AN12597

K32L2 系列 MCU 上实现 USB 转多个虚拟串口

版本 0 — 2019 年 11 月

应用笔记

目录

1 简介

本应用笔记介绍了如何在 K32L2 系列 FRDM 板（FRDM-K32L2B3/FRDM-K32L2A）上实现 USB 转多个虚拟串口的功能。USB 到 VCOM 的功能可以通过使用 USB 协议指定的 CDC 类中抽象控制模型的子类中的公共 AT 命令来实现。一个 USB 设备可以支持一个或多个 VCOM，VCOM 的数量主要取决于 USB 设备支持的端点（EP）的数量。用于 K32L2 系列 MCU 的全速（FS）USB 设备控制器支持 16 个双向端点，最多支持 15 个 VCOM。本应用笔记基于 SDK 代码（dev_composite_cdc_vcom_cdc_vcom_lite_bm），实现了 USB 至 15 个 VCOM 的功能，开发工具为 MCUXpresso IDE。

2 USB 描述符配置

USB 描述符等效于 USB 设备的名片，并描述 USB 设备的所有属性和可配置信息，例如类、接口信息和端点信息。如果获得了设备的描述符，则设备通信的类型、目的和参数等是已知的，USB 主机可以对其进行配置，以使通信双方以相同的参数进行工作。

可以使用 USB Composite 类来实现 USB 转多个 VCOM 的功能。Composite 类是一个特殊的 USB 类，可以在 USB 设备中实现多种不同的功能。例如可以实现“鼠标+键盘”功能或“ VCOM +键盘”功能的设备。实际上，USB Composite 类可以实现几乎所有 USB 功能的组合，并且不仅是两个功能的组合，还可以是三个或更多，因此您可以使用 Composite 类来实现两个 CDC 或多个 CDC 的功能。

CDC 类设备包含两个子类接口：CDC 类接口和数据类接口。CDC 设备可以包含零个或多个数据接口。

- CDC 类接口使用标准接口描述符，该描述符需要控制端点和可选的中断 IN 端点。
- 数据类接口需要 Bulk IN 和 Bulk OUT 类型的端点。但是，由于 K32L2B3/K32L2A 上的 16 个端点是双向端点，所以一个端点可以同时实现输入和输出功能，即 Bulk IN 和 Bulk OUT 的功能可以由一个端点实现。

为了支持更多的 VCOM，可以删除 CDC 类接口中的中断 IN 端点，因此，除了一个公共的控制端点外，只需要一个双向端点即可实现 VCOM。也就是说，除了将 EP0 用作控制端点外，其他 15 个端点可以实现 15 个 VCOM，并且本应用笔记已经实现了这种功能。本应用笔记中使用的包含 15 个 CDC 子类的复合类的描述符结构，如 图 1 所示。

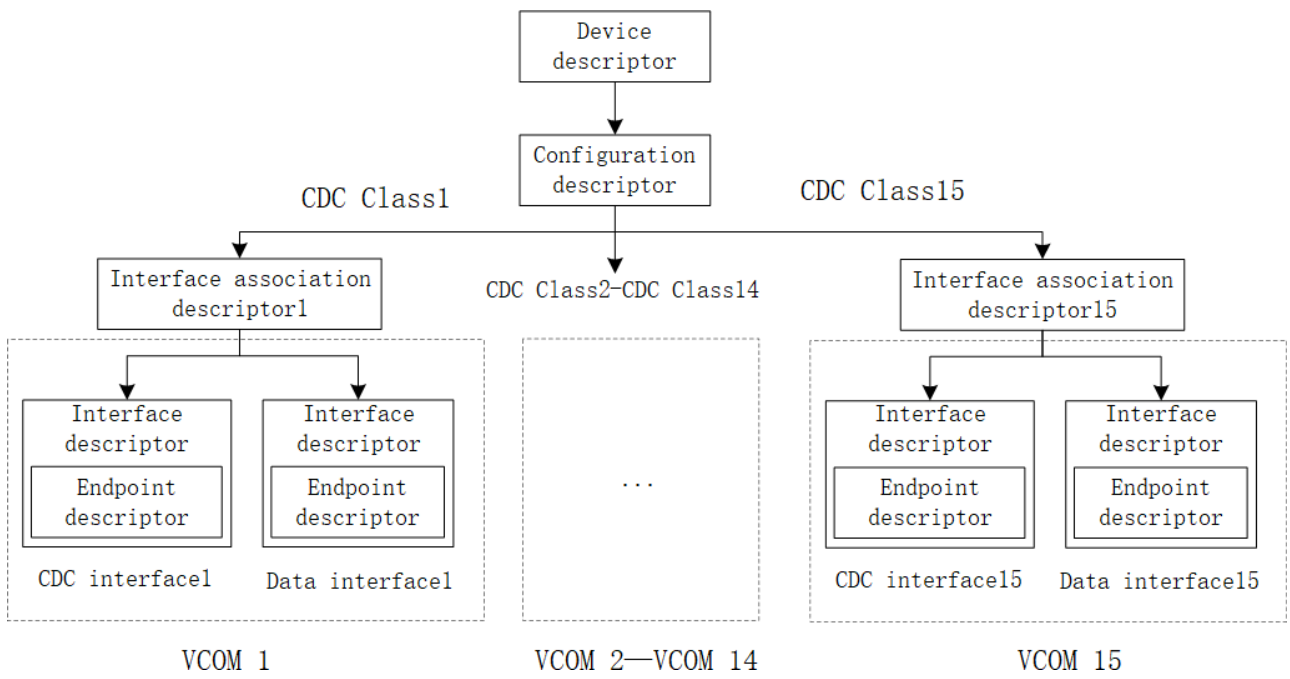

图 1. USB 描述符结构框图

在 图 1 中，接口关联描述符将 CDC 接口与数据接口相关联以描述 VCOM 功能。有关 USB 描述符的详细信息，请参阅 Chapter 9.6, Standard USB Descriptor Definitions, USB Specification 2.0（第 9.6 章：标准 USB 描述符定义和 USB 规范 2.0）以及 SDK 代码。

注意

如果删除了 CDC 接口的中断端点，则无法发送和接收串行端口状态。例如，RS232 中的 RTS 和 DTR 信号无法发送。如果 PC 上的串口调试终端已经打开，则仍然可以发送和接收串口数据。

3 端点使用

如 USB 描述符配置所述，每个 CDC 类需要 1 个双向端点，而 15 个 CDC 类需要 15 个双向端点。EP0 用作控制端点。表 1显示了本应用笔记中 USB 设备端点的用法。

表 1. 端点使用

端点	方向	端点类型	包大小（字节）	是否使用
EP0	输出（OUT）	控制（Control）	64	YES
EP0	输入（IN）	控制（Control）	64	YES
EP1	OUT	CDC1 Bulk	64	YES
EP1	IN	CDC1 Bulk	64	YES
EP2	OUT	CDC2 Bulk	64	YES
EP2	IN	CDC2 Bulk	64	YES
EP3	OUT	CDC3 Bulk	64	YES

下页继续...

表 1. 端点使用 （续上页）

端点	方向	端点类型	包大小（字节）	是否使用
EP3	IN	CDC3 Bulk	64	YES
EP4	OUT	CDC4 Bulk	64	YES
EP4	IN	CDC4 Bulk	64	YES
EP5	OUT	CDC5 Bulk	64	YES
EP5	IN	CDC5 Bulk	64	YES
EP6	OUT	CDC6 Bulk	64	YES
EP6	IN	CDC6 Bulk	64	YES
EP7	OUT	CDC7 Bulk	64	YES
EP7	IN	CDC7 Bulk	64	YES
EP8	OUT	CDC8 Bulk	64	YES
EP8	IN	CDC8 Bulk	64	YES
EP9	OUT	CDC9 Bulk	64	YES
EP9	IN	CDC9 Bulk	64	YES
EP10	OUT	CDC10 Bulk	64	YES
EP10	IN	CDC10 Bulk	64	YES
EP11	OUT	CDC11 Bulk	64	YES
EP11	IN	CDC11 Bulk	64	YES
EP12	OUT	CDC12 Bulk	64	YES
EP12	IN	CDC12 Bulk	64	YES
EP13	OUT	CDC13 Bulk	64	YES
EP13	IN	CDC13 Bulk	64	YES
EP14	OUT	CDC14 Bulk	64	YES
EP14	IN	CDC14 Bulk	64	YES
EP15	OUT	CDC15 Bulk	64	YES
EP15	IN	CDC15 Bulk	64	YES

图 2 显示了 usb_device_descriptor.h 文件中的端点配置。

```
/* Endpoint usage */
#define USB_CDC_VCOM_DIC_BULK_IN_ENDPOINT (1)
#define USB_CDC_VCOM_DIC_BULK_OUT_ENDPOINT (1)
#define USB_CDC_VCOM_DIC_BULK_IN_ENDPOINT_2 (2)
#define USB_CDC_VCOM_DIC_BULK_OUT_ENDPOINT_2 (2)
#define USB_CDC_VCOM_DIC_BULK_IN_ENDPOINT_3 (3)
#define USB_CDC_VCOM_DIC_BULK_OUT_ENDPOINT_3 (3)
#define USB_CDC_VCOM_DIC_BULK_IN_ENDPOINT_4 (4)
#define USB_CDC_VCOM_DIC_BULK_OUT_ENDPOINT_4 (4)
#define USB_CDC_VCOM_DIC_BULK_IN_ENDPOINT_5 (5)
#define USB_CDC_VCOM_DIC_BULK_OUT_ENDPOINT_5 (5)
#define USB_CDC_VCOM_DIC_BULK_IN_ENDPOINT_6 (6)
#define USB_CDC_VCOM_DIC_BULK_OUT_ENDPOINT_6 (6)
#define USB_CDC_VCOM_DIC_BULK_IN_ENDPOINT_7 (7)
#define USB_CDC_VCOM_DIC_BULK_OUT_ENDPOINT_7 (7)
#define USB_CDC_VCOM_DIC_BULK_IN_ENDPOINT_8 (8)
#define USB_CDC_VCOM_DIC_BULK_OUT_ENDPOINT_8 (8)
#define USB_CDC_VCOM_DIC_BULK_IN_ENDPOINT_9 (9)
#define USB_CDC_VCOM_DIC_BULK_OUT_ENDPOINT_9 (9)
#define USB_CDC_VCOM_DIC_BULK_IN_ENDPOINT_10 (10)
#define USB_CDC_VCOM_DIC_BULK_OUT_ENDPOINT_10 (10)
#define USB_CDC_VCOM_DIC_BULK_IN_ENDPOINT_11 (11)
#define USB_CDC_VCOM_DIC_BULK_OUT_ENDPOINT_11 (11)
#define USB_CDC_VCOM_DIC_BULK_IN_ENDPOINT_12 (12)
#define USB_CDC_VCOM_DIC_BULK_OUT_ENDPOINT_12 (12)
#define USB_CDC_VCOM_DIC_BULK_IN_ENDPOINT_13 (13)
#define USB_CDC_VCOM_DIC_BULK_OUT_ENDPOINT_13 (13)
#define USB_CDC_VCOM_DIC_BULK_IN_ENDPOINT_14 (14)
#define USB_CDC_VCOM_DIC_BULK_OUT_ENDPOINT_14 (14)
#define USB_CDC_VCOM_DIC_BULK_IN_ENDPOINT_15 (15)
#define USB_CDC_VCOM_DIC_BULK_OUT_ENDPOINT_15 (15)
```

图 2. 端点配置

4 端点缓冲区配置

每个端点都需要一个缓冲区来存储接收到的数据或要发送的数据。本章介绍 USB 端点缓冲区的配置。

4.1 缓冲区描述符表

为了管理 USB 端点通信，USBFS 在系统内存中实现了缓冲区描述符表（BDT），如 图 3 所示。BDT 位于系统内存中的 512字节对齐的空间上，并由 BDT 页面寄存器指向。每个端点方向都需要两个 8 字节的缓冲区描述符（BD）条目。因此，具有 16 个完全双向端点的系统将需要 512 字节的系统内存来实现 BDT。对于每个端点方向来说，需要两个 BD 入口，一个偶数 BD 入口和一个奇数 BD 入口。BDT 中存储的内容如 表 2 所示。

表 2. 缓冲区描述符格式

31:26	25:16	15.8	7	6	5	4	3	2	1	0
RSVD	BC (10 bits)	RSVD	OWN	DATA0/1	KEEP/ TOK_PID[3]	NINC/ TOK_PID[2]	DTS/ TOK_PID[1]	BDT_STA	0	0

下页继续...

表 2. 缓冲区描述符格式（续上页）

								LL/ TOK_PID[0]		
缓冲区地址（32-Bits）										

EP 缓冲区的地址存储在缓冲区地址字段中。BDT 存储在一个 512 字节对齐的空间中，BDT 的位置可以由 BDT 面寄存器确定：BDT 页面寄存器 1、BDT 页面寄存器 2 和 BDT 页面寄存器 3。这三个寄存器提供了 BDT 地址的第 9 位至第 31 位，其中位 0-位 8 为 0，如 表 3 所示。

表 3. BDT 地址

	BDT 页面寄存器 3	BDT 页面寄存器 2	BDT 页面寄存器 1									
BDT 地址	31 位—24 位	23 位 —16 位	15 位 — 9 位	0	0	0	0	0	0	0	0	0

为了计算进入 BDT 的入口点，将 BDT_PAGE 寄存器与当前端点以及 TX 和 ODD 字段连接起来以形成 32 位地址。表 4 中显示了此地址机制。

表 4. BDT 地址计算

31:24	23:16	15:9	8:5	4	3	2:0
BDT_PAGE_03	BDT_PAGE_02	BDT_PAGE_01[7:1]	Endpoint	TX	ODD	000

表 5. BDT 地址计算字段

字段	描述
BDT_PAGE	控制寄存器块中的 BDT_PAGE 寄存器
ENDPOINT	USB TOKEN 中的 ENDPOINT 字段
TX	1 用于传输传输，0 用于接收传输
ODD	保留在 USBFS SIE 中。它对应于当前使用的缓冲区。缓冲区以乒乓方式使用。

根据 表 4 中几个字段的值，您可以在 BDT 中找到相应的 BD 和相应的 EP Buffer，如 图 3 所示。

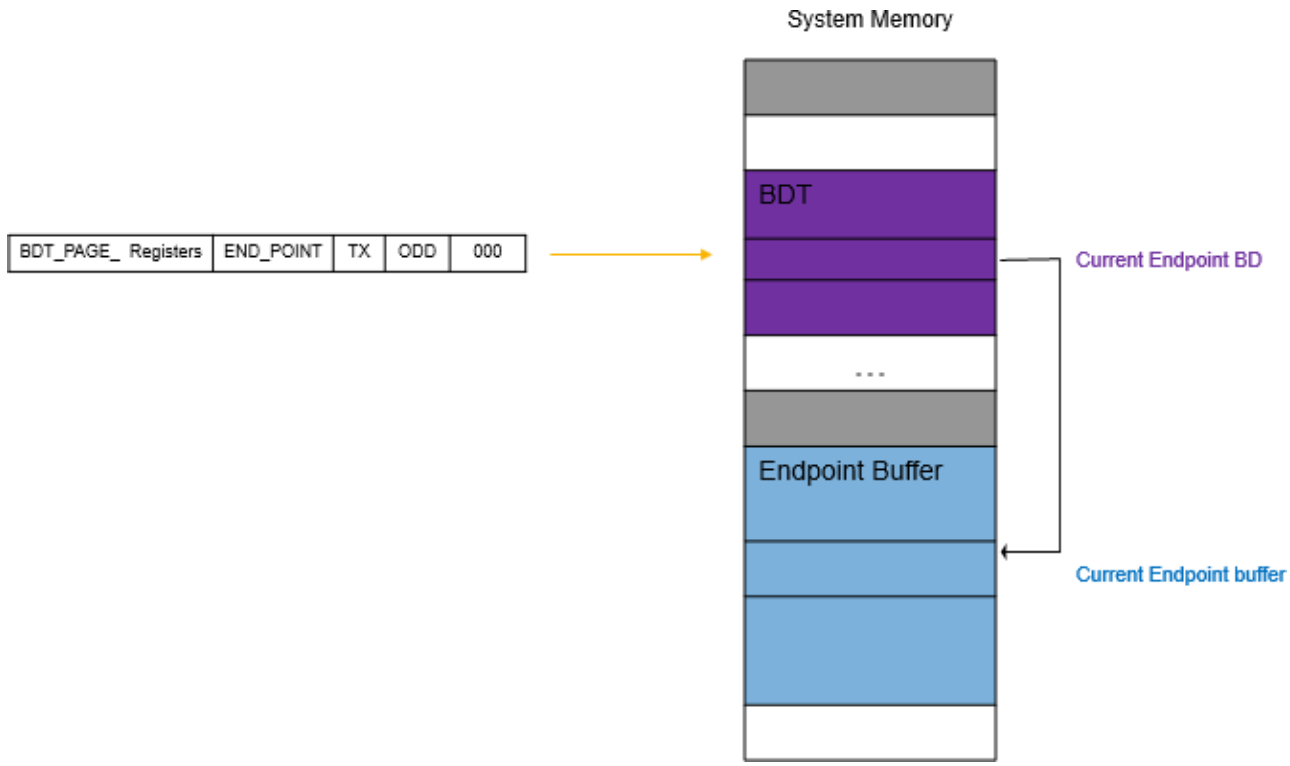

图 3. 缓冲区描述符表

软件应在需要时通过更新 BDT 来管理 USBFS 的缓冲区。这使 USBFS 可以管理数据的发送和接收，而微处理器则可以执行通信开销处理和其他相关功能的应用程序。

4.2 端点缓冲区

控制端点的缓冲区不固定。当接收到 setup 包数据时，EP0 缓冲区是 s_UsbDeviceKhciState.setupPacketBuffer[8*2]；在处理 IN 事务时，EP0 缓冲区通常是一个描述符数组：g_UsbDeviceDescriptor[]，g_UsbDeviceConfigurationDescriptor[]等。非控制端点的缓冲区配置如 表 6 所示。

表 6. EP 缓冲区配置

VCOM 序号	端点	方向	EP 缓冲区
VCOM1	1	OUT	s_currRecvBuf[0][64]
VCOM1	1	IN	s_currSendBuf[0][64]
VCOM2	2	OUT	s_currRecvBuf[1][64]
VCOM2	2	IN	s_currSendBuf[1][64]
VCOM3	3	OUT	s_currRecvBuf[2][64]
VCOM3	3	IN	s_currSendBuf[2][64]
VCOM4	4	OUT	s_currRecvBuf[3][64]

下页继续..

表 6. EP 缓冲区配置（续上页）

VCOM 序号	端点	方向	EP 缓冲区
VCOM4	4	IN	s_currSendBuf[3][64]
VCOM5	5	OUT	s_currRecvBuf[4][64]
VCOM5	5	IN	s_currSendBuf[4][64]
VCOM6	6	OUT	s_currRecvBuf[5][64]
VCOM6	6	IN	s_currSendBuf[5][64]
VCOM7	7	OUT	s_currRecvBuf[6][64]
VCOM7	7	IN	s_currSendBuf[6][64]
VCOM8	8	OUT	s_currRecvBuf[7][64]
VCOM8	8	IN	s_currSendBuf[7][64]
VCOM9	9	OUT	s_currRecvBuf[8][64]
VCOM9	9	IN	s_currSendBuf[8][64]
VCOM10	10	OUT	s_currRecvBuf[9][64]
VCOM10	10	IN	s_currSendBuf[9][64]
VCOM11	11	OUT	s_currRecvBuf[10][64]
VCOM11	11	IN	s_currSendBuf[10][64]
VCOM12	12	OUT	s_currRecvBuf[11][64]
VCOM12	12	IN	s_currSendBuf[11][64]
VCOM13	13	OUT	s_currRecvBuf[12][64]
VCOM13	13	IN	s_currSendBuf[12][64]
VCOM14	14	OUT	s_currRecvBuf[13][64]
VCOM14	14	IN	s_currSendBuf[13][64]
VCOM15	15	OUT	s_currRecvBuf[14][64]
VCOM15	15	IN	s_currSendBuf[14][64]

s_currRecvBuf 和 s_currSendBuf 是两个全局数组，定义如下：

```
USB_DMA_NONINIT_DATA_ALIGN(USB_DATA_ALIGN_SIZE) static uint8_t
s_currRecvBuf[USB_DEVICE_CONFIG_CDC_ACM][DATA_BUFF_SIZE];
USB_DMA_NONINIT_DATA_ALIGN(USB_DATA_ALIGN_SIZE) static uint8_t
s_currSendBuf[USB_DEVICE_CONFIG_CDC_ACM][DATA_BUFF_SIZE];
```

宏 DATA_BUFF_SIZE 的值为 64。

在 USB 设备从主机接收到 SetConfiguration 标准请求之后，在 *USB_DeviceCdcVcomSetConfigure()* 函数中执行 OUT 方向的非控制端点的缓冲区配置，而在使用此 IN 端点发送数据之前，先配置 IN 方向的 EP 缓冲区。

5 软件工作流程图

本应用笔记中使用的代码基于 SDK 中的 *usb_device_composite_cdc_vcom_cdc_vcom_lite* 示例。USB 设备堆栈的体系结构如图 4 所示。

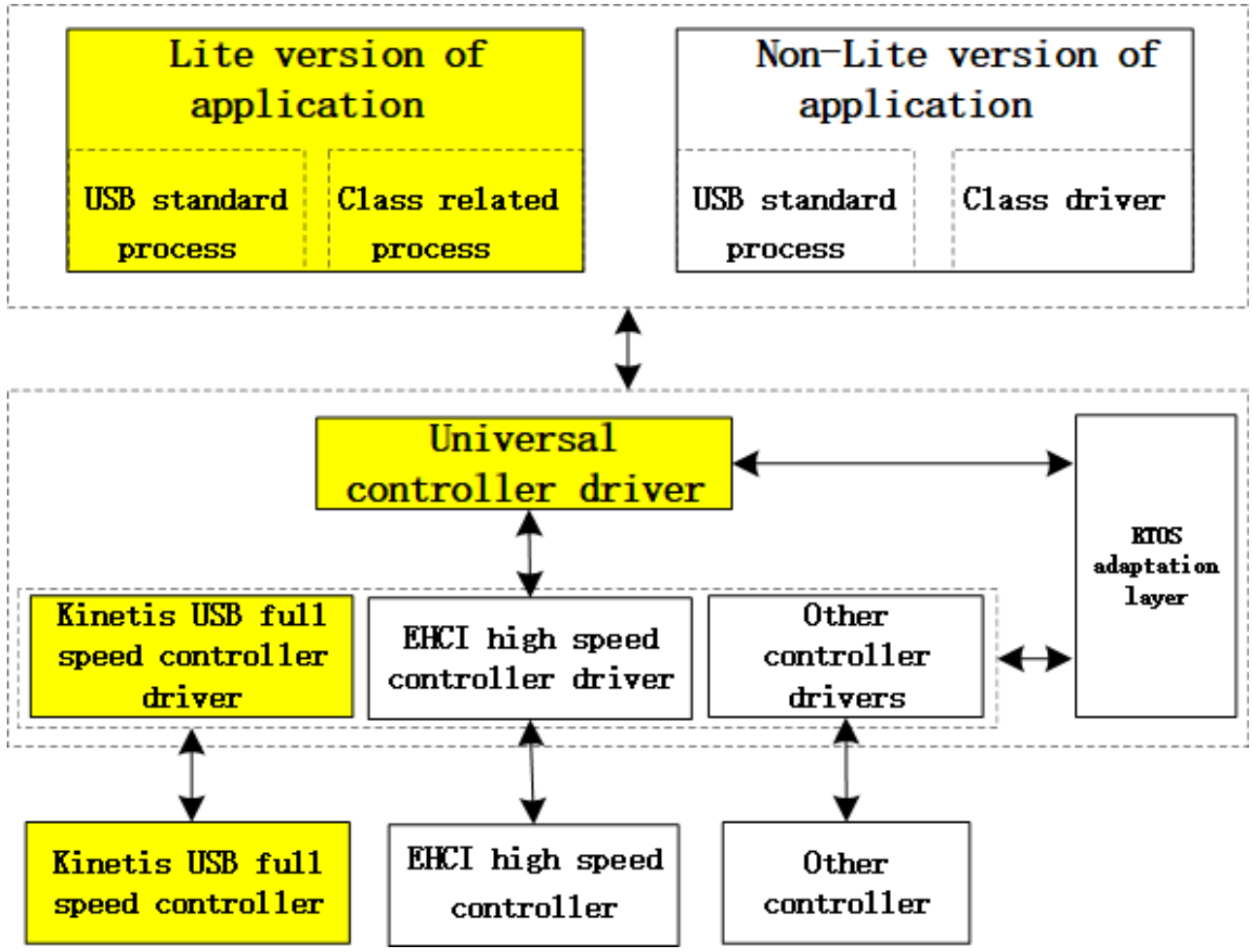

图 4. SDK USB 设备协议栈结构

USB 设备协议栈是该结构的中间部分，由通用控制器驱动程序、特定的控制器驱动程序和 RTOS（实时操作系统）适配层组成。在本应用笔记中，使用该应用的精简版本结构图标有黄色的部分。

- 通用控制器驱动程序—为应用程序提供统一的界面，用户无需关心不同硬件控制器的详细信息。
- 特定的控制器驱动程序—USB 控制器的类型很多，每种类型都实现一个特定的驱动程序。K32L2 系列 MCU 的 USB 设备控制器是 Kinetis 全速 USB 设备控制器。
- RTOS 适配层—该 USB 协议栈不仅可以在不同的 RTOS 环境中运行，而且还可以在裸板环境中运行。本应用笔记中未使用 RTOS。

项目中源代码和 USB 协议栈体系结构之间的对应关系如图 5 所示。

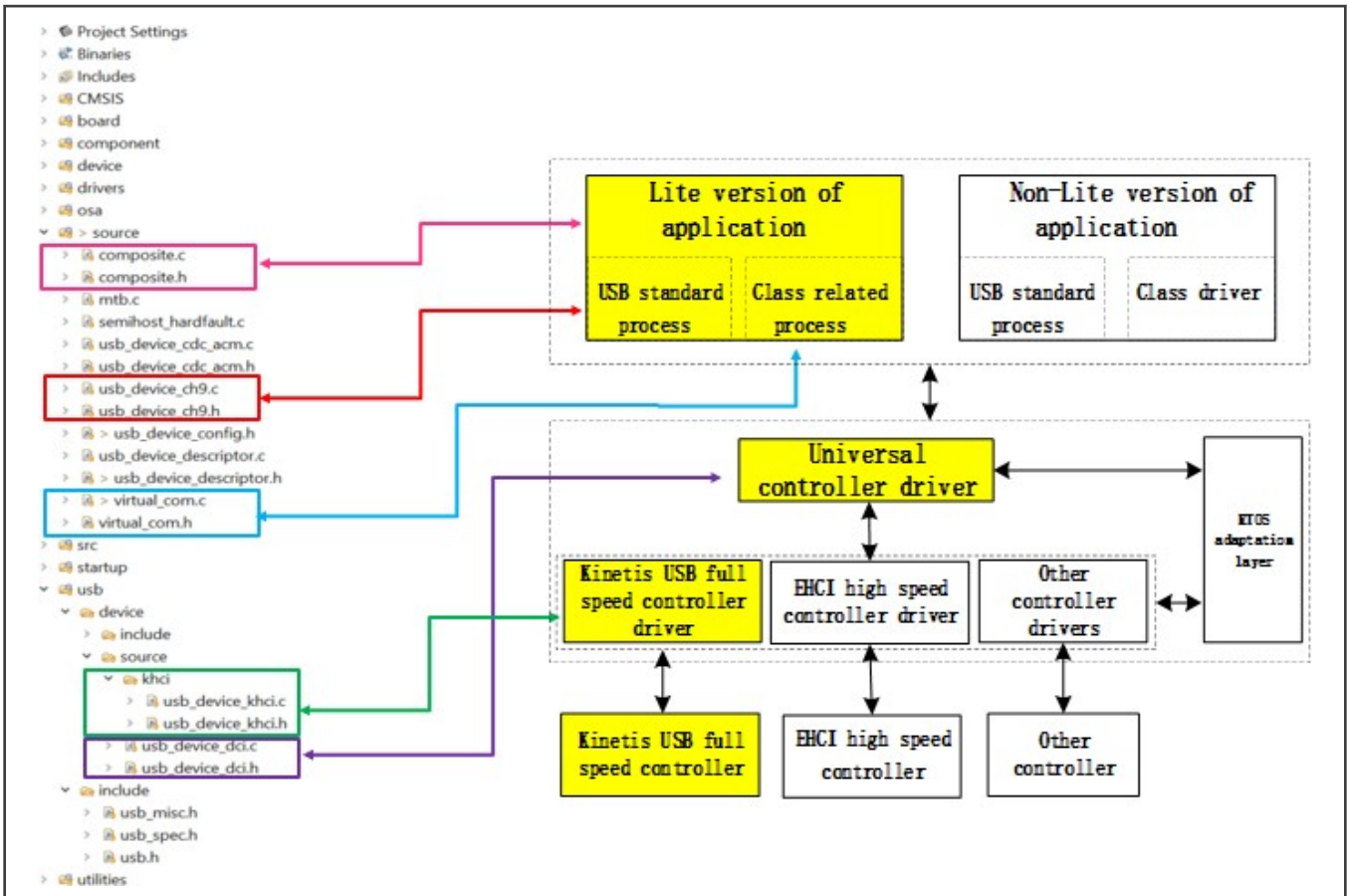

图 5. 源代码和 USB 协议栈体系结构之间的对应关系

设备协议栈的基本工作流程取决于回调函数和函数调用。 回调函数将所有状态更改和设备协议栈的数据请求通知给应用程序。

设备协议栈中有两种回调函数：

- 设备回调函数：它将设备堆栈的状态通知给应用程序。
- 端点回调函数：它将相应端点的数据传输结果通知给应用程序。控制端点回调函数处理所有 USB 标准请求和类请求。图 6 简要描述了回调函数的处理过程。

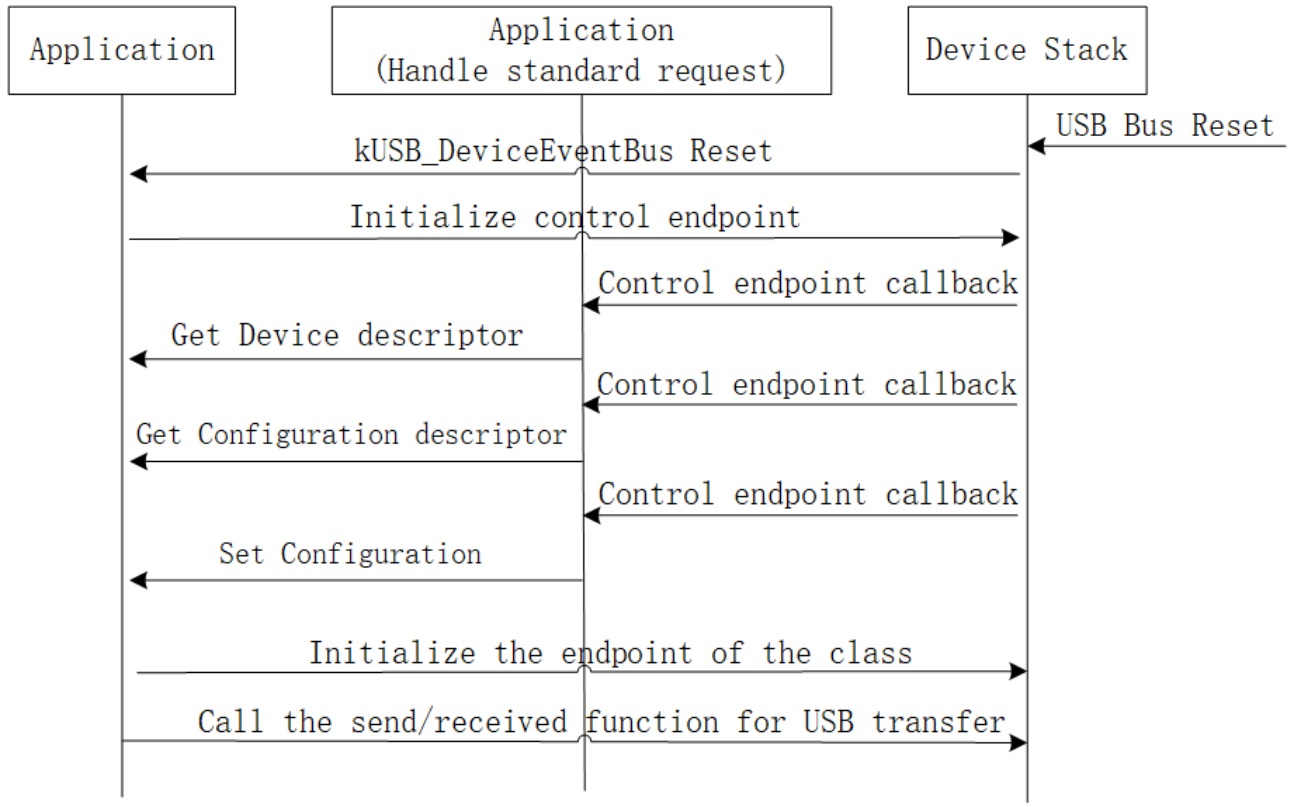

图 6. 回调函数处理过程

当 USB 主机识别出 USB 设备已插入 USB 接口时，它将启动枚举进程。USB 枚举的本质是 USB 主机获取 USB 设备的参数信息并配置可配置参数的过程。USB 枚举进程是在 USB 中断服务功能中完成的。USB 中断服务功能的处理流程如 图 7 所示。

5.1 USB 中断服务功能流程图

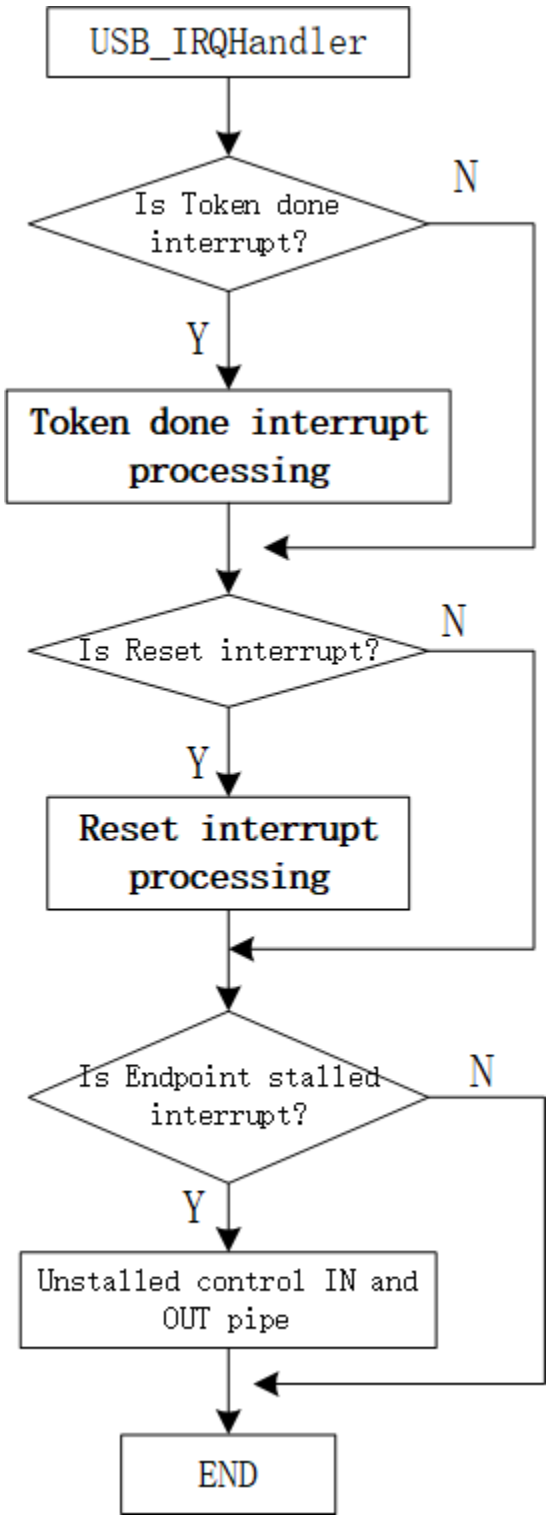

图 7. USB 中断服务功能的主要流程图

当 USB 发生中断时，中断服务功能将确定该中断类型：令牌完成中断、复位中断或端点停顿中断，然后转到相应的操作。

5.1.1 复位中断进程

如果 USB 发生复位中断，则程序执行 图 8 所示的流程。

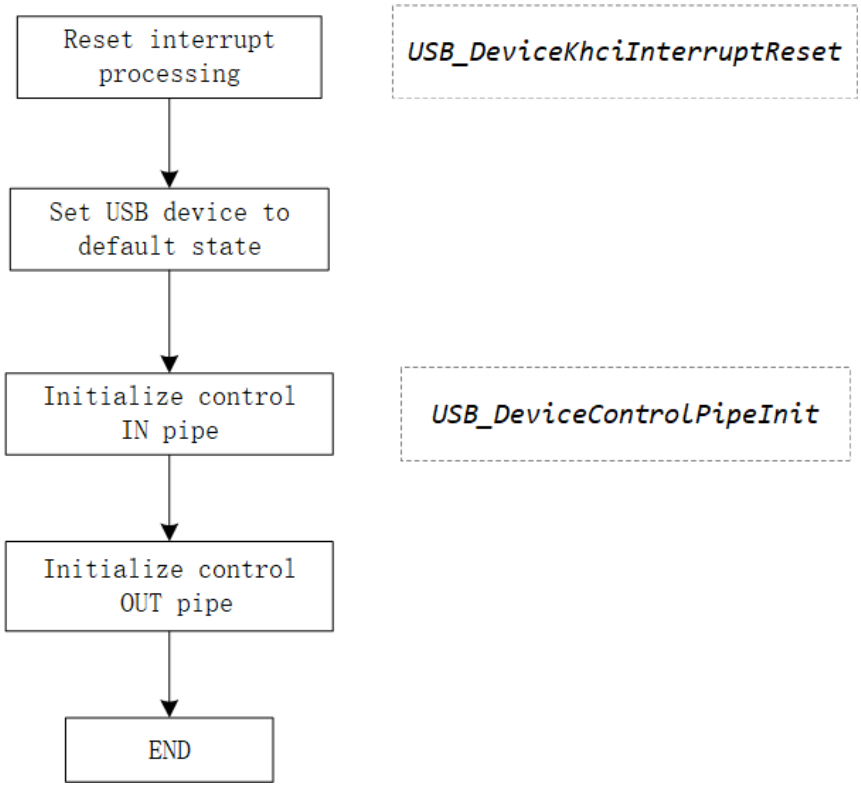

图 8. USB 设备复位过程

复位中断的主要操作是将 USB 设备设置为默认状态，并初始化控制 IN 管道和 OUT 管道。

5.1.2 令牌完成中断进程

如果发生令牌完成中断，项目执行 图 9 所示的进程。

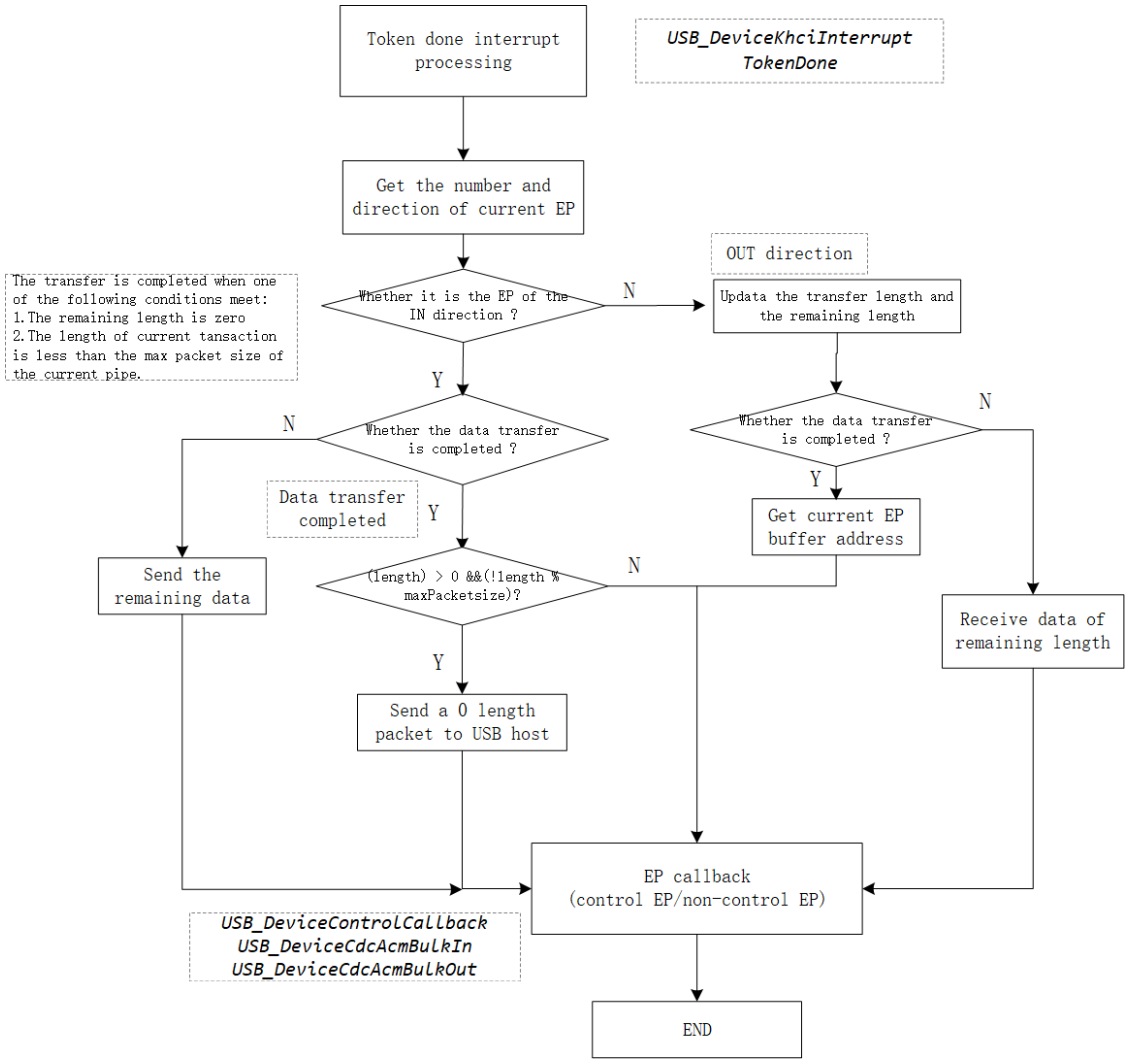

图 9. 令牌完成中断过程

端点回调函数包括两种类型：控制端点回调函数和非控制端点回调函数。如果当前端点是控制端点，则调用控制端点回调函数；如果当前端点是非控制端点，则调用非控制端点回调函数。控制端点的回调函数实现对标准和类请求的处理； 非控制端点回调函数实现数据传输功能，并将 Bluk OUT 端点接收的数据通过 Bluk IN 端点传输到主机。本应用笔记中的 15 个 VCOM 的非控制端点回调函数是相同的。IN 方向上的端点回调函数是 *USB_DeviceCdcAcmBulkIn()*，OUT 方向上的端点回调函数是 *USB_DeviceCdcAcmBulkOut()*，即 15 个 VCOM 实现的功能是相同的：将从 USB 主机接收的数据传输到 USB 主机。

5.2 USB 设备请求

标准请求和类请求的处理在控制端点的回调函数中执行。USB 主机通过向设备发送设备请求（Device Request）来获取 USB 设备的信息并在 USB 设备上进行相关配置。设备请求由长度为 8 个字节的 Setup 数据指定，其方向始终是从 USB 主机到 USB 设备。常用的标准请求如 表 7 所示。

表 7. 常用的标准请求

标准请求	方向
SetAddress	OUT

下页继续...

表 7. 常用的标准请求（续上页）

标准请求	方向
SetConfiguration	OUT
GetDescriptor(Device/Configuration/String)	IN

CDC 类设备除标准请求外，还支持某些类请求。USB 主机获得 CDC 类设备的各种描述符信息后，还会向 USB 设备发送一些类请求。常用的类请求如 表 8 所示。

表 8. 常用的类请求

标准请求	方向
SetLineCoding	OUT
SetControlLineState	OUT
GetLineCoding	IN

注意

CDC 接口中未使用中断 IN 端点，因此无法处理 SetControlLineState 类请求，请忽略此类请求。

GetLineCoding 请求是主机获取串行端口属性的请求，包括波特率、停止位、校验类型和数据位数。表 9 显示了 GetLineCoding请求的结构。

表 9. GetLineCoding 请求结构

bmRequestType	bRequestCode	wValue	wIndex	wLength	数据
10100001B	GET_LINE_CODING	Zero	接口	结构尺寸	线编码结构

线编码结构的内容如 表 10 所示。

表 10. 线编码结构

抵消	字段	大小/字节	描述
0	dwDTERate	4	数据终端速率，以每秒位数为单位
4	bCharFormat	1	停止位 0：1 停止位 1：1.5 停止位 2：2 停止
5	bParityType	1	等价 0：无 3：标记 1：奇数 4：空格

下页继续...

表 10. 线编码结构 (continued)

抵消	字段	大小/字节	描述
			2: 偶数
6	bDataBits	1	数据位（5，6，7，8 或 16）

SetLineCoding 是一个输出类请求，它对应于 GetLineCoding。USB 主机使用此命令来设置设备的串行端口属性，其数据结构与 GetLineCoding 相同。USB 主机完成这些标准请求和类请求后，枚举过程结束，USB 主机将识别出多个 VCOM。然后，PC 上的串行端口调试终端可以与 K32L2 系列 MCU 的 USB 设备通信。

6 关键步骤

6.1 如何扩展支持的 VCOM 的数量

本节以 SDK 代码 *dev_composite_cdc_vcom_cdc_vcom_lite_bm* 为例，说明如何基于两个 VCOM 扩展到三个 VCOM。具体步骤如下：

1. 修改 usb_device_config.h 文件
   - 将宏 USB_DEVICE_CONFIG_CDC_ACM 的值由 2 修改为 3

```
#define USB_DEVICE_CONFIG_CDC_ACM (2U) to
#define USB_DEVICE_CONFIG_CDC_ACM (3U)
```

   - 修改 USB_DEVICE_CONFIG_ENDPOINTS 宏的值

```
#define USB_DEVICE_CONFIG_ENDPOINTS (5U) to
/* Values that meet functional requirements are ok */
#define USB_DEVICE_CONFIG_ENDPOINTS (8U)
```

2. 修改 usb_device_descriptor.h 文件
   - 将宏 USB_INTERFACE_COUNT 的值由 4 修改为 6

```
#define USB_INTERFACE_COUNT (4) to
#define USB_INTERFACE_COUNT (6)
```

   - 为 VCOM3 增加一些宏定义

```
#define USB_CDC_VCOM_INTERFACE_COUNT_3 (2)
#define USB_CDC_VCOM_CIC_INTERFACE_INDEX_3 (4)
#define USB_CDC_VCOM_DIC_INTERFACE_INDEX_3 (5)
/* If the CDC interface does not require an Interrupt IN endpoint, then this macro is 0 */
#define USB_CDC_VCOM_CIC_ENDPOINT_COUNT_3 (1)
/* If the CDC interface does not require an Interrupt IN endpoint, you do not need to define
this macro*/
#define USB_CDC_VCOM_CIC_INTERRUPT_IN_ENDPOINT_3 (7)
#define USB_CDC_VCOM_DIC_ENDPOINT_COUNT_3 (2)
#define USB_CDC_VCOM_DIC_BULK_IN_ENDPOINT_3 (3)
#define USB_CDC_VCOM_DIC_BULK_OUT_ENDPOINT_3 (3)
#define HS_CDC_VCOM_INTERRUPT_IN_PACKET_SIZE_3 (16)
#define FS_CDC_VCOM_INTERRUPT_IN_PACKET_SIZE_3 (16)
#define HS_CDC_VCOM_INTERRUPT_IN_INTERVAL_3 (0x07)
#define FS_CDC_VCOM_INTERRUPT_IN_INTERVAL_3 (0x08)
#define HS_CDC_VCOM_BULK_IN_PACKET_SIZE_3 (512)
#define FS_CDC_VCOM_BULK_IN_PACKET_SIZE_3 (64)
```

```
#define HS_CDC_VCOM_BULK_OUT_PACKET_SIZE_3 (512)
#define FS_CDC_VCOM_BULK_OUT_PACKET_SIZE_3 (64)
```

3. 修改 usb_device_descriptor.c 文件
   - 修改 g_interface 数组

```
uint8_t g_interface[USB_CDC_VCOM_INTERFACE_COUNT + USB_CDC_VCOM_INTERFACE_COUNT_2
+ USB_CDC_VCOM_INTERFACE_COUNT_3];
```

   - 修改配置描述符数组 g_UsbDeviceConfigurationDescriptor：添加 CDC3 的接口描述符和端点描述符。有关详细信息，请参阅 AN12597SW。
4. 修改 virtual_com.c 文件
   - 扩展 s_lineCoding, s_abstractState, s_countryCode, s_usbCdcAcmInfo 数组。
   - 修改 USB_DeviceCdcVcomSetConfigure()函数以增加 CDC3 中非控制端点的初始化。

注意

多个 VCOM 中非控制端点的回调函数在本应用笔记中相同。如果要修改应用程序层的功能，则可以在此处使用不同功能的回调函数。

6.2 如何在 CDC 接口中删除 Interrupt IN 端点

为了使 USB 设备支持更多的 VCOM，可以删除 CDC 接口中的 Interrupt IN 端点。具体的实现步骤如下：

1. 修改 usb_device_config.h 文件
   - 增加宏 USB_CDC_CIC_INTERRUPT_IN_ENDPOINT_ENABLE 的定义

```
#define USB_CDC_CIC_INTERRUPT_IN_ENDPOINT_ENABLE (0U)
```

2. 修改 usb_device_descriptor.h 文件
   - 修改宏 USB_CDC_VCOM_CIC_ENDPOINT_COUNT 的值为 0

```
#define USB_CDC_VCOM_CIC_ENDPOINT_COUNT ( 0 )
```

   - 然后根据 USB_CDC_CIC_INTERRUPT_IN_ENDPOINT_ENABLE 宏的值，在 CDC 接口中隐藏 Interrupt IN 端点的定义。

```
#if USB_CDC_CIC_INTERRUPT_IN_ENDPOINT_ENABLE
/* No need interrupt IN endpoint for CIC interface */
#define USB_CDC_VCOM_CIC_INTERRUPT_IN_ENDPOINT (0)
#define USB_CDC_VCOM_CIC_INTERRUPT_IN_ENDPOINT_2 (0)
#define USB_CDC_VCOM_CIC_INTERRUPT_IN_ENDPOINT_3 (0)
#define USB_CDC_VCOM_CIC_INTERRUPT_IN_ENDPOINT_4 (0)
#define USB_CDC_VCOM_CIC_INTERRUPT_IN_ENDPOINT_5 (0)
#define USB_CDC_VCOM_CIC_INTERRUPT_IN_ENDPOINT_6 (0)
#define USB_CDC_VCOM_CIC_INTERRUPT_IN_ENDPOINT_7 (0)
#define USB_CDC_VCOM_CIC_INTERRUPT_IN_ENDPOINT_8 (0)
#define USB_CDC_VCOM_CIC_INTERRUPT_IN_ENDPOINT_9 (0)
#define USB_CDC_VCOM_CIC_INTERRUPT_IN_ENDPOINT_10 (0)
#define USB_CDC_VCOM_CIC_INTERRUPT_IN_ENDPOINT_11 (0)
```

3. 修改 usb_device_descriptor.c 文件

- 修改 g_UsbDeviceConfigurationDescriptor 配置描述符数组中的数组长度字段

```
USB_SHORT_GET_LOW(USB_DESCRIPTOR_LENGTH_CONFIGURE +
                    (USB_IAD_DESC_SIZE + USB_DESCRIPTOR_LENGTH_INTERFACE +
USB_DESCRIPTOR_LENGTH_CDC_HEADER_FUNC +
                     USB_DESCRIPTOR_LENGTH_CDC_CALL_MANAG +
USB_DESCRIPTOR_LENGTH_CDC_ABSTRACT +
                      USB_DESCRIPTOR_LENGTH_CDC_UNION_FUNC + 0 +
                      USB_DESCRIPTOR_LENGTH_INTERFACE + USB_DESCRIPTOR_LENGTH_ENDPOINT +
                      USB_DESCRIPTOR_LENGTH_ENDPOINT) * USB_DEVICE_CONFIG_CDC_ACM),
USB_SHORT_GET_HIGH(USB_DESCRIPTOR_LENGTH_CONFIGURE +
                    (USB_IAD_DESC_SIZE + USB_DESCRIPTOR_LENGTH_INTERFACE +
USB_DESCRIPTOR_LENGTH_CDC_HEADER_FUNC +
                      USB_DESCRIPTOR_LENGTH_CDC_CALL_MANAG +
USB_DESCRIPTOR_LENGTH_CDC_ABSTRACT +
                       USB_DESCRIPTOR_LENGTH_CDC_UNION_FUNC + 0 +
                       USB_DESCRIPTOR_LENGTH_INTERFACE + USB_DESCRIPTOR_LENGTH_ENDPOINT +
                       USB_DESCRIPTOR_LENGTH_ENDPOINT) *
USB_DEVICE_CONFIG_CDC_ACM),
```

- 删除所有 CDC 接口中的中断端点描述符

```
#if USB_CDC_CIC_INTERRUPT_IN_ENDPOINT_ENABLE
/*Notification Endpoint descriptor */
        USB_DESCRIPTOR_LENGTH_ENDPOINT,
        USB_DESCRIPTOR_TYPE_ENDPOINT,
        USB_CDC_VCOM_CIC_INTERRUPT_IN_ENDPOINT |(USB_IN << 7U), USB_ENDPOINT_INTERRUPT,
        USB_SHORT_GET_LOW(FS_CDC_VCOM_INTERRUPT_IN_PACKET_SIZE),
        USB_SHORT_GET_HIGH(FS_CDC_VCOM_INTERRUPT_IN_PACKET_SIZE),
        FS_CDC_VCOM_INTERRUPT_IN_INTERVAL,
        #endif
```

4. 修改 virtual_com.c 文件

- 在 USB_DeviceCdcVcomClassRequest()函数中修改 SetControlLineState 类请求的处理。

```
    if (0 == vcomInstance->hasSentState)
   {
#if USB_CDC_CIC_INTERRUPT_IN_ENDPOINT_ENABLE
        error = USB_DeviceSendRequest(handle, vcomInstance->interruptEndpoint,
        acmInfo->serialStateBuf, len);
        if (kStatus_USB_Success != error)
         {
            usb_echo("kUSB_DeviceCdcEventSetControlLineState error!");
          }
#endif
      vcomInstance->hasSentState = 1;
}
```

- 增加数组 g_CdcVcomInterruptInPackageSzie，g_CdcVcomInterruptInInterval，g_CdcVcomInterruptInEndpoint，g_CdcVcomDicBulkInEndpoint，g_CdcVcomDicBulkInEndpoint，g_CdcVcomDicBulkOutEndpoint，g_CdcVcomDicInterfaceIndex，g_CdcVcomCicInterfaceIndex，g_CdcVcomBulkInPacketSize，g_CdcVcomBulkOutPacketSize。

注意

这些数组仅用于简化 USB_DeviceCdcVcomSetConfigure()函数中每个端点的初始化。

5. 修改 USB_DeviceCdcVcomSetConfigure()函数中每个 CDC 类中的中断端点的初始化过程。

```
  if (USB_COMPOSITE_CONFIGURE_INDEX == configure)
  {
  for ( uint8_t i = 0; i < USB_DEVICE_CONFIG_CDC_ACM; i++)
  {
      /****** Initiailizecdc endpoint *****/
      g_deviceComposite-> cdcVcom [i]. attach = 1;
#if USB_CDC_CIC_INTERRUPT_IN_ENDPOINT_ENABLE
       /* Initiailizeendpoint for interrupt pipe */
       epCallback.callbackFn = USB_DeviceCdcAcmInterruptIn;
       epCallback.callbackParam = (void

       *)&g_deviceComposite->cdcVcom[i].communicationInterfaceNumber;
       epInitStruct.zlt = 0;

       epInitStruct.transferType = USB_ENDPOINT_INTERRUPT;
       epInitStruct.endpointAddress =
       g_CdcVcomCicInterruptInEndpoint[i] | (USB_IN <<

       USB_DESCRIPTOR_ENDPOINT_ADDRESS_DIRECTION_SHIFT);
       epInitStruct.maxPacketSize = FS_CDC_VCOM_INTERRUPT_IN_PACKET_SIZE;
       epInitStruct.interval = FS_CDC_VCOM_INTERRUPT_IN_INTERVAL;
       g_deviceComposite->cdcVcom[i].interruptEndpoint =

       g_CdcVcomCicInterruptInEndpoint[i];
       g_deviceComposite->cdcVcom[i].interruptEndpointMaxPacketSize =

       epInitStruct.maxPacketSize;
       g_deviceComposite->cdcVcom[i].communicationInterfaceNumber =
       g_CdcVcomCicInterfaceIndex[i];
       USB_DeviceInitEndpoint(handle, &epInitStruct, &epCallback);
#else
       g_deviceComposite-> cdcVcom [i]. communicationInterfaceNumber =
       g_CdcVcomCicInterfaceIndex[i];
#endif
```

注意

不要删除以下内容：

g_deviceComposite->cdcVcom[i].communicationInterfaceNumber = USB_CDC_VCOM_CIC_INTERFACE_INDEX;。完整代码参见 AN12597SW。

6.3 代码优化

如何扩展支持的 VCOM 的数量和如何在 CDC 接口中删除 Interrupt IN 端点中的步骤有些复杂。用户需要对 USB 协议栈有一些了解。如果存在步骤错误，则可能导致功能失败。对于每个其他的 VCOM，用户需要重复如何扩展支持的 VCOM 的数量中的步骤。为了使 VCOM 功能的操作过程更加方便，基于如何扩展支持的 VCOM 的数量和如何在 CDC 接口中删除 Interrupt IN 端点对代码进行了一些优化，这使用户可以轻松地配置 VCOM 的数量，并通过宏定义决定是否使用 CIC 接口的中断 IN 端点。

6.3.1 使用循环代替列表

举个例子，virtual_com.c 文件中的 USB_DeviceCdcVcomConfigureEndpointStatus () 函数：

```
usb_status_t USB_DeviceCdcVcomConfigureEndpointStatus(usb_device_handle handle, uint8_t ep, uint8_t
status)
{
    usb_status_t error = kStatus_USB_Error;
```

```
    if (status)
    {
        if ((USB_CDC_VCOM_DIC_BULK_IN_ENDPOINT == (ep & USB_ENDPOINT_NUMBER_MASK)) &&
            (ep & USB_DESCRIPTOR_ENDPOINT_ADDRESS_DIRECTION_MASK))
    {
        error = USB_DeviceStallEndpoint(handle, ep);
    }
    else if ((USB_CDC_VCOM_DIC_BULK_OUT_ENDPOINT == (ep & USB_ENDPOINT_NUMBER_MASK)) &&
            (!(ep & USB_DESCRIPTOR_ENDPOINT_ADDRESS_DIRECTION_MASK)))
    {
        error = USB_DeviceStallEndpoint(handle, ep);
    }
    else if ((USB_CDC_VCOM_DIC_BULK_IN_ENDPOINT_2 == (ep & USB_ENDPOINT_NUMBER_MASK)) &&
            (ep & USB_DESCRIPTOR_ENDPOINT_ADDRESS_DIRECTION_MASK))
    {
        error = USB_DeviceStallEndpoint(handle, ep);
    }
    else if ((USB_CDC_VCOM_DIC_BULK_OUT_ENDPOINT_2 == (ep & USB_ENDPOINT_NUMBER_MASK)) &&
    (          !(ep & USB_DESCRIPTOR_ENDPOINT_ADDRESS_DIRECTION_MASK)))
    {
        error = USB_DeviceStallEndpoint(handle, ep);
    }
    else
    {
    }
    }
    else
    {
        if ((USB_CDC_VCOM_DIC_BULK_IN_ENDPOINT == (ep & USB_ENDPOINT_NUMBER_MASK)) &&
            (ep & USB_DESCRIPTOR_ENDPOINT_ADDRESS_DIRECTION_MASK))
    {
        error = USB_DeviceUnstallEndpoint(handle, ep);
    }
    else if ((USB_CDC_VCOM_DIC_BULK_OUT_ENDPOINT == (ep & USB_ENDPOINT_NUMBER_MASK)) &&
            (!(ep & USB_DESCRIPTOR_ENDPOINT_ADDRESS_DIRECTION_MASK)))
    {
        error = USB_DeviceUnstallEndpoint(handle, ep);
    }
    else if ((USB_CDC_VCOM_DIC_BULK_IN_ENDPOINT_2 == (ep & USB_ENDPOINT_NUMBER_MASK)) &&
            (ep & USB_DESCRIPTOR_ENDPOINT_ADDRESS_DIRECTION_MASK))
    {
        error = USB_DeviceUnstallEndpoint(handle, ep);
    }
    else if ((USB_CDC_VCOM_DIC_BULK_OUT_ENDPOINT_2 == (ep & USB_ENDPOINT_NUMBER_MASK)) &&
            (!(ep & USB_DESCRIPTOR_ENDPOINT_ADDRESS_DIRECTION_MASK)))
    {
        error = USB_DeviceUnstallEndpoint(handle, ep);
    }
    else
    {
    }
   }
      return error;
   }

usb_status_t USB_DeviceCdcVcomConfigureEndpointStatus(usb_device_handle handle, uint8_t ep,
uint8_t status)
{
    usb_status_t error = kStatus_USB_Error;
    uint8_t i;
```

```
    if (status)
    {
        for(i = 0; i < USB_DEVICE_CONFIG_CDC_ACM; i++)
        {
            if ((g_CdcVcomDicBulkInEndpoint[i] == (ep & USB_ENDPOINT_NUMBER_MASK)) &&
                (ep & USB_DESCRIPTOR_ENDPOINT_ADDRESS_DIRECTION_MASK))
            {
                error = USB_DeviceStallEndpoint(handle, ep);
            }
            else if ((g_CdcVcomDicBulkOutEndpoint[i] == (ep & USB_ENDPOINT_NUMBER_MASK)) &&
                    ep & USB_DESCRIPTOR_ENDPOINT_ADDRESS_DIRECTION_MASK))
            {
                error = USB_DeviceStallEndpoint(handle, ep);
            }
        }
    }
    else
    {
        for(i = 0; i < USB_DEVICE_CONFIG_CDC_ACM; i++)
        {
                if ((g_CdcVcomDicBulkInEndpoint[i] == (ep & USB_ENDPOINT_NUMBER_MASK)) &&
                    (ep & USB_DESCRIPTOR_ENDPOINT_ADDRESS_DIRECTION_MASK))
                {
                    error = USB_DeviceUnstallEndpoint(handle, ep);
                }
                else if ((g_CdcVcomDicBulkOutEndpoint[i] == (ep & USB_ENDPOINT_NUMBER_MASK)) &&
                            (ep & USB_DESCRIPTOR_ENDPOINT_ADDRESS_DIRECTION_MASK))
        {
            error = USB_DeviceUnstallEndpoint(handle, ep);
        }
    }
}
return error;
}
```

实际上，有很多代码可以被循环替代，这是使应用程序自动合成类的关键一步。其他相似的代码未在此处列出，有关更多详细信息，请参见 AN12597SW。

6.3.2 删除非必需的设置

以下设置可以合并，因为在大多数情况下，用户根本不需要调整这些参数。即使在某些特殊情况下某些用户需要调整参数，运行时初始化也支持修改参数。

```
/* Packet size. */
#define HS_CDC_VCOM_INTERRUPT_IN_PACKET_SIZE (16)
#define FS_CDC_VCOM_INTERRUPT_IN_PACKET_SIZE (16)
#define HS_CDC_VCOM_INTERRUPT_IN_INTERVAL (0x07) /* 2^(7-1) = 8ms */
#define FS_CDC_VCOM_INTERRUPT_IN_INTERVAL (0x08)
/* Packet size. */
#define HS_CDC_VCOM_INTERRUPT_IN_PACKET_SIZE_2 (16)
#define FS_CDC_VCOM_INTERRUPT_IN_PACKET_SIZE_2 (16)
#define HS_CDC_VCOM_INTERRUPT_IN_INTERVAL_2 (0x07) /* 2^(7-1) = 8ms */
#define FS_CDC_VCOM_INTERRUPT_IN_INTERVAL_2 (0x08)
#define HS_CDC_VCOM_INTERRUPT_IN_PACKET_SIZE_3 (16)
#define FS_CDC_VCOM_INTERRUPT_IN_PACKET_SIZE_3 (16)
#define HS_CDC_VCOM_INTERRUPT_IN_INTERVAL_3 (0x07) /* 2^(7-1) = 8ms */
#define FS_CDC_VCOM_INTERRUPT_IN_INTERVAL_3 (0x08)
#define HS_CDC_VCOM_INTERRUPT_IN_PACKET_SIZE_4 (16)
#define FS_CDC_VCOM_INTERRUPT_IN_PACKET_SIZE_4 (16)
```

```
#define HS_CDC_VCOM_INTERRUPT_IN_INTERVAL_4 (0x07) /* 2^(7-1) = 8ms */
#define FS_CDC_VCOM_INTERRUPT_IN_INTERVAL_4 (0x08)
#define HS_CDC_VCOM_INTERRUPT_IN_PACKET_SIZE_5 (16)
#define FS_CDC_VCOM_INTERRUPT_IN_PACKET_SIZE_5 (16)
#define HS_CDC_VCOM_INTERRUPT_IN_INTERVAL_5 (0x07) /* 2^(7-1) = 8ms */
#define FS_CDC_VCOM_INTERRUPT_IN_INTERVAL_5 (0x08)
...
```

合并后，以上代码变为：

```
/* Packet size. */
#define HS_CDC_VCOM_INTERRUPT_IN_PACKET_SIZE (16)
#define FS_CDC_VCOM_INTERRUPT_IN_PACKET_SIZE (16)
#define HS_CDC_VCOM_INTERRUPT_IN_INTERVAL (0x07) /* 2^(7-1) = 8ms */
#define FS_CDC_VCOM_INTERRUPT_IN_INTERVAL (0x08)
```

其他相似的代码未在此处列出，有关更多详细信息，请参考 AN12597SW。

6.3.3 使用宏以避免手动修改代码

```
#define USB_INTERFACE_COUNT (30)

#define USB_INTERFACE_COUNT (2*USB_DEVICE_CONFIG_CDC_ACM)

#define USB_DEVICE_CONFIG_ENDPOINTS (16U)

#define USB_DEVICE_CONFIG_ENDPOINTS (1+2*USB_DEVICE_CONFIG_CDC_ACM)
```

其他相似的代码未在此处列出，有关更多详细信息，请参考 AN12597SW。

6.3.4 使用运行过程中初始化代替静态初始化

6.3.4.1 动态分配接口序号

```
void USB_CdcVcomInterfaceIndexInit(void)
{
    uint8_t i;
    for(i = 0; i < USB_DEVICE_CONFIG_CDC_ACM; i++)
    {
        g_CdcVcomCicInterfaceIndex[i] = 0 + i*2;
        g_CdcVcomDicInterfaceIndex[i] = 1 + i*2;
    }
}
```

6.3.4.2 动态分配端点序号

```
void USB_CdcVcomEndpointInit(void)
{
    uint8_t i;
    for(i = 0; i < USB_DEVICE_CONFIG_CDC_ACM; i++)
    {
        #if USB_CDC_CIC_INTERRUPT_IN_ENDPOINT_ENABLE

            g_CdcVcomCicInterruptInEndpoint[i] = 1 + i*2;
            g_CdcVcomDicBulkInEndpoint[i] = 2 + i*2;
```

```
            g_CdcVcomDicBulkOutEndpoint[i] = 2 + i*2;
        #else
           g_CdcVcomDicBulkInEndpoint[i] = 1 + i*1;
            g_CdcVcomDicBulkOutEndpoint[i] = 1 + i*1;

        #endif
    }
}
```

6.3.4.3 配置描述符的运行时初始化

```
void USB_DescriptorInit(void)
{
    uint8_t *p = NULL;
    uint8_t i;
    /* copy configuration descriptor */
    memcpy(g_UsbDeviceConfigurationDescriptor + 0, g_CdcConfigurationDescriptorTemplate,
USB_DESCRIPTOR_LENGTH_CONFIGURE);

    /* copy cdc iap, interface, endpoint descriptor */
    for(i = 0; i < USB_DEVICE_CONFIG_CDC_ACM ; i++)
    {
        memcpy(g_UsbDeviceConfigurationDescriptor + USB_DESCRIPTOR_LENGTH_CONFIGURE +
            USB_CDC_DESCRIPTOR_INSTANCE_LENGTH * i,
            g_CdcDescriptorTemplate,
            USB_CDC_DESCRIPTOR_INSTANCE_LENGTH
            );
    }

    /* update interface and endpoint descirptor */
    for(i = 0; i < USB_DEVICE_CONFIG_CDC_ACM ; i++)
    {
        p = g_UsbDeviceConfigurationDescriptor + USB_DESCRIPTOR_LENGTH_CONFIGURE +
        USB_CDC_DESCRIPTOR_INSTANCE_LENGTH * i;
        /* The first interface number associated with this function */
        p[2] = g_CdcVcomCicInterfaceIndex[i];
        p += USB_IAD_DESC_SIZE;
        /* CIC interface index */
        p[2] = g_CdcVcomCicInterfaceIndex[i];
        p += USB_DESCRIPTOR_LENGTH_INTERFACE;
        p += USB_DESCRIPTOR_LENGTH_CDC_HEADER_FUNC;
        p += USB_DESCRIPTOR_LENGTH_CDC_CALL_MANAG;
        p += USB_DESCRIPTOR_LENGTH_CDC_ABSTRACT;
        p[3] = g_CdcVcomCicInterfaceIndex[i];
        p[4] = g_CdcVcomDicInterfaceIndex[i];
        p += USB_DESCRIPTOR_LENGTH_CDC_UNION_FUNC;

#if USB_CDC_CIC_INTERRUPT_IN_ENDPOINT_ENABLE
        p[2] = g_CdcVcomCicInterruptInEndpoint[i] | (USB_IN <<7);
        p += USB_DESCRIPTOR_LENGTH_CDC_CIC_INTERRUPT_ENDPOINT;
#endif
        /* data interface descriptor */
        p[2] = g_CdcVcomDicInterfaceIndex[i];

        /* bulk in endpoint descriptor */
        p += USB_DESCRIPTOR_LENGTH_INTERFACE;
        p[2] = g_CdcVcomDicBulkInEndpoint[i] | (USB_IN <<7);
```

```
        /* bulk out endpoint descriptor */
        p += USB_DESCRIPTOR_LENGTH_ENDPOINT;
        p[2] = g_CdcVcomDicBulkOutEndpoint[i] | (USB_OUT <<7);
    }
}
```

手动添加 CDC 类描述符也是容易出错的步骤。当使用 15 个 VCOM 描述符时，数组长度非常长。这种动态方法使配置描述符变得很容易。完成上述四个优化后，用户可以通过宏 USB_DEVICE_CONFIG_CDC_ACM 设置 VCOM 的数量，还可以使用宏 USB_CDC_CIC_INTERRUPT_IN_ENDPOINT_ENABLE 来确定是否在 CIC 接口中使用中断端点。

7 功能测试

AN12597SW 提供了用于 K32L2B3 的 MCUXpresso 版本的两个项目。第一个项目尚未通过代码优化中的步骤优化，第二个项目是优化后的项目。同样，还为 K32L2A 提供了两个类似的项目。使用 K32L2B3 的优化项目进行功能测试：

将宏 USB_DEVICE_CONFIG_CDC_ACM 和宏 USB_CDC_CIC_INTERRUPT_IN_ENDPOINT_ENDABLE 设置为不同的值，然后编译项目以将程序下载到板上，结果如 图 10 所示。

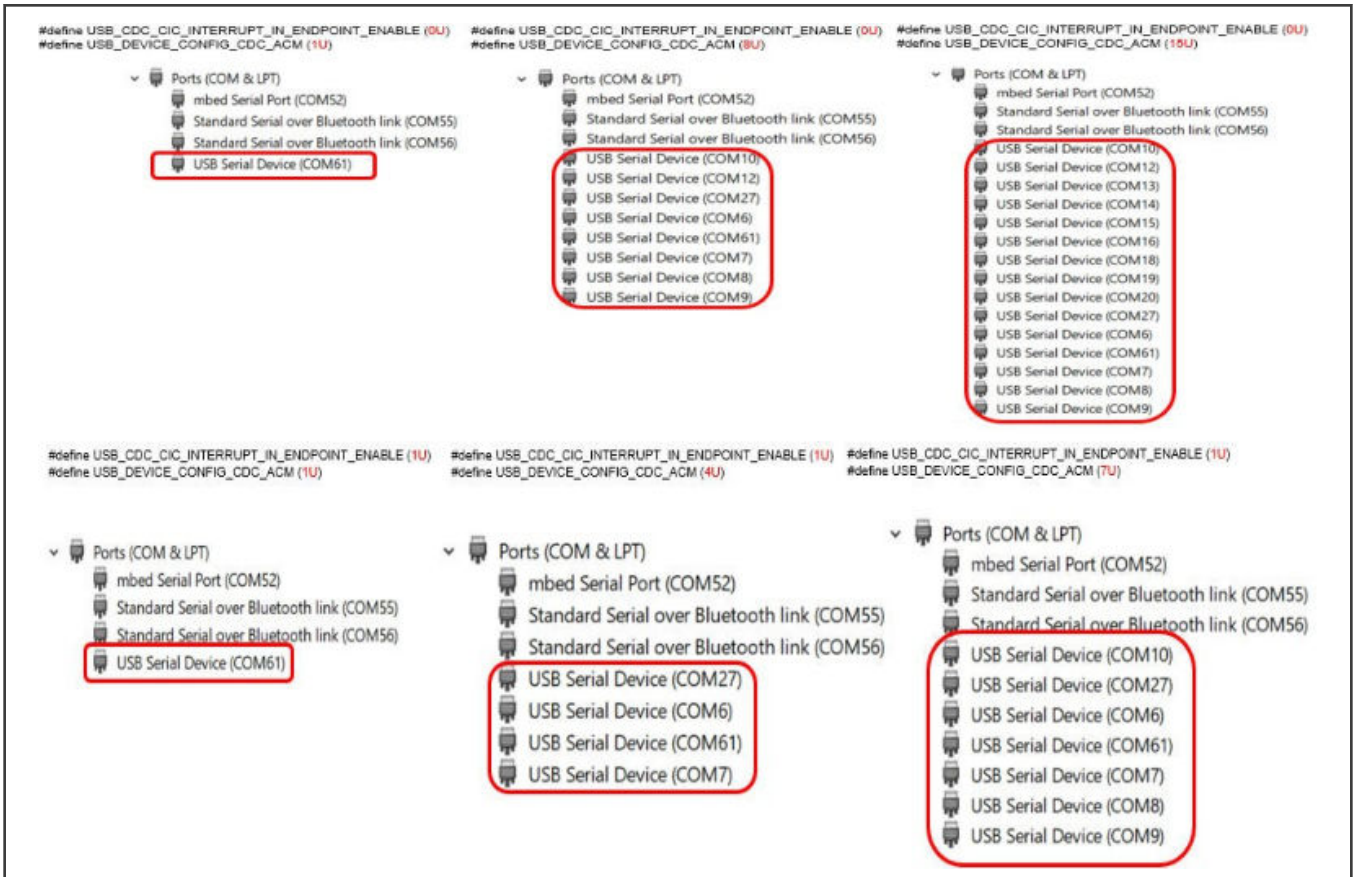

图 10. USB 主机识别结果

当不使用 CIC 接口中的 Interrupt IN 端点时，USB 设备最多可以支持 15 个 VCOM。当使用 Interrupt IN 端点时，USB 设备最多可支持 7 个 VCOM。

注意

如果 Windows 系统带有 CDC 驱动程序，则用户无需手动安装该驱动程序；如果 PC 上未安装 CDC 驱动程序，则用户需要手动驱动该驱动程序。有关驱动程序的安装，请参阅随附的 readme.pdf 文档。

8 结论

本应用笔记基于 SDK 代码，将 FS USB 设备的功能实现到 K32L2 系列 MCU 上的多个 VCOM。它最多可以支持 15 个 VCOM，并且每个 VCOM 都实现了将接收到的数据返回给主机的功能。

9 参考

1. 《LPC5500 和 LPC54018 上的 USB 转串口》（文档 AN12458）
2. K32L2B3 Reference Manual
3. USB 2.0 Specification
4. USB in MCU-Signal and Protocol
5. Access USB Technology and Application based on Microcontrollers

How To Reach Us

Home Page:

nxp.com

Web Support:

nxp.com/support

For more information, please visit: http://www.nxp.com
For sales office addresses, please send an email to: salesaddresses@nxp.com

Date of release: 2019 年 11 月
Document identifier: AN12597