 会上报端口的速度信息，枚举和传输数据时，要使用设备与 HUB 连接时的速度。

设备枚举成功后，将设备挂在 HUB 设备的 Child 节点上。HUB 处理代码如下：

```
void hub_handle(usb_device_t* device){
  channel_state_t state = tusb_pipe_wait(&device->itf.data_in, 0);
  if(state == TUSB_CS_TRANSFER_COMPLETE){
    uint8_t port_state = device->itf.in_buf[0];
    uint8_t port = 0;
    while(port_state){
      // some port changed
      if(port_state & 1){
        // handle port change here
        if (GetHUBPortStatus(&device->ctrl_pipe, port, &port_status, sizeof(port_status)) == 0){
          // Get port status success
          if(port_status.wPortStatus.PORT_POWER) {
            // port powered
            uint8_t r = 0;
            if(port_status.wPortChange.C_PORT_CONNECTION){
              r += ClearHUBPort(&device->ctrl_pipe, port, HUB_FEATURE_SEL_C_PORT_CONNECTION);
            }
            if(port_status.wPortChange.C_PORT_ENABLE){
              r += ClearHUBPort(&device->ctrl_pipe, port, HUB_FEATURE_SEL_C_PORT_ENABLE);
```

```
          }
          if(port_status.wPortChange.C_PORT_SUSPEND){
            r += ClearHUBPort(&device->ctrl_pipe, port, HUB_FEATURE_SEL_C_PORT_SUSPEND);
          }
          if(port_status.wPortChange.C_PORT_OVER_CURRENT){
            r += ClearHUBPort(&device->ctrl_pipe, port, HUB_FEATURE_SEL_C_PORT_OVER_CURRENT);
          }
          if(port_status.wPortChange.C_PORT_RESET){
            r += ClearHUBPort(&device->ctrl_pipe, port, HUB_FEATURE_SEL_C_PORT_RESET);
          }
          if(port_status.wPortStatus.PORT_CONNECTION){
            if(port_status.wPortStatus.PORT_ENABLE){
              // Enable means reset done
              usb_device_t* dev = new_device();
              if(dev){
                dev->is_root = 0;
                dev->is_low_speed = port_status.wPortStatus.PORT_LOW_SPEED;
                if(enum_device(device->host, dev) == 0){
                  usb_device_t* child = device->children[port-1];
                  if(child){
                    free_device(child);
                  }
                  device->children[port-1] = dev;
                }else{
                  free_device(dev);
                }
              }
            }else{
              // not enable, need a reset
              r += SetHUBPort(&device->ctrl_pipe, port, HUB_FEATURE_SEL_PORT_RESET);
            }
          }else{
            usb_device_t* child = device->children[port-1];
            if(child){
              free_device(child);
            }
            device->children[port-1] = 0;
          }
        }
      }
    }
    port++;
    port_state >>= 1;
  }
```

```
  }else if(state == TUSB_CS_STALL){
    // TODO: clear interface feature
  }
  if(state != TUSB_CS_XFER_ONGOING){
    tusb_pipe_xfer_data(&device->itf.data_in, device->itf.in_buf, device->itf.in_len);
  }
  for(int i=0;i<MAX_CHILD;i++){
    usb_device_t* child = device->children[i];
    if(child && child->itf.handle){
      child->itf.handle(child);
    }
  }
}
```

运行此例程，在没有 HUB 的时候可以直接连接键盘或鼠标，有 HUB 时，可以通过 HUB 连接多个键盘和鼠标，理论上 HUB 下也可以再连接 HUB。不过这个示例代码里面没有对描述符做解析，一些数据的读取频率可能与设备要求不符合，设备多了之后会有兼容性问题。

8.2.4 键盘鼠标的数据处理

此例程在鼠标的处理函数中检测鼠标的左右按键情况，如果按下时更新 kbd_led 的值。在键盘的处理函数中检测按键的 Caps Lock 和 NumLock 是否按下，如果按下更新 kbd_led 的值。在主循环中，如果检测到 kbd_led 的值有变化，调用 set_kbd_led 函数，对所有键盘设备的 LED 灯进行设置。set_kbd_led 函数的代码如下：

```
void set_kbd_led(usb_device_t* cur_dev){
  if(cur_dev){
    USBH_CfgDescTypeDef* cfg = (USBH_CfgDescTypeDef*)cur_dev->config_desc;
    if( cfg->Itf_Desc[0].bInterfaceClass == USB_HID_CLASS
     && cfg->Itf_Desc[0].bInterfaceProtocol == HID_KEYBRD_BOOT_CODE ){
        //SetBootProtocol(&cur_dev->ctrl_pipe);
        SetReport(&cur_dev->ctrl_pipe, &last_led, 1);
      }
    for(int i=0;i<MAX_CHILD;i++){
      set_kbd_led(cur_dev->children[i]);
    }
  }
}
```

示例代码每个设备都有独自的控制通道，这个也比较浪费。实际使用中可以按需申请通道，用完后就释放，节约主机通道资源。对于主机而言，IN 通道需要监听来自设备的数据，需要一直有效，而 OUT 通道需要数据时才会使用，可以在需要的时候再申请。更灵活的做法是做一个通道池，需要时从“池子”里申请，用完后立刻还到池中。

9 相关资源

9.1 STM32 芯片

9.1.1 STM32 型号与 USB 模块对应关系

STM32 的 USB 模块分为 FS 和 OTG 两大类。FS 模块只能作为 USB 从机使用，配置在 M0 内核和一般的 M3 内核芯片中。OTG 模块既能做主机又能做从机，OTG 又进一步分为 OTG_FS 和 OTG_HS 两大类。OTG_FS 和_OTG_HS 操作基本类似，OTG_HS 支持 DMA 而 OTG_FS 不支持。下表为不同的芯片系列所支持的 USB 模块版本。其中的 STM32F723 系列以及 STM32F730Z8 和 STM32F730I8 内置了高速 USB PHY。

表32　　STM32 芯片 USB 模块

STM32芯片系列	内核	USB_FS	OTG_FS	OTG_HS
STM32F07x/04x	M0	是	否	否
STM32F103	M3	是	否	否
STM32F105/107	M3	否	是	否
STM32F2x5/2x7	M3	否	是	是
STM32F3xx	M4	是	否	否
STM32F4x1/4x2/4x3	M4	否	是	否
STM32F4x5/4x6/4x7/4x9	M4	否	是	是
STM32F7xx	M7	否	是	是
STM32H7xx	M7	否	是	是

表33　　不同 USB 模块的区别

	USB_FS	OTG_FS	OTG_HS
交互方式	专用内存	FIFO	FIFO
缓存大小	512-1024	1280字节	4096字节
USB从设备	支持	支持	支持
USB主设备	不支持	支持	支持
PHY	内置	内置	内置/外置
端点数	8个	4-6个	6-9个
DMA	不支持	不支持	支持

9.2 PC 端 USB 配套软件

USB 设备开发需要 PC 端配套软件进行测试，本文中用到的测试软件由 Qt 开发，USB 通讯部分采用的是 libusb 库。USB 驱动部分采用 libusb 的通用驱动，inf 文件由描述符工具 TeenyDT 自动生成，驱动签名基于 libwdi 项目中的自签名代码实现。

9.2.1 Qt 库

Qt 是一个开源的跨平台界面库，很多商用软件采用 Qt 开发。Qt 库使用 C++开发，通过预处理器 moc 对 C++特性做了扩展。信号和槽机制是 Qt 的一大亮点，简化了对象间的通讯逻辑。

Qt 项目地址：https://www.qt.io

9.2.2 XToolbox 应用程序

XToolbox 是一个基于 Qt 的 lua 开源应用程序框架，使用 lua 脚本构造 Qt 界面元素。用 lua 封装了一些常用的接口，能够快速的进行原型程序开发。XToolbox 采用的 Qt 版本是 Qt4.8.5，lua 版本是 5.3.5。XToolbox 中封装的 libusb 版本是 1.0.22，并在此基础上增加上了对 WinUSB 同步传输的支持。

由于 Qt 和 lua 对对象生命周期管理方式的不同，在导出到 lua 时没有对 Qt 对象内存管理进行分析，全部不在 lua 中管理，这样做会导致内容泄露。例如调用 takeXXX 风格的函数后，对象生命周期不再受 Qt 管理，这个时候 lua 也没有管理，产生泄露。因此 XToolbox 可以做一些简单的测试工具，可以用来快速构造一些界面原型，不适合作为应用程序开发工具。

XToolbox 项目地址：xtoolbox.org

9.2.3 libusb 库

libusb 是一个开源的跨平台 USB 库，对 USB 设备在不同的操作系统上进行了封装，提供了统一的接口。libusb 主要操作流程为枚举设备，打开设备，读取设备描述符，获取设备接口，获取接口中的端点信息，然后以端点为基本单位对设备进行读写。这点与 USB 的管道（Pipe）概念类似，一个端点就是一个通讯管道。

libusb 的代码地址：https://github.com/libusb/libusb

最新发布版的 libusb 库没有包含 WinUSB 中的同步传输功能，在 libusb 的 pull request 中有关于同步传输的代码。XToolbox 将包含 WinUSB 同步传输的代码添加到了 libusb 的最新发布版中，代码地址：https://github.com/xtoolbox/libusb。选择 r1.0.22_with_winusb_iso 这个分支，包含了 WinUSB 同步传输功能的代码。

9.2.4 libwdi 驱动安装项目

libwdi 是一个 Windows 上的驱动安装工程，提供了多种驱动安装功能。在较新的 Windows 系统，如 Win10，驱动文件必须提供签名文件才能安装。libwdi 提供了自签名功能，可以为生成的驱动进行签名。

Libwdi 的代码地址：https://github.com/pbatard/libwdi

TeenyDT 的驱动文件签名功能采用 libwdi 的代码来实现，带签名工具 libwdi 项目在这里：https://github.com/xtoolbox/libwdi

9.2.5 WinUSB 库

WinUSB 是 Windows 上的一个通用 USB 设备驱动库，提供了基本的设备操作接口。libusb 对 WinUSB 的接口进行了封装，可以使用 WinUSB 的驱动作为 libusb 的后端（backend）。

9.2.6 QLibUsb 库

QLibUsb 库是笔者用 Qt 风格对 libusb 库做的进一步封装，根据 TeenyUSB 的一些特点做了一些简化处理。QLibUsb 源码地址在这里：http://qlibusb.tusb.org。

9.3 USB 文档资源

USB 官方文档及工具下载地址：
https://www.usb.org/documents

9.3.1 本文提到的参考资料

1. 《Universal Serial Bus Specification 2.0》即文中的《USB 2.0 规格书》
2. STM32 系列芯片的《xxx Reference Manual》 即文中的《STM32 参考手册》