深入解析 STM32_USB-FS-Device_Lib库

说明：这个是我学习STM32 USB编程时的总结，其中的部分内容是英文文档直接复制过来的，不影响我的阅读，并且我有时也觉得不能准确表达原文的意思，所以我就没有翻译。

声明：该文档只供学习之用，任何用于其他目的的行为，需征得本人同意。

董鸿勇

2009. 12. 15

donghongyong@live.cn

QQ: 262559202

图1 展示了一个典型的USB应用与USB-FS-Device library的关系图。我们可以看出图中由3个层构成分别是：外围硬件(hardware)、STM32_USB-FS_Device_Lib和用户层(User application)。我们从下到上来分析：

Figure 1. USB application hierarchy

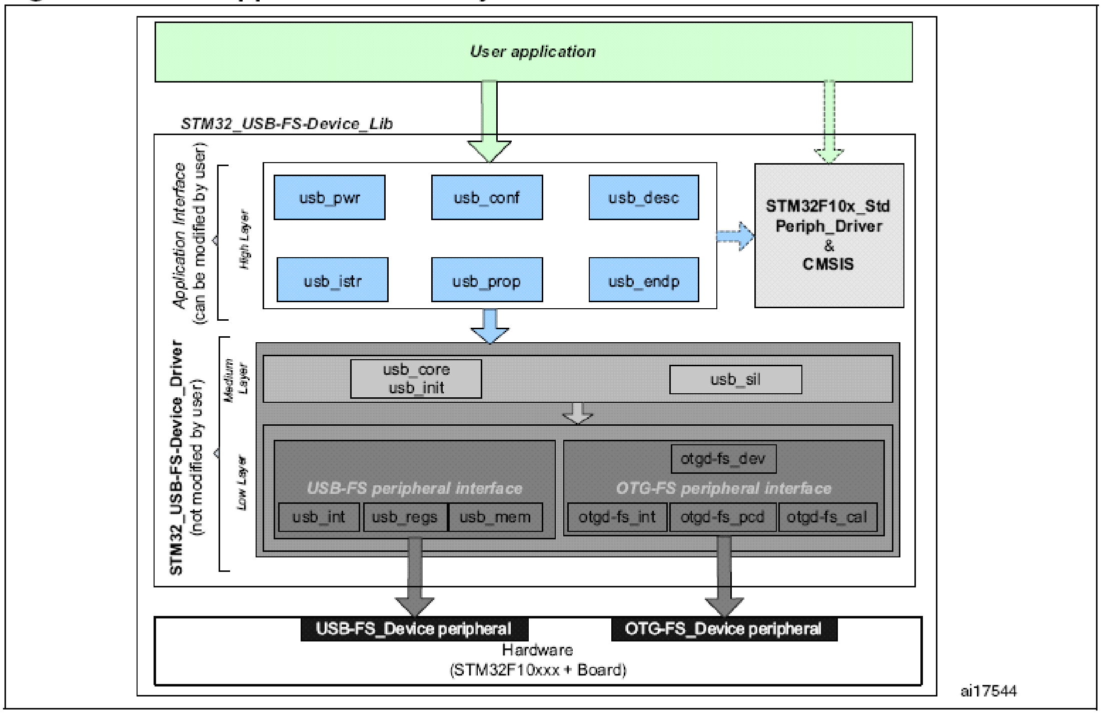

图1 典型的USB应用与USB-FS-Device library的关系图

1.外围硬件(hardware)

就是我们的购买的芯片STM32F10XXX和开发板

2 STM32_USB-FS_Device_Lib

就是STM提供给我们的The USB-FS-Device library固件库，它由STM32_USB FS_Device_Driver和Application Interface layer两个部分组成。

其中STM32_USB-FS_Device_Driver这层管理USB的硬件设备和USB标准协议的直接交互，它又由Low Layer 和 Medium Layer两个层组成；Application Interface layer-High Layer这层又叫High Layer层，它在固件库核和应用提供给用户一个完整的接口。

图2 是我给出的STM32_USB-FS-Device_Lib_V3.1.0 结构图，下面我们将对这个整个结构的运行机理分析，然后结构逐层给出具体含义。

和其他的接口一样，当受到USB的中断后，进入stm32f10x_it.c中的USB_LP_CAN1_RX0_IRQHandler()和USB_HP_CAN1_TX_IRQHandler()中断服务子程序。其中优先级高的由USB_HP_CAN1_TX_IRQHandler处理，优先级低的由USB_LP_CAN1_RX0_ IRQHandler处理。

对于USB_HP_CAN1_TX_IRQHandler函数，它直接调用**usb_int (.h , .c)**中的CTR_HP()，然后根据发送和接受数据，它调用usb_endp(.c)中的EPX_IN_Callback()或EPX_OUT_ Callback()函数。对于EPX_IN_Callback和EPX_OUT_Callback()这14个函数（X=1,

2...7）它们在usb_conf(.h)中通过

#define EPX_IN_Callback NOP_Process

#define EPX_OUT_Callback NOP_Process

的形式，来由用户决定是否提供具体的实现并调用。而将它们和CTR_HP联系在一起的操作，在usb_istr(.h,.c)中以下面的形式给出：

```
void (*pEpInt_IN[7])(void) ={
    EP1_IN_Callback,
    ...
    EP7_IN_Callback,};
void (*pEpInt_OUT[7])(void) ={
    EP1_OUT_Callback,
     ...
    EP7_OUT_Callback,};
```

对于USB_HP_CAN1_TX_IRQHandler函数，它直接调用usb_istr(.h,.c)中的USB_Istr()，USB_Istr()根据具体的请求决定是调用usb_istr(.h,.c)中下面函数

```
void CTR_Callback(void);
void DOVR_Callback(void);
void ERR_Callback(void);
void WKUP_Callback(void);
void SUSP_Callback(void);
void RESET_Callback(void);
void SOF_Callback(void);
void ESOF_Callback(void);
```

还是调用usb_int(.h,.c)中的void CTR_LP(void)。对于上面的这个函数是否给出定义，是由用户在usb_conf(.h)中，通过下面的宏决定的

```
/*#define CTR_CALLBACK*/
/*#define DOVR_CALLBACK*/
/*#define ERR_CALLBACK*/
/*#define WKUP_CALLBACK*/
/*#define SUSP_CALLBACK*/
/*#define RESET_CALLBACK*/
#define SOF_CALLBACK
/*#define ESOF_CALLBACK*/
```

如果调用了 CTR_LP()函数，CTR_LP()函数中如果不是端点0的请求，则和CTR_LP一样的顺序处理；如果是端点0，它调用usb_core(.h,.c)中的

```
uint8_t Setup0_Process(void);
```

uint8_t Post0_Process(void);

uint8_t Out0_Process(void);

uInt8_t In0_Process(void);

如果是标准的请求，便调用usb_core(.h,.c)中的下面的函数

RESULT Standard_SetEndPointFeature(void);

RESULT Standard_SetDeviceFeature(void);

uint8_t *Standard_GetConfiguration(uint16_t Length);

RESULT Standard_SetConfiguration(void);

uint8_t *Standard_GetInterface(uint16_t Length);

RESULT Standard_SetInterface(void);

Uint8_t *Standard_GetDescriptorData(uint16_t ...);

uint8_t *Standard_GetStatus(uint16_t Length);

RESULT Standard_ClearFeature(void);

void SetDeviceAddress(uint8_t);

这些函数，又调用USER_STANDARD_REQUESTS结构指定的中，用户在usb_prop(.h,.c)中定义的函数。如果不是标准请求，则调用DEVICE_PROP结构指定的中，用户在usb_prop(.h,.c)中定义的函数其他一些函数。

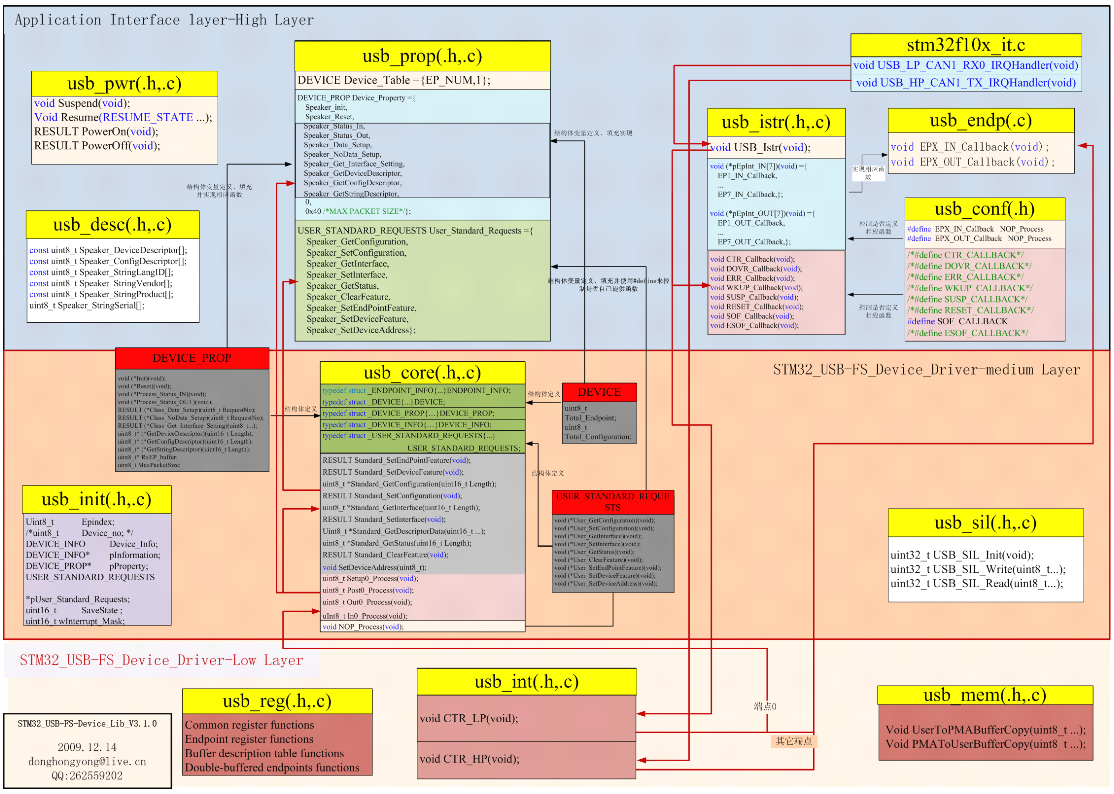

图2 STM32_USB-FS-Device_Lib_V3.1.0 结构图

下面是上面各个模块的解释，基本都是来自**STM32**的官网**UM0424** 用户手册翻译。

2.1 STM32_USB-FS_Device_Driver-Low Layer

对于USB-FS_Device来说Low Layer就是USB-FS peripheral interface，它由 Table 1中模块组成。

Table 1. USB-FS_Device peripheral interface modules

File	Description
usb_reg (.h, .c)	硬件抽象层 Hardware abstraction layer
usb_int.c	正确传输中断服务线程 Correct transfer interrupt service routine
usb_mem(.h,.c)	数据传输管理 Data transfer management (from/to packet memory area)

2.1.1 usb_reg(.h, .c)

usb_regs 模块实现了硬件抽象层，它提供了一个存取 USB-FS_Device 外围设备寄存器的函数集合。*这个集合*包括 Common register functions、Endpoint register functions、Buffer description table functions 和 Double-buffered endpoints functions 四个函数级。

注意: 这些函数集合可以用宏的形式和函数的形式调用:

– 宏形式: _NameofFunction(parameter1,...)

– 函数形式: NameofFunction(parameter1,...)

2.1.1.1 Common register functions:

这些函数可以用来设置和获得USB-FS_Device外围普通寄存器的值；其寄存器可以是CNTR、ISTR、FNR、DADDR、BTABLE。

Table 2. Common register functions

Register	function
CNTR	void SetCNTR (uint16_t wValue)
	uint16_t GetCNTR (void)
ISTR	void SetISTR (uint16_t wValue)
	uint16_t GetISTR (void)
FNR	uint16_t GetFNR (void)
DADDR	void SetDADDR (uint16_t wValue)
	uint16_t GetDADDR (void)
BTABLE	void SetBTABLE (uint16_t wValue)
	uint16_t GetBTABLE (void)

2.1.1.2 Endpoint register functions:

所有和端点寄存器(Endpoint register)相关的操作都可以用SetENDPOINT and GetENDPOINT 函数来完成。而且，还有一些继承自它们的函数可以在特定的域（field）提供一个直接而快速的操作。

a) **Endpoint set/get value**

SetENDPOINT : void SetENDPOINT(uint8_t bEpNum,uint16_t wRegValue)

bEpNum = Endpoint number, wRegValue = Value to write

GetENDPOINT : uint16_t GetENDPOINT(uint8_t bEpNum)

bEpNum = Endpoint number

return value: the endpoint register value

b) **Endpoint TYPE field**

USB_EPnR 寄存器分布

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
CTR_RX	DTOG_RX	STAT_RX[1:0]		SETUP	EP TYPE[1:0]		EP_KIND	CTR_TX	DTOG_TX	STAT_TX[1:0]		EA[3:0]			
rc w0	t	t	t	r	rw	rw	rw	rc w0	t	t	t	rw	rw	rw	rw

其中EP_TYPE根据SB_EPnR中的位置和端点类型编码给出下面的宏定义:

#define EP_BULK (0x0000) // Endpoint BULK

#define EP_CONTROL (0x0200) // Endpoint CONTROL

#define EP_ISOCHRNOUS (0x0400) // Endpoint ISOCHRONOUS

#define EP_INTERRUPT (0x0600) // Endpoint INTERRUPT

端点类型编码

EP_TYPE[1:0]	描述
00	**BULK**：批量端点
01	**CONTROL**：控制端点
10	**ISO**：同步端点
11	**INTERRUPT**：中断端点

SetEPType : void SetEPType (uint8_t bEpNum, uint16_t wtype)

bEpNum = Endpoint number, wtype = Endpoint type (value from the above define's)

GetEPType : uint16_t GetEPType (uint8_t bEpNum)

bEpNum = Endpoint number

return value: a value from the above define's

c) Endpoint STATUS field

其中STAT_TX / STAT_RX根据SB_EPnR中的位置和端点类型编码给出下面的宏定义:

#define EP_TX_DIS (0x0000) // Endpoint TX DISabled

#define EP_TX_STALL (0x0010) // Endpoint TX STALLed

#define EP_TX_NAK (0x0020) // Endpoint TX NAKed

#define EP_TX_VALID (0x0030) // Endpoint TX VALID

#define EP_RX_DIS (0x0000) // Endpoint RX DISabled

#define EP_RX_STALL (0x1000) // Endpoint RX STALLed

#define EP_RX_NAK (0x2000) // Endpoint RX NAKed

#define EP_RX_VALID (0x3000) // Endpoint RX VALID

接收状态编码

STAT_RX[1:0]	描述
00	**DISABLED**：端点忽略所有的接收请求。
01	**STALL**：端点以STALL分组响应所有的接收请求。
10	**NAK**：端点以NAK分组响应所有的接收请求。

11	**VALID**：端点可用于接收。

发送状态编码

STAT_RX[1:0]	描述
00	**DISABLED**：端点忽略所有的发送请求。
01	**STALL**：端点以STALL分组响应所有的发送请求。
10	**NAK**：端点以NAK分组响应所有的发送请求。
11	**VALID**：端点可用于发送。

SetEPTxStatus : void SetEPTxStatus(uint8_t bEpNum,uint16_t wState)
SetEPRxStatus : void SetEPRxStatus(uint8_t bEpNum,uint16_t wState)
bEpNum = Endpoint number, wState = a value from the above define's
GetEPTxStatus : uint16_t GetEPTxStatus(uint8_t bEpNum)
GetEPRxStatus : uint16_t GetEPRxStatus(uint8_t bEpNum)
bEpNum = endpoint number
return value:a value from the above define's

d) Endpoint KIND field

SetEP_KIND : void SetEP_KIND(uint8_t bEpNum)
ClearEP_KIND : void ClearEP_KIND(uint8_t bEpNum)
bEpNum = endpoint number
Set_Status_Out : void Set_Status_Out(uint8_t bEpNum)
Clear_Status_Out : void Clear_Status_Out(uint8_t bEpNum)
bEpNum = endpoint number
注意：在usb_regs.h中 #define _Set_Status_Out(bEpNum) _SetEP_KIND(bEpNum)
#define _Clear_Status_Out(bEpNum) _ClearEP_KIND(bEpNum)
SetEPDoubleBuff : void SetEPDoubleBuff(uint8_t bEpNum)
ClearEPDoubleBuff : void ClearEPDoubleBuff(uint8_t bEpNum)
bEpNum = endpoint number
注意：在usb_regs.h中 #define _SetEPDoubleBuff(bEpNum) _SetEP_KIND(bEpNum)
#define _ClearEPDoubleBuff(bEpNum) _ClearEP_KIND(bEpNum)

e) Correct Transfer Rx/Tx fields

ClearEP_CTR_RX : void ClearEP_CTR_RX(uint8_t bEpNum)
ClearEP_CTR_TX : void ClearEP_CTR_TX(uint8_t bEpNum)
bEpNum = endpoint number

f) Data Toggle Rx/Tx fields

ToggleDTOG_RX : void ToggleDTOG_RX(uint8_t bEpNum)
ToggleDTOG_TX : void ToggleDTOG_TX(uint8_t bEpNum)
bEpNum = endpoint number

g) Address field 端点地址

SetEPAdress : void SetEPAddress(uint8_t bEpNum,uint8_t bAddr)
bEpNum = endpoint number
bAddr = address to be set
GetEPAdress : uint8_t GetEPAddress(uint8_t bEpNum)
bEpNum = endpoint number

2.1.1.3 Buffer description table functions

这些函数用来设置和获得端点接受和发送缓冲区的地址和大小。

a) Tx/Rx buffer address fields

SetEPTxAddr : void SetEPTxAddr(uint8_t bEpNum,uint16_t wAddr);

SetEPRxAddr : void SetEPRxAddr(uint8_t bEpNum,uint16_t wAddr);

bEpNum = endpoint number

wAddr = address to be set (expressed as PMA buffer address)

GetEPTxAddr : uint16_t GetEPTxAddr(uint8_t bEpNum);

GetEPRxAddr : uint16_t GetEPRxAddr(uint8_t bEpNum);

bEpNum = endpoint number

return value : address value (expressed as PMA buffer address)

b) Tx/Rx buffer counter fields

SetEPTxCount : void SetEPTxCount(uint8_t bEpNum,uint16_t wCount);

SetEPRxCount : void SetEPRxCount(uint8_t bEpNum,uint16_t wCount);

bEpNum = endpoint number

wCount = counter to be set

GetEPTxCount : uint16_t GetEPTxCount(uint8_t bEpNum);

GetEPRxCount : uint16_t GetEPRxCount(uint8_t bEpNum);

bEpNum = endpoint number

return value : counter value

2.1.1.4 Double-buffered endpoints functions

在批量和同步传输中，为了获得大数据传输吞吐量，*double-buffered* 模式必须被编程。在这个操作模式，一些端点寄存器的域和缓冲区描述表单元与单缓冲区模式有不同的含义。为了更加容易的使用这个模式，一些函数被设计出来。

SetEPDoubleBuff() ：一个工作在批量模式的端点可以通过设置EP-KIND位来将其设置成双缓冲模式，SetEPDoubleBuff() 这个函数及可以完成这样的任务。

FreeUserBuffer:在double-buffered模式，该端点变成单向端点，并且那个不使用方向(接收和发送)的的缓冲区被用做使用方向的第二个缓冲区。

地址和计数器必须被用不同的方式处理。 Rx和Tx 地址和计数器单元变成Buffer0 and Buffer1单元。并且，在库中直接提供这样操作功能的函数。

在批量传输中，USB模块填充一个缓冲区的同时，另一个缓冲区为应用程序服务。应用程序必须在下一个批量传输需要下一个缓冲区前处理完数据。这个服务于用户应用程序必须被及时的释放。

FreeUserBuffer就是提供用来释放应用程序使用了的缓冲区的。

FreeUserBuffer: void FreeUserBuffer(uint8_t bEpNum, uint8_t bDir);

bEpNum = endpoint number

a) Double buffer addresses

这些函数用来在double buffered模式下，获得和设置缓冲区描述表中缓冲区的地址值。

SetEPDblBuffAddr : void SetEPDblBuffAddr(uint8_t bEpNum,uint16_t wBuf0Addr,uint16_t wBuf1Addr);

SetEPDblbuf0Addr : void SetEPDblBuf0Addr(uint8_t bEpNum,uint16_t wBuf0Addr);

SetEPDblbuf1Addr : void SetEPDblBuf1Addr(uint8_t bEpNum,uint16_t wBuf1Addr);
bEpNum = endpoint number
wBuf0Addr, wBuf1Addr = buffer addresses (expressed as PMA buffer addresses)

GetEPDblBuf0Addr : uint16_t GetEPDblBuf0Addr(uint8_t bEpNum);
GetEPDblbuf1Addr : uint16_t GetEPDblBuf1Addr(uint8_t bEpNum);
bEpNum = endpoint number
return value : buffer addresses

b) Double buffer counters

这些函数用来在double buffered模式下，获得和设置缓冲区描述表中缓冲区的计数器值。

SetEPDblBuffCount: void SetEPDblBuffCount(uint8_t bEpNum, uint8_t bDir, uint16_t wCount);
SetEPDblBuf0Count: void SetEPDblBuf0Count(uint8_t bEpNum, uint8_t bDir, uint16_t wCount);
SetEPDblBuf1Count: void SetEPDblBuf1Count(uint8_t bEpNum, uint8_t bDir, uint16_t wCount);
bEpNum = endpoint number
bDir = endpoint direction
wCount = buffer counter

GetEPDblBuf0Count : uint16_t GetEPDblBuf0Count(uint8_t bEpNum);
GetEPDblBuf1Count : uint16_t GetEPDblBuf1Count(uint8_t bEpNum);
bEpNum = endpoint number
return value : buffer counter

c) Double buffer STATUS

The simple and double buffer modes use the same functions to manage the Endpoint STATUS except for the STALL status for double buffer mode. This functionality is managed by the function:

单缓冲区和双缓冲区模式使用相同的函数去管理端点STATUS，除了STALL状态。这个功能是被下面的函数管理。

SetDouBleBuffEPStall: void SetDouBleBuffEPStall(uint8_t bEpNum,uint8_t bDir)
bEpNum = endpoint number
bDir = endpoint direction

2.1.2 usb_int (.h , .c)

usb_int 模块处理正确传输中断服务程序；它提供了USB协议事件与这个库的连接。STM32F10xxx USB-FS_Device 外围设备提供两个正确传输处理函数：

•低优先级中断Low-priority interrupt:.被 CTR_LP()函数管理，用于控制模式、中断模式和批量模式（单缓冲区）。

•高优先级中断 High-priority interrupt:被 CTR_HP()函数管理，用于快速传输模式，像同步模式和批量模式（双缓冲区）。

2.1.3 usb_mem (.h , .c)

usb_mem模块负责拷贝数据从用户内存区（user memory area）到USB模块内存区（packet memory area）（PMA）或者从USB模块内存区（packet memory area）（PMA）到用户内存区（user memory area）。它提供两个不同的函数：

void UserToPMABufferCopy(uint8_t *pbUsrBuf,uint16_t wPMABufAddr, uint16_t wNBytes);
void PMAToUserBufferCopy(uint8_t *pbUsrBuf,uint16_t wPMABufAddr, uint16_t wNBytes);

2.2 STM32_USB-FS_Device_Driver-medium Layer

Table 2. USB-FS-Device_Driver medium layer modules

File	Description
usb_init (.h,.c)	USB device initialization global variables
usb_core (.h , .c)	USB protocol management (compliant with chapter 9 of the *USB 2.0 specification*)
usb_sil (.h,.c)	Simplified functions for read & write accesses to the endpoints (abstraction layer for both USB-FS_Device and OTG-FS_Device peripherals)
usb_def.h/usb_type.h	USB definitions and Ttypes used in the library

2.2.1 usb_init(.h,.c)

usb_init模块设置在整个库中用到的usb初始化函数和全局变量。

//函数

void USB_Init(void);

//变量

uint8_t EPindex;/* The number of current endpoint, it will be used to specify an endpoint */

/*uint8_t Device_no; *//* The number of current device, it is an index to the Device_Table */

DEVICE_INFO Device_Info;

DEVICE_INFO* pInformation; /* Points to the DEVICE_INFO structure of current device */
/* The purpose of this register is to speed up the execution */

DEVICE_PROP* pProperty; /* Points to the DEVICE_PROP structure of current device */
/ * The purpose of this register is to speed up the execution */

USER_STANDARD_REQUESTS *pUser_Standard_Requests;

uint16_t SaveState ; /* Temporary save the state of Rx & Tx status. */
/* Whenever the Rx or Tx state is changed, its value is saved */
/* in this variable first and will be set to the EPRB or EPRA */
/* at the end of interrupt process */

uint16_t wInterrupt_Mask;

2.2.2 usb_core (.h , .c)

usb_core模块是这个库的“核”，它实现了USB 2.0 规范第9章中描述的所有函数。

模块中的一些子程序处理控制断点(ENDP0)的USB标准请求，提供必须的代码去完成setup枚举阶段的顺序请求。

A state machine被实现，为了去处理setup传输不同阶段的请求。

USB核模块也用**User_Standard_Requests**结构，在标准的请求和用户实现之间，实现一个动态的接口。

当需要的时候，USB核分发一些类的特定请求和一些总线事件给用户程序。这些处理程序在 **Device_Property**中指定。

这些被和用到的数据结构和函数，在下面的文章中详细的描述：

1. Device table structure

The core keeps device level information in the **Device_Table** structure. **Device_Table** is of the type: **DEVICE**.

核将设备级（ device level）信息保存在**Device_Table**结构体中。**Device_Table**是**DEVICE**类型。

```c
typedef struct _DEVICE
{
  uint8_t Total_Endpoint;        /* Number of endpoints that are used */
  uint8_t Total_Configuration    ;/* Number of configuration available */
}
DEVICE;
```

2. Device information structure

The USB core keeps the setup packet from the host for the implemented USB Device in the **Device_Info** structure. This structure has the type: **DEVICE_INFO**.

USB核将从主机获得的setup包信息保存在**Device_Info**结构体中。**Device_Info**的类型是**DEVICE_INFO**。

```c
typedef struct _DEVICE_INFO
{
  uint8_t USBbmRequestType;          /* bmRequestType */
  uint8_t USBbRequest;               /* bRequest */
  uint16_t_uint8_t USBwValues;       /* wValue */
  uint16_t_uint8_t USBwIndexs;       /* wIndex */
  uint16_t_uint8_t USBwLengths;      /* wLength */

  uint8_t ControlState;              /* of type CONTROL_STATE */
  uint8_t Current_Feature;
  uint8_t Current_Configuration;     /* Selected configuration */
  uint8_t Current_Interface;         /* Selected interface of current configuration */
  uint8_t Current_AlternateSetting; /* Selected Alternate Setting of current interface*/
  ENDPOINT_INFO Ctrl_Info;
}DEVICE_INFO;
```

在**DEVICE_INFO**中，联合体**uint16_t_uint8_t**被定义，它可以容易的存取**uint16_t** 或者**uint8_t** 格式的数据。

```c
typedef union
{
  uint16_t w;
  struct BW
  {
    uint8_t bb1;
    uint8_t bb0;
  }bw;
} uint16_t_uint8_t;
```

Description of the structure fields:

- **USBbmRequestType** is the copy of the *bmRequestType* of a setup packet
- **USBbRequest** is the copy of the *bRequest* of a setup packet
- **USBwValues** is defined as type: **uint16_t_uint8_t** and can be accessed through 3 macros:

```c
#define USBwValue USBwValues.w
#define USBwValue0 USBwValues.bw.bb0
```

#define USBwValue1 USBwValues.bw.bb1

USBwValue is the copy of *the wValue* of a setup packet

USBwValue0 is the low byte of *wValue*, and **USBwValue1** is the high byte of *wValue*.

– **USBwIndexs** is defined as USBwValues and can be accessed by 3 macros:

#define USBwIndex USBwIndexs.w

#define USBwIndex0 USBwIndexs.bw.bb0

#define USBwIndex1 USBwIndexs.bw.bb1

USBwIndex is the copy of the *wIndex* of a setup packet

USBwIndex0 is the low byte of *wIndex*, and **USBwIndex1** is the high byte of *wIndex*.

– **USBwLengths** is defined as type: **uint16_t_uint8_t** and can be accessed through 3 macros:

#define USBwLength USBwLengths.w

#define USBwLength0 USBwLengths.bw.bb0

#define USBwLength1 USBwLengths.bw.bb1

USBwLength is the copy of the *wLength* of a setup packet

USBwLength0 and **USBwLength1** are the low and high bytes of *wLength*, respectively.

– **ControlState** is the state of the core, the available values are defined in CONTROL_STATE.

– **Current_Feature** is the device feature at any time. It is affected by the SET_FEATURE and CLEAR_FEATURE requests and retrieved by the GET_STATUS request. User code does not use this field.

– **Current_Configuration** is the configuration the device is working on at any time. It is set and retrieved by the SET_CONFIGURATION and GET_CONFIGURATION requests, respectively.

– **Current_Interface** is the selected interface.

– **Current_Alternatesetting** is the alternative setting which has been selected for the current working configuration and interface. It is set and retrieved by the SET_INTERFACE and GET_INTERFACE requests, respectively.

– **Ctrl_Info** has type ENDPOINT_INFO.

Since this structure is used everywhere in the library, a global variable **pInformation** is defined for easy access to the **Device_Info** table, it is a pointer to the **DEVICE_INFO** structure. Actually, **pInformation = &Device_Info**.

3. Device property structure

The USB core dispatches the control to the user program whenever it is necessary.
User handling procedures are given in an array of **Device_Property**. The structure has the type: **DEVICE_PROP**:

```
typedef struct _DEVICE_PROP
{
  void (*Init)(void);               /* Initialize the device */
  void (*Reset)(void);              /* Reset routine of this device */
  void (*Process_Status_IN)(void);       /* Device dependent process after the status stage */
  void (*Process_Status_OUT)(void);
  /* Procedure of process on setup stage of a class specified request with data stage */
  /* All class specified requests with data stage are processed in Class_Data_Setup
    Class_Data_Setup()
```

```
    responses to check all special requests and fills ENDPOINT_INFO
    according to the request
    If IN tokens are expected, then wLength & wOffset will be filled
    with the total transferring bytes and the starting position
    If OUT tokens are expected, then rLength & rOffset will be filled
    with the total expected bytes and the starting position in the buffer
    If the request is valid, Class_Data_Setup returns SUCCESS, else UNSUPPORT

   CAUTION:
    Since GET_CONFIGURATION & GET_INTERFACE are highly related to
    the individual classes, they will be checked and processed here.
  */
  RESULT (*Class_Data_Setup)(uint8_t RequestNo);

  /* Procedure of process on setup stage of a class specified request without data stage */
  /* All class specified requests without data stage are processed in Class_NoData_Setup
   Class_NoData_Setup
    responses to check all special requests and perform the request

   CAUTION:
    Since SET_CONFIGURATION & SET_INTERFACE are highly related to
    the individual classes, they will be checked and processed here.
  */
  RESULT (*Class_NoData_Setup)(uint8_t RequestNo);

  /*Class_Get_Interface_Setting
   This function is used by the file usb_core.c to test if the selected Interface
   and Alternate Setting (uint8_t Interface, uint8_t AlternateSetting) are supported by
   the application.
   This function is writing by user. It should return "SUCCESS" if the Interface
   and Alternate Setting are supported by the application or "UNSUPPORT" if they
   are not supported. */

  RESULT    (*Class_Get_Interface_Setting)(uint8_t Interface, uint8_t AlternateSetting);

  uint8_t* (*GetDeviceDescriptor)(uint16_t Length);
  uint8_t* (*GetConfigDescriptor)(uint16_t Length);
  uint8_t* (*GetStringDescriptor)(uint16_t Length);

  uint8_t* RxEP_buffer;
  uint8_t MaxPacketSize;

}DEVICE_PROP;
```

4. User standard request structure

The User Standard Request Structure is the interface between the user code and the management of the standard request. The structure has the type: **USER_STANDARD_REQUESTS:**

```
typedef struct _USER_STANDARD_REQUESTS
{
   void (*User_GetConfiguration)(void);          /* Get Configuration */
   void (*User_SetConfiguration)(void);          /* Set Configuration */
   void (*User_GetInterface)(void);              /* Get Interface */
   void (*User_SetInterface)(void);              /* Set Interface */
   void (*User_GetStatus)(void);                 /* Get Status */
   void (*User_ClearFeature)(void);              /* Clear Feature */
   void (*User_SetEndPointFeature)(void);        /* Set Endpoint Feature */
   void (*User_SetDeviceFeature)(void);          /* Set Device Feature */
   void (*User_SetDeviceAddress)(void);          /* Set Device Address */
}
USER_STANDARD_REQUESTS;
```

If the user wants to implement specific code after receiving a standard USB Device request he has to use the corresponding functions in this structure.

An application developer must implement three structures having the **DEVICE_PROP**, **Device_Table** and **USER_STANDARD_REQUEST** types in order to manage class requests and application specific controls. The different fields of these structures are described in *Section 1.4.4: usb_type.h / usb_def.h*.

2.2.3 usb_sil(.h, .c)

The **usb_sil** module implements an additional abstraction layer for USB-FS_Device and OTG-FS_Device peripherals. It offers simple functions for accessing the Endpoints for Read and Write operations.

Endpoint simplified write function

The write operation to an endpoint can be performed through the following function:

```
void USB_SIL_Write(uint32_t EPNum, uint8_t* pBufferPointer, uint32_t wBufferSize);
```

The parameters of this function are:

- EPNum: Number of the IN endpoint related to the write operation
- pBufferPointer: Pointer to the user buffer to be written to the IN endpoint.
- wBufferSize: Number of data bytes to be written to the IN endpoint.

Depending on the peripheral interface, this function gets the address of the endpoint buffer and performs the packet write operation.

Endpoint simplified read function

The read operation from an endpoint can be performed through the following function:

```
uint32_t USB_SIL_Read(uint32_t EPNum, uint8_t* pBufferPointer);
```

The parameters of this function are:

- EPNum: Number of the OUT endpoint related to the read operation

• pBufferPointer: Pointer to the user buffer to be filled with the data read form the OUT endpoint.
Depending on the peripheral interface, this function performs two successive operations:
• Gets the number of data received from the host on the related OUT endpoint
• Copies the received data from the USB dedicated memory to the pBufferPointer address.
Then the function returns the number of received data bytes to the user application.

uint32_t USB_SIL_Init(void); 在usb_prop .c中的xxx-inti函数中调用。
USB_SIL_Write和USB_SIL_Read用户可以用它们来读取端点中的数据。

2.2.4 usb_type.h / usb_def.h
These files provides the main types and USB definitions used in the library.

2.3 Application Interface layer-High Layer
Table 3. Application interface modules

File	**Description**
usb_conf.h	USB-FS_Device configuration file
usb_desc (.h, .c)	USB-FS_Device descriptors
usb_prop (.h, .c)	USB-FS_Device application-specific properties
usb_endp.c	Correct transfer interrupt handler routines for non-control endpoints
usb_istr (.h,.c)	USB-FS_Device interrupt handler functions
usb_pwr (.h, .c)	USB-FS_Device power and connection management functions

2.3.1 usb_conf(.h)
The usb_conf.h is used to:
For USB-FS_Device peripheral
● Define the BTABLE and all endpoint addresses in the PMA.
● Define the interrupt mask according to the needed events.
For OTG-FS_Device peripheral
● Define the Endpoint number.
● Define the interrupt mask according to the needed events.

2.3.2 usb_prop (.h , .c)
The **usb_prop** module is used for implementing the **Device_Property**, **Device_Table** and **USER_STANDARD_REQUEST** structures used by the USB core. 它们的具体含义在use_core.h/.c中说明。

2.3.4 USB_endp (.c)
USB_endp module is used for:
• Handling the CTR “correct transfer” routines for endpoints other than endpoint 0 (EP0) for the USB-FS_Device peripheral.
• Handling the “transfer complete” interrupt routines for endpoints other than endpoint 0(EP0) for the OTG-FS_Device peripheral. It also allows handling the Rx FIFO level interrupts for isochronous endpoints.

For enabling the processing of these callback handlers a pre-processor switch named EPx_IN_Callback (for IN transfer) or EPx_OUT_Callback (for OUT transfer) or EPx_RX_ISOC_CALLBACK (for Isochronous Out transfer) must be defined in the ***USB_conf.h*** file.

2.3.5 usb_istr(.c)

USB_istr module provides a function named **USB_Istr()** which handles all USB interrupts.

For each USB interrupt source, a callback routine named XXX_Callback (for example, RESET_Callback) is provided in order to implement a user interrupt handler. To enable the processing of each callback routines, a preprocessor switch named XXX_Callback must be defined in the USB configuration file ***USB_conf.h***.

USB_Istr() 在 USB 的中断服务子程序 stm32f10x_it.c 中的 void USB_LP_CAN1_RX0_IRQHandler(void)中调用，这个是驱动源。

2.3.6 usb_pwr (.h , .c)

This module manages the power management of the USB device. It provides the functions shown in *Table 8*.

Table 8. Power management functions

Function name	**Description**
RESULT Power_on(void)	Handle switch-on conditions
RESULT Power_off(void)	Handle switch-off conditions
void Suspend(void)	Sets suspend mode operation conditions
Void Resume(RESUME_STATE eResumeSetVal)	Handle wakeup operations

uint32_t Power_on(void); 在usb_prop .c中的xxx-inti函数中调用。

Suspend和**Resume**一般在**void USB_Istr(void);**中处理，当然用户也可以自己根据情况调用。