应用笔记

Application Note

文档编号：**AN1087**

APM32F4xx_OTG 应用笔记

版本：**V1.0**

1 引言

当前 APM32F4xx 系列拥有高速和全速两个 USB 控制器，都支持 OTG，其中的 USB 高速控制器拥有两个接口。此应用笔记基于 APM32F4xx_OTG_SDK 进行示例讲解，该软件开发包可以从极海官网的 APM32F4xx 软件支持处下载得到。

目录

2 USB OTG 简介

APM32F4xx 中的 OTG 符合 USB2.0 规范，且符合 On-The-Go 补充标准。在常规的 USB 中，从机和主机身份相对固定，数据传输的控制都由主机处理和发起。而 OTG 则可以让器件在从机和主机中转换，既可以当主机，也可以变换为从机。OTG 常应用于设备与设备之间的交互，例如打印机和相机之间，手机和 U 盘之间都可以利用 OTG 进行数据交互。

2.1 OTG 主机和从机角色切换的原理

在 USB 的接线的 ID 线可以用来区分主机和从机。ID 线检测为低电平表示主机，ID 线检测为高电平则为从机。MCU 的 ID 线内接上拉电阻，当外部的 ID 线插入时接地，则会检测为低电平，识别为主机；当外部的 ID 线悬空插入，则会检测为高电平，识别为从机。

2.1.1 主机协商协议（HNP）

HNP 全称为 Host Negotiation Protocol。这是主机和从机之间用来进行角色互换的协议。APM32F407 可以通过拉高 OTG 外设的全局寄存器 GUSBCFG 的 HNPEN 位开启该功能。

2.1.2 会议请求协议（SRP）

SRP 全称为 Session Request Protocol。此功能允许 A-device 在总线未使用时停止 Vbus 供电以降低功耗。APM32F407 可以通过拉高 OTG 外设的全局寄存器 GUSBCFG 的 SRPEN 位开启该功能

2.2 APM32F407 的 USB 从机特性

对于目前的 APM32F4 系列，APM32F407、F405、417、415 系列的 USB OTG 模块都一样，但未来也可能出现 F4 系列 USB 模块变动的情况，所以这里就将 F407 当作示例。

APM32F407 作从机时，可用的 IN 端点有 4 个，OUT 端点有 4 个，合计 8 个端点。端点 0 比较特殊，仅作为控制端点。作为从机时支持全速和高速，其中，高速控制器拥有两个接口，其中有个内嵌的高速 PHY，可以减少外围设备的设计。

2.3 APM32F407 的 USB 主机特性

APM32F407 作主机时，可用的主机通道有 8 个，每个通道在使用的时候需要配置对应的端点编号，传输类型等信息。作为主机时支持低速、全速和高速，其中高速控制器拥有两个接口。

2.4 初学者常见的疑惑解答

2.4.1 USB 和 OTG 之间的关系？

答：USB 中包含 OTG，OTG 可以视为 USB 中的一种特殊类型，专门用来设计便携式设备，实现主机和从机之间的角色切换。OTG 也能当作正常的 USB 来使用，即忽略 ID 线的作用，强制 MCU 为主机或从机即可。

2.4.2 A-device、B-device 和标准从机的区别？

答：A device 为 OTG 主机，和标准的主机相比，A device 的 Vbus 可以提供更小的电流以降低功耗，它可以切换为从机。B device 为 OTG 从机，用法和标准从机基本一致，只不过可以通过控制器使其能够切换为主机。标准从机就是我们通常情况下解除的 USB 从机，它只能当从机，且 ID 线无效。

2.4.3 Vbus 有什么作用？

答：Vbus 即总线的电压，由 USB 的主机向从机提供。从机可以分为自供电设备和总线供电设备，由 USB 的配置描述符告知主机。从机上的 Vbus 线主要靠接收，在 APM32F4 的从机控制器中有 Vbus 检测的功能，也就是 Vbus 引脚拉高之后就判断 USB 插入。从机也可以选择不需要 Vbus，关闭 Vbus 检测，在 APM32F4 的从机中会默认 Vbus 有效，且能够使用内部的上拉电阻，并通过软件控制上拉电阻的有效与否。

3 USB 的从机使用示例

我们在极海官网的 APM32F4xx 软件支持中可以找到 APM32F4xx_OTG_SDK 并下载，此 SDK 包专门为 APM32F4 系列的 USB 提供了驱动代码和相关的主机与从机例程。在 SDK 包中的 Project\Device_Examples 文件夹中包含了三个例程，其中 HID 表示鼠标例程，使用的是 HID 类；MSC 表示 U 盘例程，使用的是 MSC 大容量类；VCP 表示虚拟串口例程，使用的是 CDC 传输类。

3.1 初始化 USB 从机

我们可以通过 USBD_Init 函数对 USB 初始化，其需要配置的结构体参数如下：

```
typedef struct
{
    USBD_Descriptor_T *pDeviceDesc;              //!< 设备描述符
    USBD_Descriptor_T *pConfigurationDesc;       //!< 配置描述符
    USBD_Descriptor_T *pStringDesc;              //!< 字符串描述符
    USBD_Descriptor_T *pQualifierDesc;           //!< 设备限定描述符
    USBD_Descriptor_T *pHidReportDesc;           //!< 报告描述符
    USBD_StdReqCallback_T *pStdReqCallback;   //!< 标准请求回调函数集合
    USBD_ReqHandler_T stdReqExceptionHandler; //!< 标准请求出错回调函数
    USBD_ReqHandler_T classReqHandler;           //!< class 请求回调函数
    USBD_ReqHandler_T vendorReqHandler;          //!< 厂商请求回调函数
    USBD_CtrlTxStatusHandler_T txStatusHandler;//!< 控制过程 IN 状态回调函数
    USBD_CtrlRxStatusHandler_T rxStatusHandler; //!< 控制过程 OUT 状态回调函数
    USBD_EPHandler_T        outEpHandler;     //!< 端点 0 以外的 OUT 端点回调函数
    USBD_EPHandler_T        inEpHandler;      //!< 端点 0 以外的 IN 端点回调函数
    USBD_ResetHandler_T    resetHandler;      //!< USB 复位回调函数
    USBD_InterruptHandler_T intHandler;       //!< 中断补充处理回调函数
} USBD_InitParam_T;
```

图 1 USBD_InitParam_T 结构体

结构体成员的含义可以参考上述代码中的注释部分。USB_Init 函数的作用是传递形参中的回调函数，并且初始化 USB 的硬件、全局寄存器、设备寄存器和 USB 中断等内容。

用户可以通过配置 usb_config.h 文件的全局宏定义来配置全局中断、端点中断、FIFO 大小等内容。关于定义的宏，下列内容作了简要解释：

- **USB_INT_G_SOURCE**：全局中断源配置；
- **USB_INT_EP_OUT_SOURCE**：OUT 端点中断源配置；
- **USB_INT_EP_IN_SOURCE**：IN 端点中断源配置；
- **USB_VBUS_SWITCH**：为 1 时，初始化时会开启 VBUS 感应，为 0 关闭；
- **USB_SOF_OUTPUT_SWITCH**：为 1 时，会开启 SOF 输出，为 0 关闭；
- **USBD_CONFIGURATION_NUM**：描述符配置数，一般为 1；
- **USB_EP0_PACKET_SIZE**：端点 0 最大包的大小；
- **USB_FS_RX_FIFO_SIZE**：全速接收 FIFO 的大小配置
- **USB_FS_TX_FIFO_0_SIZE**：全速发送 FIFO 0 的大小配置
- **USB_FS_TX_FIFO_1_SIZE**：全速发送 FIFO 1 的大小配置
- **USB_HS_RX_FIFO_SIZE**：高速接收 FIFO 的大小配置
- **USB_HS_TX_FIFO_0_SIZE**：高速发送 FIFO 0 的大小配置
- **USB_HS_TX_FIFO_1_SIZE**：高速发送 FIFO 1 的大小配置
- **DELAY_SOURCE**：当其值为 USE_DEFAULT 时，使用默认的延时函数，值为 USE_USER 时，用户需要在 usb_user.c 文件中自定义延时函数。

3.2 中断处理说明

USB 作为从机时，其通信的过程通常使用中断来进行处理。从机的中断处理过程可参考下图：

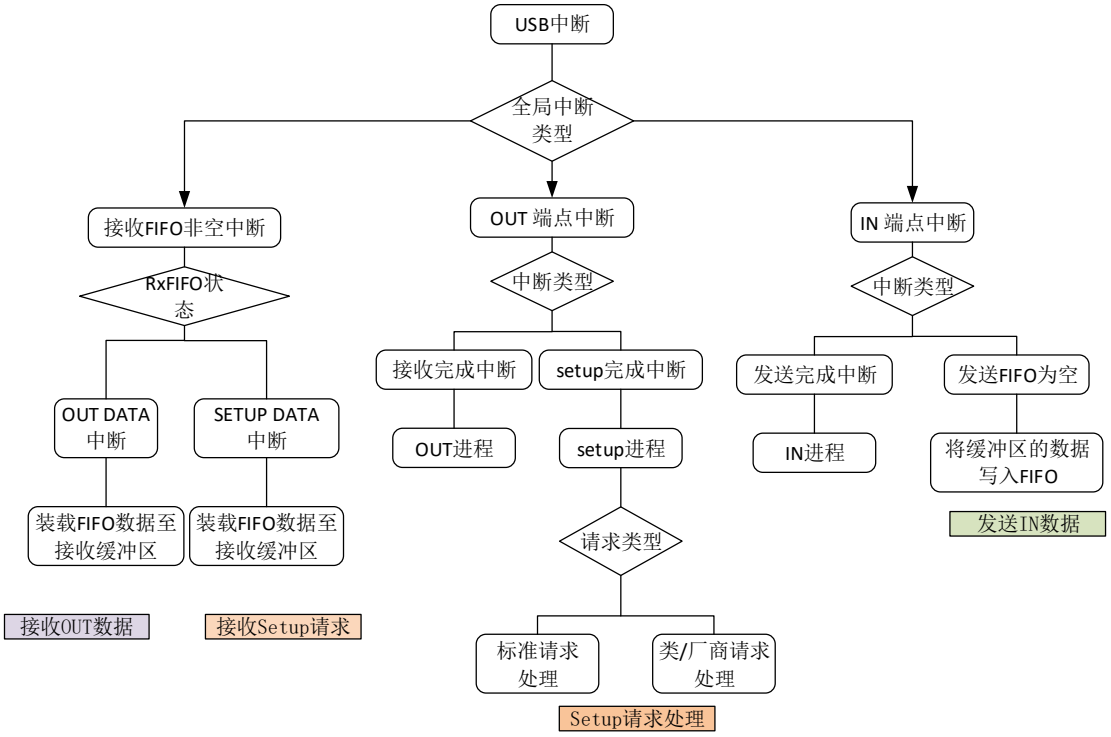

图 2 程序流程图

实际上，USB 从机模式下的枚举和通信基本都依靠中断，即使在轮询中调用了收发 USB 数据的函数，最终也是会通过中断处理数据传输。

3.3 SETUP 事务处理

① 启动 Setup 包传输

调用 drv_usb_device.h 文件中的 USB_OTG_ReceiveSetupPacket 函数来启动 Setup 包的传输。

② 在 RxFIFO 非空中断接收 Setup 包

USB 接收到数据时触发 RxFIFO 非空中断， USBD_RxFifoNoEmptyIsrHandler 函数（usbd_interrupt.c 文件中）会进行处理。当 RxFIFO 出栈状态指示该包为 Setup 包时，保存至全局变量 g_usbDev.reqData.pack 中。

③ 处理 Setup 包

处理 Setup 包由 usbd_core.c 中的 USBD_SetupProcess 函数执行。当判断出接收到的 Setup 包为标准请求，则由 usbd_stdReq.c 中的 USBD_StandardReqeust 函数进行处理；类请求由全局变量的回调函数 g_usbDev.classReqHandler 处理；厂商请求由全局变量的回调函数 g_usbDev.vendorReqHandler 处理。

上述的两个回调函数是在调用 USBD_Init 函数初始化时对其赋值，用户在初始化时将函数名作函数指针赋值即可。

3.4 IN 事务处理

3.4.1 控制端点 IN 事务

① 启动 IN 数据传输

调用 usbd_core.c 中的 USBD_CtrlInData 函数来启动 IN 事务并配置 IN 数据发送缓冲区。对于控制传输，一次启动最多传输端点 0 的最大包长，若需要传输的数据长度大于最大包长，则拆分成多次传输。

需要发送设备状态时，即发送 0 长度 IN 包，可使用 usbd_core.c 中的 USBD_CtrlTxStatus 函数来发送设备的状态阶段。

若 IN 事务中有多个 IN 数据包需要发送时，则除第一个包外的 IN 数据由 drv_usb_device.c 中的 USB_OTG_EnableInEpTransfer 来使能传输。

USBD_CtrlInData 与 USB_OTG_EnableInEpTransfer 函数的区别在于前者配置了发送缓冲区，发送时会使用该缓冲区发送数据，后者在缓冲区配置好的情况下，直接操作寄存器使能传输，即前者包含后者

② 在 TxFIFO 空中断发送 IN 数据包

OTG SDK 使用中断的方式来压栈 TxFIFO，当 TxFIFO 达到空的阈值则会触发 TxFIFO 空中断，阈值由 GAHBCFG 寄存器中的 TXFEL 位决定。在中断里，调用 usbd_core.c 中的 USBD_PushDataToTxFIFO 函数来将缓冲区数据写入 TxFIFO。

③ IN 发送完成

IN 发送完成后，会调用 usbd_core.c 中的 USBD_CtrlInProcess 函数进行处理，判断是否继续发送数据或者需要接收 OUT 状态。

3.4.2 其他端点 IN 事务

① 启动 IN 数据传输

调用 usbd_core.c 中的 USBD_RxData 函数来启动 OUT 事务。

② 在 TxFIFO 空中断发送 IN 数据包

在 TxFIFO 空中断，调用 USBD_PushDataToTxFIFO 函数来将缓冲区数据写入 TxFIFO。

③ IN 发送完成

IN 发送完成后，由全局变量的回调函数 g_usbDev.inEpHandler 进行处理，该函数在初始化时赋值。

3.5 用户发送数据示例

这里的“发送”指的是从机向主机发送数据，在事务上属于 IN 事务。端点 0 发送数据调用的是 USBD_CtrlInData 函数，其他端点发送数据可以调用 USBD_TxData 函数。

在我们调用发送函数之后，不管我们是在中断还是轮询，这只是“预”发送，真正的发送时刻在 USB 的发送 FIFO 空中断触发之后。我们调用发送函数时输入的数据指针和长度信息将会传递到 g_usbDev. inBuf[ep]中，其中 ep 表示不同的端点。我们如果需要判断需要发送的数据是否成功地执行，我们可以去判断 g_usbDev. inBuf[ep]. bufLen 的时，因为在每次写入数据至发送 FIFO 之后，g_usbDev. inBuf[ep]. bufLen 会随之递减，直到为 0 表示所有数据发送完成。

```
uint8_t data[4] = {1, 2, 3, 4};


/** 端点 0 默认进行的是控制传输，这里演示采用端点 0 发送数据 */
USBD_CtrlInData(data, 4);


/** 这里演示采用端点 1 发送数据，不论中断、批量还是同步端点都适用此函数 */
USBD_TxData(USB_EP_1, data, 4);
```

图 3 发送数据的代码示例

3.6 用户接收数据示例

这里的“接收”指的是从机向主机接收数据，在事务上属于 OUT 事务。端点 0 发送数据调用的是 USBD_CtrlOutData 函数，其他端点发送数据可以调用 USBD_RxData 函数。

在实际的 USB 从机应用中，我们能够接收什么数据是由主机决定的，而接收函数只能启动一次接收，这时候当数据来临时，我们能够将数据准确地保存。接收数据常用于 Class 请求的处理，当我们收到主机的请求后，我们就能得知主机将要发送什么数据或者需要接收什么数据，这时候可以调用接收数据的函数来启动一次接收。

```
uint8_t data[4] = {0, 0, 0, 0};


/** 端点 0 默认进行的是控制传输，这里演示采用端点 0 接收数据 */
USBD_CtrlOutData(data, 4);


/** 这里演示采用端点 1 发送数据，不论中断、批量还是同步端点都适用此函数 */
USBD_RxData(USB_EP_1, data, 4);
```

图 4 接收数据的代码示例

4 USB 的主机使用示例

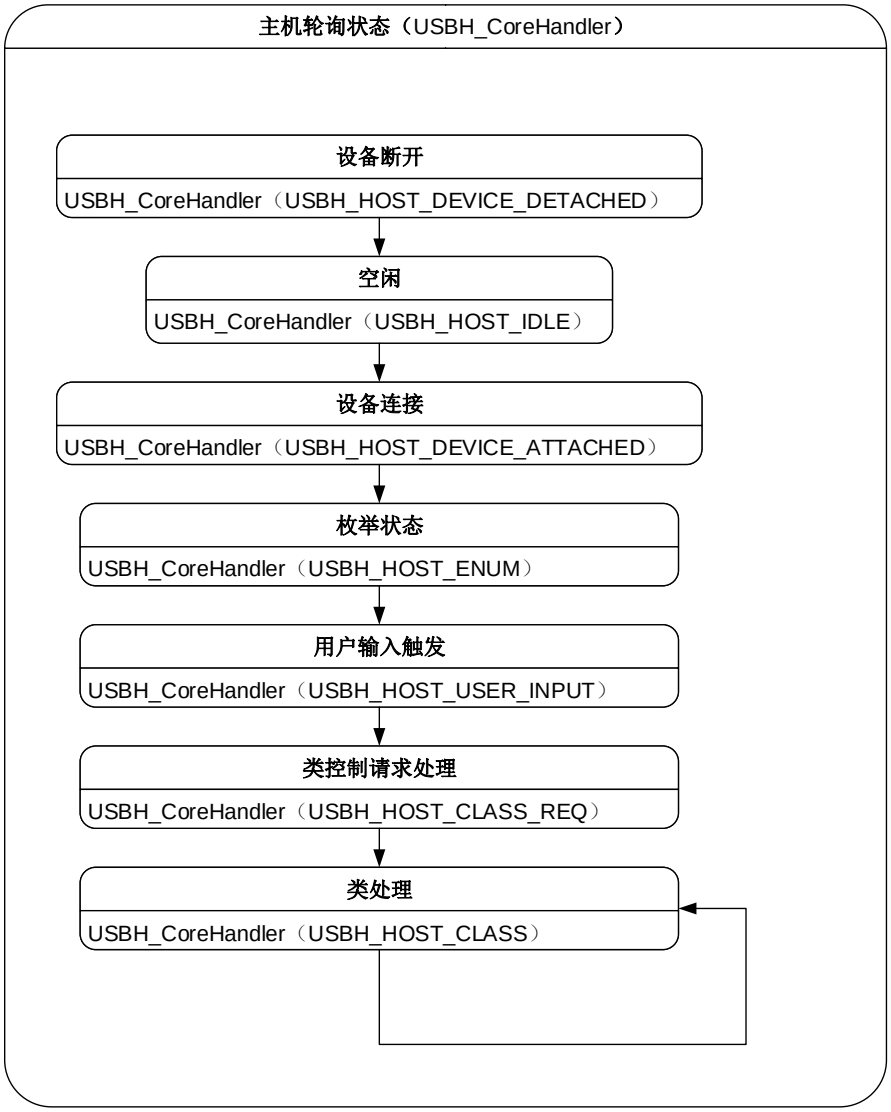

图 5 主机轮询状态机

主机轮询状态可以理解为状态机，需要一直被轮询。主机的状态机处理我们可以理解为是一个回调函数数组，通过访问不同的数组，达到不同的状态。我们可以用 USBH_ConfigHostState 函数来切换不同的主机状态。上图中的状态机中，程序刚运行时一般是设备断开的状态，若有设备插入，则会进入空闲状态，空闲状态判断是否已经插入设备，插入设备后将执行设备连接状态。前三个状态相对固定，用户不需要做额外的操作。

4.1 枚举状态

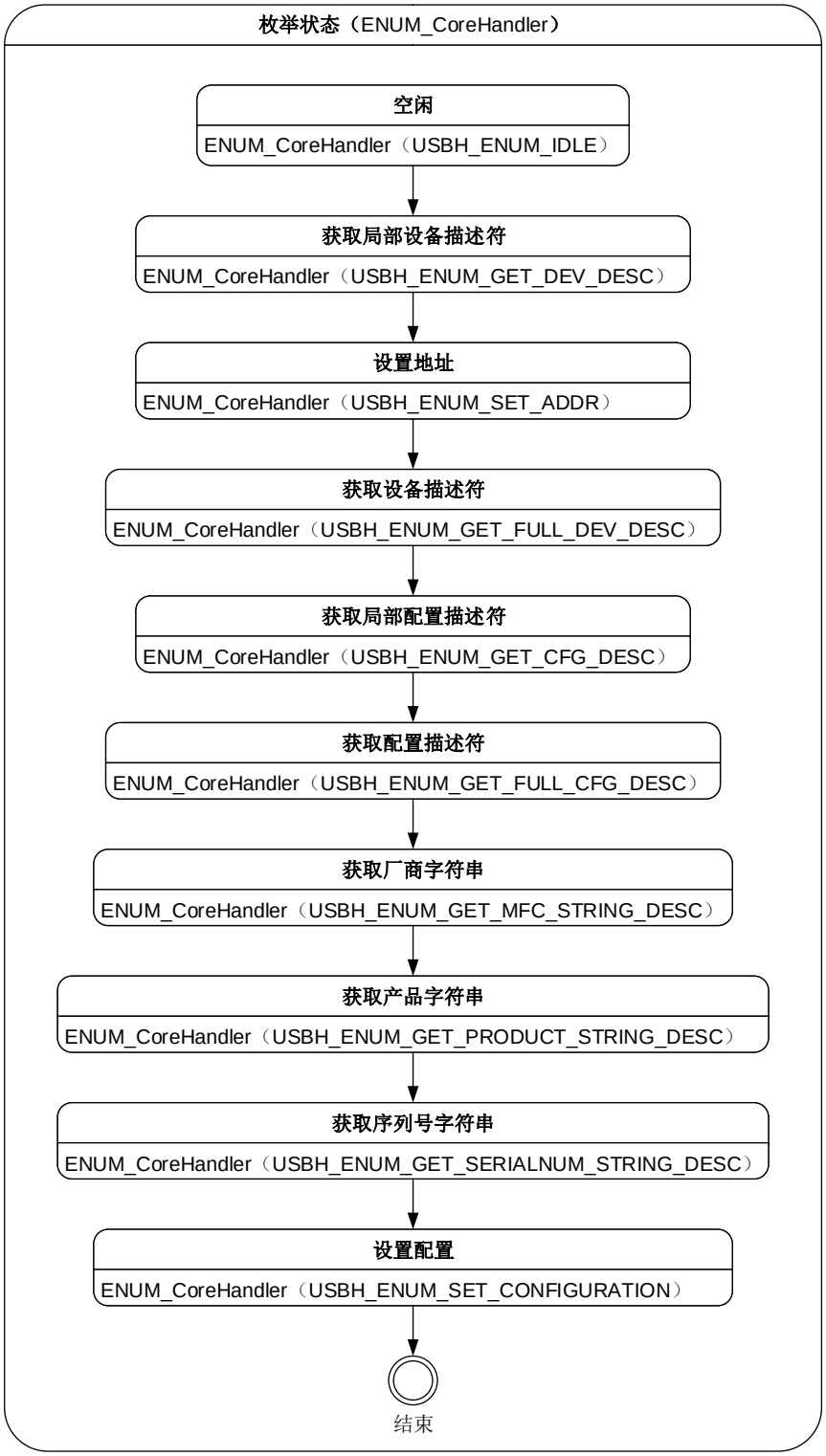

图 6 枚举状态

枚举状态是上电后识别设备的重要状态，实际上，枚举阶段只要达到设置地址、获得设备描述符及获得配置描述符三步就可以实现设备枚举。实际应用中，我们可能还需要查看设备的字符

串描述符，驱动中只会默认获取厂商字符串、产品字符串和序列号字符串。

4.2 用户输入状态

```
static void USBH_UserInputHandler(void)
{
    if (g_userCallback.userInputHandler() == USER_OK)
    {
        g_usbHost.classInitHandler();
    }
}
```

图 7 用户输入状态处理

上图的函数是用户输入状态的处理函数，处于 usbd_core.c 文件。我们可以看到，当用户回调函数 g_userCallback.userInputHandler 的返回值为 USER_OK 时，将会执行 class 初始化函数即 g_usbHost.classInitHandler。

g_userCallback.userInputHandler 回调函数由用户定义，例如可以在按键被按下时，返回 USER_OK

class 初始化函数是在 USB 主机初始化（USBH_Init）的时候作为输入参数将其配置完成。通常情况下，建议用户在 class 初始化函数中，将主机状态改变为 USBH_HOST_CLASS_REQ 或者 USBH_HOST_CLASS 状态。

4.3 Class 状态

关于 Class 状态可以分为两个部分，一部分是采用端点 0 通信的控制请求，另一部分是端点 0 以外的其他端点传输。例如 HID 类的获取报告描述符就输入控制请求，而进行中断传输的是另一种 class 处理。

USBH_Init 初始化函数中的 classReqHandler 结构体成员是 class 请求处理的回调函数，而另一个成员 classCoreHandler 是 class 处理的主要回调函数。用户应当在初始化阶段构建合适的 class 回调函数。

4.4 Suspend、Wake-up 和 Error 状态

这三个状态分别是挂起状态，唤醒状态和错误状态。错误状态会在一些出错情景下触发，而挂起和唤醒状态需要用户在特点的场合去切换，并在调用 USBH_Init 初始化函数的时候构建好对应的回调函数。

4.5 用户接口说明

在 usb_user.c 文件中构建了一个 g_userCallback 全局结构体变量，其结构体成员都是回调函数。此全局变量可以由用户根据需要而进行修改。

```
typedef struct
{
   USER_InitHandler_T                  initHandler;
   USER_DeInitHandler_T                deInitHandler;
   USER_ResetDevHandler_T              resetDevHandler;
   USER_DevAttachedHandler_T           devAttachedHandler;
   USER_DevDetachedHandler_T           devDetachedHandler;
   USER_DevSpeedDetectedHandler_T      devSpeedDetectedHandler;
   USER_DevDescHandler_T               devDescHandler;
   USER_CfgDescHandler_T               cfgDescHandler;
   USER_ManufacturerStringHandler_T    manufacturerStringHandler;
   USER_ProductStringHandler_T         productStringHandler;
   USER_SerialNumStringHandler_T       serialNumStringHandler;
   USER_EnumDoneHandler_T              enumDoneHandler;
   USER_UserInputHandler_T             userInputHandler;
   USER_ApplicationHandler_T           applicationHandler;
   USER_DeviceNotSupportedHandler_T deviceNotSupportedHandler;
   USER_UnrecoveredErrHandler_T        unrecoveredErrHandler;
   USER_DelayCallBack_T                delay;
} USB_UserCallBack_T;
```

关于回调函数结构体的成员：

initHandler 将在 USBH_Init 函数执行的时候触发，用户可以进行初始化操作；

deInitHandler 在 USB 设备断开或者发送错误的时候触发，用户可以进行某些恢复操作；

resetDevHandler 在设备连接时复位端口之后触发，用户可以进行复位操作；

devAttachedHandler 在设备连接之后触发，用户可以进行连接之后的某些操作；

devDetachedHandler 在设备断开状态中触发，用户可以进行设备断开的处理；

devSpeedDetectedHandler 在设备连接时，检测出设备的速度之后触发，关于设备的速度，即 g_usbHost.speed，值为 0 时为高速，1 代表全速，2 代表低速；

devDescHandler 在获得设备描述符之后触发，用户可以在这里提取设备描述符；

cfgDescHandler 在获得配置描述符之后触发，用户可以在这里提取配置描述符；

manufacturerStringHandler 在获得厂商字符串之后触发，用户可以在这里提取厂商字符串；

productStringHandler 在获得产品字符串之后触发，用户可以在这里提取产品字符串；

serialNumStringHandler 在获得序列号字符串之后触发，用户可以在这里提取序列号字符串；

enumDoneHandler 在设备枚举完成之后触发，用户可以在这里进行特点的操作；

userInputHandler 在主机轮询状态机中触发，其返回值为 USER_OK 时，会执行初始化时配置的 classInitHandler 回调函数，意味着处理枚举阶段之后的 class 阶段；

applicationHandler 实际上由用户去构造和安放，通常可以放在应用层空闲时轮询阶段；

deviceNotSupportedHandler 属于错误状态的一种，当主机不支持此类设备时触发；

unrecoveredErrHandler 属于错误状态的一种，表面此类错误为未知错误；

delay 是 USB 延时函数，用户可以使用定时器延时或者变量计数延时等。

5 版本历史

表格 1 文件版本历史

日期	版本	变更历史
2022.06.23	1.0	新建

声明

本手册由珠海极海半导体有限公司（以下简称“极海”）制订并发布，所列内容均受商标、著作权、软件著作权相关法律法规保护，极海保留随时更正、修改本手册的权利。使用极海产品前请仔细阅读本手册，一旦使用产品则表明您（以下称“用户”）已知悉并接受本手册的所有内容。用户必须按照相关法律法规和本手册的要求使用极海产品。

1、权利所有

本手册仅应当被用于与极海所提供的对应型号的芯片产品、软件产品搭配使用，未经极海许可，任何单位或个人均不得以任何理由或方式对本手册的全部或部分内容进行复制、抄录、修改、编辑或传播。

本手册中所列带有“®”或“TM”的“极海”或“Geehy”字样或图形均为极海的商标，其他在极海产品上显示的产品或服务名称均为其各自所有者的财产。

2、无知识产权许可

极海拥有本手册所涉及的全部权利、所有权及知识产权。

极海不应因销售、分发极海产品及本手册而被视为将任何知识产权的许可或权利明示或默示地授予用户。

如果本手册中涉及任何第三方的产品、服务或知识产权，不应被视为极海授权用户使用前述第三方产品、服务或知识产权，除非在极海销售订单或销售合同中另有约定。

3、版本更新

用户在下单购买极海产品时可获取相应产品的最新版的手册。

如果本手册中所述的内容与极海产品不一致的，应以极海销售订单或销售合同中的约定为准。

4、信息可靠性

本手册相关数据经极海实验室或合作的第三方测试机构批量测试获得，但本手册相关数据难免会出现校正笔误或因测试环境差异所导致的误差，因此用户应当理解，极海对本手册中可能出现的该等错误无需承担任何责任。本手册相关数据仅用于指导用户作为性能参数参照，不构成极海对任何产品性能方面的保证。

用户应根据自身需求选择合适的极海产品，并对极海产品的应用适用性进行有效验证和测试，以确认极海产品满足用户自身的需求、相应标准、安全或其它可靠性要求；若因用户未充分对极海产品进行有效验证和测试而致使用户损失的，极海不承担任何责任。

5、合规要求

用户在使用本手册及所搭配的极海产品时，应遵守当地所适用的所有法律法规。用户应了解产品可能受到产品供应商、极海、极海经销商及用户所在地等各国有关出口、再出口或其它法律的限制，用户（代表其本身、子公司及关联企业）应同意并保证遵守所有关于取得极海产品及／或技术与直接产品的出口和再出口适用法律与法规。

6、免责声明

本手册由极海“按原样”（as is）提供，在适用法律所允许的范围内，极海不提供任何形式的明示或暗示担保，包括但不限于对产品适销性和特定用途适用性的担保。

对于用户后续在针对极海产品进行设计、使用的过程中所引起的任何纠纷，极海概不承担责任。

7、责任限制

在任何情况下，除非适用法律要求或书面同意，否则极海和/或以“按原样”形式提供本手册的任何第三方均不承担损害赔偿责任，包括任何一般、特殊因使用或无法使用本手册相关信息而产生的直接、间接或附带损害（包括但不限于数据丢失或数据不准确，或用户或第三方遭受的损失）。

8、适用范围

本手册的信息用以取代本手册所有早期版本所提供的信息。