GigaDevice Semiconductor Inc.

GD32 USBFS&USBHS 固件库使用指南

应用笔记

AN050

目录

图索引

表索引

1. 前言

本文是专为 GD32 MCU 通用串行总线全速接口 USBFS 模块提供。USBHS 模块整体与 USB FS 相同，也可以参考该文档，相关的差异会在文档中标出。该笔记的目的是为了让客户更为方便的使用 USBFS 和 USBHS 固件库，能够更快的使用这套库进行项目的开发。

本笔记总共分为四个部分来讲述：

1. 库的架构和文件；
2. 库的底层和中间层驱动函数介绍；
3. 设备库的介绍；
4. 主机库的介绍。

表 1-1. 适用产品

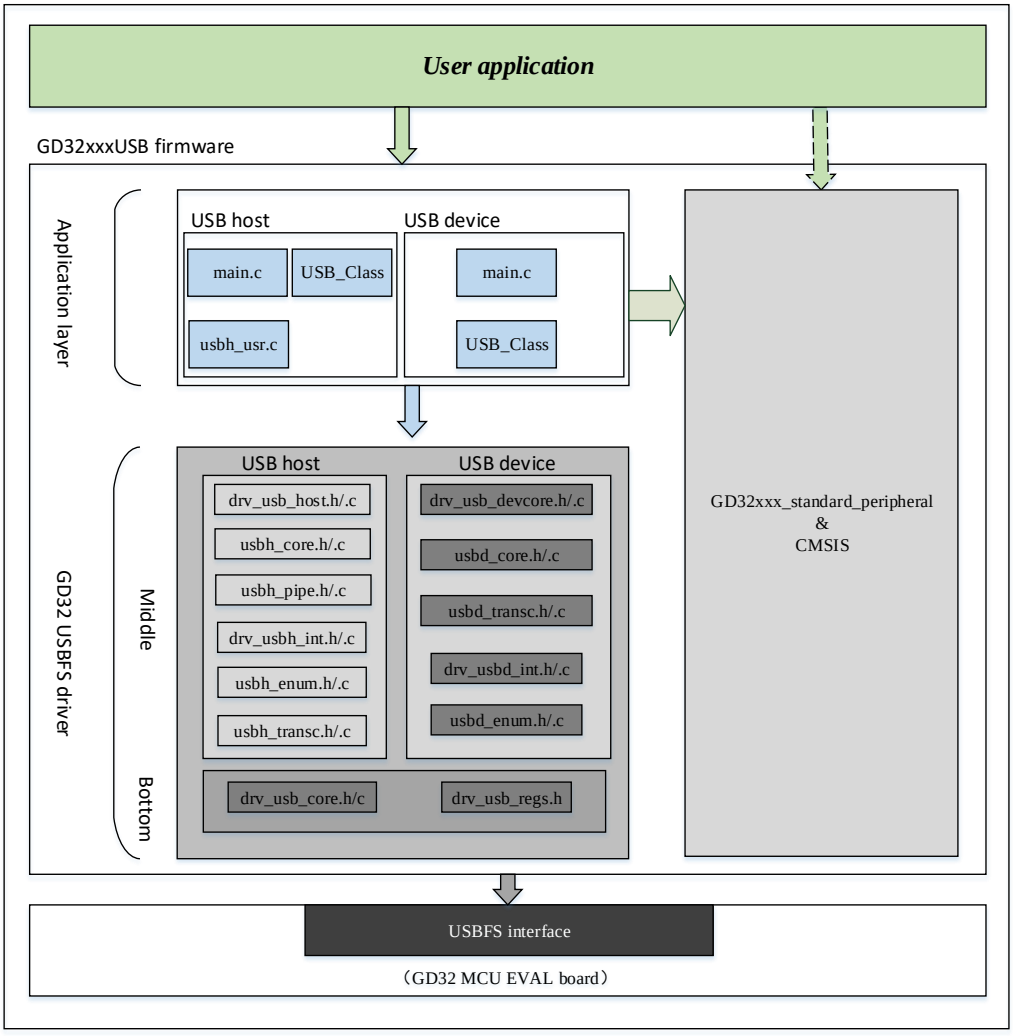
类型	型号	
MCU	FS	GD32F105xx 系列
		GD32F107xx 系列
		GD32F205xx 系列
		GD32F207xx 系列
		GD32F305xx 系列
		GD32F307xx 系列
		GD32F350xx 系列
		GD32F4xx 系列
		GD32E103xx 系列
		GD32C103xx 系列
		GD32E505xx 系列
		GD32E507xx 系列
		GD32E508xx 系列
		GD32VF103xx 系列
		GD32W515xx 系列
	HS	GD32F4xx 系列
		GD32E505xx 系列
		GD32E507xx 系列
		GD32E508xx 系列

2. 库架构和文件结构

2.1. 库架构

GD32 系列 USBFS 模块固件结构如*图 2-1. GD32 USBFS 固件库框架*所示。该图展示了 USBFS 主机和设备结构，用户应用程序调用了 USBFS 固件库实现 USB 的数据通信，该结构的底层是硬件，即 MCU 评估板。USBFS 固件库分为应用层和驱动层，用户层用户可以修改，驱动层包括 USB 主机或设备驱动和 USB 底层驱动。驱动层被称为“USBFS 固件库驱动”，用户不可修改。USB 应用类文件作为应用层的一部分，实现具体主机应用类或设备应用类请求。USBHS 的库架构和 USBFS 的基本类似，同样可以参考*图 2-1. GD32 USBFS 固件库框架*所示。

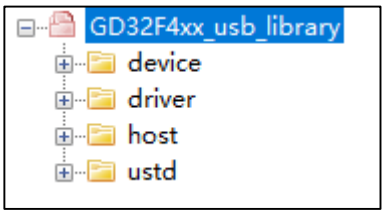
图 2-1. GD32 USBFS 固件库框架



2.2. 文件结构

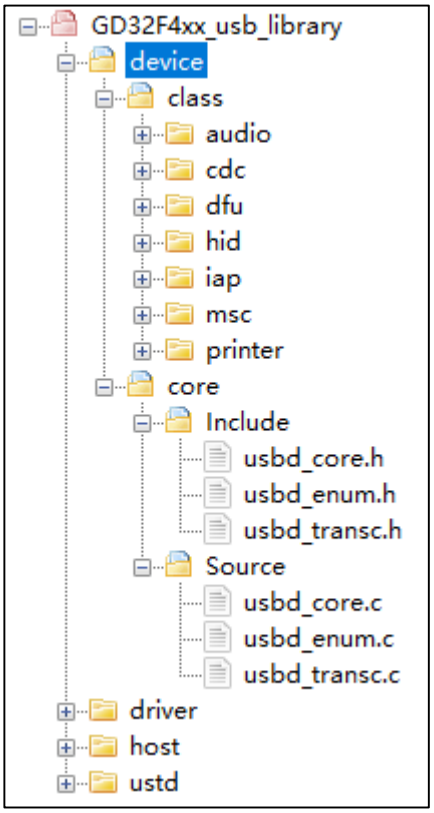
以 GD32F4xx 的 USBFS 固件库为例，包含如*图 2-2. USBFS 固件库文件夹*四个文件夹：

图 2-2. USBFS 固件库文件夹



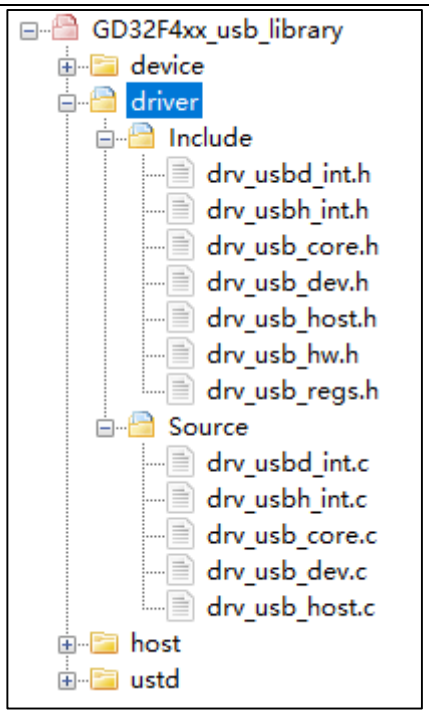
device 文件夹包含 USB 设备所需的协议层文件和设备类相关文件。

图 2-3. device 文件夹



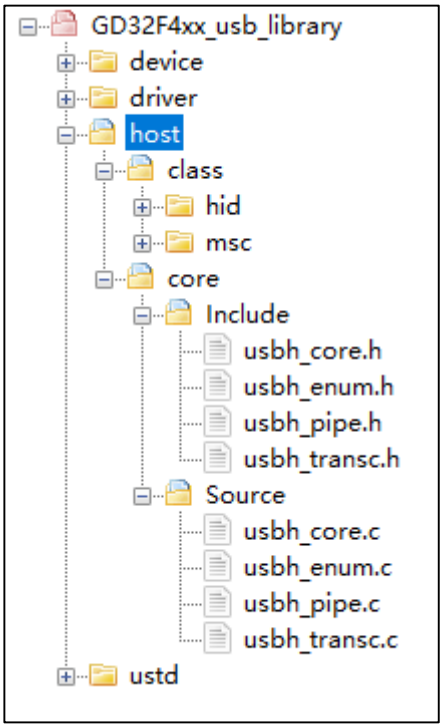
driver 文件夹包含整个库所需的寄存器定义、底层驱动和 USB 中断相关代码，主机设备都需要使用该文件夹中的部分文件。

图 2-4. driver 文件夹



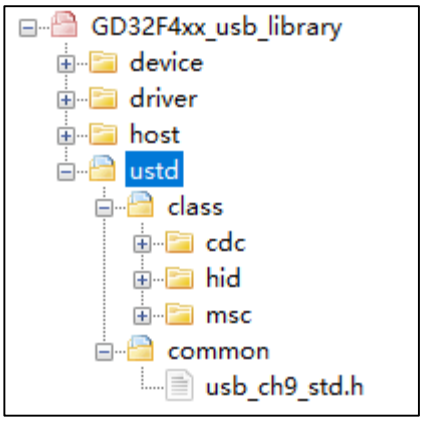
host 文件夹包含 USB 主机所需的协议层文件和设备类相关文件。

图 2-5. host 文件夹



ustd 文件夹包含通用的设备类文件和标准枚举需要的头文件，主机和设备都需要用到。

图 2-6. ustd 文件夹


GD32F4xx_usb_library
device
driver
host
ustd
class
cdc
hid
msc
common
usb_ch9_std.h

3. USBFS 底层驱动

USBFS 底层驱动文件位于 driver 文件夹中，如*表 3-1. USBFS 底层文件*。底层驱动是整个固件库和芯片中 USB 硬件模块直接关联的一层，包括寄存器读写、FIFO 操作等。

表 3-1. USBFS 底层文件

使用范围	文件名称	功能描述
共用	drv_usb_core.h/.c	USB 内核驱动
	drv_usb_regs.h	USB 内核底层驱动头文件
	drv_usb_hw.h	USB 硬件配置头文件

对上述.c 文件中的函数，详细描述如*表 3-2. usb_core.h/.c 文件函数*所示。

表 3-2. usb_core.h/.c 文件函数

函数名称	功能描述
usb_core_reset	配置 USB 内核软复位
usb_basic_init	配置 USB 内核基本功能
usb_core_init	初始化 USB 寄存器和准备内核的设备或主机模式操作
usb_txfifo_write	写数据至目标端点对应的发送 FIFO
usb_rxfifo_read	从目标端点的接收 FIFO 读取数据
usb_txfifo_flush	冲刷发送 FIFO
usb_rxfifo_flush	冲刷接收 FIFO
usb_set_rxfifo	设置接收 FIFO 大小
usb_set_txfifo	设置发送 FIFO 大小

drv_usb_regs.h 文件定义了整个 USBFS 模块的寄存器内容，所有底层的操作，都需要用到这里定义的内容。drv_usb_hw.h 文件声明了 USB 时钟、GPIO、延时、中断使能、CTC 相关的函数。

4. USBFS 中间层驱动

中间层驱动按照使用范围可以分为主机中间层和设备中间层，如*表 4-1. USBFS 中间层驱动文件*。设备中间层封装了 USB 设备传输所需的各种事务、基本功能的实现。主机中间层则封装了 USB 主机传输所需的各项事务、基本功能的实现。

表 4-1. USBFS 中间层驱动文件

使用范围	文件位置	文件名称	功能描述
主机	driver	drv_usb_host.h /.c	USB 主机模式底层驱动
		drv_usbh_int.h /.c	USB 主机模式中断处理文件
	host/core	usbh_core.h/.c	USB 主机内核状态机驱动
		usbh_enum.h/.c	USB 主机模式枚举驱动
		usbh_pipe.h/.c	USB 主机模式通道操作驱动
		usbh_transc.c	USB 主机模式事务驱动
设备	driver	drv_usb_dev.h /.c	USB 设备底层驱动
		drv_usbd_int.h /.c	USB 设备模式中断
	device/core	usbd_core.h /.c	USB 设备模式内核函数
		usbd_enum.h /.c	USB 枚举函数
		usbd_transc.h/.c	USB 事务内核函数

4.1. 主机中间层驱动函数

表 4-2. drv_usb_host.h/.c 文件函数

函数名称	功能描述
usb_host_init	初始化主机模式 USB 内核
usb_portvbus_switch	控制 VBUS
usb_port_reset	复位主机端口
usb_pipe_init	初始化主机通道
usb_pipe_xfer	为传输数据包准备主机通道
usb_pipe_halt	停止通道
usb_pipe_ping	配置主机通道执行 ping 操作
usb_host_stop	暂停主机和清空 FIFO

表 4-3. drv_usbh_int.h/.c 文件函数

函数名称	功能描述
usbh_isr	处理全局主机中断
usbh_int_port	处理主机端口中断
usbh_int_pipe	处理所有的主机通道中断
usbh_int_pipe_in	处理 IN 通道中断
usbh_int_pipe_out	处理 OUT 通道中断
usbh_int_rxfifonoempty	处理接收 FIFO 非空中断
usbh_int_txfifoempty	处理发送 FIFO 空中断

表 4-4. usbh_core.h/.c 文件函数

函数名称	功能描述
usbh_init	USB 主机栈初始化
usbh_deinit	USB 主机去初始化
usbh_class_register	USB 主机注册设备类
usbh_core_task	USB 主机内核状态机处理
usbh_error_handler	USB 主机错误处理
usbh_sof	USB SOF 回调函数
usbh_connect	USB 连接回调函数
usbh_disconnect	USB 断开连接回调函数
usbh_port_enabled	USB 端口使能回调函数
usbh_port_disabled	USB 端口除能回调函数
usbh_enum_task	USB 主机枚举任务处理

表 4-5. usbh_enum.h/.c 文件函数

函数名称	功能描述
usbh_ctlstate_config	配置 USB 控制状态参数
usbh_devdesc_get	获取设备描述符
usbh_cfgdesc_get	获取配置描述符
usbh_strdesc_get	获取字符串描述符
usbh_setaddress	设置相连设备地址
usbh_setcfg	设置相连设备配置值
usbh_setinterface	设置相连设备接口值
usbh_setdevfeature	设置或使能特定设备特性
usbh_clrdevfeature	清除或禁止特定设备特性
usbh_clrfeature	清除具体特性
usbh_nextdesc_get	解析下一个描述符头部
usbh_interface_select	选定一个接口
usbh_interface_find	查询某个特定设备类接口和复选接口值
usbh_devdesc_parse	解析设备描述符
usbh_cfgdesc_parse	解析配置描述符
usbh_cfgset_parse	解析配置描述符集
usbh_itfdesc_parse	解析接口描述符
usbh_epdesc_parse	解析端点描述符
usbh_strdesc_parse	解析字符串描述符

表 4-6. usbh_pipe.h/.c 文件函数

函数名称	功能描述
usbh_pipe_create	创建通道
usbh_pipe_update	更改通道
usbh_pipe_allocate	分配新通道
usbh_pipe_free	释放通道
usbh_pipe_delete	删除 USB 主机通道

函数名称	功能描述
usbh_freepipe_get	获取所释放的通道号

表 4-7. usbh_transc.h/.c 文件函数

函数名称	功能描述
usbh_ctlsetup_send	发送 SETUP 包至 USB 设备
usbh_data_send	发送数据包至 USB 设备
usbh_data_recev	从 USB 设备接收数据包
usbh_ctl_handler	USB 控制传输处理
usbh_urb_wait	等待 USB URB 状态
usbh_setup_transc	USB SETUP 事务
usbh_data_in_transc	USB 数据 IN 事务
usbh_data_out_transc	USB 数据 OUT 事务
usbh_status_in_transc	USB 状态 IN 事务
usbh_status_out_transc	USB 状态 OUT 事务
usbh_request_submit	准备通道，开始传输

4.2. 设备中间层驱动函数

表 4-8. drv_usb_dev.h/.c 文件函数

函数名称	功能描述
usb_devcore_init	初始化设备模式 USB 内核寄存器
usb_devint_enable	使能 USB 设备模式中断
usb_transc0_active	激活 USB 端点 0 事务
usb_transc_active	激活 USB 事务
usb_transc_deactivate	关闭 USB 事务
usb_transc_inxfer	配置 USB 事务，开始 IN 传输
usb_transc_outxfer	配置 USB 事务，开始 OUT 传输
usb_transc_stall	设置 USB 事务 stall 状态
usb_transc_clrstall	清除 USB 事务 stall 状态
usb_iepintr_read	读取设备 IN 端点中断标志寄存器
usb_ctlep_startout	配置 OUT 端点 0，接收 setup 包
usb_rwkup_active	激活远程唤醒信号
usb_clock_active	激活 USB 时钟
usb_dev_suspend	挂起 USB 设备
usb_dev_stop	暂停设备、清空 FIFO
usb_dev_disconnect	配置 USB 设备断开连接
usb_dev_connect	配置 USB 设备连接
usb_devaddr_set	设置 USB 设备地址
usb_oepintnum_read	读设备 OUT 端点中断寄存器
usb_oepintr_read	读取设备 OUT 端点中断标志寄存器
usb_iepintnum_read	读取设备 IN 端点中断寄存器

函数名称	功能描述
usb_rwkup_reset	复位远程唤醒信号
usb_rwkup_set	置位远程唤醒

表 4-9. drv_usbd_int.h/.c 文件函数

函数名称	功能描述
usbd_int_dedicated_ep1out	USB 专用 OUT 方向端点 1 中断处理
usbd_int_dedicated_ep1in	USB 专用 IN 方向端点 1 中断处理
usbd_isr	USB 设备模式全局中断处理
usbd_int_epout	指示 OUT 端点有挂起中断
usbd_int_epin	指示 IN 端点有挂起中断
usbd_int_rxfifo	处理接收状态序列中断
usbd_int_reset	处理 USB 复位中断
usbd_int_enumfinish	处理 USB 枚举完成中断
usbd_int_suspend	处理 USB 挂起中断
usbd_emptytxfifo_write	为写入下一包检查是否发送 FIFO 为空

表 4-10. usbd_core.h/.c 文件函数

函数名称	功能描述
usbd_init	初始化 USB 设备模式栈和应用类驱动
usbd_ep_setup	端点初始化
usbd_ep_clear	除能端点
usbd_ep_recev	端点准备接收数据
usbd_ep_send	端点准备发送数据
usbd_ep_stall	设置端点为 stall 状态
usbd_ep_stall_clear	清除端点的 stall 状态
usbd_fifo_flush	冲刷端点 FIFO
usbd_connect	设备连接
usbd_disconnect	断开设备连接
usbd_addr_set	设置 USB 设备地址
usbd_rxcount_get	获取接收数据长度

表 4-11. usbd_enum.h/.c 文件函数

函数名称	功能描述
usbd_standard_request	处理 USB 标准设备请求
usbd_class_request	处理 USB 设备类请求
usbd_vendor_request	处理厂商自定义请求
usbd_enum_error	处理枚举错误
int_to_unicode	转换 Hex 32 位数值为无符号字符
serial_string_get	获取序列号字符串
_usb_std_reserved	无操作，保留
_usb_dev_desc_get	获取设备描述符
_usb_config_desc_get	获取配置描述符

函数名称	功能描述
_usb_bos_desc_get	获取 BOS 描述符
_usb_str_desc_get	获取字符串描述符
_usb_std_getstatus	处理 Get_Status 请求
_usb_std_clearfeature	处理 Clear_Feature 请求
_usb_std_setfeature	处理 Set_Feature 请求
_usb_std_setaddress	处理 Set_Address 请求
_usb_std_getdescriptor	处理 Get_Descriptor 请求
_usb_std_setdescriptor	处理 Set_Descriptor 请求
_usb_std_getconfiguration	处理 Get_Configuration 请求
_usb_std_setconfiguration	处理 Set_Configuration 请求
_usb_std_getinterface	处理 Get_Interface 请求
_usb_std_setinterface	处理 Set_Interface 请求
_usb_std_synchframe	处理同步帧请求

表 4-12. usbd_transc.h/.c 文件函数

函数名称	功能描述
usbd_ctl_send	USB 控制事务中发送数据
usbd_ctl_recev	USB 控制事务中接收数据
usbd_ctl_status_send	USB 发送控制事务状态
usbd_ctl_status_recev	USB 接收控制事务状态
usbd_setup_transc	USB SETUP 阶段处理
usbd_out_transc	USB 数据 OUT 阶段处理
usbd_in_transc	USB 数据 IN 阶段处理

5. USBFS 设备库

USBFS 设备库是基于上述底层驱动和中间层驱动的基础上来实现的，主要包括设备库配置、描述符、中断、用户接口、设备类接口、例程介绍等内容。

5.1. 设备库配置

设备库配置包含两个文件，均位于例程的工程文件夹中，和固件库的其他驱动文件不在同一个目录下。

5.1.1. usbd_conf.h

文件配置选项如下：

```
#define USBD_CFG_MAX_NUM                1
#define USBD_ITF_MAX_NUM                1

#define USBD_HID_INTERFACE              0

/* USB user string supported */
/* #define USB_SUPPORT_USER_STRING_DESC */

#define HID_IN_EP                       EP1_IN

#define HID_IN_PACKET                   8
```

各个配置的定义如*表 5-1. usbd_conf.h 配置说明*:

表 5-1. usbd_conf.h 配置说明

配置名称	功能描述
USBD_CFG_MAX_NUM	配置最大数目
USBD_ITF_MAX_NUM	接口最大数目
USBD_HID_INTERFACE	设备类接口序号
HID_IN_EP	IN 端点号
HID_IN_PACKET	端点数据包长

5.1.2. usb_conf.h

usb_conf.h 文件中主要定义了 USBFS 模块的 FIFO 分配。USBFS 内部包含 1.25KB 的 RAM 缓冲区，即 320 个字的 FIFO 大小。而 USBHS 内部包含 4KB 的 RAM 缓冲区，即 1K 个字的 FIFO 大小。

文件配置选项如下：

```
#ifdef USB_FS_CORE
```

```
    #define RX_FIFO_FS_SIZE                 128
    #define TX0_FIFO_FS_SIZE                64
    #define TX1_FIFO_FS_SIZE                128
    #define TX2_FIFO_FS_SIZE                0
    #define TX3_FIFO_FS_SIZE                0
#endif /* USB_FS_CORE */

#define USB_SOF_OUTPUT              1
#define USB_LOW_POWER               1

//#define VBUS_SENSING_ENABLED

//#define USE_HOST_MODE
#define USE_DEVICE_MODE
//#define USE_OTG_MODE
```

表 5-2. usb_conf.h 配置说明

配置名称	功能描述
RX_FIFO_FS_SIZE	接收 FIFO 大小设置
TXx_FIFO_FS_SIZE	发送 FIFO 大小设置
USB_SOF_OUTPUT	使能 SOF 输出功能
USB_LOW_POWER	使能低功耗模式
VBUS_SENSING_ENABLED	使能 VBUS SENSING 注：该功能指示硬件是否支持 VBUS 功能，如果使能了该功能，则必须要连接 VBUS 引脚
USE_HOST_MODE	主机模式
USE_DEVICE_MODE	设备模式
USE_OTG_MODE	OTG 模式

注意：

1. 如果需要通过示波器来测量 SOF，则需要将 PA8 引脚配置为输出。
2. USE_HOST_MODE、USE_DEVICE_MODE 和 USE_OTG_MODE 三种模式只能选择其中一种。
3. GD32F105xx/F107xx 和 GD32F205xx/F207xx 系列的 USB 并不具备 VBUS_SENSING_ENABLED 的功能。所以这两个系列的 USB 在作为设备时，必须要连接 VBUS 引脚。

5.2. 固件库流程

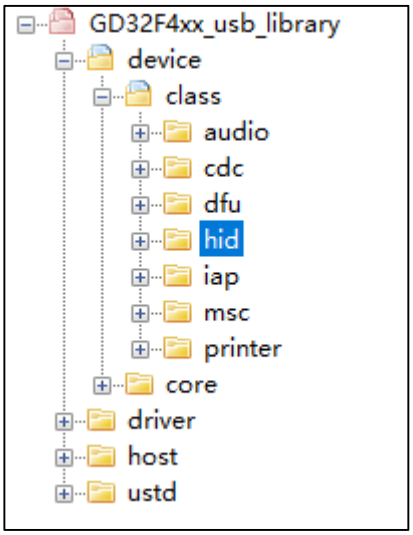
USBFS 固件库的流程包含时钟配置、中断使能、USB 寄存器配置、端点配置、连接、枚举、数据传输等过程。具体如*图 5-1. 固件库流程图*所示：

图 5-1. 固件库流程图

5.3. 描述符

USBFS 设备库的描述符均包含在设备类文件中，如*图 5-2. 设备类文件路径*所示。

图 5-2. 设备类文件路径



描述符主要包括设备描述符、配置描述符集和字符串描述符等。有些设备类还会包括特殊的描述符，如 HID 设备包含报告描述符，高速设备会包含 other speed 配置描述符集和 qualifier 描述符。

描述符都是在枚举阶段发送给主机，对应到设备库代码是在 device/core/usbd_enum.c 文件中，具体是在标准枚举阶段对如下两个指针数组进行回调实现的：

```
static usb_reqsta (*_std_dev_req[])(usb_core_driver *udev, usb_req *req) =
{
    [USB_GET_STATUS]          = _usb_std_getstatus,
    [USB_CLEAR_FEATURE]       = _usb_std_clearfeature,
    [USB_RESERVED2]           = _usb_std_reserved,
    [USB_SET_FEATURE]         = _usb_std_setfeature,
    [USB_RESERVED4]           = _usb_std_reserved,
    [USB_SET_ADDRESS]         = _usb_std_setaddress,
    [USB_GET_DESCRIPTOR]      = _usb_std_getdescriptor,     // get descriptor
    [USB_SET_DESCRIPTOR]      = _usb_std_setdescriptor,
    [USB_GET_CONFIGURATION] = _usb_std_getconfiguration,
    [USB_SET_CONFIGURATION] = _usb_std_setconfiguration,
    [USB_GET_INTERFACE]       = _usb_std_getinterface,
    [USB_SET_INTERFACE]       = _usb_std_setinterface,
    [USB_SYNCH_FRAME]         = _usb_std_synchframe,
};

/* get standard descriptor handler */
static uint8_t* (*std_desc_get[])(usb_core_driver *udev, uint8_t index, uint16_t *len) = {
    [(uint8_t)USB_DESCTYPE_DEV - 1U]                = _usb_dev_desc_get,
    [(uint8_t)USB_DESCTYPE_CONFIG - 1U]             = _usb_config_desc_get,
    [(uint8_t)USB_DESCTYPE_STR - 1U]                = _usb_str_desc_get,
#ifdef USE_USB_HS
    [(uint8_t)USB_DESCTYPE_DEV_QUALIFIER - 3U]      = _usb_qualifier_desc_get,
```

```
    [(uint8_t)USB_DESCTYPE_OTHER_SPD_CONFIG - 3U] =
_usb_other_speed_config_desc_get,
#endif
};
```

5.4. 中断处理

USBFS 设备接口的中断如*表 5-3. USBFS 设备中断*所示，每个中断标志对应着中断函数中一个处理项，其中与数据传输相关的有 OEPIF、IEPIF 和 RXFNEIF。在 USBFS 设备接口模块中 IN 和 OUT 事务的正确传输分为两个不同的中断，分别为 OUT 端点中断标志 GINTF_OEPIF 和 IN 端点中断标志 GINTF_IEPIF。

表 5-3. USBFS 设备中断

中断标志	描述	运行模式
WKUPIF	唤醒中断标志位	主机或设备模式
SESIF	会话中断	主机或设备模式
IDPSC	ID 引脚状态变化	主机或设备模式
LPMIF(1)	LPM 中断标志	主机或设备模式
ISOONCIF/PXNCIF	周期性传输未完成中断标志 / 同步 OUT 传输未完成中断标志	主机或设备模式
ISOINCIF	同步 IN 传输未完成中断标志	设备模式
OEPIF	OUT 端点中断标志	设备模式
IEPIF	IN 端点中断标志	设备模式
EOPFIF	周期性帧结束中断标志	设备模式
ISOOPDIF	同步 OUT 丢包中断标志	设备模式
ENUMF	速度枚举完成中断标志	设备模式
RST	USB 复位中断标志	设备模式
SP	USB 挂起中断标志	设备模式
ESP	早期挂起中断标志	设备模式
GONAK	全局 OUT NAK 有效中断标志	设备模式
GNPINAK	全局非周期 IN NAK 有效中断标志	设备模式
RXFNEIF	Rx FIFO 非空中断标志	主机或设备模式
SOF	帧起始中断标志	主机或设备模式
OTGIF	OTG 中断标志	主机或设备模式
MFIF	模式错误中断标志	主机或设备模式

注意：

(1) 仅在E50X系列中有该位。

OUT 端点中断处理函数如下列代码所示。

```
static uint32_t usbd_int_epout (usb_core_driver *udev)
{
    uint32_t epintnum = 0U;
```

```
    uint8_t ep_num = 0U;

    for (epintnum = usb_oepintnum_read (udev); epintnum; epintnum >>= 1, ep_num++) {
        if (epintnum & 0x01U) {
            __IO uint32_t oepintr = usb_oepintr_read (udev, ep_num);

            /* transfer complete interrupt */
            if (oepintr & DOEPINTF_TF) {
                /* clear the bit in DOEPINTF for this interrupt */
                udev->regs.er_out[ep_num]->DOEPINTF = DOEPINTF_TF;

                if ((uint8_t)USB_USE_DMA == udev->bp.transfer_mode) {
                    __IO uint32_t eplen = udev->regs.er_out[ep_num]->DOEPLEN;

                    udev->dev.transc_out[ep_num].xfer_count =
udev->dev.transc_out[ep_num].max_len - \
                                                        (eplen & DEPLEN_TLEN);
                }

                /* inform upper layer: data ready */
                (void)usbd_out_transc (udev, ep_num);       // out transaction

                if ((uint8_t)USB_USE_DMA == udev->bp.transfer_mode) {
                    if ((0U == ep_num) && ((uint8_t)USB_CTL_STATUS_OUT ==
udev->dev.control.ctl_state)) {
                        usb_ctlep_startout (udev);
                    }
                }
            }

            /* setup phase finished interrupt (control endpoints) */
            if (oepintr & DOEPINTF_STPF) {
                /* inform the upper layer that a setup packet is available */
                (void)usbd_setup_transc (udev);       // setup transaction

                udev->regs.er_out[ep_num]->DOEPINTF = DOEPINTF_STPF;
            }
        }
    }

    return 1U;
}
```

在 OUT 端点中断处理函数中，又可根据端点中断标志寄存器 out_endp_intr 进行判断产生 OUT

端点中断的事件，具体可分为：传输完成中断处理和 SETUP 令牌包完成中断处理。产生相应的 OUT 端点中断事件后，根据轮询中断标志位进入不同的中断处理函数。

IN 端点的中断处理函数如下列代码所示。

```
static uint32_t usbd_int_epin (usb_core_driver *udev)
{
    uint32_t epintnum = 0U;
    uint8_t ep_num = 0U;

    for (epintnum = usb_iepintnum_read (udev); epintnum; epintnum >>= 1, ep_num++) {
        if (epintnum & 0x1U) {
            __IO uint32_t iepintr = usb_iepintr_read (udev, ep_num);

            if (iepintr & DIEPINTF_TF) {
                udev->regs.er_in[ep_num]->DIEPINTF = DIEPINTF_TF;

                /* data transmission is completed */
                (void)usbd_in_transc (udev, ep_num);      // IN transaction

                if ((uint8_t)USB_USE_DMA == udev->bp.transfer_mode) {
                    if ((0U == ep_num) && ((uint8_t)USB_CTL_STATUS_IN ==
udev->dev.control.ctl_state)) {
                        usb_ctlep_startout (udev);
                    }
                }
            }

            if (iepintr & DIEPINTF_TXFE) {
                usbd_emptytxfifo_write (udev, (uint32_t)ep_num);      // write FIFO

                udev->regs.er_in[ep_num]->DIEPINTF = DIEPINTF_TXFE;
            }
        }
    }

    return 1U;
}
```

在 IN 端点中断处理函数中，包含传输完成中断处理、发送 FIFO 空中断处理。产生相应的 IN 端点中断事件后，根据轮询中断标志位进入不同的中断处理函数。

Rx FIFO 非空的中断处理函数如下列代码所示。

```
static uint32_t usbd_int_rxfifo (usb_core_driver *udev)
{
    usb_transc *transc = NULL;
```

```
    uint8_t data_PID = 0U;
    uint32_t bcount = 0U;

    __IO uint32_t devrxstat = 0U;

    /* disable the Rx status queue non-empty interrupt */
    udev->regs.gr->GINTEN &= ~GINTEN_RXFNEIE;

    /* get the status from the top of the FIFO */
    devrxstat = udev->regs.gr->GRSTATP;

    uint8_t ep_num = (uint8_t)(devrxstat & GRSTATRP_EPNUM);

    transc = &udev->dev.transc_out[ep_num];

    bcount = (devrxstat & GRSTATRP_BCOUNT) >> 4U;
    data_PID = (uint8_t)((devrxstat & GRSTATRP_DPID) >> 15U);

    switch ((devrxstat & GRSTATRP_RPCKST) >> 17U) {
    case RSTAT_GOUT_NAK:
        break;

    case RSTAT_DATA_UPDT:
        if (bcount > 0U) {
            (void)usb_rxfifo_read (&udev->regs, transc->xfer_buf, (uint16_t)bcount);      // read
FIFO

            transc->xfer_buf += bcount;
            transc->xfer_count += bcount;
        }
        break;

    case RSTAT_XFER_COMP:
        /* trigger the OUT endpoint interrupt */
        break;

    case RSTAT_SETUP_COMP:
        /* trigger the OUT endpoint interrupt */
        break;

    case RSTAT_SETUP_UPDT:
        if ((0U == transc->ep_addr.num) && (8U == bcount) && (DPID_DATA0 == data_PID)) {
```

```
                /* copy the setup packet received in FIFO into the setup buffer in RAM */
                (void)usb_rxfifo_read (&udev->regs, (uint8_t *)&udev->dev.control.req,
(uint16_t)bcount);          // read FIFO

                transc->xfer_count += bcount;
            }
            break;

        default:
            break;
        }

        /* enable the Rx status queue level interrupt */
        udev->regs.gr->GINTEN |= GINTEN_RXFNEIE;

        return 1U;
}
```

在 Rx FIFO 非空的中断处理函数中，主要进行 FIFO 数据的接收，包括 setup 事务中的请求数据和 out 事务的数据。

5.5. USB 设备类接口

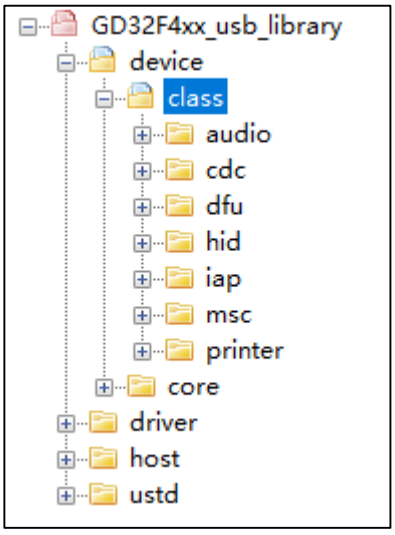
USB 设备类接口由下列的结构体实现：

```
typedef struct _usb_class_core
{
    uint8_t    command;                                                    /*!<
device class request command */
    uint8_t    alter_set;                                                  /*!<
alternative set */

    uint8_t    (*init)          (usb_dev *udev, uint8_t config_index);     /*!< initialize
handler */
    uint8_t    (*deinit)        (usb_dev *udev, uint8_t config_index);     /*!< de-initialize
handler */

    uint8_t    (*req_proc)       (usb_dev *udev, usb_req *req);            /*!< device
request handler */

    uint8_t    (*set_intf)      (usb_dev *udev, usb_req *req);             /*!< device set
interface callback */
```

```
    uint8_t  (*ctlx_in)          (usb_dev *udev);                    /*!< device
contrl in callback */
    uint8_t  (*ctlx_out)         (usb_dev *udev);

    uint8_t  (*data_in)          (usb_dev *udev, uint8_t ep_num);    /*!< device
data in handler */
    uint8_t  (*data_out)         (usb_dev *udev, uint8_t ep_num);    /*!< device
data out handler */

    uint8_t  (*SOF)              (usb_dev *udev);                    /*!< Start
of frame handler */

    uint8_t  (*incomplete_isoc_in)  (usb_dev *udev);                 /*!<
Incomplete synchronization IN transfer handler */
    uint8_t  (*incomplete_isoc_out) (usb_dev *udev);                 /*!<
Incomplete synchronization OUT transfer handler */
} usb_class_core;
```

该结构体的初始化工作，由各个设备类单独实现。每个设备类的接口文件保存在 device/class 路径下的设备类文件夹中如*图 5-3.设备类文件*所示。

图 5-3.设备类文件



例如 HID 设备的初始化如下：

```
usb_class_core usbd_hid_cb = {
    .command        = NO_CMD,
    .alter_set      = 0U,
    .init           = hid_init,
    .deinit         = hid_deinit,
    .req_proc       = hid_req,
    .data_in        = hid_data_in
};
```

通过上述的初始化，可以实现设备类的初始化、去初始化、设备类请求和数据传输等工作。

5.6. 数据传输过程

枚举完成后，USB 会进行数据的收发，整个过程完全是由主机控制。主机通过给设备发送 IN 包来收数据，通过 OUT 包来给设备发送数据。下面通过 CDC 的例程来简要说明 USB 数据在 GD FS 库中的简要传输过程。

5.6.1. IN 方向

数据传输阶段，IN 方向非零端点的数据是在 usbd_in_transc 函数中处理的。data_in 的回调函数实际调用的是 cdc_acm_in 函数，进入到 cdc_acm_in 函数中则表示有数据成功的从设备发到主机，后续则可以继续调用发送函数 cdc_acm_data_send，准备发送下一次的数据。

```
uint8_t usbd_in_transc (usb_core_driver *udev, uint8_t ep_num)
{
    if (0U == ep_num) {
        usb_transc *transc = &udev->dev.transc_in[0];
        /* ..... */
    } else {
        if ((udev->dev.cur_status == (uint8_t)USBD_CONFIGURED) &&
(udev->dev.class_core->data_in != NULL)) {
            (void)udev->dev.class_core->data_in (udev, ep_num);
        }
    }

    return (uint8_t)USBD_OK;
}
```

5.6.2. OUT 方向

数据传输阶段，OUT 方向非零端点的数据是在 usbd_out_transc 函数中处理的。data_out 的回调函数实际调用的是 cdc_acm_out 函数，进入到 cdc_acm_out 函数中则表示有数据被设备成功接收，后续则可以继续调用 cdc_acm_data_receive 来准备进行下一包数据。

```
uint8_t usbd_out_transc (usb_core_driver *udev, uint8_t ep_num)
{
    if (0U == ep_num) {
        usb_transc *transc = &udev->dev.transc_out[0];
        /*   .....   */
    } else if ((udev->dev.class_core->data_out != NULL) && (udev->dev.cur_status ==
(uint8_t)USBD_CONFIGURED)) {
        (void)udev->dev.class_core->data_out (udev, ep_num);
    } else {
        /* no operation */
```

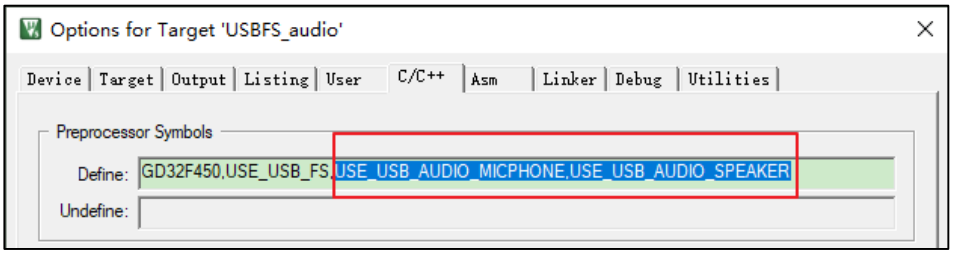
```
    }

    return (uint8_t)USBD_OK;
}
```

5.7. USB 设备类例程

5.7.1. 音频 AUDIO

AUDIO 设备包含两个接口：Speaker 和 Micphone。可以在工程的配置中选择包含这两个接口，如*图 5-4. AUDIO 宏配置*所示：

图 5-4. AUDIO 宏配置



AUDIO 描述符介绍

设备描述符中包含了 AUDIO 设备的 VID 和 PID，分别为 0x28e9、0x9574。在配置描述符集中，包含了 Speaker 和 Micphone 相关的描述符项。AUDIO 中的 Speaker 和 Micphone 分别对应一个接口，接口的相关描述符内容如*表 5-4. AUDIO 相关描述符*所示：

表 5-4. AUDIO 相关描述符

描述符名称	功能描述
usb_desc_AC_itf	AC 接口描述符
usb_desc_input_terminal	输入终端描述符
usb_desc_mono_feature_unit	单声道特征单元描述符
usb_desc_output_terminal	输出终端描述符
usb_desc_AS_itf	AS 接口描述符
usb_desc_format_type	格式类型描述符
usb_desc_std_ep	标准端点描述符
usb_desc_AS_ep	AS 端点描述符

AUDIO 设备类接口

AUDIO 设备类接口如下面结构体所示，结构体的函数实现参见*表 5-5. AUDIO 设备类接口函数*：

```
usb_class_core usbd_audio_cb = {
    .init          = audio_init,
    .deinit        = audio_deinit,
```

```
    .req_proc    = audio_req_handler,
    .ctlx_out    = audio_ctlx_out,
    .data_in     = audio_data_in,
    .data_out    = audio_data_out,
    .SOF         = usbd_audio_sof
};
```

表 5-5. AUDIO 设备类接口函数

函数名称	功能描述
audio_init	初始化 AUDIO 设备
audio_deinit	去初始化 AUDIO 设备
audio_req_handler	AUDIO 设备类请求函数
audio_ctlx_out	OUT 控制传输回调
audio_data_in	IN 数据传输回调
audio_data_out	OUT 数据传输回调
usbd_audio_sof	SOF 事件回调

AUDIO 设备类请求

AUDIO 包含如*表 5-6. AUDIO 设备类请求*所示的各个设备类请求：

表 5-6. AUDIO 设备类请求

请求名称	功能描述
AUDIO_REQ_SET_CUR	设置当前值请求
AUDIO_REQ_GET_CUR	获取当前值请求
AUDIO_REQ_SET_MIN	设置最小值请求
AUDIO_REQ_GET_MIN	获取最小值请求
AUDIO_REQ_SET_MAX	设置最大值请求
AUDIO_REQ_GET_MAX	获取最大值请求
AUDIO_REQ_SET_RES	设置分辨率请求
AUDIO_REQ_GET_RES	获取分辨率请求

AUDIO 用户接口

AUDIO 的用户接口定义在如下结构体中：

```
audio_fops_struct audio_out_fops =
{
    init,
    deinit,
    audio_cmd,
    volume_ctl,
    mute_ctl,
    periodic_tc,
    get_state
};
```

各个函数功能如*表 5-7. AUDIO 用户接口函数*所示：

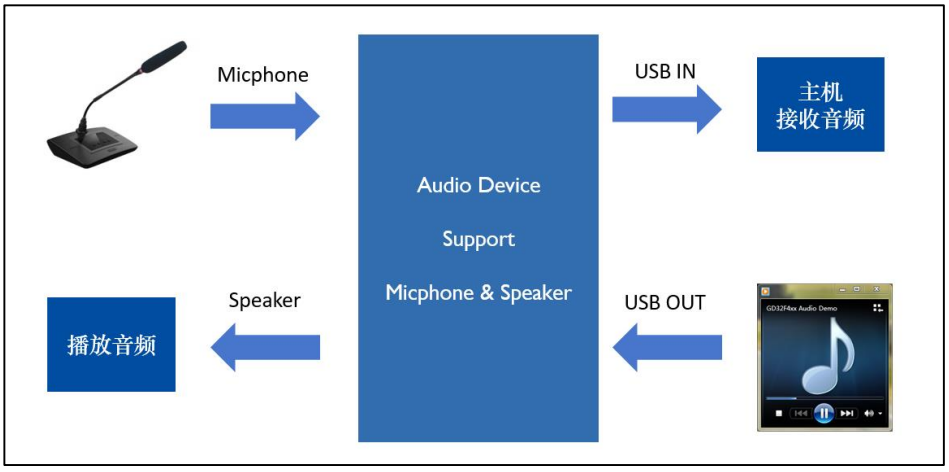
表 5-7. AUDIO 用户接口函数

函数名称	功能描述
init	初始化 AUDIO 需要用到的资源
deinit	停止 AUDIO 功能并释放所有资源
audio_cmd	播放、停止、暂停、重新开始等命令
volume_ctl	音量控制
mute_ctl	静音控制
periodic_tc	周期传输控制
get_state	获取 AUDIO 的当前状态

AUDIO 例程操作指南

将 GD 库中 Audio 例程下载到开发板中，在设备管理器中可以看到如下新添加的设备。

图 5-5. AUDIO 设备



1） 数据 OUT 阶段

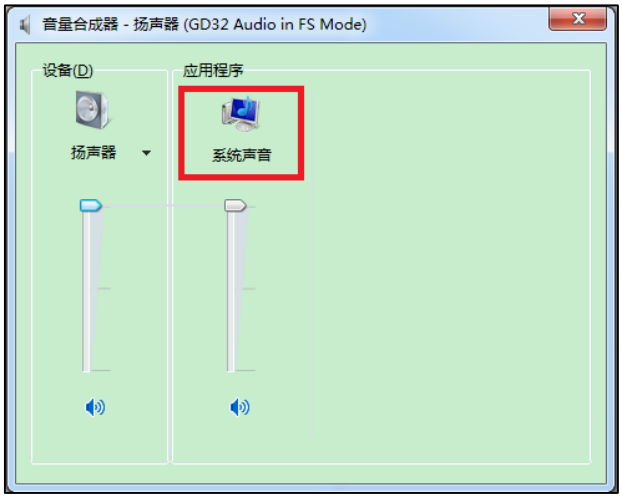
打开电脑上的音频文件，如*图 5-6. AUDIO 播放文件*所示，在开发板上接上耳机即可听到电脑上播放的音频。

图 5-6. AUDIO 播放文件

2） 数据 IN 阶段

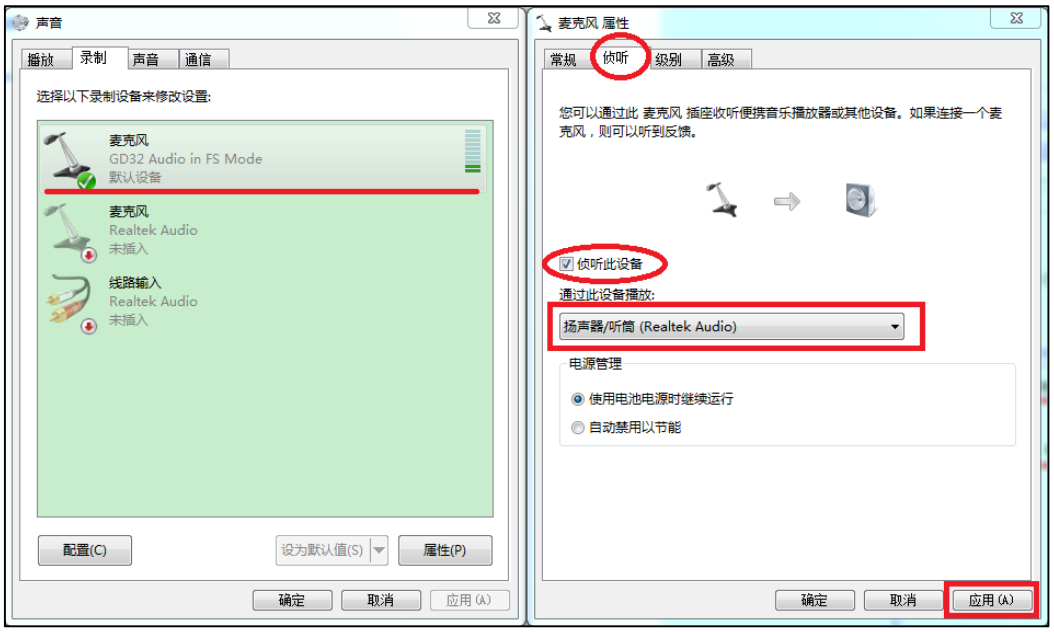
点击桌面右下角的小喇叭，调出扬声器的音量合成器，如*图 5-7. AUDIO 系统声音配置*。点击图上的系统声音后，弹出声音的操作界面。

图 5-7. AUDIO 系统声音配置



进入“录制”标题下，双击麦克风，在弹出的麦克风属性中选择侦听，在侦听界面中勾选侦听此设备，并选择默认的播放设备，如*图 5-8. AUDIO 录制侦听配置*。之后就可以通过默认的播放设备听到从 MCU 上传到 PC 端的音频了。

图 5-8. AUDIO 录制侦听配置



5.7.2. 虚拟串口 CDC

虚拟串口 CDC 例程遵循了 USB 通信类相关子协议，可以将 USB 虚拟成 COM 口，可以像普通的串口一样进行操作。一般 win7、win8、XP 系统上需要安装相应的驱动，而 win10 系统本身带有 CDC 相关的驱动，可以实现免驱。

CDC 描述符介绍

设备描述符中包含了 CDC 设备的 VID 和 PID，分别为 0x28e9、0x018a。在配置描述符集中，包含了 CDC 相关的描述符项。CDC 本身包含两个接口，一个命令接口、一个数据接口，命令接口相关描述符内容如*表 5-8. CDC 相关描述符*所示：

表 5-8. CDC 相关描述符

描述符名称	功能描述
usb_desc_header_func	功能开启描述符
usb_desc_call_managment_func	通信管理描述符
usb_desc_acm_func	抽象控制管理描述符
usb_desc_union_func	联合功能描述符

CDC 设备类接口

CDC 设备类接口如下面结构体所示，结构体的函数实现参见*表 5-9. CDC 设备类接口函数*：

```
usb_class_core cdc_class =
{
    .command     = NO_CMD,
    .alter_set   = 0U,

    .init        = cdc_acm_init,
```

```
    .deinit      = cdc_acm_deinit,

    .req_proc    = cdc_acm_req,
    .ctlx_out    = cdc_ctlx_out,

    .data_in     = cdc_acm_in,
    .data_out    = cdc_acm_out
};
```

表 5-9. CDC 设备类接口函数

函数名称	功能描述
cdc_acm_init	初始化 AUDIO 设备
cdc_acm_deinit	去初始化 AUDIO 设备
cdc_acm_req	AUDIO 设备类请求函数
cdc_ctlx_out	OUT 控制传输回调
cdc_acm_in	IN 数据传输回调
cdc_acm_out	OUT 数据传输回调

CDC 设备类请求

CDC 包含如*表 5-10. CDC 设备类请求*所示的各个设备类请求：

表 5-10. CDC 设备类请求

请求名称	功能描述
SEND_ENCAPSULATED_COMMAND	未用到
GET_ENCAPSULATED_RESPONSE	未用到
SET_COMM_FEATURE	未用到
GET_COMM_FEATURE	未用到
CLEAR_COMM_FEATURE	未用到
SET_LINE_CODING	设置串口属性
GET_LINE_CODING	获取串口属性
SET_CONTROL_LINE_STATE	未用到
SEND_BREAK	未用到

CDC 用户接口

CDC 的用户接口包含如下几个函数：

```
/* function declarations */
/* check CDC ACM is ready for data transfer */
uint8_t cdc_acm_check_ready(usb_dev *udev);
/* send CDC ACM data */
void cdc_acm_data_send(usb_dev *udev);
/* receive CDC ACM data */
void cdc_acm_data_receive(usb_dev *udev);
```

各个函数功能如*表 5-11. CDC 用户接口函数*所示：

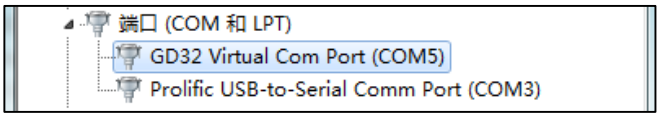
表 5-11. CDC 用户接口函数

函数名称	功能描述
cdc_acm_check_ready	CDC 准备好进行数据传输
cdc_acm_data_send	CDC 数据发送
cdc_acm_data_receive	CDC 数据接收

CDC 例程操作指南

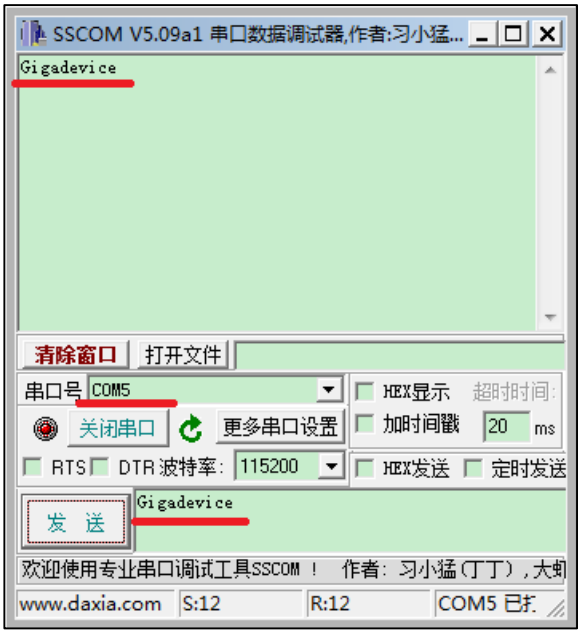
将 GD 库中 CDC_ACM 例程下载到开发板中，在设备管理器中可以看到如下新添加的 COM 口。

图 5-9. CDC 设备



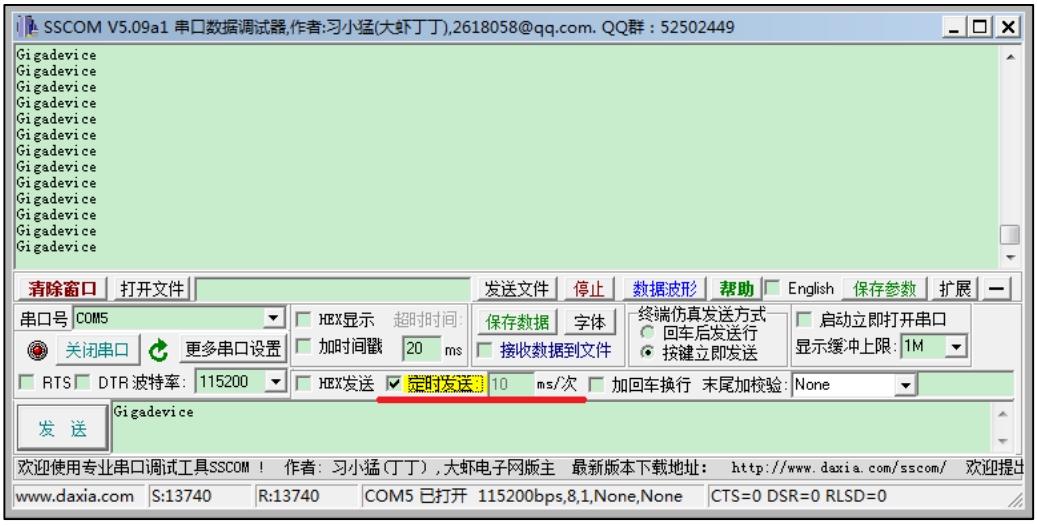
在电脑上打开串口调试助手，选择设备管理器中新添加的 COM 口打开，由于该例程的功能是回显，所以发送一串字符，在接收区域会接收到一串相同的字符。

图 5-10. 虚拟串口数据收发



为了进行大数据量的测试，可以增加发送的字节数量，并设置串口助手的定时发送功能，如*图 5-11. 虚拟串口大数据量收发*所示。

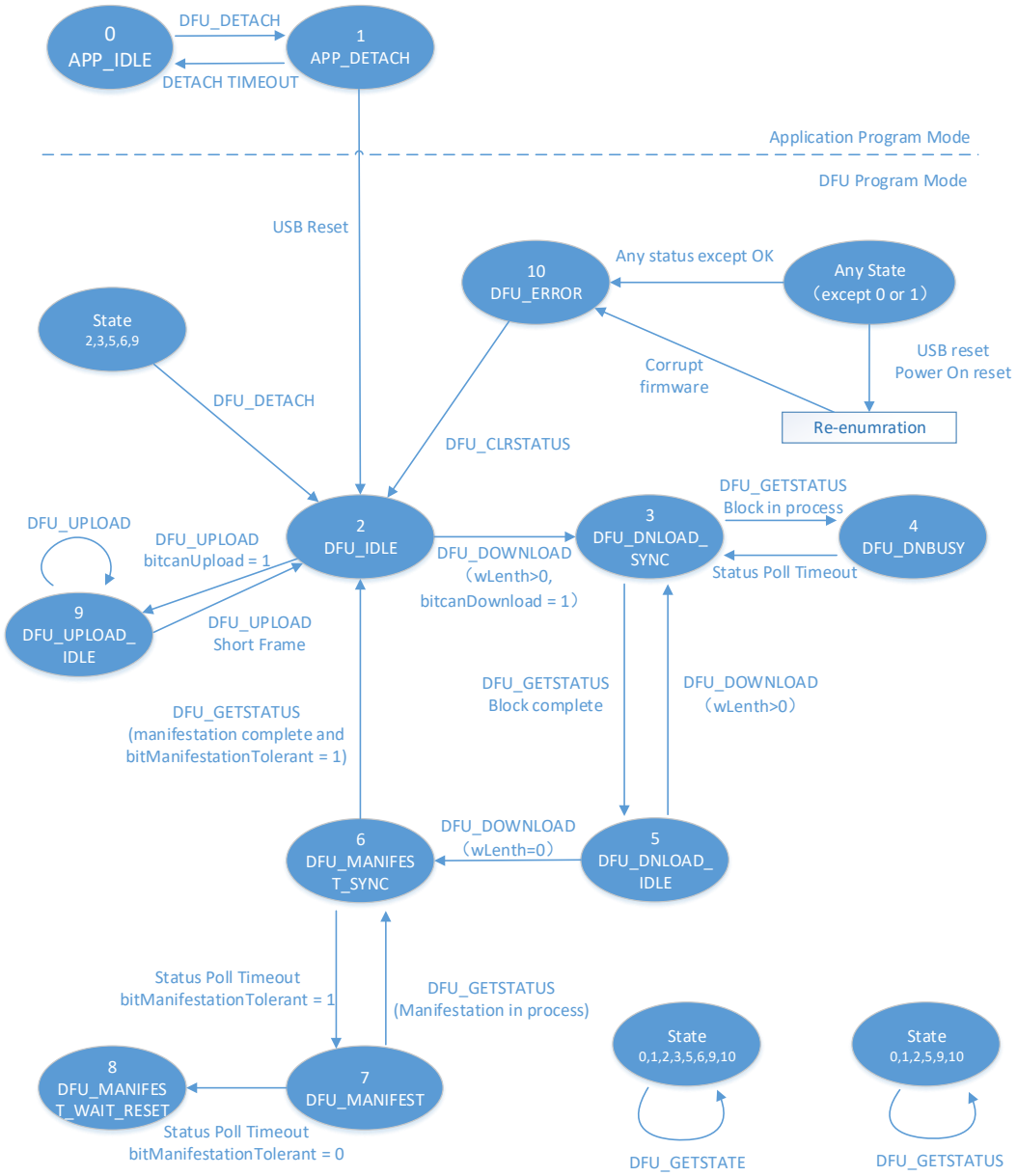
图 5-11. 虚拟串口大数据量收发



5.7.3. 设备固件升级 DFU

DFU 是专门用作固件升级的一套协议，整个数据传输过程使用的都是端点 0，没有用到其他非零端点。整个 DFU 协议流程是有状态机驱动的，状态机流程如*图 5-12. DFU 状态机流程图*所示：

图 5-12. DFU 状态机流程图



DFU 描述符介绍

设备描述符中包含了 DFU 设备的 VID 和 PID，分别为 0x28e9、0x0189。在配置描述符集中，包含了 DFU 相关的描述符项。相关描述符内容如*表 5-12. DFU 相关描述符*所示：

表 5-12. DFU 相关描述符

描述符名称	功能描述
usb_desc_dfu_func	DFU 功能描述符

DFU 设备类接口

DFU 设备类接口如下面结构体所示，结构体的函数实现参见*表 5-13. DFU 设备类接口函数*：

```
usb_class_core dfu_class = {
    .init               = dfu_init,
```

```
    .deinit          = dfu_deinit,
    .req_proc        = dfu_req_handler,
    .ctlx_in         = dfu_ctlx_in
};
```

表 5-13. DFU 设备类接口函数

函数名称	功能描述
dfu_init	初始化 DFU 设备
dfu_deinit	去初始化 DFU 设备
dfu_req_handler	DFU 设备类请求函数
dfu_ctlx_in	IN 控制传输回调

DFU 设备类请求

DFU 包含如*表 5-14. DFU 设备类请求*所示的各个设备类请求：

表 5-14. DFU 设备类请求

请求名称	值	功能描述
DFU_DETACH	0	DFU 分离
DFU_DNLOAD	1	下载
DFU_UPLOAD	2	上传
DFU_GETSTATUS	3	获取状态
DFU_CLRSTATUS	4	清除状态
DFU_GETSTATE	5	获取状况
DFU_ABORT	6	取消

DFU 用户接口

DFU 的用户接口主要是 Flash 操作相关的函数，内容包含在如下结构体中：

```
dfu_mal_prop DFU_Flash_cb =
{
    (const uint8_t *)FLASH_IF_STRING,

    flash_if_init,
    flash_if_deinit,
    flash_if_erase,
    flash_if_write,
    flash_if_read,
    flash_if_checkaddr,
    60, /* flash erase timeout in ms */
    80   /* flash programming timeout in ms (80us * RAM Buffer size (1024 Bytes) */
};
```

各个函数功能如*表 5-15. DFU 用户接口函数*所示：

表 5-15. DFU 用户接口函数

函数名称	功能描述
flash_if_init	存储介质接口初始化
flash_if_deinit	存储介质接口去初始化
flash_if_erase	存储介质擦除操作
flash_if_write	存储介质写操作
flash_if_read	存储介质读操作
flash_if_checkaddr	存储介质地址合法检测
FLASH_IF_STRING	接口字符串

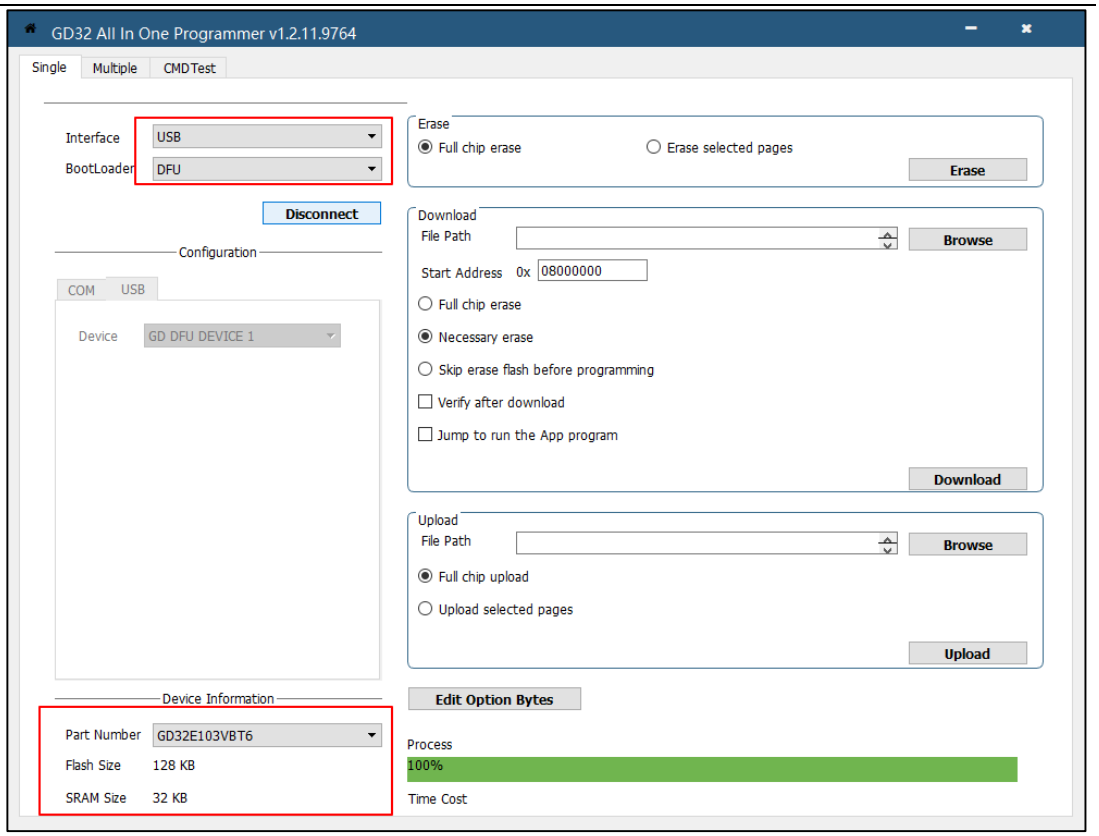
DFU 例程操作指南

将 GD 库中 DFU 例程下载到开发板中，在设备管理器中可以看到如*图 5-13. DFU 设备*新添加的设备。

图 5-13. DFU 设备

打开 GD32AllInOneProgrammer，如果设备成功连接，则在 DFU 界面中会显示连接状态，并显示连接的设备型号。GD32AllInOneProgrammer 的功能主要分为三块：分别是下载、上传和选项字节操作。

图 5-14. AllInOne 连接



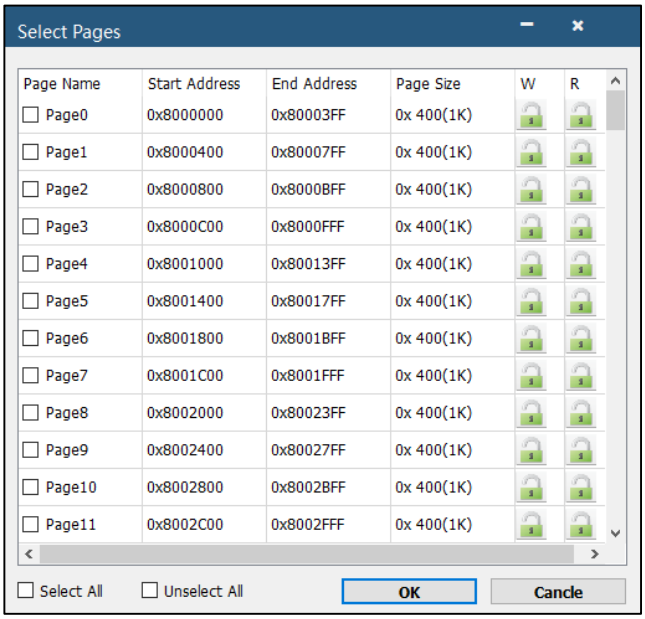
1） 下载

选择需要下载的文件，同时需要设置好对应的下载地址，下载完成后复位芯片，即可执行 APP 程序。

2） 上传

选择需要上传的文件，点击 OK 即可弹出 Select Pages 的界面，选择需要上传的 Page 后即可上传对应的代码。

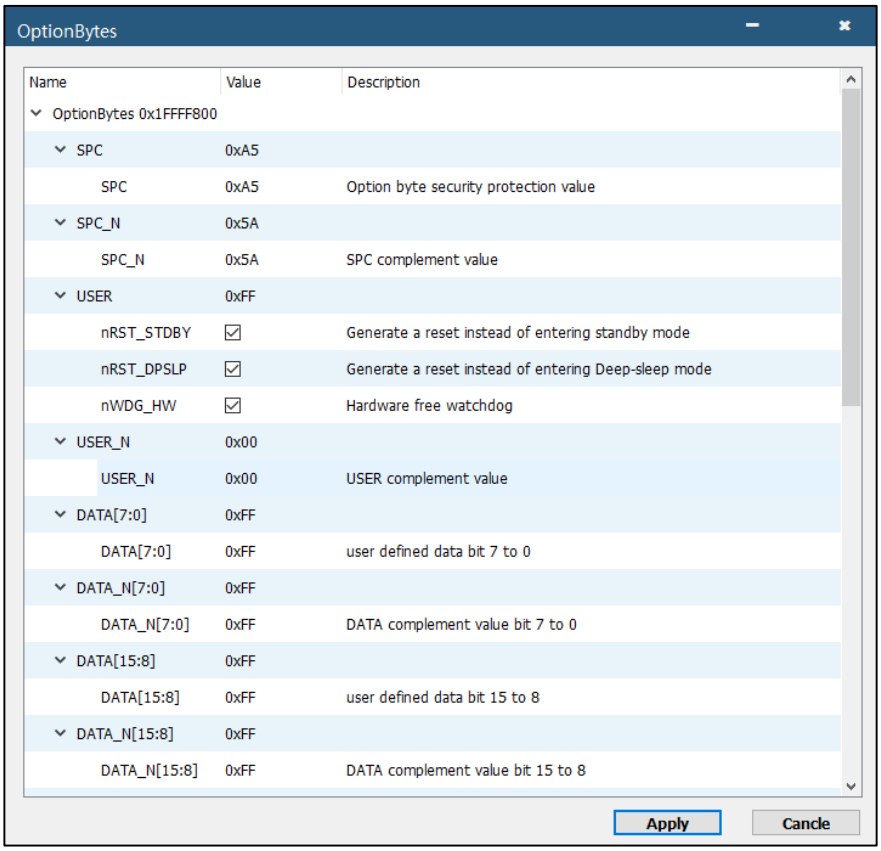
图 5-15. AllInOne 上传



3） 选项字节操作

双击 Edit Option Bytes 处，即可弹出选项字节相关信息，如*图 5-16. AllInOne 选项字节操作*所示。

图 5-16. AllInOne 选项字节操作



5.7.4. 大容量存储设备 MSC

MSC 设备是大容量存储设备，包括 U 盘和 CDROM 等。

MSC 描述符介绍

设备描述符中包含了 MSC 设备的 VID 和 PID，分别为 0x28e9、0x028F。MSC 设备仅包含常用的配置、接口和端点描述符。

MSC 设备类接口

MSC 设备类接口如下面结构体所示，结构体的函数实现参见*表 5-16. MSC 设备类接口函数*：

```
usb_class_core msc_class =
{
    .init      = msc_core_init,
    .deinit    = msc_core_deinit,

    .req_proc = msc_core_req,

    .data_in   = msc_core_in,
    .data_out = msc_core_out
};
```

表 5-16. MSC 设备类接口函数

函数名称	功能描述
msc_core_init	初始化 MSC 设备
msc_core_deinit	去初始化 MSC 设备
msc_core_req	MSC 设备类请求函数
msc_core_in	IN 数据传输回调
msc_core_out	OUT 数据传输回调

MSC 设备类请求

MSC 包含如*表 5-17. MSC 设备类请求*所示的各个设备类请求：

表 5-17. MSC 设备类请求

请求名称	值	功能描述
BBB_GET_MAX_LUN	0xFE	获取最大逻辑单元号
BBB_RESET	0xFF	复位

MSC 用户接口

MSC 的用户接口主要是存储介质的初始化、读写和获取信息等操作，如下列结构体所示：

```
usbd_mem_cb USBD_Internal_Storage_fops =
{
    .mem_init          = STORAGE_Init,
    .mem_ready         = STORAGE_IsReady,
    .mem_protected     = STORAGE_IsWriteProtected,
```

```
    .mem_read         = STORAGE_Read,
    .mem_write        = STORAGE_Write,
    .mem_maxlun       = STORAGE_GetMaxLun,

    .mem_inquiry_data = {(uint8_t *)STORAGE_InquiryData},

    .mem_block_size   = {ISRAM_BLOCK_SIZE},
    .mem_block_len    = {ISRAM_BLOCK_NUM}
};
```

各个函数和变量功能如*表 5-18. MSC 用户接口函数*所示：

表 5-18. MSC 用户接口函数

函数/变量名称	功能描述
STORAGE_Init	存储介质接口初始化
STORAGE_IsReady	存储介质是否准备就绪
STORAGE_IsWriteProtected	存储介质是否写保护
STORAGE_Read	存储介质读操作
STORAGE_Write	存储介质写操作
STORAGE_GetMaxLun	获取支持的逻辑单元数
STORAGE_InquiryData	存储介质标准查询数据
ISRAM_BLOCK_SIZE	存储介质块大小
ISRAM_BLOCK_NUM	存储介质块数量

可以通过修改 ISRAM_BLOCK_SIZE 和 ISRAM_BLOCK_NUM 的值，来确定 U 盘的存储空间大小。

MSC 例程操作指南

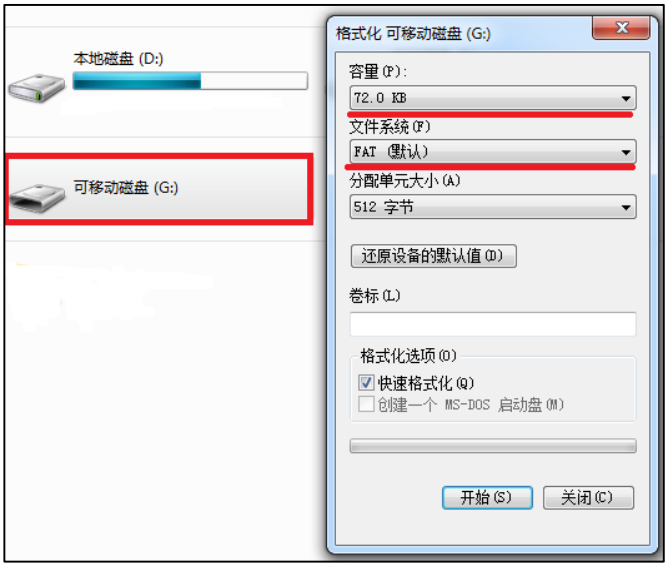
将 GD 库中 MSC 例程下载到开发板中，在设备管理器中可以看到如下新添加的大容量存储设备。

图 5-17. MSC 设备

打开我的电脑可以看到新添加的磁盘，由于磁盘中没有文件系统，所以需要先进行格式化，具

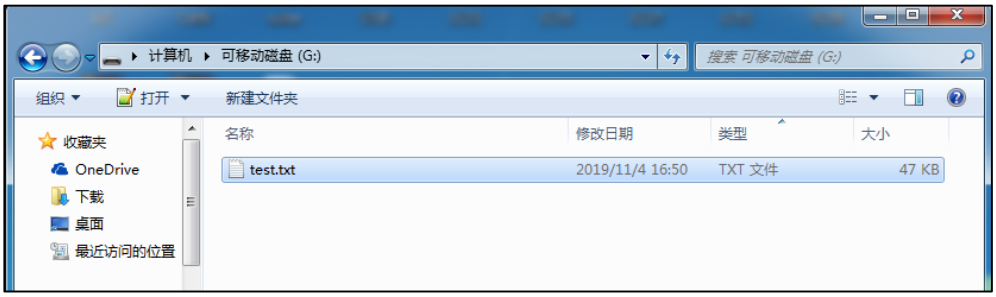
体操作见*图 5-18. MSC 设备格式化*。

图 5-18. MSC 设备格式化



格式化完毕后，可以往磁盘中复制一个测试文件，进行写入测试，再将该文件复制出来，进行读取测试。最后可以将写入和读出的文件进行对比，测试读写过程的正确性。

图 5-19. MSC 设备读写测试



5.7.5. 人机界面设备 HID

HID 设备类是可以实现人机界面交互的各种 USB 设备，除了常用的鼠标、键盘、红外触摸设备外，还可以自定义 HID 设备，使用范围特别广泛。

HID 描述符介绍

设备描述符中包含了 HID 设备的 VID 和 PID，分别为 0x28e9、0x0380。HID 设备类在配置描述符中包含 HID 描述符，并且在除配置描述符集以外，还具有额外的报文描述符。相关描述符内容如*表 5-19. HID 相关描述符*所示：

表 5-19. HID 相关描述符

描述符名称	功能描述
hid_vendor	HID 描述符

描述符名称	功能描述
hid_report_desc	报文描述符

HID 设备类接口

HID 设备类接口如下面结构体所示，结构体的函数实现参见*表 5-20. HID 设备类接口函数*:

```
usb_class_core usbd_hid_cb = {
    .command            = NO_CMD,
    .alter_set          = 0U,
    .init               = hid_init,
    .deinit             = hid_deinit,
    .req_proc           = hid_req,
    .data_in            = hid_data_in
};
```

表 5-20. HID 设备类接口函数

函数名称	功能描述
hid_init	初始化 HID 设备
hid_deinit	去初始化 HID 设备
hid_req	HID 设备类请求函数
hid_data_in	IN 数据传输回调

HID 设备类请求

HID 包含如*表 5-21. HID 设备类请求*所示的各个设备类请求：

表 5-21. HID 设备类请求

请求名称	值	功能描述
GET_REPORT	0x01	获取报告
GET_IDLE	0x02	获取空闲
GET_PROTOCOL	0x03	获取协议
SET_REPORT	0x09	设置报告
SET_IDLE	0x0A	设置空闲
SET_PROTOCOL	0x0B	设置协议

HID 用户接口

以键盘为例，HID 用户接口如下列结构体所示：

```
hid_fop_handler fop_handler = {
    .hid_itf_config = key_config,
    .hid_itf_data_process = hid_key_data_send
};
```

表 5-22. HID 用户接口函数

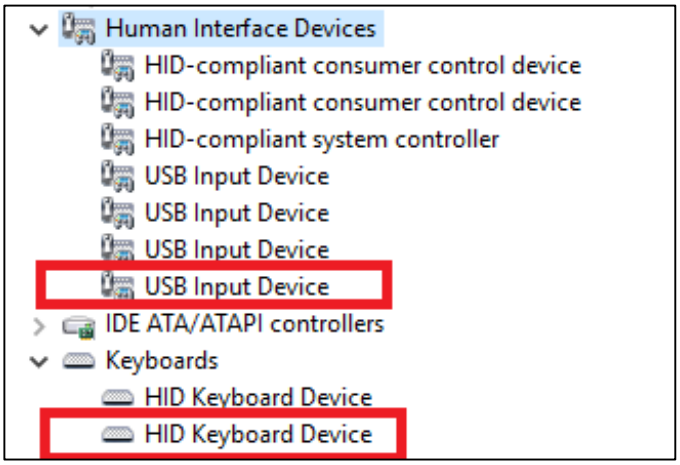
函数/变量名称	功能描述
key_config	按键配置

函数/变量名称	功能描述
hid_key_data_send	键值发送

HID 例程操作指南

将 GD 库中 HID 键盘例程下载到开发板中，在设备管理器中可以看到如*图 5-20. HID 设备*新添加的 HID 输入设备。

图 5-20. HID 设备



按下 Wakeup 键，输出‘b’；按下 Tamper 键，输出‘a’；按下 User 键，输出‘c’。

可利用以下步骤所说明的方法验证 USB 远程唤醒的功能：

—手动将 PC 机切换到睡眠模式；

—等待主机完全进入睡眠模式；

—按下 Wakeup 按键；

—如果 PC 被唤醒，表明 USB 远程唤醒功能正常，否则失败。

5.7.6. USB 打印机

打印机描述符介绍

设备描述符中包含了打印机设备的 VID 和 PID，分别为 0x28e9、0x028D。打印机设备，仅包含常用的配置、接口和端点描述符。

打印机设备类接口

打印机设备类接口如下面结构体所示，结构体的函数实现参见*表 5-23. 打印机设备类接口函数*：

```
usb_class_core usbd_printer_cb = {
    .init           = printer_init,
    .deinit         = printer_deinit,
```

```
    .req_proc        = printer_req,
    .data_in         = printer_in,
    .data_out        = printer_out
};
```

表 5-23. 打印机设备类接口函数

函数名称	功能描述
printer_init	初始化打印机设备
printer_deinit	去初始化打印机设备
printer_req	打印机设备类请求函数
printer_in	IN 数据传输回调
printer_out	OUT 数据传输回调

打印机设备类请求

打印机包含如*表 5-24. 打印机设备类请求*所示的各个设备类请求：

表 5-24. 打印机设备类请求

请求名称	值	功能描述
GET_DEVICE_ID	0x00	获取设备 ID
GET_PORT_STATUS	0x01	获取端口状态
SOFT_RESET	0x02	软件复位

打印机用户接口

目前打印机例程仅实现了枚举部分，数据传输部分受限于打印机的硬件，并没有实现该部分的功能，所以暂没有相关的用户接口。

打印机例程操作指南

将 GD 库中 Printer 例程下载到开发板中，在设备管理器中可以看到如*图 5-21. 打印机设备*新添加的设备。

图 5-21. 打印机设备

6. USBFS 主机库

6.1. 主机库配置

6.1.1. usbh_conf.h

文件配置选项如下：

```
#define USBH_MAX_EP_NUM                 2
#define USBH_MAX_INTERFACES_NUM          2
#define USBH_MAX_ALT_SETTING            2
#define USBH_MAX_SUPPORTED_CLASS         2

#define USBH_DATA_BUF_MAX_LEN           0x200
#define USBH_CFGSET_MAX_LEN             0x200
```

各个配置的定义如*表 6-1. usbh_conf.h 配置说明*:

表 6-1. usbh_conf.h 配置说明

配置名称	功能描述
USBH_MAX_EP_NUM	端点最大数目
USBH_MAX_INTERFACES_NUM	接口最大数目
USBH_MAX_ALT_SETTING	备用接口最大数据
USBH_MAX_SUPPORTED_CLASS	最大支持的设备类数目
USBH_DATA_BUF_MAX_LEN	数据缓存最大长度
USBH_CFGSET_MAX_LEN	配置描述符集最大长度（需要宏来开启）

6.1.2. usb_conf.h

```
#ifdef USB_FS_CORE
    #define USB_RX_FIFO_FS_SIZE                     128
    #define USB_HTX_NPFIFO_FS_SIZE                   96
    #define USB_HTX_PFIFO_FS_SIZE                    96
#endif

#ifdef USB_HS_CORE
    #define USB_RX_FIFO_HS_SIZE                     512
    #define USB_HTX_NPFIFO_HS_SIZE                   256
    #define USB_HTX_PFIFO_HS_SIZE                    256

    #ifdef USE_ULPI_PHY
        #define USB_ULPI_PHY_ENABLED
    #endif

    #ifdef USE_EMBEDDED_PHY
        #define USB_EMBEDDED_PHY_ENABLED
    #endif

//      #define USB_HS_INTERNAL_DMA_ENABLED
#endif

#define USB_SOF_OUTPUT                              0
#define USB_LOW_POWER                               0

//#define USB_LOW_PWR_ENABLE

/****************** USB OTG MODE CONFIGURATION ********************************/
#define USE_HOST_MODE
//#define USE_DEVICE_MODE
//#define USE_OTG_MODE
```

表 6-2. usb_conf.h 配置说明

配置名称	功能描述
USB_RX_FIFO_FS_SIZE	接收 FIFO 大小
USB_HTX_NPFIFO_FS_SIZE	非周期发送 FIFO 大小
USB_HTX_PFIFO_FS_SIZE	周期发送 FIFO 大小
USB_ULPI_PHY_ENABLED	使能 ULPI PHY
USB_EMBEDDED_PHY_ENABLED	使能嵌入式 PHY
USB_SOF_OUTPUT	使能 SOF 输出（PA8 脚）
USB_LOW_POWER	使能低功耗模式
USB_LOW_PWR_ENABLE	使能 VBUS SENSING
USE_HOST_MODE	主机模式

配置名称	功能描述
USE_DEVICE_MODE	设备模式
USE_OTG_MODE	OTG 模式

注意：三种模式只能选择其中一种。

6.2. 主机 VBUS 配置

GD32 开发板上的 USB 主机存在两种类型电路：

1. 通过三极管搭建电路控制 VBUS（包括 F10X/F20X/F30X/F450Z-EVAL）

图 6-1. 三极管搭建电路控制 VBUS

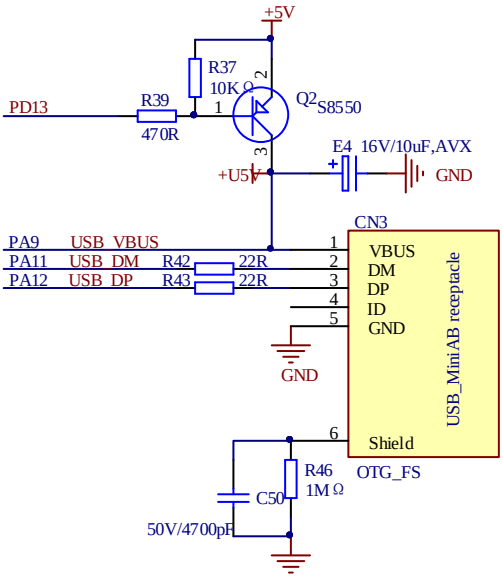


*图 6-1. 三极管搭建电路控制 VBUS*PD13 需要配置为：普通 GPIO 开漏输出（OD）

使能 USB VBUS 输出 5V：PD13 输出低电平（0）

禁止 USB VBUS 输出 5V：PD13 输出高电平（1）

2. 通过逻辑芯片电路控制 VBUS（F450I-EVAL）

图 6-2. 通过逻辑芯片电路控制 VBUS

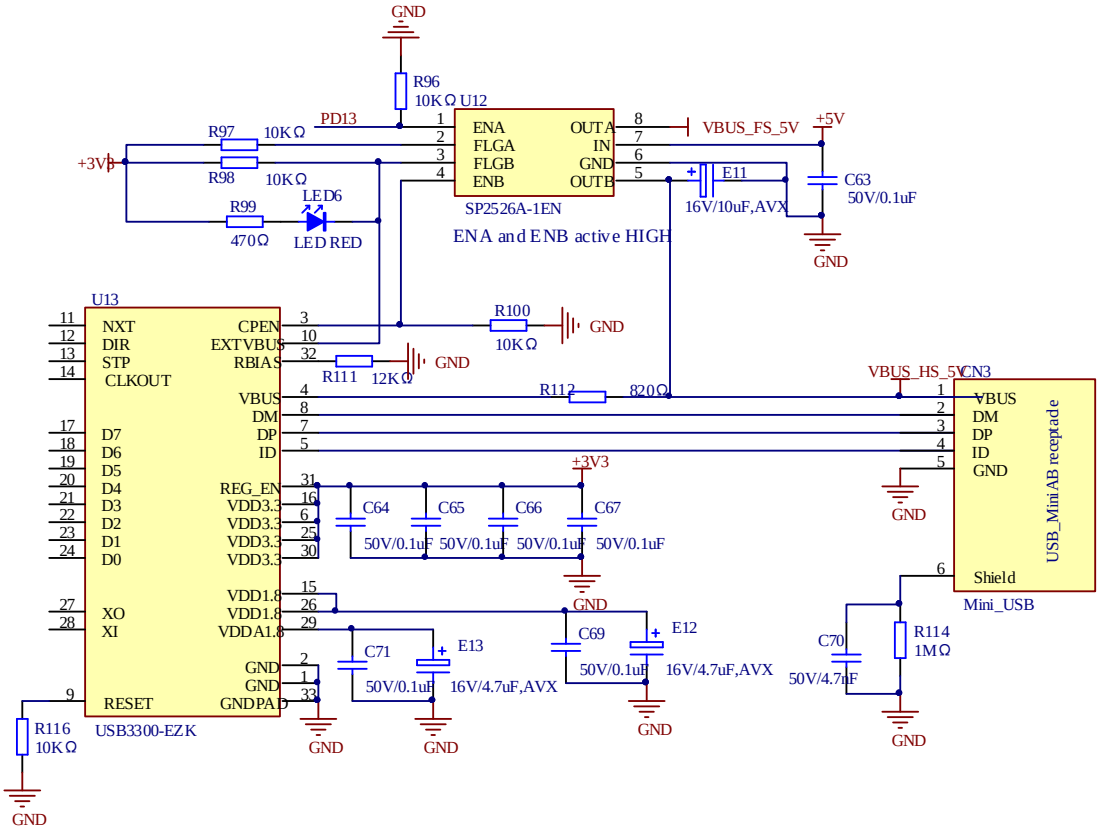


*图 6-2. 通过逻辑芯片电路控制 VBUS*PD13 需要配置为：普通 GPIO 推挽输出（PP）

使能 USB VBUS 输出 5V：PD13 输出高电平（1）

禁止 USB VBUS 输出 5V：PD13 输出低电平（0）

6.3. 中断处理

USBFS 主机接口的全局中断如*表 6-3. USBFS 主机中断*所示，每个中断标志对应着中断函数中一个处理项，和数据传输相关的有 RXFNEIF、NPTXFEIF 和 PTXFEIF。在 USBFS 主机接口中，接收数据 RXFNEIF 来实现，发送数据通过 NPTXFEIF 和 PTXFEIF 实现。

表 6-3. USBFS 主机中断

中断标志	描述	运行模式
WKUPIF	唤醒中断标志位	主机或设备模式
SESIF	会话中断	主机或设备模式
DISCIF	断开连接中断标志	主机模式
IDPSC	ID 引脚状态变化	主机或设备模式
LPMIF(1)	LPM 中断标志	主机或设备模式
PTXFEIF	周期性 Tx FIFO 空中断标志	主机模式
HCIF	主机通道中断标志	主机模式
HPIF	主机端口中断	主机模式
ISOONCIF/PXNCIF	周期性传输未完成中断标志 / 同步 OUT 传输未完成中断标志	主机或设备模式

中断标志	描述	运行模式
NPTXFEIF	非周期 Tx FIFO 空中断标志	主机模式
RXFNEIF	Rx FIFO 非空中断标志	主机或设备模式
SOF	帧起始中断标志	主机或设备模式
OTGIF	OTG 中断标志	主机或设备模式
MFIF	模式错误中断标志	主机或设备模式

数据接收处理主要通过如下函数实现：

```
static uint32_t usbh_int_rxfifonoempty (usb_core_driver *udev)
{
    uint32_t count = 0U;

    __IO uint8_t pp_num = 0U;
    __IO uint32_t rx_stat = 0U;

    /* disable the RX status queue level interrupt */
    udev->regs.gr->GINTEN &= ~GINTEN_RXFNEIE;

    rx_stat = udev->regs.gr->GRSTATP;
    pp_num = (uint8_t)(rx_stat & GRSTATRP_CNUM);

    switch ((rx_stat & GRSTATRP_RPCKST) >> 17U) {
    case GRXSTS_PKTSTS_IN:
        count = (rx_stat & GRSTATRP_BCOUNT) >> 4U;

        /* read the data into the host buffer. */
        if ((count > 0U) && (NULL != udev->host.pipe[pp_num].xfer_buf)) {
            (void)usb_rxfifo_read (&udev->regs, udev->host.pipe[pp_num].xfer_buf,
(uint16_t)count);      // 读 FIFO 内容

            /* manage multiple transfer packet */
            udev->host.pipe[pp_num].xfer_buf += count;
            udev->host.pipe[pp_num].xfer_count += count;

            udev->host.backup_xfercount[pp_num] = udev->host.pipe[pp_num].xfer_count;

            if (udev->regs.pr[pp_num]->HCHLEN & HCHLEN_PCNT) {
                /* re-activate the channel when more packets are expected */
                __IO uint32_t pp_ctl = udev->regs.pr[pp_num]->HCHCTL;

                pp_ctl |= HCHCTL_CEN;
                pp_ctl &= ~HCHCTL_CDIS;

                udev->regs.pr[pp_num]->HCHCTL = pp_ctl;
```

```
            }
        }
        break;

    case GRXSTS_PKTSTS_IN_XFER_COMP:
        break;

    case GRXSTS_PKTSTS_DATA_TOGGLE_ERR:
        count = (rx_stat & GRSTATRP_BCOUNT) >> 4U;

        while (count > 0U) {
            rx_stat = udev->regs.gr->GRSTATP;
            count--;
        }
        break;

    case GRXSTS_PKTSTS_CH_HALTED:
        break;

    default:
        break;
    }

    /* enable the RX status queue level interrupt */
    udev->regs.gr->GINTEN |= GINTEN_RXFNEIE;

    return 1U;
}
```

数据发送处理主要通过如下函数实现：

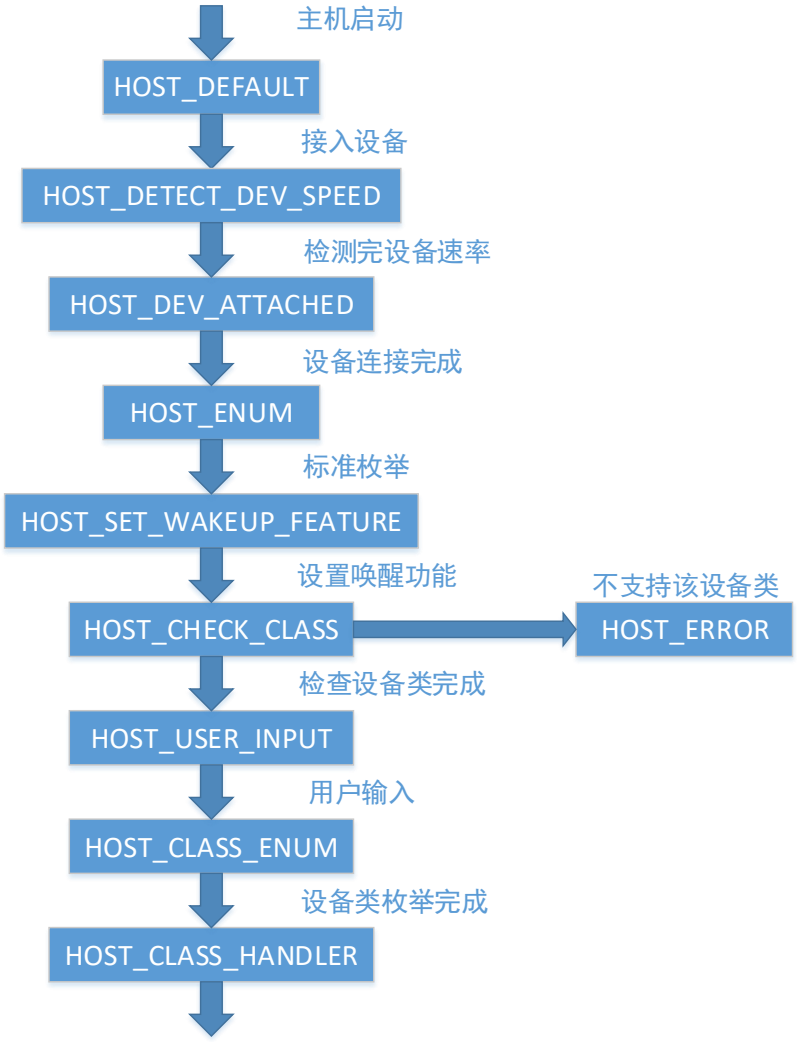
```
static uint32_t usbh_int_txfifoempty (usb_core_driver *udev, usb_pipe_mode pp_mode)
{
    uint8_t pp_num = 0U;
    uint16_t word_count = 0U, len = 0U;
    __IO uint32_t *txfiforeg = 0U, txfifostate = 0U;

    if (PIPE_NON_PERIOD == pp_mode) {
        txfiforeg = &udev->regs.gr->HNPTFQSTAT;
    } else if (PIPE_PERIOD == pp_mode) {
        txfiforeg = &udev->regs.hr->HPTFQSTAT;
    } else {
        return 0U;
    }
```

```
    txfifostate = *txfiforeg;

    pp_num = (uint8_t)((txfifostate & TFQSTAT_CNUM) >> 27U);

    word_count = (uint16_t)(udev->host.pipe[pp_num].xfer_len + 3U) / 4U;

    while (((txfifostate & TFQSTAT_TXFS) >= word_count) && (0U !=
udev->host.pipe[pp_num].xfer_len)) {
        len = (uint16_t)(txfifostate & TFQSTAT_TXFS) * 4U;

        if (len > udev->host.pipe[pp_num].xfer_len) {
            /* last packet */
            len = (uint16_t)udev->host.pipe[pp_num].xfer_len;

            if (PIPE_NON_PERIOD == pp_mode) {
                udev->regs.gr->GINTEN &= ~GINTEN_NPTXFEIE;
            } else {
                udev->regs.gr->GINTEN &= ~GINTEN_PTXFEIE;
            }
        }

        word_count = (uint16_t)((udev->host.pipe[pp_num].xfer_len + 3U) / 4U);
        usb_txfifo_write (&udev->regs, udev->host.pipe[pp_num].xfer_buf, pp_num, len); // 写
FIFO

        udev->host.pipe[pp_num].xfer_buf += len;
        udev->host.pipe[pp_num].xfer_len -= len;
        udev->host.pipe[pp_num].xfer_count += len;

        txfifostate = *txfiforeg;
    }

    return 1U;
}
```

6.4. 状态机流程

USB 对于设备的连接、检测、枚举等操作，都是基于下图的状态机来实现的。状态机的循环执行是在主函数中进行的。

图 6-3. USB 主机状态机



6.5. USB 主机库用户接口

USB主机用户接口定义在下面的结构体中：

```
/* points to the DEVICE_PROP structure of current device */
usbh_user_cb usr_cb =
{
    usbh_user_init,
    usbh_user_deinit,
    usbh_user_device_connected,
    usbh_user_device_reset,
    usbh_user_device_disconnected,
    usbh_user_over_current_detected,
    usbh_user_device_speed_detected,
    usbh_user_device_desc_available,
    usbh_user_device_address_assigned,
    usbh_user_configuration_descavailable,
```

```
    usbh_user_manufacturer_string,
    usbh_user_product_string,
    usbh_user_serialnum_string,
    usbh_user_enumeration_finish,
    usbh_user_userinput,
    usbh_usr_msc_application,
    usbh_user_device_not_supported,
    usbh_user_unrecovered_error
};
```

各个函数功能如*表 6-4. USB 主机库用户接口函数*所示：

表 6-4. USB 主机库用户接口函数

函数名称	功能描述
usbh_user_init	用户接口初始化
usbh_user_deinit	用户接口去初始化
usbh_user_device_connected	设备连接回调函数
usbh_user_device_reset	设备复位回调函数
usbh_user_device_disconnected	设备断开回调函数
usbh_user_over_current_detected	过流检测事件回调函数
usbh_user_device_speed_detected	设备速度检测回调函数
usbh_user_device_desc_available	已获取设备描述符回调函数
usbh_user_device_address_assigned	已分配设备地址回调函数
usbh_user_configuration_descavailable	已获取配置描述符回调函数
usbh_user_manufacturer_string	已获取厂商字符串回调函数
usbh_user_product_string	已获取产品字符串回调函数
usbh_user_serialnum_string	已获取序列号字符串回调函数
usbh_user_enumeration_finish	设备枚举完成回调函数
usbh_user_userinput	用户输入回调函数
usbh_usr_xxx_application	用户应用代码回调函数
usbh_user_device_not_supported	不支持设备时回调函数
usbh_user_unrecovered_error	无法恢复错误发生时回调函数

6.6. USB 主机库设备类接口

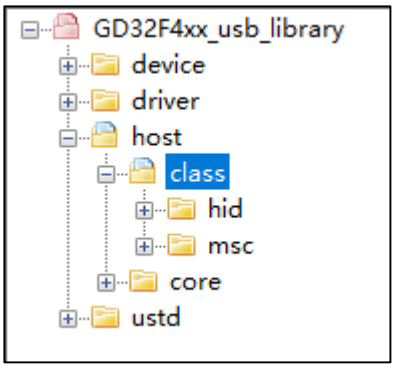
USB 设备类接口由下列的结构体实现：

```
/* device class callbacks */
typedef struct
{
    uint8_t          class_code;          /*!< USB class type */

    usbh_status (*class_init)        (struct _usbh_host *phost);
    void         (*class_deinit)     (struct _usbh_host *phost);
```

```
    usbh_status (*class_requests)   (struct _usbh_host *phost);
    usbh_status (*class_machine)    (struct _usbh_host *phost);
    usbh_status (*class_sof)        (struct _usbh_host *uhost);

    void            *class_data;
} usbh_class;
```

该结构体的初始化工作，由各个设备类单独实现。每个设备类的接口文件保存在 host/class 路径下的设备类文件夹中。

图 6-4. 主机设备类接口文件路径



通过对该结构体进行初始化操作，可以实现设备类的初始化、去初始化、设备类请求和数据传输等工作。

6.6.1. HID 设备类

HID 设备的初始化如下：

```
usbh_class usbh_hid =
{
    USB_HID_CLASS,
    usbh_hid_itf_init,
    usbh_hid_itf_deinit,
    usbh_hid_class_req,
    usbh_hid_handle,
    usbh_hid_sof
};
```

该结构体的初始化函数在 usbh_hid_core.c 文件中，该文件除了包含上述的结构体初始化函数外，还包括其他 HID 设备类相关的函数，另外还有其他几个相关文件实现的函数，总结如*表 6-5. HID 主机类库函数*：

表 6-5. HID 主机类库函数

设备类	文件名称	函数名称	功能描述
HID host class	usbh_hid_core.h/c	usbh_get_report	获取报文
		usbh_set_report	设置报文

设备类	文件名称	函数名称	功能描述
		usbh_hid_device_type_get	获取 HID 设备类型
		usbh_hid_poll_interval_get	获取 HID 设备轮询时间
		usbh_hid_fifo_read	读取 FIFO 数据
		usbh_hid_fifo_write	写入数据至 FIFO
		usbh_hid_fifo_init	初始化 FIFO
		usbh_hiddesc_parse	解析 HID 描述符
		usbh_hid_itf_deinit	去初始化用于 HID 类的主机通道
		usbh_hid_itf_init	初始化 HID 类
		usbh_hid_class_req	处理 HID 类请求
		usbh_hid_handle	HID 处理
		usbh_hid_reportdesc_get	发送获取报文描述符请求
		usbh_hid_sof	SOF 处理
		usbh_hid_desc_get	发送获取 HID 描述符请求
		usbh_set_idle	设置空闲状态
		usbh_set_protocol	设置协议状态
	Usbh_hid_keybd.h/c	usbh_hid_keybd_init	初始化键盘功能
		usbh_hid_keybd_info_get	获取键盘信息
		usbh_hid_ascii_code_get	获取键盘 ASCII 值
		usbh_hid_keybrd_machine	按键处理
		usbh_hid_keybrd_decode	解码按下按键
	usbh_hid_mouse.h/c	usbh_hid_mouse_init	初始化鼠标状态
		usbh_hid_mouse_info_get	获取鼠标信息
		usbh_hid_mouse_machine	鼠标处理
		usbh_hid_mouse_decode	解码鼠标数据
	usbh_hid_parser.h/c	hid_item_read	读取报文条目
		hid_item_write	写入报文条目

6.6.2. MSC 设备类

MSC 设备的初始化如下：

```
usbh_class usbh_msc =
{
    USB_CLASS_MSC,
    usbh_msc_itf_init,
    usbh_msc_itf_deinit,
    usbh_msc_req,
    usbh_msc_handle,
};
```

该结构体的初始化函数在 usbh_msc_core.c 文件中，该文件除了包含上述的结构体初始化函数外，还包括其他 MSC 设备类相关的函数，另外还有其他几个相关文件实现的函数，总结如 *表 6-6. MSC 主机类库函数*：

表 6-6. MSC 主机类库函数

设备类	文件名称	函数名称	功能描述
MSC host class	usbh_msc_bbb.h/c	usbh_msc_init	初始化大容量存储参数
		usbh_msc_bot_process	管理 BOT 传输的不同状态和更新上层状态
		usbh_msc_bot_abort	管理不同错误处理
		usbh_msc_bot_reset	复位 MSC 请求
		usbh_msc_csw_decode	解码设备接收的 CSW
	usbh_msc_core.h/c	usbh_msc_lun_info_get	获取 MSC 设备 LUN 信息
		usbh_msc_read	MSC 读取接口
		usbh_msc_write	MSC 写入接口
		usbh_msc_itf_deinit	去初始化接口，释放接口对应的主机通道
		usbh_msc_itf_init	MSC 接口初始化
		usbh_msc_req	初始化主机状态机
		usbh_msc_handle	处理 MSC 状态机
		usbh_msc_maxlun_get	获取大容量存储设备的最大 LUN
		usbh_msc_rdwr_process	MSC 设备读写处理
	usbh_msc_scsi.c	usbh_msc_scsi_inquiry	发送 Inquiry 命令至设备
		usbh_msc_test_unitready	发送 Test unit ready 命令
		usbh_msc_read_capacity10	发送 read capacity 10 命令
		usbh_msc_mode_sense6	发送 mode sense6 命令
		usbh_msc_request_sense	发送 request sense 命令
		usbh_msc_write10	发送 write10 命令
		usbh_msc_read10	发送 read10 命令
	usbh_msc_fatfs.h/c	disk_initialize	初始化磁盘驱动
		disk_status	获取磁盘状态
		disk_read	读取磁盘块
		disk_write	写磁盘块
		disk_ioctl	I/O 控制功能
		get_fattime	获取 FAT 时间

6.7. USB 主机库例程

6.7.1. HID 主机

HID 主机例程可以用来识别键盘和鼠标，整个枚举过程和数据传输阶段的现象均可以在显示屏上显示。

首先将 OTG 电缆线插入 USB 接口，然后下载 HID_Host 工程至开发板中并运行。

如果一个鼠标被连入，用户将会在显示屏上看到鼠标枚举的信息。按照显示屏的提示进行操作，

将会看到插入的设备是鼠标；然后移动鼠标，将会在液晶上看到鼠标的位置和按键的状态。

如果一个键盘被连入，用户将会看到键盘枚举的信息。按照显示屏的提示进行操作，将会看到插入的设备是键盘，然后按下键盘按键，将会通过液晶显示字符。

图 6-5. HID 主机例程操作示意图

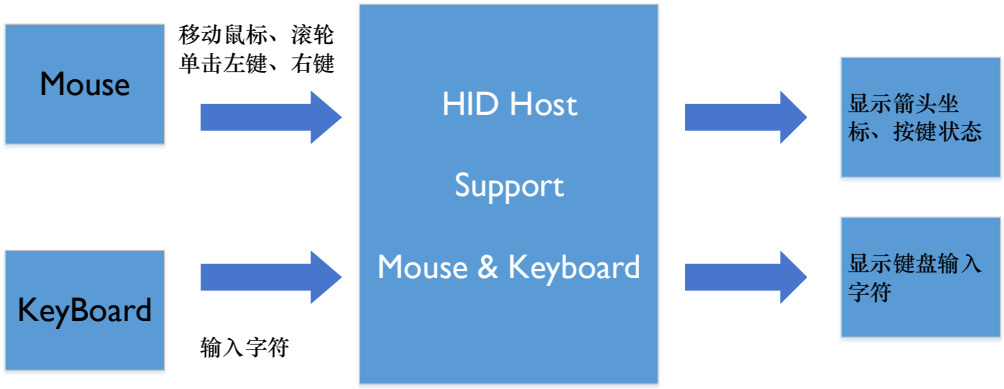
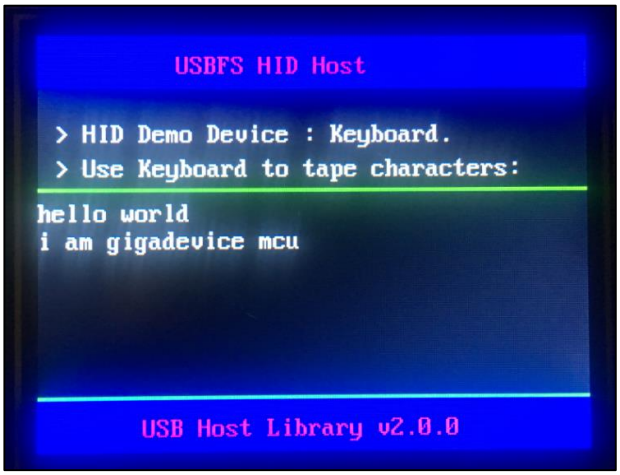


图 6-6. HID 主机例程接鼠标显示

图 6-7. HID 主机例程接键盘显示



6.7.2. MSC 主机

MSC 主机例程可以用来识别 U 盘，整个枚举过程和 U 盘的读写操作，均可以在显示屏上显示。

MSC 主机例程操作步骤如下图所示，首先将 OTG 电缆线插入 USB 接口，然后下载 MSC_Host 工程至开发板中并运行。如果一个 U 盘被连入，用户将会看到 U 盘枚举信息。首先按下 USER 按键将会看到U盘信息；之后按下TAMPER按键将会看到U盘根目录内容；然后按下WAKEUP 按键将会向 U 盘写入文件；最后用户将会看到 MSC 主机示例结束的信息。

图 6-8. MSC 主机例程操作步骤

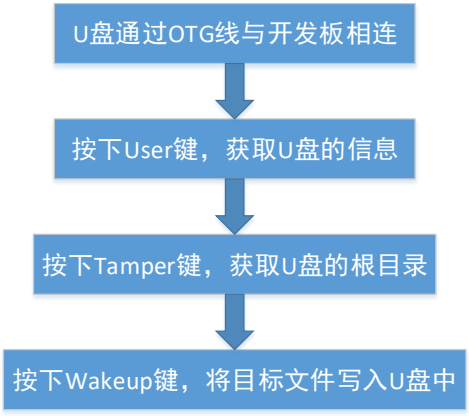
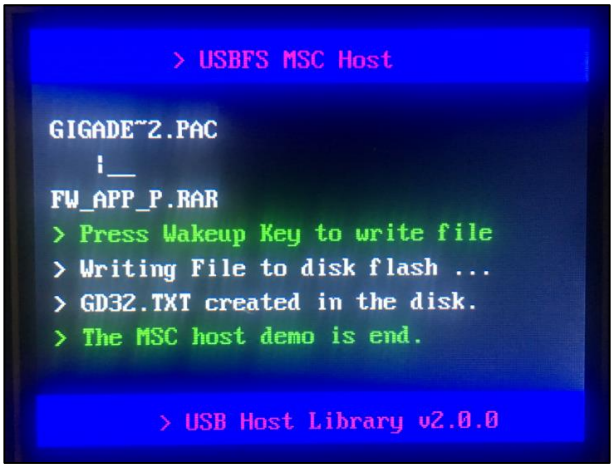


图 6-9. MSC 主机例程显示屏显示



7. 版本历史

表 7-1. 版本历史

版本号	描述	日期
1.0	首次发布	2022 年 3 月 28 日

Important Notice

This document is the property of GigaDevice Semiconductor Inc. and its subsidiaries (the "Company"). This document, including any product of the Company described in this document (the “Product”), is owned by the Company under the intellectual property laws and treaties of the People’s Republic of China and other jurisdictions worldwide. The Company reserves all rights under such laws and treaties and does not grant any license under its patents, copyrights, trademarks, or other intellectual property rights. The names and brands of third party referred thereto (if any) are the property of their respective owner and referred to for identification purposes only.

The Company makes no warranty of any kind, express or implied, with regard to this document or any Product, including, but not limited to, the implied warranties of merchantability and fitness for a particular purpose. The Company does not assume any liability arising out of the application or use of any Product described in this document. Any information provided in this document is provided only for reference purposes. It is the responsibility of the user of this document to properly design, program, and test the functionality and safety of any application made of this information and any resulting product. Except for customized products which has been expressly identified in the applicable agreement, the Products are designed, developed, and/or manufactured for ordinary business, industrial, personal, and/or household applications only. The Products are not designed, intended, or authorized for use as components in systems designed or intended for the operation of weapons, weapons systems, nuclear installations, atomic energy control instruments, combustion control instruments, airplane or spaceship instruments, transportation instruments, traffic signal instruments, life-support devices or systems, other medical devices or systems (including resuscitation equipment and surgical implants), pollution control or hazardous substances management, or other uses where the failure of the device or Product could cause personal injury, death, property or environmental damage ("Unintended Uses"). Customers shall take any and all actions to ensure using and selling the Products in accordance with the applicable laws and regulations. The Company is not liable, in whole or in part, and customers shall and hereby do release the Company as well as it’s suppliers and/or distributors from any claim, damage, or other liability arising from or related to all Unintended Uses of the Products. Customers shall indemnify and hold the Company as well as it’s suppliers and/or distributors harmless from and against all claims, costs, damages, and other liabilities, including claims for personal injury or death, arising from or related to any Unintended Uses of the Products.

Information in this document is provided solely in connection with the Products. The Company reserves the right to make changes, corrections, modifications or improvements to this document and Products and services described herein at any time, without notice.