USB 传输数据时出现卡顿现象

1 前言

在进行 USB 开发的过程中，有多个客户反馈，USB 传输数据时出现卡顿现象。本文将针对这一问题进行分析。

2 问题分析

这几个客户问题现象基本差不多，采用 STM32 作为 Device 设备，在与上位机或者 PC 端双向通讯一段时间后，从 Device 端到 Host 端的数据能够正常，而从 Host 端到 Device 端的数据异常，也就是说，STM32 在一段时间后不再能正常接收数据，但是，如果只是单向通信，就一直都是正常的。

这几个客户，有用 STM32F2 的，也有用 STM32F4 的，有用 CDC 类的，也有用作 HID 设备的，但都使用了 Cube 库。

下面就具体问题以其中一个客户使用 STM32F411 的 USB CDC 类的案例来分析问题，现象如下 USB 通讯数据(CDC 类)：

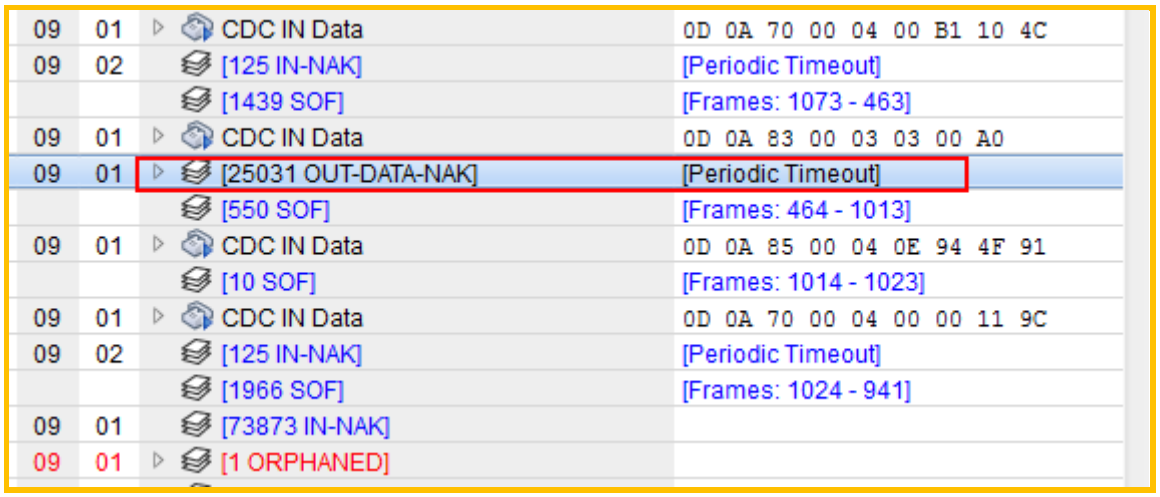

Figure 1 通信一段时间后 Data Out 数据异常

展开 Data Out 数据：

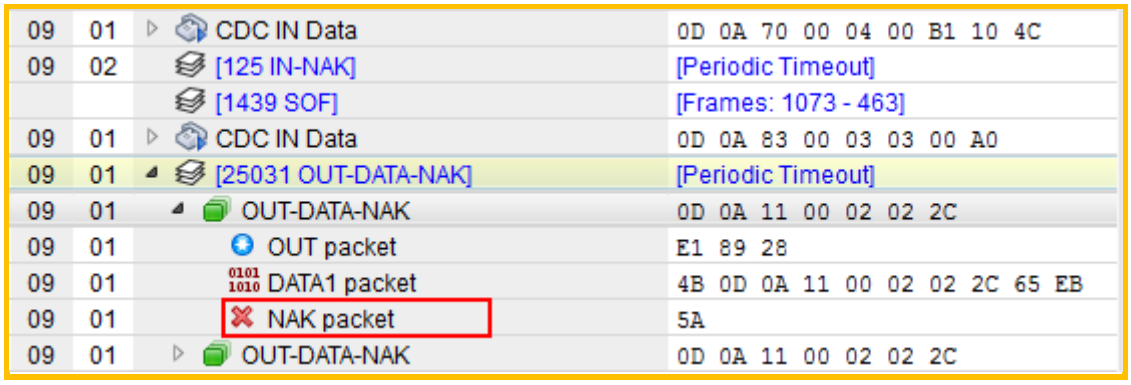

Figure 2 Data Out 数据异常

分析上图发现，并不是 Host 端没有向 Device 端发送 Data Out 数据，而是确实发送了，但被 Device 端 NAK 了。那么为什么会被 NAK 呢？
通过在调试下查看寄存器，我们发现当出现问题时，Data OUT 对应的端点 1 是处于关闭状态，那么为什么端点 1 会关闭？查看 STM32 端的接收代码：

usbd_cdc_if.c:

```
static int8_t CDC_Receive_FS (uint8_t* Buf, uint32_t *Len)
{
      /* USER CODE BEGIN 6 */
      //USBTask_ReceiveMsg(Buf, *Len);       //UserRxBufferFS
      USBD_CDC_SetRxBuffer(&hUsbDeviceFS, &Buf[0]);
      USBD_CDC_ReceivePacket(&hUsbDeviceFS);

      return (USBD_OK);
      /* USER CODE END 6 */
}
```

如上代码，在 MCU 端接收到赖在 Host 端的数据后不做任何处理就立即接收下一次数据传输，问题是，这里对接收到的数据啥也没有做，居然还会出现 Data Out 端点关闭的问题，那么 OUT 端点到底是怎么关闭的呢？我们接下来看子函数：
CDC_Receive_FS() ->USBD_CDC_ReceivePacket() ->USBD_LL_PrepareReceive() ->HAL_PCD_EP_Receive() ->USB_EPStartXfer()
最后在 USB_EPStartXfer 函数中有发现再次使能 OUT 端点的代码：

```
//…
  else /* OUT endpoint */
  {
    /* Program the transfer size and packet count as follows:
    * pktcnt = N
    * xfersize = N * maxpacket
    */
    USBx_OUTEP(ep->num)->DOEPTSIZ &= ~(USB_OTG_DOEPTSIZ_XFRSIZ);
    USBx_OUTEP(ep->num)->DOEPTSIZ &= ~(USB_OTG_DOEPTSIZ_PKTCNT);

    if (ep->xfer_len == 0U)
    {
      USBx_OUTEP(ep->num)->DOEPTSIZ |= (USB_OTG_DOEPTSIZ_XFRSIZ & ep->maxpacket);
      USBx_OUTEP(ep->num)->DOEPTSIZ |= (USB_OTG_DOEPTSIZ_PKTCNT & (1U << 19U));
    }
    else
    {
      pktcnt = (ep->xfer_len + ep->maxpacket -1U)/ ep->maxpacket;
      USBx_OUTEP(ep->num)->DOEPTSIZ |= (USB_OTG_DOEPTSIZ_PKTCNT & (pktcnt << 19U));
      USBx_OUTEP(ep->num)->DOEPTSIZ |= (USB_OTG_DOEPTSIZ_XFRSIZ & (ep->maxpacket *
pktcnt));
    }

    if (dma == 1U)
    {
```

```
      USBx_OUTEP(ep->num)->DOEPDMA = (uint32_t)ep->xfer_buff;
    }

    if (ep->type == EP_TYPE_ISOC)
    {
      if ((USBx_DEVICE->DSTS & ( 1U << 8U )) == 0U)
      {
        USBx_OUTEP(ep->num)->DOEPCTL |= USB_OTG_DOEPCTL_SODDFRM;
      }
      else
      {
        USBx_OUTEP(ep->num)->DOEPCTL |= USB_OTG_DOEPCTL_SD0PID_SEVNFRM;
      }
    }
    /* EP enable */
    USBx_OUTEP(ep->num)->DOEPCTL |= (USB_OTG_DOEPCTL_CNAK | USB_OTG_DOEPCTL_EPENA);
  }
```

也就是说，在调用这个函数之前这个 OUT 端点原本就是关闭的？真的吗？这怎么跟我们理解的不一样？不应该是 OUT 端点一旦打开就一直开着的吗？带着这些疑问，我们查看 STM32F411 的参考手册，终于在 22.17.6 Operational model 一节中找到这么一幅图(由于 CDC 类数据传输采用的是 BULK 传输):

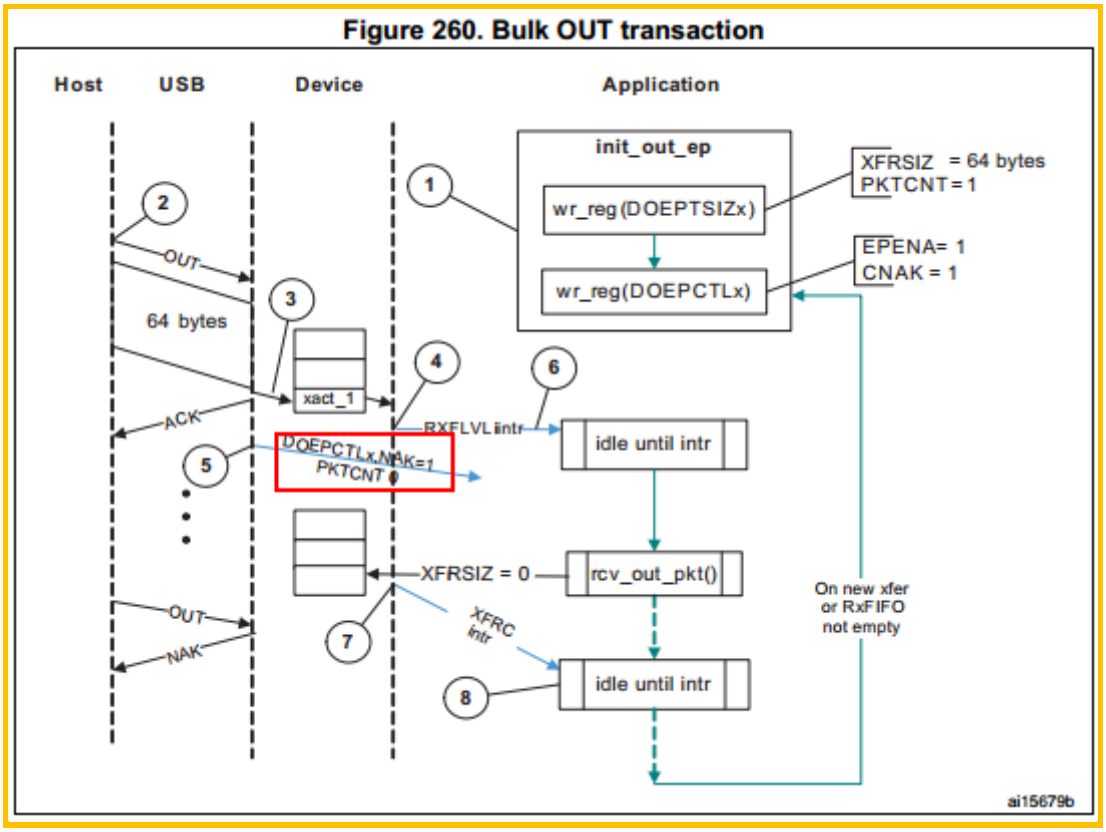

Figure 3 MCU 对 OUT 数据传输的处理流程

如上图，MCU 对 BULK 类型的 OUT 数据处理例程大体如下：

1> Host 端试图向一个端点发送 OUT token；

2> 当 Device 端的 USB 外设接收到这么一个 OUT token 后，如果 RXFIFO 空间足够，它将数据包存储到 RXFIFO 中；
3> 在将数据包内容存储到 RXFIFO 后，USB 外设将产生一个 RXFLVL 中断(OTG_FS_GINTSTS)；
4> 在接收到 USB 数据包的个数后(PKTCNT),USB 核将内部自动将这个 OUT 端点的 NAK 为置 1，以阻止接收更多数据包；
5> 应用程序处理 RXFLVL 中断和从 RXFIFO 读取数据；
6> 当应用读取完所有数据(等于 XFRSIZ)后，USB 核将产生一个 XFRC 中断(OTG_FS_DOEPINTx);
7> 应用处理这个 OTG_FS_DOEPINTx 中断并通过 OTG_FS_DOEPINTx 的中断为 XFRC 来判断传输完成；

从上面步骤中的第 4 步中可以看出，当 USB 核收到来自 Host 端的数据后会自动将 OUT 端点关闭，这也就是为什么在接收函数中在接收下一次数据时要再次使能这个 OUT 端点的原因。因此我们大体可以判断出在 OUT 数据传输的过程中，USB 核会禁止端点->打开端点->禁止端点…如此不断循环中；那么问题到底出现在哪里呢？会不会在 USB 核自动关闭端点后就没有再次成功打开？带着这样的怀疑心态逐句查看代码，最终在接收函数的子函数中发现这么一段代码：

```
HAL_StatusTypeDef HAL_PCD_EP_Receive(PCD_HandleTypeDef *hpcd, uint8_t ep_addr, uint8_t
*pBuf, uint32_t len)
{
  USB_OTG_EPTypeDef *ep;

  ep = &hpcd->OUT_ep[ep_addr & 0x7FU];

  /*setup and start the Xfer */
  ep->xfer_buff = pBuf;
  ep->xfer_len = len;
  ep->xfer_count = 0U;
  ep->is_in = 0U;
  ep->num = ep_addr & 0x7FU;

  if (hpcd->Init.dma_enable == 1U)
  {
    ep->dma_addr = (uint32_t)pBuf;
  }

  __HAL_LOCK(hpcd);

  if ((ep_addr & 0x7FU) == 0U)
  {
    USB_EP0StartXfer(hpcd->Instance , ep, hpcd->Init.dma_enable);
  }
  else
  {
    USB_EPStartXfer(hpcd->Instance , ep, hpcd->Init.dma_enable);
  }
  __HAL_UNLOCK(hpcd);
```

```
  return HAL_OK;
}
```

之所以会怀疑这里，这是客户提供了一个信息，单向通信的时候就不会有问题！这是因为在发送数据时，发送函数的底层函数内也使用到了这个互斥锁:

```
CDC_Transmit_FS() -> USBD_CDC_TransmitPacket() -> USBD_LL_Transmit() ->
HAL_PCD_EP_Transmit() :
HAL_StatusTypeDef HAL_PCD_EP_Transmit(PCD_HandleTypeDef *hpcd, uint8_t ep_addr, uint8_t
*pBuf, uint32_t len)
{
  USB_OTG_EPTypeDef *ep;

  ep = &hpcd->IN_ep[ep_addr & 0x7FU];

  /*setup and start the Xfer */
  ep->xfer_buff = pBuf;
  ep->xfer_len = len;
  ep->xfer_count = 0U;
  ep->is_in = 1U;
  ep->num = ep_addr & 0x7FU;

  if (hpcd->Init.dma_enable == 1U)
  {
    ep->dma_addr = (uint32_t)pBuf;
  }

  __HAL_LOCK(hpcd);

  if ((ep_addr & 0x7FU) == 0U)
  {
    USB_EP0StartXfer(hpcd->Instance , ep, hpcd->Init.dma_enable);
  }
  else
  {
    USB_EPStartXfer(hpcd->Instance , ep, hpcd->Init.dma_enable);
  }

  __HAL_UNLOCK(hpcd);
  return HAL_OK;
}
```

接收处理数据时，底层是通过接收中断回调上来的，但发送时，我们往往将发送放到 main 等用户函数中。这两个是不一样的，一个在中断内，一个在中断外，优先级别是不一样的，优先级不一样就有可能导致资源冲突；

我们进一步查看__HAL_LOCK()宏定义:

```
  #define __HAL_LOCK(__HANDLE__)                                          \
                                  do{                                     \
```

```
                                    if((__HANDLE__)->Lock == HAL_LOCKED)   \
                                    {                                      \
                                       return HAL_BUSY;                    \
                                    }                                      \
                                    else                                   \
                                    {                                      \
                                       (__HANDLE__)->Lock = HAL_LOCKED;    \
                                    }                                      \
                                  }while (0U)
```

若__HAL_LOCK(hpcd);失败则直接返回 return HAL_BUSY 的。为了验证在接收过程中是否__HAL_LOCK 失败，我们引进全局变量 Lock_Flag,在发送函数中若成功 LOCK 则设置 Lock_Flag=1,UNLOCK 后则复位为 0：

```
uint8_t Lock_Flag =0;
HAL_StatusTypeDef HAL_PCD_EP_Transmit(PCD_HandleTypeDef *hpcd, uint8_t ep_addr, uint8_t
*pBuf, uint32_t len)
{
  USB_OTG_EPTypeDef *ep;

  ep = &hpcd->IN_ep[ep_addr & 0x7FU];

  /*setup and start the Xfer */
  ep->xfer_buff = pBuf;
  ep->xfer_len = len;
  ep->xfer_count = 0U;
  ep->is_in = 1U;
  ep->num = ep_addr & 0x7FU;

  if (hpcd->Init.dma_enable == 1U)
  {
    ep->dma_addr = (uint32_t)pBuf;
  }

  __HAL_LOCK(hpcd);
  Lock_Flag =1;

  if ((ep_addr & 0x7FU) == 0U)
  {
    USB_EP0StartXfer(hpcd->Instance , ep, hpcd->Init.dma_enable);
  }
  else
  {
    USB_EPStartXfer(hpcd->Instance , ep, hpcd->Init.dma_enable);
  }

  __HAL_UNLOCK(hpcd);
  Lock_Flag =0;
  return HAL_OK;
```

```
}
```

接下来在接收函数中对全局变量 Lock_Flag 值进行判断，若为 1 则锁死程序，因为在 Lock_Flag=1 时，则表示发送函数中已经获取了锁没有释放，此时若再去获取则会导致失败从而返回 HAL_BUSY;这里通过锁死代码以便判断这种情况:

```
HAL_StatusTypeDef HAL_PCD_EP_Receive(PCD_HandleTypeDef *hpcd, uint8_t ep_addr, uint8_t
*pBuf, uint32_t len)
{
  USB_OTG_EPTypeDef *ep;

  ep = &hpcd->OUT_ep[ep_addr & 0x7FU];

  /*setup and start the Xfer */
  ep->xfer_buff = pBuf;
  ep->xfer_len = len;
  ep->xfer_count = 0U;
  ep->is_in = 0U;
  ep->num = ep_addr & 0x7FU;

  if (hpcd->Init.dma_enable == 1U)
  {
    ep->dma_addr = (uint32_t)pBuf;
  }

  if(Lock_Flag ==1)
  {
      while(1);
  }
  __HAL_LOCK(hpcd);

  if ((ep_addr & 0x7FU) == 0U)
  {
    USB_EP0StartXfer(hpcd->Instance , ep, hpcd->Init.dma_enable);
  }
  else
  {
    USB_EPStartXfer(hpcd->Instance , ep, hpcd->Init.dma_enable);
  }
  __HAL_UNLOCK(hpcd);

  return HAL_OK;
}
```

通过调试，当出现问题时，程序果然被锁死在这个 while(1)了，这也证明了正是这个互斥锁所致。因此，我们大体可以判断出现问题时流程大致如下:

1> 在 mian 函数中发送数据 CDC_Transmit_FS()
2> USBD_CDC_TransmitPacket()
3> USBD_LL_Transmit()
4> HAL_PCD_EP_Transmit()

5> __HAL_LOCK(hpcd); 此时成功获取互斥锁
6> 恰好此时有一个接收中断，由于 USB 中断具有优先级，跳转到接收中断内执行；同时，USB 核会自动关闭 OUT 端点；
7> HAL_PCD_DataOutStageCallback()
8> USBD_CDC_DataOut()
9> CDC_Receive_FS()
10> USBD_CDC_ReceivePacket()
11> USBD_LL_PrepareReceive()
12> HAL_PCD_EP_Receive()
13> __HAL_LOCK(hpcd); 此时获取互斥锁失败导致返回，接收函数在 OUT 端点没有再次打开就已经提前结束，导致接收循环无以为继。

3 解决方案

知道了问题原因所在，接下来解决问题就相对来说比较容易的了。由于此问题是发送与接收处于不同优先等级导致资源冲突所致，那么我们可以将发送也放到与 USB 接收中断相同的中断等级中去，例如可以利用 USB 的 EOPF 中断，在开启 EOPF 中断后，在此中断内发送数据，这样发送与接收中断就处于相同等级了，EOPF 每 1ms 触发一次，速度完全可以。当然开启一个相同优先级的定时器来做发送数据也是可以，只不过定时器间隔得控制好。

此外，其实此问题是出现在 Cube 库的低版本中，例如 CubeF4 V1.5.0 和 CubeF2 V1.3.0 中都存在，但是在最新本的 CubeF4 V1.16.0,CubeF2 V1.6.0 版本中此问题得到了解决；此问题虽然后来发现是版本太旧所致，但从多个客户反馈此问题来看，此问题依然不失为一个很好的参考和教训。

重要通知 – 请仔细阅读

意法半导体公司及其子公司（“ST”）保留随时对ST 产品和/ 或本文档进行变更、更正、增强、修改和改进的权利，恕不另行通知。买方在订货之前应获取关于ST 产品的最新信息。ST 产品的销售依照订单确认时的相关ST 销售条款。

买方自行负责对ST 产品的选择和使用， ST 概不承担与应用协助或买方产品设计相关的任何责任。

ST 不对任何知识产权进行任何明示或默示的授权或许可。

转售的ST 产品如有不同于此处提供的信息的规定，将导致ST 针对该产品授予的任何保证失效。

ST 和ST 徽标是ST 的商标。所有其他产品或服务名称均为其各自所有者的财产。

本文档中的信息取代本文档所有早期版本中提供的信息。