life.augmented

UM1734
用户手册

STM32Cube USB 设备库

前言

通用串行总线 （USB）为个人计算机史上最成功的互连技术，用于连接鼠标、游戏机、操纵杆、扫描仪、数码相机、打印机等设备。 USB 也已经移植到消费电子和移动产品中。

本文的目的是说明怎样使用 STM32Cube USB 从设备库 (该库支持所有意法半导体的 STM32 系列 MCU)，并如何使用该库实现通用 USB 设备类 （HID 类、 MSC 类、音频类、 CDC 类 ……）产品的开发与应用。

USB 设备库为 STM32Cube 固件包的一部分 （如 STM32CubeF0、 STM32CubeF2、STM32CubeF3、 STM32CubeF4 和 STM32CubeL0），可从 ST 网站（http://www.st.com/stm32cube）免费下载。

目录

表格索引

图片索引

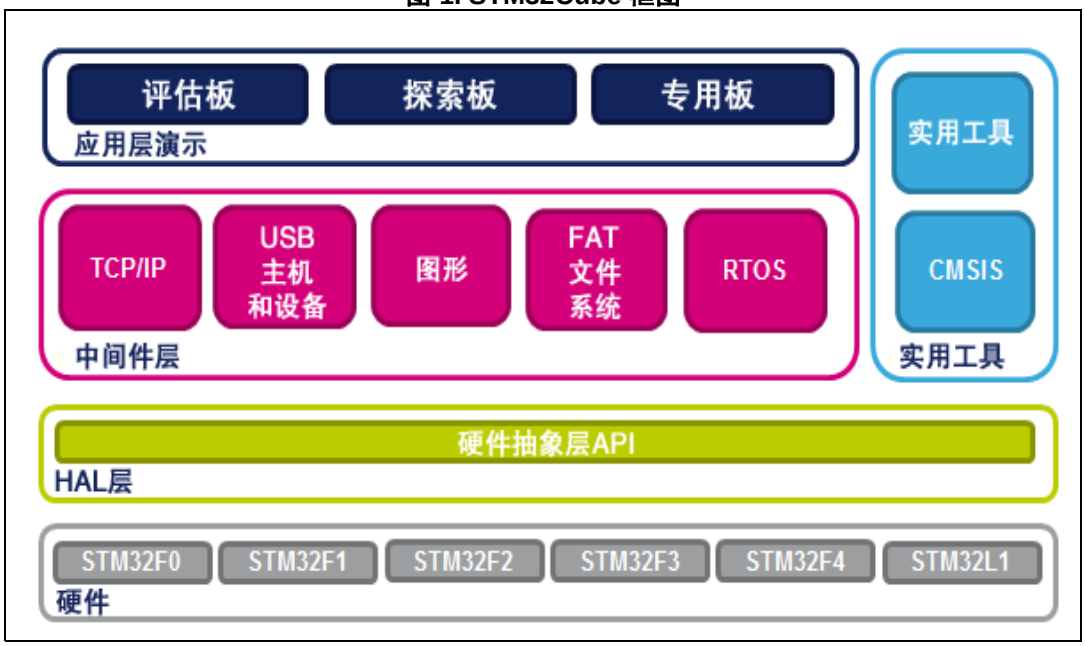
1 STM32Cube 概述

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 涵盖 STM32 整个产品系列。

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。
- 综合的嵌入式软件平台，并针对每个系列提供单独的库文件（例如 STM32CubeF2 用于 STM32F2 系列， STM32CubeF4 用于 STM32F4 系列）
  - STM32 抽象层嵌入式软件 STM32Cube HAL，确保在 STM32 各个产品之间实现最大限度的可移植性
  - 一套一致的中间件，比如 RTOS、 USB、 TCP/IP、图形。
  - 所有嵌入式软件实用工具均配备一套完整的示例。

图 1. STM32Cube 框图



2 前言

2.1 缩写和缩略语

表 1 简要介绍本文档中所用首字母缩略词和缩写词的定义：

表 1. 术语列表

术语	意义
API	应用编程接口
CDC	通信设备类
DFU	设备固件升级
FS	全速 （12 Mbps）
HID	人机界面设备
Mbps	兆比特每秒
MSC	大容量存储类
OTG	On-The-Go：OTG 外设可运行时切换 HOST/DEVICE 角色
PID	USB 产品标识
SCSI	小型计算机系统接口
SOF	帧起始
VID	USB 厂商标识
USB	通用串行总线

2.2 附加信息

除了本文档，意法半导体还提供了关于 USB 的若干其它资源：

- USB HOST 用户手册 **UM1720**
- **UM1725**（STM32F4xx HAL 驱动描述）在此文档中，您可看到两个 USB 通用驱动描述（用于主机的 HCD 和用于设备的 PCD）

2.3 参考

- 通用串行总线规范，版本 2.0， http: //www.usb.org
- USB 设备类规范 （音频、 HID、 MSC 等等）：http://www.usb.org

3 简介

3.1 概述

意法半导体为其客户提供了新型 USB 栈：设备栈和主机栈，可支持所有 STM32 MCU 及多种开发工具，例如 *Atollic*® *TrueSTUDIO*、*IAR 嵌入式 Workbench （用于 ARM* ®），以及 Keil uVision®。

本文侧重于 USB 设备栈。对于主机栈，请参考相关用户手册。

USB 设备库对于所有 STM32 微控制器通用，仅需 HAL 层适配每款 STM32 设备。

USB 设备库位于 STM32Cube USB 设备 HAL 驱动之上。下文讲述了 STM32Cube USB 设备库中间件模块，并举例说明了用户如何使用此库提供的所需 API，轻松开发自己的 USB 设备应用。
USB 设备库是每个 STM32 系列 STM32Cube 包的一部分，包含了 USB 底层驱动、通用类驱动，以及常用 USB 设备类样例的应用示例，可用于 USB 全速和高速传输类型（控制、中断、批量、同步）。 USB 设备库的目的是为每种 USB 传输类型提供至少一个固件演示：

人机界面设备 HID：

- HID 摇杆演示基于 EVAL 板上的嵌入式摇杆及自定义的 HID 样例

音频：

- 音频设备样例用于流音频数据

通信设备 （CDC）：

- VCP USB-RS232 桥，实现了虚拟 COM 端口。

批量：

- 大容量存储演示，基于 EVAL 板上的 microSD 卡。

设备固件升级：

- DFU 用于固件下载和上传

双核设备演示

- 基于具有人机接口的大容量存储和具有 CDC 设备样例的大容量存储

涉及的主题：

- USB 设备库架构
- USB 设备库描述
- USB 设备库状态机概述
- USB 设备类概述

3.2 特性

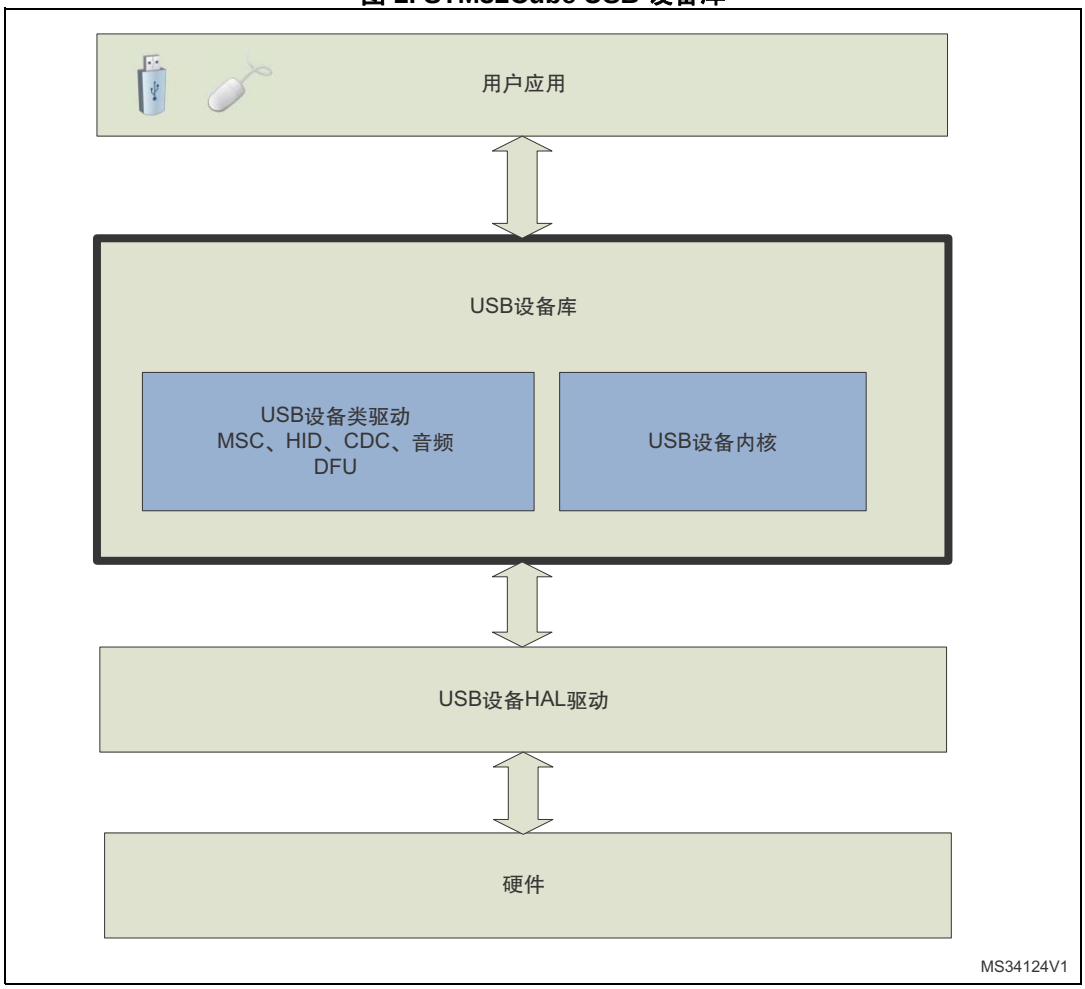
USB 设备库：

- 支持多包传输特性：不需按最大包尺寸划分，即可发送大量数据。
- 支持控制端点上最多 3 个双向传输 （兼容 OHCI 控制器）。
- 无需更改库代码 （只读），使用配置文件更改内核和库配置。
- 包括 32 位对齐数据结构体以处理高速模式中基于 DMA 的传输。
- 支持用户级别的多 USB OTG 内核实例 （配置文件）。

注： *- USB 设备库可与 RTOS 共用或单独使用； CMSIS RTOS 封装的作用是对 OS 内核抽象。*

- USB 设备样例不显示消息。

图 2. STM32Cube USB 设备库



4 USB 设备库架构

4.1 架构概述

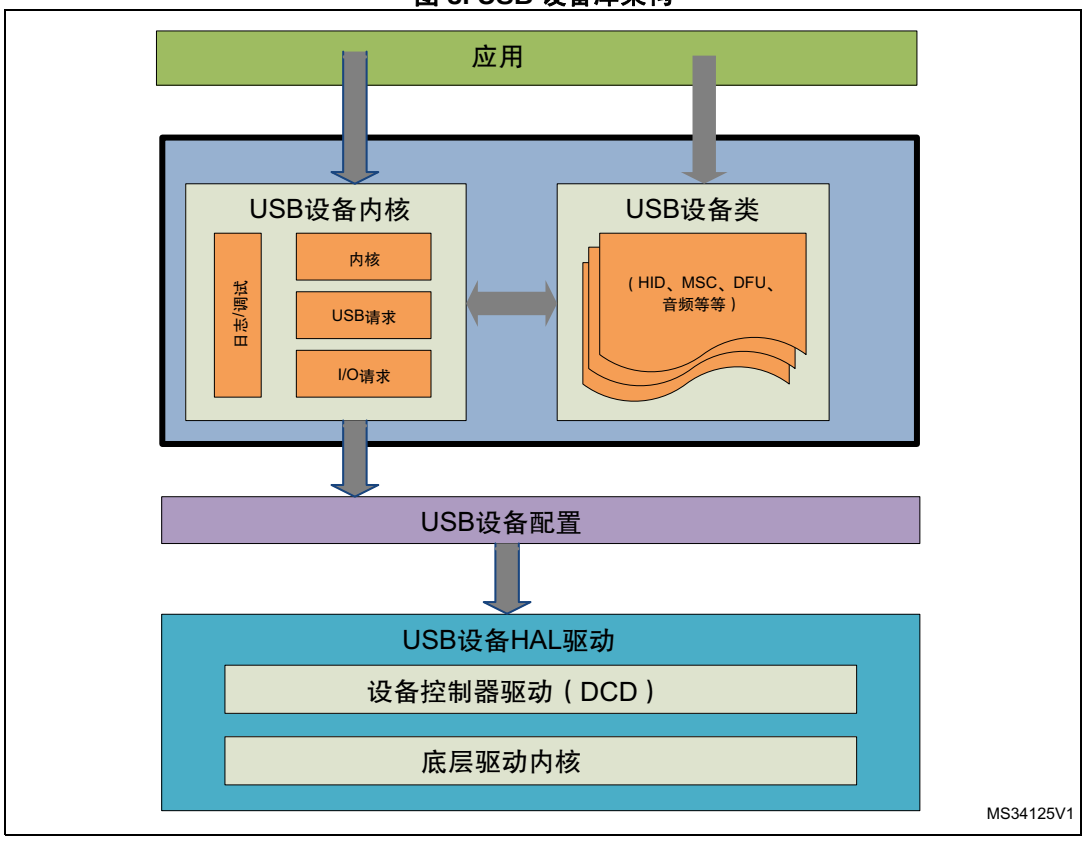
USB 设备库主要分为三层，应用在这三层之上，如下图所示，*图 3: USB 设备库架构*

第一层主要包含两部分：**内核**和**类**驱动。

- 内核包含四个主要模块：
  - USB 内核模块：提供本级 API、管理内部 USB 设备库状态机、处理 USB 中断的回调
  - USB 请求模块：处理 USB 标准协议中第 9 章所规定的请求
  - USB I/O 请求模块：处理底层 I/O 请求
  - USB 日志和调试模块：遵循调试级别 *USB_DEBUG_LEVEL*，输出用户、日志、错误和调试消息。
- USB 设备库包括一组预定义的类驱动，可通过 USBD_RegisterClass () 程序链接至 USB 内核。

USB 设备库为兼容 USB 2.0 的通用 USB 设备栈，与所有 STM32 USB 内核兼容，由于配置封装文件解除了 USB 库与底层驱动的依赖关系，因此它可轻松链接至任何 USB HAL 驱动。

图 3. USB 设备库架构

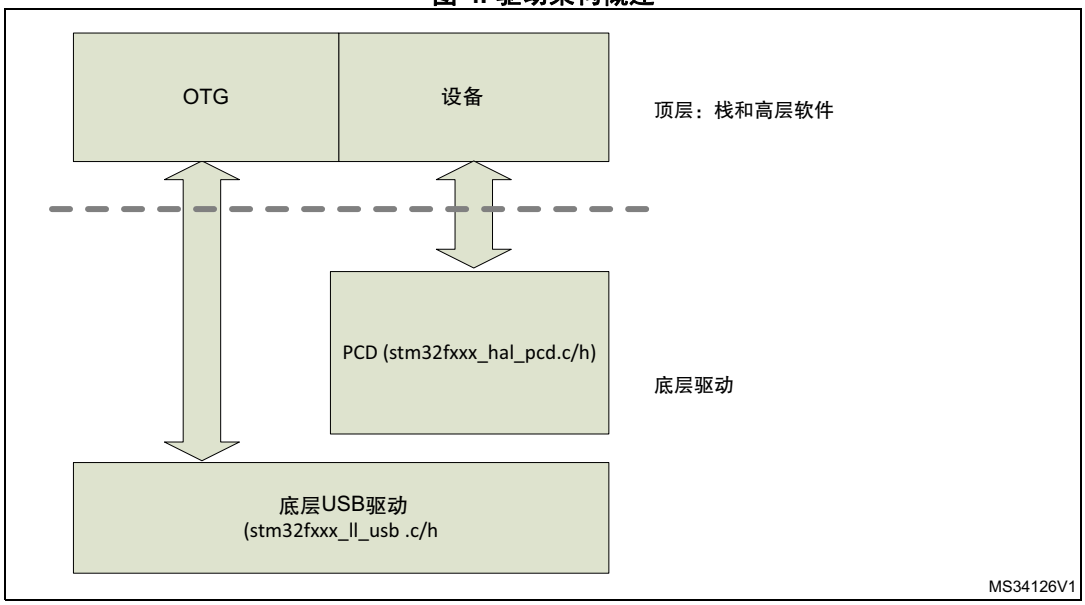


5 USB OTG 硬件抽象层

可使用底层驱动将 USB OTG 内核与高层栈连接起来。

5.1 驱动架构

图 4. 驱动架构概述



- 底层（底层 USB 驱动）为设备和 OTG 模式提供了通用 API：每种模式中的内核初始化及对传输流的控制
- 外设控制器驱动 （PCD）层提供了访问设备模式及此模式中处理中断程序的 API。
- OTG 控制器驱动 （OTG）层提供了访问 OTG 模式及此模式中处理中断程序的 API。

注： *关于如何使用 PCD 驱动的更多信息，请参考* ***UM1725****。在 UM1725 中说明了所有 PCD 驱动 API。*

5.2 USB 驱动编程手册

5.2.1 配置 USB 驱动结构体

设备初始化

使用 stm32fxxx_hal_pcd.c 文件中的下述函数初始化设备：

HAL_StatusTypeDef HAL_PCD_Init(PCD_HandleTypeDef *hpcd)

端点配置

USB 内核初始化之后，上层可能调用底层驱动，打开或关闭激活端点，开始传输数据。使用下列两个 API：

HAL_StatusTypeDef **HAL_PCD_EP_Open**(PCD_HandleTypeDef *hpcd, uint8_t ep_addr, uint16_t ep_mps, uint8_t ep_type)

HAL_StatusTypeDef **HAL_PCD_EP_Close**(PCD_HandleTypeDef *hpcd, uint8_t ep_addr)

ep_addr、 ep_mps 和 ep_type 分别为端点地址、最大数据传输和传输类型。

设备内核结构体

设备库中使用的主要结构体为设备句柄，其类型为 “USBD_HandleTypedef” *第 13 页图 5*

USB 全局设备结构体包含所有变量和结构体，用以实时保存与设备、控制传输状态机以及端点的状态相关的所有信息。

在本结构体中，dev_config 保存着当前 USB 设备配置，ep0_state 控制着状态机，它具有些列状态：

```
/* EP0 状态 */
#define USBD_EP0_IDLE                           0
#define USBD_EP0_SETUP                          1
#define USBD_EP0_DATA_IN                        2
#define USBD_EP0_DATA_OUT                       3
#define USBD_EP0_STATUS_IN                      4
#define USBD_EP0_STATUS_OUT                     5
#define USBD_EP0_STALL                          6
```

在此结构体中， dev_state 定义了连接、配置和上电状态：

```
/* 设备状态 */
#define USBD_DEFAULT                            1
#define USBD_ADDRESSED                          2
#define USBD_CONFIGURED                         3
#define USBD_SUSPENDED                          4
```

注： *USB 规范 （第 9 章中）定义了 USB 设备的六种状态：*

***连接**：设备连接到 USB，但未由 USB 供电*

***供电**：设备连接到 USB 并由其供电，但还未接到复位请求。*

***默认**：设备连接到 USB，由其供电并复位，但并未分配唯一地址。*

***地址**：设备连接到 USB，由其供电，已复位并被分配唯一地址。*

***配置**：设备已处于地址状态，被配置，且处在挂起状态。*

***挂起**：设备被连接并配置，但在总线上已 3ms 未激活*

图 5. USBD_HandleTypedef

```
typedef struct _USBD_HandleTypeDef
{
  uint8_t                 id;
  uint32_t                dev_config;
  uint32_t                dev_default_config;
  uint32_t                dev_config_status;
  USBD_SpeedTypeDef       dev_speed;
  USBD_EndpointTypeDef    ep_in[15];
  USBD_EndpointTypeDef    ep_out[15];
  uint32_t                ep0_state;
  uint32_t                ep0_data_len;
  uint8_t                 dev_state;
  uint8_t                 dev_old_state;
  uint8_t                 dev_address;
  uint8_t                 dev_connection_status;
  uint8_t                 dev_test_mode;
  uint32_t                dev_remote_wakeup;
  USBD_SetupReqTypedef    request;
  USBD_DescriptorsTypeDef *pDesc;
  USBD_ClassTypeDef       *pClass;
  void                    *pClassData;
  void                    *pUserData;
  void                    *pData;
} USBD_HandleTypeDef;
```

USB 数据传输流程

PCD 层提供所需的所有 API，可启动及控制传输流，

见下列函数：

HAL_StatusTypeDef HAL_PCD_EP_Transmit(PCD_HandleTypeDef *hpcd, uint8_t ep_addr, uint8_t *pBuf, uint32_t len)

HAL_StatusTypeDef HAL_PCD_EP_Receive(PCD_HandleTypeDef *hpcd, uint8_t ep_addr, uint8_t *pBuf, uint32_t len)

HAL_StatusTypeDef HAL_PCD_EP_SetStall(PCD_HandleTypeDef *hpcd, uint8_t ep_addr)

HAL_StatusTypeDef HAL_PCD_EP_ClrStall(PCD_HandleTypeDef *hpcd, uint8_t ep_addr)

HAL_StatusTypeDef HAL_PCD_EP_Flush(PCD_HandleTypeDef *hpcd, uint8_t ep_addr)

PCD 层有一个函数必须被 USB 中断调用：

void HAL_PCD_IRQHandler(PCD_HandleTypeDef *hpcd)

stm32fxxx_hal_pcd.*h* 文件包含了库内核层处理 USB 事件时调用的函数原型。

重要的枚举类型定义

USBD_StatusTypeDef

几乎所有库函数都会返回类型为 USBD_StatusTypeDef 的状态，应用应该始终检查返回的状态。

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

下表说明了上面的返回状态的含义：

表 2. USB 设备状态

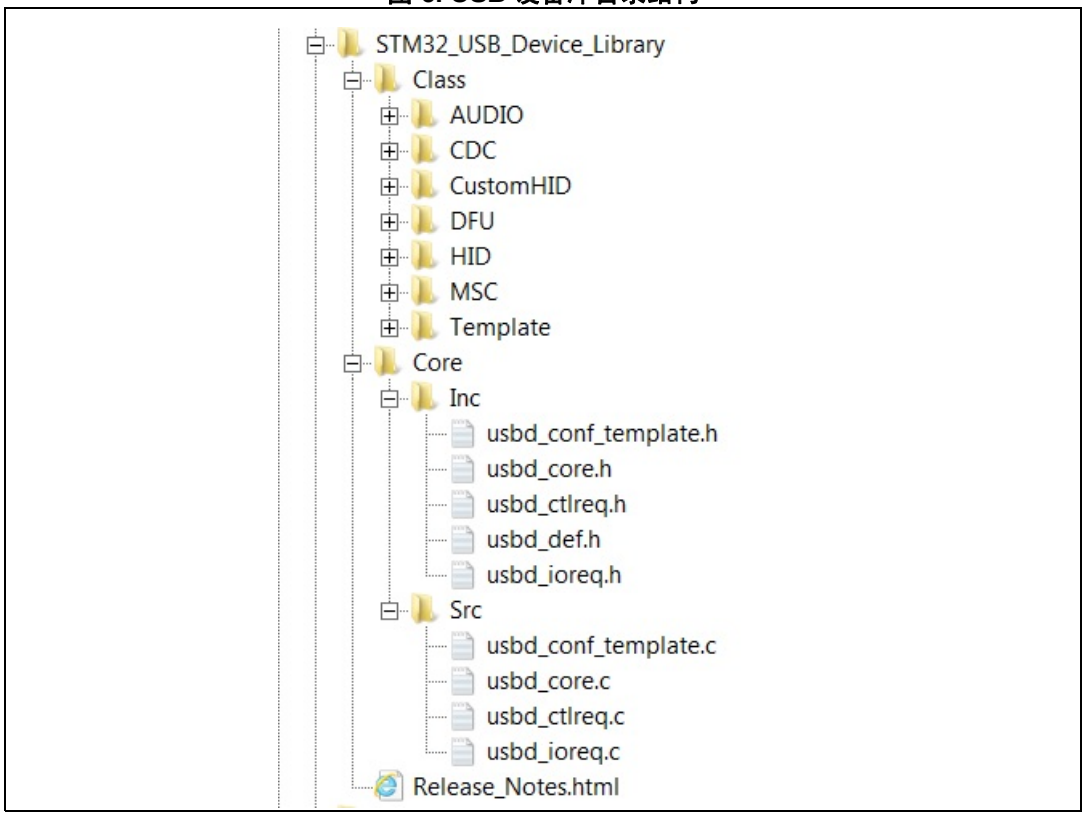
状态	说明
USBH_OK	操作成功完成时返回
USBH_BUSY	操作未完成 （忙）时返回
USBH_FAIL	因底层错误或协议失败导致操作失败时返回

6 USB 设备库概述

USB 设备库基于通用 USB 底层驱动开发，可在全速和高速模式中工作。

它实现了USB 2.0版本定义的USB设备库状态机。此模块功能由USB设备库固件包中**"Core"**目录下的文件提供 （参见*图 6*）。 USB 类模块为类层，与协议规范兼容。

图 6. USB 设备库目录结构



6.1 USB 设备库描述

6.1.1 USB 设备库流程

处理控制端点 0

USB 规范定义了四种传输类型：控制传输、中断传输、批量传输和同步传输。USB 主机通过控制端点向设备发送请求（在这种情况下，控制端点为端点 0）。请求作为 SETUP 包发送到设备。这些请求可分为三类：标准、特定类、特定厂商。因为标准请求对所有 USB 设备都是通用的，所以库可收到和处理控制端点 0 上的所有标准请求。

特定类和特定厂商请求的格式和意义并不对所有 USB 设备通用。

所有 SETUP 请求都由处于中断模式的状态机处理。 USB 正确传输的末尾会产生一个中断。库代码收到该中断。在中断处理程序中，触发端点被识别。如果事件为端点 0 上的设置，则保存所收到设置的参数且启动状态机。

非控制端点上的事务

特定类内核使用非控制端点，方法是通过数据 IN 和 OUT 阶段回调，调用一组函数发送或接收数据。

SETUP 包的数据结构

当一个新的 SETUP 包到达时， SETUP 包的所有八个字节都被复制到一个内部结构体 *USB_SETUP_REQ req*，因此在处理期间，下一个 SETUP 包不会覆盖前一个包。此内部结构体定义为：

```
typedef  struct  usb_setup_req
{
    uint8_t   bmRequest;
    uint8_t   bRequest;
    uint16_t  wValue;
    uint16_t  wIndex;
    uint16_t  wLength;
}USBD_SetupReqTypedef;
```

标准请求

下面 USB 规范表中的大多数请求都作为库中的标准请求处理。该表列出了所有标准请求及其库中的有效参数。此表中没有的请求则认为非标准请求。

表 3. 标准请求

-	状态	bmRequestType	*wValue* 的低字节	*wValue* 的高字节	*wIndex* 的低字节	*wIndex* 的高字节	wLength	注释
GET_STATUS	A, C	80	00	00	00	00	2	获取设备状态。
	C	81	00	00	N	00	2	获取接口状态，其中 N 为有效接口号。
	A, C	82	00	00	00	00	2	获取端点 0 OUT 方向的状态。
	A, C	82	00	00	80	00	2	获取端点 0 IN 方向的状态。
	C	82	00	00	EP	00	2	获取端点 EP 的状态。

表 3. 标准请求 （续）

-	状态	bmRequestType	*wValue* 的低字节	*wValue* 的高字节	*wIndex* 的低字节	*wIndex* 的高字节	wLength	注释
CLEAR_FEATURE	A, C	00	01	00	00	00	00	清除设备远程唤醒特性。
	C	02	00	00	EP	00	00	清除端点 EP 的 STALL 状态。 EP 不指向端点 0。
SET_FEATURE	A, C	00	01	00	00	00	00	设置设备远程唤醒特性。
	C	02	00	00	EP	00	00	设置端点 EP 的 STALL 状态。 EP 不指向端点 0。
SET_ADDRESS	D, A	00	N	00	00	00	00	设置设备地址， N 为有效设备地址。
GET_DESCRIPTOR	ALL	80	00	01	00	00	非0值	得到设备描述符。
	ALL	80	N	02	00	00	非0值	得到配置描述符，其中 N 为有效配置索引。
	ALL	80	N	03	LangID		非0值	获取字符串描述符，其中 N 为有效字符串索引。此请求仅在支持字符串描述符时有效。
GET_CONFIGURATION	A, C	80	00	00	00	00	1	获取设备配置。
SET_CONFIGURATION	A, C	80	N	00	00	00	00	设置设备配置，其中 N 为有效配置号。
GET_INTERFACE	C	81	00	00	N	00	1	得到接口 N 的另一设置；其中 N 为有效接口号。
SET_INTERFACE	C	01	M	00	N	00	00	设置接口 N 的另一设置 M ；其中 N 为有效接口号， M 为接口 N 的有效另一设置。

注： *在列状态中：D = 默认状态； A = 地址状态； C = 配置状态； All = 全部状态。*
EP: D0-D3 = 端点地址； D4-D6 = 保留为零； D7= 0: OUT 端点， 1：IN 端点。

非标准请求

所有非标准请求都通过回调函数传至特定类的代码。

– SETUP 阶段

库传递所有非标准请求到特定类的代码，回调为*pdev->pClass->Setup (pdev, req)*函数。

非标准请求包括用户解释请求和无效请求。用户解释请求为特定类请求、特定厂商请求，或库认为无效但应用欲解释为有效的请求。

无效请求为非标准请求、非用户解释请求。因为 *pdev->pClass->Setup (pdev, req)* 在 SETUP 阶段后、数据阶段前调用，所以用户代码应负责在 **pdev->pClass->Setup (pdev, req)** 中解析 SETUP 包的内容（req）。若请求无效，则用户代码必须调用 **USBD_CtlError(pdev , req)**，并返回 **pdev->pClass->Setup (pdev, req)** 的调用者

对于用户解释请求，如果请求有数据阶段，用户代码应为后续数据阶段准备数据缓冲；否则用户代码执行请求，返回 pdev->pClass->Setup (pdev, req) 的调用者。

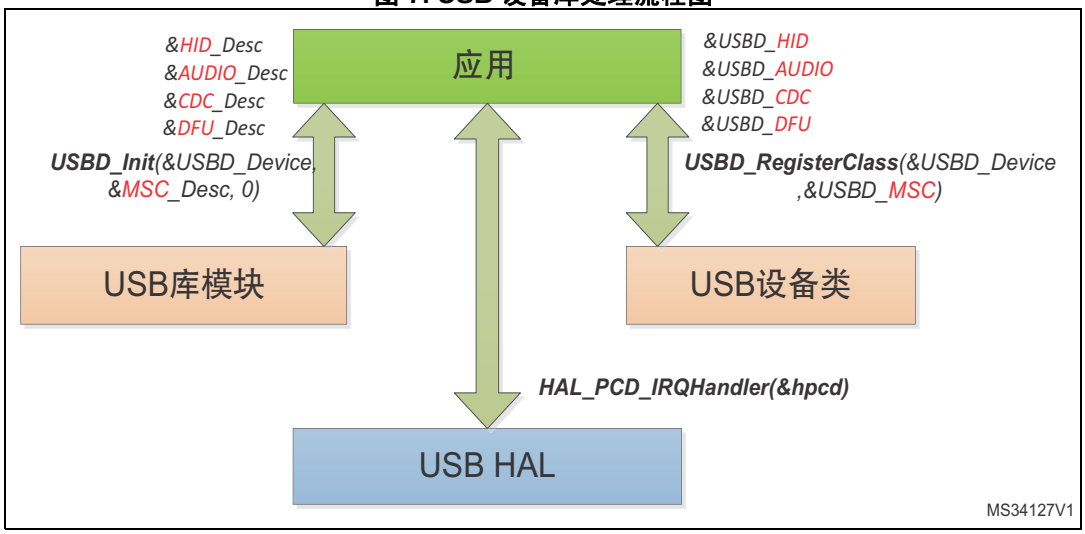
– DATA 阶段

类层使用标准 USBD_CtlSendData 和 USBD_CtlPrepareRx 发送或接收数据，数据传输流由库内部处理，用户不需要将数据切分为 ep_size 大小的包。

– 状态阶段

当从 pdev->pClass->Setup (pdev, req) 回调返回之后，状态阶段由库处理。

图 7. USB 设备库处理流程图



如*图7: USB 设备库处理流程图*中所示，USB编程只需要三个模块 USB库、USB类和主应用。

主应用执行用户定义程序，**main.c、stm32fxx_it.c、usbd_conf.c 和 usbd_desc.c** 及其头文件为用户开发自己的应用时的主要文件（应用强制），用户可根据应用对其修改（类驱动）

仅需为应用层和 USB 库模块间的接口调用简单的 API，它们处理 USB 初始化并得到 USB 的当前状态。

若需初始化 USB HAL 驱动、 USB 设备库、和板级支持包 （BSP）并启动库，用户需调用这三个 API

– **USBD_Init ()：**此函数初始化设备栈，加载类驱动与描述符地址。

设备描述符存储于 *usbd_desc.c* 和 *usbd_desc.h （用于配置描述符类型）*文件中。

– **USBD_RegisterClass()：**此函数将类驱动链接到设备内核。

– **USBD_Start()：**此函数允许用户开启 USB 设备内核

例如在 **usbd_conf file 文件中：**

– 根据调用 USBD_LL_Init() 函数时的类要求，用户可在 *dev_endpoints* 变量中增加其它端点，它应包含遵循 USB 类规范的所有强制端点。

通过 usbd_conf.h 文件，USB 设备库提供了若干配置，请参考第 6.1.5 章节: 配置 USB 设备固件库第 22 页 以得到更多信息。

注： *HAL 库初始化通过 stm32fxxx_hal.c 中的 HAL_Init() API 完成，该方法执行如下操作：*

- 复位所有外设

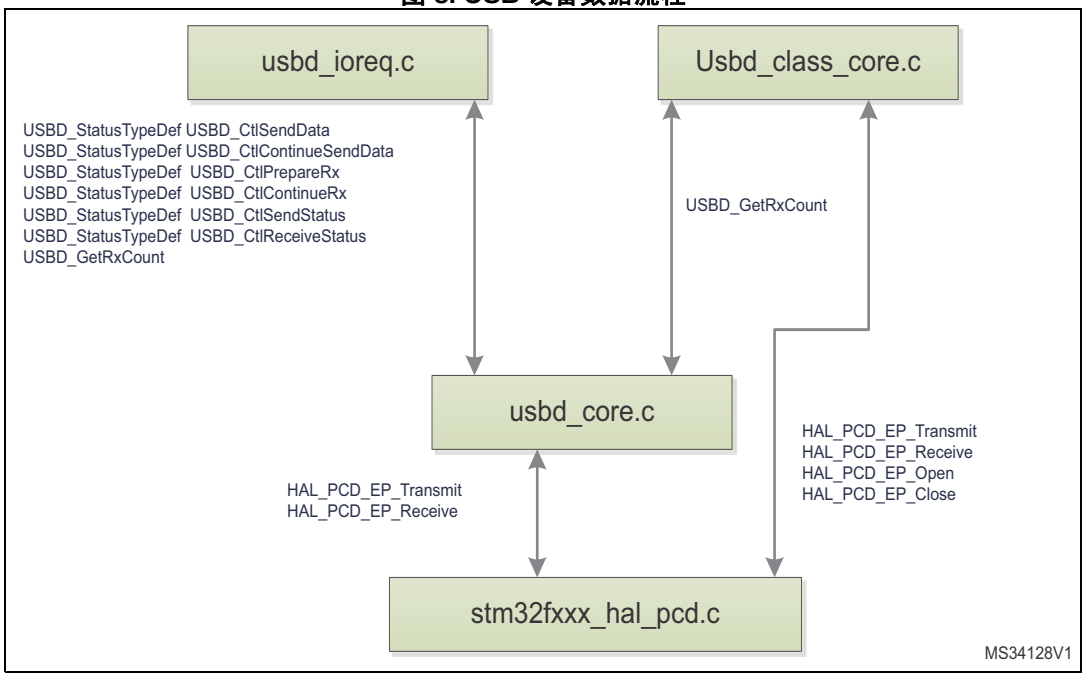
- 配置 Flash 预取、指令缓存、数据缓存

- 使能 systick，配置 1ms 滴答时钟 （复位之后的默认时钟为 HSI）

6.1.2 USB 设备数据流程

因为 USB OTG 内核支持多包特性，所以在需要封装来管理控制端点上的多包特性时， USB 库 （USB 内核和 USB 类层）通过 IO 请求层处理端点
0 （EP0）上的数据处理，当其它端点被使用时，直接从 stm32fxxx_hal_pcd 层处理。下图显示了该数据流方案。

图 8. USB 设备数据流程



6.1.3 具有底层驱动的内核接口

如前面所述，底层接口层为 STM32Cube HAL 的链接层， USB 设备库用其与 STM32Cube HAL 底层驱动接口。

底层接口实现了底层 API 函数，在 USB 事件后调用库内核回调函数。

在 STM32Cube 解决方案中，因为底层接口的一些部分依赖于板子和系统，所以底层接口的实现作为 USB 设备样例的一部分提供。

下表列出了底层 API 函数：

注： *这些 API 由 USB 设备配置文件 （usbd_conf.c）提供。用户应在用户文件中实现，适配到 USB 设备控制器驱动。*

用户可从 STM32Cube 包内提供的 usbd_conf.c 文件开始。此文件也可复制到应用目录，根据应用需要修改。

表 4. API 描述

API	说明
USBD_LL_Init	底层初始化
USBD_LL_DeInit	底层取消初始化
USBD_LL_Start	底层开始
USBH_LL_Stop	底层停止
USBD_LL_OpenEP	初始端点
USBD_LL_CloseEP	关闭并对端点状态取消初始化

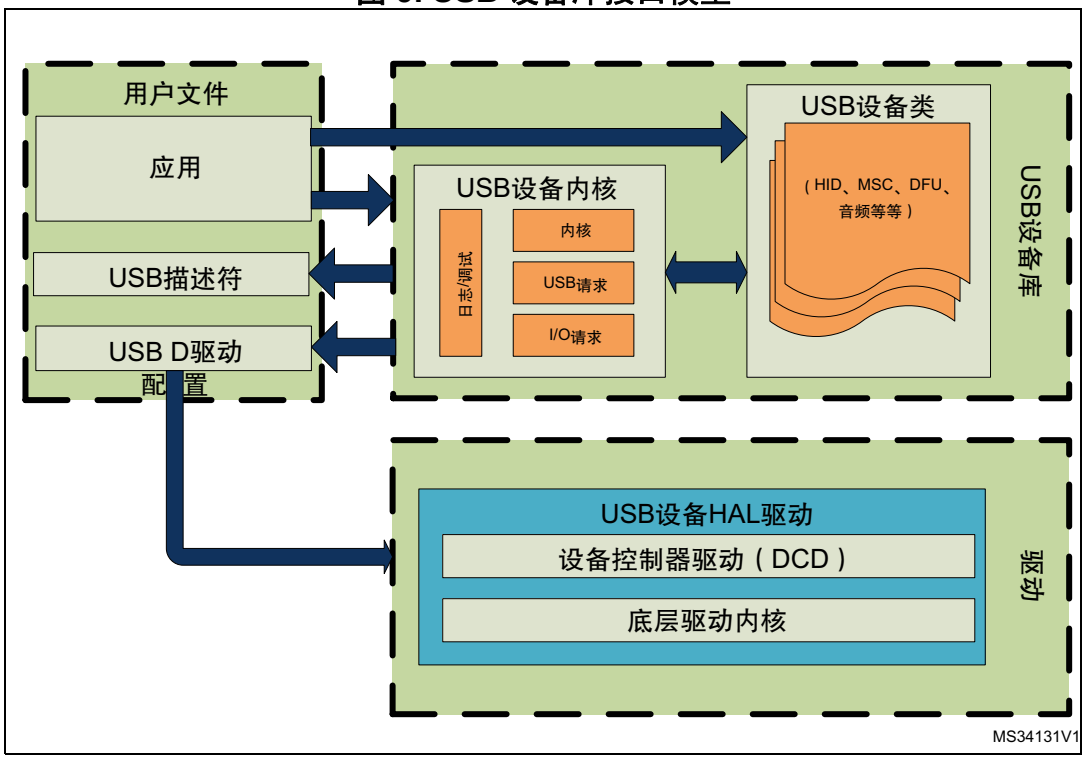
表 4. API 描述 （续）

API	说明
USBD_LL_FlushEP	除掉底层驱动的一个端点。
USBD_LL_StallEP	在底层驱动的端点上设置暂停状态。
USBD_LL_ClearStallEP	在底层驱动的端点上清除暂停状态。
USBD_LL_IsStallEP	返回暂停状态。
USBD_LL_SetUSBAddress	为设备指定 USB 地址
USBD_LL_Transmit	通过端点传输数据
USBD_LL_PrepareReceive	为接收准备端点
USBD_LL_GetRxDataSize	返回最后传输的包大小。

6.1.4 USB 设备库接口模型

USB 设备库是由通用可移植的 USB 设备内核和类模块构建而成。

图 9. USB 设备库接口模型



下面是在 USB 事件后，底层接口调用的设备库回调函数。

表 5. 底层事件回调函数

回调函数	说明
HAL_PCD_ConnectCallback	设备连接回调
HAL_PCD_DataInStageCallback	数据 IN 阶段回调
HAL_PCD_DataInStageCallback	数据 OUT 阶段回调
HAL_PCD_DisconnectCallback	中断连接回调
HAL_PCD_ISOINIncompleteCallback	ISO IN 事务回调
HAL_PCD_ISOINIncompleteCallback	ISO OUT 事务回调
HAL_PCD_ResetCallback	USB 复位回调
HAL_PCD_ResumeCallback	USB 重新开始回调
HAL_PCD_SetupStageCallback	设置阶段回调
HAL_PCD_SOFCallback	帧回调起始
HAL_PCD_SuspendCallback	挂起回调

6.1.5 配置 USB 设备固件库

可使用 *usbd_conf.h* 文件配置 USB 设备库。

usbd_conf.h 为特定的配置文件，用于定义一些全局参数及特定配置。usbd_conf.c 文件为接口文件，用于将上层库与 HAL 驱动和 BSP 驱动链接起来

表 6. USB 库配置

项目	参数	说明
通用配置	USBD_MAX_NUM_CONFIGURATION	所支持配置的最大数 [1 至 255]
	USBD_MAX_NUM_INTERFACES	所支持接口的最大数 [1 至 255]
	USBD_MAX_STR_DESC_SIZ	字符串描述符的最大大小 [uint16]
	USBD_SELF_POWERED	使能自开机特性 [0/1]
	USBD_DEBUG_LEVEL	调试和日志等级
	USBD_SUPPORT_USER_STRING	使能对用户字符串的支持 [0/1]
大容量存储配置	MSC_MEDIA_PACKET	媒体 I/O 缓冲大小，为 512 的倍数 [512 至 32K]
HID 配置	NA	NA

表 6. USB 库配置 （续）

项目	参数	说明
DFU 配置	USBD_DFU_MAX_ITF_NUM	最大媒体接口数 [1 至 255]
	USBD_DFU_XFER_SIZE	媒体 I/O 缓冲大小，为 512 的倍数 [512 至 32 K]
	USBD_DFU_APP_DEFAULT_ADD	应用地址 (0x0800C000)
CDC 配置	NA	NA
音频配置	USBD_AUDIO_FREQ	8 KHz 到 48 KHz

注： *用户可从 STM32Cube 提供的 usbd_conf.h 文件开始。此文件也可复制到应用目录，根据应用需求修改。*

注： *默认情况下，对于 USB 设备样例，库与用户消息**并不会显示**在 LCD 上。*

*但是用户可实现自己的消息（**若要将库消息重定向到 LCD 屏幕上，需将 lcd_log.c 驱动添加到应用源中**）并可选择是否显示，这可通过符合应用要求的方式修改配置文件 “usbd_conf.h” 中的 USBD_DEBUG_LEVEL 实现，该文件位于项目的包含目录下，修改方式为：*

***0**：无日志 / 调试消息*

***1**：使能日志消息*

***2**：使能日志和调试消息*

6.1.6 USB 控制功能

用户应用可受益于 USB 设备中包含的一些 USB 功能，例如：

设备复位 当设备从 USB 收到复位信号时，库会复位，并初始化软硬件上的应用。此函数为中断程序的一部分。

设备挂起 当设备检测到 USB 上的挂起条件，库会停止所有操作，并将系统置于挂起状态（前提条件为 *usbd_conf.c* 文件中使能了低功耗模式管理）。

设备重新开始 当设备检测到 USB 上的重新开始信号时，库会恢复 USB 内核时钟并将系统置于空闲状态 （前提条件为 *usbd_conf.c* 文件中使能了低功耗模式管理）。

6.2 USB 设备库功能

Core 目录包含了 USB 设备库状态机，它由通用串行总线规范版本 2.0 定义。

表 7. USB 设备内核文件

文件	说明
usbd_core (.c, .h)	此文件包含了处理所有 USB 通信和状态机的函数。
usbd_req(.c,.h)	此文件包含了 USB 规范的第 9 章列出的请求实现。
usbd_ctlreq(.c,.h)	此文件处理 USB 事务结果。
usbd_conf_template(.c,.h)	为底层接口文件的模板文件，用户应对其修改并包括在应用文件中
usbd_def(.c, .h)	通用的库定义

Class 目录包含与类实现有关的所有文件，满足了这些类中协议构建规范的要求。

表 8. 类驱动文件

USB 类	文件	说明
大容量存储	usbh_msc (.c,.h)	大容量存储类处理程序
	usbh_msc_bot(.c,.	大容量存储类处理程序
	usbh_msc_scsi(.c,.h)	SCSI 指令
	usbd_msc_data (.c,.h)	重要的查询页面和传感数据
HID 摇杆鼠标	usbh_hid(.c,.h	HID 类状态处理程序
音频扬声器	usbh_audio(.c,.h)	音频类处理程序
音频扬声器	usbh_cdc(.c,.h)	CDC 虚拟通信端口处理程序
自定义 HID	usbd_customhid(.c,.h)	自定义 HID 类处理程序
DFU 类	usbd_dfu(.c,.h)	DFU 类处理程序

表 9. usbd_core (.c,.h) 文件

函数	说明
USBD_StatusTypeDef USBD_Init(USBD_HandleTypeDef *pdev, USBD_DescriptorsTypeDef *pdesc, uint8_t id)	初始化设备库，加载类驱动和用户回调。
USBD_StatusTypeDef USBD_DeInit(USBD_HandleTypeDef *pdev)	取消初始化设备库
USBD_StatusTypeDef USBD_RegisterClass(USBD_HandleTypeDef *pdev, USBD_ClassTypeDef *pclass)	加载类驱动
USBD_StatusTypeDef USBD_Start (USBD_HandleTypeDef *pdev)	开始设备库过程
USBD_StatusTypeDef USBD_Stop (USBD_HandleTypeDef *pdev)	停止设备库过程，释放相关资源。

表 9. usbd_core (.c,.h) 文件 （续）

函数	说明
USBD_StatusTypeDef USBD_LL_SetupStage(USBD_HandleTypeDef *pdev, uint8_t *psetup)	处理来自 ISR 的设置阶段
USBD_StatusTypeDef USBD_LL_DataOutStage(USBD_HandleTypeDef *pdev , uint8_t epnum, uint8_t *pdata)	处理来自 ISR 的数据 OUT 阶段
USBD_StatusTypeDef USBD_LL_DataInStage(USBD_HandleTypeDef *pdev ,uint8_t epnum, uint8_t *pdata)	处理数据 IN 阶段
USBD_StatusTypeDef USBD_LL_Reset(USBD_HandleTypeDef *pdev)	处理来自 ISR 的 USB 复位
USBD_StatusTypeDef USBD_LL_SetSpeed(USBD_HandleTypeDef *pdev, USBD_SpeedTypeDef speed)	设置 USB 内核速度
USBD_StatusTypeDef USBD_LL_Suspend(USBD_HandleTypeDef *pdev)	处理挂起事件
USBD_StatusTypeDef USBD_LL_Resume(USBD_HandleTypeDef *pdev)	处理重新开始事件
USBD_StatusTypeDef USBD_LL_SOF(USBD_HandleTypeDef *pdev);	处理帧事件的开始
USBD_StatusTypeDef USBD_LL_IsoINIncomplete(USBD_HandleTypeDef *pdev, uint8_t epnum)	处理未完成的 ISO IN 事务事件
USBD_StatusTypeDef USBD_LL_IsoOUTIncomplete(USBD_HandleTypeDef *pdev, uint8_t epnum)	处理未完成的 ISO OUT 事务事件
USBD_StatusTypeDef USBD_LL_DevConnected(USBD_HandleTypeDef *pdev)	通知来自 ISR 的设备连接
USBD_StatusTypeDef USBD_LL_DevDisconnected(USBD_HandleTypeDef *pdev)	通知来自 ISR 的设备连接拆断

表 10. usbd_ioreq (.c,.h) 文件功能

函数	说明
USBD_StatusTypeDef USBD_CtlSendData (USBD_HandleTypeDef *pdev, uint8_t *pbuf,uint16_t len)	在控制管道上发送数据
USBD_StatusTypeDef USBD_CtlContinueSendData (USBD_HandleTypeDef *pdev, uint8_t *pbuf, uint16_t len)	在控制管道上继续发送数据。
USBD_StatusTypeDef USBD_CtlPrepareRx (USBD_HandleTypeDef *pdev,uint8_t *pbuf, uint16_t len)	准备内核，以接收控制管道上的数据。

表 10. usbd_ioreq (.c,.h) 文件功能 （续）

函数	说明
USBD_StatusTypeDef USBD_CtlContinueRx (USBD_HandleTypeDef *pdev, uint8_t *pbuf, uint16_t len)	在控制管道上继续接收数据。
USBD_StatusTypeDef USBD_CtlSendStatus (USBD_HandleTypeDef *pdev)	在控制管道上发送长度为零的包。
USBD_StatusTypeDef USBD_CtlReceiveStatus (USBD_HandleTypeDef *pdev)	在控制管道上接收长度为零的包。
uint16_t USBD_GetRxCount (USBD_HandleTypeDef *pdev , uint8_t ep_addr)	返回收到的数据长度

表 11. usbd_ctrlq (.c,.h) 文件功能

函数	说明
USBD_StatusTypeDef USBD_StdDevReq (USBD_HandleTypeDef *pdev, USBD_SetupReqTypedef *req)	处理标准 USB 设备请求。
USBD_StatusTypeDef USBD_StdItfReq (USBD_HandleTypeDef *pdev, USBD_SetupReqTypedef *req)	处理标准 USB 接口请求
USBD_StatusTypeDef USBD_StdEPReq (USBD_HandleTypeDef *pdev, USBD_SetupReqTypedef *req)	处理标准 USB 端点请求
static void USBD_GetDescriptor(USBD_HandleTypeDef *pdev ,USBD_SetupReqTypedef *req)	处理获取描述符请求。
static void USBD_SetAddress(USBD_HandleTypeDef *pdev , USBD_SetupReqTypedef *req)	设置新 USB 设备地址。
static void USBD_SetConfig(USBD_HandleTypeDef *pdev, USBD_SetupReqTypedef *req)	处理设置设备配置请求
static void USBD_GetConfig(USBD_HandleTypeDef *pdev, USBD_SetupReqTypedef *req)	处理获取设备配置请求。
static void USBD_GetStatus(USBD_HandleTypeDef *pdev, USBD_SetupReqTypedef *req)	处理获取状态请求。
static void USBD_SetFeature(USBD_HandleTypeDef *pdev, USBD_SetupReqTypedef *req)	处理设置设备特性请求
static void USBD_ClrFeature(USBD_HandleTypeDef *pdev, USBD_SetupReqTypedef *req)	处理清除设备特性请求。
void USBD_CtlError(USBD_HandleTypeDef *pdev, USBD_SetupReqTypedef *req)	处理控制管道上的 USB 错误。
void USBD_GetString(uint8_t *desc, uint8_t *unicode, uint16_t *len)	将 Ascii 字符串转换为 unicode
static uint8_t USBD_GetLen(uint8_t *buf)	返回字符串长度
void USBD_ParseSetupRequest (USBD_SetupReqTypedef *req, uint8_t *pdata)	将请求缓冲复制到设置结构体。

6.3 USB 设备类接口

在USB设备库初始化期间选择USB类，方法是选择响应的类回调结构体。类结构体如下定义：

USB 类回调结构体

```
typedef struct _Device_cb
{
  uint8_t  (*Init)            (struct _USBD_HandleTypeDef *pdev ,
uint8_t cfgidx);
  uint8_t  (*DeInit)          (struct _USBD_HandleTypeDef *pdev ,
uint8_t cfgidx);
 /* 控制端点 */
  uint8_t  (*Setup)           (struct _USBD_HandleTypeDef *pdev ,
USBD_SetupReqTypedef  *req);
  uint8_t  (*EP0_TxSent)      (struct _USBD_HandleTypeDef *pdev );
  uint8_t  (*EP0_RxReady)     (struct _USBD_HandleTypeDef *pdev );
  /* 特定类端点 */
  uint8_t  (*DataIn)          (struct _USBD_HandleTypeDef *pdev ,
uint8_t epnum);
  uint8_t  (*DataOut)         (struct _USBD_HandleTypeDef *pdev ,
uint8_t epnum);
  uint8_t  (*SOF)             (struct _USBD_HandleTypeDef *pdev);
  uint8_t  (*IsoINIncomplete) (struct _USBD_HandleTypeDef *pdev ,
uint8_t epnum);
  uint8_t  (*IsoOUTIncomplete) (struct _USBD_HandleTypeDef *pdev ,
uint8_t epnum);
  uint8_t  *(*GetHSConfigDescriptor)(uint16_t *length);
  uint8_t  *(*GetFSConfigDescriptor)(uint16_t *length);
  uint8_t  *(*GetOtherSpeedConfigDescriptor)(uint16_t *length);
  uint8_t  *(*GetDeviceQualifierDescriptor)(uint16_t *length);
} USBD_ClassTypeDef;
```

- ***Init***：当设备收到设置配置请求时，会调用此回调；在此函数中类接口使用的端点打开。
- ***DeInit***：当收到清除配置请求时，会调用此回调；此函数会关闭类接口使用的端点。
- ***Setup***：调用此回调可处理特定类设置请求。
- ***EP0_TxSent***：当发送状态完成时，会调用此回调。
- ***EP0_RxSent***：当接收状态完成时，会调用此回调。
- ***DataIn***：调用此回调可执行非控制端点相关数据输入阶段的数据。
- ***DataOut***：调用此回调可执行非控制端点相关数据输出阶段的数据。
- ***SOF***：当收到 SOF 中断时调用此回调；可使用此回调将一些过程与帧开始同步。
- ***IsoINIncomplete***：当最后一个同步 IN 传输未完成时，调用此回调。

- ***IsoOUTIncomplete*** 当最后一个同步 OUT 传输未完成时，调用此回调。
- GetHSConfigDescriptor：此回调返回 HS USB 配置描述符。
- GetFSConfigDescriptor：此回调返回 FS USB 配置描述符。
- GetOtherSpeedConfigDescriptor：此回调返回高速模式中所用类的其它配置描述符。
- GetDeviceQualifierDescriptor：此回调返回设备合格描述符。

库还提供了描述符回调结构体，以允许用户在应用运行时管理设备和字符串描述符。描述符结构体如下定义：

USB 设备描述符结构体

```
typedef struct
{
  uint8_t  *(*GetDeviceDescriptor)( USBD_SpeedTypeDef speed ,
uint16_t *length);
  uint8_t  *(*GetLangIDStrDescriptor)( USBD_SpeedTypeDef speed ,
uint16_t *length);
  uint8_t  *(*GetManufacturerStrDescriptor)( USBD_SpeedTypeDef speed
, uint16_t *length);
  uint8_t  *(*GetProductStrDescriptor)( USBD_SpeedTypeDef speed ,
uint16_t *length);
  uint8_t  *(*GetSerialStrDescriptor)( USBD_SpeedTypeDef speed ,
uint16_t *length);
  uint8_t  *(*GetConfigurationStrDescriptor)( USBD_SpeedTypeDef speed
, uint16_t *length);
  uint8_t  *(*GetInterfaceStrDescriptor)( USBD_SpeedTypeDef speed ,
uint16_t *length);
} USBD_DescriptorsTypeDef;
```

- ***GetDeviceDescriptor***：此回调返回设备描述符。
- ***GetLangIDStrDescriptor***：此回调返回语言 ID 字符串描述符。
- ***GetManufacturerStrDescriptor***：此回调返回制造商字符串描述符。
- ***GetProductStrDescriptor*** 此回调返回产品字符串描述符。
- ***GetSerialStrDescriptor***：此回调返回序列号字符串描述符。
- ***GetConfigurationStrDescriptor***：此回调返回配置字符串描述符。
- ***GetInterfaceStrDescriptor***：此回调返回接口字符串描述符。

注： *USB 设备样例内提供的 usbd_desc.c 文件实现了这些回调实体。*

7 USB 设备库类模块

类模块包含了关于类实现的所有文件。它与这些类中构建协议的规范兼容。下表展示了 MSC、 HID、 DFU、音频、 CDC 类的 USB 设备类文件。

表 12. USB 设备类文件

分类	文件	说明
HID	usbd_hid (.c, .h)	此文件包含了 HID 类回调 （驱动）和与该类相关的配置描述符。
MSC	usbd_msc(.c, .h)	此文件包含了 MSC 类回调 （驱动）和与该类相关的配置描述符。
	usbd_bot (.c, .h)	此文件处理仅批量传输协议。
	usbd_scsi (.c, .h)	此文件处理 SCSI 指令。
	usbd_msc_data (.c,.h)	此文件包含大容量存储设备的重要查询页面和传感数据。
	usbd_msc_storage_template (.c,.h)	此文件提供了一个模板驱动，可允许您实现 MSC 的附加功能。
DFU	usbd_dfu (.c,.h)	此文件包含了 DFU 类回调 （驱动）和与该类相关的配置描述符。
	usbd_dfu_media_template_if (.c,.h)	此文件提供了一个模板驱动，可允许您实现附加的存储接口。
音频	usbd_audio (.c,.h)	此文件包含了 AUDIO 类回调 （驱动）和与该类相关的配置描述符。
	usbd_audio_if_template (.c,.h)	此文件提供了一个模板驱动，可允许您实现音频的附加功能。
CDC	usbd_cdc (.c,.h)	此文件含有 CDC 类回调 （驱动）以及与该类相关的配置描述符。
	usbd_cdc_if_template (.c,.h)	此文件提供了一个模板驱动，可允许您实现 CDC 终端的底层功能。
自定义 HID	usbd_customhid (.c,.h)	此文件包含了自定义 HID 类回调 （驱动）和与该类相关的配置描述符。

7.0.1 HID 类

HID 类实现

此模块管理 HID 类，符合“ 人机接口设备（HID）设备类定义，版本 1.11，2001 年 6 月 27 日 ”。

若需 HID 规范，可在 www.st.com 网站搜索 “hidpage”。

此驱动实现了规范的下列方面：

- 启动接口子类
- 鼠标协议
- 使用页：通用桌面
- 使用：摇杆
- 收集：应用

HID 用户接口

输入报告仅通过**中断进入**管道发送 （HID 鼠标样例）。

必须通过控制管道或中断输出管道，由主机启动特性和输出报告 （自定义 HID 样例）

USBD_HID_SendReport 可被HID鼠标应用使用，来发送HID报告，在这个版本中，HID驱动仅处理 IN 传输。此函数的用法举例如下所示：

```
  static __IO uint32_t counter=0;
  HAL_IncTick();
  /* check Joystick state every 10ms */
  if (counter++ == 10)
  {
    GetPointerData(HID_Buffer);
    /* send data though IN endpoint*/
    if((HID_Buffer[1] != 0) || (HID_Buffer[2] != 0))
    {
      USBD_HID_SendReport(&USBD_Device, HID_Buffer, 4);
    }
    counter =0;
  }
  Toggle_Leds();
}
```

HID 类驱动 API

所有 HID 类驱动 API 都定义于 usbd_hid.c 中，并总结于下表中

表 13. usbd_hid.c,h 文件

函数	说明
static uint8_t USBD_HID_Init (USBD_HandleTypeDef *pdev, uint8_t cfgidx)	初始化 HID 接口，打开所用的端点。
static uint8_t USBD_HID_DeInit (USBD_HandleTypeDef *pdev, uint8_t cfgidx)	解除初始化 HID 层，关闭所用的端点。
static uint8_t USBD_HID_Setup (USBD_HandleTypeDef *pdev, USBD_SetupReqTypedef *req)	处理 HID 特定请求。

表 13. usbd_hid.c,h 文件 （续）

函数	说明
uint8_t USBD_HID_SendReport (USBD_HandleTypeDef *pdev, uint8_t *report, uint16_t len)	发送 HID 报告。

HID栈由调用USBD_HID_Init()来初始化，之后应用必须调用USBD_HID_SendReport()函数以发送 HID 报告。

下列 HID 特定请求通过端点 0 （控制）实现：

```
#define HID_REQ_SET_PROTOCOL          0x0B
#define HID_REQ_GET_PROTOCOL          0x03
#define HID_REQ_SET_IDLE              0x0A
#define HID_REQ_GET_IDLE              0x02
#define HID_REQ_SET_REPORT            0x09
#define HID_REQ_GET_REPORT            0x01
```

IN 端点地址和可发送的最大字节数由这些定义给出：

```
#define HID_EPIN_ADDR                 0x81
#define HID_EPIN_SIZE                 0x04
```

7.0.2 大容量存储类

大容量存储类实现

此模块管理 MSC 类，符合 “ 通用串行总线大容量存储类 （MSC）仅批量传输 （BOT）版本 1.0， 9/ 31, 1999".

此驱动实现了规范的下列方面：

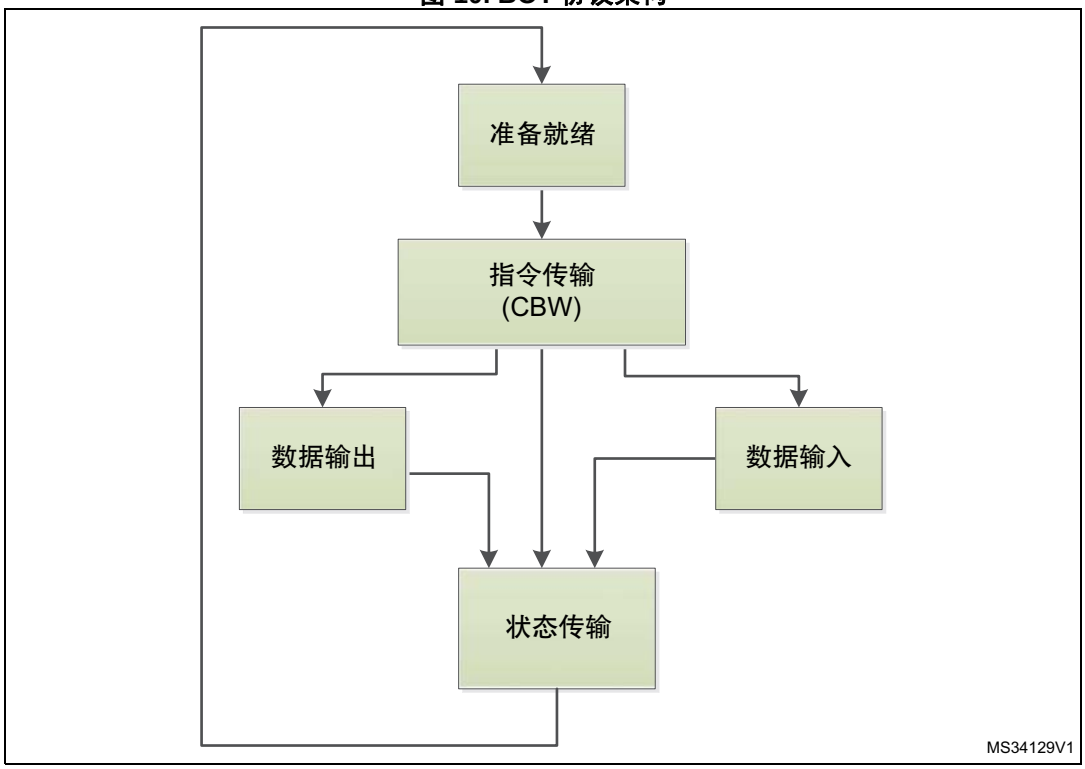
- 仅批量传输协议
- 子类：SCSI 传输指令集 （参考 SCSI 指令集 - 第三章）

USB 大容量存储类构建于仅批量传输 （BOT）之上。它使用 SCSI 传输指令集。

通用 BOT 事务基于一个简单的基本状态机：它从就绪状态（空闲状态）开始，若从主机收到 CBW，则可管理三种情况：

- 数据输出状态：当方向标志设为 “0” 时，设备必须准备接收 CBW 块中 cbw.dDataLength 指示数量的数据。在数据传输结束时，会返回带有剩余数据和 STATUS 字段的 CSW。
- 数据输入状态：当方向标志设为 “1” 时，设备必须准备发送 CBW 块中 cbw.dDataLength 指示数量的数据。在数据传输结束时，会返回带有剩余数据和 STATUS 字段的 CSW。
- 无数据状态：在此情况下，不需要数据阶段：在 CBW 之后立刻发送 CSW 块。

图 10. BOT 协议架构



下表显示了所支持的 SCSI 指令。

表 14. SCSI 指令

指令规范	指令	注释
SCSI	SCSI_PREVENT_REMOVAL, SCSI_START_STOP_UNIT, SCSI_TEST_UNIT_READY, SCSI_INQUIRY, SCSI_READ_CAPACITY10, SCSI_READ_FORMAT_CAPACITY, SCSI_MODE_SENSE6, SCSI_MODE_SENSE10 SCSI_READ10, SCSI_WRITE10, SCSI_VERIFY10	READ_FORMAT_CAPACITY（0x23）为 UFI 指令。

根据 BOT 规范的要求，实现了下列请求：

- **仅批量大容量存储复位（特定类请求）**

此请求用来复位大容量存储设备及其相关的接口。此特定类请求应为来自主机的下一个 CBW 准备好设备。

若要生成 BOT Mass Storage Reset，主机必须在如下默认管道上发送设备请求：

- bmRequestType：到设备的类、接口、主机
- bRequest 字段设为 255 （FFh）
- wValue 字段设为 ‘0’
- wIndex 字段设为接口号
- wLength 字段设为 ‘0’

Get Max LUN （特定类请求）

设备可实现若干逻辑单元，共享通用设备特性。主机使用 bCBWLUN 指示设备的哪个逻辑单元为 CBW 终点。 Get Max LUN 设备请求用于确定设备支持的逻辑单元数。

若要生成 Get Max LUN 设备请求，主机必须在如下默认管道上发送设备请求：

- bmRequestType：到主机的类、接口、设备
- bRequest 字段设为 254 （FEh）
- wValue 字段设为 ‘0’
- wIndex 字段设为接口号
- wLength 字段设为 ‘1’

MSC 内核文件

表 15. usbd_msc （.c,.h）文件

函数	说明
uint8_t USBD_MSC_Init (USBD_HandleTypeDef *pdev, uint8_t cfgidx)	初始化 MSC 接口，打开所用的端点。
uint8_t USBD_MSC_DeInit (USBD_HandleTypeDef *pdev, uint8_t cfgidx)	解除初始化 MSC 层，关闭所用的端点。
uint8_t USBD_MSC_Setup (USBD_HandleTypeDef *pdev, USBD_SetupReqTypedef *req)	处理 MSC 特定请求。
uint8_t USBD_MSC_DataIn (USBD_HandleTypeDef *pdev, uint8_t epnum)	处理 MSC 数据 In 阶段。
uint8_t USBD_MSC_DataOut (USBD_HandleTypeDef *pdev, uint8_t epnum)	处理 MSC 数据 Out 阶段。

表 16. usbd_msc_bot （.c,.h）文件

函数	说明
void MSC_BOT_Init (USBD_HandleTypeDef *pdev)	初始化 BOT 过程与物理媒体。
void MSC_BOT_Reset (USBD_HandleTypeDef *pdev)	复位 BOT 状态机。
void MSC_BOT_DeInit (USBD_HandleTypeDef *pdev)	解除初始化 BOT 过程。

表 16. usbd_msc_bot （.c,.h）文件 （续）

函数	说明
void MSC_BOT_DataIn (USBD_HandleTypeDef *pdev, uint8_t epnum)	处理 BOT 数据 IN 阶段。
void MSC_BOT_DataOut (USBD_HandleTypeDef *pdev, uint8_t epnum)	处理 BOT 数据 OUT 阶段。
static void MSC_BOT_CBW_Decode (USBD_HandleTypeDef *pdev)	解码 CBW 指令，相应地设置 BOT 状态机。
static void MSC_BOT_SendData(USBD_HandleTypeDef *pdev, uint8_t* buf, uint16_t len)	发送请求的数据。
void MSC_BOT_SendCSW (USBD_HandleTypeDef *pdev, uint8_t CSW_Status)	发送指令状态封装。
static void MSC_BOT_Abort (USBD_HandleTypeDef *pdev)	放弃当前传输。
void MSC_BOT_CplClrFeature (USBD_HandleTypeDef *pdev, uint8_t epnum)	完成 Clear Feature 请求。

表 17. usbd_msc_scsi (.c,.h)

函数	说明
int8_t SCSI_ProcessCmd(USBD_HandleTypeDef *pdev, uint8_t lun, uint8_t *params)	处理 SCSI 指令。
static int8_t SCSI_TestUnitReady(USBD_HandleTypeDef *pdev, uint8_t lun, uint8_t *params)	处理 SCSI Test Unit Ready 指令。
static int8_t SCSI_Inquiry(USBD_HandleTypeDef *pdev, uint8_t lun, uint8_t *params)	处理 Inquiry 指令。
static int8_t SCSI_ReadCapacity10(USBD_HandleTypeDef *pdev, uint8_t lun, uint8_t *params)	处理 Read Capacity 10 指令。
static int8_t SCSI_ReadFormatCapacity(USBD_HandleTypeDef *pdev, uint8_t lun, uint8_t *params)	处理 Read Format Capacity 指令。
static int8_t SCSI_ModeSense6 (USBD_HandleTypeDef *pdev, uint8_t lun, uint8_t *params)	处理 Mode Sense 6 指令。
static int8_t SCSI_ModeSense10 (USBD_HandleTypeDef *pdev, uint8_t lun, uint8_t *params)	处理 Mode Sense 10 指令。
static int8_t SCSI_RequestSense (USBD_HandleTypeDef *pdev, uint8_t lun, uint8_t *params)	处理 Request Sense 指令。

表 17. usbd_msc_scsi (.c,.h) （续）

函数	说明
void SCSI_SenseCode (USBD_HandleTypeDef *pdev, uint8_t lun, uint8_t sKey, uint8_t ASC)	装载错误列表中的最后一个错误代码。
static int8_t SCSI_StartStopUnit (USBD_HandleTypeDef *pdev, uint8_t lun, uint8_t *params)	处理 Start Stop Unit 指令。
static int8_t SCSI_Read10 (USBD_HandleTypeDef *pdev, uint8_t lun , uint8_t *params)	处理 Read10 指令。
static int8_t SCSI_Write10 (USBD_HandleTypeDef *pdev, uint8_t lun , uint8_t *params)	处理 Write10 指令。
static int8_t SCSI_Verify10 (USBD_HandleTypeDef *pdev, uint8_t lun , uint8_t *params)	处理 Verify10 指令。
static int8_t SCSI_CheckAddressRange (USBD_HandleTypeDef *pdev, uint8_t lun , uint32_t blk_offset , uint16_t blk_nbr)	检查 LBA 是否位于地址范围之内。
static int8_t SCSI_ProcessRead (USBD_HandleTypeDef *pdev, uint8_t lun)	处理 Burst Read 过程。
static int8_t SCSI_ProcessWrite (USBD_HandleTypeDef *pdev, uint8_t lun)	处理 Burst Write 过程。

磁盘操作结构体定义

```
USBD_StorageTypeDef USBD_DISK_fops = {
  STORAGE_Init,
  STORAGE_GetCapacity,
  STORAGE_IsReady,
  STORAGE_IsWriteProtected,
  STORAGE_Read,
  STORAGE_Write,
  STORAGE_GetMaxLun,
  STORAGE_Inquirydata,
};
```

注： *MicorSD 是库提供的默认媒体接口，但是您可使用所提供的模板文件 usbd_msc_storage_template.c，添加其它媒体 （Flash....）*

MSC 类的存储回调如下添加在用户应用中：USBD_MSC_RegisterStorage(&USBD_Device, &USBD_DISK_fops)。标准查询数据由用户在 STORAGE_Inquirydata 数组内给出。它应定义为：

```
int8_t STORAGE_Inquirydata[] = { /* 36 */
  /* LUN 0 */
  0x00,
  0x80,
  0x02,
  0x02,
  (STANDARD_INQUIRY_DATA_LEN - 5),
  0x00,
  0x00,
  0x00,
  'S', 'T', 'M', ' ', ' ', ' ', ' ', ' ', /* 制造商：8 字节 */
  'P', 'r', 'o', 'd', 'u', 'c', 't', ' ', /* 产品：16 字节 */
  ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ',
  '0', '.', '0','1',                      /* 版本：4 字节 */
};
```

磁盘操作功能

表 18. 函数

函数	说明
int8_t STORAGE_Init (uint8_t lun)	初始化存储媒介。
int8_t STORAGE_GetCapacity (uint8_t lun, uint32_t *block_num, uint16_t *block_size)	返回媒介容量和块大小。
int8_t STORAGE_IsReady (uint8_t lun)	检查媒介是否就绪。
int8_t STORAGE_IsWriteProtected (uint8_t lun)	检查媒介是否写保护。
int8_t STORAGE_Read (uint8_t lun, uint8_t *buf, uint32_t blk_addr, uint16_t blk_len)	从媒介读取数据： blk_address 在扇区单元中给出 blk_len 为要处理的扇区号。
int8_t STORAGE_Write (uint8_t lun, uint8_t *buf, uint32_t blk_addr, uint16_t blk_len)	向媒介写入数据： blk_address 在扇区单元中给出 blk_len 为要处理的扇区号。
int8_t STORAGE_GetMaxLun (void)	返回支持的逻辑单元数。

7.0.3 设备固件升级 （DFU）类

DFU 内核管理了 DFU 类，符合 “ 设备固件升级设备类规范版本 1.1 2004-8-5”。

此内核实现了规范的下列方面：

- 设备描述符管理
- 配置描述符管理
- 枚举为 DFU 设备 （仅在 DFU 模式中）
- 请求管理 （支持 ST DFU 子协议）
- 内存请求管理 （下载 / 上传 / 擦除 / 分离 / GetState / GetStatus）。
- DFU 状态机实现。

注： *ST DFU 子协议与 DFU 协议兼容。它使用子请求管理内存寻址、指令处理、特定内存操作（即内存擦除等等）*

按 DFU 规范要求，仅端点 0 可用于此应用中。

可能添加其它端点和功能到此应用 （即 HID 等等）

可能针对特定用户应用增加或修改这些方面。

此驱动没有实现规范的下述方面 （但有可能修改驱动以管理这些特性）：

– 表现容忍模式

设备固件升级 （DFU）类实现

DFU 事务基于端点 0 （控制端点）的传输。所有请求和状态控制都通过此端点发送 / 接收。

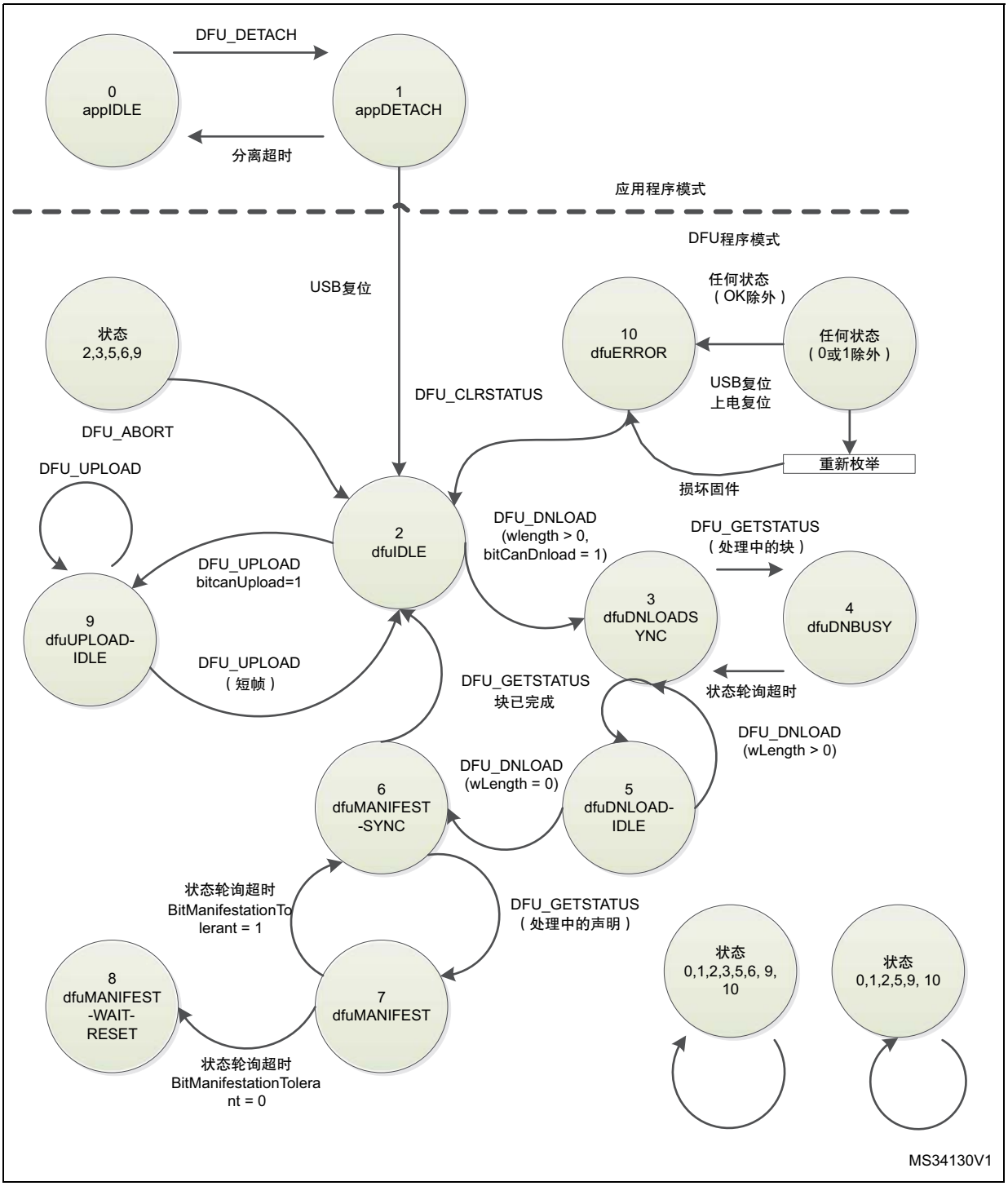
DFU 状态机基于下列状态：

表 19. DFU 状态

状态	状态码
appIDLE	0x00
appDETACH	0x01
dfuIDLE	0x02
dfuDNLOAD-SYNC	0x03
dfuDNBUSY	0x04
dfuDNLOAD-IDLE	0x05
dfuMANIFEST-SYNC	0x06
dfuMANIFEST	0x07
dfuMANIFEST-WAIT-RESET	0x08
dfuUPLOAD-IDLE	0x09
dfuERROR	0x0A

规范文档中描述的状态迁移条件，如下图所示：

图 11. DFU 接口状态转移图



为了在初始化之前保护应用不受虚假访问，DFU 内核的初始状态（启动之后）为 dfuERROR。之后，主机必须在生成任何其它请求之前清除此状态 （通过发送 DFU_CLRSTATE 请求）。

DFU 内核管理了所有支持的请求。

表 20. 支持的请求

请求	代码	详细信息
DFU_DETACH	0x00	当 *bmAttributes* 中的第 3 位 （*WillDetach* 位）置位，设备收到此请求时，设备会在总线上生成分离 - 连接序列。
DFU_DNLOAD	0x01	通过 *DFU_DNLOAD* 特定类请求发起的控制 - 写传输，下载固件映像。
DFU_UPLOAD	0x02	上传的目的是提供获取和存储设备固件的容量。
DFU_GETSTATUS	0x03	主机利用 *DFU_GETSTATUS* 请求与设备同步。
DFU_CLRSTATUS	0x04	收到 *DFU_CLRSTATUS* 后，设备设置 OK 状态，转移至 *dfuIDLE* 状态。
DFU_GETSTATE	0x05	此请求会征求关于设备状态的报告。
DFU_ABORT	0x06	*DFU_ABORT* 请求使主机从特定状态离开，返回到 *DFU_IDLE* 状态。

每个到控制端点的传输都可分为两类：

数据传输：这些传输用于：

- 从设备得到一些数据 （DFU_GETSTATUS、 DFU_GETSTATE 和 DFU_UPLOAD）。
- 或者，向设备发送数据 （DFU_DNLOAD）。

无数据传输：这些传输被用来从主机到设备发送控制请求 （DFU_CLRSTATUS、DFU_ABORT 和 DFU_DETACH）。

设备固件升级 （DFU）类内核文件

usbd_dfu (.c, .h)

此驱动为主要的 DFU 内核。它允许管理所有 DFU 请求和状态机。它不直接处理存储媒体（由底层驱动管理）。

表 21. usbd_dfu （.c,.h）文件

函数	说明
static uint8_t USBD_DFU_Init (USBD_HandleTypeDef *pdev, uint8_t cfgidx);	初始化 DFU 接口。
static uint8_t USBD_DFU_DeInit (USBD_HandleTypeDef *pdev, uint8_t cfgidx);	解除初始化 DFU 层。
static uint8_t USBD_DFU_Setup (USBD_HandleTypeDef *pdev, USBD_SetupReqTypedef *req);	处理 DFU 请求解析。
static uint8_t USBD_DFU_EP0_TxReady (USBD_HandleTypeDef *pdev);	处理 DFU 控制端点数据 IN 阶段。
static uint8_t USBD_DFU_EP0_RxReady (USBD_HandleTypeDef *pdev);	处理 DFU 控制端点数据 OUT 阶段。
static uint8_t* USBD_DFU_GetUsrStringDesc (USBD_HandleTypeDef *pdev, uint8_t index , uint16_t *length);	管理内存接口字符串描述符的传输。

表 21. usbd_dfu （.c,.h）文件 （续）

函数	说明
static void DFU_Detach (USBD_HandleTypeDef *pdev, USBD_SetupReqTypedef *req);	处理 DFU DETACH 请求。
static void DFU_Download (USBD_HandleTypeDef *pdev, USBD_SetupReqTypedef *req);	处理 DFU DNLOAD 请求。
static void DFU_Upload (USBD_HandleTypeDef *pdev, USBD_SetupReqTypedef *req);	处理 DFU UPLOAD 请求。
static void DFU_GetStatus (USBD_HandleTypeDef *pdev);	处理 DFU GETSTATUS 请求。
static void DFU_ClearStatus (USBD_HandleTypeDef *pdev);	处理 DFU CLRSTATUS 请求。
static void DFU_GetState (USBD_HandleTypeDef *pdev);	处理 DFU GETSTATE 请求。
static void DFU_Abort (USBD_HandleTypeDef *pdev);	处理 DFU ABORT 请求。
static void DFU_Leave (USBD_HandleTypeDef *pdev);	处理子协议 DFU 离开 DFU 模式请求 （离开 DFU 模式，复位设备，跳转至用户加载的代码）。

注： *内部 Flash 存储器为库提供的默认存储器。但是您可使用提供的模板文件 `usbd_dfu_media_template.c` 添加其它存储器*

怎样使用驱动：

- 使用文件 `usbd_conf.h`，您可配置：
  - 支持的媒体 （存储器）个数 （定义 USBD_DFU_MAX_ITF_NUM）。
  - 应用默认地址 （映像代码应被加载之处）：定义 USBD_DFU_APP_DEFAULT_ADD。

调用 USBD_DFU_Init() 函数，初始化所有存储器接口和 DFU 状态机。

所有控制 / 请求操作都通过控制端点 0 执行，使用的函数为：USBD_DFU_Setup() 和 USBD_DFU_EP0_TxReady()。根据所生成的 DFU 请求，可使用这些函数调用每个存储器接口回调 （读 / 写 / 擦除 / 获取状态 ...）。这些操作不需要用户干预。

若要关闭通信，请调用 USBD_DFU_DeInit() 函数。

注： *当 DFU 应用启动时，默认 DFU 状态为* DFU_STATE_ERR0R*。设置此状态的目的是在有正确的配置之前，保护应用不受虚假操作的影响。*

7.0.4 音频类

此驱动管理音频类，符合 “ 音频设备 USB 设备类定义 V1.0 1998-3-18”。

此驱动实现了规范的下列方面：

- 设备描述符管理
- 配置描述符管理
- 标准 AC 接口描述符管理
- 1 个音频流接口 （单通道、 PCM、立体声模式）
- 1 个音频流端点
- 1 个音频终端输入 （1 个通道）
- 音频特定类 AC 接口
- 音频特定类 AS 接口
- 音频控制请求：仅支持 SET_CUR 和 GET_CUR 请求 （静音）
- 音频特性单元 （限为静音控制）
- 音频同步类型：异步
- 单固定音频采样率 （可在 *usbd_conf.h* 文件中配置）

注： *音频类基于 USB 规范 1.0，因此仅支持低速模式和全速模式，不支持高速传输。请参考 “ 音频设备 USB 设备类定义 V1.0 1998-3-18” 以获取更详细信息。*

可针对特定用户应用增加或修改这些方面。

此驱动没有实现规范的下述方面 （但有可能修改驱动以管理这些特性）：

- 音频控制端点管理
- SET_CUR 和 GET_CUR 以外的音频控制请求
- 音频控制请求的抽象层 （仅管理静音功能）
- 音频同步类型：自适应
- 音频压缩模块和接口
- MIDI 接口和模块
- 混合 / 选择 / 处理 / 扩展单元 （所列单元限为静音控制）
- 其它任何特定应用模块
- 复合及可变的音频采样率
- 音频输出流端点 / 接口 （麦克风）

音频类实现

音频传输基于同步端点事务。音频控制请求还通过控制端点 （端点 0）管理。

在每一帧传输的音频数据包必须在此帧时间之内 （下一帧之前）处理掉。音频质量取决于数据传输和数据处理之间的同步情况。此驱动依赖所交付 I2S 时钟的精度，实现简单的同步机制。在每帧开始时，驱动会检查是否正确执行了前一帧的处理，若仍在进行则将其停止。为防止任何数据覆盖，主要使用了两种保护方式：

- 在 USB 缓冲和输出设备寄存器 （I2S）之间使用 DMA 进行数据传输。
- 使用多缓冲存储从 USB 接收的数据。

基于此机制，如果时钟精度或处理速率不够高，则会导致较差的音频质量。

此机制可通过实现更灵活的音频流控制来加强，如 USB 反馈模式、动态音频时钟纠正，或使用 SOF 事件生成 / 控制音频时钟。

驱动还支持基本音频控制请求。为简化驱动，仅实现了两个请求。然而，仅需稍微修改音频内核驱动即可支持其他请求。

表 22. 音频控制请求

请求	支持	意义
SET_CUR	有	设置静音模式开或关 （也可更新为设置音量 ...）。
SET_MIN	无	NA
SET_MAX	无	NA
SET_RES	无	NA
SET_MEM	无	NA
GET_CUR	有	获取静音模式状态 （也可更新，以得到音量 ...）。
GET_MIN	无	NA
GET_MAX	无	NA
GET_RES	无	NA
GET_MEM	无	NA

音频内核文件

usbd_audio (.c, .h)

此驱动为音频内核。它管理音频数据传输并控制请求。它不直接处理音频硬件 （由底层驱动管理）。

表 23. usbd_audio_core （.c,.h）文件

函数	说明
static uint8_t USBD_AUDIO_Init (USBD_HandleTypeDef *pdev, uint8_t cfgidx);	初始化音频接口。
static uint8_t USBD_AUDIO_DeInit (USBD_HandleTypeDef *pdev, uint8_t cfgidx);	解除初始化音频接口。
static uint8_t USBD_AUDIO_Setup (USBD_HandleTypeDef *pdev, USBD_SetupReqTypedef *req);	处理音频控制请求解析。
static uint8_t USBD_AUDIO_EP0_RxReady (USBD_HandleTypeDef *pdev);	处理音频控制请求数据。

表 23. usbd_audio_core （.c,.h）文件 （续）

函数	说明
static uint8_t USBD_AUDIO_DataIn (USBD_HandleTypeDef *pdev, uint8_t epnum);	处理音频输入数据阶段。
static uint8_t USBD_AUDIO_DataOut (USBD_HandleTypeDef *pdev, uint8_t epnum);	处理音频输出数据阶段。
static uint8_t USBD_AUDIO_SOF (USBD_HandleTypeDef *pdev);	处理 SOF 事件 （数据缓冲更新和同步）。
static void AUDIO_REQ_GetCurrent(USBD_HandleTypeDef *pdev, USBD_SetupReqTypedef *req);	处理 GET_CUR 音频控制请求。
static void AUDIO_REQ_SetCurrent(USBD_HandleTypeDef *pdev, USBD_SetupReqTypedef *req);	处理 SET_CUR 音频控制请求。

底层硬件接口通过它们相应的驱动结构体管理：

```
typedef struct
{
    int8_t  (*Init)         (uint32_t  AudioFreq, uint32_t Volume,
uint32_t options);
    int8_t  (*DeInit)       (uint32_t options);
    int8_t  (*AudioCmd)     (uint8_t* pbuf, uint32_t size, uint8_t
cmd);
    int8_t  (*VolumeCtl)    (uint8_t vol);
    int8_t  (*MuteCtl)      (uint8_t cmd);
    int8_t  (*PeriodicTC)   (uint8_t cmd);
    int8_t  (*GetState)     (void);
}USBD_AUDIO_ItfTypeDef;
```

每个音频硬件接口驱动都应该提供一个类型为USBD_AUDIO_ItfTypeDef的结构体指针。（下边的章节会写了车该结构体所指向的函数和变量）。如果给定的存储器接口不支持某功能，则相应的字段置为 NULL 值。

usbd_audio_if (.c, .h)

此驱动管理底层音频硬件。 *usbd_audio_if.c/.h* 驱动管理音频输出接口 （从 USB 到音频扬声器 / 耳机）。用户可调用底层编解码器驱动 （即 stm324xg_eval_audio.*c/.h*）以进行基本的音频操作 （播放 / 暂停 / 音量控制 ...）。

此驱动提供了结构体指针：

extern USBD_AUDIO_ItfTypeDef USBD_AUDIO_fops;

表 24. usbd_audio_if （.c,.h）文件

函数	说明
static int8_t Audio_Init(uint32_t AudioFreq, uint32_t Volume, uint32_t options);	初始化音频接口。
static int8_t Audio_DeInit(uint32_t options);	解除初始化音频接口，释放所用的资源。
static int8_t Audio_PlaybackCmd(uint8_t* pbuf, uint32_t size, uint8_t cmd);	处理音频播放器指令 （播放、暂停 …）
static int8_t Audio_VolumeCtl(uint8_t vol);	处理音频播放器音量控制。
static int8_t Audio_MuteCtl(uint8_t cmd);	处理音频播放器静音状态。
static int8_t Audio_PeriodicTC(uint8_t cmd);	Handles the end of current packet transfer (not needed for the current version of the driver).
static int8_t Audio_GetState(void);	返回驱动音频播放器的当前状态 （正在播放 / 已暂停 / 错误 …）。

注： *usbd_audio_if_template （.c,.h）文件提供了模板驱动，它允许您为您的音频应用实现附加功能。*

通过下列状态列表来获得当前音频播放器的状态：

表 25. 音频播放器状态

状态	代码	说明
AUDIO_CMD_START	0x01	音频播放器已初始化并就绪。
AUDIO_CMD_PLAY	0x02	音频播放器当前正在播放。
AUDIO_CMD_STOP	0x04	音频播放器已停止。

怎样使用此驱动：

此驱动使用了硬件驱动的抽象层（即 HW 编解码器、I2S 接口、I2C 控制接口 ...）。此抽象通过底层 （即 *usbd_audio_if.c*）执行，您可根据您的应用以及相应的硬件对其修改。

若要使用此驱动：

通过文件 *usbd_conf.h*，您可配置：

- 音频采样率 （定义 USBD_AUDIO_FREQ）

在启动时调用函数 USBD_AUDIO_Init()，配置所有必要的固件和硬件部件 （特定应用的硬件配置函数也由此函数调用）硬件部件由底层接口 （即 *usbd_audio_if.c*）管理，可由用户根据应用需要修改。

整个传输由下述函数管理 （用户不需为输出传输调用任何函数）：

- usbd_audio_DataIn()和usbd_audio_DataOut()使用收到的或发送的数据更新音频缓冲。对于输出传输，当收到数据时，它们会被直接复制到音频缓冲中，写缓冲 （wr_ptr）增加。

音频控制请求由函数 USBD_AUDIO_Setup() 和 USBD_AUDIO_EP0_RxReady() 管理。这些函数会将音频控制请求路由至底层 （即 *usbd_audio_if.c*） 。在当前版本中，仅管理了 SET_CUR 和 GET_CUR 请求，仅用于静音控制。

音频的已知限制

- 如果配置了低音频采样率（将USBD_AUDIO_FREQ定义为24 kHz以下），则在暂停/重新开始 / 停止操作时可能导致噪声问题。这是由于在停止 I2S 时钟和发送静音指令到外部编解码器之间的软件时序调节。
- 支持的音频采样率为: 96 kHz 到 24 kHz（此驱动不支持非整数 kHz 值，例如 11.025 kHz、22.05 kHz 或 44.1 kHz）。对于 1000 Hz 的整数倍数频率，主机会在每帧 （1 ms）发送整数个字节。当频率不是 1000Hz 的整数倍时，主机会在每帧发送非整数个字节。实际上，这是通过发送不同大小的帧来管理的 （即对于 22.05 kHz，主机将发送 19 帧的 22 字节和一帧的 23 字节）。音频内核不会管理此大小差别，多余的字节会一直被忽略。建议设置高采样率和标准采样率，以得到最好的音频质量 （即 96 kHz 或 48 kHz）。请注意，最大允许的音频频率为 96 kHz（此限制的原因是评估板上使用的编解码器。STM32 I2S 单元可达到 192 kHz）。

7.0.5 通信设备类 （CDC）

此驱动管理了 “ 通信设备通用串行总线类定义版本 1.2 2007 年 11 月 16 日 ” 和 “PSTN 设备通用串行总线通信类子类规范版本 1.2 2007 年 2 月 9 日 ” 子协议规范。

此驱动实现了规范的下列方面：

- 设备描述符管理
- 配置描述符管理
- 枚举为具有 2 个数据端点 （IN 和 OUT）的 CDC 设备和一个指令端点 （IN）
- 请求管理 （如规范 6.2 节所述）
- 抽象控制模型兼容
- 联合体功能收集 （使用 1 个 IN 端点控制）
- 数据接口类

注 : *对于抽象控制模型，此内核仅能向底层调度 （即 usbd_cdc_vcp.c/.h）发送请求，底层调度应管理每个请求并执行相应行为。*

可能针对特定用户应用增加或修改这些方面。

此驱动没有实现规范的下述方面 （但有可能修改驱动以管理这些特性）：

- 任何与通信类相关的特定类方面都应由用户应用管理。
- PSTN 以外的所有通信类 都未被管理。

通信

CDC 内核使用两种端点 / 传输类型：

- 用于数据传输的批量端点 （1 个 OUT 端点和 1 个 IN 端点）
- 用于通信控制的中断端点 （CDC 请求； 1 个 IN 端点）

对于 IN 和 OUT 传输，数据传输的管理不同：

数据 IN 传输管理 （从设备到主机）

取决于主机请求，周期性管理数据传输 （设备指定包请求之间的间隔） 。因此，使用了一个循环静态缓冲以存储设备终端发送的数据 （即，在虚拟 COM 端口终端情况下为 USART）。

数据 OUT 传输管理 （从主机到设备）

总的来说，USB 比输出终端快很多（即，USART 最大比特率为 115.2 Kbps，USB 比特率在全速模式时为 12 Mbps，在高速模式时为 480 Mbps）。因此，在发送新的包之前，主机必须等待设备处理完主机之前发送的数据。因此，当从主机收到包后，不需要循环数据缓冲：驱动会调用底层 OUT 传输函数，在允许 OUT 端点上的新传输之前，等待此函数完成 （与此同时， OUT 包会被 NACK）。

指令请求管理

在此驱动中，控制端点 （端点 0）用于管理控制请求。但是数据中断端点也可能被用于指令管理。如果请求数据的大小没有超过 64 字节，则端点 0 足够管理这些请求。

CDC 驱动不管理指令请求解析。相反，它使用请求码、长度和数据缓冲调用底层驱动控制管理函数。然后此函数应该解析请求，执行所需的动作。

通信设备类 （CDC）内核文件

usbd_cdc (.c, .h)

此驱动为 CDC 内核。它管理 CDC 数据传输并控制请求。它不直接处理 CDC 硬件 （由底层驱动管理）。

表 26. usbd_cdc （.c,.h）文件

函数	说明
static uint8_t USBD_CDC_Init (USBD_HandleTypeDef *pdev, uint8_t cfgidx);	初始化 CDC 接口。
static uint8_t USBD_CDC_DeInit (USBD_HandleTypeDef *pdev, uint8_t cfgidx);	解除初始化 CDC 接口。
static uint8_t USBD_CDC_Setup (USBD_HandleTypeDef *pdev, USBD_SetupReqTypedef *req);	处理 CDC 控制请求。
static uint8_t USBD_CDC_EP0_RxReady (USBD_HandleTypeDef *pdev);	处理 CDC 控制请求数据。
static uint8_t USBD_CDC_DataIn (USBD_HandleTypeDef *pdev, uint8_t epnum);	处理 CDC IN 数据阶段。

表 26. usbd_cdc （.c,.h）文件 （续）

函数	说明
static uint8_t USBD_CDC_DataOut (USBD_HandleTypeDef *pdev, uint8_t epnum);	处理 CDC Out 数据阶段。
uint8_t USBD_CDC_RegisterInterface (USBD_HandleTypeDef *pdev, USBD_CDC_ItfTypeDef *fops)	添加 CDC 接口类
uint8_t USBD_CDC_SetTxBuffer (USBD_HandleTypeDef *pdev, uint8_t *pbuff, uint16_t length)	设置应用 TX 缓冲
uint8_t USBD_CDC_SetRxBuffer (USBD_HandleTypeDef *pdev, uint8_t *pbuff)	设置应用 RX 缓冲
uint8_t USBD_CDC_TransmitPacket(USBD_HandleTypeDef *pdev)	发送传输完成回调
uint8_t USBD_CDC_ReceivePacket(USBD_HandleTypeDef *pdev)	接收传输完成回调

底层硬件接口通过它们相应的驱动结构体管理：

```
typedef struct _USBD_CDC_Itf
{
  int8_t (* Init)          (void);
  int8_t (* DeInit)        (void);
  int8_t (* Control)       (uint8_t, uint8_t * , uint16_t);
  int8_t (* Receive)       (uint8_t *, uint32_t *);

}USBD_CDC_ItfTypeDef;
```

每个硬件接口驱动都应该提供一个类型为 USBD_CDC_ItfTypeDef 的结构体指针。下节中列出了此结构体指向的函数。

如果给定的存储器接口不支持某功能，则相应的字段置为 NULL 值。

注： *为得到最佳性能，建议计算下列参数需要的值 （它们全都可通过 usbd_cdc.h 和 usbd_cdc_interface.h 文件中的定义配置）：*

表 27. 可配置 CDC 参数

定义	参数	典型值	
		全速	高速
CDC_DATA_HS_IN_PACKET_SIZE /CDC_DATA_FS_IN_PACKET_SIZE	每个 IN 数据包的大小	64	512
CDC_DATA_HS_OUT_PACKET_SIZE/CDC_DATA_FS_OUT_PACKET_SIZE	每个 OUT 数据包的大小	64	512
APP_TX_DATA_SIZE	OUT 数据传输循环临时缓冲的总大小。	2048	2048
APP_RX_DATA_SIZE	IN 数据传输循环临时缓冲的总大小。	2048	2048

usbd_cdc_interface (.c, .h)

此驱动可以是用户应用的一部分。它不在库中提供，但可使用 **usbd_cdc_if_template** ***（.c, .h）*** 模板来构建它，且为 USART 接口提供了一个样例。它管理底层 CDC 硬件。*usbd_cdc_interface.c/.h* 驱动管理终端接口配置和通信（即 USART 接口配置和数据收发）。

此驱动提供了结构体指针：

```
USBD_CDC_ItfTypeDef USBD_CDC_fops =
{
  CDC_Itf_Init,
  CDC_Itf_DeInit,
  CDC_Itf_Control,
  CDC_Itf_Receive
};
```

表 28. usbd_cdc_interface （.c,.h）文件

函数	说明
static int8_t CDC_Itf_Init (void);	初始化底层 CDC 接口。
static int8_t CDC_Itf_DeInit (void);	解除初始化底层 CDC 接口。
static int8_t CDC_Itf_Control (uint8_t cmd, uint8_t* pbuf, uint16_t length);	处理 CDC 控制请求解析和执行。
static int8_t CDC_Itf_Receive (uint8_t* pbuf, uint32_t *Len);	处理从 USB 主机到底层终端的 CDC 数据接收（OUT 传输）。

为加速 IN/OUT 传输的数据管理，底层驱动（*usbd_cdc_interface.c/.h*）使用全局变量：

表 29. usbd_cdc_xxx_if.c/.h 使用的变量

变量	使用
uint8_t UserRxBuffer[APP_RX_DATA_SIZE]	在此缓冲中写入 CDC 接收的数据。这些数据将在 CDC 内核函数中通过 USB IN 端点发送。
uint32_t UserTxBufPtrOu	当在缓冲 UserRxBuffer 中写入接收的数据时，应增加此指针或将它返回到开始地址。
uint8_t UserTxBuffer[APP_TX_DATA_SIZE]	在此缓冲中写入 CDC 接收的数据。这些数据将在 CDC 内核函数中通过 USB OUT 端点发送。
UserTxBufPtrIn	当从 USART 接收数据时，应增加此指针或将它返回到开始地址。

怎样使用此驱动：

此驱动使用了硬件驱动的抽象层（即 USART 控制接口 ...）。此抽象通过底层（即 *stm32fxxx_hal_msp.c*）执行，您可根据您的应用以及相应的硬件对其修改。

若要使用此驱动：

通过文件 *usbd_cdc.h* 和 *usbd_cdc_interface.h*，您可配置：

- 数据 IN 和 OUT 以及指令包大小（定义 CDC_DATA_XX_IN_PACKET_SIZE、CDC_DATA_XX_OUT_PACKET_SIZE）
- IN/OUT 数据传输的临时循环缓冲大小（定义 APP_RX_DATA_SIZE and APP_TX_DATA_SIZE）。
- 设备字符串描述符。

在启动时调用函数 USBD_CDC_Init()，配置所有必要的固件和硬件部件（特定应用的硬件配置函数也由此函数调用）。硬件部件由底层接口（即 *usbd_cdc_interface.c*）管理，可由用户根据应用需要修改。

CDC IN 和 OUT 数据传输由两个函数管理：

- USBD_CDC_SetTxBuffer 应在每次有数据（或一定量的数据）从硬件终端到 USB 主机发送时由用户应用调用。
- USBD_CDC_SetRxBuffer 在每次从 USB 主机发送缓冲时由 CDC 内核调用，应发送到硬件终端。此函数应仅在缓冲中所有数据都发送完之后退出（CDC 内核之后会阻塞所有发送来的 OUT 包，直到此函数完成对前一个包的处理）。

CDC 控制请求应由函数 CDC_Itf_Control() 处理。每次收到主机来的请求时，都会调用此函数，且它所有相关的数据都可用。此函数应该解析请求，执行所需的动作。

若要关闭通信，请调用 USBD_CDC_DeInit() 函数。这会关闭使用的端点，并调用底层解除初始化的函数。

CDC 的已知限制

当此驱动与 OTG HS 内核共同使用时，使能 DMA 模式（定义 *usb_conf.h* 文件中的 USB_OTG_HS_INTERNAL_DMA_ENABLED）会导致数据只能以 4 字节的倍数发送。这是因为 USB DMA 不允许从字不对齐的地址发送数据。对于这个特定应用，建议除非需要，不要使能这一选项。

7.0.6 添加自定义类

本节将解释怎样基于已有的 USB 类，创建一个新的自定义类。

创建一个新的自定义类需要一些步骤：

- 用户必须如*第 6.3 章节*中所述，添加 USBD_CustomClass_cb（以接收各种 USB 总线事件），位置为 **Class/Template 目录下的 usbd_template.c/.h**。此模板包含需适配应用需要的所有函数，也可能需要勇于实现任何类型的 USB 设备类。
- 自定义描述符：必须配置主机取得的描述符，以按照下述应用类设备规范描述设备。下表并不完整，但它给出了可能需要的各种描述符概览：
- 标准设备描述符
- 标准配置描述符
- 所实现类的标准接口描述符
- IN 和 OUT 端点的标准端点描述符
- 取决于这里的用户应用，固件必须配置 STM32，以使能 USB 传输 （同步、批量、中断或控制）。下面是详细说明：
- 在 DataIn 和 DataOut 函数中，用户可实现内部协议或状态机
- 在 *Setup* 中；实现特定类请求。配置描述符被作为数组添加，传递到 USB 设备库。
- 通过 GetConfigDescriptor 函数，它应该返回指向 USB 配置描述符的指针及其长度。
- 作为 IsoINIncomplete 和 IsoOUTIncomplete 添加的附加功能，最终可能会用于处理未完成的同步传输 （若需更多信息，请参考 *USB 音频设备样例*）。
- EP0_TxSent 和 EP0_RxReady 最终可能在应用需要处理零长度包之前的事件时使用（参见 *DFU 样例*）。
- 内存分配过程：使用 malloc （USBD_malloc）分配内存：
- USBD_malloc(sizeof (USBD_**CUSTOM_CLASS**_HandleTypeDef))：这会为类结构体动态分配内存

7.0.7 库大小优化

在本节中，我们回顾一些基本技巧，涉及怎样优化 USB 设备库之上开发的应用大小。

缩小 USB 样例是一个重要的目标，尤其对于具有较少 Flash/RAM 内存的 STM32 产品 （如 STM32 L0 和 F0）来说尤其重要。

- **降低堆和栈大小设置 （在连接 （linker）文件中）**

栈为程序存储的内存区，例如：

- 本地变量
- 返回地址
- 函数参数
- 编译器临时量
- 中断上下文

如果您的连接器配置保留了大量的堆和栈，而您的应用并不需要，您可以决定其合适的大小。

- **尽可能使用局部变量，而不用全局变量**

如果一个变量仅在一个函数中使用，那么应在函数内将其声明为局部变量。

- **常量应在闪存中分配**

建议将永不变化的所有常量全局变量分配至只读区。例如，使用 C 关键字 “const” 将 USB 描述符声明为常量。

```
/* USB Standard Device Descriptor */
const uint8_t USBD_DeviceDesc[USB_LEN_DEV_DESC]= {
  0x12,                       /* bLength */
  USB_DESC_TYPE_DEVICE,       /* bDescriptorType */
  0x00,                       /* bcdUSB */
  0x02,
  0x00,                       /* bDeviceClass */
  0x00,                       /* bDeviceSubClass */
  0x00,                       /* bDeviceProtocol */
  USB_MAX_EP0_SIZE,           /* bMaxPacketSize */
  LOBYTE(USBD_VID),           /* idVendor */
  HIBYTE(USBD_VID),           /* idVendor */
  LOBYTE(USBD_PID),           /* idVendor */
  HIBYTE(USBD_PID),           /* idVendor */
  0x00,                       /* bcdDevice rel. 2.00 */
  0x02,
  USBD_IDX_MFC_STR,           /* Index of manufacturer string */
  USBD_IDX_PRODUCT_STR,       /* Index of product string */
  USBD_IDX_SERIAL_STR,        /* Index of serial number string */
  USBD_MAX_NUM_CONFIGURATION  /* bNumConfigurations */
}; /* USB_DeviceDescriptor */
```

- **使用静态内存分配，而不是 malloc**

USB 设备库为类处理结构体使用动态内存分配以支持多实例 （在双核工作情况下），这意味着我们可以将同一 USB 类用于 USB 的两个实例 （HS 和 FS）。

使用动态分配的第二个原因是当 USB 不再使用时，可释放内存。

然而，动态内存分配会增加一些空间上的开销，主要是 ROM 内存。因此，对于低内存 STM32 设备，当不需要多实例支持时，建议使用静态分配。在这种情况下，需要声明静态缓冲，大小为类处理结构体的大小。

下面是一个实现样例：

1. 在 usbd_conf.h 文件中，定义内存静态分配和程序；
   USBD_static_malloc() **和** *USBD_static_free()*

```
#define MAX_STATIC_ALLOC_SIZE 4 /* HID类结构体大小 */
#define USBD_malloc      (uint32_t *)USBD_static_malloc
#define USBD_free          USBD_static_free
```

2. 在 usbd_conf.c 文件中如下实现：

```
/**
  * @brief  static single allocation.
  * @param  size: size of allocated memory
  * @retval None
  */
void *USBD_static_malloc(uint32_t size)
{
  static uint32_t mem[MAX_STATIC_ALLOC_SIZE];
  return mem;
}

/**
  * @brief  Dummy memory free
  * @param  *p pointer to allocated  memory address
  * @retval None
  */
void USBD_static_free(void *p)
{

}
```

8 常见问题

1. 怎样在运行时修改设备和字符串描述符？

在 *usbd_desc.c* 文件中，可使用 Get Descriptor 回调修改设备和字符串相关的描述符。应用可使用 switch case 语句，返回应用索引相关的正确描述符缓冲。

2. 大容量存储类怎样支持超过一个逻辑单元 （LUN）？

在 *usbd_msc_storage_template.c* 文件中，定义了需要使用物理媒介的所有 API。每个函数都带有 “LUN” 参数以选择寻址媒介。

所支持的LUN数可使用定义STORAGE_LUN_NBR更改，它在*usbd_msc_storage_xxx.c* 文件中 （其中， xxx 为使用的媒介）。

对于查询数据， STORAGE_Inquirydata 缓冲对于每个 LUN 都包含了标准查询数据。

举例：使用了 2 个 LUN

```
const int8_t  STORAGE_Inquirydata[] = {

  /* LUN 0 */
  0x00,
  0x80,
  0x02,
  0x02,
  (USBD_STD_INQUIRY_LENGTH - 5),
  0x00,
  0x00,
  0x00,
  'S', 'T', 'M', ' ', ' ', ' ', ' ', ' ', /* Manufacturer:
    8 bytes */
  'm', 'i', 'c', 'r', 'o', 'S', 'D', ' ', /* Product:
    16 bytes */
  'F', 'l', 'a', 's', 'h', ' ', ' ', ' ',
  '1', '.', '0' ,'0', /* Version: 4 Bytes */

  /* LUN 0 */
  0x00,
  0x80,
  0x02,
  0x02,
  (USBD_STD_INQUIRY_LENGTH - 5),
  0x00,
  0x00,
  0x00,
  'S', 'T', 'M', ' ', ' ', ' ', ' ', ' ', /* Manufacturer:
    8 bytes */
  'N', 'a', 'n', 'd', ' ', ' ', ' ', ' ', /* Product:
    16 Bytes */
```

```
'F', 'l', 'a', 's', 'h', ' ', ' ', ' ',
'1', '.', '0' ,'0', /* Version: 4 Bytes */
 };
```

3. 端点地址在哪里定义？

端点地址在类驱动的头文件中定义。例如，对于 MSC 演示，IN/OUT 端点地址如下定义在 usbd_msc.h 文件中：

```
#define MSC_EPIN_ADDR        0x81 For Endpoint 1 IN
#define MSC_EPOUT_ADDR       0x01 For Endpoint 1 OUT
```

4. USB 设备库可配置为在高速或全速模式运行吗？

是的，库可处理 USB OTG HS 和 USB OTG FS 内核，如果 USB OTG FS 内核仅能工作于全速模式， USB OTG HS 可工作于高速或全速模式。

若要选择合适的 USB 内核，用户必须在编译预处理器内增加下列宏定义 （在样例提供的预配置项目中已经完成）：

- "USE_USB_HS" 当使用 USB 高速 （HS）内核时
- "USE_USB_FS" 当使用 USB 全速 （FS）内核时
- "USE_USB_HS" 和 "USE_USB_HS_IN_FS" 当在 FS 模式使用 USB 高速（HS）内核时

5. 怎样在 USB 设备类驱动中更改所用的端点？

若要更改端点或增加一个新端点，请：

a) 使用 USBD_LL_OpenEP() 执行端点初始化。

b) 对于在 USBD_LL_Init() 函数中使用这些 API 的 *usb_conf.c* 文件，配置新定义端点的 TX 或 Rx FIFO 大小。

- 对于 STM32F2 和 STM32F4 系列 （FS 和 HS 内核）：

  HAL_PCD_SetRxFiFo();

  HAL_PCD_SetTxFiFo();

Rx 和 Tx FIFO 的总大小应该小于所用内核的总 FIFO 大小

（对于 USB OTG FS 内核，为 320 x 32 比特；对于 USB OTG HS 内核，为 1024 x 32 比特）。

- 对于 STM32F0、 STM32L0、 STM32F1 和 STM32F3 系列 （仅 FS 内核）：

  HAL_PCD_PMA_Config();

6. USB 设备库与实时操作系统 （RTOS）兼容吗？

是的，USB 设备库可与 RTOS 共用使用，CMSIS RTOS 封装的作用是对 OS 内核抽象。

9 修订历史

表 30. 文档修订历史

日期	修订	变更
2014 年 5 月 27 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