UM1720
用户手册

STM32Cube™ USB 主机库

前言

本文档描述 STM32Cube™ 的 USB 主机库中间件模块。

众所周知，通用串行总线 （USB）是介于作为主机的个人计算机 （PC）与所连接的 USB 外设之间的事实通信标准。目前嵌入式便携设备数量日益增多，USB 主机已不再局限于 PC，嵌入式消费终端与工业设备作为主机已越来越普遍，而嵌入式主机功能有限，仅支持特定的 USB 类 （比如大容量存储类、虚拟通信端口等）或特定的供应商设备。

本文档详细描述了 STM32Cube™ USB 主机库，即在 STM32CubeF2 和 STM32CubeF4 软件包中所提供的 USB 主机协议栈。它使用 STM32 微控制器作为 USB 嵌入式主机，与各种 USB 类的 USB 外设进行通信 （MSC、 HID、 CDC、音频和 MTP）。

STM32Cube™ 软件包是一款免费解决方案，可以从意法半导体网站下载：
http://www.st.com/stm32cube。

目录

表格索引

图片索引

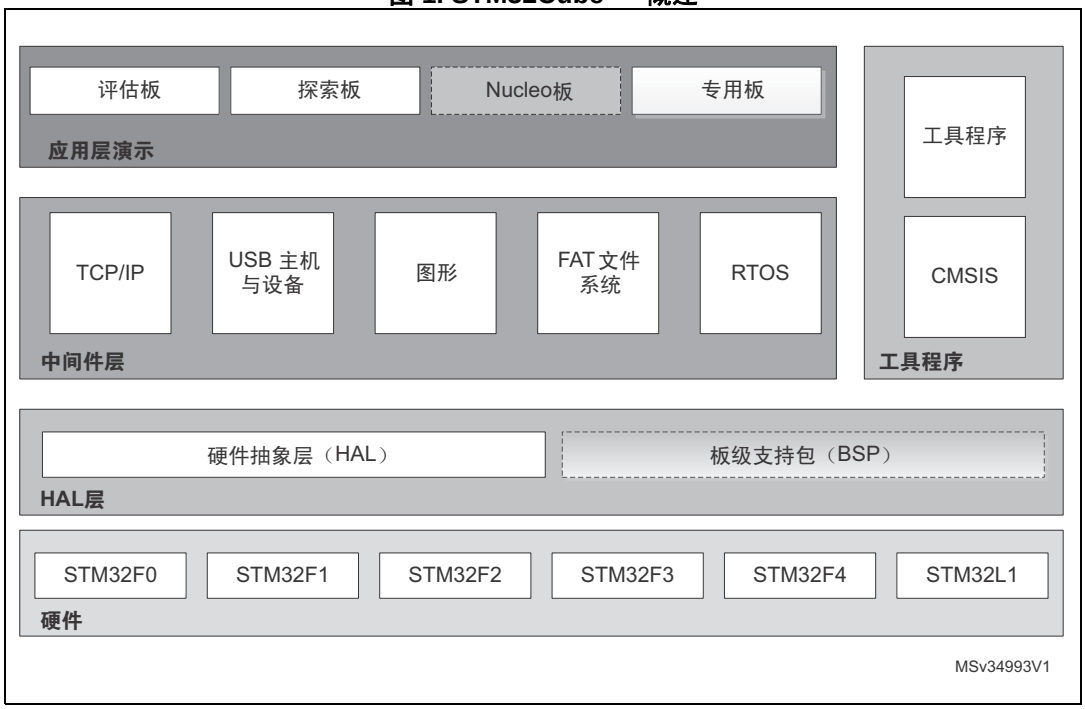
1 STM32Cube™ 概述

STMCube™ 计划源自意法半导体，旨在通过减少开发的工作量、时间与成本，使开发者受益。 STM32Cube™ 将涵盖 STM32 全线产品组合。

STM32Cube™1.x 版包括：

- 图形软件配置工具 STM32CubeMX，可通过图形向导生成初始化 C 代码。
- 针对每个系列提供综合的嵌入式软件平台，（比如用于 STM32F4 系列的 STM32CubeF4）
  - STM32 抽象层嵌入式软件 STM32Cube HAL，确保用户应用在 STM32 各个产品之间实现最大限度的可移植性。
  - 一套中间件组件，比如 RTOS、 USB、 TCP/IP、图形处理。
  - 一整套例程用于演示所有嵌入式软件功能。

图 1. STM32Cube™ 概述



2 简介

本文档描述 STM32Cube 的 USB 的主机库中间件模块。

USB 主机库处于 STM32Cube USB 主机 HAL 驱动之上。该库提供用于访问各类 USB 设备的 API。

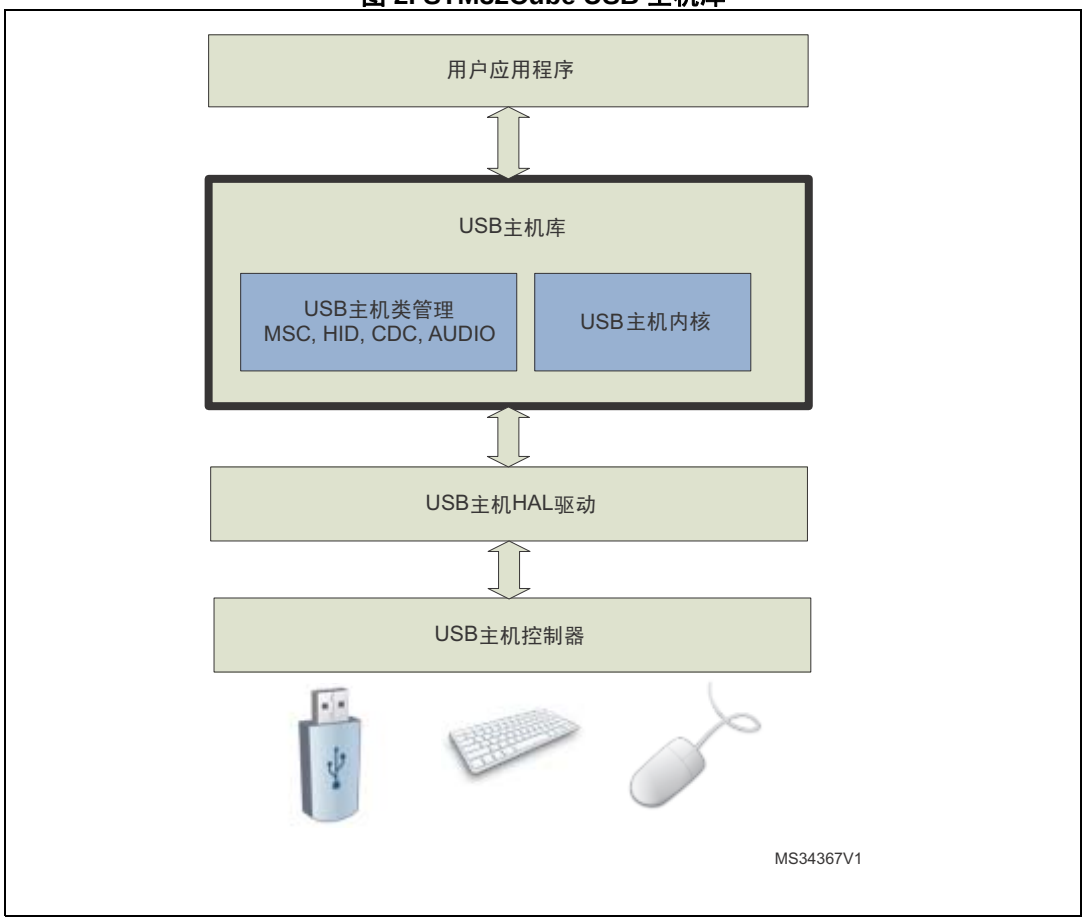
USB 主机模块可用于实现主要的 USB 类。

- 大容量存储类 （MSC）
- 人机接口鼠标与键盘类 （HID）
- 通信设备类 （CDC）
- 音频类 （AUDIO）
- 媒体传输协议类 （MTP）

除了以上所列的类以外，用户还能通过库提供的 API 来实现自己的应用类别。

该库可单独运行也可以运行在 RTOS 模式之下。该库还支持多实例，比如可以在两个或更多的 USB 主机模块上工作。

图 2. STM32Cube USB 主机库

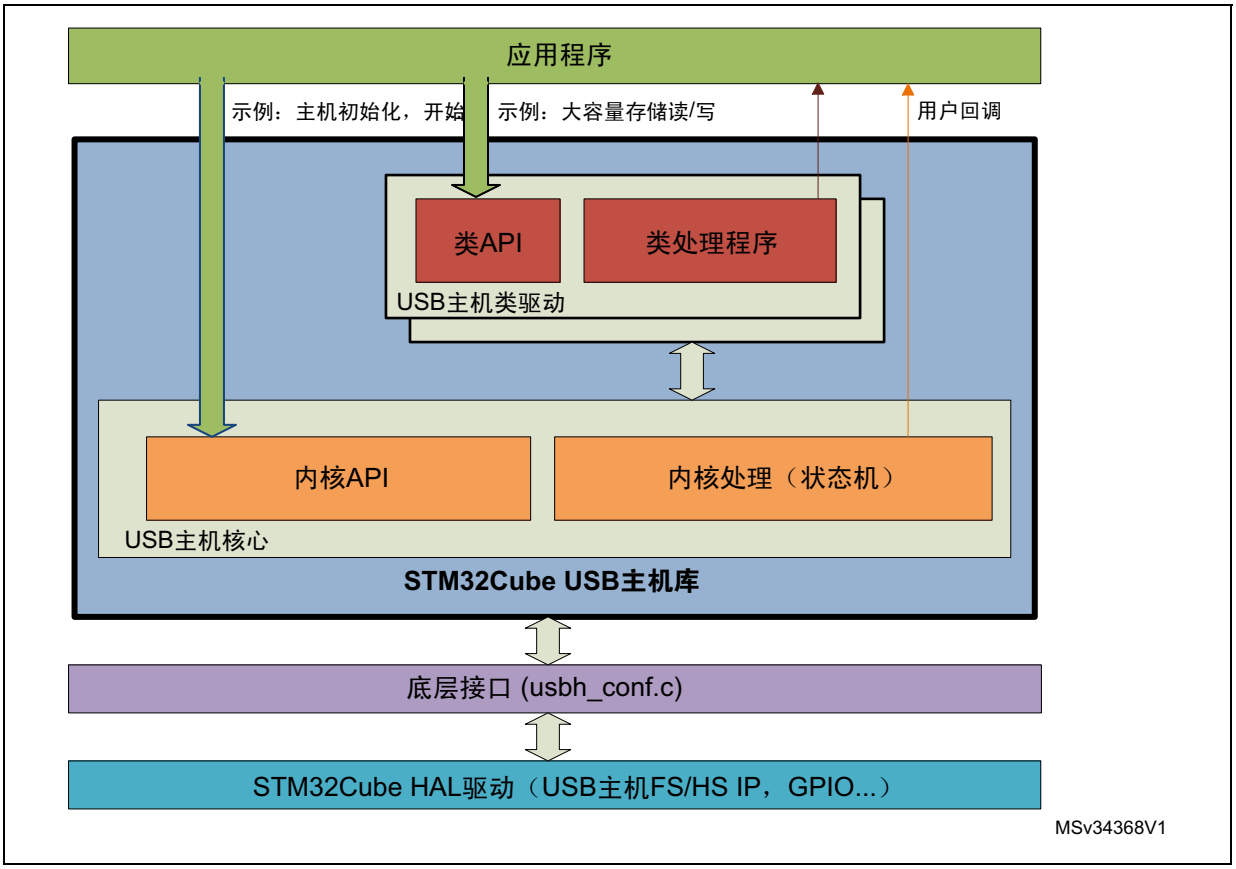


3 USB 主机库架构与文件组织

3.1 USB 主机库架构

图 3 显示了库架构的概览。

图 3. USB 主机库架构



如 *图 3* 所示， USB 主机库分为两个主要部分：内核部分和类相关的驱动部分。

主机内核用于处理库的内核服务，这些服务主要用于：

- 设备的连接 / 断连管理与标准枚举。
- 主机管道控制与 USB 传输管理 （控制、中断、批量、同步）。

内核数据块被细分为两个主要子数据块：

- 被用户应用或类驱动所调用的内核 API。
- 处理主机状态机的内核过程。

类驱动用于实现各种 USB 类。

这些类驱动包括：

- 一套类专用的 API （比如 disk_read、 disk_write），可在应用层调用。
- 一个类处理程序，由内核主机状态机 （内核过程）所调用，用于处理类操作 （类的初始化、解除初始化、类过程）。

主机内核和类驱动在经历某些已定义事件 （比如设备的连接 / 断连，数据接收）后，会调用用户回调函数。

USB 主机库利用接口层与 STM32Cube HAL 驱动相连接。

库中提供了实现底层接口层的模板文件。用户可以定制文件中与板支持包要求有关的内容部分。

3.2 USB 主机库文件组织

3.2.1 USB 主机内核文件

USB 主机内核文件位于 *Core* 文件夹 （*STM32_USB_Host_Library\Core*）中。 *表 1* 列出了内核文件。

表 1. USB 主机内核文件

文件	说明
usbh_core.c/.h	主要的主机内核文件。 实现主机状态机 管理设备检测与枚举，控制类模块进行类操作。
usbh_ctlreq.c/.h	实现标准控制请求 （USB 第 9 章）
usbh_ioreq.c/.h	USB 传输管理的 API （控制、批量、中断、同步）。
usbh_pipes.c/.h	管道控制的 API （例如分配、开启、关闭）
usbh_conf_template.c/.h	底层接口文件的模板文件，由用户定制，并包含在应用文件中 （详情请参见 5.x 章节）
usbh_def.h	通用的库定义

3.2.2 USB 主机类文件

表 2 列出了位于 *Class* 文件夹 （*TM32_USB_Host_Library\Core*）中的类文件

表 2. 类驱动文件

USB 类	文件	说明
大容量存储	usbh_msc.c	大容量存储类处理程序
	usbh_msc_bot.c	专用批量传输 （BOT）协议
	usbh_msc_scsi.c	SCSI 指令
HID 鼠标与键盘	usbh_hid.c	HID 类处理程序
	usbh_hid_parser.c	HID 描述符解析
	usbh_hid_mouse.c	HID 鼠标子类处理程序
	usbh_hid_keybd.c	HID 键盘子类处理程序
	usbh_hid_usage.h	HID 的通用定义
音频扬声器	usbh_audio.c	音频类处理程序
CDC 虚拟通信端口	usbh_cdc.c	CDC 虚拟通信端口处理程序
MTP 类	usbh_mtp.c	MTP 类处理程序
	usbh_mtp_ptp	MTP 类的 PTP 规范实现

4 主机库内核模块

4.1 内核特性

该 USB 主机内核具备以下主要特性：

- 和具体类无关的设备连接管理与枚举
- 基于状态机的结构，可在后台的主循环中运行或作为 RTOS 任务线程
- 采用用户事件回调，将关于设备连接 / 断连、错误状态等主机事件通知应用层
- 可重定向至任何接口 （比如串行端口、 LCD）的用户事件记录
- 错误管理与报告

4.2 内核 API，用户回调与数据结构

如 *第 3.1 章节 : USB 主机库架构* 所详述，内核 API 可被应用程序及类驱动调用。

以下是关于各个内核 API 的内容。欲了解详细信息，请参考头文件中的 C 代码。

主要 API 的用途会进一步在*第 6 章*中详述。

4.2.1 应用层调用的内核 API

用户应用程序 API 列于*表 3* 中。

表 3. 应用的内核 API

函数	说明
USBH_Init	初始化主机栈和底层。驱动应在应用程序启动时调用
USBH_DeInit	清理主机栈变量，并运行底层清理 （比如关闭所有打开的管道，清除中断标志）
USBH_RegisterClass	注册所支持的 USB 类处理程序。经过枚举后，主机检查当前的设备类是否对应于注册的类。
USB_ReEnumerate	通过重新初始化主机栈并强制实施 VBUS 的断连 / 连接，对设备进行重新枚举。
USBH_Start	使能主机端口的 VBUS 电源，然后开始底层操作。
USBH_Stop	关闭主机端口的 VBUS 电源，然后停止底层操作。
USBH_GetActiveClass	设备枚举和类初始化之后，返回当前的活动 USB 类
USBH_Process	以单任务运作模式实现内核状态机的主机过程函数。该函数应在主循环的后台中调用，以处理主机状态机。

4.2.2 内核用户回调

主机内核通过调用用户回调函数，将 USB 事件传递给应用层。当调用 *USBH_Init* API 时，该函数会作为参数进行传递。

用户回调函数的原型应为：

void (*pUsrFunc)(USBH_HandleTypeDef * phost, uint8_t event)

内核用户回调事件作为 *event* 参数。*表 4* 列出了 主机内核的用户回调事件。

表 4. 内核用户回调事件

内核用户回调事件	说明
HOST_USER_CONNECT	将设备连接的有关信息通知应用程序
HOST_USER_DISCONNECT	将设备断连的有关信息通知应用程序
HOST_USER_CLASS_ACTIVE	将类初始化过程结束的有关信息通知应用程序
HOST_USER_SET_CONFIGURATION	将设备标准枚举结束的有关信息通知应用程序
HOST_USER_CLASS_SELECTED	通知已找到受支持的类

4.2.3 类处理程序的内核 API

专门用于类处理程序的内核 API 分为以下几类：

- I/O 请求 API
- 管道控制 API
- 标准类控制请求 API
- 接口功能 API

表 5 列出了供类处理程序调用的内核 API。

表 5. 类处理程序的专用内核 API

函数	类别	说明
USBH_BulkReceiveData	IO 请求	接收批量传输的数据
USBH_BulkSendData		发送批量传输的数据
USBH_CtlReceiveData		接收控制传输的数据
USBH_CtlSendData		发送控制传输的数据
USBH_CtlSendSetup		通过控制传输发送命令包
USBH_InterruptReceiveData		接收来自中断管道的数据
USBH_InterruptSendData		将数据发送至中断管道
USBH_IsocReceiveData		接收来自同步管道的数据
USBH_IsocSendData		将数据发送至同步管道
USBH_OpenPipe	管道控制	打开管道
USBH_ClosePipe		关闭管道
USBH_AllocPipe		分配新的管道
USBH_FreePipe		释放已分配的管道
USBH_GetDescriptor	标准控制请求	获取描述符的通用函数
USBH_SetInterface		标准控制请求，用于设置接口的复用设定值
USBH_FindInterface	接口功能	解析配置描述符，找寻与特定的类、子类和协议相对应的接口描述符
USBH_FindInterfaceIndex		解析配置描述符，根据指定的接口编号和复用设定值找寻接口描述符的索引序号

4.2.4 主要的主机内核数据结构体与枚举类型定义

内核的数据结构体和枚举类型定义在文件 *usbh_def.h* 中。主要数据结构体包括：

- *USBH_HandleTypedef* 类型的主机内核句柄结构体
- *USBH_DeviceTypedef* 类型的设备句柄结构体，
- *USBH_ClassTypedef* 类型的类句柄结构体。

以下小节将会详述主机内核句柄结构体与设备句柄结构体。类句柄结构体将在*第 5 章*的类章节中详述。

主机内核句柄结构体

主机库中所采用的主要结构体是 *USBH_HandleTypedef* 类型的主机句柄。

图 4. USBH_HandleTypedef

```
typedef struct _USBH_HandleTypeDef
{
  HOST_StateTypeDef     gState;
  ENUM_StateTypeDef     EnumState;
  CMD_StateTypeDef      RequestState;
  USBH_CtrlTypeDef      Control;
  USBH_DeviceTypeDef    device;
  USBH_ClassTypeDef*    pClass[USBH_MAX_NUM_SUPPORTED_CLASS];
  USBH_ClassTypeDef*    pActiveClass;
  uint32_t              ClassNumber;
  uint32_t              Pipes[15];
  __IO uint32_t         Timer;
  void*                 pData;
  void                 (* pUser )(struct _USBH_HandleTypeDef
*pHandle, uint8_t id);

#if (USBH_USE_OS == 1)
  osMessageQId          os_event;
  osMessageQId          class_ready_event;
  osThreadId            thread;
#endif
} USBH_HandleTypeDef;
```

表 6 详细描述了主机句柄结构体。

表 6. 主机句柄结构体

结构体成员	说明
gState	给出主机全局状态机的当前状态
EnumState	给出枚举状态机的当前状态
RequestState	给出控制请求的当前状态（IDLE、 SEND 或 WAIT）
Control	保存控制传输管理有关信息的结构体
Device	保存所连接设备有关信息的结构体

表 6. 主机句柄结构体 （续）

结构体成员	说明
pClass[USBH_MAX_NUM_SUPPORTED_CLASS]	指针数组，指针分别指向注册的类句柄结构体
pActiveClass	指向当前活动的类句柄结构体
ClassNumber	给出已注册类的总数
Pipes[15]	保存主机各个管道的状态 （已分配或已释放）。还表示与该管道通信的设备的端点号 （如果有的话）
Timer	用于时间管理的计数变量 每一帧开始时会自动增加
pData	以底层主机控制器数据结构体 （在 *usbh_conf.c 配置*文件中）进行初始化的指针，可将库与底层驱动相连接
(* pUser)(struct _USBH_HandleTypeDef *pHandle, uint8_t id);	主机用户事件回调函数

注： *图 4 中，USBH_USE_OS 定义下的成员与 RTOS 有关。它们用于保存 RTOS 消息队列事件的信息。*

主机内核设备结构体

主机内核设备结构体保存了关于所连接设备的信息。该结构体的类型为 *USBH_DeviceTypeDef*，其声明如*图 5* 所示。

图 5. 设备描述符

```
typedef struct {
#if (USBH_KEEP_CFG_DESCRIPTOR == 1)
  uint8_t
CfgDesc_Raw[USBH_MAX_SIZE_CONFIGURATION];
#endif
  uint8_t                          Data[USBH_MAX_DATA_BUFFER];
  uint8_t                           address;
  uint8_t                           speed;
  uint8_t                           is_connected;
  uint8_t                           current_interface;
  USBH_DevDescTypeDef               DevDesc;
  USBH_CfgDescTypeDef               CfgDesc;
}USBH_DeviceTypeDef;
```

表 7 列出了主机句柄结构体的成员：

表 7. 主机句柄结构体

结构体成员	说明
CfgDesc_Raw	设备的完整配置描述符
Data	用于接收描述符的数据缓冲
Address	设备地址
Speed	设备速度 （低速、全速、高速）
is_connected	设备的连接状态
current_interface	当前所选择的接口
DevDesc	含有设备描述符数据的结构体
CfgDesc	含有配置描述符数据的结构体 （仅指配置描述符的前 9 个字节）

主要枚举类型定义

USBH_StatusTypeDef

几乎所有库函数均会返回 *USBH_StatusTypeDef* 类型的状态。应用程序应必须检查库函数的返回状态。

```
typedef enum
{
  USBH_OK   = 0,
  USBH_BUSY,
  USBH_FAIL,
  USBH_NOT_SUPPORTED,
  USBH_UNRECOVERED_ERROR,
  USBH_ERROR_SPEED_UNKNOWN,
}USBH_StatusTypeDef;
```

表 8 描述了上述的返回状态。

表 8. USB 主机状态

状态	说明
USBH_OK	当操作顺利完成时的返回状态
USBH_BUSY	当操作仍未完成 （忙）时的返回状态
USBH_FAIL	当由于底层错误或协议失效而导致操作失败时的返回状态
USBH_NOT_SUPPORTED	不支持所请求的操作 / 功能时的返回状态
USBH_UNRECOVERED_ERROR	主机过程自身无法从错误中恢复时的返回状态
USBH_ERROR_SPEED_UNKNOWN	设备连接后，通知应用程序无法检测到设备速度时的返回状态

以下枚举类型定义供 USB 内核内部使用。然而为了调试目的，用户仍可通过主机句柄结构体来访问它们。

HOST_StateTypeDef

该枚举类型定义列出了可能出现的主机内核状态 （请参见*第 4.3 章节*所述的主机状态机）。

```
typedef enum
{
  HOST_IDLE =0,
  HOST_DEV_ATTACHED,
  HOST_DEV_DISCONNECTED,
  HOST_DETECT_DEVICE_SPEED,
  HOST_ENUMERATION,
  HOST_CLASS_REQUEST,
  HOST_INPUT,
  HOST_SET_CONFIGURATION,
  HOST_CHECK_CLASS,
  HOST_CLASS,
  HOST_SUSPENDED,
  HOST_ABORT_STATE,
}HOST_StateTypeDef;
```

ENUM_StateTypeDef

该枚举类型定义列出了设备枚举过程中的状态。

```
typedef enum
{
  ENUM_IDLE = 0,
  ENUM_GET_FULL_DEV_DESC, /* 获取设备描述符 */
  ENUM_SET_ADDR,          /* 设置设备地址 */
  ENUM_GET_CFG_DESC,      /* 获取配置描述符 （前 9 个字节） */
  ENUM_GET_FULL_CFG_DESC, /* 获取完整的配置描述符 */
  ENUM_GET_MFC_STRING_DESC, /* 获取制造商字符串描述符 */
  ENUM_GET_PRODUCT_STRING_DESC, /* 获取产品字符串描述符 */
  ENUM_GET_SERIALNUM_STRING_DESC, /* 获取序列号字符串描述符 */
} ENUM_StateTypeDef;
```

CTRL_StateTypeDef

该枚举类型定义列出了控制传输过程中的状态。

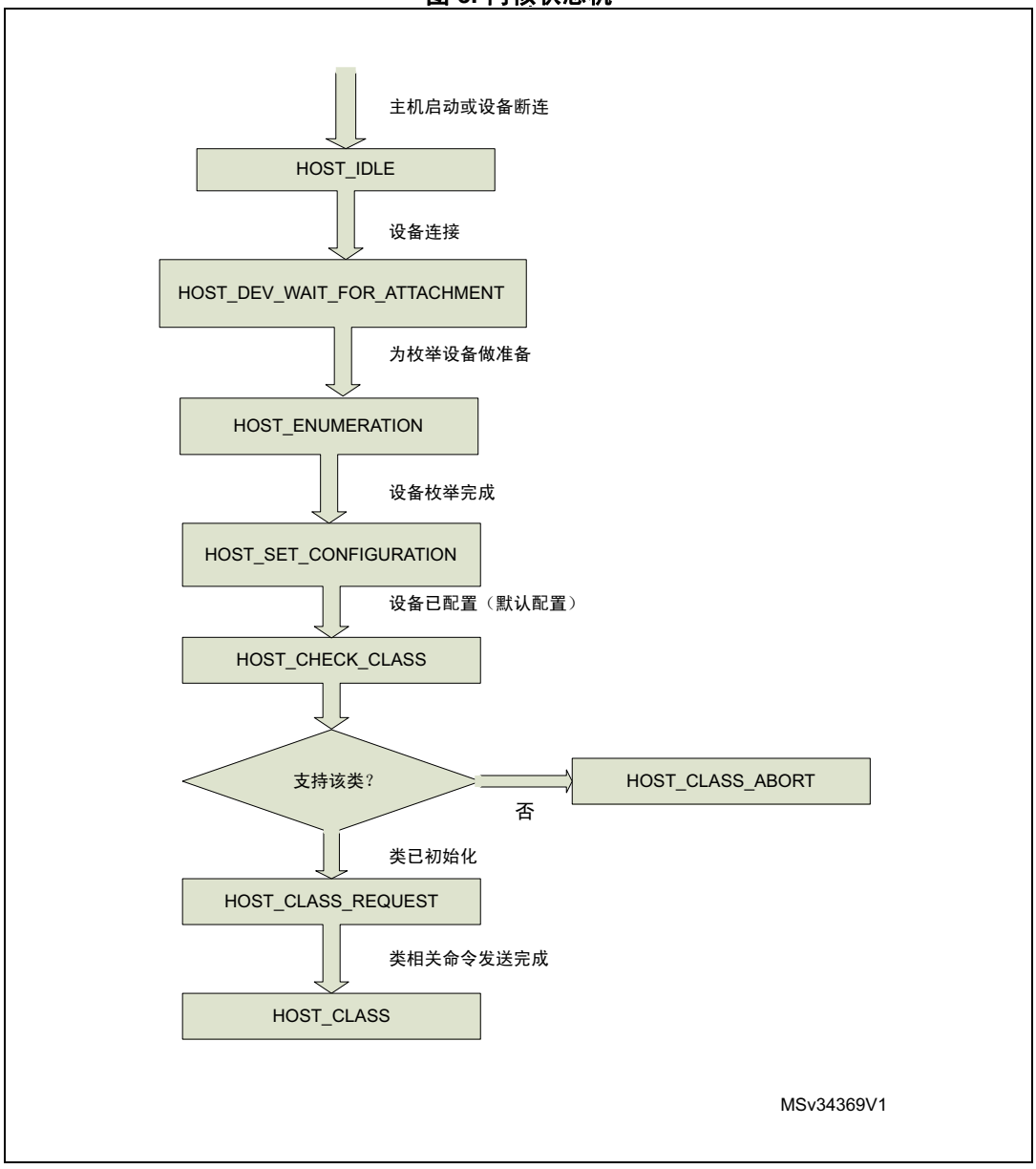
```
typedef enum
{
  CTRL_IDLE =0,
  CTRL_SETUP, /* 发出设置数据包 */
  CTRL_SETUP_WAIT, /* 等待设置 ACK */
  CTRL_DATA_IN, /* 开始执行数据 IN 级 */
```

```
  CTRL_DATA_IN_WAIT, /* 等待数据 IN 级的完成 */
  CTRL_DATA_OUT, /* 开始执行数据 OUT 级 */
  CTRL_DATA_OUT_WAIT, /* 等待数据 OUT 级的完成 */
  CTRL_STATUS_IN, /* 开始状态 IN 级 */
  CTRL_STATUS_IN_WAIT, /* 等待状态 IN 级的完成 */
  CTRL_STATUS_OUT, /* 开始状态 OUT 级 */
  CTRL_STATUS_OUT_WAIT, /* 等待状态 OUT 级的完成 */
  CTRL_ERROR, /* 控制出错 */
  CTRL_STALLED, /* 请求停止 */
  CTRL_COMPLETE    /* 传输全部完成 */
}CTRL_StateTypeDef;
```

4.3 内核状态机概述

图 6 显示了内核状态机的概览。状态机由内核函数 USBH_Process 执行，该函数可在主循环中轮询调用 （单任务模式中）或在 RTOS 任务中调用。

图 6. 内核状态机


主机启动或设备断连
HOST_IDLE
设备连接
HOST_DEV_WAIT_FOR_ATTACHMENT
为枚举设备做准备
HOST_ENUMERATION
设备枚举完成
HOST_SET_CONFIGURATION
设备已配置（默认配置）
HOST_CHECK_CLASS
支持该类?
否
HOST_CLASS_ABORT
类已初始化
HOST_CLASS_REQUEST
类相关命令发送完成
HOST_CLASS
MSv34369V1

以下是各种状态的描述。

HOST_IDLE：当主机启动后或设备断连时进入该状态

HOST_DEV_WAIT_FOR_ATTACHMENT：当设备连接时进入该状态。该状态下，主机会处理设备的连接过程。

HOST_ENUMERATION：当设备完成连接时进入该状态 该状态下，主机会运行设备的标准枚举。

HOST_SET_CONFIGURATION：该状态下，会发出 Set_Config 请求来选择默认配置。

HOST_CHECK_CLASS：该状态会检查已注册的类是否支持所枚举的设备。

HOST_CLASS_ABORT：当枚举设备类不被支持时，会进入该状态。

HOST_CLASS_REQUEST：类接口经过初始化后，主机会进入该状态。该状态下，主机会发出所需的类控制请求。

HOST_CLASS：该状态下，主机会调用类过程函数来处理类后台过程。

4.4 内核的底层驱动接口

如*第 3.1 章节*所述， USB 主机库利用底层接口层作为与 STM32Cube HAL 的接口，与 STM32Cube HAL 底层驱动交互。

底层接口层实现以下功能：

- *表 9* 中定义的 USB 主机库底层接口 API；
- USB 主机板级支持功能 （时钟、 GPIO、中断）；
- USB 主机 HAL 驱动回调函数，用于调用定义在*表 10* 中的 USB 主机库回调函数。

在 STM32Cube 方案中，由于底层接口的某些部分依赖于板和系统，所以提供底层接口实现作为 USB 主机示例的组成部分。

表 9 列出了底层 API 函数。

表 9. 底层接口 API

API	说明
USBH_LL_Init	底层初始化
USBH_LL_DeInit	底层解除初始化
USBH_LL_Start	底层开始
USBH_LL_Stop	底层停止
USBH_LL_GetSpeed	用于获取连接设备所检测到的速度
USBH_LL_ResetPort	发送 USB 复位信号
USBH_LL_GetLastXferSize	获取最近一次完成的传输大小

表 9. 底层接口 API （续）

API	说明
USBH_LL_DriverVBUS	启用或禁用 VBUS
USBH_LL_OpenPipe	打开管道
USBH_LL_ClosePipe	关闭管道
USBH_LL_SubmitURB	提交主机传输请求
USBH_LL_GetURBState	获取管道状态
USBH_LL_SetToggle	设定传输的初始数据同步位 （DATA0 或 DATA1）
USBH_LL_GetToggle	获取数据同步信息

表 10 列出了底层接口在某些 USB 事件后所调用的主机库回调函数。

表 10. 底层事件回调函数

回调函数	说明
USBH_LL_SetTimer	在 USB 主机启动过程中应由 USB 主机 HAL 驱动进行调用，以初始化主机定时器
USBH_LL_IncTimer	每次 SOF 事件均应被调用，使主机定时器变量递增
USBH_LL_SOF	处理需要 SOF 同步的 USB 类过程时，应在 HAL SOF 事件回调中被调用
USBH_LL_Connect	设备连接时，应在 USB 主机 HAL 事件回调中被调用
USBH_LL_Disconnect	设备断连时，应在 USB 主机 HAL 事件回调中被调用
USBH_LL_NotifyURBChange	采用 RTOS 模式时，当 USB 状态发生变化，该回调函数应在 USB 主机 HAL 事件回调中被调用

5 USB 主机库类模块

5.1 类实现模块

类驱动由以下部分组成：

- 一套可在应用层调用的类专用 API。
- 一套事件回调函数 （如果需要的话）
- 由 *USBH_ClassTypedef* 类型结构体所实现的类处理句柄。这些结构体中的函数成员会被 USB 内核处理程序所调用。

图 7 显示了 *USBH_ClassTypedef* 的定义

图 7. 类结构

```
typedef struct
{
  uint8_t              *Name;
  uint8_t               ClassCode;
  USBH_StatusTypeDef  (*Init) (struct _USBH_HandleTypeDef
*phost);
  USBH_StatusTypeDef  (*DeInit) (struct _USBH_HandleTypeDef
*phost);
  USBH_StatusTypeDef  (*Requests) (struct _USBH_HandleTypeDef
*phost);
  USBH_StatusTypeDef  (*BgndProcess) (struct
_USBH_HandleTypeDef *phost);
   USBH_StatusTypeDef  (*SOFProcess)(struct _USBH_HandleTypeDef
```

表 11 列出了类处理程序结构体的成员：

表 11. 主机类句柄结构体

结构体成员	说明
名称	类名称
ClassCode	USB 类代码
初始化	类接口 Init：为类处理操作初始化所需的管道。在主机内核处理程序的 HOST_CHECK_CLASS 阶段被调用
Deinit	类接口 DeInit：解除接口的初始化。设备断连或当主机停止时被调用
Requests	类控制请求：处理类相关请求的状态机，在主机内核处理程序的 HOST_CLASS_REQUEST 阶段被调用

表 11. 主机类句柄结构体 （续）

结构体成员	说明
BgndProcess	类操作的后台处理。在主机内核程序的 HOST_CLASS 阶段被调用。
SOFProcess	SOF 的类处理：需要定期在 SOF 中断处理程序中执行的类相关操作。可用于安排周期性传输 （中断、同步）。
pData	在类的初始化过程中，用保存有类过程变量的类句柄结构体进行初始化

注： *也可在 HOST_CLASS 阶段发送类相关的控制请求。*

5.2 USB 大容量存储类 （MSC）

MSC 后台操作通过 BOT“ 专用批量传输 ” 协议以及 SCSI 指令集，访问 U 盘。

通过以下*表 12* 中的文件实现 MCS 类。

表 12. 用于实现 USB MSC 的文件

文件	说明
usbh_msc.c	执行大容量存储类处理程序以及类的 API
usbh_msc_bot.c	实现专用批量传输 （BOT）协议状态机
usbh_msc_scsi.c	执行 SCSI 指令 （Read10、 Write10、 .....）

MSC 类句柄通过 USBH_ClassTypeDef 类型的结构体来实现：

```
USBH_ClassTypeDef  USBH_msc =
{
  "MSC",
  USB_MSC_CLASS,
  USBH_MSC_InterfaceInit,
  USBH_MSC_InterfaceDeInit,
  USBH_MSC_ClassRequest,
  USBH_MSC_BgndProcess,
  NULL,
  USBH_MSC_Handle
};
```

5.2.1 MSC 类接口初始化

在 MSC 类的接口初始化过程中，会创建两个批量类型的管道：一个批量 IN （输入）管道以及一个批量 OUT （输出）管道，来处理 BOT 协议。

5.2.2 MSC 类相关控制请求

表 13 列出了所实现的类相关控制请求。

表 13. USB 主机大容量存储类处理程序

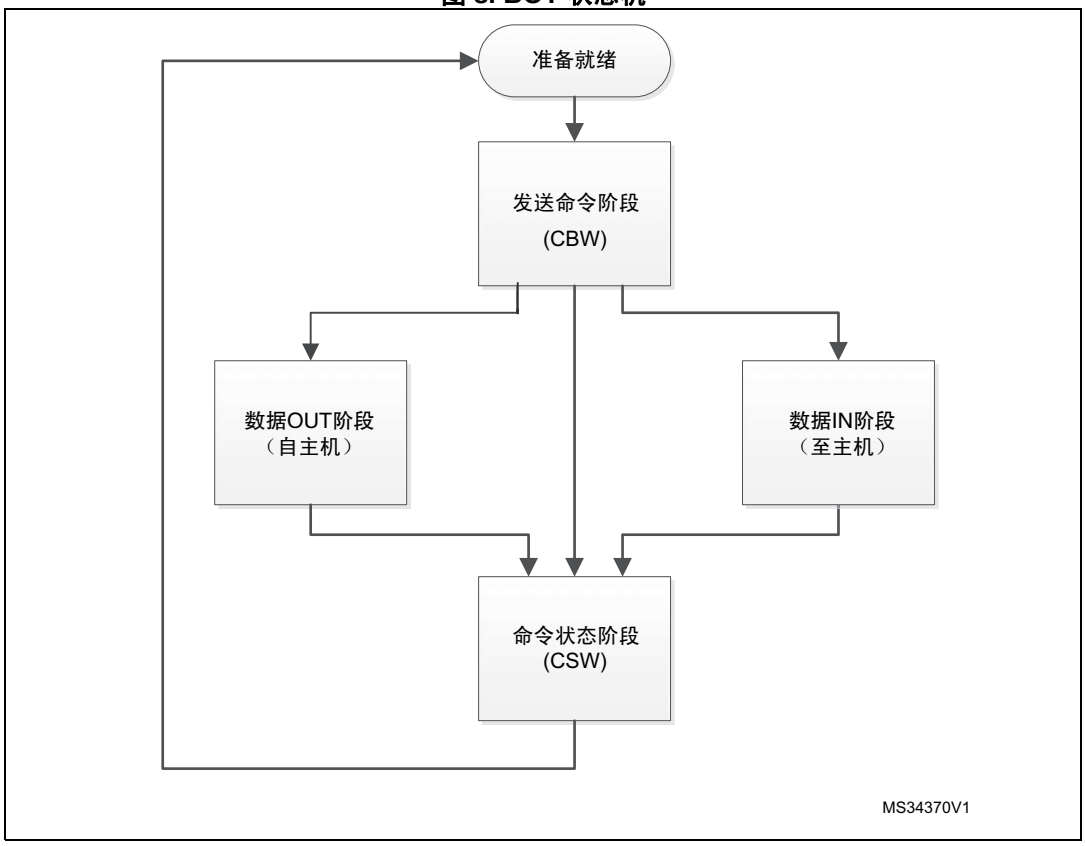
类请求	说明
Get_Max_LUN	用于获取 U 盘的盘符数量。该指令会在类初始化的 HOST_CLASS_REQUEST 阶段发出
BOT_Reset	当 HOST_CLASS 阶段出现 BOT 错误时，主机会发出该请求。

5.2.3 MSC 类处理

MSC 发送 SCSI 命令，以获取逻辑单元相关信息并进而处理磁盘的读 / 写操作。

SCSI 命令基于专用批量传输协议（BOT）通过三级状态机来实现 命令、数据和状态（*图 8*）。

图 8. BOT 状态机



欲了解更多关于 BOT 的详细信息，请参考网站上的规范。（*http://www.usb.org*）。

表 14 列出了所支持的 SCSI 指令。

表 14. SCSI 指令

SCSI 指令	说明
TestUnitReady	测试存储设备是否已准备好
ReadCapacity10	读取存储设备容量
Inquiry	获取大容量存储设备的有关信息 （比如供应商、版本）
RequestSense	用于获取错误信息
Write10	写入一个数据块 （指定扇区起始地址和扇区数量）
Read10	读取一个数据块 （指定扇区起始地址和扇区数量）

5.2.4 MSC 类专用 API

MSC 类提供以下可由文件系统（比如 STM32Cube 中的 FATFS 磁盘管理）调用的类专用 API。

表 15. MSC 类专用 API

API	说明
USBH_MSC_Read	读取一个逻辑单元中的多个扇区 （操作结束之前，该函数不会返回）
USBH_MSC_Write	写入一个逻辑单元中的多个扇区 （操作结束之前，该函数不会返回）
USBH_MSC_GetMaxLUN	获取逻辑单元的数量 （LUN）
USBH_MSC_GetLUNInfo	返回含有逻辑单元信息 （比如容量）的数据结构体
USBH_MSC_IsReady	检查大容量存储设备是否做好读取 / 写入操作的准备

5.2.5 MSC 类的典型使用流程

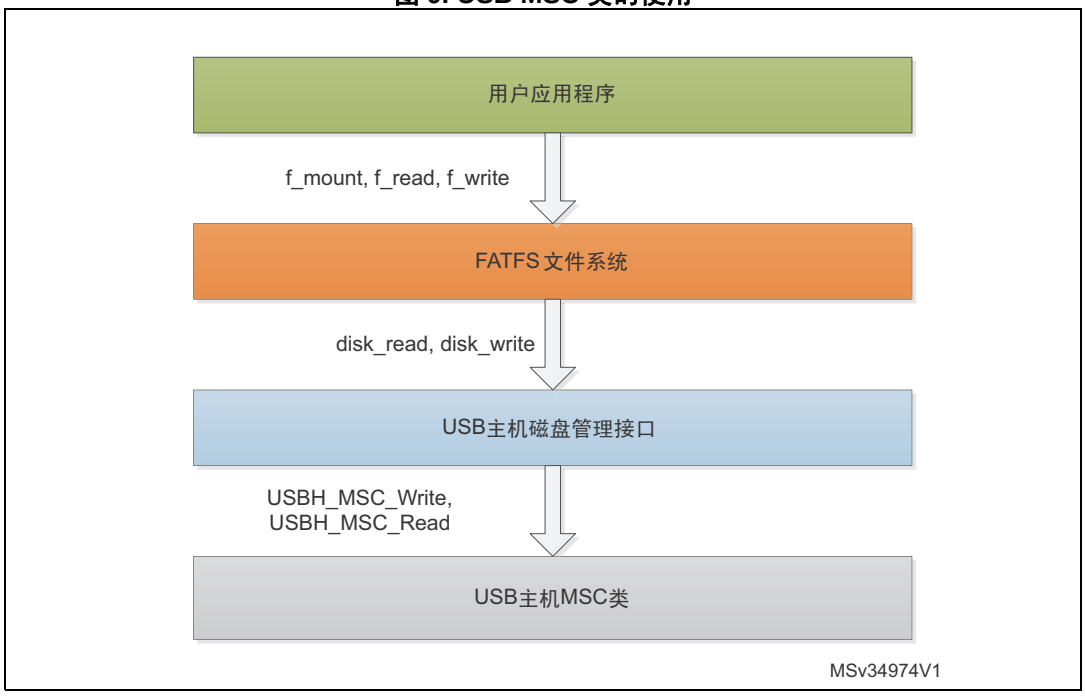
如需访问大容量存储类的设备，主机应实现文件系统。在 STM32Cube 解决方案中，采用 FATFS 文件系统来访问基于 FAT 的大容量存储设备。

如下图所示， FATFS 文件系统与 USB 主机 MSC 类之间是磁盘管理接口。

如果不使用 FATFS，也可以在 MSC 主机库上架构自己的文件系统，只需要参照 STM32Cube 主机库中为 FATFS 提供的磁盘接口来撰写自己的存储介质接口即可。

可参考 USB 主机 MSC_Standalone 或 MSC_RTOS 例程中的 *usbh_diskio.c* 文件来查看磁盘管理接口的实现。

图 9. USB MSC 类的使用



注： *在单任务模式下实现磁盘驱动管理，磁盘访问函数（disk_read, disk_write）应该只能在盘读取或写入操作结束，或超时之后才能返回。*

对于 RTOS 环境下磁盘管理驱动的实现，如果数据传输还未结束，对 MSC 类 API 中的 USBH_MSC_Write 和 USBH_MSC_Read 函数的调用会被阻塞，从而允许存储介质访问未结束前，调用任务可把 CPU 使用权交予其他任务。

5.3 USB HID 鼠标与键盘类 （HID）

HID 类用于通过启动协议来访问鼠标和键盘。类还提供通用的 HID API，用于对类进行定制，以处理其它 HID 设备。

通过以下*表 16* 中的文件实现 HID 类。

表 16. 用于实现 HID 类的文件

文件	说明
usbh_hid.c	实现 HID 类
usbh_hid_parser.c	HID 描述符与报告解析函数
usbh_hid_mouse.c	HID 鼠标专用函数
usbh_hid_keybd.c	HID 键盘专用函数
usbh_hid_usage.h	HID 类的通用定义

HID 类句柄通过 USBH_ClassTypeDef 类型的结构体来实现，其定义如下：

```
USBH_ClassTypeDef  HID_Class =
{
  "HID",
  USB_HID_CLASS,
  USBH_HID_InterfaceInit,
  USBH_HID_InterfaceDeInit,
  USBH_HID_ClassRequest,
  USBH_HID_BgndProcess,
  USBH_HID_SOF_Process,    /* 用于处理中断管道的 SOF 过程 */
  USBH_HID_Handle          /* 处理程序结构体，用于保存 HID
                              的后台过程变量 */
};
```

5.3.1 HID 类接口初始化

接口初始化过程中，根据设备端点描述符，会分配和打开一个中断 IN 端点和 / 或一个中断 OUT 端点。对于启动鼠标与键盘，仅使用中断 IN 管道。

5.3.2 HID 类请求

对 HID 鼠标 / 键盘设备发出的类相关控制请求如*表 17* 所述。

表 17. HID 类请求

类请求	说明
Set Idle	设定 HID 鼠标 / 键盘的轮询周期。在 HOST_CLASS_REQUEST 阶段发送该请求。
Set Protocol	将 HID 协议设为鼠标和 HID 的启动协议。在 HOST_CLASS_REQUEST 阶段发送该请求。

5.3.3 HID 类处理

使用中断 IN 管道，通过接收 HID 报告来处理鼠标 / 键盘操作。

由后台函数和 SOF 函数协同进行鼠标、键盘的数据处理

- SOF 函数处理与 SOF 事件同步的 IN 通信事务的发起。
- 后台函数处理数据的接收以及通过数据接收回调函数来通知应用层。

使用 *USBH_HID_Handle* 结构体来保存过程变量。

5.3.4 HID 类专用 API 与事件回调

表 18 列出了 HID 类中所定义的 API 与事件回调。某些 API 并非用于处理启动鼠标 / 键盘，提供这些 API 给用户是为了定制 HID 类用于除了鼠标 / 键盘以外的其它用途。

表 18. HID API 与事件回调

API	说明
USBH_HID_GetReport	通过控制通道获取报告数据 （HID 启动鼠标 / 键盘例程中没有用到）
USBH_HID_SetReport	通过控制通道发送报告数据 （HID 启动鼠标 / 键盘例程中没有用到）
USBH_HID_SetIdle	设定 HID 轮询周期
USBH_HID_EventCallback	提供给应用层的事件回调：当 IN 中断管道接收到鼠标或键盘 HID 报告数据时被调用。
USBH_HID_SetProtocol	发送 Set_Protocol 控制请求
USBH_HID_GetHIDDescriptor	获取 HID 描述符
USBH_HID_GetHIDReportDescriptor	获取 HID 报告描述符 （启动鼠标 / 键盘例程中没有用到）
USBH_HID_GetDeviceType	返回 HID 设备类型：鼠标或键盘
USBH_HID_GetMouseInfo	获取 HID_MOUSE_Info_TypeDef 类型结构体中的鼠标报告数据 （如下所示）
USBH_HID_GetKeybdInfo	获取 HID_KEYBD_Info_TypeDef 类型结构体中的键盘报告数据 （如下所示）
USBH_HID_GetASCIICode	将键盘按键转换为 ASCII 码

以下分别是鼠标和键盘报告数据中所使用的结构体。

```
typedef struct _HID_MOUSE_Info
{
  int8_t              x;
  int8_t              y;
  uint8_t          buttons[3];
}
HID_MOUSE_Info_TypeDef;
typedef struct
{
  uint8_t lctrl;   /* 左 ctrl 键 */
  uint8_t lshift;  /* 左 shift 键 */
  uint8_t lalt;    /* 左 alt 键   */
  uint8_t lgui;      /* 左 alt 图标 */
  uint8_t rctrl;     /* 右 ctrl 键  */
  uint8_t rshift;    /* 右 shift 键 */
  uint8_t ralt;    /* 右 alt 键 */
  uint8_t rgui;    /* 右 alt 图标 */
  uint8_t keys[KBR_MAX_NBR_PRESSED];    /* 同时按下的键盘键值数组 */
}
HID_KEYBD_Info_TypeDef;
```

5.3.5 HID 类的使用流程

当连接至 HID 鼠标或键盘设备时，应用程序可利用 *USBH_HID_GetKeybdInfo* 或 *USBH_HID_GetMouseInfo* 来周期性轮询 HID 报告，从而获取 HID 鼠标或键盘的报告数据，这些 API 从专用软件 FIFO 内存中读取 HID 报告。

应用程序还可以利用 HID 类回调函数 *USBH_HID_EventCallback( )*，把 HID 报告接收事件同步地报告给上层应用。

对于定制的 HID 设备（非启动鼠标 / 键盘），应用程序可使用 *USBH_HID_SetReport* 或 *USBH_HID_GetReport* 来通过控制管道发送或获取 HID 报告。

5.4 USB 通信设备类（CDC）

CDC 类主机的实现用于访问与抽象控制模型（ACM）子类兼容的 CDC 虚拟通信端口设备。

CDC 类在类文件 *usbh_cdc.c/h* 中实现。该文件定义了 CDC 类处理程序以及类专用 API。CDC 类句柄为 *USBH_ClassTypeDef* 类型，其定义如下：

```
USBH_ClassTypeDef  CDC_Class =
{
  "CDC",
```

```
  USB_CDC_CLASS,
  USBH_CDC_InterfaceInit,
  USBH_CDC_InterfaceDeInit,
  USBH_CDC_ClassRequest,
  USBH_CDC_BgndProcess,
  NULL,
  USBH_CDC_Handle          /* 用于保存过程变量的 CDC 处理程序 */
};
```

5.4.1 CDC 接口初始化

在类的初始化过程中， CDC/ACM 类接口会分配并打开三个管道：

- 两个批量管道 其中一个为 IN 批量管道，另一个为 OUT 批量管道，用于 CDC/ACM 数据传输。
- 一个 IN 中断管道，用于 CDC/ACM 事件通知 （在当前 CDC 类实现中并未用到）。

5.4.2 CDC 类请求

表 19 列出了主机发出的类相关请求。

表 19. CDC 类请求

类请求	说明
Get coding line	获取当前数据帧参数。在 HOST_CLASS_REQUEST 阶段发送该请求。
Set coding line	设定数据帧参数 （比如波特率、奇偶性、停止位）。在 HOST_CLASS 阶段发送该请求。

5.4.3 CDC 类处理

CDC 类的后台函数处理批量管道中的数据接收与传输。

通过回调函数在操作完成时通知应用程序。

USBH_CDC_Handle 结构体内部保存了数据传输所用到的变量。从而在应用中可以使用多个 CDC 类句柄。

5.4.4 CDC 专用 API 和回调函数

表 20 所列的 API 用于处理 CDC 类。

表 20. CDC 类 API 与回调函数

函数	说明
USBH_CDC_GetLineCoding	获取当前数据帧格式参数
USBH_CDC_SETLineCoding	设定当前数据帧格式参数
USBH_CDC_Transmit	发送数据
USBH_CDC_Receive	接收数据

表 20. CDC 类 API 与回调函数 （续）

函数	说明
USBH_CDC_TransmitCallback	数据发送事件的回调函数
USBH_CDC_ReceiveCallback	数据接收事件的回调函数

数据帧格式在 *CDC_LineCodingTypeDef* 型结构体中管理，其定义如下：

```
typedef union _CDC_LineCodingStructure
{
  uint8_t Array[LINE_CODING_STRUCTURE_SIZE];
  struct
  {
    uint32_t dwDTERate;     /* 数据终端的速度，单位为“位/每秒”*/
    uint8_t bCharFormat;    /* 停止位  0 - 1 停止位
                                       1 - 1.5 停止位
                                       2 - 2 停止位 */
    uint8_t bParityType;    /* 奇偶校验性    0 - 无
                                       1 - 奇
                                       2 - 偶
                                       3 - 标志
                                       4 - 空间 */
    uint8_t bDataBits;      /* 数据位（5、6、7、8 或 16）个数。 */
  }b;
}
CDC_LineCodingTypeDef;
```

5.4.5 CDC 类的使用流程

当使用 CDC 类时，应用程序首先需要通过 *USBH_CDC_SETLineCoding* 函数来设定虚拟通信端口设备的数据帧格式参数。然后，应用程序可通过 API 函数 *USBH_CDC_Transmit/USBH_CDC_Receive* 来开始发送或接收数据。这些函数均是非阻塞函数，函数调用之后即可返回。可通过回调函数 *USBH_CDC_TransmitCallback* / *USBH_CDC_TransmitCallback* 向上层应用通知发送 / 接收的结束。

5.5 USB 音频类

USB 音频类主机可用于访问符合 USB 音频类 1.0 规范的 USB 扬声器设备。

USB 音频类在类文件 *usbh_audio.c/.h* 中实现，采用了如下的类句柄：

```
USBH_ClassTypeDef  AUDIO_Class =
{
```

```
  "AUDIO",
  AC_CLASS,
  USBH_AUDIO_InterfaceInit,
  USBH_AUDIO_InterfaceDeInit,
  USBH_AUDIO_ClassRequest,
  USBH_AUDIO_Bgnd_Process,
  USBH_AUDIO_SOF_Process, /* 用于管理同步管道的 SOF 过程（备注） */
  USBH_AUDIO_Handle
};
```

注： *在 V3.0.0 的 STM32 主机库中， SOF 回调函数并不用于处理同步数据传输，所有处理均由后台过程来完成。这一点从库的 V3.1.0 版本发布后便开始改变， SOF 函数中传输的同步音频数据包由 SOF 回调函数来保存，并且由后台操作来处理音频播放的状态机。*

5.5.1 音频类接口初始化

接口初始化过程中，可以分配和开启多达三个管道：

- 一个用于处理扬声器音频数据的异步 OUT 管道
- 一个同步 IN 管道，当检测到麦克风时使用 （该管道并未在当前音频类的实现中使用）
- 一个用于处理音频控制的中断 IN 管道。

注： *该音频类主机的实现没有处理音频同步的反馈管道。音频设备应支持一些其它的同步方法。*

5.5.2 音频类控制请求

表 21 列出了所支持的音频类控制请求。

表 21. 音频类控制请求

请求	说明
音频控制 SET 请求 （CUR、 MAX、 MIN、 RES）	由音频处理状态机发送，用于设置音量控制信息
音频控制 GET 请求 （CUR、 MAX、 MIN、 REST）	在 HOST_CLASS_REQUEST 阶段发送该请求，用于获取音量控制信息 （最大值、最小值、分辨率）
音频设置采样频率	由音频类函数发送，用于在音频播放前设置音频采样频率

5.5.3 音频类处理

音频类处理由后台函数和 SOF 函数共同处理。

SOF 函数用于传输同步音频数据，后台函数用于处理音频播放和音量控制的状态机。

用户应用程序需要调用 *USBH_AUDIO_Play* 来播放开始音频缓冲，该函数提供音频数据缓冲指针，输出音频流从该指针位置处开始，它会将后台过程状态机改为音频播放操作的 AUDIO_PLAYBACK_PLAY 状态。

开始播放过程之后，用户应用程序可以在音频数据缓冲中轮询当前读取位置。可通过调用 *USBH_AUDIO_GetOutOffset* API 函数来完成。

如果 *USBH_AUDIO_GetOutOffset* 返回的当前缓冲读取位置到达既定阈值，用户应用程序就需要分配新的数据缓冲，以便稍后在当前缓冲耗尽时使用。可通过调用 *USBH_AUDIO_ChangeOutBuffer* API 函数来完成。

应用程序的音频数据缓冲大小应根据以下两个参数来进行计算：

- 应用程序对音频数据缓冲的吞吐量 （比如当从 SD 卡读取音频文件时）
- 所需要的音频采用率，它决定了每 1ms （帧）所需发送的字节数。

5.5.4 音频类 API

表 22 列出了所支持的音频类 API。

表 22. 音频类 API

API	说明
USBH_AUDIO_SetVolume	设置音量
USBH_AUDIO_SetFrequency	设置采样率
USBH_AUDIO_Play	从数据源缓冲播放音频流
USBH_AUDIO_Pause	停止音频流播放
USBH_AUDIO_Resume	恢复音频流播放
USBH_AUDIO_ChangeOutBuffer	更改音频流数据源缓冲
USBH_AUDIO_GetOutOffset	返回音频数据源缓冲的当前读取位置

5.5.5 音频类的使用流程

开始播放音频之前，应用程序会调用 *USBH_AUDIO_SetFrequency* 函数来设置设备所需的采样率，另外还能通过调用 *USBH_AUDIO_SetVolume* 函数来设置默认播放音量。

然后，当第一个音频流数据缓冲准备好之后，应用程序可通过调用 *USBH_AUDIO_Play* 函数来开始音频播放。

应用程序需利用 *USBH_AUDIO_GetOutOffset* 函数轮询音频缓冲中的当前读取位置。当到达既定阈值，应用程序需要提供新的音频数据缓冲，以便在播放至当前缓冲末尾时切换数据缓冲区。为分配该新缓冲所调用的函数为 *USBH_AUDIO_ChangeOutBuffer*。

5.6 USB 媒体传输协议类 （MTP）

媒体传输协议类广泛适用于类似安卓智能手机等便携式设备。该协议类似于大容量存储类，可以访问媒体文件（比如音频和图片），差别在于设备中的媒体并非以文件系统的方式呈现，而是以事务方式 （比如获取对象或删除对象）操作访问一套对象，不像文件系统那样允许对扇区的读 / 写访问。

MTP 协议是 ISO 15740 规范中定义的 PTP（图片传输协议）的延伸。关于 MTP 协议的更多详细信息，请参见 USB-IF MTP 类规范文件：《媒体传输协议 1.1 版》。

MTP 类在两个文件中实现：usbh_mtp.c/.h 和 usbh_mtp_ptp.c/.h，它使用以下类句柄

```
USBH_ClassTypeDef  MTP_Class =
```

```
{
  "MTP",
  USB_MTP_CLASS,
  USBH_MTP_InterfaceInit,
  USBH_MTP_InterfaceDeInit,
  USBH_MTP_ClassRequest,
  USBH_MTP_Process,
  USBH_MTP_SOFProcess,
  USBH_MTP_Handle,
};
```

5.6.1 MTP 接口初始化

MTP 类初始化三个通道：

- 一个用于 MTP 数据输入传输的批量 IN 通道，
- 一个用于 MTP 数据输出传输的批量 OUT 通道，
- 一个用于 MTP 事件通知的中断输入通道。

5.6.2 MTP 类控制请求

MTP 类没有实现任何的类专用控制请求。

5.6.3 MTP 类处理

MTP 类由后台函数和 SOF 函数共同处理。

后台函数开启与 MTP 设备的会话，然后获取要用到设备信息及存储单元信息。

SOF 函数处理中断 IN 通道，以获取 MTP 设备的事件通知。

开启与设备的会话之后，接下来的所有操作均通过 MTP 用户应用 API 来完成，这些 API 均为阻塞型 API （比如 USBH_MTP_GetObjectInfo、 USBH_MTP_GetObject）。

5.6.4 MTP 用户应用 API 与回调函数

下表列出 MTP 类所提供的 API 和回调函数：

表 23. MTP API 和回调函数

API	说明
USBH_MTP_IsReady	检查 MTP 设备是否准备好 （开启会话）
USBH_MTP_GetNumStorage	获取 MTP 设备中的存储单元数量
USBH_MTP_SelectStorage	选择特定的存储单元
USBH_MTP_GetStorageInfo	获取存储单元信息
USBH_MTP_GetNumObjects	获取存储单元内具有特定对象格式的对象数量
USBH_MTP_GetObjectHandles	返回包含存储单元内所有对象句柄的数组
USBH_MTP_GetObjectInfo	获取对象信息
USBH_MTP_DeleteObject	删除对象
USBH_MTP_GetObject	获取对象
USBH_MTP_GetPartialObject	获取对象的一部分内容
USBH_MTP_GetObjectPropsSupported	检查所支持的对象属性
USBH_MTP_GetObjectPropDesc	获取对象属性
USBH_MTP_SendObject	发送一个对象至 MTP 设备
USBH_MTP_GetDevicePropDesc	获取设备属性描述数据集
USBH_MTP_EventsCallback	MTP 事件通知的用户回调

5.6.5 MTP 类的使用流程

访问MTP设备之前，应用程序需要调用API函数*USBH_MTP_IsReady*来检查设备是否就绪。

如果就绪的话，应用程序会检查对象格式（比如 WAV 对象）。可通过 *USBH_MTP_GetNumObjects* 函数来检查具有特定格式的可用对象数量。可通过调用 *USBH_MTP_GetObjectHandles* 函数来获取这些对象的句柄。

如需读取对象信息（比如对象文件名），应用程序需要调用*USBH_MTP_GetObjectInfo*函数。

最后，应用程序可以通过 *USBH_MTP_GetPartialObject* 或 *USBH_MTP_GetObject* API 开始访问 MTP 对象。

6 使用 USB 主机库

6.1 USB 主机库配置选项

主机库的配置选项定义在 *usbh_conf.h* 文件中，如下表所述。

表 24. USB 主机库配置选项

配置选项	说明
USBH_MAX_NUM_ENDPOINTS	所支持端点的最大数量
USBH_MAX_NUM_INTERFACES	所支持接口的最大数量
USBH_MAX_NUM_CONFIGURATION	所支持配置的最大数量
USBH_MAX_NUM_SUPPORTED_CLASS	所支持类的最大数量
USBH_KEEP_CFG_DESCRIPTOR	当定义为 1 时，所有配置描述符均保存在内存中
USBH_MAX_SIZE_CONFIGURATION	定义配置描述符的最大大小
USBH_MAX_DATA_BUFFER	定义用于数据传输的最大数据缓冲
USBH_DEBUG_LEVEL	定义日志记录级别： – 0：无日志 – 1：记录用户信息 – 2：记录用户消息与错误消息 – 3：记录用户消息、错误消息与调试消息
USBH_USE_OS	当定义为 1 时，配置主机工作于 OS 模式

usbh_conf.h 文件还为库中所使用的内存管理函数提供重新定义：*USBH_malloc*、*USBH_free* *USBH_memset*、 *USBH_memory*。

6.2 在单任务模式中使用主机库

当库在单任务模式中使用时，典型的 main 函数应包含：

```
void main ()
{
...
/* 初始化主机库 */
  USBH_Init(&hUSBHost, USBH_UserProcess, 0);

  /* 添加支持的类：比如 HID 类 */
  USBH_RegisterClass(&hUSBHost, USBH_HID_CLASS);


/* 开始主机过程 */
  USBH_Start(&hUSBHost);
```

```
  /* 应用程序主循环 */
  while (1)
  {
    /* 应用程序后台过程 */
   Application_Process();

   /* USB 主机过程：应在主循环中调用，运行主机协议栈 */
   USBH_Process();
  }
```

USBH_UserProcess 回调函数用于处理 USB 主机内核事件（比如设备断连、设备连接、设备类激活等）。一般来说，它的实现方式如下所示，用于处理应用的过程状态机：

```
void USBH_UserProcess  (USBH_HandleTypeDef *phost, uint8_t id)
{
  switch (id)
  {
  case HOST_USER_DISCONNECTION:
    Appli_state = APPLICATION_DISCONNECT;
    break;

  /* 当接收到 HOST_USER_CLASS_ACTIVE 事件时，应用程序可以开始与设备通信 */
  case HOST_USER_CLASS_ACTIVE:
    Appli_state = APPLICATION_READY;
    break;

  case HOST_USER_CONNECTION:
    Appli_state = APPLICATION_START;
    break;

  default:
   break;
  }
}
```

请注意，应用程序可以接收多个类的注册，比如：

```
/* 添加支持的类：比如 HID 类和 MSC 类 */
  USBH_RegisterClass(&hUSBHost, USBH_HID_CLASS);
            USBH_RegisterClass(&hUSBHost, USBH_MSC_CLASS);
```

当发生 HOST_USER_CLASS_ACTIVE 事件时，用户应用程序可以通过 *API 函数 USBH_GetActiveClass() 来确定枚举的类。*

6.3 在 RTOS 模式中使用主机库

在 RTOS 模式中，应用程序需要在 *usbh_conf.h* 文件中定义 USBH_USE_OS。

当使用 RTOS 模式时，主机内核后台程序会作为单独的 RTOS 任务来运行。应用程序任务与主机内核任务之间的通信采用 RTOS 消息队列机制。 USB 主机库提供 CMSIS RTOS API 来支持 RTOS 模式。

从用户角度来看，在 RTOS 模式中使用主机库几乎是透明的，因为在 RTOS 模式或单任务模式中使用相同的 API。

6.3.1 RTOS 模式中的典型操作

在 RTOS 模式中，用户需要定义至少一个应用程序，而且还要定义应用程序和主机初始化的初始化线程。以下是库中的 MSC_RTOS 模式示例的摘录。该示例初始化了一个用户线程和一个启动线程。

启动线程初始化应用程序以及 USB 主机，注册其所支持的 USB 类，并在 RTOS 模式下开始主机操作。

```
void main ()
{
   ...
   /* 定义一个用户线程和一个启动线程 */
  osThreadDef(USER_Thread, StartThread, osPriorityNormal, 0,
8onfigMINIMAL_STACK_SIZE);
  osThreadCreate(osThread(USER_Thread), NULL);
   ...
for( ;; );
}


static void StartThread(void const *argument)
{
  osEvent event;

  /* 初始化 MSC 应用程序 */
  MSC_InitApplication();

  /* 启动主机库 */
  USBH_Init(&hUSBHost, USBH_UserProcess, 0);

  /* 添加支持的类 */
  USBH_RegisterClass(&hUSBHost, USBH_MSC_CLASS);

  /* 启动主机处理任务 （因为 USBH_USE_OS =1） */
```

```
  USBH_Start(&hUSBHost);

  for( ;; )
  {
    event = osMessageGet(AppliEvent, osWaitForever);

    if(event.status == osEventMessage)
    {
      switch(event.value.v)
      {
      case APPLICATION_DISCONNECT:
        Appli_state = APPLICATION_DISCONNECT;
        break;

      case APPLICATION_READY:
        Appli_state = APPLICATION_READY;

      default:
        break;
      }
    }
  }
}
```

如上述的代码摘录所示，启动线程首先初始化应用程序，然后初始化主机 （与单任务模式一样），最后进入阻塞状态，等待 USB 事件（Application_Disconnect、Application Ready）。这些事件通过主机用户过程回调函数进行通知，实现方式如下所示：

```
static void USBH_UserProcess(USBH_HandleTypeDef *phost, uint8_t id)
{
  switch(id)
  {
  case HOST_USER_DISCONNECTION:
    osMessagePut(AppliEvent, APPLICATION_DISCONNECT, 0);
    break;

  case HOST_USER_CLASS_ACTIVE:
    osMessagePut(AppliEvent, APPLICATION_READY, 0);
    break;
  default:
    break;
  }
```

```
}
```

注： *正如在 USBH_UserProcess 函数中所示，采用消息队列机制来通知 USB 事件的启动线程。*

6.4 定制底层接口文件 usbh_conf.c

如需定制底层接口文件 *usbh_conf.c*，建议从示例所提供的参考实现开始。

作为参考示例，我们可以参看运行在 STM324xG 评估板上的 MSC 类单任务示例中所使用的 *usbh_conf.c* 文件。该文件保存于 STM32Cube_FW_F4 文件夹，路径如下所示：
`\Projects\STM324xG_EVAL\Applications\USB_Host\MSC_Standalone\Src`

打开 *usbh_conf.c 文件之后，可以找到三组函数：*

- HCD BSP 例程：USB 主机控制器板机支持包函数；
- 底层驱动回调：USB 主机控制器 HAL 驱动回调的实现函数；
- LL 驱动接口：USB 主机库底层接口 API 的实现函数 （请参考*表 9: 底层接口 API*）。

6.4.1 USB 主机控制器 BSP 函数

HCD BSP 功能包含两个函数：

- void HAL_HCD_MspInit(HCD_HandleTypeDef *hhcd)
- void HAL_HCD_MspDeInit(HCD_HandleTypeDef *hhcd)

HAL_HCD_MspInit 函数的作用如下：

- 启用 USB 主机控制器和使能 GPIO 端口时钟。
- 配置所需的 IO
- 配置和使能 USB 主机中断。

如文件 *usbh_conf.c* 所示， BSP 的功能有两种实现方式，取决于采用全速 （FS）还是高速（HS）主机控制器。

用户需要根据硬件的引脚选择来定制 IO 配置。比如， ULPI DIR 功能可以重映射到多个引脚上。

除了配置 USB IO 以外，用户还需要配置一个 GPIO 引脚，用于驱动电荷泵 IC 产生 VBUS 电压。参见示例 *usbh_conf.c* 文件，对于 FS 控制器，引脚 PH5 配置为输出推挽，用于驱动电荷泵 IC。而 HS 控制器则不使用任何引脚，因为电荷泵 IC 的控制由 PHY 来完成。

6.4.2 USB 主机控制器 HAL 驱动回调函数

USB 主机控制器 （HCD） HAL 驱动实现的回调函数列在下述表格中。

表 25. USB 主机控制器 （HCD） HAL 驱动回调

USB 主机 HAL 驱动回调函数	说明
HAL_HCD_SOF_Callback	主机驱动在 SOF 事件时调用，它会调用主机库回调 USBH_LL_IncTimer，对于采用中断和同步管道的 USB 类，该回调函数还会调用 USBH_LL_SOF 主机库回调函数。
HAL_HCD_Connect_Callback	主机驱动在设备连接事件时调用，它会调用主机库回调 USBH_LL_Connect
HAL_HCD_Disconnect_Callback	主机驱动在设备断连事件时调用，它会调用主机库回调 USBH_LL_Disconnect

调用以上函数时不要修改它们。

6.4.3 USB 主机库底层接口 API

该组函数实现了 USB 主机库中的底层 API 层。这是与 USB 主机控制器 HAL 驱动 API 之间的连接层。

所实现的函数如*表 9: 底层接口 API* 所列。

举个例子， *usbh_conf.c* 文件开始时，用户可以定制三个函数，而其它函数必须保持不变。

- USBH_LL_Init
- USBH_LL_DriverVBUS
- USBH_Delay

用户可以使用下表所列的参数来配置 *USBH_LL_Init()* 函数。

表 26. USBH_LL_Init 配置选项

配置选项	说明
hhcd.Instance	可以是 USB_OTG_FS 或 USB_OTG_FS
hhcd.Init.dma_enable	当采用 OTG_HS 模块时，可以使能或关闭 DMA
hhcd.Init.low_power_enable	保留供将来使用。在主机挂起模式中进行低功耗管理
hhcd.Init.phy_itface	选择 PHY 接口，可以是 HCD_PHY_EMBEDDED 或 HCD_PHY_ULPI
hhcd.Init.Sof_enable	使能会在 SOF 引脚上输出 SOF 脉冲
hhcd.Init.speed	主机速度，可以是 HCD_SPEED_FULL 或 HCD_SPEED_HIGH
hhcd.Init.use_external_vbus	当使用 USB HS 主机控制器时，该参数会使能 ULPI PHY 驱动 VBUS

用户需要定制 U*SBH_LL_DriverVBUS()* 函数，以正确驱动电荷泵 IC 以提供 USB VBUS。以下是 *usbh_conf.c* 示例文件中的实现。它驱动 GPIO 的 PH5 引脚：

```
USBH_StatusTypeDef USBH_LL_DriverVBUS(USBH_HandleTypeDef *phost,
uint8_t state)
{
  if(state == 0)
  {
    HAL_GPIO_WritePin(GPIOH, GPIO_PIN_5, GPIO_PIN_SET);
  }
  else
  {
    HAL_GPIO_WritePin(GPIOH, GPIO_PIN_5, GPIO_PIN_RESET);
  }

  HAL_Delay(200); // 200ms 延迟，作为 VBUS 的稳定时间
  return USBH_OK;
}
```

最后，采用 *USBH_LL_Delay* 函数来处理 USB 主机库的延迟，单位为 ms。这一步既可通过调用 HAL 库的延迟函数 *HAL_Delay()* 来实现，也可以利用其它自定义的延迟程序来实现。

6.5 FAQ

USB 主机库支持复合设备 （比如大容量存储 +HID）吗？

支持，前提是用户针对复合设备编写自定义的复合类处理程序。

可以用自己的 USB 主机控制器驱动来使用主机库吗？

可以，仅需实现底层链接接口层（*usbh_conf.c*文件），来调用你自己的USB主机控制器驱动。

可以同时使用 OTG HS 和 OTG FS 主机控制器吗？

可以，请参考双模块示例：`\Applications\USB_Host\DualCore_Standalone`

USB 主机库可以处理通过 USB HUB 连接的设备吗？

不能。 USB 主机库不支持 HUB 类。

USB 主机库允许处理多配置设备吗？

不允许。仅支持单一配置的设备。

7 修订历史

表 27. 文档修订历史

日期	修订	变更
2014 年 5 月 21 日	1	初始版本。

请仔细阅读：

中文翻译仅为方便阅读之目的。该翻译也许不是对本文档最新版本的翻译，如有任何不同，以最新版本的英文原版文档为准。

本文档中信息的提供仅与 ST 产品有关。意法半导体公司及其子公司 （“ST”）保留随时对本文档及本文所述产品与服务进行变更、更正、修改或改进的权利，恕不另行通知。

所有 ST 产品均根据 ST 的销售条款出售。

买方自行负责对本文所述 ST 产品和服务的选择和使用，ST 概不承担与选择或使用本文所述 ST 产品和服务相关的任何责任。

无论之前是否有过任何形式的表示，本文档不以任何方式对任何知识产权进行任何明示或默示的授权或许可。如果本文档任何部分涉及任何第三方产品或服务，不应被视为 ST 授权使用此类第三方产品或服务，或许可其中的任何知识产权，或者被视为涉及以任何方式使用任何此类第三方产品或服务或其中任何知识产权的保证。

除非在 ST 的销售条款中另有说明，否则，ST 对 ST 产品的使用和 / 或销售不做任何明示或默示的保证，包括但不限于有关适销性、适合特定用途 （及其依据任何司法管辖区的法律的对应情况），或侵犯任何专利、版权或其他知识产权的默示保证。

意法半导体的产品不得应用于武器。此外，意法半导体产品也不是为下列用途而设计并不得应用于下列用途：（A）对安全性有特别要求的应用，例如，生命支持、主动植入设备或对产品功能安全有要求的系统； （B）航空应用； （C）汽车应用或汽车环境，且 / 或 （D）航天应用或航天环境。如果意法半导体产品不是为前述应用设计的，而采购商擅自将其用于前述应用，即使采购商向意法半导体发出了书面通知，采购商仍将独自承担因此而导致的任何风险，意法半导体的产品设计规格明确指定的汽车、汽车安全或医疗工业领域专用产品除外。根据相关政府主管部门的规定，ESCC、QML 或 JAN 正式认证产品适用于航天应用。

经销的 ST 产品如有不同于本文档中提出的声明和 / 或技术特点的规定，将立即导致 ST 针对本文所述 ST 产品或服务授予的任何保证失效，并且不应以任何形式造成或扩大 ST 的任何责任。

本文档中的信息取代之前提供的所有信息。

意法半导体集团公司

澳大利亚 - 比利时 - 巴西 - 加拿大 - 中国 - 捷克共和国 - 芬兰 - 法国 - 德国 - 中国香港 - 印度 - 以色列 - 意大利 - 日本 - 马来西亚 - 马耳他 - 摩洛哥 - 菲律宾 - 新加坡 - 西班牙 - 瑞典 - 瑞士 - 英国 - 美国

www.st.com