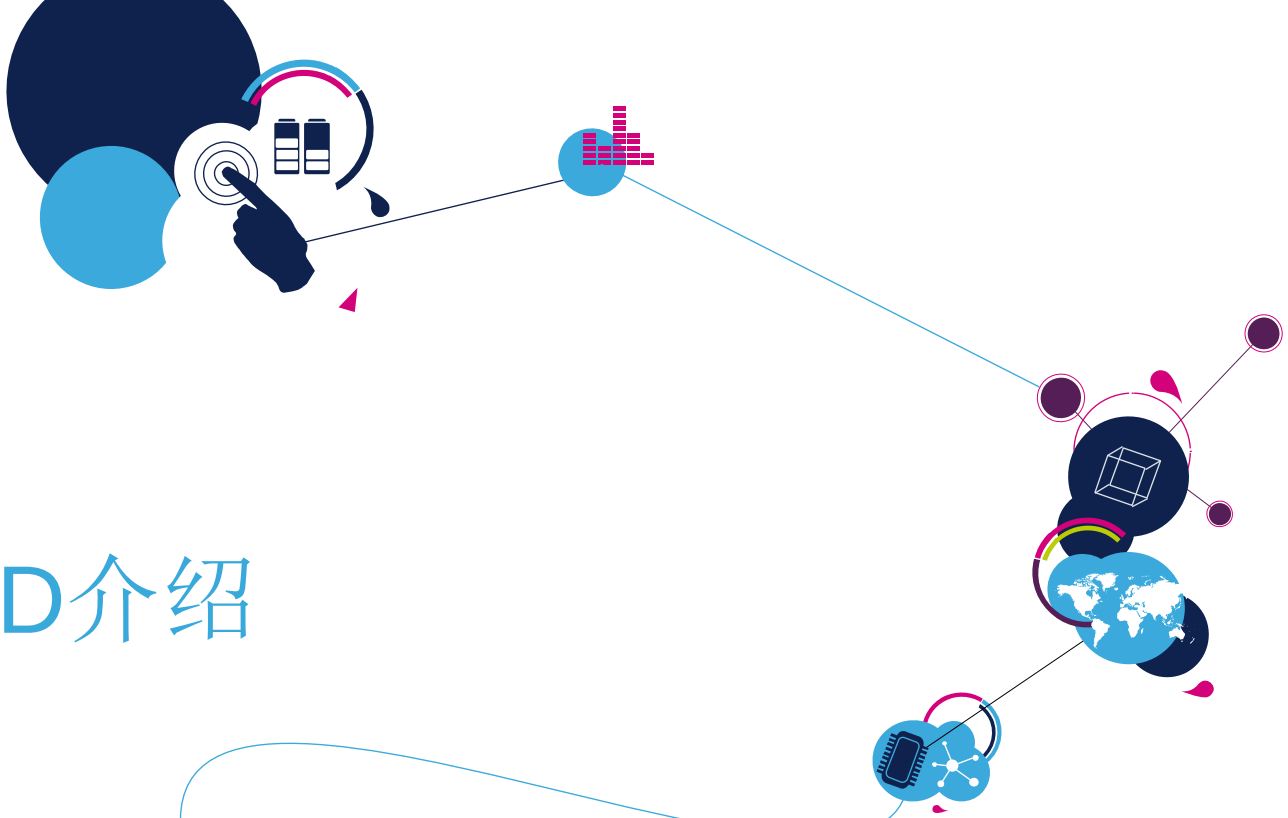


IP role

USB-USBD介绍

Lilian YAO

life.augmented

STM32 MCU有两种带USB功能的IP

- USB IP
  - 可作为全速USB设备
  - 存在于STM32F102、STM32F103

USB+ IP

可作为全速USB设备

存在于STM32F0x2

- FS OTG IP
  - 可作为全速和低速USB主机
  - 可作为全速USB设备
  - 存在于STM32F105、STM32F107、STM32F2、STM32F4
- HS OTG IP
  - 可作为高速、全速和低速USB主机
  - 可作为高速和全速USB设备
  - 存在于STM32F2、STSM32F4

本PPT讲解USB IP

	USB+	USB	OTG	
	FS	FS	FS	HS
STM32F0x2	Y			
STM32F102/103/F3		Y		
STM32F105/107			Y	
STM32F2/F4			Y	Y

USB(+) IP对应的三种库

USB(+) IP		
	Legacy library	**Cube library**
F0x2	**STSW-STM32092**	STM32CubeF0
F103	**STSW-STM32121**	
F3		STM32CubeF3
L1		STM32CubeL1
L0		STM32CubeL0

STSW-STM32092 STM32F0x2xx USB FS device library (UM1717)

Active

STSW-STM32121

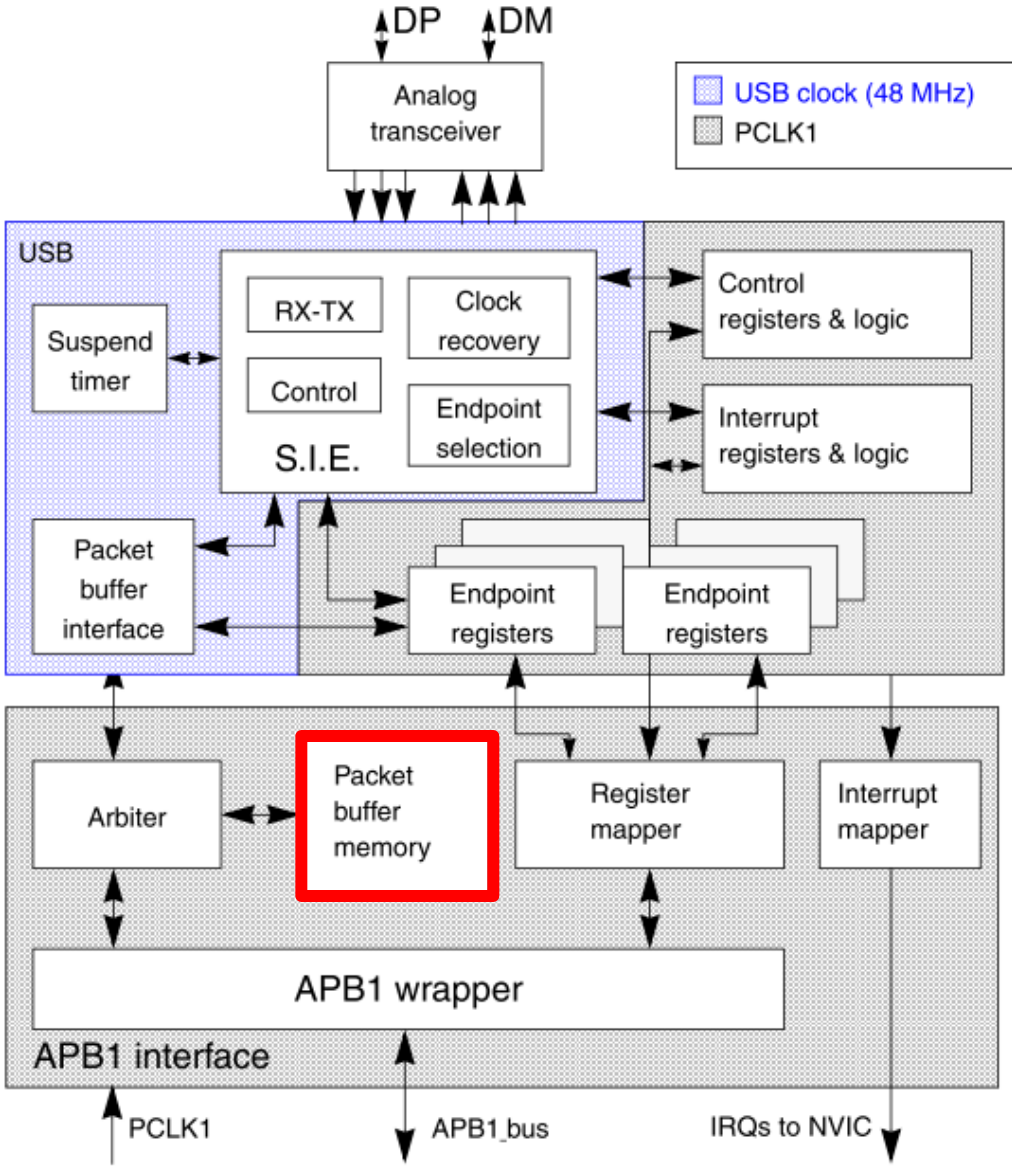
STM32F10x, STM32L1xx and STM32F3xx USB full speed device library (UM0424)

Active

模块特点和注意事项

- 符合USB2.0中的全速规范
- 可用资源：8个双向端点
- 支持四种传输类型
  - 对于bulk和同步传输，还支持double buffer模式；使得一个buffer用于USB硬件和PC交换数据的同时，另外一个buffer可被MCU使用
- 支持USB设备的挂起和唤醒操作（写控制寄存器）
  - 从而停止设备时钟，以进入低功耗模式
- **注意事项**：
  - F102/103中的USB和CAN共享512字节的专用SRAM来进行数据收发操作，因此两个IP不能同时使用
  - F105/107/F2/F4上OTG IP不受此限制

USB模块的功能框图



PC和MCU的数据交换是通过专门的**packet buffer memory**实现的

SIE使用固定的48MHz精确时钟

为使USB正常工作，APB1时钟不能低于8MHz！

- SIE
  - 硬件识别同步信号、进行比特填充、产生以及校验CRC、产生以及验证PID、握手
  - 根据外设事件来产生SOF、复位信号等。。。
- 定时器
  - 产生和帧信号同步的时钟脉冲
  - 检测挂起信号（即3ms内USB总线上没有信号）
- Packet Buffer接口
  - 通过一组收、发buffer来管理512字节的local memory
  - 硬件根据来自SIE的请求来选择合适的buffer
- 寄存器
  - EP相关的寄存器：该EP的传输类型、地址、当前状态
    - 【USB_EPnR n=0~7】
  - 控制寄存器：控制USB模块事件（比如唤醒和休眠）和反应USB模块当前状态
    - 【USB_CNTR.功耗方面的控制】、【USB_DAADR.模块使能/设备地址】、【USB_BTABLE】、【USB_FNR.】、【USB_ADDR/CNT_TX/RX】
  - 中断寄存器：中断掩码，记录事件
    - 【USB_CNTR.中断掩码控制】、【USB_ISTR】

- USB模块和APB1总线的接口部分包含以下子模块

- Packet buffer memory
  - 实际包含packet buffer的地方
  - 最大容量=512字节=256个半字*16比特

- 仲裁
  - 接收来自APB1总线和来自USB接口的memory request，前者优先级更高

- 寄存器 mapper
  - 把USB外设的字节宽度和位宽度的寄存器，组合成能被APB1访问的16位宽度字

- APB1 wrapper
  - 为packet buffer memory和寄存器提供了APB1接口
  - 把所有USB外设映射到APB1的地址空间

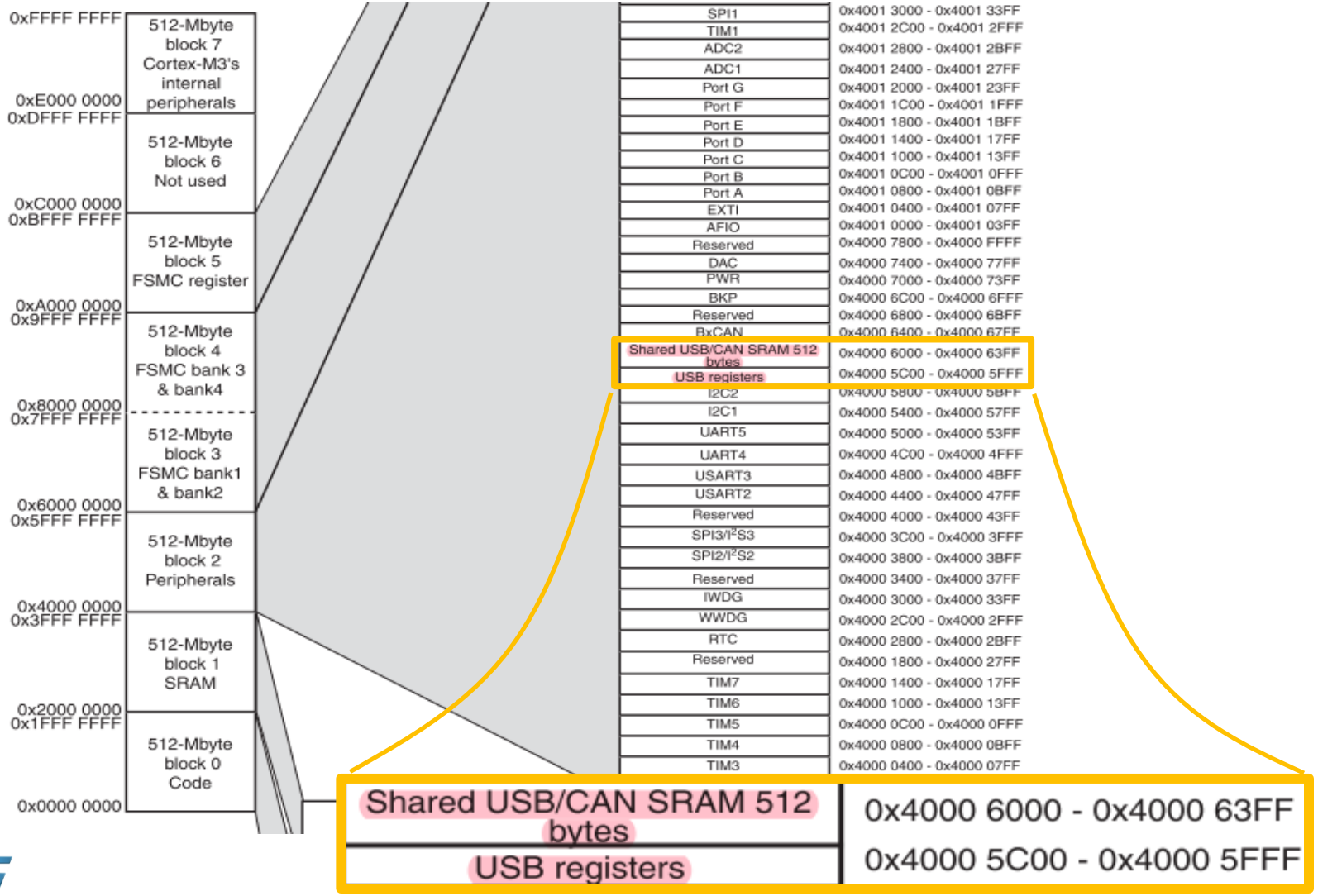
- 中断 mapper
  - NVIC.向量20：所有USB事件（正确传输结束、USB复位等）都可触发
  - NVIC.向量19：只能被同步和double buffer bulk传输的正确传输结束事件触发
  - NVIC.向量42：只能被(唤醒USB挂起模式)的事件触发 @EXTI_18

1. 复位

- 系统复位和上电复位
  - 提供USB外设模块时钟取消施加在该模块上的复位信号
    - RCC寄存器操作：使得软件能够访问该模块的寄存器组
  - 打开和USB收发器相连的模拟部分（打开内部参考电压给端口收发器供电）
    - 复位PDWN@CNTR；
    - 等待内部参考电压稳定时间$t_{STARTUP}$
    - 移除施加在该USB模块上的复位条件：软件清零FRES@CNTR
    - 清除ISTR寄存器，以移除spurious pending interrupt，然后再使能其他单元
- USB复位信号及其对应中断
  - 该事件发生时，USB外设的状态和系统复位后状态一样
  - 软件应该在10ms之内使能USB功能：EF@DADDR
  - 初始化EP0R寄存器和ep0对应的packet buffer
  - 如果置位了RESETM@CNTR，还会产生中断；直到RESET标志被清零之前，数据收发都被disable的

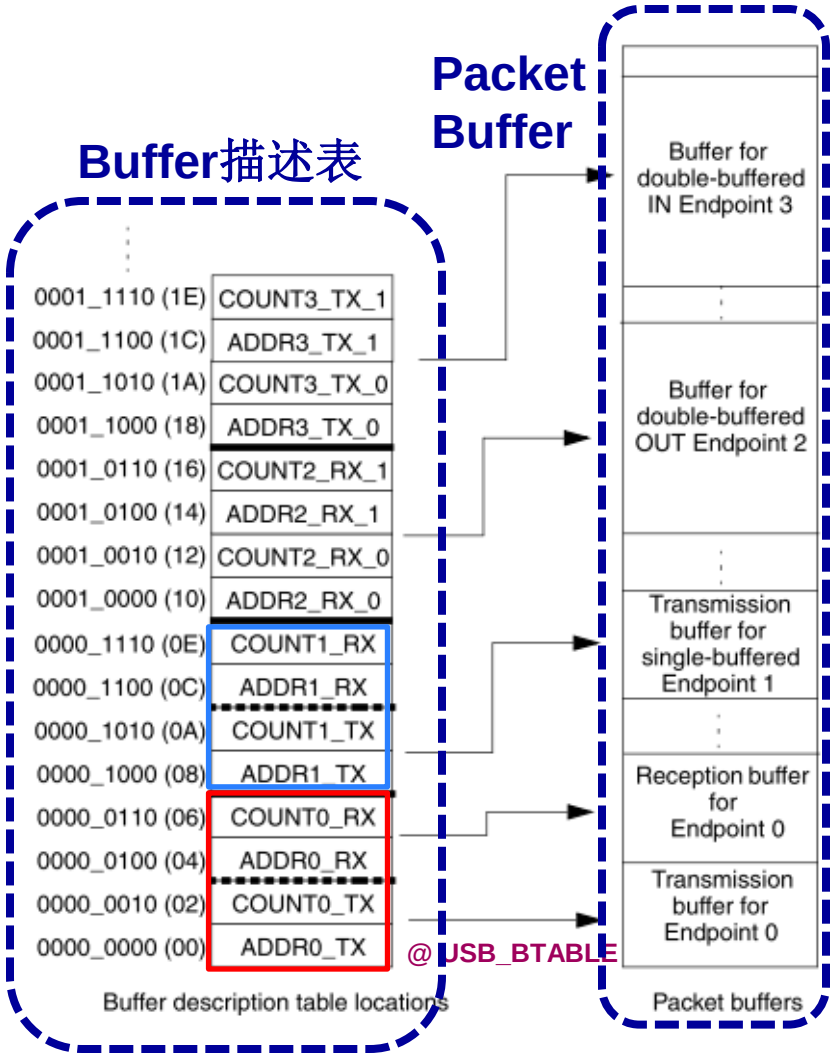
2. Packet buffer

- 硬件缓冲区 @ 0x4000 6000



Packet buffer的使用

- 每个双向EP对应两个packet buffer，分别用于发送和接收
  - 软件通过packet buffer interface来访问它们
  - 这些packet buffer的位置和大小都可配置，由buffer描述表指定
  - Buffer描述表本身也在这块memory里，它自己的地址是由USB_BTABLE寄存器指定的
    - Table里每个entry由4个半字组成（分别表示双向EP的接收packet和发送packet的位置和大小）
    - 因此该table的位置本身必须以8字节对齐，即USB_BTABLE的低三位全部为0
  - USB外设硬件不会把本EP的数据溢出到与其相邻的其他packet
    - 如果收到的数据多于buffer的长度，则只把前length个数据放到该EP对应的Packet buffer中

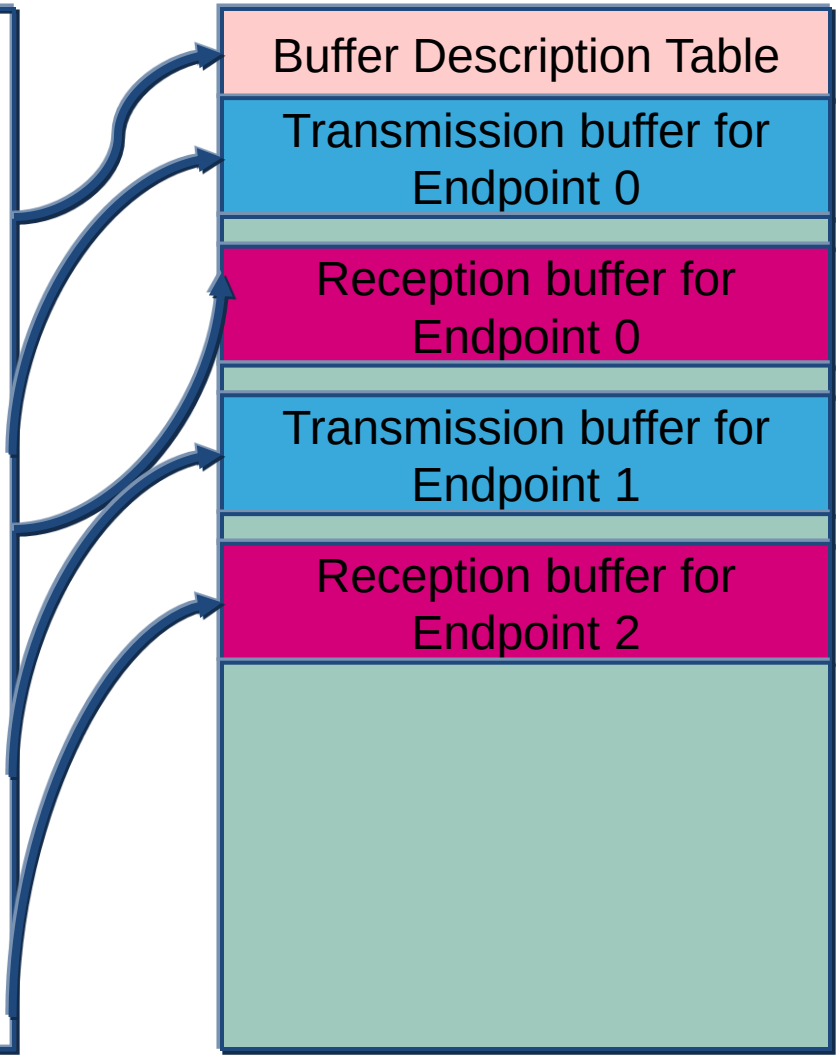


硬件缓冲区

Packet Buffer 的设置@Legacy library

@ <Usb_conf.h>

```
#define BTABLE_ADDRESS          (0x00)

/* EP0  */
/* rx/tx buffer base address */
#define ENDP0_TXADDR            (0x18)
#define ENDP0_RXADDR            (0x58)

/* EP1  */
/* tx buffer base address */
#define ENDP1_TXADDR            (0x98)

/* EP2  */
/* Rx buffer base address */
#define ENDP2_RXADDR            (0xD8)
```

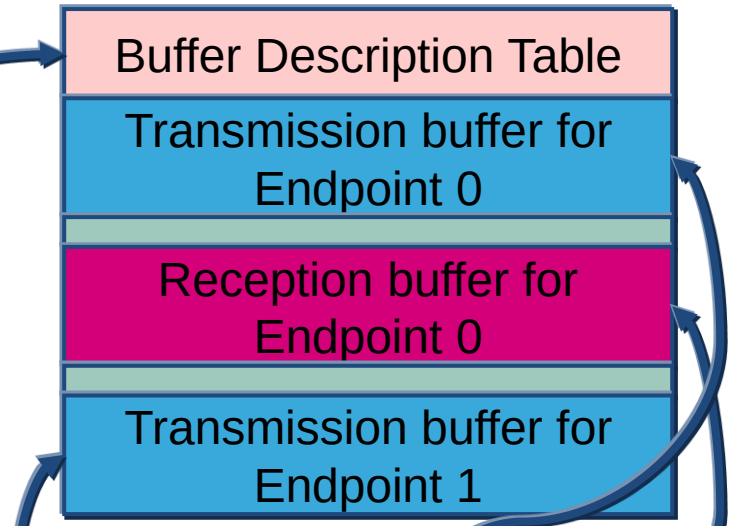


life.augmented

Packet Buffer 的设置@Cube library

@ <stm32f0xx_hal_pcd.c>

```
#define BTABLE_ADDRESS          (0x00)
```



@ <usbd_conf.c>

```
USBD_LL_Init ()
{    ……
HAL_PCDEx_PMAConfig(&hpcd , 0x00 , PCD_SNG_BUF, 0x18);
HAL_PCDEx_PMAConfig(&hpcd , 0x80 , PCD_SNG_BUF, 0x58);
HAL_PCDEx_PMAConfig(&hpcd , 0x81 , PCD_SNG_BUF, 0x100);
}
```

- 硬件**发送**缓冲区
  - 在初始化时设定各个EP硬件发送缓冲区的起始地址 @ADDRn_TX@硬件缓冲描述表
  - 在准备好要发送的数据后，设置发送长度@COUNTn_TX@硬件缓冲描述表

- 硬件**接收**缓冲区
  - 在初始化时设定各个EP硬件接收缓冲区的起始地址 @ADDRn_RX@硬件缓冲描述表
  - 在初始化时设定各个EP硬件接收缓冲区的长度 @COUNTn_RX的高位@硬件缓冲描述表，以允许接收缓冲区的溢出检测；一般都是接收EP的最大包长
  - 在收到数据并产生ISR中，从硬件接收缓存读取数据之前先要看收到了多少数据（实际收到的数据不一定填满接收缓存的）

MASS_Reset() @ <usb_prop.c> **Legacy Library**

```
SetEPTxAddr(ENDP0, ENDP0_TXADDR);
SetEPRxAddr(ENDP0, ENDP0_RXADDR);
SetEPRxCount(ENDP0, Device_Property.MPS);

SetEPTxAddr(ENDP1, ENDP1_TXADDR);

SetEPRxAddr(ENDP2, ENDP2_RXADDR)
SetEPRxCount(ENDP2, Device_Property.M
```

USB_SIL_Write() @ <usb_sil.c> **Legacy Library**

```
UserToPMABufferCopy(pBuf, GetEPTxAddr(bEpAddr & 0x7F), size)
SetEPTxCount((bEpAddr & 0x7F), size);
```

USB_SIL_Read () @ <usb_sil.c>

```
Length = GetEPRxCount(bEpAddr & 0x7F);
PMAToUserBufferCopy(pBuf, GetEPRxAddr(bEpAddr & 0x7F), Length);
```

3.端点的初始化

- 端点（EP）的初始化
  - 配置EP_TYPE、EP_KIND@USB_EPnR
  - 配置ADDRn_TX和ADDRn_RX寄存器
  - 发送功能通过STAT_TX@USB_EPnR使能EP的发送功能，配置COUNTn_TX
  - 接收：通过STAT_RX@USB_EPnR使能EP的接收功能，配置BL_SIZE、NUM_BLOCK

Transmission byte count n (USB_COUNTn_TX)

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Reserved						COUNTn_TX[9:0]									
						rw	rw	rw	rw	rw	rw	rw	rw	rw	rw

Bits 15:10 These bits are not used since packet size is limited by USB specifications to 1023 bytes. Their value is not considered by the USB peripheral.

Bits 9:0 **COUNTn_TX[9:0]:** Transmission byte count

These bits contain the number of bytes to be transmitted by the endpoint associated with the USB_EPnR register at the next IN token addressed to it.

Reception byte count n (USB_COUNTn_RX)

【可配置】分配了多少用于接收

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
BLSIZE	NUM_BLOCK[4:0]					COUNTn_RX[9:0]									
rw	rw	rw	rw	rw	rw	r	r	r	r	r	r	r	r	r	r

【只读】实际收到了多少，可检测接收溢出

This table location is used to store two different values, both required during packet reception. The most significant bits contains the definition of allocated buffer size, to allow buffer overflow detection, while the least significant part of this location is written back by the USB peripheral at the end of reception to give the actual number of received bytes. Due to

```
@ <MASS_Reset>

  SetBTABLE(0x00);

  SetEPRxAddr(ENDP0,   0x18);
  SetEPRxCount(ENDP0, 0x40);

  SetEPTxAddr(ENDP0,   0x58);

  SetEPTxAddr(ENDP1,   0x98);

  SetEPRxAddr(ENDP2,  0xD8);
  SetEPRxCount(ENDP2, 0x40);
```

*可见阴影部分就浪费了，使得该buffer description Table的大小偏大了（在512字节的硬件buf中占用了24字节）
要是非0EP的接收在EP1上实现，在512字节的硬件Buf中就只占用16字节

可见对IN EP的tx-cnt，不是在初始化时设置；
而是在要真正发送数据时才设置，每次都可能不同~

低字节 → 高字节

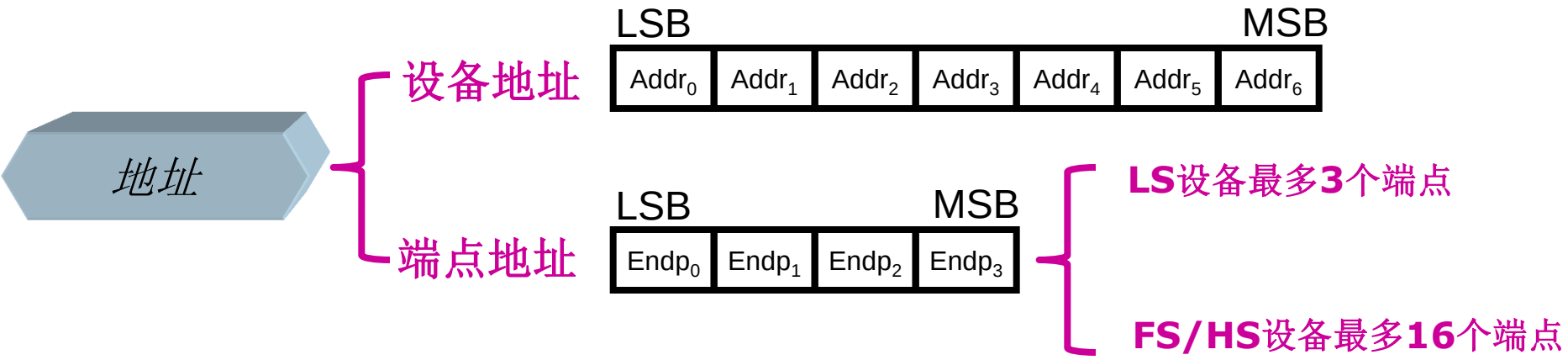
地址	TX-ADDR	TX-CNT	RX-ADDR	RX-CNT	说明
0x40006000	00000058	0000XXXX	00000018	00008400	EP0
0x40006010	00000098	0000YYYY	0000….	0000….	EP1
0x40006020	0000….	0000….	000000D8	00008400	EP2

XXXX：DataStageOut() → SetEPTxCount(ENDP0, 0);
DataStageIn() → SetEPTxCount(ENDP0, Length);

YYYY：Read_Memory() → SetEPTxCount(ENDP1, BULK_MAX_PACKET_SIZE);

端点资源

- 有8个双向端点
  - 双向EP0是必备的
  - 其他EPi，应用可同时使用收、发两个方向或只使用一个方向
- 名字叫做EP0~EP7
  - 对应的状态和控制寄存器是USB_EPnR (n=0~7)
    - 每个端点的地址是软件可配置的，4位地址（USB规范指定）：EA[3:0]@USB_EPnR
    - 并非EP1的地址就一定要是1，可以是10或15，但是两个方向的EP要是同一个地址
  - 对应的收、发硬件缓冲和长度寄存器是USB_ADDR/COUNTn_Tx/Rx



- 物理上16个单向端点
  - 16个房间，其中8个作接收、8个作发送
- 每两个端点共享同一**端点地址**
  - 每2个房间共享一个门牌号
  - 门牌号在EA[3:0]@USB_EPnR (n=0~7)中设置
  - USB IP最多向主机提供8种门牌号供主机寻址
  - 作为端点地址的门牌号，遵循USB规范，由4bit组成
- 对物理上这16个单向端点
  - 由8个寄存器USB_EPnR (n=0~7)控制
  - 在512字节packet buffer中的接收/发送硬件缓冲由USB_ADDRn_TX/RX寄存器指向

USB endpoint n register (USB_EPnR), n=[0..7]

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

USBD_MSC demo
U盘demo，使用1个双向EP0；
1个IN方向的EP1，一个OUT方向的EP2

```
@ <usb_desc.c>
0x81,   /*Endpoint address (IN, address 1) */
0x02,   /*Endpoint address (OUT, address 2) */

@ <usb_conf.h>
#define EP_NUM                    (3)

@ <usb_conf.h>
#define BTABLE_ADDRESS      (0x00)
#define ENDP0_RXADDR        (0x18)
#define ENDP0_TXADDR        (0x58)
#define ENDP1_TXADDR        (0x98)
#define ENDP2_RXADDR        (0xD8)

@ <usb_prop.c>
MASS_Reset --> SetDeviceAddress(0) @ <usb_core.c>

 uint32_t nEP = Device_Table.Total_Endpoint;

  /* set address in every used endpoint */
  for (i = 0; i < nEP; i++)
  {
    _SetEPAddress((uint8_t)i, (uint8_t)i);
  } /* for */
  _SetDADDR(Val | DADDR_EF);
```

可见库里的代码这里固定写死了，其rule就是
>> 依次往前排
>> 每个双向端点仅使用其一个方向

如果上位机固定是对0x0f和0x0e两个端点地址访问，
需要修改下位机程序如下……

```
@ <usb_desc.c>
0x8E,   /*Endpoint address (IN, address 0x0e) */
0x0F,   /*Endpoint address (OUT, address 0x0f) */

@ <usb_conf.h>
#define EP_NUM                    (3)

@ <usb_conf.h>
#define BTABLE_ADDRESS      (0x00)
#define ENDP0_RXADDR        (0x18)
#define ENDP0_TXADDR        (0x58)
#define ENDP1_TXADDR        (0x98)
#define ENDP2_RXADDR        (0xD8)

@ <usb_prop.c>
MASS_Reset --> SetDeviceAddress(0) @ <usb_core.c>

 uint32_t nEP = Device_Table.Total_Endpoint;

  /* set address in every used endpoint */
 _SetEPAddress((uint8_t)0, (uint8_t)0);
  _SetEPAddress((uint8_t)1, (uint8_t)0x0e);
  _SetEPAddress((uint8_t)2, (uint8_t)0x0f);

MASS_Reset
    /* Initialize Endpoint 0 */
    /* Initialize Endpoint 1 */  不变
    /* Initialize Endpoint 2 */  不变
```

USBD_MSC demo
U盘demo，使用1个双向EP0；
1个IN方向的EP1，一个OUT方向的EP2

@ <usb_desc.c>
0x81, /*Endpoint address (IN, address 1) */
0x02, /*Endpoint address (OUT, address 2) */

@ <usb_conf.h>
#define EP_NUM   (3)

@ <usb_conf.h>
#define BTABLE_ADDRESS   (0x00)
#define ENDP0_RXADDR   (0x18)
#define ENDP0_TXADDR   (0x58)
#define ENDP1_TXADDR   (0x98)
#define ENDP2_RXADDR   (0xD8)

@ <usb_prop.c>
MASS_Reset --> SetDeviceAddress(0) @ <usb_core.c>

```
uint32_t nEP = Device_Table.Total_Endpoint;

 /* set address in every used endpoint */
 for (i = 0; i < nEP; i++)
 {
   _SetEPAddress((uint8_t)i, (uint8_t)i);
 } /* for */
 _SetDADDR(Val | DADDR_EF);
```

可见库里的代码这里固定写死了，其rule就是
>> 依次往前排
>> 每个双向端点仅使用其一个方向

如果上位机固定是对0x01的两个端点双向地址访问，
需要修改下位机程序如下……

@ <usb_desc.c>
0x81, /*Endpoint address (IN, address 0x0e) */
0x01, /*Endpoint address (OUT, address 0x0f) */

@ <usb_conf.h>
#define EP_NUM   (3)

@ <usb_conf.h>
#define BTABLE_ADDRESS   (0x00)
#define ENDP0_RXADDR   (0x18)
#define ENDP0_TXADDR   (0x58)
#define ENDP1_TXADDR   (0x98)   ENP5_TXADDR
#define ENDP1_RXADDR   (0xD8)   ENP5_RXADDR

@ <usb_prop.c>
MASS_Reset --> SetDeviceAddress(0) @ <usb_core.c>

```
uint32_t nEP = Device_Table.Total_Endpoint;

 /* set address in every used endpoint */
_SetEPAddress((uint8_t)0, (uint8_t)0);
 _SetEPAddress((uint8_t)1, (uint8_t)0x01);
               (uint8_t)5,
```

MASS_Reset
  /* Initialize Endpoint 0 */ 不变
  /* Initialize Endpoint 1 */ 增加对EP1另外一个方向端点的初始化
  /* Initialize Endpoint 2 */ 去掉对EP2的配置

@ <usb_endp.c>
  EP2_OUT_Callback()改成EP1_OUT_Callback……

各device demo对EP的使用

	EP_NUM	EP0		EP1		EP2		EP3	
		Rx	Tx	Rx	Tx	Rx	Tx	Rx	Tx
Audio	2			双缓冲					
Composite	3								
Custom HID	2								
DFU	1								
Joystick Mouse	2								
MSC	3								
VCP	4								

1. 应用中使用到的EP尽量内部编号考前（0~7），这样可以使得【硬件缓冲描述表】尽可能小

2. 库函数默认把EPx的地址设置成x SetDeviceAddress(Val) @ <usb_core.c>

```
uint32_t nEP = Device_Table.Total_Endpoint;
 for (i = 0; i < nEP; i++)
  {
    _SetEPAddress((uint8_t)i, (uint8_t)i);
  }
_SetDADDR(Val | DADDR_EF);
```

设置设备地址的同时使能USB模块

实际问题解决

- http://bbs.21ic.com/icview-180644-1-2.html
- 基于Joystick demo的改动，引起hardfault
  - 把EP1的Tx移到EP4上

	EP_NUM	EP0		EP1		EP2		EP3		EP4	
		Rx	Tx	Rx	Tx	Rx	Tx	Rx	Tx	Rx	Tx
Joystick Mouse	2				●						●

```
<usb_conf.h>

#define EP_NUM     (2)
#define BTABLE_ADDRESS      (0x00)

/* EP0  */
/* rx/tx buffer base address */
#define ENDP0_RXADDR        (0x18)
#define ENDP0_TXADDR        (0x58)

/* EP1  */
/* tx buffer base address */
#define ENDP1_TXADDR        (0x100)
```

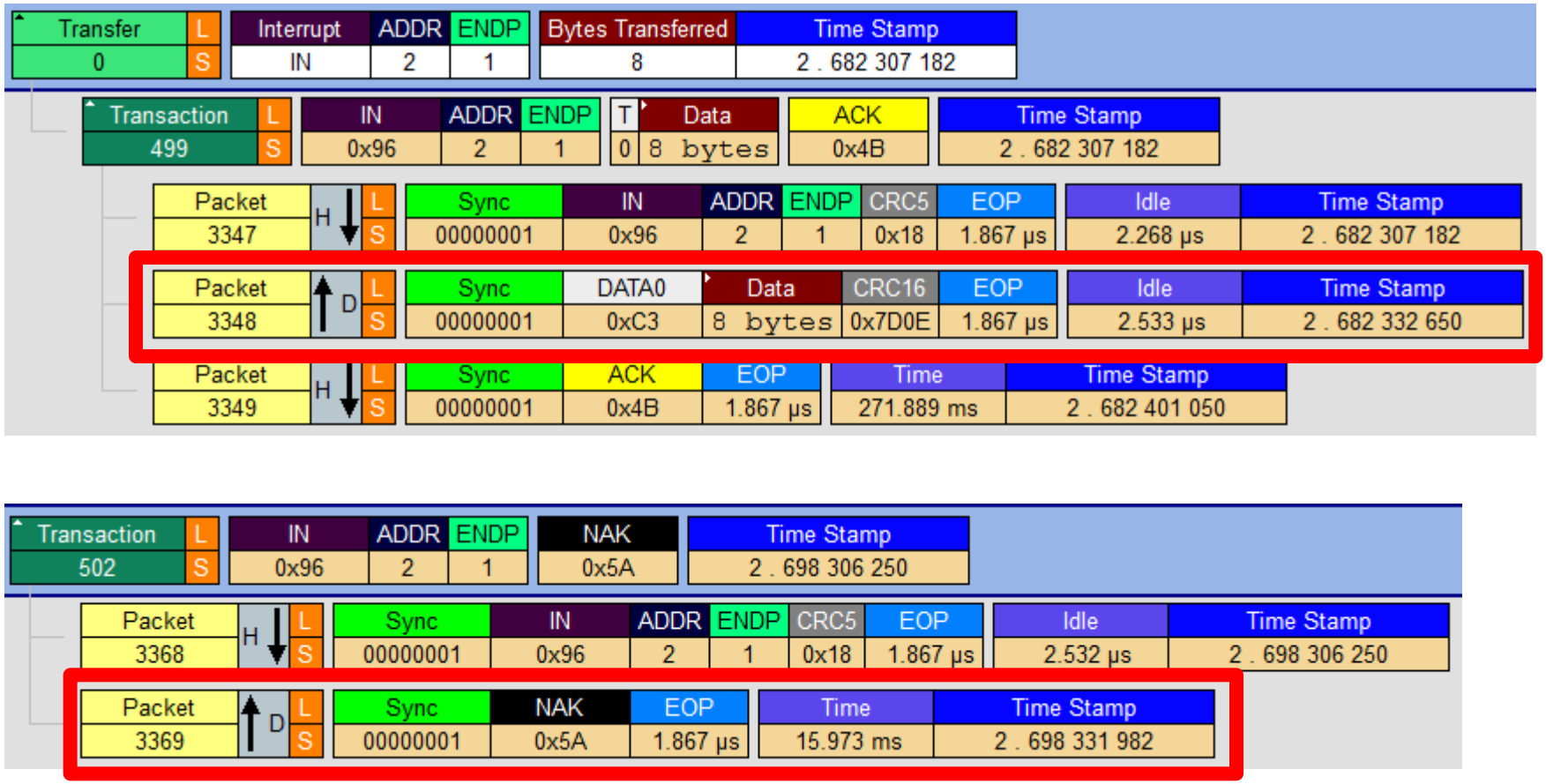
如何改动？

```
<usb_conf.h>

#define EP_NUM                    (? )
#define BTABLE_ADDRESS            (? )

/* EP0  */
/* rx/tx buffer base address */
#define ENDP0_RXADDR              (? )
#define ENDP0_TXADDR              (? )

/* EP1  */
/* tx buffer base address */
#define ENDP1_TXADDR              (? )
```

4. IN packet的处理

- USB外设收到主机发送的PID为IN的令牌包后
  - 如果这个packet中的设备地址信息和端点号信息有效，**并且端点为Valid状态**
    - USB硬件发送DATA0或DATA1的PID（根据DTOG_TX@USB_EPnR）
    - USB硬件发送待发送的数据
    - USB硬件发送计算好的CRC
  - 如果该端点状态不是valid
    - USB硬件不发送data packet，而是根据STAT_TX@USB_EPnR发送NAK或STALL的握手包
- USB外设收到主机返回的应答（ PID为ACK的握手包）后
  - 硬件
    - toggle DTOG_TX@USB_EPnR
    - 硬件把该端点设置为invalid状态（STAT_TX=NAK）
    - 硬件置位CTR_TX，**产生中断**
  - 软件
    - 通过检查EP_ID和DIR@USB_ISTR来识别是哪个端点上的通信
    - 响应**CTR_TX中断**：
      - 标志清零；
      - 软件准备下次要发送的数据；
      - 更新COUNTn_TX如有必要
      - 软件重新设置STAT_TX=VALID来重新把该EP设置到发送valid状态

IN transfer实例



Transfer	Interrupt	ADDR	ENDP	Bytes Transferred	Time Stamp
0	IN	2	1	8	2 . 682 307 182

Transaction	IN	ADDR	ENDP	T	Data	ACK	Time Stamp
499	0x96	2	1	0	8 bytes	0x4B	2 . 682 307 182

Packet	Sync	IN	ADDR	ENDP	CRC5	EOP	Idle	Time Stamp
3347	00000001	0x96	2	1	0x18	1.867 µs	2.268 µs	2 . 682 307 182

Packet	Sync	DATA0	Data	CRC16	EOP	Idle	Time Stamp
3348	00000001	0xC3	8 bytes	0x7D0E	1.867 µs	2.533 µs	2 . 682 332 650

Packet	Sync	ACK	EOP	Time	Time Stamp
3349	00000001	0x4B	1.867 µs	271.889 ms	2 . 682 401 050

Transaction	IN	ADDR	ENDP	NAK	Time Stamp
502	0x96	2	1	0x5A	2 . 698 306 250

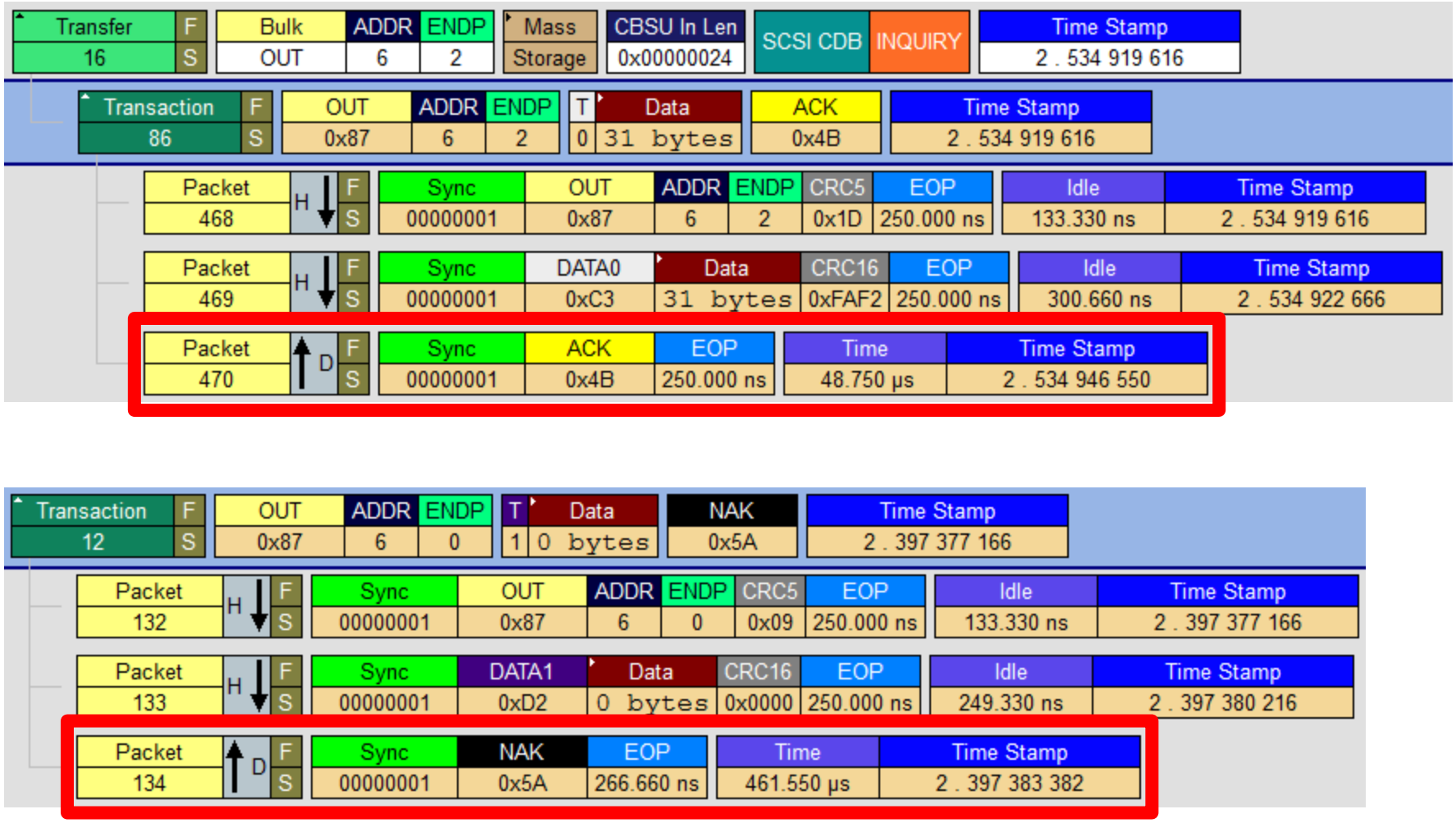
Packet	Sync	IN	ADDR	ENDP	CRC5	EOP	Idle	Time Stamp
3368	00000001	0x96	2	1	0x18	1.867 µs	2.532 µs	2 . 698 306 250

Packet	Sync	NAK	EOP	Time	Time Stamp
3369	00000001	0x5A	1.867 µs	15.973 ms	2 . 698 331 982

5. OUT/SETUP packet的处理

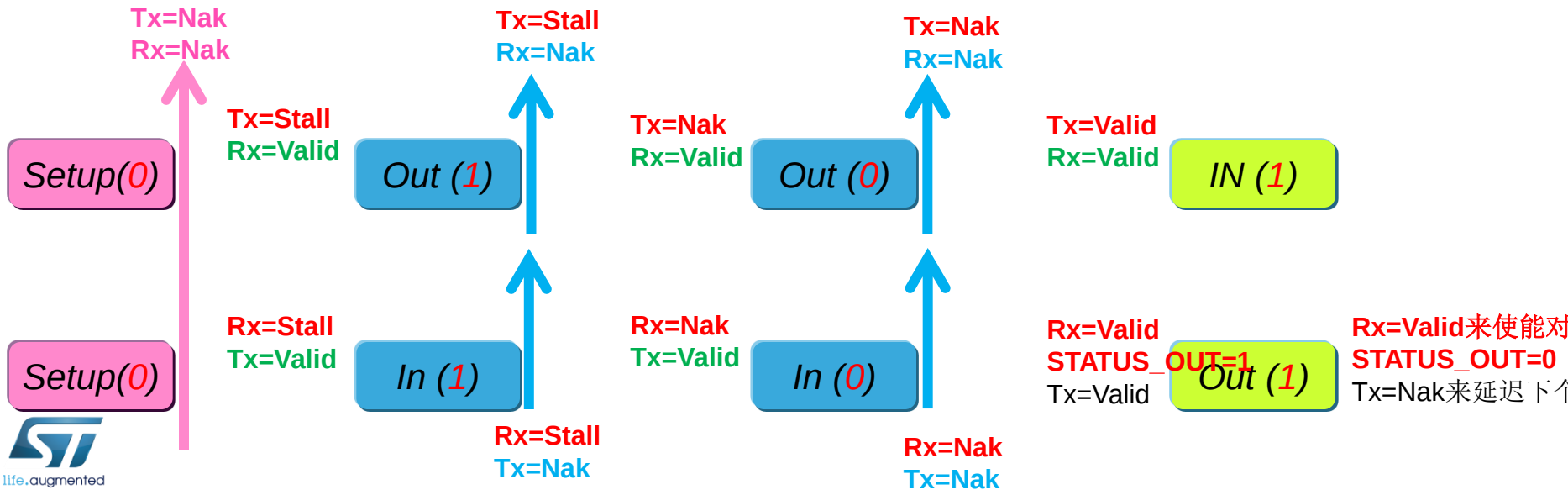
- USB外设收到主机发送的PID为OUT/SETUP的令牌包后
  - 如果这个packet中的设备地址信息和端点号信息有效，**并且端点为Valid状态**
    - USB硬件从硬件buf中把数据搬移到用户可访问的packet buffer中
    - USB硬件核对收到的CRC无误，就发出ACK握手包
    - 如果CRC有误，就不会发出ACK握手包，并置位ERR@USB_ISTR；一般USB模块会自动恢复来准备接收下一次传输
    - 如果收到的数据长度超过了分配的空间，硬件回复STALL，置位溢出错误，但不产生中断
  - 如果该端点状态不是valid
    - USB硬件根据STAT_RX@USB_EPnR发送NAK或STALL的握手包
- USB外设回复主机应答（ PID为ACK的握手包）后
  - 硬件
    - toggle DTOG_RX@USB_EPnR
    - 硬件把该端点设置为invalid状态（STAT_RX=NAK）
    - 硬件置位CTR_RX，**产生中断**
  - 软件
    - 通过检查EP_ID和DIR@USB_ISTR来识别是哪个端点上的通信
    - 响应**CTR_RX中断**：
      - 标志清零；
      - 软件对收下来的数据进行处理；
      - 软件重新设置STAT_RX=VALID来重新把该EP设置到接收valid状态

OUT transfer实例



Control transfer的处理

- 控制传输组成
  - 一个Setup transaction + 0个或多个同方向的data stage + 一个status stage
  - Setup transaction只能由控制端点完成
  - 初始化：DTOG_TX=1, DTOG_RX=0
  - 触发CTR_RX中断在此ISR中查看SETUP位来确定这是个SETUP/OUT transaction
    - 软件先把TX和RX的状态都设置成NAK，根据命令包中内容判断后续的数据stage的方向
  - 在除最后一个的每个数据stage时，对未用方向都设置成STALL
    - 以免主机过早结束传输，device可在status阶段返回STALL应答
  - 使能最后一次数据stage时，把未用方向设置成NAK
    - 以免主机在最后一次data stage后立马开始status stage时，device可在处理完所有数据前NAK掉



life.augmented

Control transfer实例

Transfer	F	Control	ADDR	ENDP	bRequest	wValue	wIndex	Descriptors	Time Stamp
2	S	GET	6	0	GET_DESCRIPTOR	DEVICE type	0x0000	DEVICE Descriptor	2 . 397 314 332

Transaction	F	SETUP	ADDR	ENDP	T	D	Tp	R	bRequest	wValue	wIndex	wLength	ACK	Time Stamp
8	S	0xB4	6	0	0	D->H	S	D	0x06	0x0100	0x0000	18	0x4B	2 . 397 314 332

Packet	H	F	Sync	SETUP	ADDR	ENDP	CRC5	EOP	Idle	Time Stamp
122	↓	S	00000001	0xB4	6	0	0x09	250.000 ns	133.330 ns	2 . 397 314 332

Packet	H	F	Sync	DATA0	Data	CRC16	EOP	Idle	Time Stamp
123	↓	S	00000001	0xC3	8 bytes	0x072F	250.000 ns	250.000 ns	2 . 397 317 382

Packet	D	F	Sync	ACK	EOP	Time	Time Stamp
124	↑	S	00000001	0x4B	250.000 ns	27.618 µs	2 . 397 325 882

Transaction	F	IN	ADDR	ENDP	T	Data	ACK	Time Stamp
11	S	0x96	6	0	1	18 bytes	0x4B	2 . 397 353 500

Packet	H	F	Sync	IN	ADDR	ENDP	CRC5	EOP	Idle	Time Stamp
129	↓	S	00000001	0x96	6	0	0x09	250.000 ns	299.330 ns	2 . 397 353 500

Packet	D	F	Sync	DATA1	Data	CRC16	EOP	Idle	Time Stamp
130	↑	S	00000001	0xD2	18 bytes	0x2A00	266.660 ns	200.660 ns	2 . 397 356 716

Packet	H	F	Sync	ACK	EOP	Time	Time Stamp
131	↓	S	00000001	0x4B	250.000 ns	16.516 µs	2 . 397 371 850

Transaction	F	OUT	ADDR	ENDP	T	Data	ACK	Time Stamp
13	S	0x87	6	0	1	0 bytes	0x4B	2 . 397 388 366

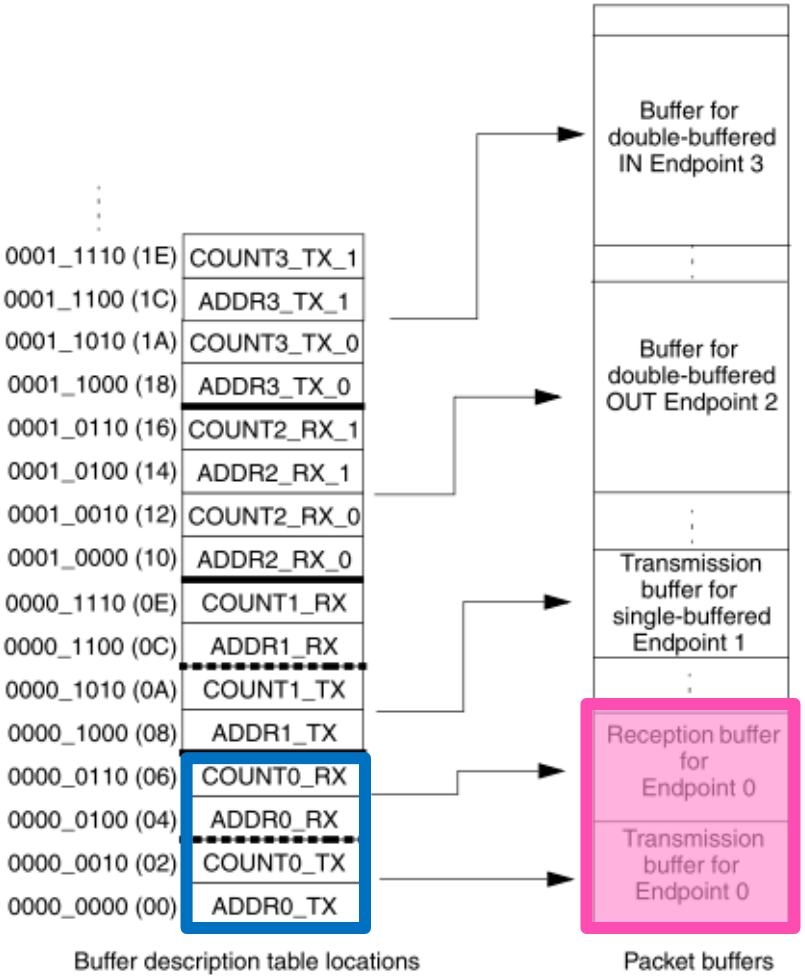
Packet	H	F	Sync	OUT	ADDR	ENDP	CRC5	EOP	Idle	Time Stamp
135	↓	S	00000001	0x87	6	0	0x09	250.000 ns	133.330 ns	2 . 397 388 366

Packet	H	F	Sync	DATA1	Data	CRC16	EOP	Idle	Time Stamp
136	↓	S	00000001	0xD2	0 bytes	0x0000	250.000 ns	299.330 ns	2 . 397 391 416

Packet	D	F	Sync	ACK	EOP	Time	Time Stamp
137	↑	S	00000001	0x4B	250.000 ns	450.300 µs	2 . 397 394 632

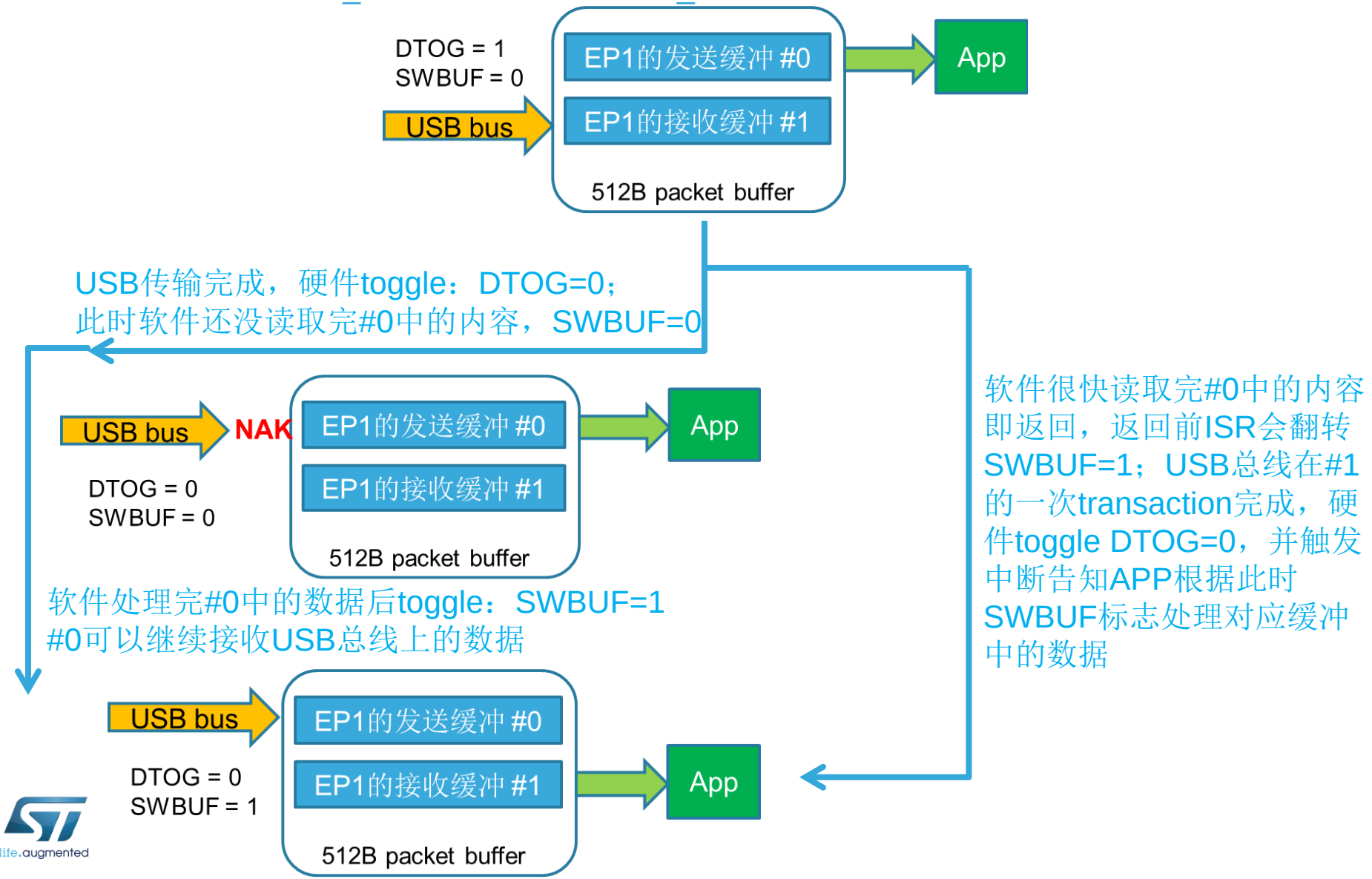
6. 双缓冲端点

- 使能控制
  - 只能针对某个BULK类型的EP进行使能
    - EP_TYPE = “BULK” @ USB_EPnR
    - EP_KIND = “DBL_BUF” @ USB_EPnR
- 当某个BULK类型的EP被使能“双缓冲”后
  - 这个EP只能作为单向端点使用
    - 作为接收方向：STAT_TX被设置成disable
    - 作为发送方向：STAT_RX被设置成disable
  - 它的“发送packet memory”和“接收packet memory”都被使用
    - 一个被USB硬件使用时
    - 另一个就可被软件使用
  - 由data toggle序列来选择当前哪个packet memory正在被USB硬件使用
    - 接收：由DTOG_RX，DTOG_TX作为SW_BUF标志当前哪个memory正在被application s/w使用
    - 发送：DTOG_TX



使用举例

- EP1用作OUT方向，使能“双缓冲”
  - DTOG = DTOG_RX、SWBUF = DTOG_TX



life.augmented

7. 同步传输

- 需要固定和精确数据率的传输被定义成“同步传输”
  - 在枚举时主机知道这是个同步类型的EP，就会在后期每个frame中给它分配所要求的带宽
  - 为节约带宽，同步传输没有重传机制，即没有握手阶段（数据包后没有握手包）；因此也就不需要数据toggle机制，它总是以DATA0作为PID发送数据

- 同步EP的定义
  - EP_TYPE = 10 (同步)；EP状态STAT_RX/TX只有Disable（00）和Valid（11）两种可能
  - 同步EP总是使用双缓冲结构，来缓解SW响应的压力

Endpoint Type	DTOG bit value	Packet buffer used by the USB peripheral	Packet buffer used by the application software
IN	0	ADDRn_TX_0 / COUNTn_TX_0 buffer description table locations.	ADDRn_TX_1 / COUNTn_TX_1 buffer description table locations.
	1	ADDRn_TX_1 / COUNTn_TX_1 buffer description table locations.	ADDRn_TX_0 / COUNTn_TX_0 buffer description table locations.
OUT	0	ADDRn_RX_0 / COUNTn_RX_0 buffer description table locations.	ADDRn_RX_1 / COUNTn_RX_1 buffer description table locations.
	1	ADDRn_RX_1 / COUNTn_RX_1 buffer description table locations.	ADDRn_RX_0 / COUNTn_RX_0 buffer description table locations.

DTOG指哪，USB硬件就用哪个buf收发数据；Sw去与之相反的buf处理数据；每次传输完成后由hw来toggle DTOG

8. 挂起和唤醒

- USB规范中定义的外设状态之一：【挂起】
  - 从USB总线获得电流不能超过2.5mA
  - 对总线供电设备的要求，自供电设备可不受此限制

- 设备检测到连续3个SOF包都没有收到，就认为主机要求它挂起
  - 硬件置位SUSP@USB_ISTR，产生中断；软件通常做以下操作
    - 置位FSUSP@USB_CNTR：激活USB外设的挂起模式（但是时钟和模拟收发模块的静态功耗还在）
    - 关闭或降低除USB之外的其他外设的静态功耗
    - 置位LP_MODE@USB_CNTR：关闭模拟收发模块的静态功耗；但收发模块仍能检测到resume序列，USB模块也要继续给上拉电阻供电
  - 在挂起状态时，设备也要能够检测Reset序列

```
if (wIstr & ISTR_SUSP & wInterrupt_Mask)
{
。。。。。。。。。。。
}
```

- 设备的唤醒
  - 可以由主机来发起Resume或Reset序列
  - 也可直接由设备自己来发起，但resume序列总是由主机来终结
    - SW置位RESUME@USB_CNTR，持续1ms~15ms后，SW再复位之
  - 一旦被唤醒后，设备的软件需要如下处理
    - 复位FSUSP@USB_CNTR
    - 还可通过RXDP和RXDM位来识别resume triggering event

F0x2/L0的USB功能扩展

- F0x2/L0基于F1的USB模块，扩展并改善了以下功能
- 性能提高

	USB IP	USB+ IP
USB专用数据包缓冲区	和CAN模块共享512B空间	数据包缓冲区扩展到1024B空间，后256B仍和CAN模块共享
APB时钟最低频率	8MHz	10MHz

  - 中断映射：不再有LP/HP之分，且没有专门的EXTI做USB模块唤醒，都在一个NVIC
- 新增特性
  - 支持USB 2.0链路层功耗管理
    - L1REQM @ USB_CNTR　　L1REQ @ USB_CNTR
  - 支持电池充电规范R1.2
  - D+线上集成可控上拉电阻（控制USB的连接和断开连接）

各系列下对比

- L1，没有CAN的冲突，有集成上拉(USB_PU@SYSCFG_PMC)
- F3，没有CAN的冲突，没有集成上拉
- L0，和F0一样，SRAM扩展到1024B，后面仍有CAN冲突，有集成上拉

	Packet buffer	D+上的上拉电阻	高级特性
F1	512B，和CAN共享	没有集成	
L1	512B，和CAN共享	有集成上拉，可软件控制 USB_PU@SYSCFG_PMC	
F3	512B，和CAN共享	没有集成	
F0/L0	1024B，后256B和CAN共享	有集成上拉，可软件控制 DPPU@USB_BCDR	支持LPM、BCD

CONFIDENTIALITY OBLIGATIONS:

THIS DOCUMENT CONTAINS SENSITIVE INFORMATION.

IT IS CLASSIFIED “MICROCONTROLLERS, MEMORIES & SECURE MCUs (MMS) RESTRICTED AND ITS DISTRIBUTION IS SUBMITTED TO ST/MMS AUTHORIZATION

AT ALL TIMES YOU SHOULD COMPLY WITH THE FOLLOWING SECURITY RULES:

- DO NOT COPY OR REPRODUCE ALL OR PART OF THIS DOCUMENT
- KEEP THIS DOCUMENT LOCKED AWAY
- FURTHER COPIES CAN BE PROVIDED ON A “NEED TO KNOW BASIS”, PLEASE CONTACT YOUR LOCAL ST SALES OFFICE OR DOCUMENT WRITTER.