IP role class

USB-USBD-MSC/HID

Lilian YAO

life.augmented

STM32 MCU有两种带USB功能的IP

- USB IP
  - 可作为全速或低速的USB**设备**
  - 存在于STM32F102、STM32F103
- 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

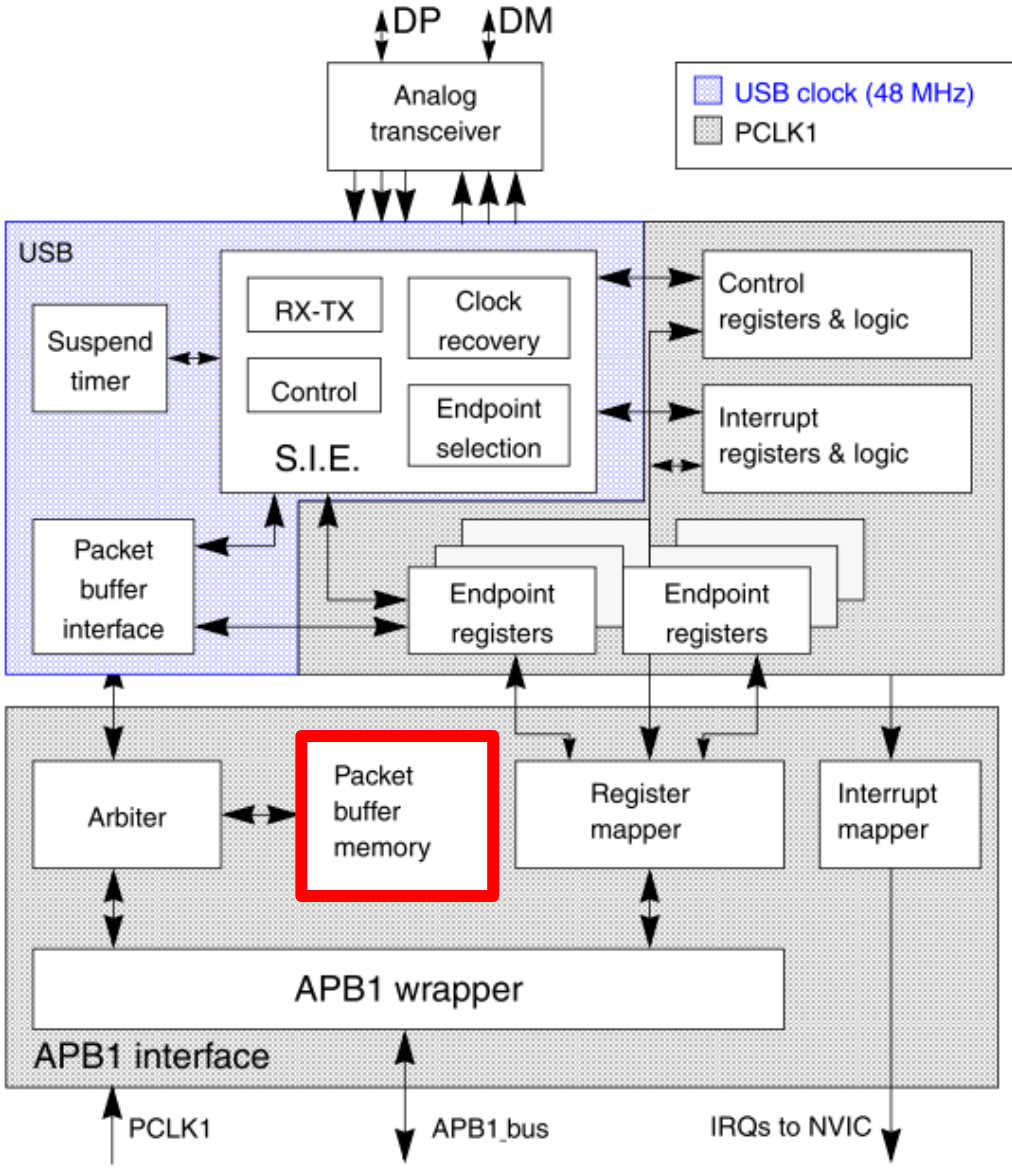
本PPT讲解USB IP

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

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
- 可产生SOF脉冲
- **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。。。
- 时钟
  - 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

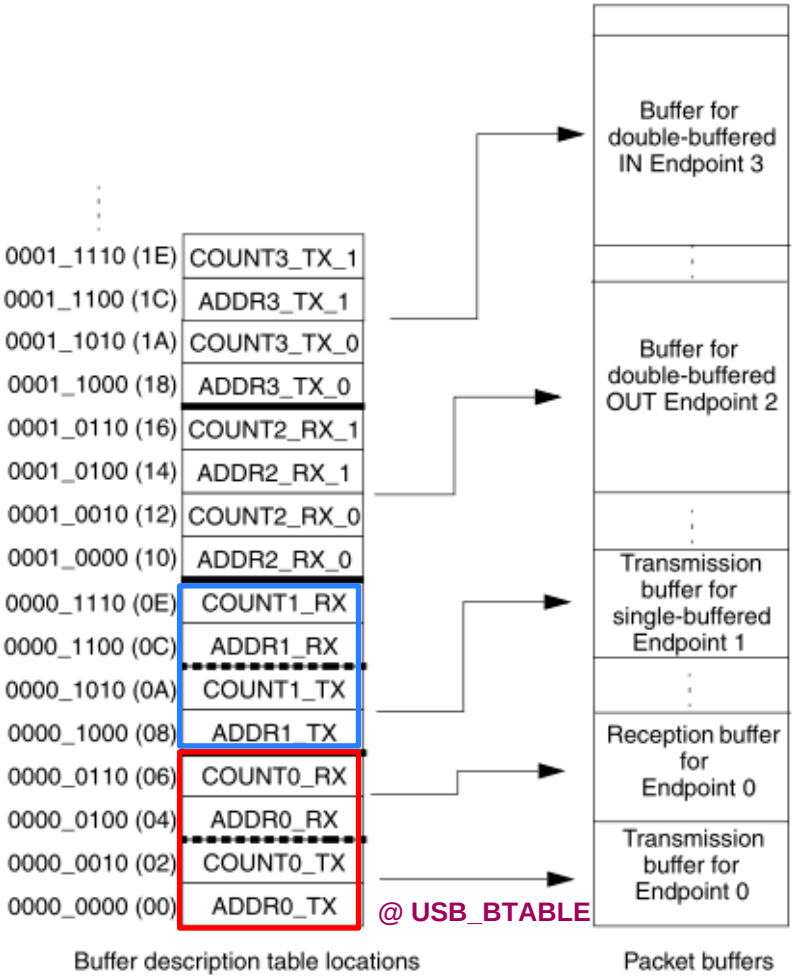
- 中断 mapper
  - NVIC.向量20：所有USB事件（正确传输结束、USB复位等）都可触发
  - NVIC.向量19：只能被同步和double buffer bulk传输的正确传输结束事件触发
  - NVIC.向量42：只能被(唤醒USB挂起模式)的事件触发 @EXTI_18
    - 从page100可以看到，只有host唤醒device，才会触发device的这个中断

1. 复位

- 系统复位和上电复位
  - 提供USB外设模块时钟
  - De-assert该模块的复位信号，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

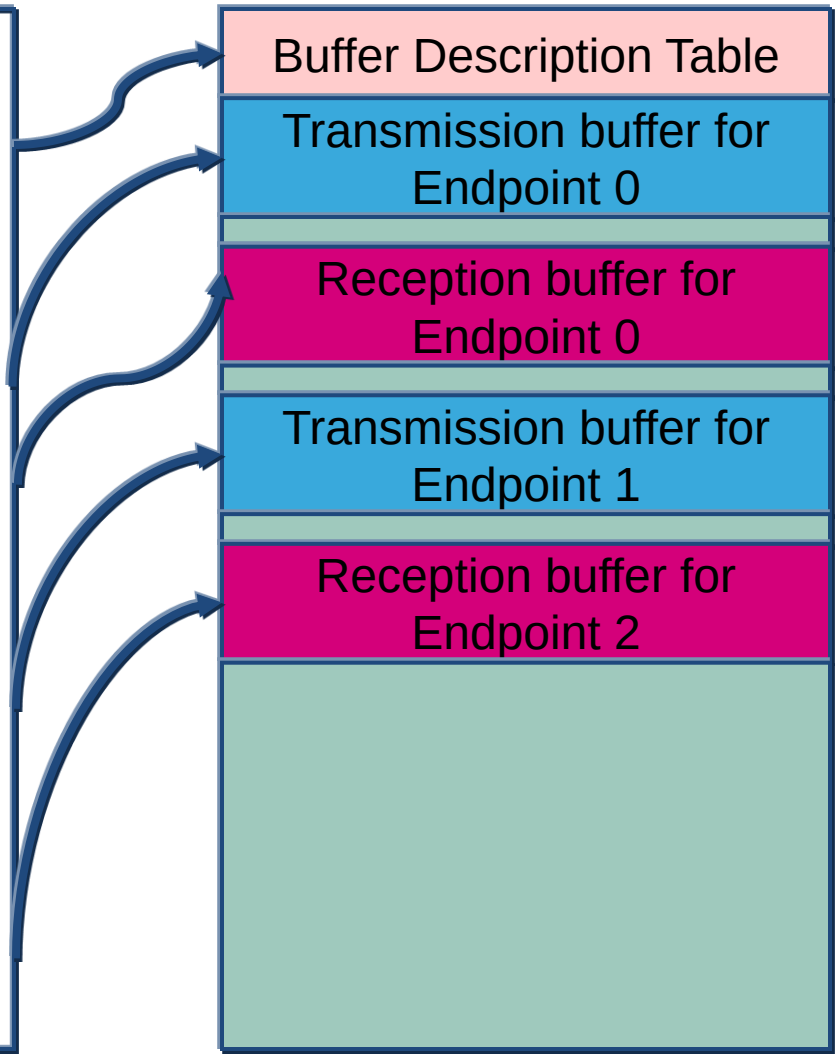
- 每个双向EP对应两个packet buffer，分别用于发送和接收
  - 软件通过packet buffer interface来访问它们
  - 这些packet buffer的位置和大小都可配置，由buffer描述表指定
  - Buffer描述表本身也在这块memory里，它自己的地址是由USB_BTABLE寄存器指定的
    - Table里每个entry由4个半字组成（分别表示双向EP的接收packet和发送packet的位置和大小）
    - 因此该table的位置本身必须以8字节对齐，即USB_BTABLE的低三位全部为0
  - USB外设硬件不会把本EP的数据溢出到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

```
@ <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

- 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>

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.MPS);
```

```
USB_SIL_Write() @ <usb_sil.c>

UserToPMABufferCopy(pBuf, GetEPTxAddr(bEpAddr & 0x7
SetEPTxCount((bEpAddr & 0x7F), size);

USB_SIL_Read () @ <usb_sil.c>

Length = GetEPRxCount(bEpAddr & 0x7F);
PMAToUserBufferCopy(pBuf, GetEPRxAddr(bEpAddr & 0x7
```

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

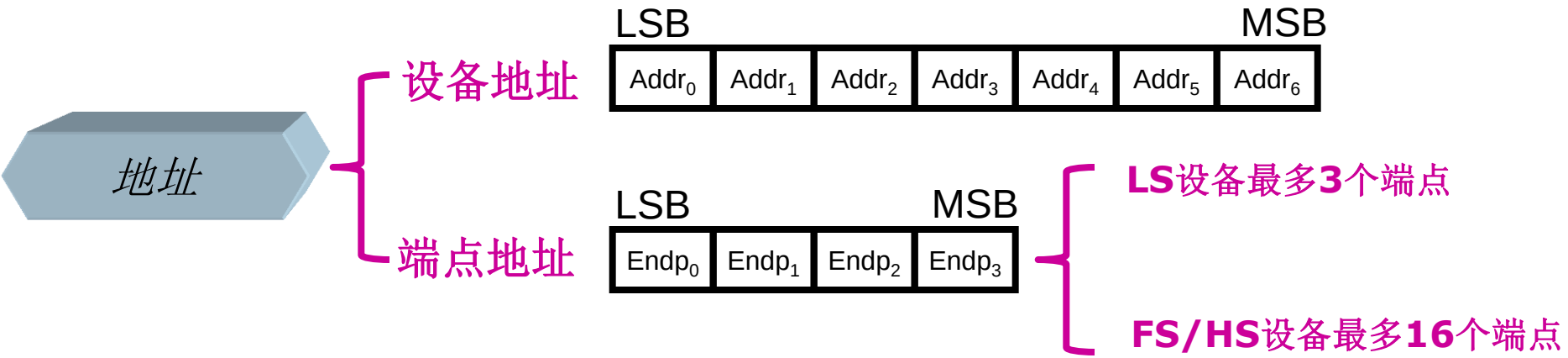
```
@ <MASS_Reset>

  SetBTABLE(0x00);

  SetEPRxAddr(ENDP0,   0x18);
  SetEPRxCount(ENDP0, 0x40);
  SetEPTxAddr(ENDP0,   0x58);

  SetEPTxAddr(ENDP1,   0x98);

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

地址	TX-ADDR	TX-CNT	RX-ADDR	RX-CNT	
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)
#define ENDP1_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)0x01);
```

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

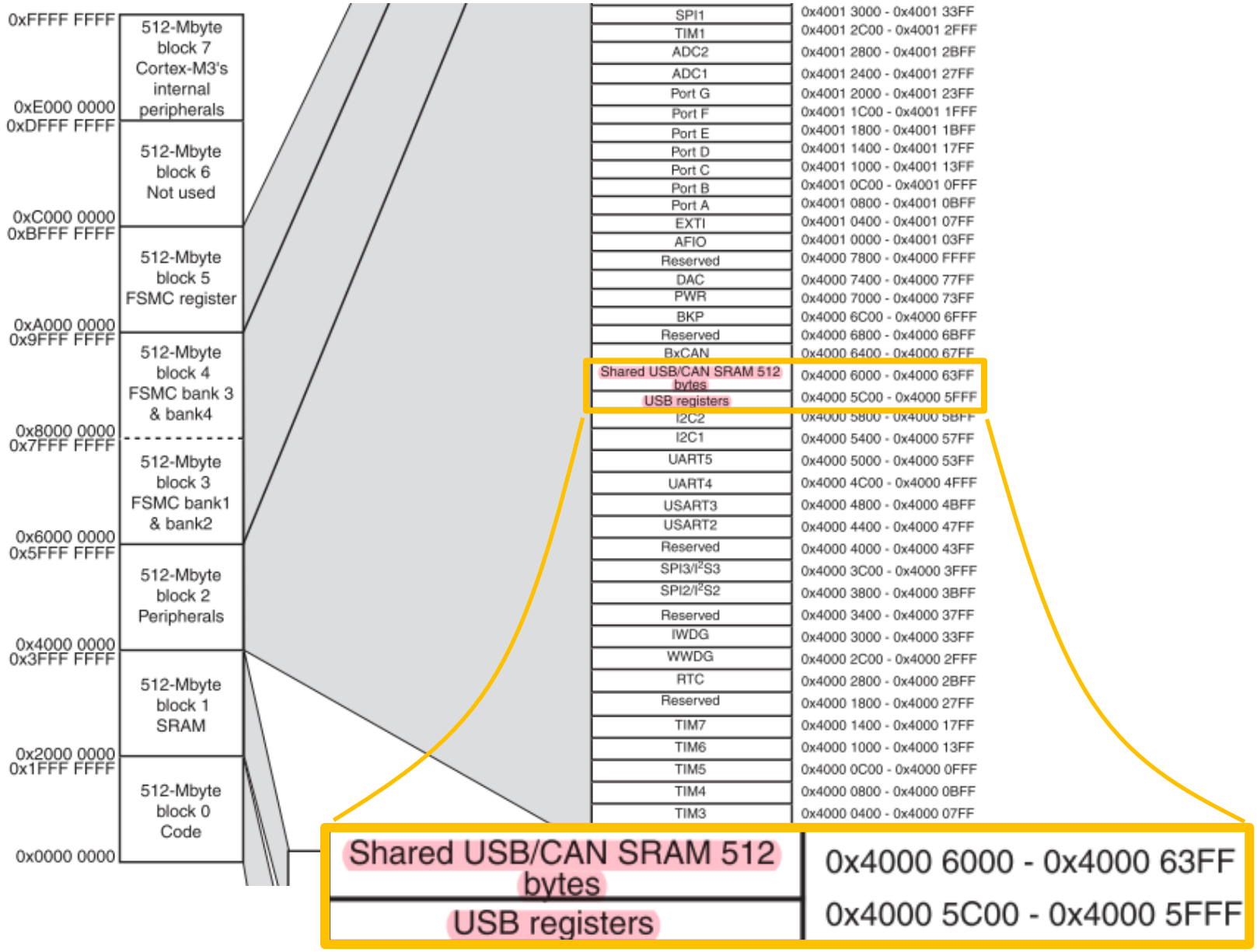
	EP_NUM	EP0		EP1		EP2		EP3	
		Rx	Tx	Rx	Tx	Rx	Tx	Rx	Tx
Audio	2			双缓冲					
Composite	3								
Custom HID	2								
DFU	1								
Joystick Mouse	2								
MSC	3								
VCP	4								

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

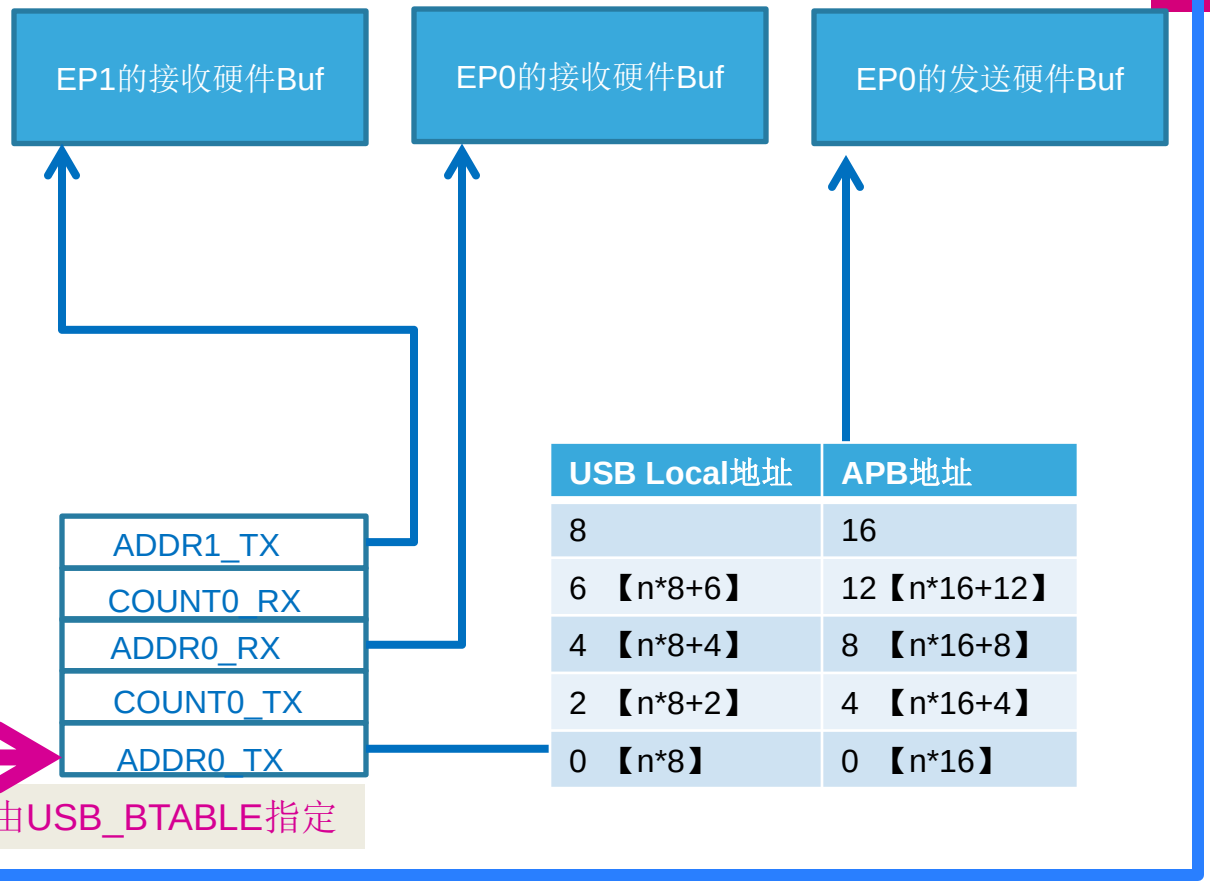

0xFFFF FFFF
512-Mbyte block 7 Cortex-M3's internal peripherals
0xE000 0000
0xDFFF FFFF
512-Mbyte block 6 Not used
0xC000 0000
0xBFFF FFFF
512-Mbyte block 5 FSMC register
0xA000 0000
0x9FFF FFFF
512-Mbyte block 4 FSMC bank 3 & bank4
0x8000 0000
0x7FFF FFFF
512-Mbyte block 3 FSMC bank1 & bank2
0x6000 0000
0x5FFF FFFF
512-Mbyte block 2 Peripherals
0x4000 0000
0x3FFF FFFF
512-Mbyte block 1 SRAM
0x2000 0000
0x1FFF FFFF
512-Mbyte block 0 Code
0x0000 0000
SPI1 0x4001 3000 - 0x4001 33FF
TIM1 0x4001 2C00 - 0x4001 2FFF
ADC2 0x4001 2800 - 0x4001 2BFF
ADC1 0x4001 2400 - 0x4001 27FF
Port G 0x4001 2000 - 0x4001 23FF
Port F 0x4001 1C00 - 0x4001 1FFF
Port E 0x4001 1800 - 0x4001 1BFF
Port D 0x4001 1400 - 0x4001 17FF
Port C 0x4001 1000 - 0x4001 13FF
Port B 0x4001 0C00 - 0x4001 0FFF
Port A 0x4001 0800 - 0x4001 0BFF
EXTI 0x4001 0400 - 0x4001 07FF
AFIO 0x4001 0000 - 0x4001 03FF
Reserved 0x4000 7800 - 0x4000 FFFF
DAC 0x4000 7400 - 0x4000 77FF
PWR 0x4000 7000 - 0x4000 73FF
BKP 0x4000 6C00 - 0x4000 6FFF
Reserved 0x4000 6800 - 0x4000 6BFF
BxCAN 0x4000 6400 - 0x4000 67FF
Shared USB/CAN SRAM 512 bytes 0x4000 6000 - 0x4000 63FF
USB registers 0x4000 5C00 - 0x4000 5FFF
I2C2
I2C1 0x4000 5400 - 0x4000 57FF
UART5 0x4000 5000 - 0x4000 53FF
UART4 0x4000 4C00 - 0x4000 4FFF
USART3 0x4000 4800 - 0x4000 4BFF
USART2 0x4000 4400 - 0x4000 47FF
Reserved 0x4000 4000 - 0x4000 43FF
SPI3/I2S3 0x4000 3C00 - 0x4000 3FFF
SPI2/I2S2 0x4000 3800 - 0x4000 3BFF
Reserved 0x4000 3400 - 0x4000 37FF
IWDG 0x4000 3000 - 0x4000 33FF
WWDG 0x4000 2C00 - 0x4000 2FFF
RTC 0x4000 2800 - 0x4000 2BFF
Reserved 0x4000 1800 - 0x4000 27FF
TIM7 0x4000 1400 - 0x4000 17FF
TIM6 0x4000 1000 - 0x4000 13FF
TIM5 0x4000 0C00 - 0x4000 0FFF
TIM4 0x4000 0800 - 0x4000 0BFF
TIM3 0x4000 0400 - 0x4000 07FF
Shared USB/CAN SRAM 512 bytes
0x4000 6000 - 0x4000 63FF
USB registers
0x4000 5C00 - 0x4000 5FFF

23.5.4 USB register map

Offset	Register
0x00	USB_EP0R
	Reset value
0x04	USB_EP1R
	Reset value
0x08	USB_EP2R
	Reset value
0x0C	USB_EP3R
	Reset value
0x10	USB_EP4R
	Reset value
0x14	USB_EP5R
	Reset value

Offset	Register
0x18	USB_EP6R
	Reset value
0x1C	USB_EP7R
	Reset value
0x20-0x3F	
0x40	USB_CNTR
	Reset value
0x44	USB_ISTR
	Reset value
0x48	USB_FNR
	Reset value
0x4C	USB_DADDR
	Reset value
0x50	USB_BTABLE

Table的起始地址由USB_BTABLE指定



USB Local地址	APB地址
8	16
6 【n*8+6】	12【n*16+12】
4 【n*8+4】	8 【n*16+8】
2 【n*8+2】	4 【n*16+4】
0 【n*8】	0 【n*16】

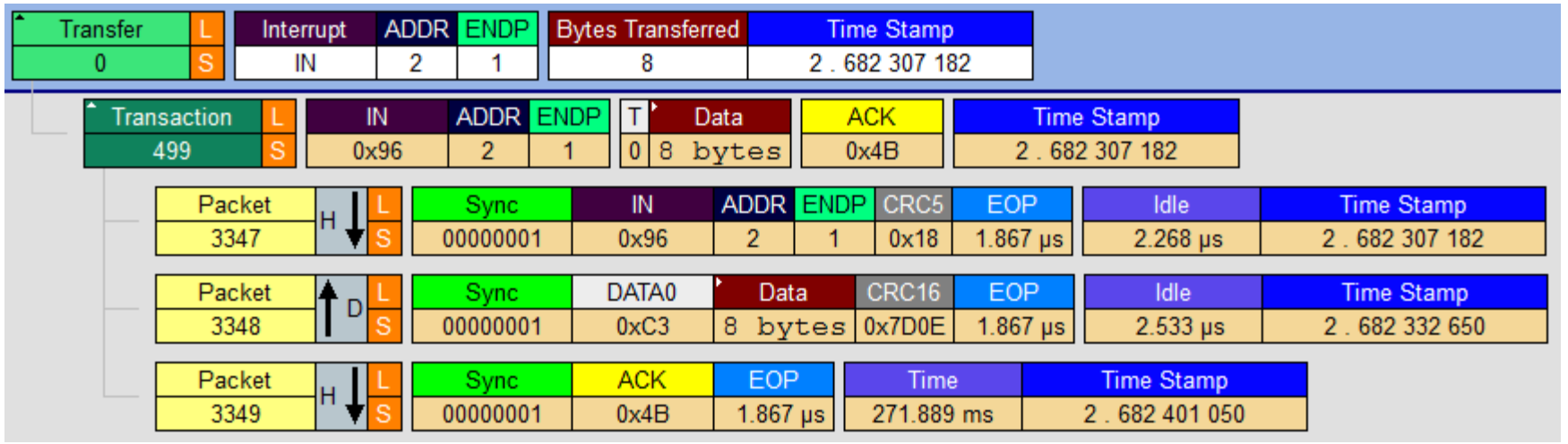
@0x4000 5c00
APB按字访问
：寄存器地址都是4字节对齐

@0x4000 6000 512字节专用RAM
Table中每个表项都是16位寄存器，USB local地址每个寄存器间隔2字节；APB地址每个寄存器间隔4字节（总是word访问，因此每半字插入16位的零填充）

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



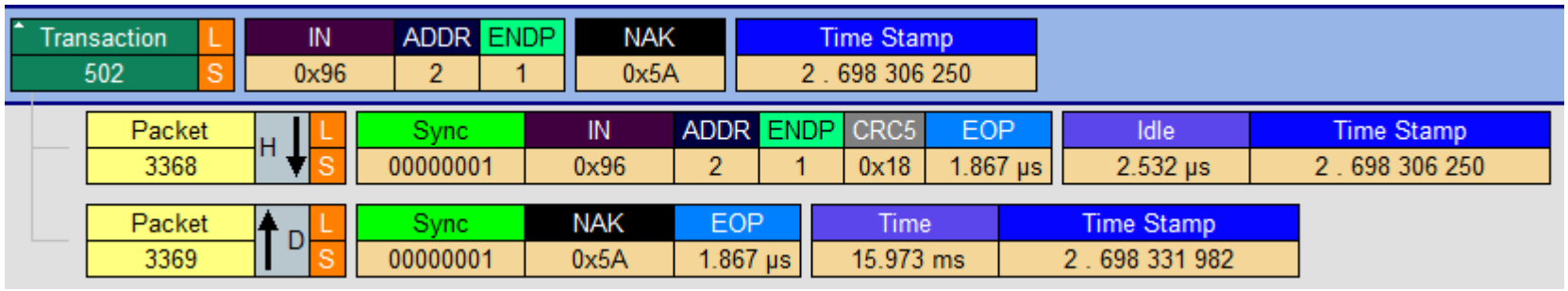
Transfer	L/S	Interrupt	ADDR	ENDP	Bytes Transferred	Time Stamp
0	S	IN	2	1	8	2 . 682 307 182

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

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

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

Packet	H/D	L/S	Sync	ACK	EOP	Time	Time Stamp
3349	H	S	00000001	0x4B	1.867 µs	271.889 ms	2 . 682 401 050



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

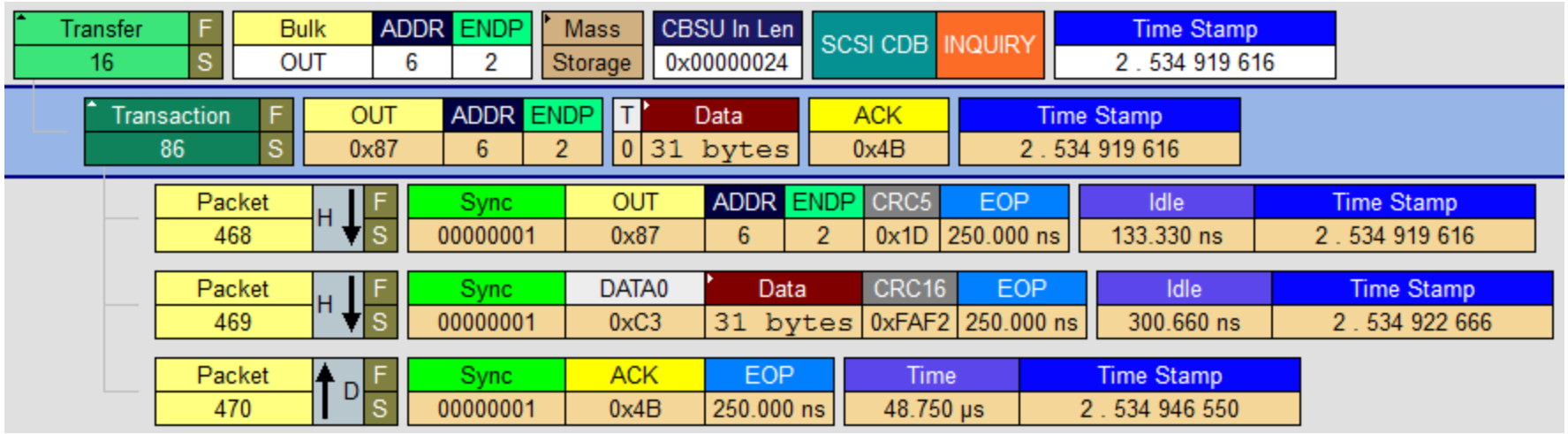
Packet	H/D	L/S	Sync	IN	ADDR	ENDP	CRC5	EOP	Idle	Time Stamp
3368	H	S	00000001	0x96	2	1	0x18	1.867 µs	2.532 µs	2 . 698 306 250

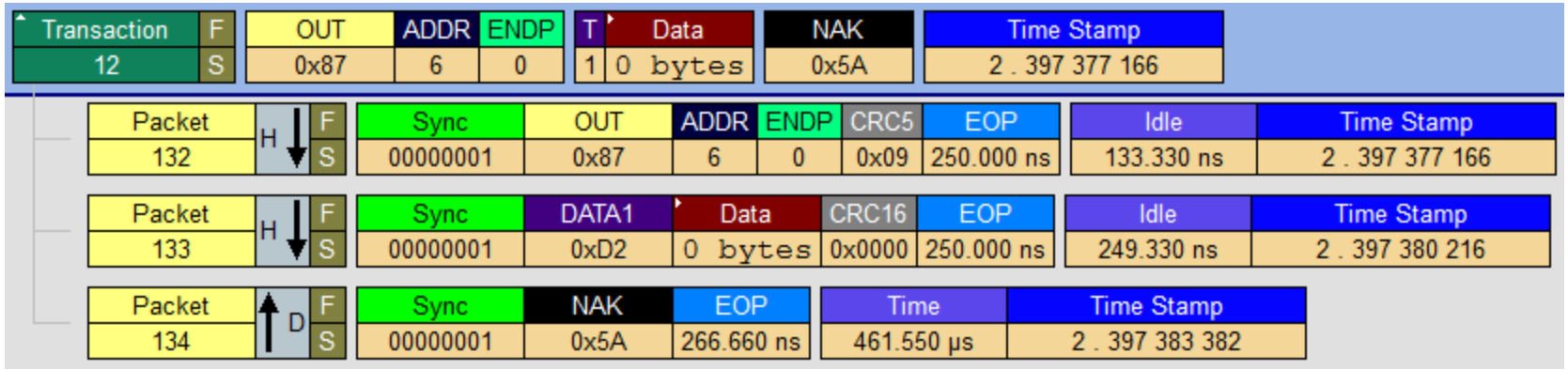
Packet	H/D	L/S	Sync	NAK	EOP	Time	Time Stamp
3369	D	S	00000001	0x5A	1.867 µs	15.973 ms	2 . 698 331 982

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外设收到主机返回的应答（ 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

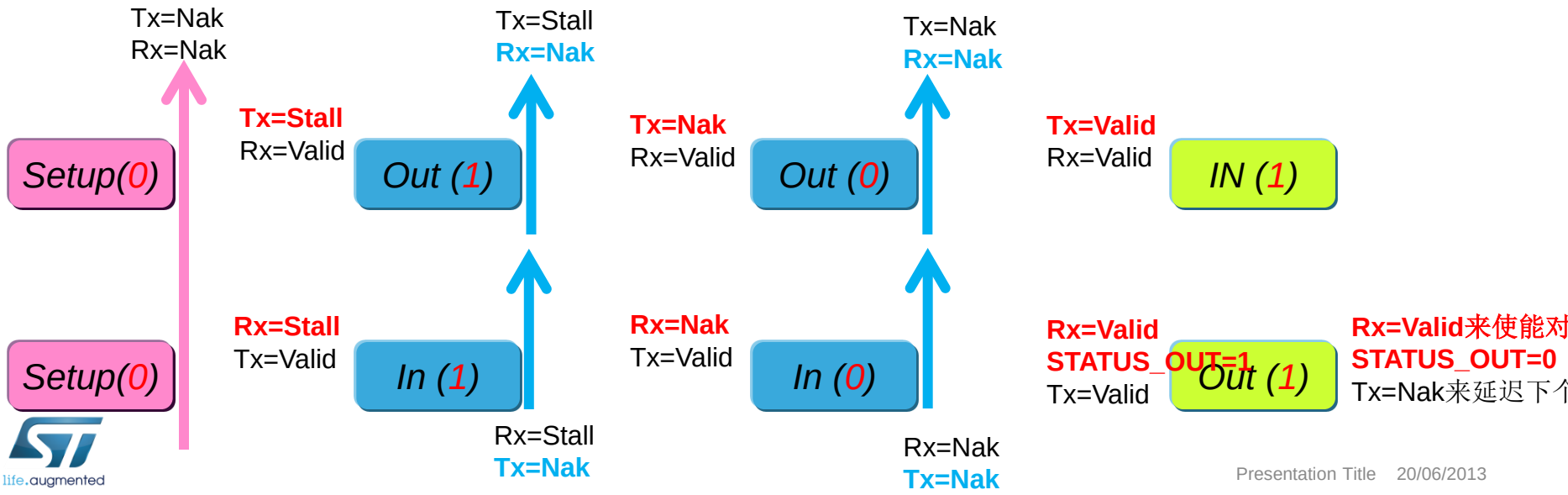
OUT transfer实例





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
    - 硬件把TX和RX的状态都设置成NAK
  - 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



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

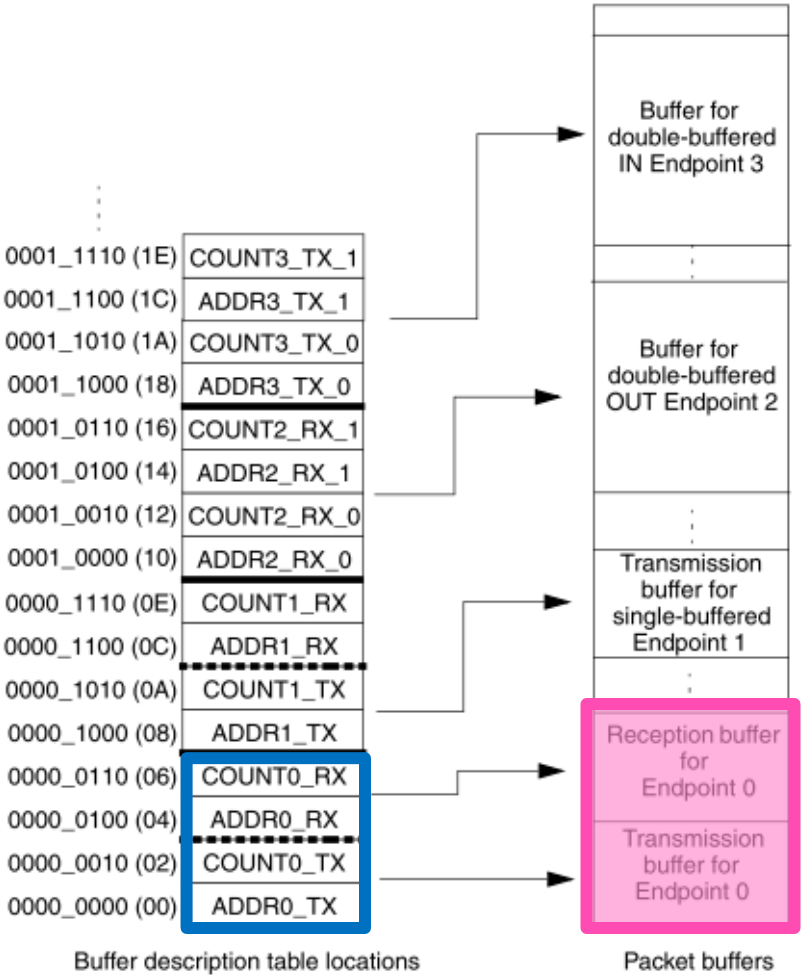
Packet	H	F	Sync	OUT	ADDR	ENDP	CRC5	EOP	Idle	Time Stamp
135	↓	S	00000001	0x87	6	0	0x09	250.000 ns	133.330 ns	2 . 397 388 366

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



- 双缓冲下的流控机制
  - 由应用程序负责对DTOG和SW_BUF进行初始化
  - USB通过SW_BUF(DTOG_X)来知道应用软件当前在用哪个packet memory
  - 只有在USB外设访问的buf和软件访问的buf冲突时才NAK数据流；而不是像通常情况下的一次transaction接收后就把当前EP当前方向状态更改成NAK
  - 每次transaction结束后，CTR_Rx或Tx会被硬件置位；该EP所在方向对应的DTOG被硬件翻转；STAT仍然保持Valid，只有当新的transaction的令牌包已经收到了，而此时DTOG和SW_BUF值还相同，则该EP的STAT被置成NAK
  - CTR_Rx或Tx置位后产生中断：sw需要对中断标志清零，并且在根据SW_BUF访问完相应buf后，软件toggle SW_BUF来告知USB外设它新toggle后对应的buf可以被使用了。如果在新的transaction来之前，sw没有及时更新SW_BUF，使得此时的SW_BUF还和刚才USB完成上次transaction后硬件自动toggle的DTOG一样的值，则EP被自动回NAK
  - 只要DBL_BUF被软件设置后，第一次transaction的结束就触发这个特殊的EP流控机制，直到DBL_BUF被软件复位后这种特殊的流控才不再apply

DTOG	SW_BUF	描述【假设该EP是OUT方向的BULK，即接收数据】
0	1	s/w初始化。那么1st数据将放到#0 buf中
1	1	1st数据到了，置位CTR_Rx，**硬件toggleDTOG成为1，EP的STAT仍然是Valid**
1	**0**	CTR对应的ISR中，sw根据1#去处理完数据后，**toggle SW_BUF成为0**
0	0	2nd数据到了，由于DTOG和SW_BUF不同，EP能继续接收，数据放在#1 buf中；置位CTR_RX，**硬件toggle DOTG成为0**，EP的STAT仍然是Valid
		这次的ISR没能及时响应，SW_BUF还是0，而3rd数据来了，发现DTOG=SW_BUF，知道发生conflict了，**因此EP的STAT被置成NAK**

Buffer flag	'Transmission' endpoint	'Reception' endpoint
DTOG	DTOG_TX (USB_EPnRbit 6)	DTOG_RX (USB_EPnRbit 14)
SW_BUF	USB_EPnR bit 14	USB_EPnR bit 6

Endpoint Type	DTOG	SW_BUF	Packet buffer used by USB Peripheral	Packet buffer used by Application Software
IN	0	1	ADDRn_TX_0 / COUNTn_TX_0 Buffer description table locations.	ADDRn_TX_1 / COUNTn_TX_1 Buffer description table locations.
	1	0	ADDRn_TX_1 / COUNTn_TX_1 Buffer description table locations	ADDRn_TX_0 / COUNTn_TX_0 Buffer description table locations.
	0	0	NAK	ADDRn_TX_0 / COUNTn_TX_0 Buffer description table locations.
	1	1		ADDRn_TX_0 / COUNTn_TX_0 Buffer description table locations.
OUT	0	1	ADDRn_RX_0 / COUNTn_RX_0 Buffer description table locations.	ADDRn_RX_1 / COUNTn_RX_1 Buffer description table locations.
	1	0	ADDRn_RX_1 / COUNTn_RX_1 Buffer description table locations.	ADDRn_RX_0 / COUNTn_RX_0 Buffer description table locations.
	0	0	NAK	ADDRn_RX_0 / COUNTn_RX_0 Buffer description table locations.
	1	1		ADDRn_RX_1 / COUNTn_RX_1 Buffer description table locations.

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

描述子@<usb_desc.c>

- **MASS_DeviceDescriptor** [18]
  - 该设备遵循USB2.0.0规范
  - 厂家来自ST（VID=0x0483）
  - EP0最大包长64字节
  - 该设备包含1个configuration

```
ONE_DESCRIPTOR Device_Descriptor =
 {
   (uint8_t*)MASS_DeviceDescriptor,
   MASS_SIZ_DEVICE_DESC
 }; @ <usb_prop.c>
```

```
ONE_DESCRIPTOR Config_Descriptor =
 {
   (uint8_t*)MASS_ConfigDescriptor,
   MASS_SIZ_CONFIG_DESC
 }; @ <usb_prop.c>
```

- **MASS_ConfigDescriptor** [32]
  - 该configuration包含1个interface
  - 该configuration的序号是1（用于SetConfiguration命令时）
  - 该设备由总线供电（全速工作时需要从总线获取100mA电流），不支持远程唤醒
  - 该interface的序号是0
  - 该interface拥有2个非零EP，它们遵循MSC类中的SCSI子类和BOT协议
  - EP1：IN方向，最大包长64字节
  - EP2：OUT方向，最大包长64字节

设备描述符的调用过程


```
ONE_DESCRIPTOR Device_Descriptor =
 {
   (uint8_t*)MASS_DeviceDescriptor,
   MASS_SIZ_DEVICE_DESC
 }; @ <usb_prop.c>
```

```
uint8_t *MASS_GetDeviceDescriptor(uint16_t Length)
{
  return Standard_GetDescriptorData(Length, &Device_Descriptor );
}
```

```
DEVICE_PROP Device_Property =
 {  @  《usb_prop.c》
   MASS_init,
   MASS_Reset,
   MASS_Status_In,
   MASS_Status_Out,
   MASS_Data_Setup,
   MASS_NoData_Setup,
   MASS_Get_Interface_Setting,
   MASS_GetDeviceDescriptor,
   MASS_GetConfigDescriptor,
   MASS_GetStringDescriptor,
   0,
   0x40 /*MAX PACKET SIZE*/
 };
```

```
typedef struct _DEVICE_PROP @  《usb_core.h》
{
  void (*Init)(void);        /* Initialize the device */
  void (*Reset)(void);       /* Reset routine of this device */
  void (*Process_Status_IN)(void);
  void (*Process_Status_OUT)(void);
  RESULT (*Class_Data_Setup)(uint8_t RequestNo);
  RESULT (*Class_NoData_Setup)(uint8_t RequestNo);
  RESULT  (*Class_Get_Interface_Setting)(uint8_t Interface, uint
  uint8_t* (*GetDeviceDescriptor)(uint16_t Length);
  uint8_t* (*GetConfigDescriptor)(uint16_t Length);
  uint8_t* (*GetStringDescriptor)(uint16_t Length);
  void* RxEP_buffer;
  uint8_t MaxPacketSize;
}DEVICE_PROP;
```

```
oid) @  《usb_init.c》

 &Device_Info;
ControlState = 2;
Device_Property;
rd_Requests = &User_Standard_Requests;
```

Data_Setup0 ← Setup0_Process ← OTGD_FS_Handle_OutEP_ISR

```
Data_Setup0() @  《usb_core.c》
{ ……
CopyRoutine = pProperty->GetDeviceDescriptor;
```

```
DEVICE_PROP Device_Property =
 { @ 《usb_prop.c》
   MASS_init,                          1
   MASS_Reset,                         2
   MASS_Status_In,                     3
   MASS_Status_Out,                    4
   MASS_Data_Setup,                    5
   MASS_NoData_Setup,                  6
   MASS_Get_Interface_Setting,         7
   MASS_GetDeviceDescriptor,           8
   MASS_GetConfigDescriptor,           9
   MASS_GetStringDescriptor,           10
   0,
   0x40 /*MAX PACKET SIZE*/
 };
```

```
typedef struct _DEVICE_PROP @ 《usb_core.h》
{
  void (*Init)(void);
  void (*Reset)(void);
  void (*Process_Status_IN)(void);
  void (*Process_Status_OUT)(void);
  RESULT (*Class_Data_Setup)(uint8_t RequestNo);
  RESULT (*Class_NoData_Setup)(uint8_t RequestNo);
  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);
  void* RxEP_buffer;
  uint8_t MaxPacketSize;
}DEVICE_PROP;
```

```
void USB_Init(void) @ 《usb_init.c》
{
  pProperty->Init();          1
}
```

```
Void USB_Istr(void) @ 《usb_istr.c》
{
  Device_Property.Reset();
}                             2
```

```
uint8_t In0_Process(void) @ 《usb_core.c》
{
  (*pProperty->Process_Status_IN)();      3
}
```

```
uint8_t Out0_Process(void) @ 《usb_core.c》
{
  (*pProperty->Process_Status_OUT)();     4
}
```

```
void Data_Setup0(void)  @ 《usb_core.c》
{
 pProperty->GetDeviceDescriptor;                                          8

 pProperty->GetConfigDescriptor;                                          9

 pProperty->GetStringDescriptor;                                          10

 *pProperty->Class_Get_Interface_Setting)(pInformation->USBwIndex0, 0);   7

 (*pProperty->Class_Data_Setup)(pInformation->USBbRequest);               5
}
```

```
void NoData_Setup0(void)  @ 《usb_core.c》
{
 (*pProperty->Class_NoData_Setup)(RequestNo);        6
}
```

字符串描述符

- MASS_StringLangID [4]
  - /* LangID = 0x0409: U.S. English */ 序号=0
- MASS_StringVendor [38]
  - /* Manufacturer: "STMicroelectronics" */ 序号=1
- MASS_StringProduct [38]
  - /* Product name: "STM32F10x: USB Mass Storage" */ 序号=2
- MASS_StringSerial [26]
  - /* Serial number ： STM32L1或者STM3210 */ 序号=3
- MASS_StringInterface [16]
  - /* Interface 0: "ST Mass" */ 序号=4

```
const uint8_t MASS_DeviceDescriptor [18] =
 {
   ……
   1, // string index of Manufacturer
   2, // string index of product
   3, // string index of serial
   ……
 };
```

```
ONE_DESCRIPTOR String_Descriptor[5] =
 {
   {(uint8_t*)MASS_StringLangID, MASS_SIZ_STRING_LANGID},
   {(uint8_t*)MASS_StringVendor, MASS_SIZ_STRING_VENDOR},
   {(uint8_t*)MASS_StringProduct, MASS_SIZ_STRING_PRODUCT},
   {(uint8_t*)MASS_StringSerial, MASS_SIZ_STRING_SERIAL},
   {(uint8_t*)MASS_StringInterface, MASS_SIZ_STRING_INTERFACE},
 };
```

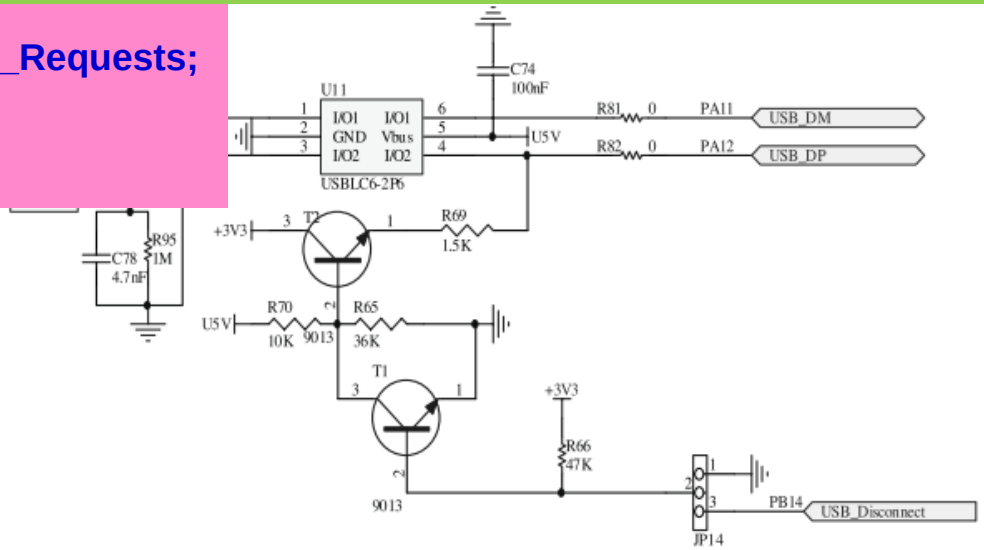
初始化


```
@ 《main.c》

Set_System();                 // 配置PB14控制USB插入或拔除；初始化SD卡（作为模拟U盘的唯一一个盘符）
Set_USBClock();               // 配置USB时钟分频以达到48MHz，并使能USB模块时钟
Led_Config();
USB_Interrupts_Config();      // 使能USB在NVIC中的中断通道，LP和HP都打开，WakeUp没有（U盘无唤醒功能）

USB_Init();

while (bDeviceState != CONFIGURED);
```

```
bDeviceState是一个全局变量，可在调试时查看当前USB模块的状态：
typedef enum _DEVICE_STATE
{
  UNCONNECTED,        <- MASS_init即pProperty->Init()；默认值
  ATTACHED,           <- Leave_LowPowerMode的条件二; MASS_Reset即
  POWERED,
  SUSPENDED,          <- Enter_LowPowerMode
  ADDRESSED,          <- Mass_Storage_SetDeviceAddress
  CONFIGURED          <- Leave_LowPowerMode的条件一;Mass_Storage_
} DEVICE_STATE;
```

```
void USB_Init(void) @ 《usb_init.c》
{
  pInformation = &Device_Info;
  pInformation->ControlState = 2;
  pProperty = &Device_Property;
  pUser_Standard_Requests = &User_Standard_Requests;

  pProperty->Init();          // 即MASS_Init()
}
```

1



结构体初始化（1）

- 分配了一个结构体（Device_Info）来记录收到的主机命令，以及设备当前状态；并用一个指针（pInformation）来指向它

```
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 interfa

  ENDPOINT_INFO Ctrl_Info;
}DEVICE_INFO;

DEVICE_INFO  Device_Info;
DEVICE_INFO  *pInformation;

@ 《USB_Init》
pInformation = &Device_Info;
pInformation->ControlState = 2;
```

```
typedef struct _ENDPOINT_INFO
{
uint16_t   Usb_wLength;
 uint16_t  Usb_wOffset;
 uint16_t  PacketSize;
 uint8_t    *(*CopyData)(uint16_t Length);
}ENDPOINT_INFO;
```

```
typedef enum
_CONTROL_STATE
{
  WAIT_SETUP,          /* 0 */
  SETTING_UP,          /* 1 */
  IN_DATA,             /* 2 */
  OUT_DATA,            /* 3 */
  LAST_IN_DATA,        /* 4 */
  LAST_OUT_DATA,       /* 5 */
  WAIT_STATUS_IN,      /* 7 */
  WAIT_STATUS_OUT,     /* 8 */
  STALLED,             /* 9 */
  PAUSE                /* 10 */
} CONTROL_STATE;
```

ENDPOINT_INFO


```
typedef struct _ENDPOINT_INFO
{
 uint16_t   Usb_wLength;
 uint16_t  Usb_wOffset;
 uint16_t  PacketSize;
 uint8_t    *(*CopyData)(uint16_t Length);
}ENDPOINT_INFO;
```

@ <Data_Setup0>

标准命令：

```
 if (CopyRoutine)
  {
    pInformation->Ctrl_Info.Usb_wOffset = wOffset; （0）
    pInformation->Ctrl_Info.CopyData = CopyRoutine;
    (*CopyRoutine)(0);  → pInformation->Ctrl_Info.Usb_wLength = …
    Result = USB_SUCCESS;
  }
```

类相关或用户自定义命令：

```
(*pProperty->Class_Data_Setup)(pInformation->USBbRequest)
即              pInformation->Ctrl_Info.CopyData = CopyRoutine;
                pInformation->Ctrl_Info.Usb_wOffset = 0;
                (*CopyRoutine)(0);  → pInformation->Ctrl_Info.Usb_wLength = …
```

如果该控制传输的数据阶段是device->host，在此用MPS来初始化之

```
pInformation->Ctrl_Info.PacketSize = pProperty->MaxPacketSize
```

@ <DataStageIn>

```
UserToPMABufferCopy(DataBuffer, GetEPTxAddr(ENDP0),
SetEPTxCount(ENDP0, Length);

pEPinfo->Usb_wLength -= Length;
pEPinfo->Usb_wOffset += Length;

vSetEPTxStatus(EP_TX_VALID);
```

@ <DataStageOut>

```
pEPinfo->Usb_rLength -= Length;        #define      Usb_r
pEPinfo->Usb_rOffset += Length;        #define      Usb_r

PMAToUserBufferCopy(Buffer, GetEPRxAddr(ENDP0), Lengt

if (pEPinfo->Usb_rLength != 0)
  {
    vSetEPRxStatus(EP_RX_VALID);/* re-enable for next data
    SetEPTxCount(ENDP0, 0);
```

life.augmented

结构体初始化（2）

- 分配了一个结构体（Device_Property）来定义该设备的各个回调函数；并用一个指针（pProperty）来指向它

```
DEVICE_PROP Device_Property =
 {
   MASS_init,
   MASS_Reset,
   MASS_Status_In,
   MASS_Status_Out,
   MASS_Data_Setup,
   MASS_NoData_Setup,
   MASS_Get_Interface_Setting,
   MASS_GetDeviceDescriptor,
   MASS_GetConfigDescriptor,
   MASS_GetStringDescriptor,
   0,
   0x40 /*MAX PACKET SIZE*/
 };

DEVICE_PROP *pProperty;

@《usb_init》
pProperty = &Device_Property;
pProperty->Init();
```

```
typedef struct _DEVICE_PROP
{
  void      (*Init)                    (void);
  void      (*Reset)                   (void);
  void      (*Process_Status_IN)       (void);
  void      (*Process_Status_OUT)      (void);
  RESULT    (*Class_Data_Setup)        (uint8_t RequestNo);
  RESULT    (*Class_NoData_Setup)      (uint8_t RequestNo);
  RESULT    (*Class_Get_Interface_Setting)(uint8_t Interface, uint8_t AlternateSettin
  uint8_t*  (*GetDeviceDescriptor)     (uint16_t Length);
  uint8_t*  (*GetConfigDescriptor)     (uint16_t Length);
  uint8_t*  (*GetStringDescriptor)     (uint16_t Length);
  void* RxEP_buffer;
  uint8_t MaxPacketSize;
}DEVICE_PROP;
```

结构体初始化（3）

- 分配了一个结构体（User_Standard_Requests）来定义该设备用来响应主机标准命令时的各种回调函数；并用一个指针（pUser_Standard_Requests）来指向它

```
USER_STANDARD_REQUESTS User_Standard_Requests =
 {
   Mass_Storage_GetConfiguration,      （nop）
   Mass_Storage_SetConfiguration,      更新全局变量bDeviceState，
                                       复位DTOG
   Mass_Storage_GetInterface,          （nop）
   Mass_Storage_SetInterface,          （nop）
   Mass_Storage_GetStatus,             （nop）
   Mass_Storage_ClearFeature,          STALL EP1-Tx, EP2-RX
   Mass_Storage_SetEndPointFeature,    （nop）
   Mass_Storage_SetDeviceFeature,      （nop）
   Mass_Storage_SetDeviceAddress       更新全局变量bDeviceState
 };
USER_STANDARD_REQUESTS  *pUser_Standard_Requests;

@《usb_init》
pUser_Standard_Requests = &User_Standard_R
```

```
typedef struct _USER_STANDARD_REQUESTS
{
  void (*User_GetConfiguration)(void);
  void (*User_SetConfiguration)(void);
  void (*User_GetInterface)(void);
  void (*User_SetInterface)(void);
  void (*User_GetStatus)(void);
  void (*User_ClearFeature)(void);
  void (*User_SetEndPointFeature)(void);
  void (*User_SetDeviceFeature)(void);
  void (*User_SetDeviceAddress)(void);
}
USER_STANDARD_REQUESTS;
```

```
#define Mass_Storage_GetConfiguration        NOP_Process
/* #define Mass_Storage_SetConfiguration     NOP_Process*/
#define Mass_Storage_GetInterface            NOP_Process
#define Mass_Storage_SetInterface            NOP_Process
#define Mass_Storage_GetStatus               NOP_Process
/* #define Mass_Storage_ClearFeature         NOP_Process*/
#define Mass_Storage_SetEndPointFeature      NOP_Process
#define Mass_Storage_SetDeviceFeature        NOP_Process
/*#define Mass_Storage_SetDeviceAddress      NOP_Process*/
```

life.augmented

MASS_init()

- 获取序列号
  - 根据该芯片的Unique ID来更新“序号字符串”（MASS_StringSerial[26]）
- 连接设备：PowerOn()
  - 拉低PB14来使能USB_D+线上的上拉电阻
  - 置位并复位FRES@USB_CNTR来对USB外设进行复位
  - 复位ISTR所有位来清除可能pending的中断
  - 设置本应用关心的中断事件：RESET、SUSP、WKUP
- USB_SIL_Init()
  - 复位ISTR所有位来清除可能pending的中断
  - 用**IMR_MSK**重新设置本应用关心的中断事件：CTRM、RESET
- pInformation->Current_Configuration = 0
- bDeviceState = UNCONNECTED

```
#define IMR_MSK (CNTR_CTRM  | CNTR_RESETM)
```

整个USB通信是由中断驱动的

- 不同应用关心不同的中断事件

```
void USB_Istr(void)
{
 wIstr = _GetISTR();

#if (IMR_MSK & ISTR_CTR)
  if (wIstr & ISTR_CTR & wInterrupt_Mask)
  {
    CTR_LP();
#ifdef CTR_CALLBACK
    CTR_Callback();
#endif
  }
#endif

#if (IMR_MSK & ISTR_RESET)
  if (wIstr & ISTR_RESET & wInterrupt_Mask)
  {
    _SetISTR((uint16_t)CLR_RESET);
    Device_Property.Reset();
#ifdef RESET_CALLBACK
    RESET_Callback();
#endif
  }
#endif
}
```

```
@ 《usb_conf.h》 USB MSC Device demo
/* mask defining which events has to be handled */
/* by the device application software */
#define IMR_MSK (CNTR_CTRM  | CNTR_RESETM)

@ 《usb_conf.h》 USB JOYSTICK demo
#define IMR_MSK (CNTR_CTRM  | CNTR_WKUPM
                 | CNTR_SUSPM | CNTR_ERRM  | CNTR_SOFM \
                 | CNTR_ESOFM | CNTR_RESETM )
```

枚举过程Tracer file

RESET中断

- 软件写0来清零该标志，否则收发操作被disable
- MASS_Reset
  - Device_Info.Current_Configuration = 0（同前）
  - pInformation->Current_Feature = MASS_ConfigDescriptor[7]
  - 设置Buffer table的地址（BTABLE_ADDRESS），设备地址默认为0
  - 初始化EP
    - EP0：控制类型；发送NAK、接收Valid；设置接收buffer地址和长度；设置发送buffer地址
    - EP1：批量类型；发送NAK、接收Disable；设置发送buffer地址
    - EP2：批量类型；发送Disable、接收VALID；设置接收buffer地址和长度
  - bDeviceState = ATTACHED 全局变量，表示设备当前已被插入主机

```
@ 《usb_conf.h》
#define BTABLE_ADDRESS      (0x00)

#define ENDP0_RXADDR        (0x18)
#define ENDP0_TXADDR        (0x58)

#define ENDP1_TXADDR        (0x98)
#define ENDP2_RXADDR        (0xD8)
```

端点	类型	发送			接收		
		状态	地址	长度	状态	地址	长度
EP0	CTL	NAK	0x58	64	VALID	0x18	64
EP1	BULK	NAK	0x98	64	DIS		
EP2	BULK	DIS			VALID	0xD8	64

CTR中断

- CTR @USB_ISTR
  - 硬件置位表示一个transaction正确完成了
  - 软件需要根据EP_ID和DIR来得知这次transaction是发生在哪个EP上的何种transaction（SETUP/OUT/IN transaction?）

中断使能控制→

USB control register (USB_CNTR)

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
CTRM	PMAOVRM	ERRM	WKUPM	SUSPM	RESETM	SOFM	ESOFM	Reserved			RESUME	FSUSP	LP_MODE	PDWN	FRES
rw	rw	rw	rw	rw	rw	rw	rw				rw	rw	rw	rw	rw

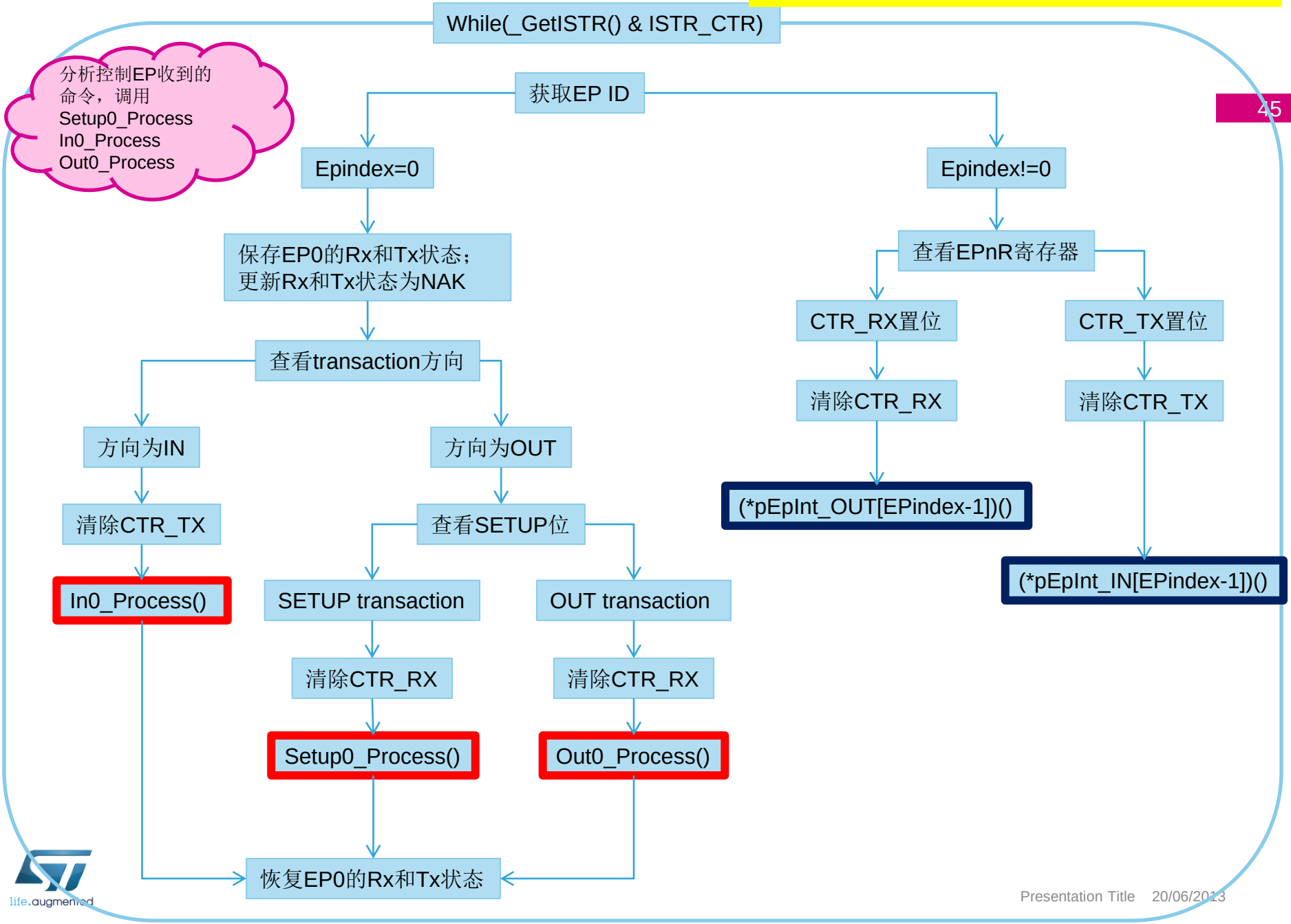
中断标志→

USB interrupt status register (USB_ISTR)

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
CTR	PMA OVR	ERR	WKUP	SUSP	RESET	SOF	ESOF	Reserved			DIR	EP_ID[3:0]			
r	rc_w0	rc_w0	rc_w0	rc_w0	rc_w0	rc_w0	rc_w0				r	r	r	r	r

- CTR_RX & CTR_TX @USB_EPnR (n=0…7)
  - CTR_TX
    - 该EP上的**IN transaction**正确完成，硬件置位它；对于被NAK或STALL的IN transaction，硬件不会置位（比如协议错误，或数据toggle位不匹配）
  - CTR_RX
    - 该EP上的**OUT/SETUP transaction**正确完成时，硬件置位它（由**SETUP位决定**到底是哪种）；对于被NAK或STALL的transaction，硬件不会置位（比如协议错误，或数据toggle位不匹配）

void CTR_LP(void) @《usb_int.c》



Setup0_Process()

从EP0的接收Packet buf中读取主机命令，
填充pInformation所指向的**Device_Info**结构体，
pInformation->ControlState = SETTING UP

pInformation->USBwLength

=0

! =0

NoData_Setup0()

处理USB协议的标准request；对于类相关request，在***pProperty->Class_NoData_Setup**中处理。对于MSC demo，该回调函数就是**MASS_NoData_Setup()**，处理的就是BOT_RESET命令
CS = WAIT_STATUS_IN
USB_StatusIN()，即Send0LengthData()
（即设置EP0的Tx为Valid，Tx CNT=0）

Data_Setup0()

处理USB协议的标准request；对于类相关request，在***pProperty->Class_Data_Setup**中处理。对于MSC demo，该回调函数就是**MASS_Data_Setup()**，处理就是Get_Max_LUN命令
再根据数据阶段的传输方向：
>>【IN】**DataStageIn()** → UserToPMABufferCopy
（CS=(LAST_)IN_DATA/WAIT_STATUS_OUT）
>>【OUT】设置EP的Rx为Valid，来使能继续接收
（ControlState = OUT_DATA）

Post0_Process()

三种控制传输

Transfer	L	Control	ADDR	ENDP	bRequest	wValue	wIndex	Descriptors	Time Stamp
1	S	GET	2	0	GET_DESCRIPTOR	DEVICE type	0x0000	DEVICE Descriptor	2 . 391 318 516

Transaction	L	SETUP	ADDR	ENDP	T	D	Tp	R	bRequest	wValue	wIndex	wLength	ACK	Time	Time Stamp
14	S	0xB4	2	0	0	D->H	S	D	0x06	0x0100	0x0000	18	0x4B	999.200 µs	2 . 391 318 516

Transaction	L	IN	ADDR	ENDP	T	Data	ACK	Time	Time Stamp
15	S	0x96	2	0	1	8 bytes	0x4B	2.983 ms	2 . 392 317 716
18	S	0x96	2	0	0	8 bytes	0x4B	3.018 ms	2 . 395 301 050
22	S	0x96	2	0	1	2 bytes	0x4B	1.000 ms	2 . 398 319 050
23	S	0x87	2	0	1	0 bytes	0x4B	999.734 µs	2 . 399 319 182

Tp（类型）= standard
R（接收） = device

Transfer	L	Control	ADDR	ENDP	D	Tp	R	bRequest	wValue	wIndex	wLength	Bytes Transferred	Time Stamp
12	S	SET	2	0	H->D	C	I	0x09	0x0200	0x0000	1	1	4 . 248 208 116

Transaction	L	SETUP	ADDR	ENDP	D	Tp	R	bRequest	wValue	wIndex	wLength	ACK	Time	Time Stamp
459	S	0xB4	2	0	H->D	C	I	0x09	0x0200	0x0000	1	0x4B	2.000 ms	4 . 248 208 116

Transaction	L	OUT	ADDR	ENDP	T	Data	ACK	Time	Time Stamp
461	S	0x87	2	0	1	1 byte	0x4B	975.734 µs	4 . 250 208 116

Transaction	L	IN	ADDR	ENDP	T	Data	ACK	Time Stamp
462	S	0x96	2	0	1	0 bytes	0x4B	4 . 251 183 850

Tp（类型）= class
R（接收） = interface

Transfer	L	Control	ADDR	ENDP	bRequest	wValue	wIndex	wLength	Time Stamp
9	S	SET	2	0	SET_CONFIGURATION	New Configuration 1	0x0000	0	2 . 525 295 182

Transaction	L	SETUP	ADDR	ENDP	T	D	Tp	R	bRequest	wValue	wIndex	wLength	ACK	Time	Time Stamp
116	S	0xB4	2	0	0	H->D	S	D	0x09	0x0001	0x0000	0	0x4B	2.002 ms	2 . 525 295 182

Transaction	L	IN	ADDR	ENDP	T	Data	ACK	Time	Time Stamp
118	S	0x96	2	0	1	0 bytes	0x4B	26.997 ms	2 . 527 297 316

Data_Setup0()

pInformation指向的是Device_Info结构体

指针pProperty指向结构体Device_Property

Request_No = pInformation->USBwLength

- GetDescriptor（6）
  - 根据参数USBwValue1
    - 设备描述符CopyRoutine= pProperty->GetDeviceDescriptor，即MASS_GetDeviceDescriptor
    - 配置描述符CopyRoutine = pProperty->GetConfigDescriptor，即MASS_GetConfigDescriptor
    - 字符串描述符CopyRoutine = pProperty->GetStringDescriptor，即MASS_GetStringDescriptor
- GetStatus（0）
  - 根据参数recipent
    - 设备状态 CopyRoutine = Standard_GetStatus
    - 接口状态 *pProperty->Class_Get_Interface_Setting，即MASS_Get_Interface_Setting
CopyRoutine = Standard_GetStatus
    - 端点状态 Status = _GetEPTxStatus /_GetEPRxStatus
若EP没有被disable，才CopyRoutine = Standard_GetStatus;
- GetConfiguration（8）
  - CopyRoutine = Standard_GetConfiguration
- GetInterface（10）
  - *pProperty->Class_Get_Interface_Setting，即MASS_Get_Interface_Setting
CopyRoutine = Standard_GetInterface

未完！

CopyRoutine

【原型】函数指针：函数参数是16位变量，函数返回值是指向8位变量的指针
【默认值】CopyRoutine = NULL @Data_Setup0(void)和MASS_Data_Setup(RequestNo)
【根据主机发出的Request而动态赋值】

@ <Data_Setup0>

>> CopyRoutine = pProperty->GetDeviceDescriptor （A）
>> CopyRoutine = pProperty->GetConfigDescriptor （B）
>> CopyRoutine = pProperty->GetStringDescriptor （C）
>> CopyRoutine = Standard_GetStatus （D）
>> CopyRoutine = Standard_GetConfiguration （E）
>> CopyRoutine = Standard_GetInterface （F）

@ <usb_prop.c>

@ <usb_core.c>

@ <MASS_Data_Setup>

>> CopyRoutine = Get_Max_Lun （G） → @ <usb_prop.c>

【作用和意义】通过这个函数获得用户的数据缓冲区地址，从而可以在OUT过程中把收到的数据拷贝到用户缓冲区，或在IN过程中把用户缓冲区的数据拷贝到USB发送缓冲区

>>OUT过程，多个DATA_OUT传输，库需要多次调用回调函数CopyRoutine，返回将要容纳已经收在硬件buf中数据的缓冲区指针

>>IN过程，多个DATA_IN传输，每次需要向主机传输数据时，USB库都会调用一次回调函数CopyRoutine，它返回一个包含要发送的数据的缓冲区指针，库再把这个缓冲区的内容拷贝的硬件buf以择机发送出去

当以length=0调用CopyRoutine时，CopyRoutine需要返回用户缓冲区的长度

@ <Data_Setup0>

>> pInformation->Ctrl_Info.Usb_wOffset = wOffset
>> pInformation->Ctrl_Info.CopyData = CopyRoutine
>> (*CopyRoutine)(0)

@ <Mass_Data_Setup>

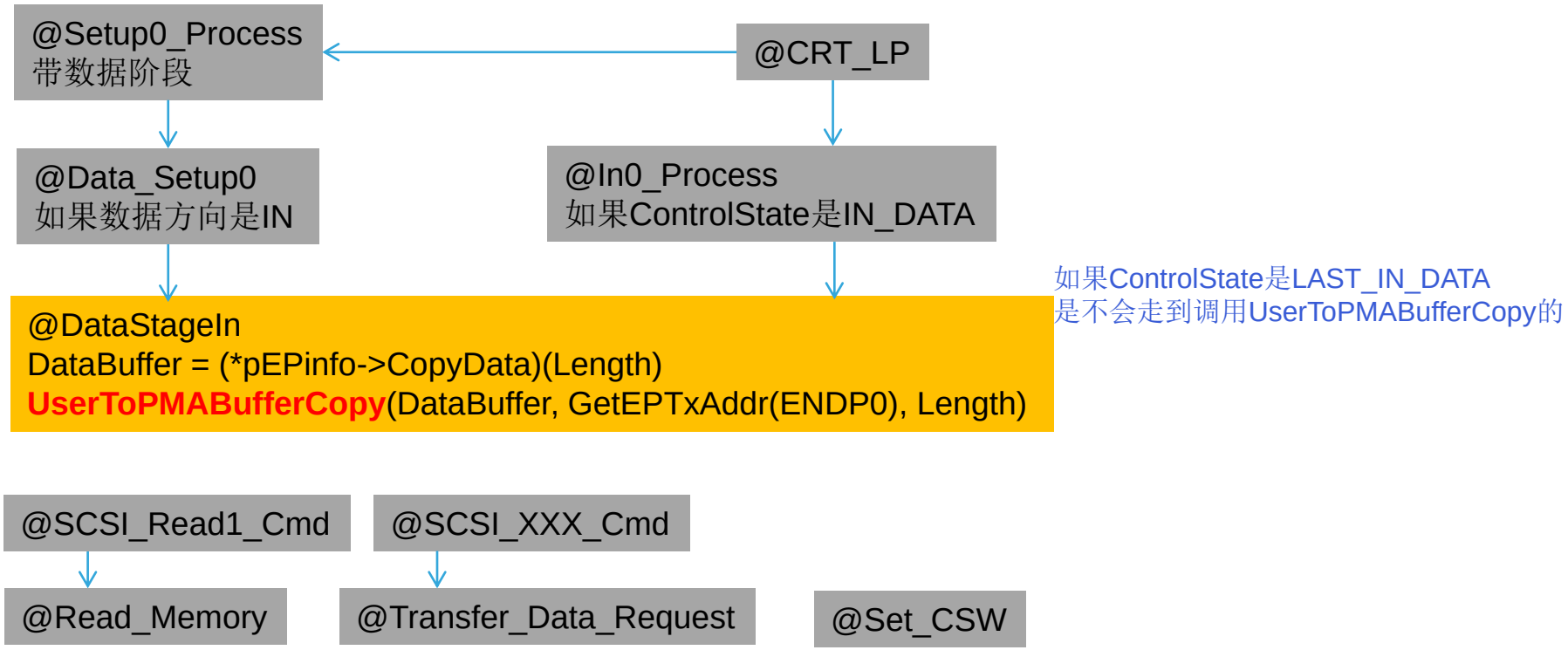
pInformation->Ctrl_Info.Usb_wOffset = 0
pInformation->Ctrl_Info.CopyData = CopyRoutine
(*CopyRoutine)(0)

函数 CopyRoutine	Length=0 获得用户缓冲区长度+返回空指针 pInformation->Ctrl_Info.Usb_wLength =	Length!=0 返回用户缓冲区指针
(A) Mass_GetDevDes 即Std_GetDes(,&DevDes)	18 - pInformation->Ctrl_Info.Usb_wOffset Return 0	Return Device_Descriptor – pInformation->Ctrl_Info.Usb_wOffset
(B) Mass_GetConfDes 即Std_GetDes(,&ConfDes)	32 - pInformation->Ctrl_Info.Usb_wOffset Return 0	Return Config_Descriptor – pInformation->Ctrl_Info.Usb_wOffset
(C) Mass_GetStringDes 即Std_GetDec(,&StrDes[X])	Y - pInformation->Ctrl_Info.Usb_wOffset Return 0	Return String_Descriptor[x] – pInformation->Ctrl_Info.Usb_wOffset
(E) Std_GetConf()	sizeof(pInformation->Current_Configuration) Return 0	return (uint8_t *)&pInformation-> Current_Configuration
(F) Std_GetIntf()	sizeof(pInformation>Current_AlternateSetting) Return 0	Return (uint8_t *)&pInformation->Current_AlternateSetting
(G) Get_Max_LUN()	LUN_DATA_LENGTH (即1) Return 0	return((uint8_t*)(&Max_Lun))
	调用形式: (*CopyRoutine)(0);	调用形式: Buffer = (*pEPinfo->CopyData)(Length); DataBuffer = (*pEPinfo->CopyData)(Length);

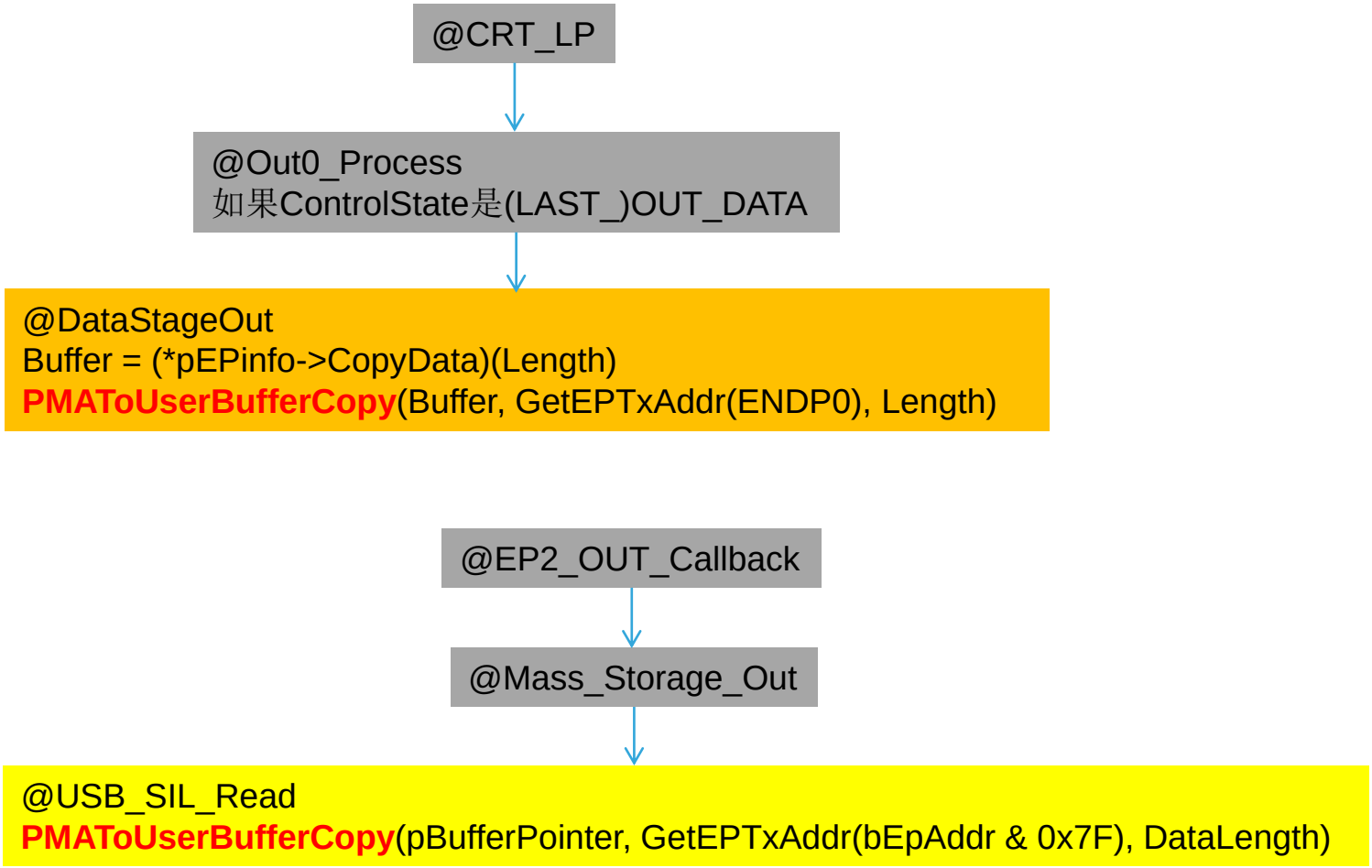
@ Data_Setup0：根据收到的主机Request，给CopyRoutine赋值，然后通过调用(*CopyRoutine)(0)来把buffer长度赋值给pInformation->Ctrl_Info.Usb_wLength。然后根据数据阶段方向，当是device->host的情况，就调用DataStageIn()

@ DataStageIn()：调用(*pEPinfo->CopyData(Length))，即调用(pInformation->Ctrl_Info->CopyRoutine)来获得要发送的Length个数据所在的buf位置，赋值给DataBuffer；然后再调用**UserToPMABufferCopy**(DataBuffer, GetEPTxAddr(ENDP0), Length);来把buf中的Length个数据拷贝到硬件buf中，伺机发出去

UserToPMABufferCopy



PMAToUserBufferCopy



- 库里只统一处理了标准request，其中对CopyRoutine做了赋值；对于类相关的request，是通过(*pProperty->Class_Data_Setup)来由用户自定义的

```
@Data_Setup0 @《usb_core.c》
……
If (CopyRoutine)   {……}
Else
   {  Result = (*pProperty->Class_Data_Setup)(pInformation->USBbRequest); ……..}
```

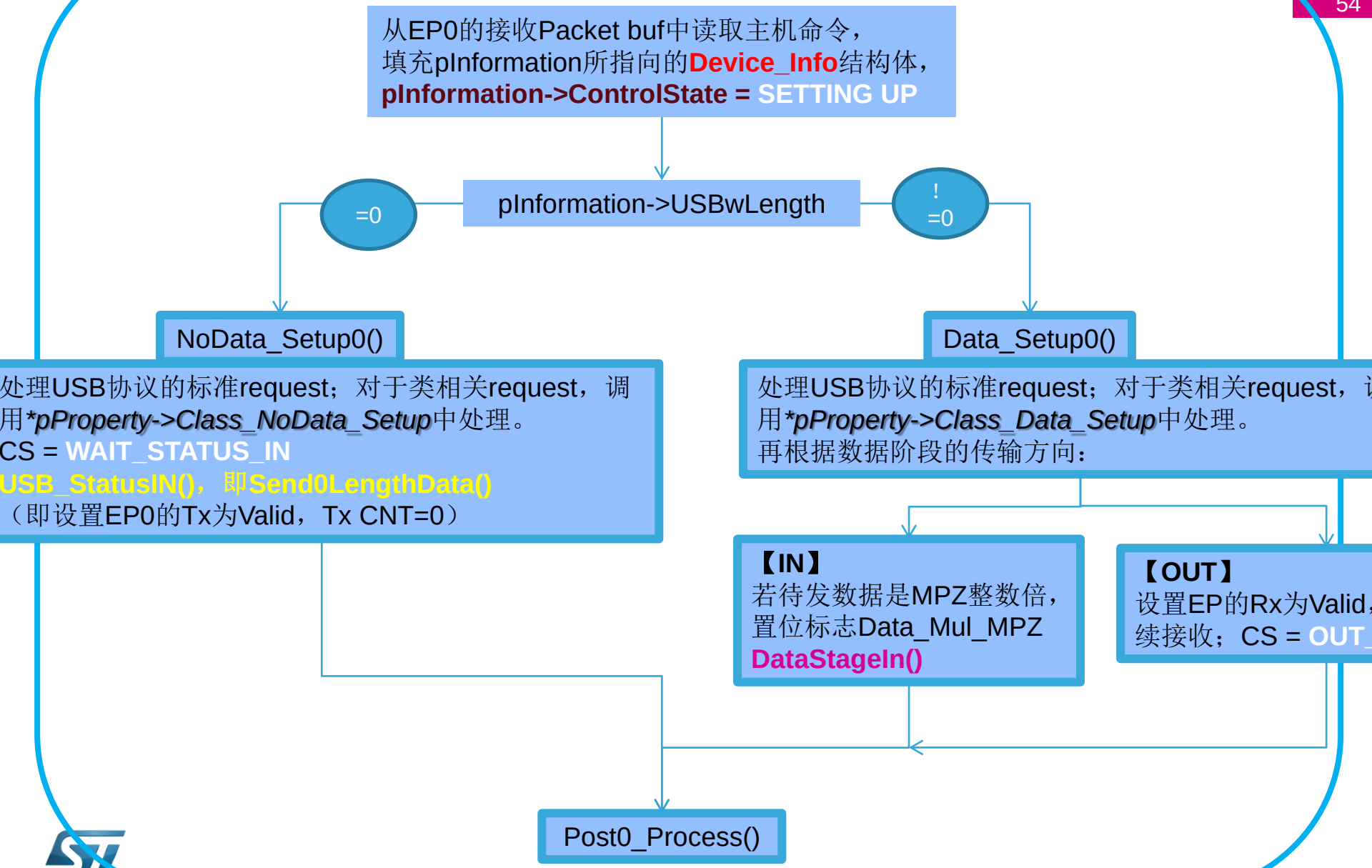
```
@MASS_Data_Setup @ 《usb_prop.c》

 uint8_t    *(*CopyRoutine)(uint16_t);  CopyRoutine = NULL;

  if ((Type_Recipient == (CLASS_REQUEST |
INTERFACE_RECIPIENT))
      && (RequestNo == GET_MAX_LUN) && (pInformation-
>USBwValue == 0)
      && (pInformation->USBwIndex == 0) && (pInformation-
>USBwLength == 0x01))
  {
    CopyRoutine = Get_Max_Lun;
  }
  else
  {
    return USB_UNSUP
  }

  if (CopyRoutine == N
  {
    return USB_UNSUP
  }

  pInformation->Ctrl_In
  pInformation->Ctrl_In
  (*CopyRoutine)(0);

  return USB_SUCCESS;
```

```
uint8_t *Get_Max_Lun(uint16_t Length)
{
  if (Length == 0)
  {
    pInformation->Ctrl_Info.Usb_wLength = 1;
    return 0;
  }
  else
  {
    return((uint8_t*)(&Max_Lun));
  }
}
```

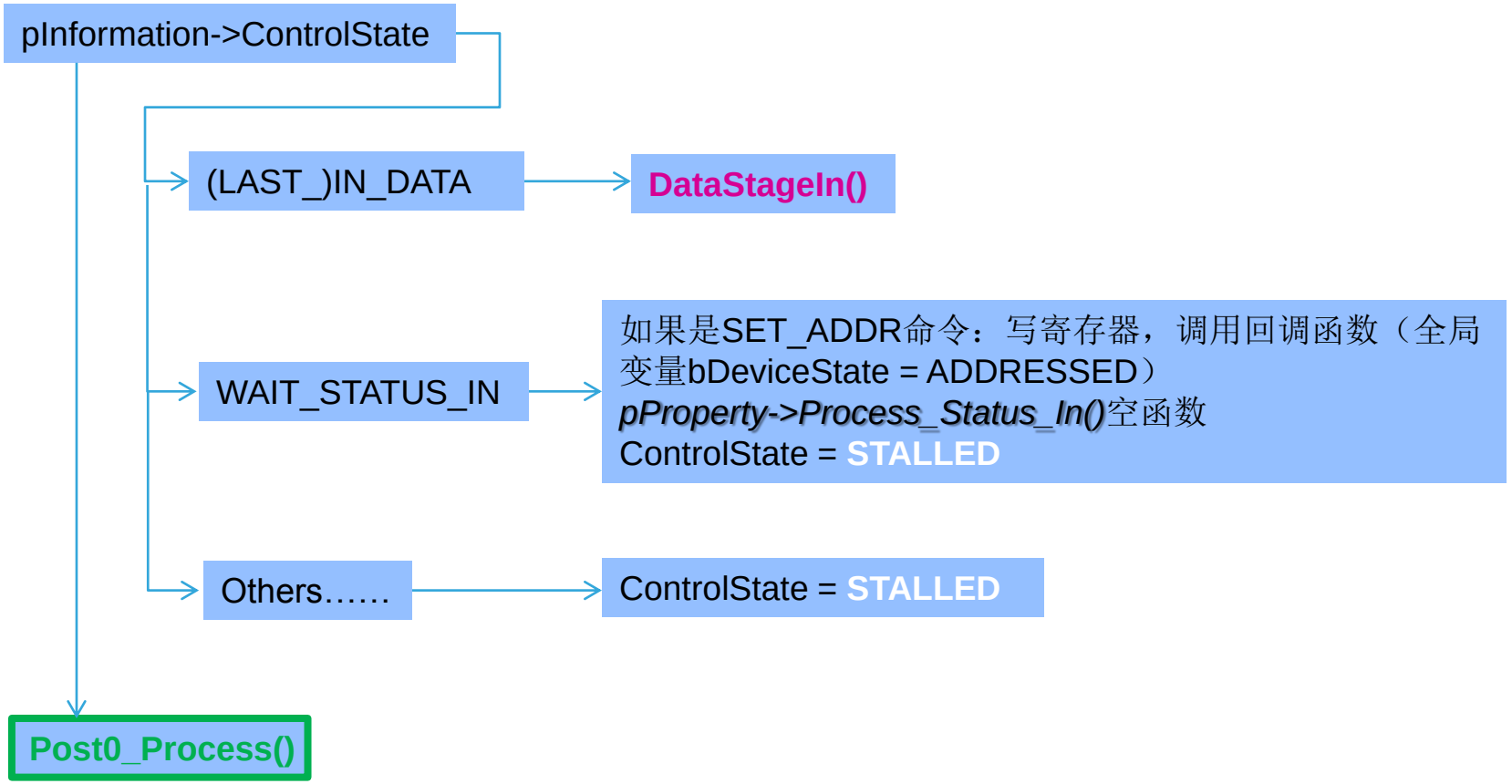
```
@Joystick_Data_Setup @ 《usb_prop.c》

 uint8_t    *(*CopyRoutine)(uint16_t);  CopyRoutine = NULL;

  if ((RequestNo == GET_DESCRIPTOR)
     && (Type_Recipient == (STANDARD_REQUEST |
INTERFACE_RECIPIENT))
     && (pInformation->USBwIndex0 == 0))
  {
    if (pInformation->USBwValue1 == REPORT_DESCRIPTOR)
    {
      CopyRoutine = Joystick_GetReportDescriptor;
    }
    else if (pInformation->USBwValue1 == HID_DESCRIPTOR_TYPE)
    {
      CopyRoutine = Joystick_GetHIDDescriptor;
    }
  } else if ((Type_Recipient == (CLASS_REQUEST |
INTERFACE_RECIPIENT))
          && RequestNo == GET_PROTOCOL)
  {
    CopyRoutine = Joystick_GetProtocolValue;
  }


  if (CopyRoutine == NULL)
  {
    return USB_UNSUPPORT;
  }
```

```
uint8_t *Joystick_GetProtocolValue(uint
{
  if (Length == 0)
  {
    pInformation->Ctrl_Info.Usb_wLength
    return NULL;
  }
```

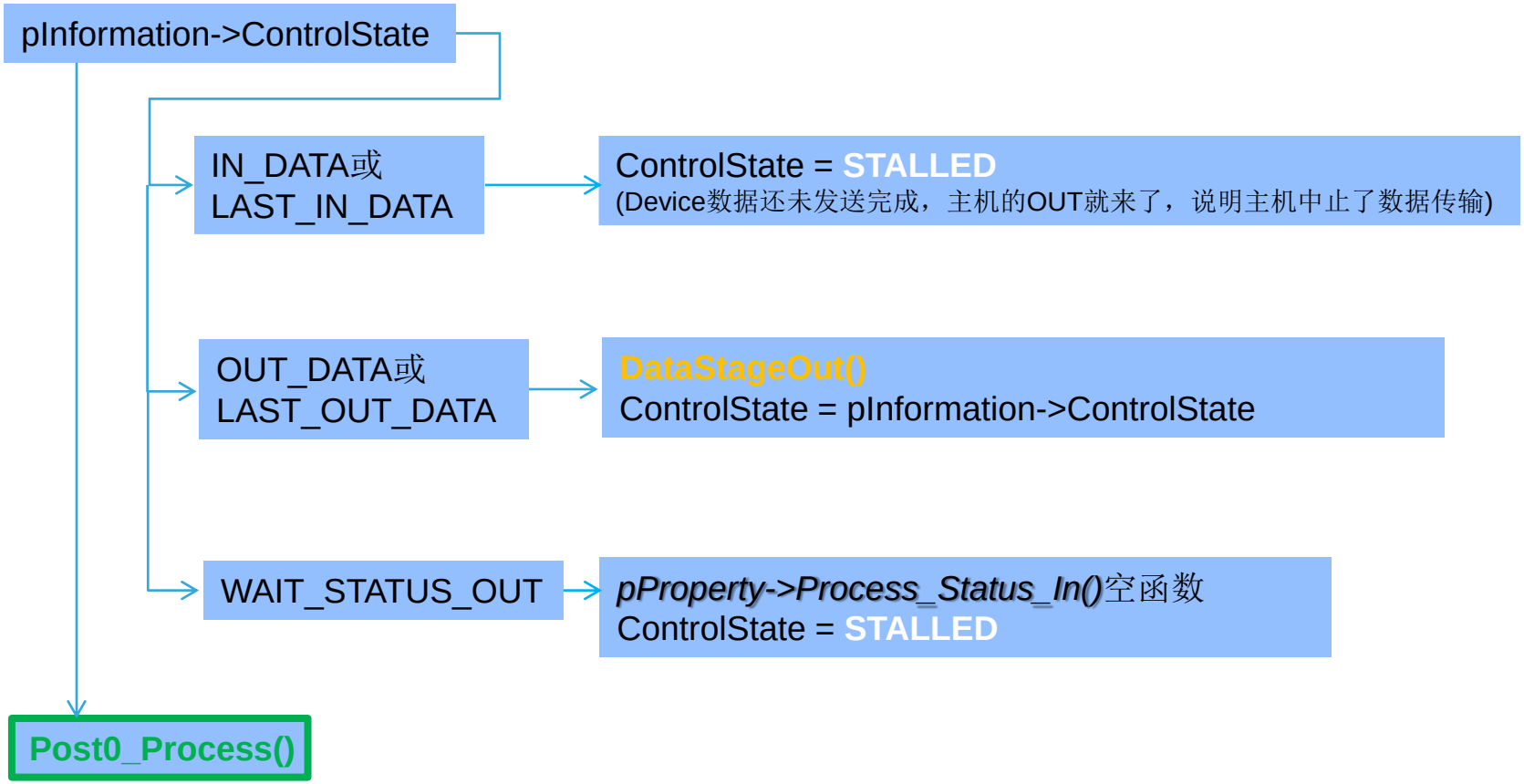
Setup0_Process()



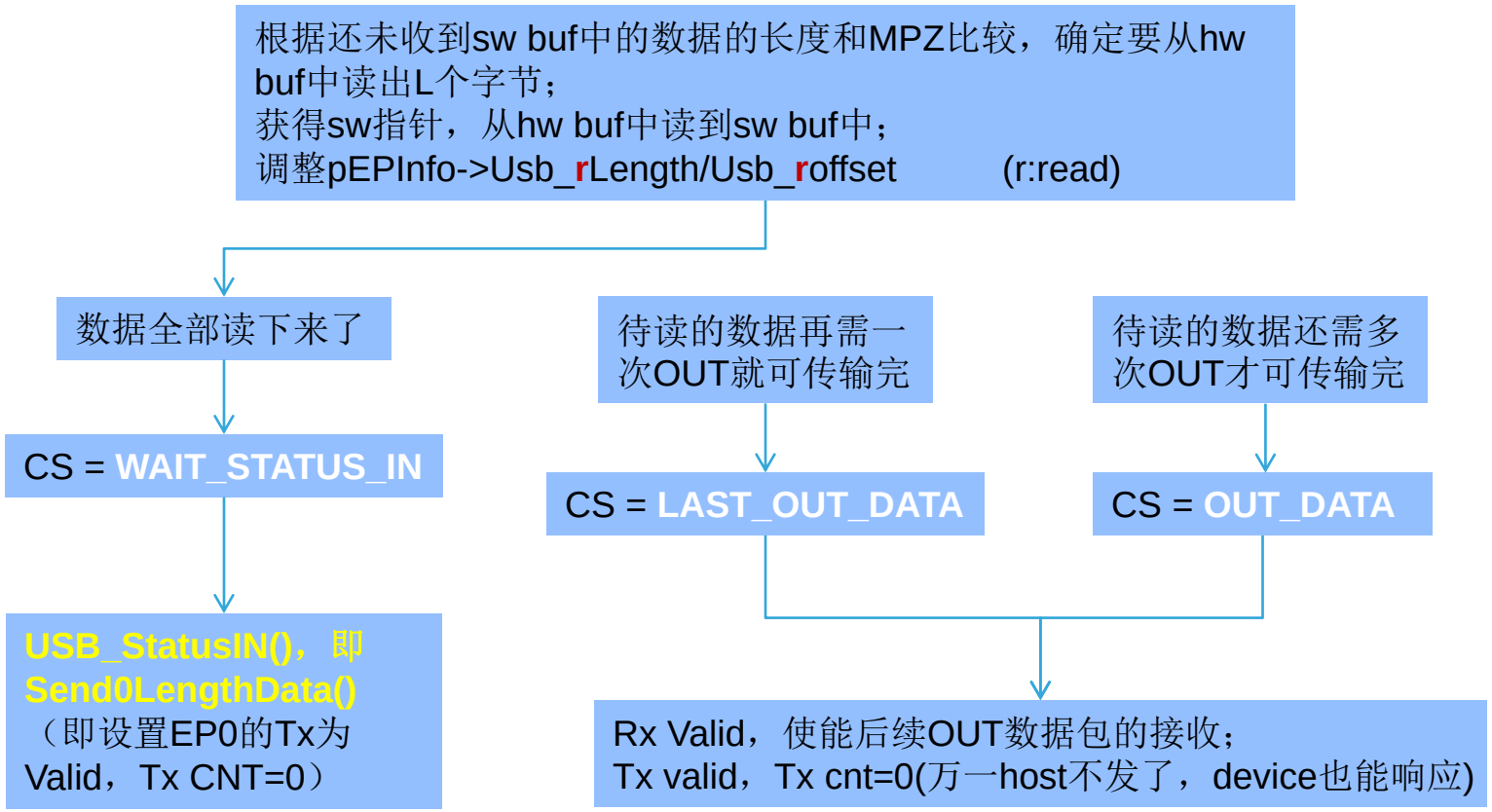
In0_Process()


pInformation->ControlState
(LAST_)IN_DATA
DataStageIn()
WAIT_STATUS_IN
如果是SET_ADDR命令：写寄存器，调用回调函数（全局变量bDeviceState = ADDRESSED）
pProperty->Process_Status_In()空函数
ControlState = STALLED
Others……
ControlState = STALLED
Post0_Process()

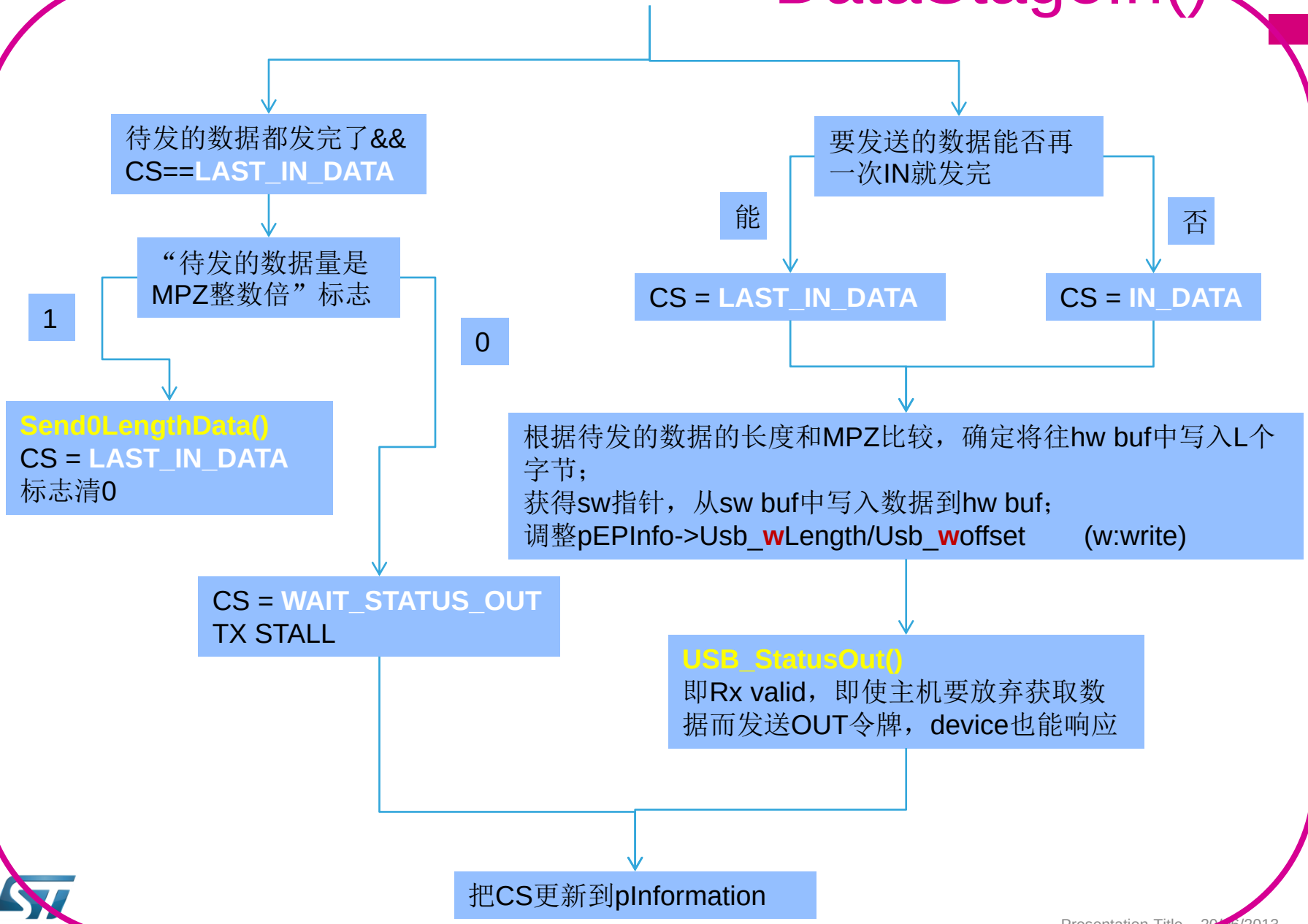
Out0_Process()



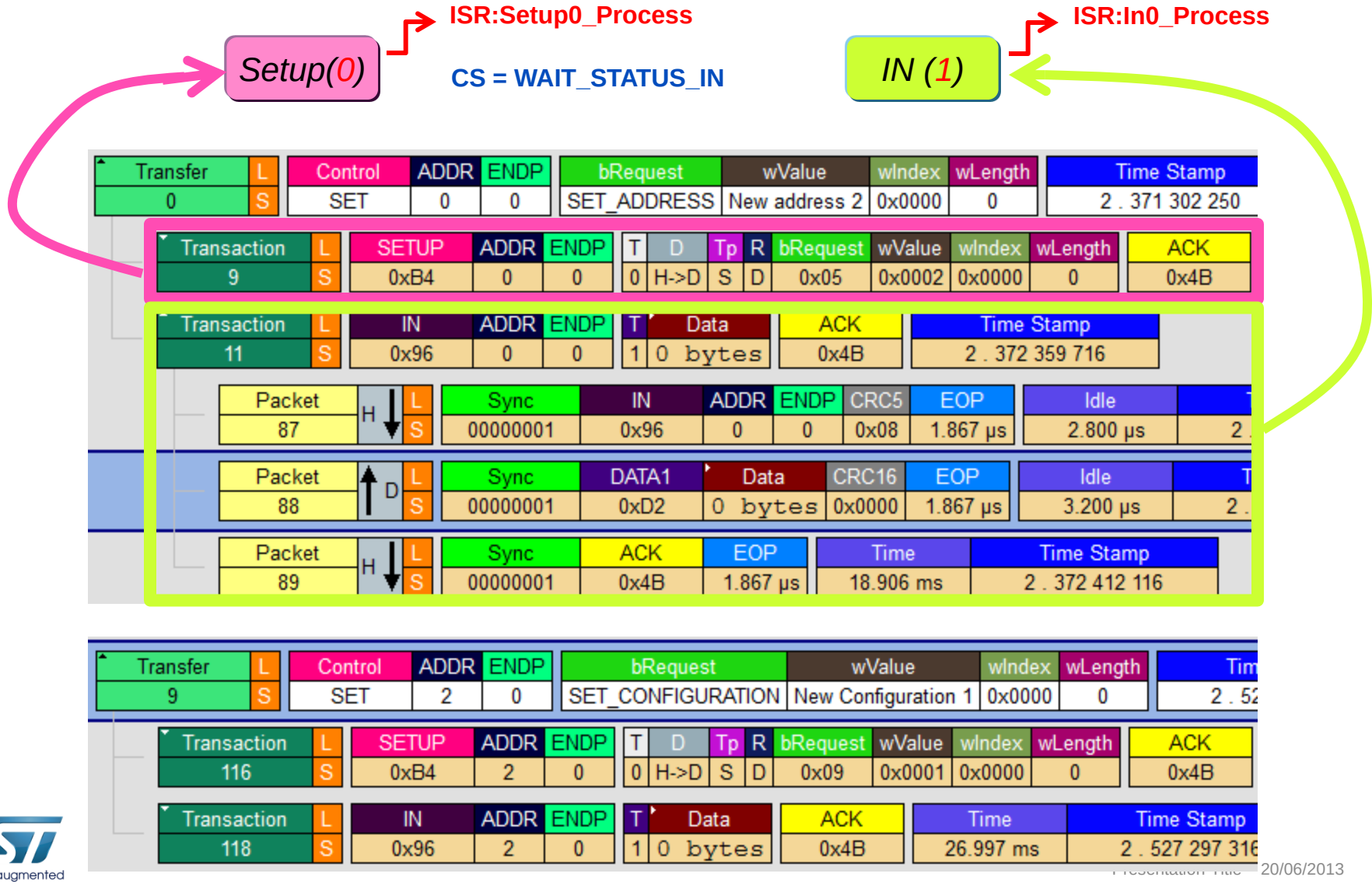
DataStageOut()



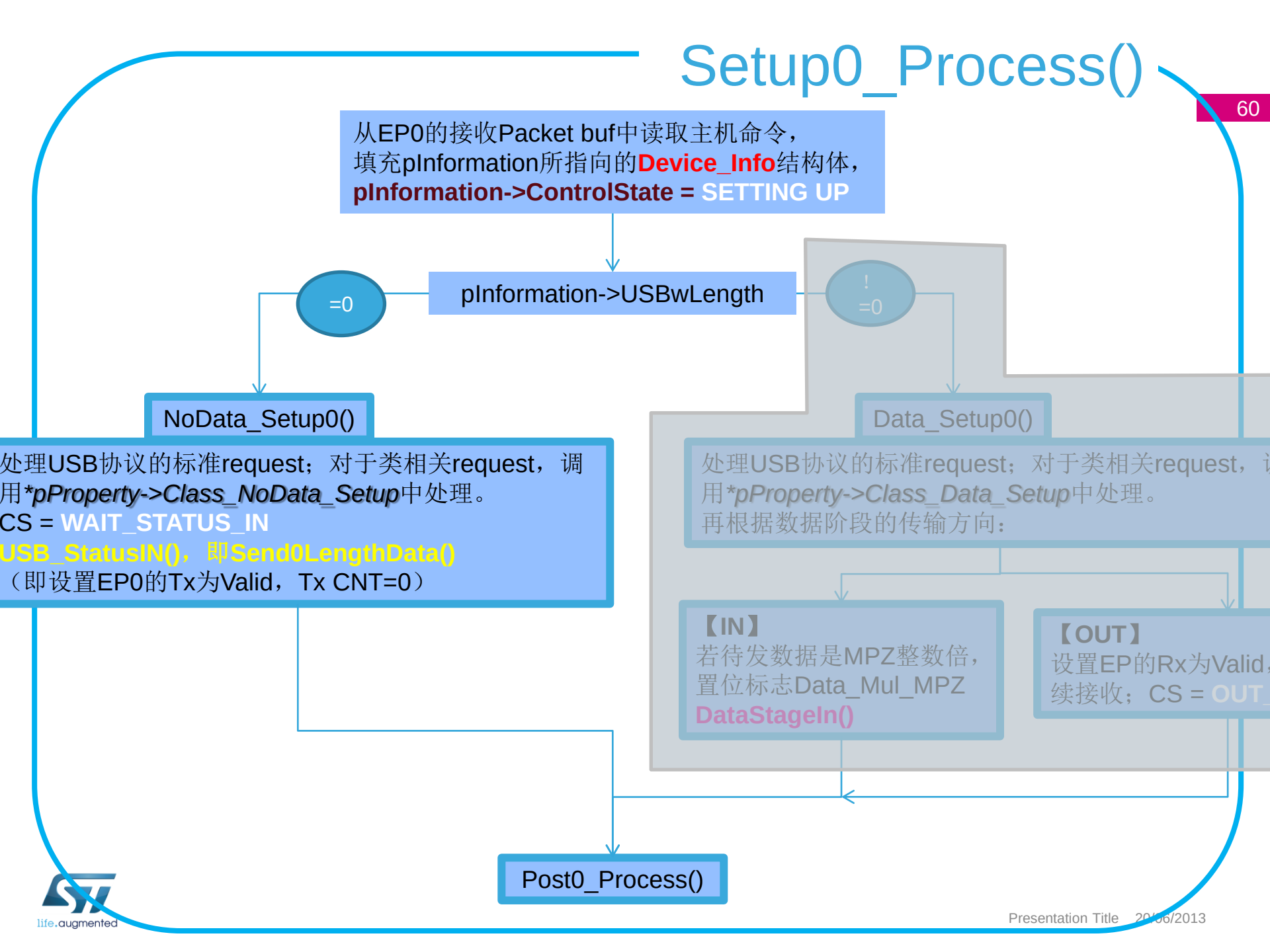
DataStageIn()



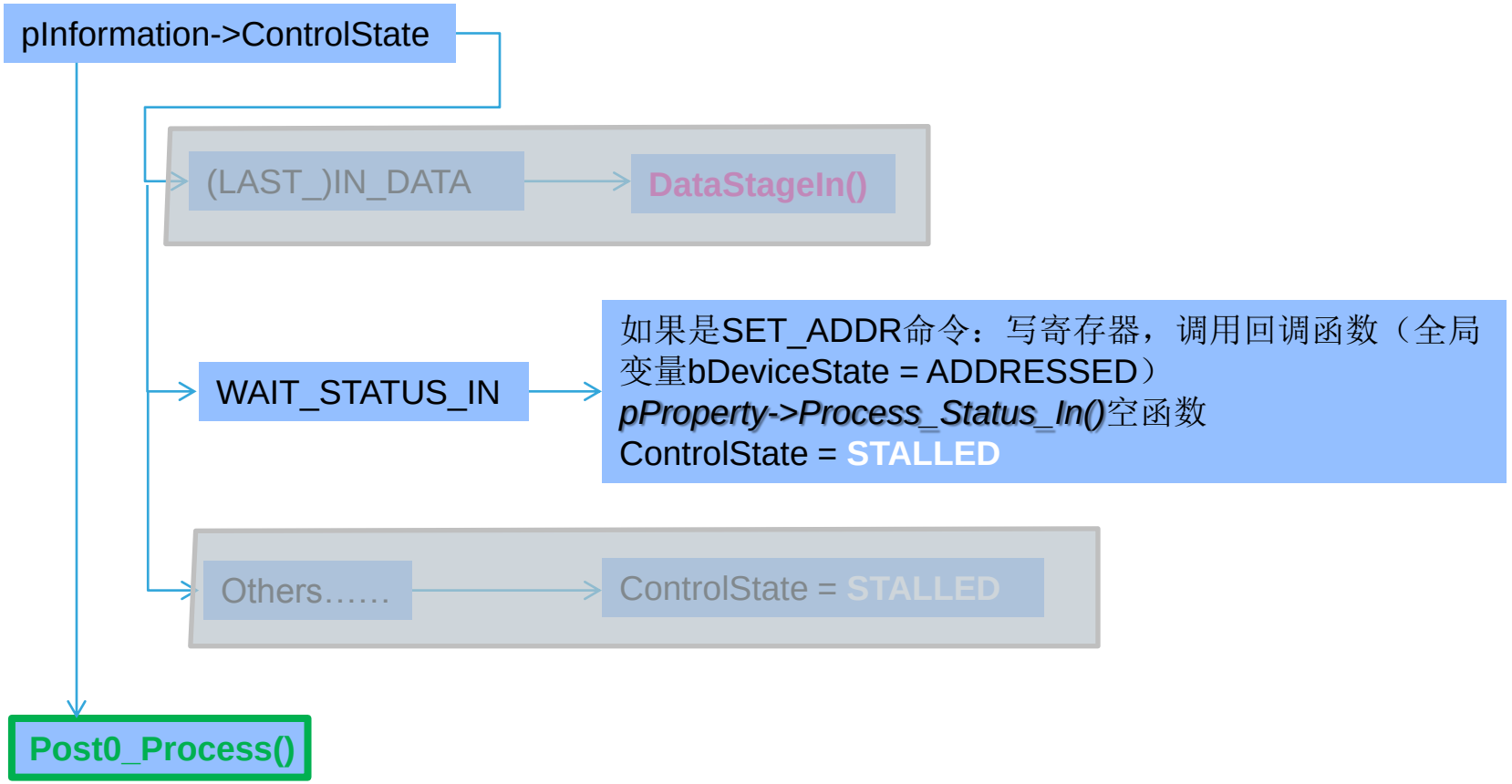
无数据控制传输在库函数中的流程



Setup0_Process()



In0_Process()


pInformation->ControlState
(LAST_)IN_DATA
DataStageIn()
WAIT_STATUS_IN
如果是SET_ADDR命令：写寄存器，调用回调函数（全局变量bDeviceState = ADDRESSED）
pProperty->Process_Status_In()空函数
ControlState = STALLED
Others……
ControlState = STALLED
Post0_Process()

读数据控制传输在库函数中的流程

Setup(0) → ISR: Setup0_Process -> DataStageIN

CS = IN_DATA

In (1) → ISR: In0_Process -> DataStageIN

CS = LAST_IN_DATA

In (0) → ISR: In0_Process -> DataStageIN

CS = WAIT_STATUS_OUT

Out (1) → ISR: OUT0_Process

CS = STALL

Transfer	L	Control	ADDR	ENDP	bRequest	wValue	wIndex	Descriptors
2	S	GET	2	0	GET_DESCRIPTOR	CONFIGURATION type, Index 0	0x0000	CONFIGURATION Descriptor

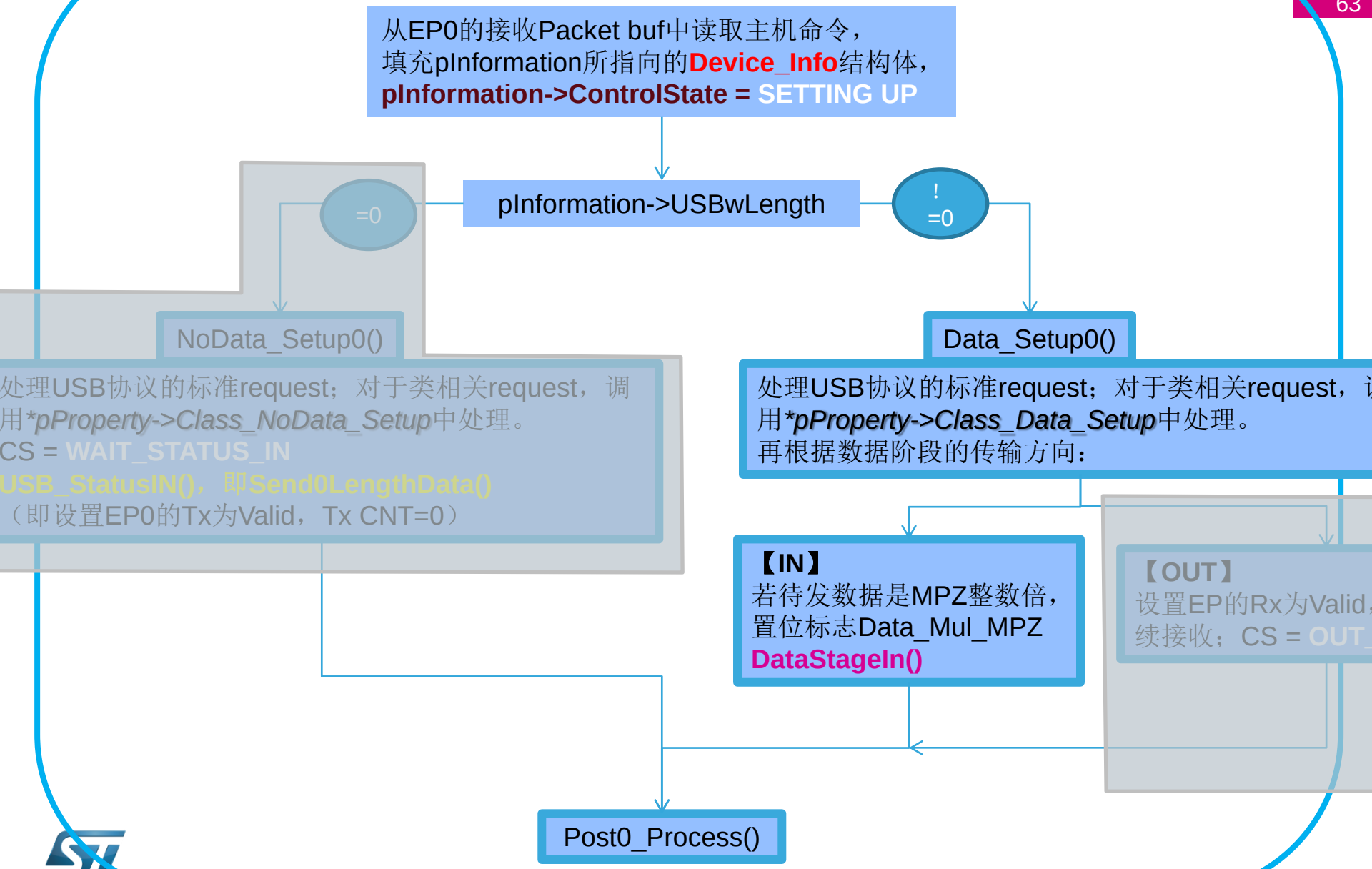
Transaction	L	SETUP	ADDR	ENDP	T	D	Tp	R	bRequest	wValue	wIndex	wLength	ACK	Time
24	S	0xB4	2	0	0	D->H	S	D	0x06	0x0200	0x0000	9	0x4B	1.981 ms

Transaction	L	IN	ADDR	ENDP	T	Data	ACK	Time	Time Stamp
26	S	0x96	2	0	1	8 bytes	0x4B	3.019 ms	2 . 402 300 250

Transaction	L	IN	ADDR	ENDP	T	Data	ACK	Time	Time Stamp
30	S	0x96	2	0	0	1 byte	0x4B	998.800 µs	2 . 405 319 716

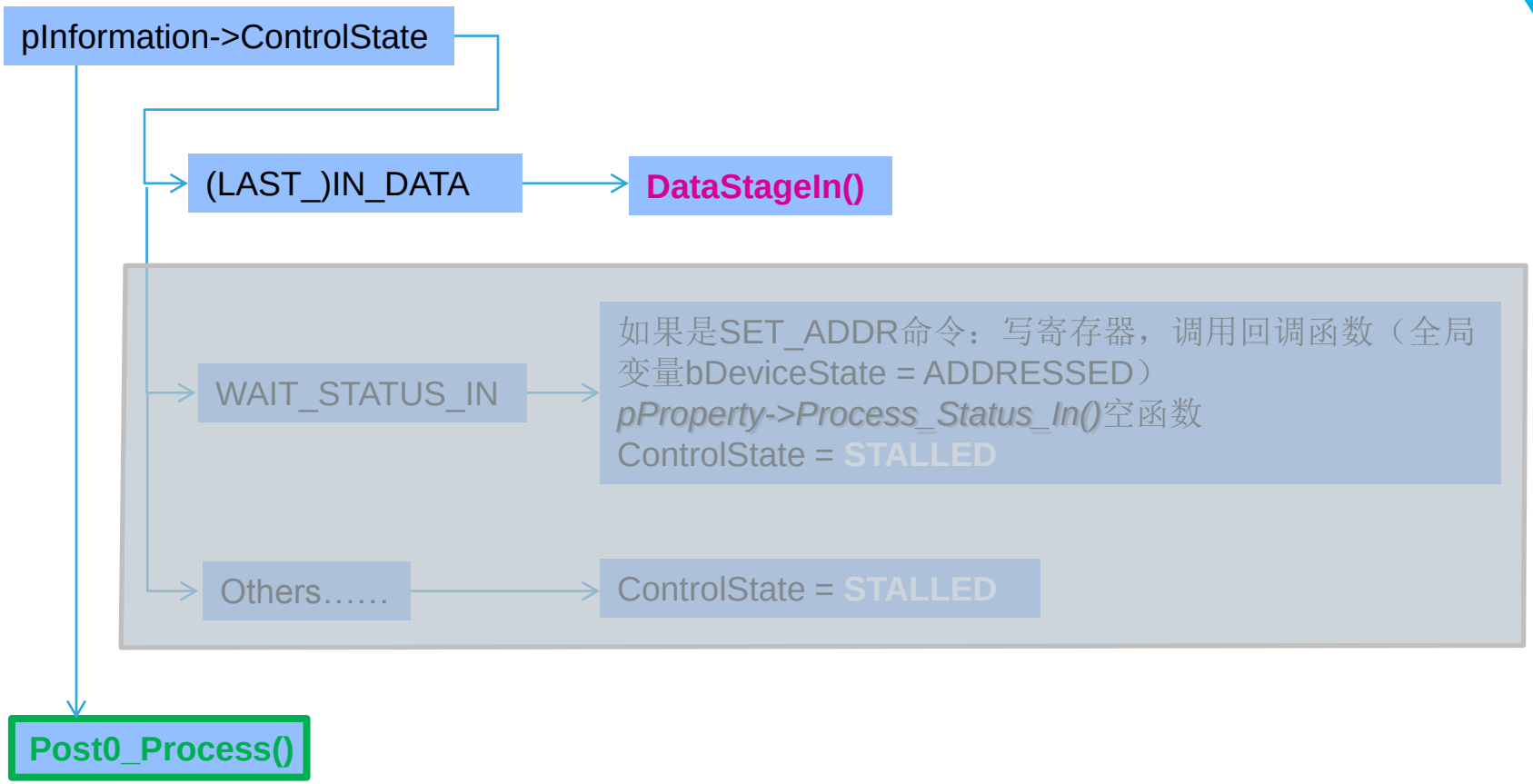
Transaction	L	OUT	ADDR	ENDP	T	Data	ACK	Time	Time Stamp
31	S	0x87	2	0	1	0 bytes	0x4B	999.600 µs	2 . 406 318 516

Setup0_Process()

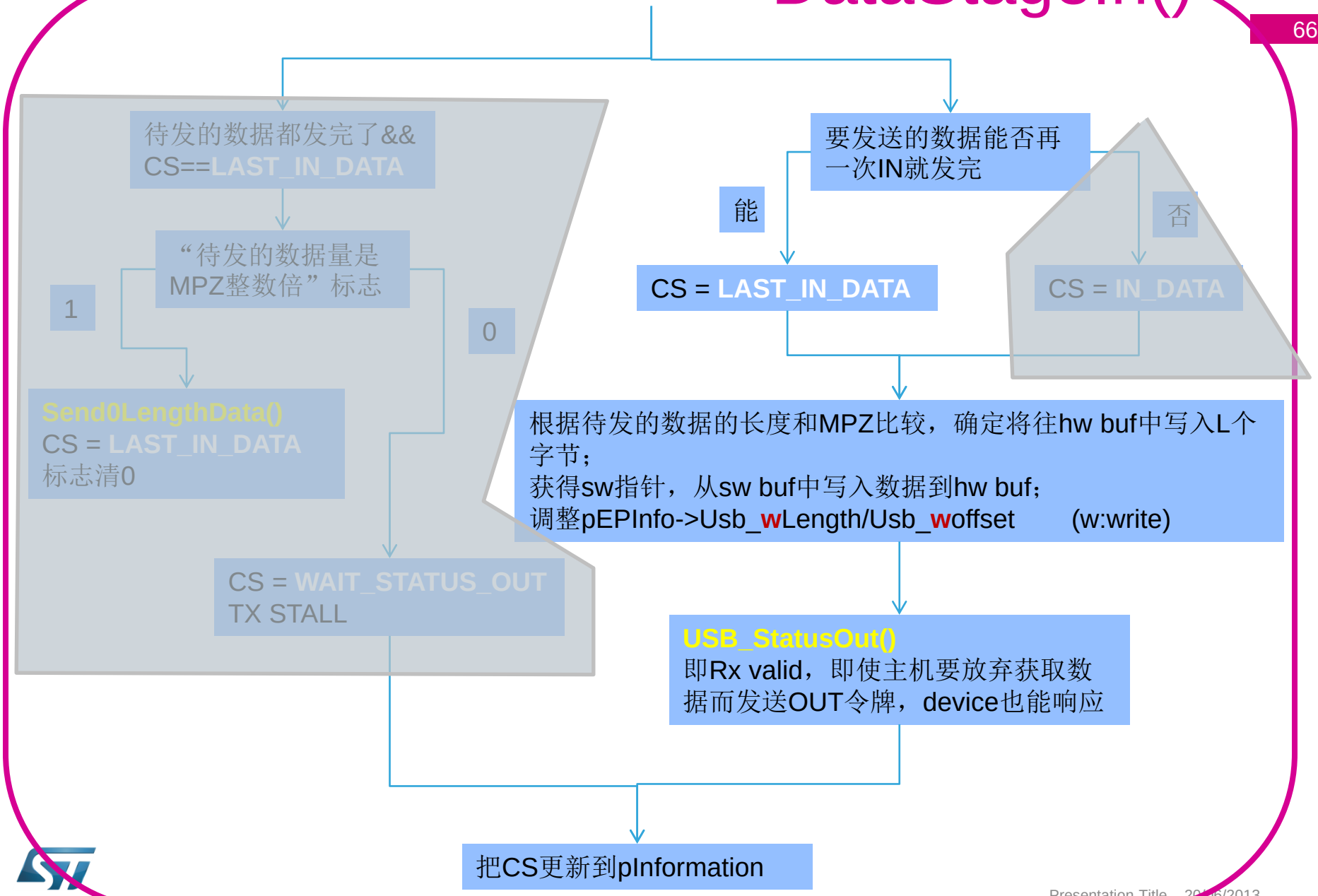


DataStageIn()

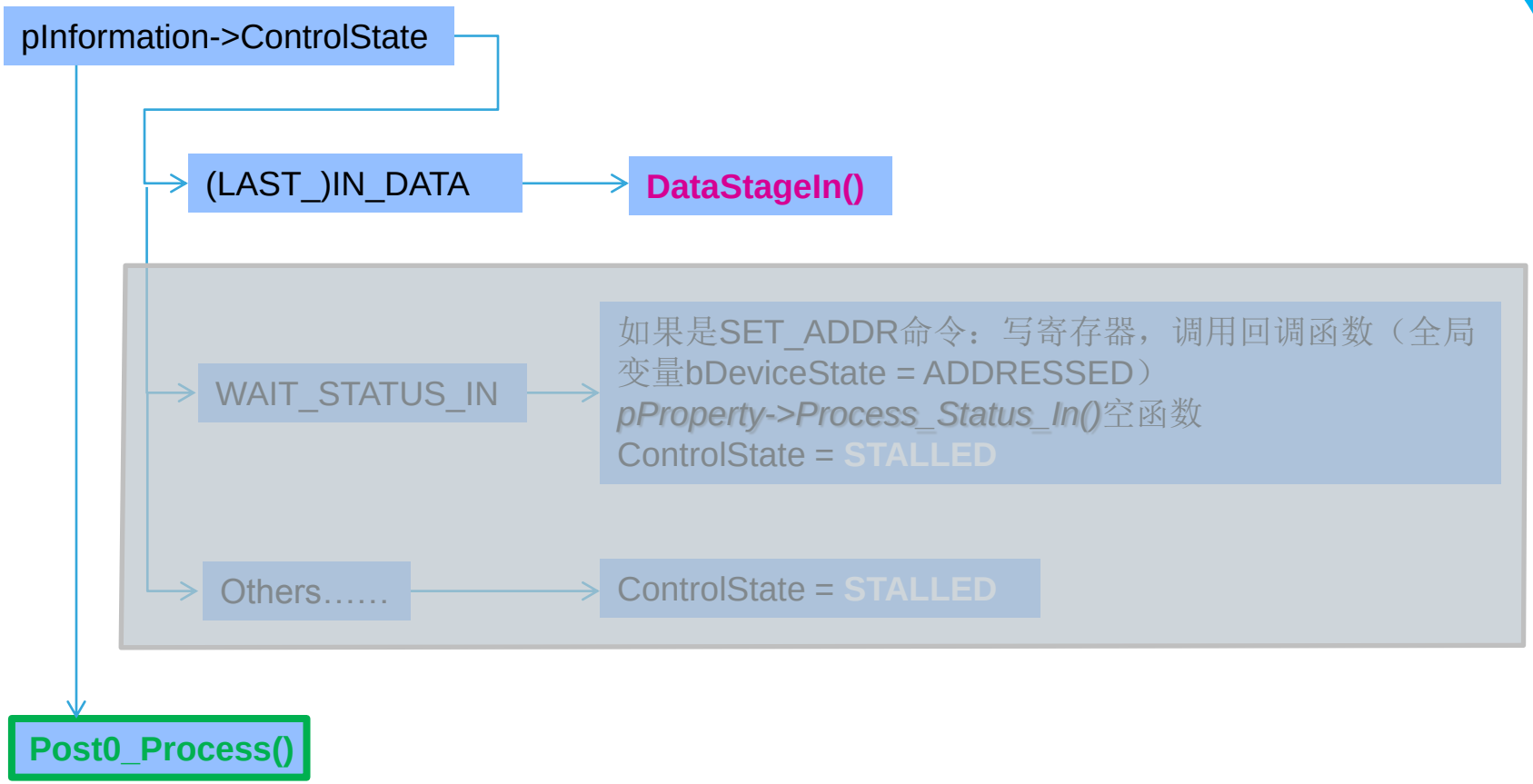
In0_Process()


pInformation->ControlState
(LAST_)IN_DATA
DataStageIn()
WAIT_STATUS_IN
如果是SET_ADDR命令：写寄存器，调用回调函数（全局变量bDeviceState = ADDRESSED）
pProperty->Process_Status_In()空函数
ControlState = STALLED
Others……
ControlState = STALLED
Post0_Process()

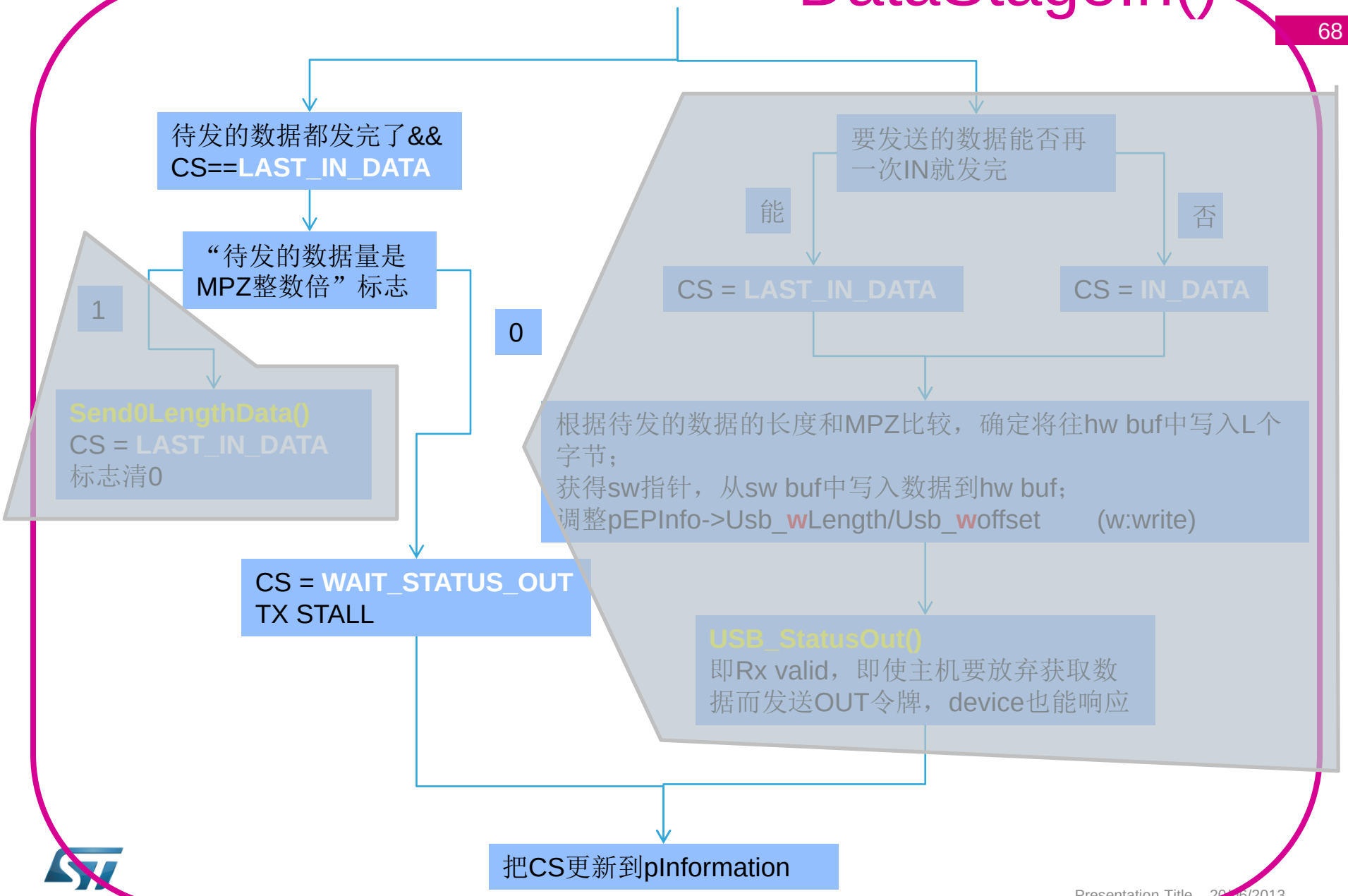
DataStageIn()



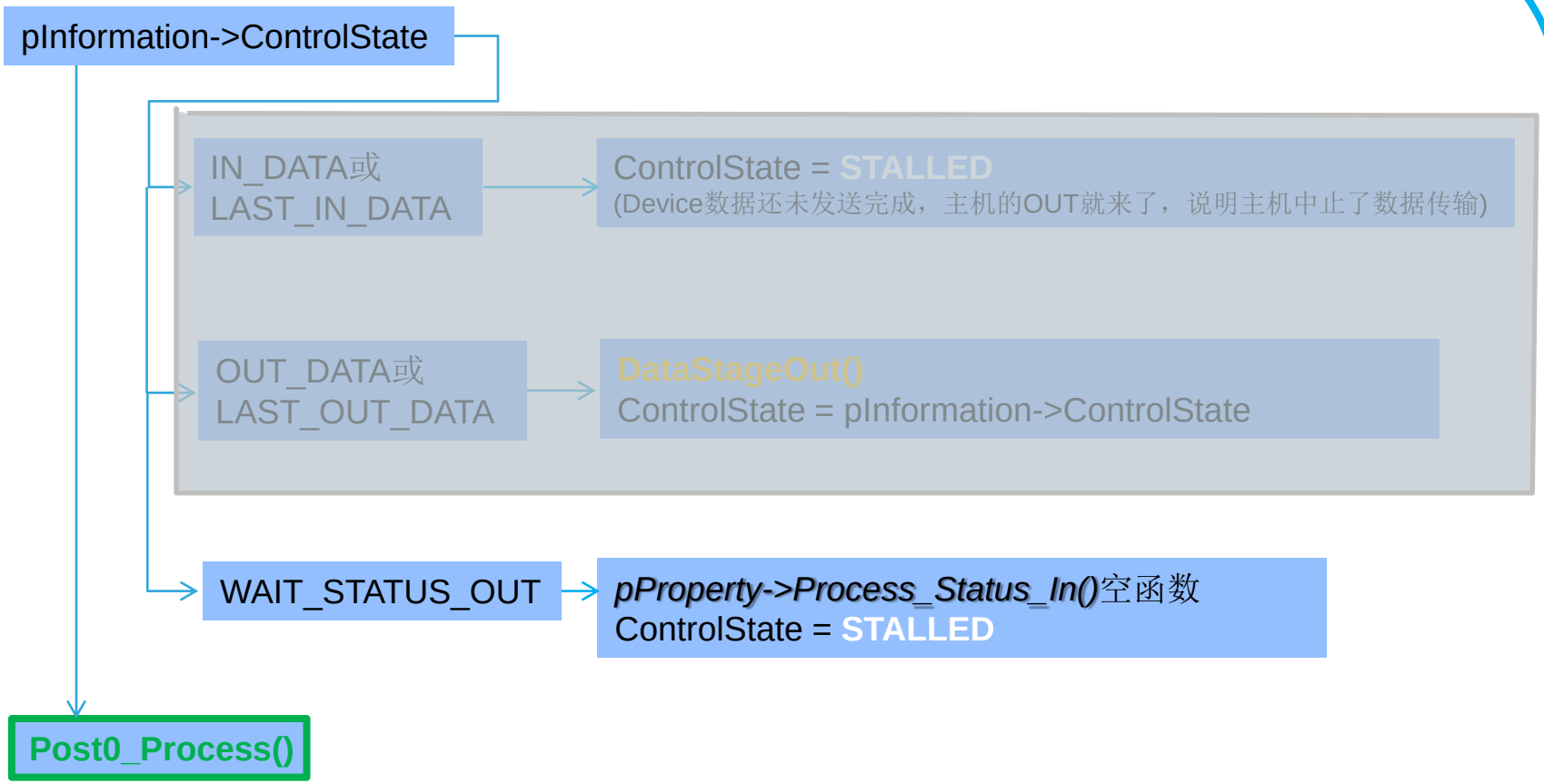
In0_Process()


pInformation->ControlState
(LAST_)IN_DATA
DataStageIn()
WAIT_STATUS_IN
如果是SET_ADDR命令：写寄存器，调用回调函数（全局变量bDeviceState = ADDRESSED）
pProperty->Process_Status_In()空函数
ControlState = STALLED
Others……
ControlState = STALLED
Post0_Process()

DataStageIn()



Out0_Process()



写数据控制传输在库函数中的流程

Setup(0) → ISR: Setup0_Process

CS = OUT_DATA

Out (1) → ISR: Out0_Process -> DataStageOut

CS = WAIT_STATUS_IN

In (1) → ISR: In0_Process

CS = STALL

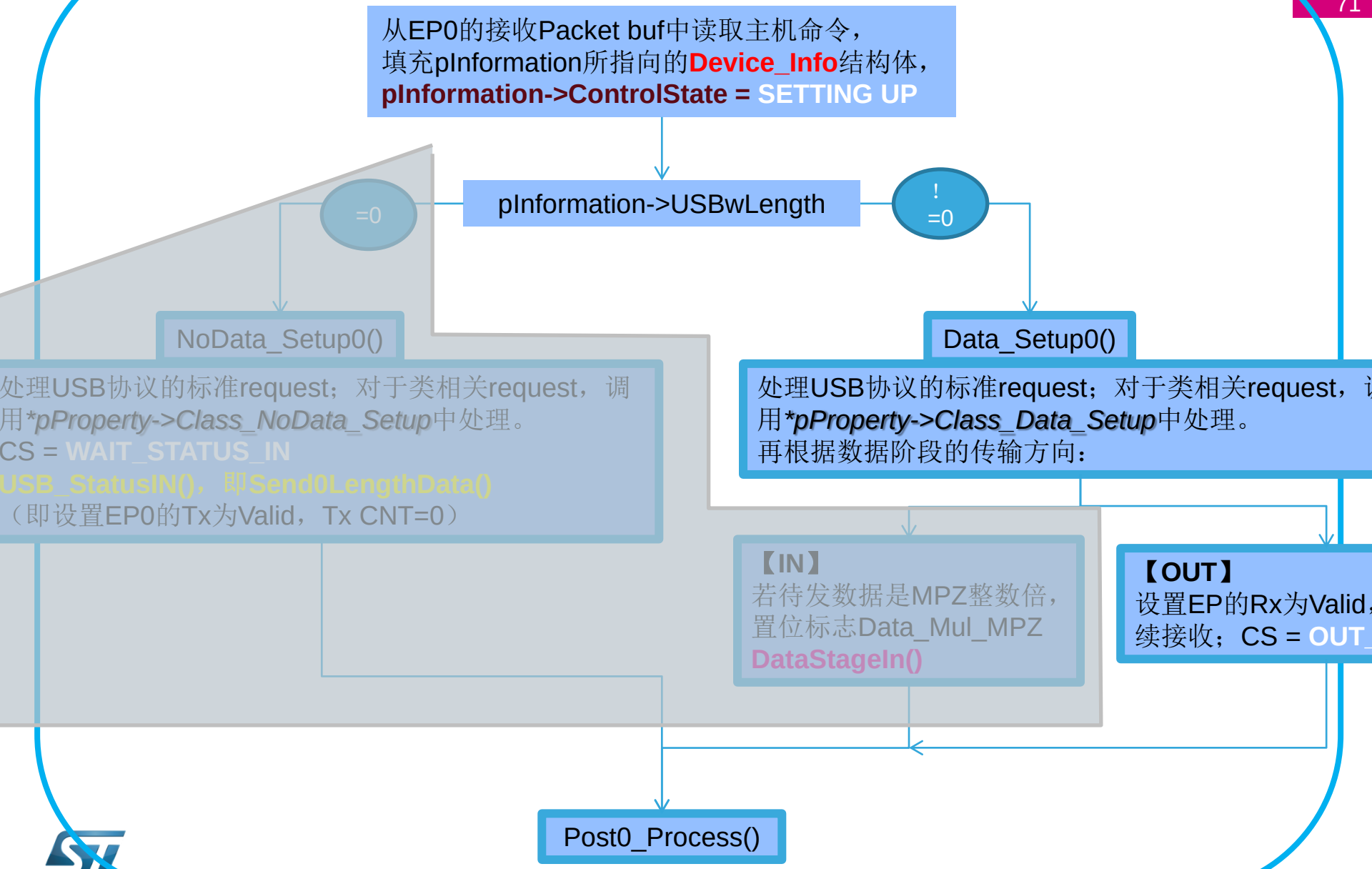
Transfer	L	Control	ADDR	ENDP	D	Tp	R	bRequest	wValue	wIndex	wLength	Bytes Transferred
12	S	SET	2	0	H->D	C	I	0x09	0x0200	0x0000	1	1

Transaction	L	SETUP	ADDR	ENDP	T	D	Tp	R	bRequest	wValue	wIndex	wLength	ACK
459	S	0xB4	2	0	0	H->D	C	I	0x09	0x0200	0x0000	1	0x4B

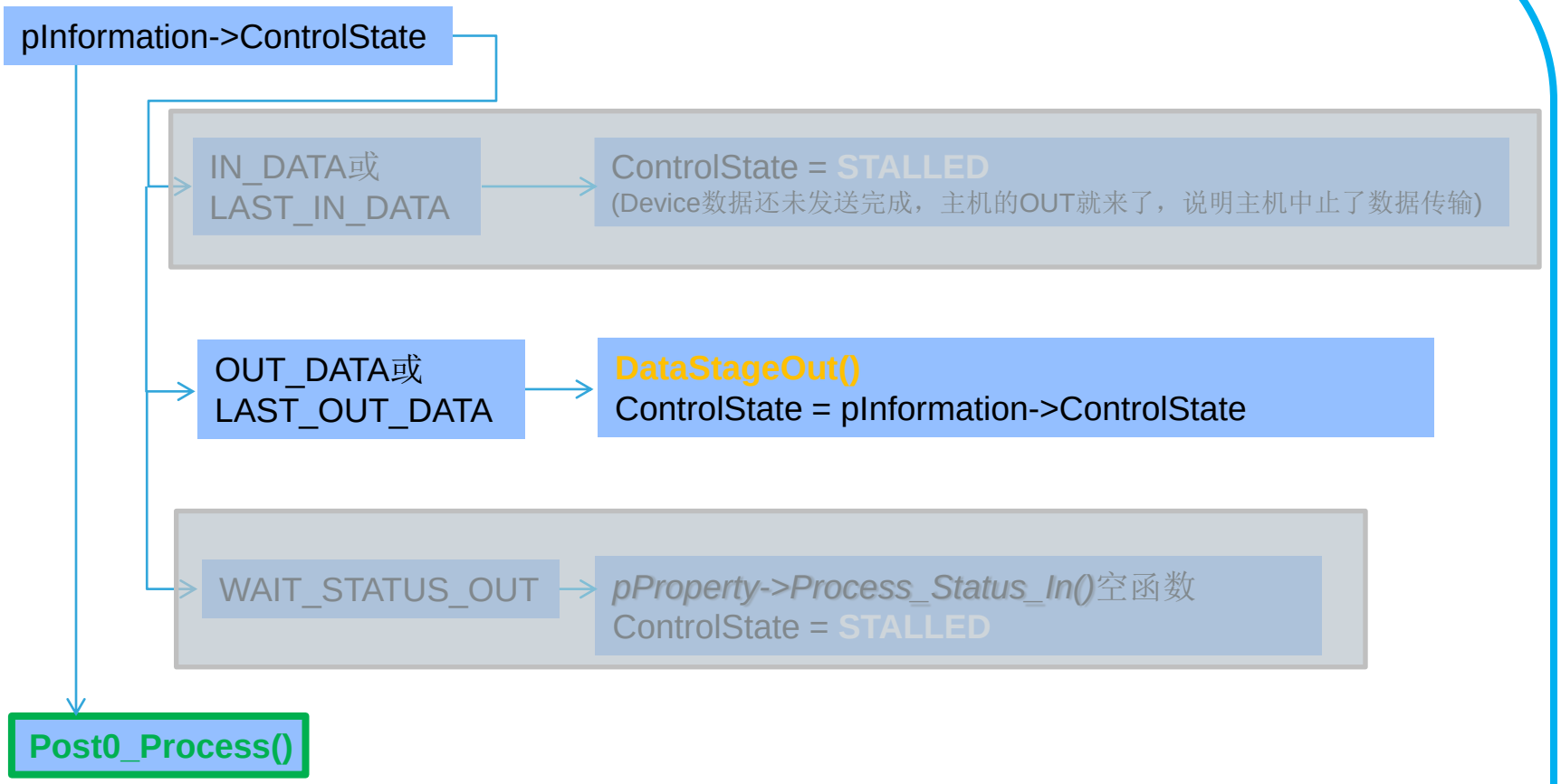
Transaction	L	OUT	ADDR	ENDP	T	Data	ACK	Time	Time Stamp
461	S	0x87	2	0	1	1 byte	0x4B	975.734 µs	4 . 250 208 116

Transaction	L	IN	ADDR	ENDP	T	Data	ACK	Time Stamp
462	S	0x96	2	0	1	0 bytes	0x4B	4 . 251 183 850

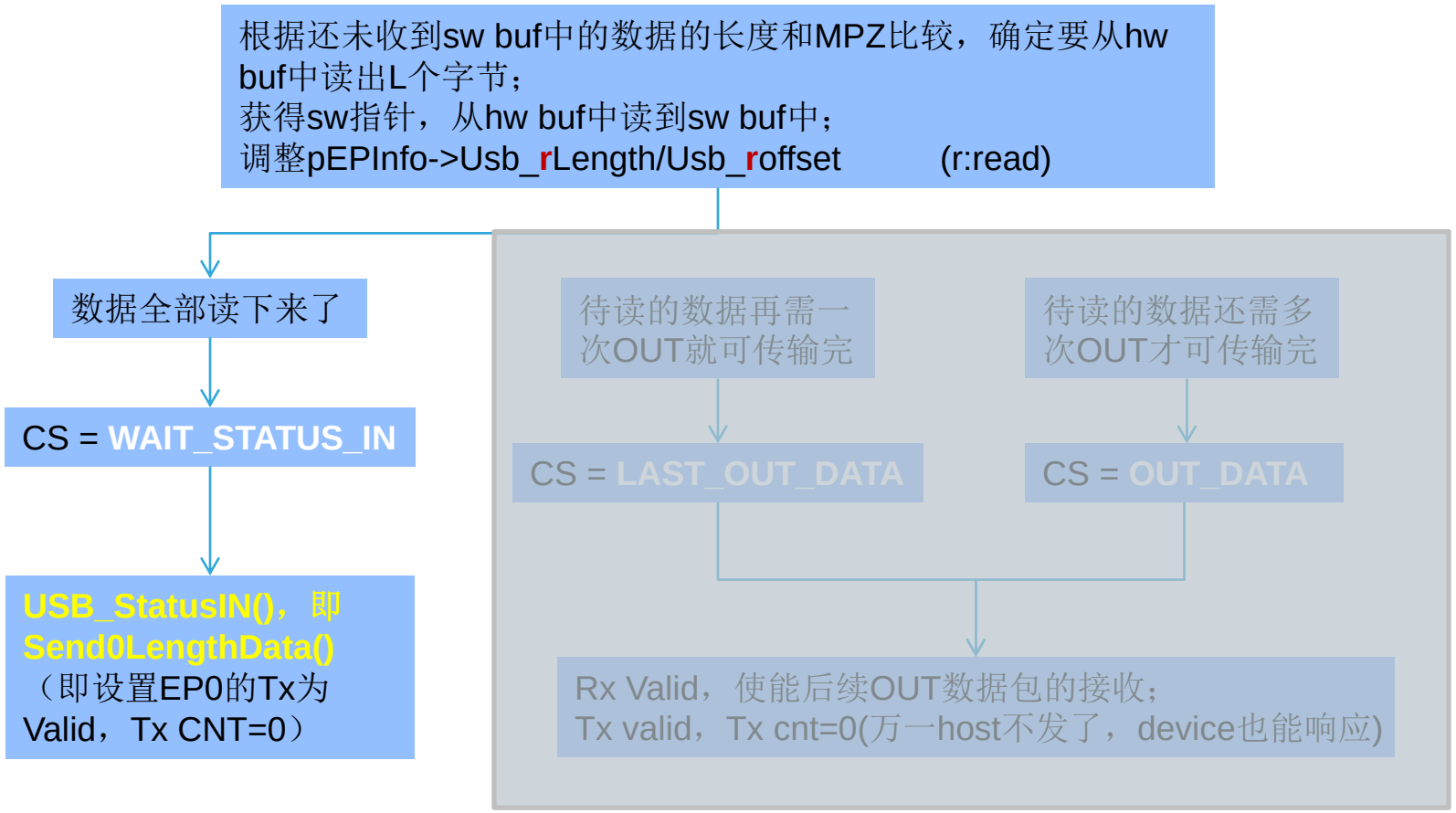
Setup0_Process()



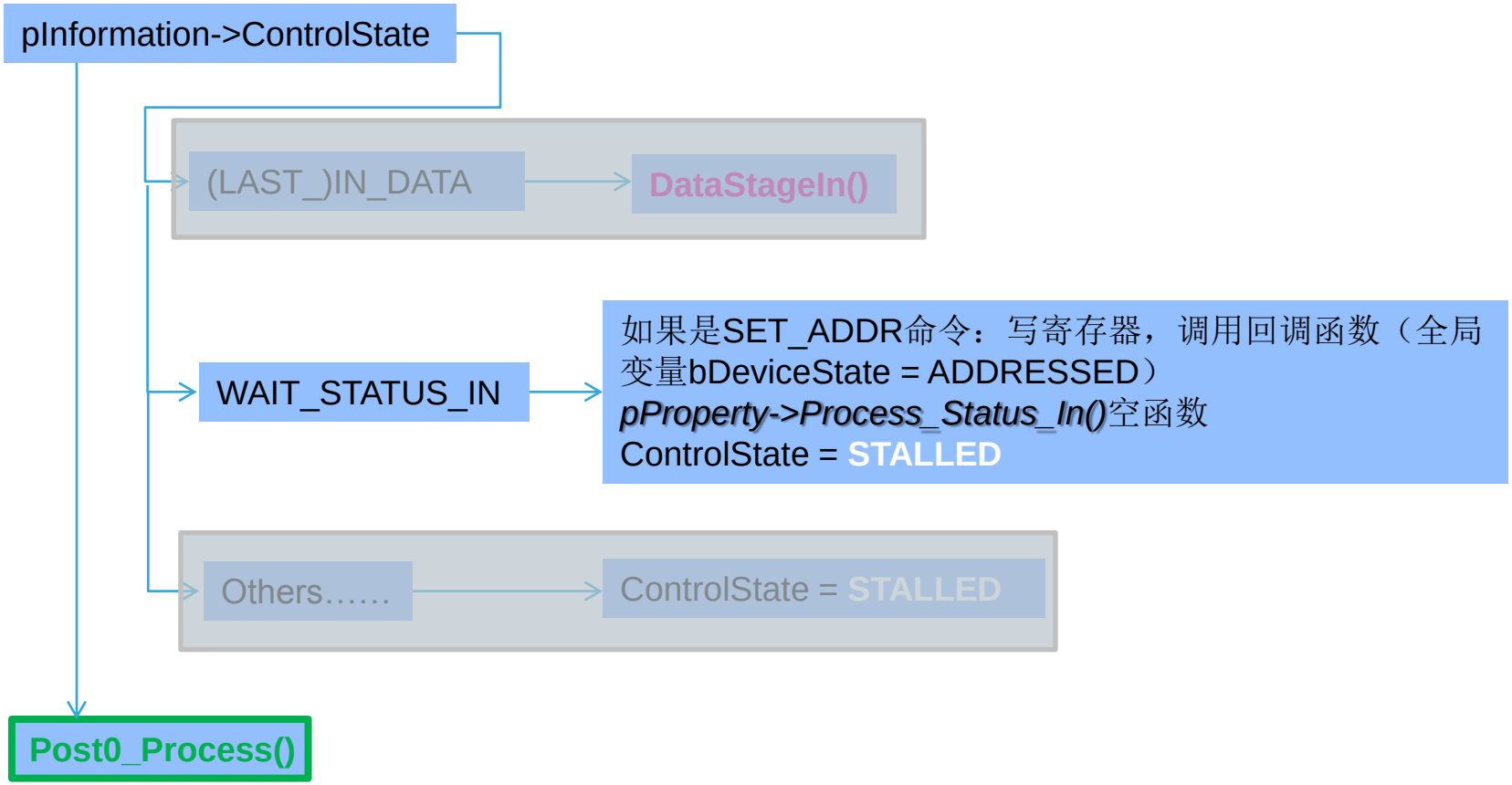
Out0_Process()



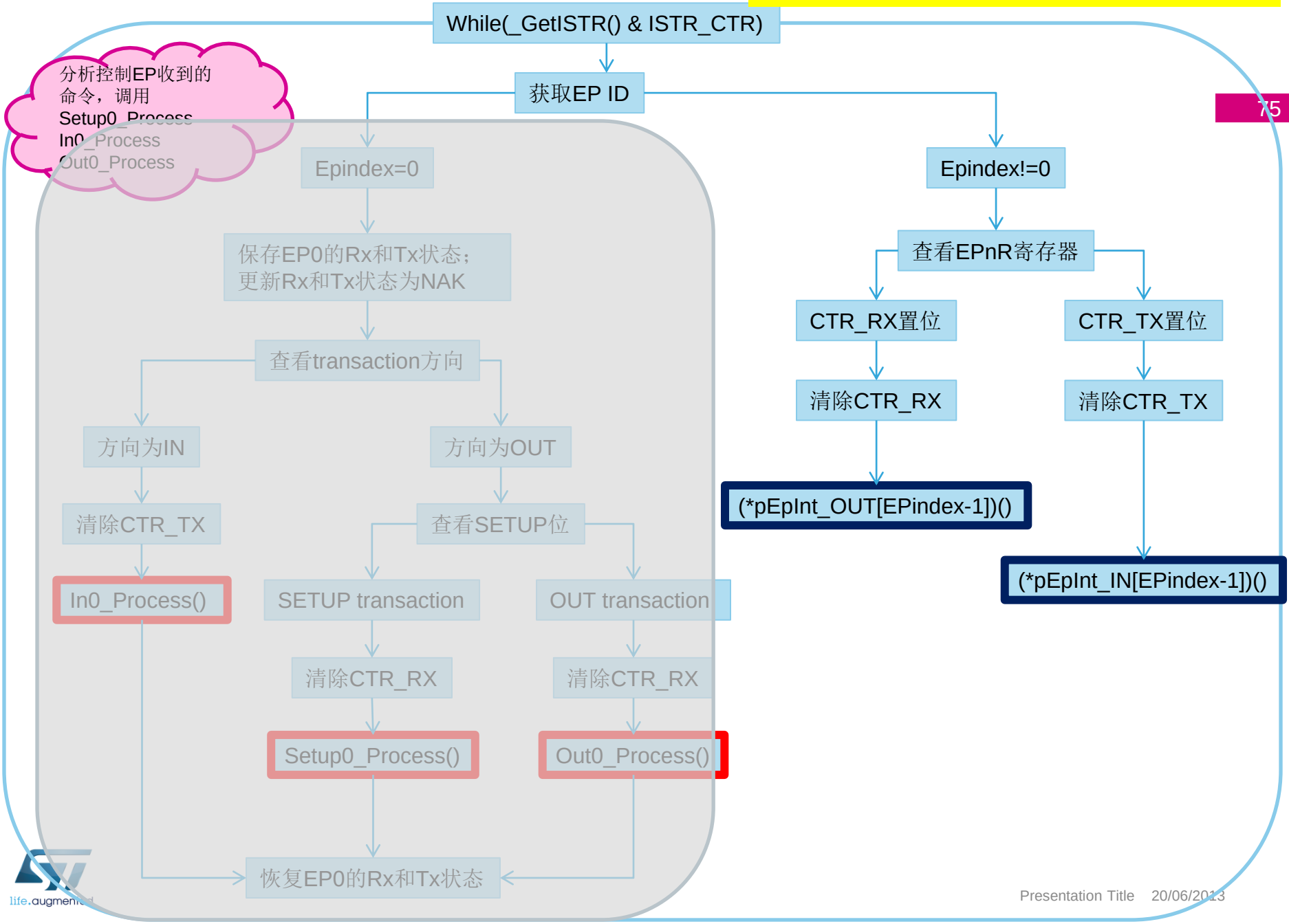
DataStageOut()



In0_Process()


pInformation->ControlState
(LAST_)IN_DATA
DataStageIn()
WAIT_STATUS_IN
如果是SET_ADDR命令：写寄存器，调用回调函数（全局变量bDeviceState = ADDRESSED）
pProperty->Process_Status_In()空函数
ControlState = STALLED
Others……
ControlState = STALLED
Post0_Process()

void CTR_LP(void) @《usb_int.c》



非0端点上的处理，应用相关的…

@ 《usb_istr.c》

```
void (*pEpInt_IN[7])(void) =
 {
   EP1_IN_Callback,
   EP2_IN_Callback,
   EP3_IN_Callback,
   EP4_IN_Callback,
   EP5_IN_Callback,
   EP6_IN_Callback,
   EP7_IN_Callback,
 };

void (*pEpInt_OUT[7])(void) =
 {
   EP1_OUT_Callback,
   EP2_OUT_Callback,
   EP3_OUT_Callback,
   EP4_OUT_Callback,
   EP5_OUT_Callback,
   EP6_OUT_Callback,
   EP7_OUT_Callback,
 };
```

@ 《usb_conf.h》

```
//#define  EP1_IN_Callback   NOP_Process
#define  EP2_IN_Callback   NOP_Process
#define  EP3_IN_Callback   NOP_Process
#define  EP4_IN_Callback   NOP_Process
#define  EP5_IN_Callback   NOP_Process
#define  EP6_IN_Callback   NOP_Process
#define  EP7_IN_Callback   NOP_Process


#define  EP1_OUT_Callback   NOP_Process
//#define  EP2_OUT_Callback   NOP_Process
#define  EP3_OUT_Callback  NOP_Process
#define  EP4_OUT_Callback   NOP_Process
#define  EP5_OUT_Callback   NOP_Process
#define  EP6_OUT_Callback   NOP_Process
#define  EP7_OUT_Callback   NOP_Process
```

@ 《usb_endp.c》

```
void EP1_IN_Callback(void)
{
  Mass_Storage_In();
}

void EP2_OUT_Callback(void)
{
  Mass_Storage_Out();
}
```

@ 《usb_bot.c》

基于BOT传输协议的SCSI命令传输

支持的SCSI命令


```
#define SCSI_REQUEST_SENSE          0x03
#define SCSI_INQUIRY                0x12
#define SCSI_START_STOP_UNIT        0x1B
#define SCSI_ALLOW_MEDIUM_REMOVAL   0x1E
#define SCSI_MODE_SENSE6            0x1A
#define SCSI_MODE_SENSE10           0x5A
#define SCSI_READ_FORMAT_CAPACITIES 0x23
#define SCSI_READ_CAPACITY10        0x25
#define SCSI_TEST_UNIT_READY        0x00
#define SCSI_READ10                 0x28
#define SCSI_WRITE10                0x2A
#define SCSI_VERIFY10               0x2F
#define SCSI_FORMAT_UNIT            0x04
```

```
#define SCSI_MODE_SELECT10          0x55
#define SCSI_MODE_SELECT6           0x15
#define SCSI_SEND_DIAGNOSTIC        0x1D
#define SCSI_READ6                  0x08
#define SCSI_READ12                 0xA8
#define SCSI_READ16                 0x88
#define SCSI_READ_CAPACITY16        0x9E
#define SCSI_WRITE6                 0x0A
#define SCSI_WRITE12                0xAA
#define SCSI_WRITE16                0x8A
#define SCSI_VERIFY12               0xAF
#define SCSI_VERIFY16               0x8F
```

SCSI_Invalid_Cmd() 出错处理：以下三点

>> 根据数据传输方向，STALL住IN、OUT或双向的EP

>> 更新CSW的bStatue=0x01 (Cmd failed)
把CSW写入EP IN的发送buf
Bot_state = BOT_ERROR

>> 填写Scsi_Sense_Data[]

实现一个USB设备的步骤：初始化

步骤1

```
/* 根据应用定义需使用的端点数 */
@ <usb_conf.h>
#define EP_NUM     (2)
```

```
usb_regs.h:
#define EP_BULK        (0x0000)
#define EP_CONTROL             (0x0200)
#define EP_ISOCHRONOUS         (0x0400)
#defie   EP_INTERRUPT          (0x0600)
```

步骤2

```
/* 初始化端点 */
@ <usb_prop.c>
SetEPType(ENDP1, EP_BULK);
SetEPTxAddr(ENDP1, ENDP1_TXADDR);
SetEPTxStatus(ENDP1, EP_TX_NAK);
SetEPRxStatus(ENDP1, EP_RX_DIS);
```

```
#define EP_TX_DIS              (0x0000)
#define EP_TX_STALL            (0x0010)
#define EP_TX_NAK              (0x0020)
#define EP_TX_VALID            (0x0030)
#define EP_RX_DIS              (0x0000)
#define EP_RX_STALL            (0x1000)
#define EP_RX_NAK              (0x2000)
#define EP_RX_VALID            (0x3000)
```

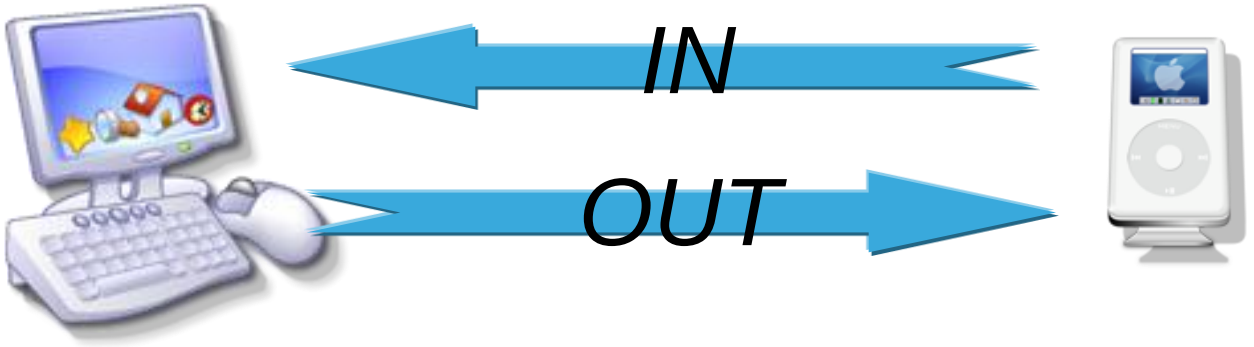
life.augmented

实现一个USB设备的步骤：使能端点

步骤3

```
/* 使能需要的端点 */

对于IN端点:
    USB_SIL_Write(EP1_IN, UserBuffer, Count);
    SetEPTxStatus (ENDP1, EP_TX_VALID);
对于OUT端点:
    SetEPRxStatus(ENDP1, EP_RX_VALID);
    ……
    USB_SIL_Read(EP1_OUT, UserBuffer)
```

主机



设备

life.augmented

实现一个USB设备的步骤：处理中断(1)

IN transaction

IN Token

NAK

步骤3：使能端点

IN Token

DATA

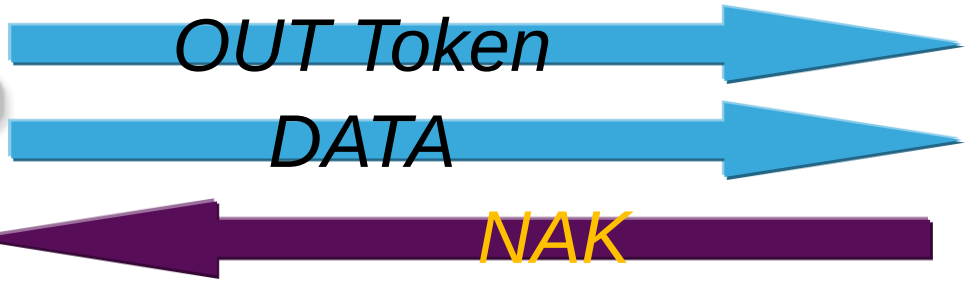
ACK

产生CTR中断，表示传输完毕：
>> 调用EPx_IN_Callback
>> 用户可以重复步骤3，继续下一次IN transaction

life.augmented

实现一个USB设备的步骤：处理中断(2)

OUT transaction



步骤3：使能端点

产生CTR中断，表示传输完毕：
>> 调用EPx_OUT_Callback
>> 用户需要将数据从端点换成拷贝到用户缓冲区
>> 用户可以重复步骤3，继续下一次OUT transaction

life.augmented

小结：实现USB设备的步骤

- 根据应用选择合适的USB类实现
- 根据所选择的USB类协议，完成各个描述符
  - 包括设备描述符，配置描述符，接口描述符，端点描述符和字符描述符
- 根据描述符初始化端点数目，分配各端点所需使用的Packet Buffer
- 初始化所使用的端点，配置端点的传输类型、方向，Packet Buffer地址和初始状态
- 在需要发送或接收数据的时候，使能端点
- 在该端点的中断回调函数中，处理数据，如果需要则使能下一次传输

USB通信由中断驱动


```
void USB_Istr(void)
{
 wIstr = _GetISTR();

#if (IMR_MSK & ISTR_WKUP)
  if (wIstr & ISTR_WKUP & wInterrupt_Mask)
  {
    _SetISTR((uint16_t)CLR_WKUP);
    Resume(RESUME_EXTERNAL);
#ifdef WKUP_CALLBACK
    WKUP_Callback();
#endif
  }
#endif
#if (IMR_MSK & ISTR_SUSP)
  if (wIstr & ISTR_SUSP & wInterrupt_Mask)
  {
    if (fSuspendEnabled)
    {
      Suspend();
    }
    else
    {
      /* if not possible then resume after xx ms */
      Resume(RESUME_LATER);
    }
    _SetISTR((uint16_t)CLR_SUSP);
#ifdef SUSP_CALLBACK
    SUSP_Callback();
#endif
    }
#endif
```

唤醒

挂起

- 不同应用关心不同的中断事件

```
@ 《usb_conf.h》 USB MSC Device demo
/* mask defining which events has to be handled */
/* by the device application software */
#define IMR_MSK (CNTR_CTRM  | CNTR_RESETM)

@ 《usb_conf.h》 USB JOYSTICK demo
#define IMR_MSK (CNTR_CTRM  | CNTR_WKUPM
                 | CNTR_SUSPM | CNTR_ERRM  | CNTR_SOFM \
                 | CNTR_ESOFM | CNTR_RESETM )
```

挂起

@ 《usb_istr.c》

```
#if (IMR_MSK & ISTR_SUSP)
if (wIstr & ISTR_SUSP & wInterrupt_Mask)
  {
    /* check if SUSPEND is possible */
    if (fSuspendEnabled)
    {
      Suspend();
    }
    else
    {
      Resume(RESUME_LATER);
    }

// ISTR的清零，必需要在软件置位FSUSP之后
_SetISTR((uint16_t)CLR_SUSP);
  }
#endif
```

@ 《usb_pwr.c》

```
__IO bool fSuspendEnabled = TRUE;
```

@ 《usb_pwr.c》

```
void Suspend(void)
{
  // 1. set FSUSP    @ USB_CNTR
  // 2. set LPMODE @ USB_CNTR
  // 3. Enter_LowPowerMode()
        >> bDeviceState = SUSPENDED
        >> clear EXT18 pending bit (USB唤醒事件)
        >> MCU进入STOP模式, WFI
}
```

唤醒

- The **Remote Wake Up is an optional feature** specified by the USB to allow the device to wake the host up from a stand by mode
  - 可见 Remote wakeup是设备唤醒主机
  - A USB device **reports its ability to support remote wakeup in its configuration descriptor** . If a device supports remote wakeup, it must also be allowed the capability to be enabled and disabled using the standard USB requests.
  - If the device supports the Remote Wake Up feature, the user has to manage the Set_Feature(DEVICE_REMOTE_WAKEUP) request

- This request is **the only request which can be initiated by the device**, but it has to be allowed by the host.
  - **The host sends a Set Feature request to enable the Remote Wake Up feature just before sending the suspend request**. If the host did not send the Set Feature (RemoteWakeUpEnable), the device is not allowed to perform this feature

[Q] When is SetFeature(DEVICE_REMOTE_WAKEUP) issued from the host?"
[A] When the 'Remote Wakeup' bit is enabled in the bmAttributes on the device configuration descriptor,
- Windows:
- - **puts SetFeature(DEVICE_REMOTE_WAKEUP) to the device just before it goes to stand-by**. **Then suspend the bus (stop SOF)**
- - **puts ClearFeature(DEVICE_REMOTE_WAKEUP) as the first request just after wakeup.**
- **Linux:**
- - doesn't support this feature.

Joystick远程唤醒的原理

- 正常运行时，鼠标工作OK，主机也在一直发送SOF
- 当手动把PC调到standby状态不再发送SOF，那么ESOF会被置位
  - 1st ESOF set : ISR s/w clear
  - 2nd ESOF set : ISR s/w clear
  - 3rd ESOF set : SUSPEND也被置位了！在SUSPEND对应的ISR中，进入了MCU的standby模式，(with ESOF flag set)
- 直到press KEY at board, 该key对应的EXTI中断唤醒了MCU，在EXTI的ISR中开始执行：Resume(Internal)，配置PLL等，然后把ResumeS.eState改成START，退出
  - 由于此时ESOF还置位，因此在执行ISR中对应case，Resume(ESOF)，于是通过s/w置位ESOF@CNTR，发出Resume信号给host，把PC唤醒了

Joystick用到的中断处理

@ 《stm32f10x_it.c》

```
void USB_LP_CAN1_RX0_IRQHandler(void)    // USB正常通信
{
  USB_Istr();
}

void EXTI9_5_IRQHandler(void)   // KEY键唤醒
{
  if (EXTI_GetITStatus(KEY_BUTTON_EXTI_LINE) != RESET)
  {
    if (pInformation->Current_Feature & 0x20)
    {
      /* Exit low power mode and re-configure clocks */
      Resume(RESUME_INTERNAL);
     }

    /* Clear the EXTI line pending bit */
    EXTI_ClearITPendingBit(KEY_BUTTON_EXTI_LINE);
  }
}

void USBWakeUp_IRQHandler(void)     // USB唤醒
{
  EXTI_ClearITPendingBit(EXTI_Line18);
}
```

AA

BB

@ 《usb_istr.c》

```
#if (IMR_MSK & ISTR_WKUP)
  // 当在挂起模式时检测到唤醒USB的事件
  // 该事件会异步地触发USB_WAKUP线
  if (wIstr & ISTR_WKUP & wInterrupt_Mask)
  {
    _SetISTR((uint16_t)CLR_WKUP);
    Resume(RESUME_EXTERNAL);
   }
#endif
#if (IMR_MSK & ISTR_SUSP)
  if (wIstr & ISTR_SUSP & wInterrupt_Mask)
  {
    if (fSuspendEnabled)
    {  Suspend();  }
    else
    { Resume(RESUME_LATER); }
    gl_array[gl_index++] = 0xCC;
    _SetISTR((uint16_t)CLR_SUSP);
}
#endif
#if (IMR_MSK & ISTR_ESOF)
  if (wIstr & ISTR_ESOF & wInterrupt_Mask)
  {
    _SetISTR((uint16_t)CLR_ESOF);
    /* resume handling timing is made with ESOFs */
    Resume(RESUME_ESOF);
    /* request without change of the machine state */
```

DD

CC

pInformation->Current_Feature

- Joystick_Reset，即**Device_Property.Reset()** @《USB_Istr》
  - pInformation->Current_Feature = Joystick_ConfigDescriptor[7]; // 0xE0
- NoData_Setup0 →Standard_ClearFeature @ 《usb_core.c》
  - ClrBit(pInformation->Current_Feature, 5);
- NoData_Setup0 →Standard_SetDeviceFeature @ 《usb_core.c》
  - SetBit(pInformation->Current_Feature, 5);

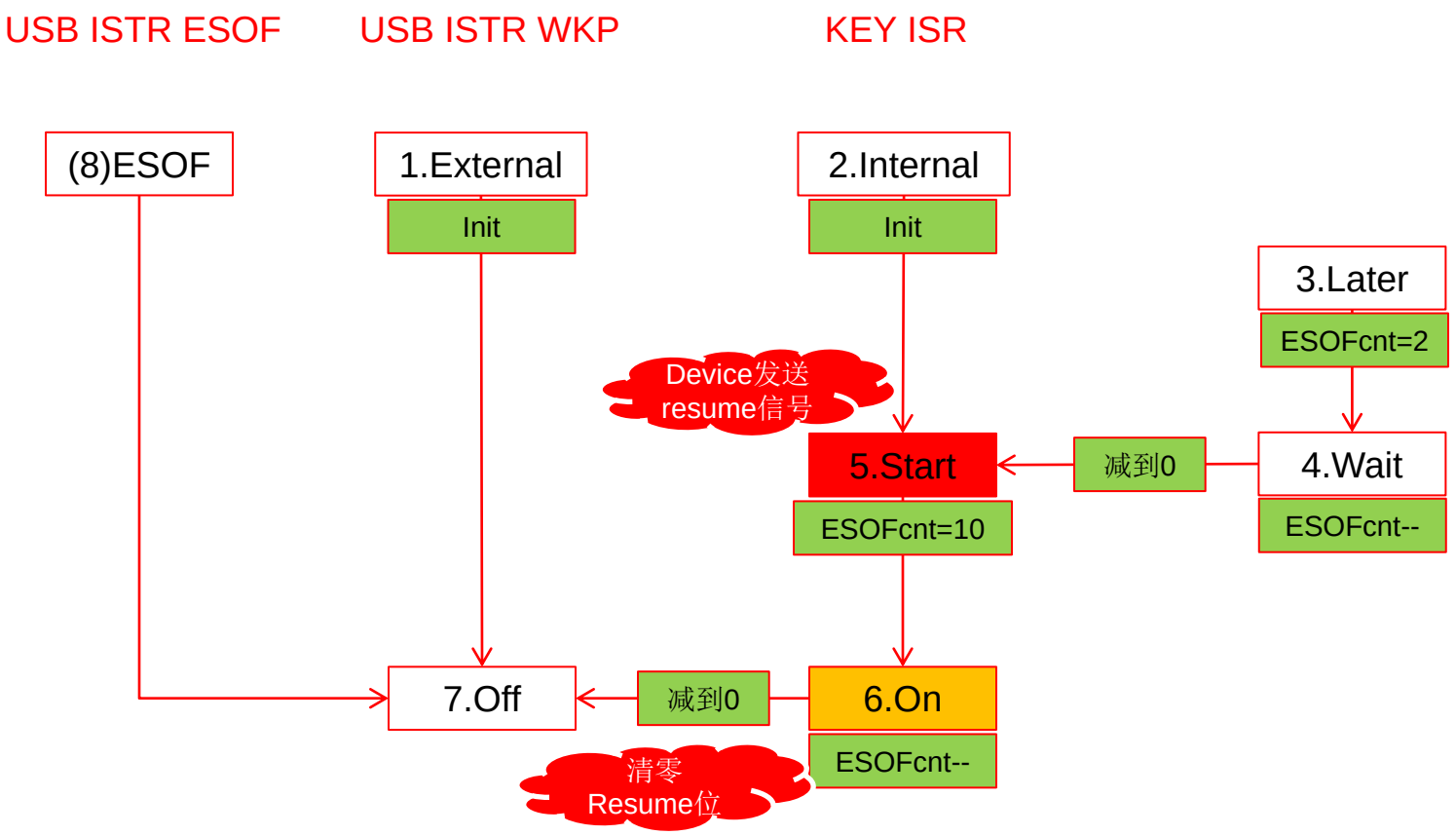
Table 9-10. Standard Configuration Descriptor (Continued)

Offset	Field	Size	Value	Description
7	*bmAttributes*	1	Bitmap	Configuration characteristics D7: Reserved (set to one) D6: Self-powered D5: Remote Wakeup D4...0: Reserved (reset to zero)

Resume

Bit 4 **RESUME:** Resume request

The microcontroller can set this bit to send a Resume signal to the host. It must be activated, according to USB specifications, for no less than 1mS and no more than 15mS after which the Host PC is ready to drive the resume sequence up to its end.



- PC自己从standby唤醒
  - BB：USBWakeup_IRQHandler → CC：从进入stop mode的地方原地唤醒继续往后执行 → 77/88/99 USB ISR ESOF → 11、DD：USB ISTR Wakeup → Resume(External) → 77/88/99 若干USB ISR ESOF → 正常USB通信
  - 可见：主机发出的Resume信号后，先触发了EXTI18（USBWakeup）把MCU唤醒，从刚才休眠的地方继续往后执行；开始计时仍未收到SOF，进入USB ISTR ESOF；进入USB ISTR WKUP；继续仍未收到SOF；直到PC开始正常通信（有SOF了）

- PC被Joystick的KEY按键远程唤醒
  - AA：KEY ISR → 22：Resume(Internal) → CC：从进入stop mode的地方原地唤醒继续往后执行 → 55：**Resume(Start) 这个case是由ESOF触发的，就是调用Resume(ESOF)，由于输入参数是ESOF，因此用上次在Resume函数中得到的下个状态，就是Start case** → 66：Resume(On) 这个case也是由ESOF触发的，同样由于参数是ESOF，因此用上次Resume函数中得到的下个状态，即On case…若干个 → 77/88/99：若干USB ISR ESOF

- 由主机唤醒，会触发的INT有
  - USBWakeup_IRQ(EXTI18)、USB_IRQ.Wakeup（在此执行ResuemInit，即重新配置系统时钟）

- 由设备远程唤醒，会触发的INT有
  - KEY_IRQ(EXTIx)、在总线恢复正常之前的USB_IRQ.ESOF执行Resume(state)，包括设备发送Resume总线信号

Some extending…

USBWakeup	USB wakeup from suspend through EXTI line interrupt	0x0000_00E8

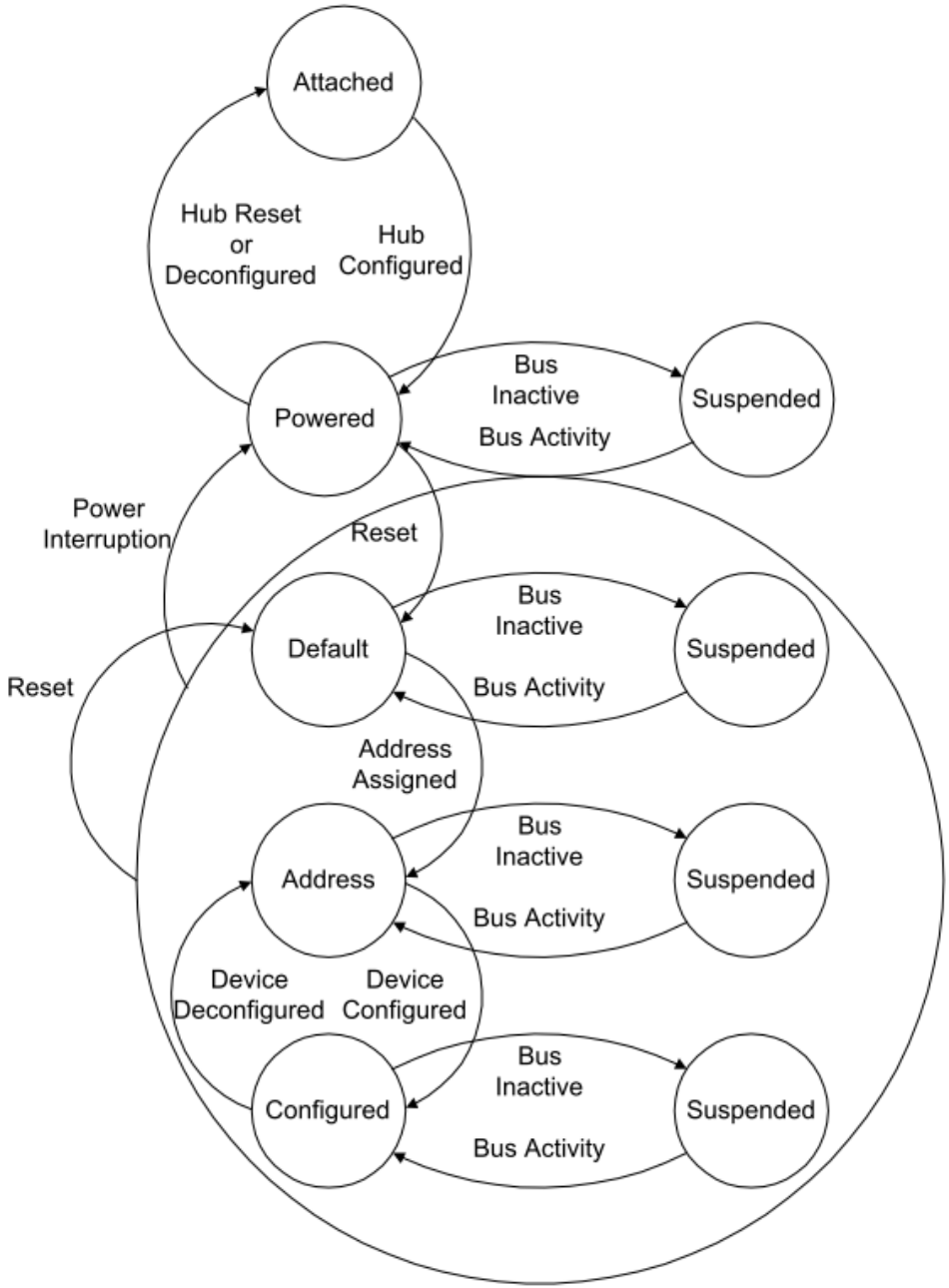
- F103关于USB的中断
  - USB_HP_CAN_TX：USB High Priority interrupt
  - USB_LP_CAN_RX0：USB low priority interrupt
  - USBWakeukp：USB wakeup from suspend through EXTI18
    - 只有device才有suspend模式，因此这个中断是用于唤醒USB DEVICE IP的
    - 当USB总线上3ms没有信号了会触发USB中断，软件通过写控制器来把MCU进入低功耗模式；软件也可以什么都不做。。。
    - Device处于suspend模式时，host开始恢复总线活动：1）触发EXTI18来把MCU从stop模式唤醒；2）MCU执行EXTI18的ISR；3）MCU从刚才进入stop模式的地方继续执行；4）响应USB ISR中的wakeup对应处理程序

- F105/107（可做device，也可作host）关于USB的中断
  - OTG_FS_WKUP：USB OTG FS Wakeup through EXTI line
  - OTG_FS：USB OTG global interrupt
    - USB OTG FS Wakeup through EXTI line interrupt

OTG_FS_WKUP	USB On-The-Go FS Wakeup through EXTI line interrupt	0x0000_00E8

USB协议...设备的状态...

Attached	Powered	Default	Address	Configured	Suspended	State
No	--	--	--	--	--	Device is not attached to the USB. Other attributes are not significant.
Yes	No	--	--	--	--	Device is attached to the USB, but is not powered. Other attributes are not significant.
Yes	Yes	No	--	--	--	Device is attached to the USB and powered, but has not been reset.
Yes	Yes	Yes	No	--	--	Device is attached to the USB and powered and has been reset, but has not been assigned a unique address. Device responds at the default address.
Yes	Yes	Yes	Yes	No	--	Device is attached to the USB, powered, has been reset, and a unique device address has been assigned. Device is not configured.
Yes	Yes	Yes	Yes	Yes	No	Device is attached to the USB, powered, has been reset, has a unique address, is configured, and is not suspended. The host may now use the function provided by the device.
Yes	Yes	--	--	--	Yes	Device is, at minimum, attached to the USB and is powered and has not seen bus activity for 3 ms. It may also have a unique address and be configured for use. However, because the device is suspended, the host may not use the device's function.



从USB协议来看Suspend

- Suspend是**设备**的众多状态中的一种
- 一旦被power了，设备就要随时准备进入suspend状态，无论它是否已被分配了地址和是否已经完成了来自主机的配置
  - 对于STM32F103：the suspend condition check is enabled immediately after any USB reset and it's disabled by hw when suspend mode is active by FSUSP=1 until the end of resume sequence
- 设备是否支持remote wakeup特性，要在自己的配置描述子中说清楚；如果支持的话，那么它也要继续支持host发来的对该特性的使能和禁止的request
  - 对于STM32F105作为host，当检测到remote wakeup，产生WKUPINT中断
  - 该中断对于device同样有效，也是检测到resume信号，产生对应中断

PC休眠并自我唤醒

Transfer	L/S	Interrupt	ADDR	ENDP	Bytes Transferred	Time	Time Stamp
17	S	IN	2	1	4	87.996 ms	0 . 361 179 450
18	S	IN	2	1	4	31.998 ms	0 . 449 175 182
19	S	IN	2	1	4	1.126 sec	0 . 481 173 582

Transfer	L/S	Control	ADDR	ENDP	bRequest	wValue	wIndex	wLength
20	S	SET	2	0	SET_FEATURE	DEVICE_REMOTE_WAKEUP	0x0000	0

Packet	H	Suspend	Time Stamp
2045	↓	16.263 sec	1 . 655 108 232

Packet	?	Resume	Time	Time Stamp
2046		46.830 ms	203.889 ms	17 . 918 054 100

Transfer	L/S	Control	ADDR	ENDP	bRequest	wValue	wIndex	wLength	Time
21	S	SET	2	0	CLEAR_FEATURE	0x0001	0x0000	0	3.000 ms

Transfer	L/S	Control	ADDR	ENDP	D	Tp	R	bRequest	wValue	wIndex	wLength	STALL
22	S	SET	2	0	H->D	C	I	0x0A	0x0000	0x0000	0	0x08
23	S	SET	2	0	H->D	C	I	0x0A	0x0000	0x0000	0	0x08

Transfer	L/S	Interrupt	ADDR	ENDP	Bytes Transferred	Time	Time Stamp
24	S	IN	2	1	4	8.000 ms	26 . 721 531 450
25	S	IN	2	1	4	8.000 ms	26 . 729 531 050
26	S	IN	2	1	4	8.000 ms	26 . 737 530 650

PC休眠并被鼠标唤醒

Transfer	L/S	Interrupt	ADDR	ENDP	Bytes Transferred	Time	Time Stamp
344	S	IN	1	1	4	63.997 ms	5 . 305 548 916
345	S	IN	1	1	4	63.997 ms	5 . 369 545 850
346	S	IN	1	1	4	1.178 sec	5 . 433 542 782

Transfer	L/S	Control	ADDR	ENDP	bRequest	wValue	wIndex	wLength
347	S	SET	1	0	SET_FEATURE	DEVICE_REMOTE_WAKEUP	0x0000	0

Packet	H	Suspend	Time Stamp
8621	↓	14.163 sec	6 . 653 475 332

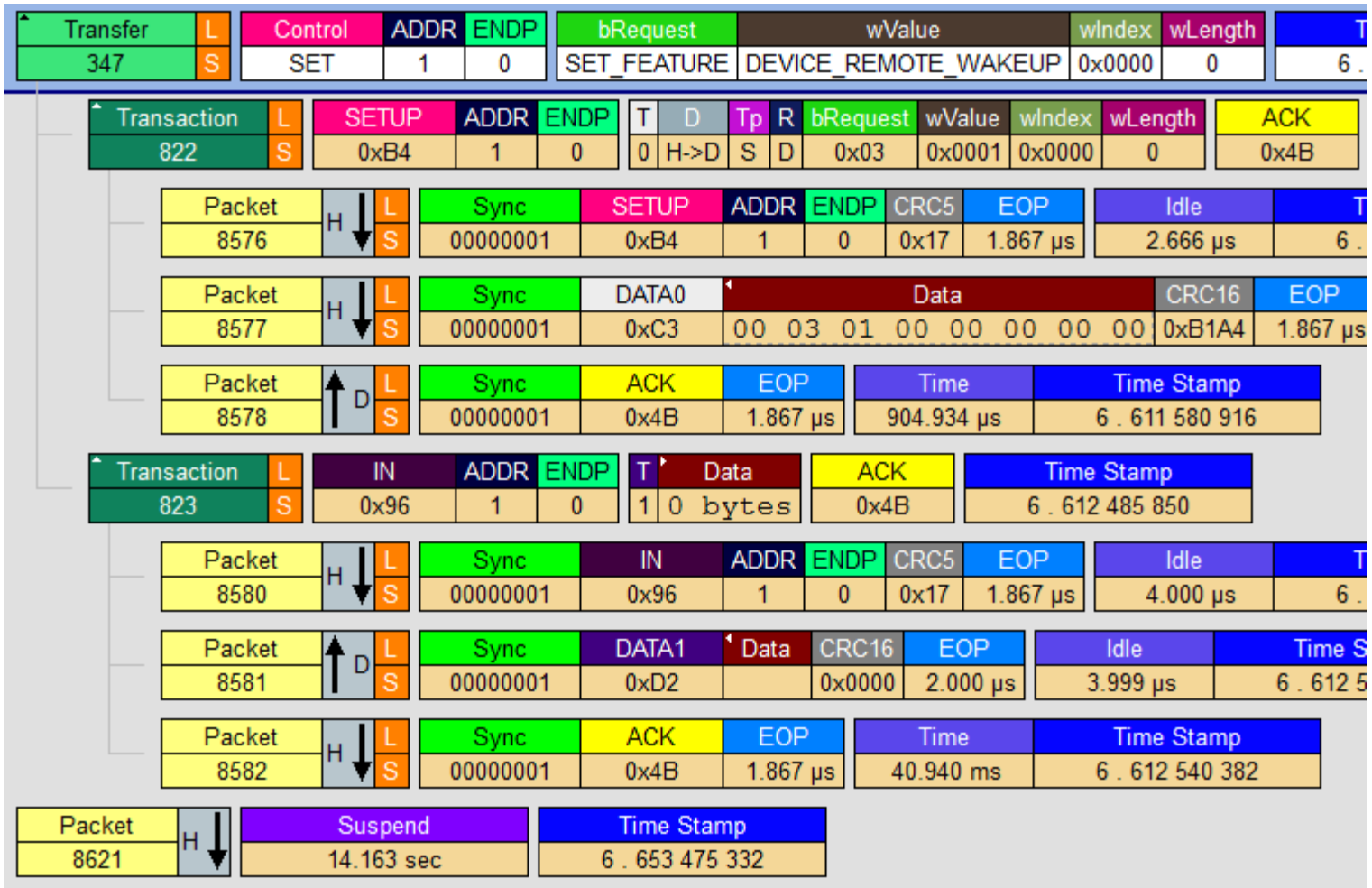
Packet	?	Resume	Time	Time Stamp
8622		1.506 sec	1.664 sec	20 . 816 025 932

Transfer	L/S	Control	ADDR	ENDP	bRequest	wValue	wIndex	wLength	Time
348	S	SET	1	0	CLEAR_FEATURE	0x0001	0x0000	0	3.000 ms

Transfer	L/S	Control	ADDR	ENDP	D	Tp	R	bRequest	wValue	wIndex	wLength	STALL
349	S	SET	1	0	H->D	C	I	0x0A	0x0000	0x0000	0	0x08
350	S	SET	1	0	H->D	C	I	0x0A	0x0000	0x0000	0	0x08

Transfer	L/S	Interrupt	ADDR	ENDP	Bytes Transferred	Time	Time Stamp
351	S	IN	1	1	4	8.000 ms	22 . 489 561 716
352	S	IN	1	1	4	8.000 ms	22 . 497 561 316
353	S	IN	1	1	4	8.000 ms	22 . 505 560 916

PC在休眠前使能设备的远程唤醒feature



PC被鼠标唤醒后禁止掉其远程唤醒feature