USB CDC 类入门培训

1 前言

本文节选自 2017 年度 USB CDC 类培训内容的整理，主要目的是以方便些没有到现场参加培训的碟粉们可以参阅学习。本文力求从理论到实践，尽量给读者一个整体了解 USB CDC 类的窗口。当然，阅读此文，还是需要基本的 USB 知识，这个请读者自行预备。

2 USB CDC 类基础理论知识介绍

2.1 USB CDC 类、USB2.0 标准与 PSTN 之间的关系

CDC(Communication Device Class)类是 USB2.0 标准下的一个子类，定义了通信相关设备的抽象集合。它与 USB2.0 标准以及其下的子类的相互关系如下图所示:

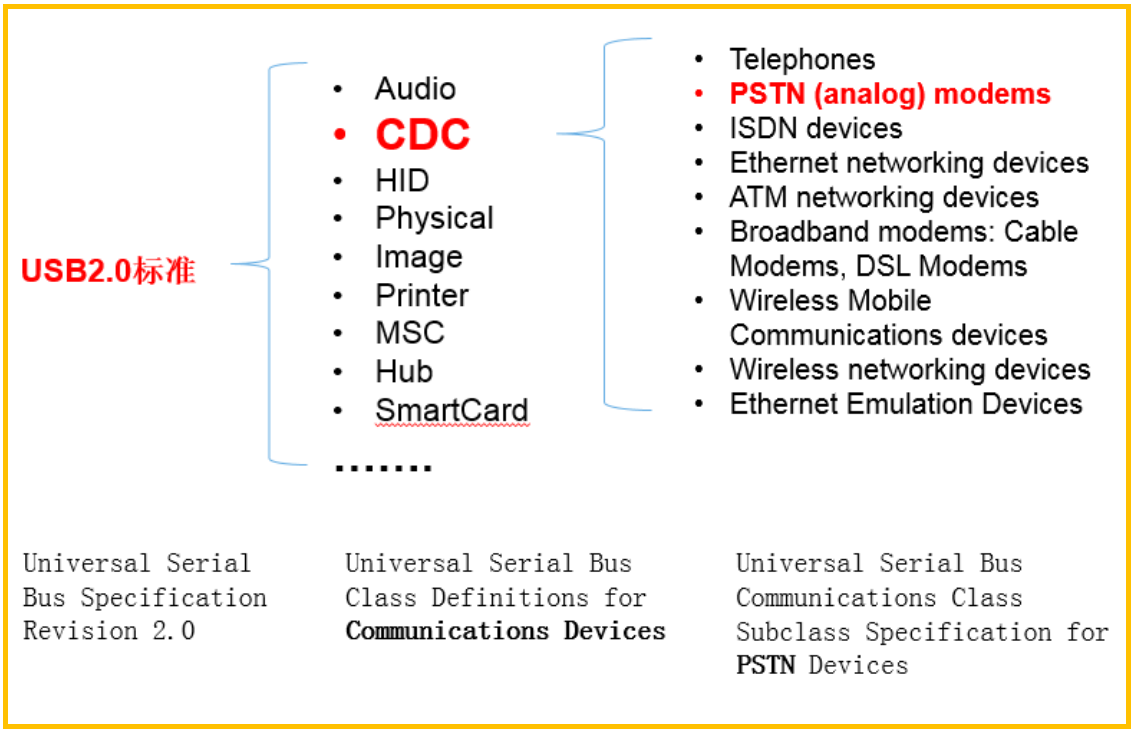
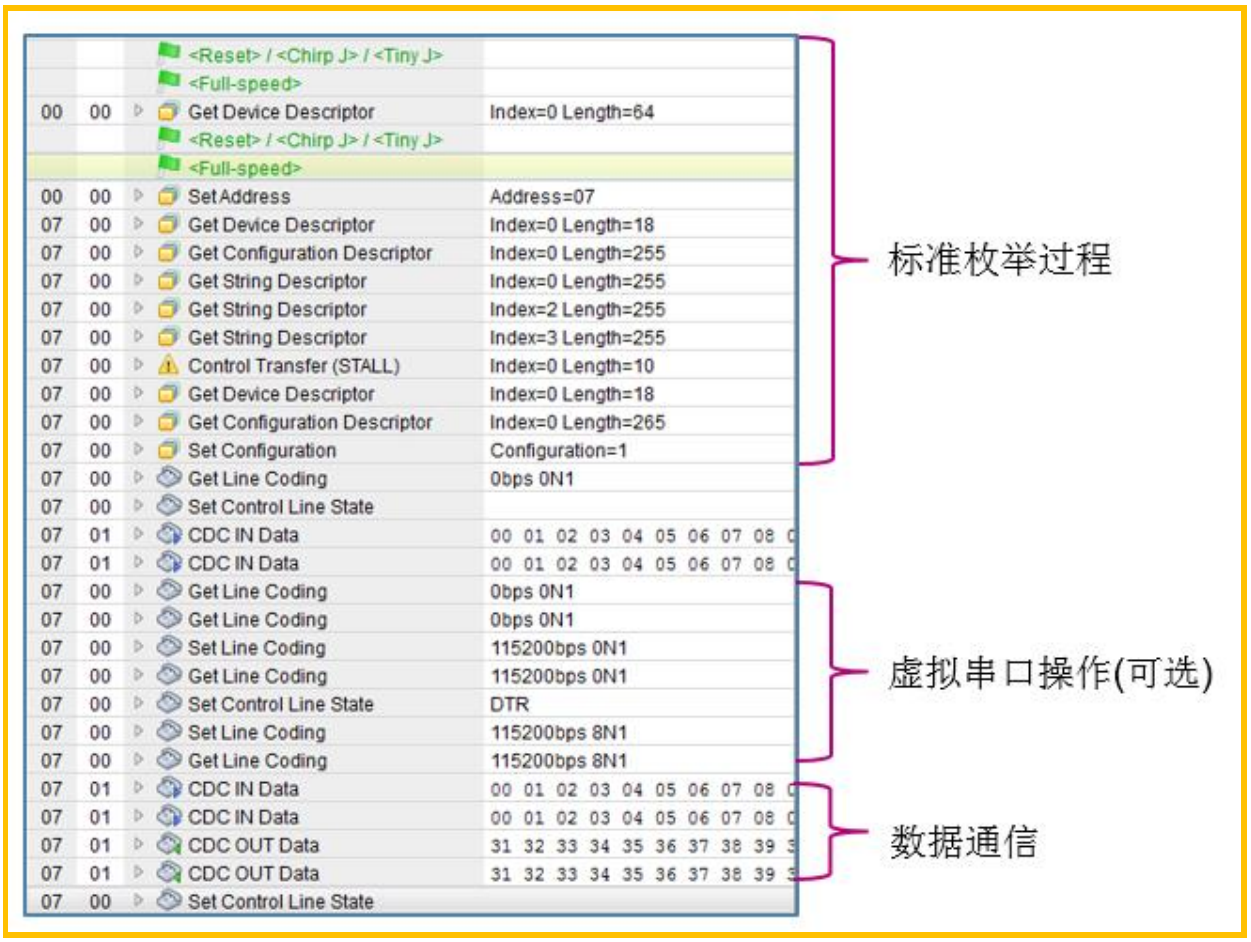


图 1 USB2.0 标准、CDC、PSTN 之间的关系

如上图，USB2.0 标准下定义了很多子类，有音频类，CDC 类，HID，打印，大容量存储类，HUB，智能卡等等，这些在 urb.org 官网上有具体的定义，这里我们主要讲的是通信类 CDC，CDC 类下面，根据具体的应用场合，又有一些子类，这里我们主要讲的是 **PSTN**(Public Switched Telephone Network)。从 PSTN 官方标准文档来看，PSTN 子类是一个与电信相关的子类，而这里，我们只是将它作为一个普通的通信设备使用，并没有使用到它的一些电话特性。

2.2 从一个具体的 CDC 类通信数据说起



		<Reset> / <Chirp J> / <Tiny J>	
		<Full-speed>	
00	00	Get Device Descriptor	Index=0 Length=64
		<Reset> / <Chirp J> / <Tiny J>	
		<Full-speed>	
00	00	Set Address	Address=07
07	00	Get Device Descriptor	Index=0 Length=18
07	00	Get Configuration Descriptor	Index=0 Length=255
07	00	Get String Descriptor	Index=0 Length=255
07	00	Get String Descriptor	Index=2 Length=255
07	00	Get String Descriptor	Index=3 Length=255
07	00	Control Transfer (STALL)	Index=0 Length=10
07	00	Get Device Descriptor	Index=0 Length=18
07	00	Get Configuration Descriptor	Index=0 Length=265
07	00	Set Configuration	Configuration=1
07	00	Get Line Coding	0bps 0N1
07	00	Set Control Line State	
07	01	CDC IN Data	00 01 02 03 04 05 06 07 08 0
07	01	CDC IN Data	00 01 02 03 04 05 06 07 08 0
07	00	Get Line Coding	0bps 0N1
07	00	Get Line Coding	0bps 0N1
07	00	Set Line Coding	115200bps 0N1
07	00	Get Line Coding	115200bps 0N1
07	00	Set Control Line State	DTR
07	00	Set Line Coding	115200bps 8N1
07	00	Get Line Coding	115200bps 8N1
07	01	CDC IN Data	00 01 02 03 04 05 06 07 08 0
07	01	CDC IN Data	00 01 02 03 04 05 06 07 08 0
07	01	CDC OUT Data	31 32 33 34 35 36 37 38 39 3
07	01	CDC OUT Data	31 32 33 34 35 36 37 38 39 3
07	00	Set Control Line State	

图 2 一个具体的 CDC 类设备通信数据

如上图，USB CDC 类的通信部分主要包含三部分：枚举过程、虚拟串口操作和数据通信。其中虚拟串口操作部分并不一定强制需要，因为若跳过这些虚拟串口的操作，实际上 USB 依然是可以通信的，这也就是为什么上图中，在操作虚拟串口之前会有两条数据通信的数据。之所以会有虚拟串口操作，主要是我们通常使用 PC 作为 Host 端，在 PC 端使用一个串口工具来与其进行通信，PC 端的对应驱动将其虚拟成一个普通串口，这样一来，可以方便 PC 端软件通过操作串口的方式来与其进行通信，但实际上，Host 端与 Device 端物理上是通过 USB 总线来进行通信的，与串口没有关系，这一虚拟化过程，起决定性作用的是对应驱动，包含如何将每一条具体的虚拟串口操作对应到实际上的 USB 操作。这里需要注意地是，Host 端与 Device 端的 USB 通信速率并不受所谓的串口波特率影响，它就是标准的 USB2.0 全速(12Mbps)速度，实际速率取决于总线的实际使用率、驱动访问 USB 外设有效速率(两边)以及外部环境对通信本身造成的干扰率等等因素组成。

2.3 CDC 类设备枚举过程

CDC 类设备与其他标准 USB 设备枚举过程的并没有什么特殊的地方。在设备描述符内可以使用 DeviceClass=0x00, SubClass=0x00, Protocol=0x00 表示此类信息在接口描述符内给出；或者也可以使用 0x02,0x00,0x00；来表明该设备为 CDC 类设备。或者使用 0xef, 0x02,0x01 表示当前为复合设备。

CDC 类设备在枚举过程中最主要的信息存储在配置描述符内：

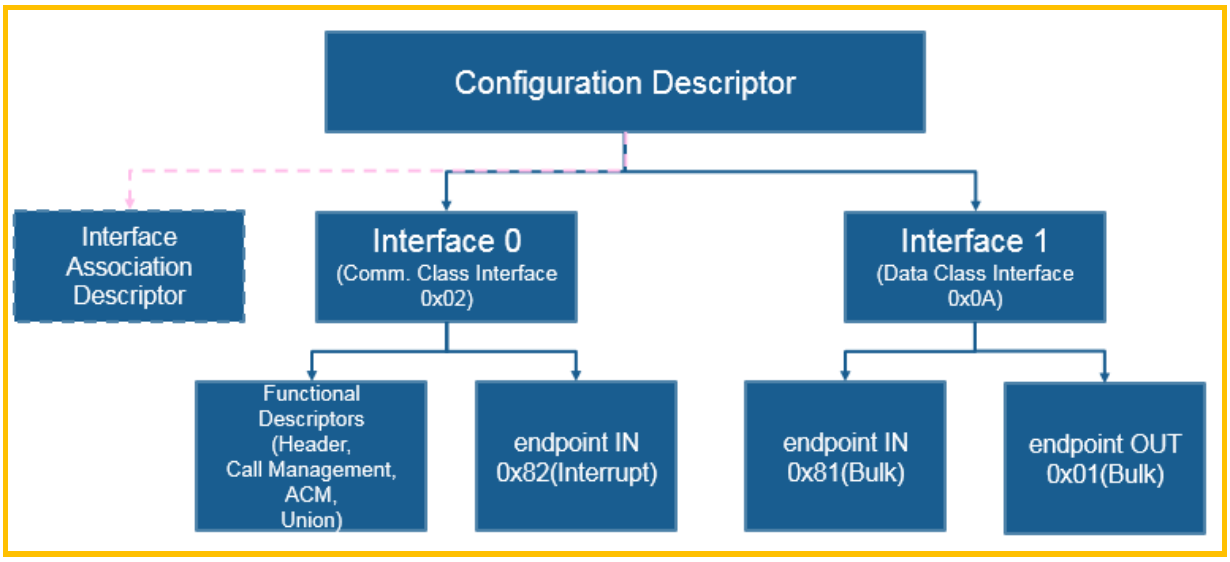


图 3 USB CDC 类配置描述符的结构

如上图所示，CDC 类的配置描述符一般包含两个接口(Interface 0)，一个控制接口，另外一个是数据接口(Interface 1), 除此之外，还有一个虚线指向的 IAD(Interface Association Description)，这个表示这个是不是可选的，得根据实际情况来确定其是否真实存在。

2.3 1 控制接口

控制接口下包含类描述符合一个端点(ie:0x82)，这个端点(中断传输模式)为异步通知消息的端点，当设备端需要向 Host 端发送异步消息时，可以通错此端点来发送，但平时主机端都是通过端点 0 来向设备端发送控制消息的，比如那些虚拟串口的操作指令等等。

除这异步通知端点外，控制接口下还包含 CDC 类相关描述符，这其中就包含 Header 描述符，Call Management 描述符，ACM 描述符以及 Union 描述符。这些功能描述符整合在一起用来描述此 USB 设备的一些功能特性，比如 AT 指令支持情况，ACM 模型下的指令集支持情况，以及还有哪些接口与此接口一起对应 Host 端的一个功能(驱动)。

在具体配置描述符内的控制接口内，功能描述符紧跟在接口描述符后，最后才是端点描述符。

- 控制接口

Interface Descriptor	
bLength	9
bDescriptorType	INTERFACE(0x04)
bInterfaceNumber	0
bAlternateSetting	0
bNumEndpoints	1
bInterfaceClass	Communications and CDC Control (0x02)
bInterfaceSubClass	Abstract Control Model (0x02)
bInterfaceProtocol	AT Commands: V.250 etc (0x01)
iInterface	None (0)

图 4 控制接口描述符

控制接口主要用来做**设备管理**和**电话管理**（可选），设备管理涉及到请求(request)和通知(notification)，端点 0 一般用做请求，一般用来控制和配置设备的运行状态，而非 0 端点(0x82)一般用作异步事件通知，设备端通过此端点向主机端发送设备内部的一些事件，比如串口状态变化事件，电话状态改变等等。

这里使用到 ACM 模型，后续将讲到这个模型，并且这里指明使用到 V250 版本的 AT 指令，这些指令是与电话相关的，但在我们这里讲的 CDC 通信实际上并不需要使用这些与电话相关的指令，它只是简单通信而已，这里指出 AT 指令也没有关系，只是实际不用它而已。

如上图，bNumEndpoints 表示此接口下包含的端点数，这里为 1 个，即那个异步通知端点。bInterfaceSubClass 为 0x02,ACM 通信模型，bInterfaceProtocol 表示 AT 指令集的版本，虽然这里举例为 V2.50，但实际上并没有使用到任何 AT 指令，因此它放

- Header 功能描述符

Header Functional Descriptor	
bFunctionLength	5
bDescriptorType	CS_INTERFACE (0x24)
bDescriptorSubtype	Header Functional Descriptor (0x00)
bcdCDC	1.10 (0x0110)

图 5 Header 功能描述符

Header 功能描述符表示功能描述符的开始，其他紧跟的内容就是此设备的功能描述符的内容。bcdCDC 表示的是 CDC 的版本。

- ACM 功能描述符

ACM Functional Descriptor	
bFunctionLength	4
bDescriptorType	CS_INTERFACE (0x24)
bDescriptorSubtype	Abstract Control Management (0x02)
bmCapabilities	Bit0:CommFeature Bit1:LineStateCoding Bit2:SendBreak Bit3:NetworkConnection

图 6 ACM 功能描述符

ACM(Abstract Control Model)，即抽象控制模型，PSTN 下，除了 ACM 模型还有还有 DLM(Direct Line Mode), TCM(Telephone Control Model)。

PSTN 定义了三种模型:DLM(Direct Line Mode),ACM(Abstract Control Model)和 TCM(Telephone Control Model).

- DLM 模型下，USB 设备直接将模拟信号转化为数字信号，并放到 USB 上传输，数据接口直接使用 Audio 类传输音频数据，控制接口传输的也都是些比较原始的指令，比如脉宽设置，发送脉宽等等；
- ACM 模型则可以很好的支持 AT V250 指令集，数据接口可以使用 Audio 类或 CDC DATA，控制接口传输的也是比较抽象的高层指令，比如设置、获取波特率，设置获取与通信相关的参数等等，而 AT 指令可以通过控制接口或者数据接口，这个在控制接口下的功能描述符 Call Management Descriptor 中指明。
- TCM 是指在物理上存在多个连接，可以将接口 0 和接口 1 分别对应到不同的物理连接上。

此外，不同的通信模型对应的指令集合（控制指令）也是不同的，而上图中 bmCapliblities 为位图，内部 bit0~bit3 分别表示 4 类控制指令集在此设备的支持情况。

Request	Summary	Req'd/Opt	reference
SendEncapsulatedCommand	Issues a command in the format of the supported control protocol.	Required	[USBCDC1.2]
GetEncapsulatedResponse	Requests a response in the format of the supported control protocol.	Required	[USBCDC1.2]
SetCommFeature	Controls the settings for a particular communications feature.	Optional	6.3.1
GetCommFeature	Returns the current settings for the communications feature.	Optional	6.3.2
ClearCommFeature	Clears the settings for a particular communications feature.	Optional	6.3.3
SetLineCoding	Configures DTE rate, stop-bits, parity, and number-of-character bits.	Optional +	6.3.10
GetLineCoding	Requests current DTE rate, stop-bits, parity, and number-of-character bits.	Optional +	6.3.11
SetControlLineState	[V24] signal used to tell the DCE device the DTE device is now present.	Optional	6.3.12
SendBreak	Sends special carrier modulation used to specify [V24] style break.	Optional	6.3.13

图 7 ACM 模型下的控制指令集

如上表，为 ACM 模型下的指令集，但不是说，这些个指令就一定会在 ACM 模型下存在，此 USB 设备是不是支持此某个控制指令，还得看 bmCapliblities 这个参数具体对应位是否使能。

在实际的 STM32 USB 协议栈中，针对于 CDC 类，使用 LineStateCoding,GetLineCoding,SetControlState 类指令，用来读取，设置串口波特率以及串口的打开与关闭，这个具体的映射实现是通过主机端的驱动来实现；从设备端来看，当设备端收到这些来自主机端操作串口的控制指令时，这些指令具体怎么执行完全取决于设备端，也就说，所有的这些操作，比如设置波特率为 115200，对于设备端来说这个只是个通过 SetLineCoding 指令传过来的一个参数而已，具体怎么处理这个参数，取决于设备端应用程序具体怎么处理这个参数，这个有用户来处理，这个 115200 波特率与 USB 本身的波特率 12Mbps(全速)是没有关系的。

- Call Management 功能描述符

Call Management Functional Descriptor	
bFunctionLength	5
bDescriptorType	CS_INTERFACE (0x24)
bDescriptorSubtype	Call Management Functional Descriptor (0x01)
bmCapabilities	Bit0(CallManagement): Does not handle call management (0b0) Bit1(DataClass): Call Management over Comm Class interface (0b0)
bDataInterface	1

图 8 Call Management 功能描述符

Call Management 描述的就是电话相关的东西，AT 指令集的支持情况。但在这里，我们并没有用到任何与电话相关的指令，因此 bmCapabilities 下的位图各个位都是为 0：Bit0:是否支持电话相关的指令(AT 指令集)；Bit1:电话相关的指令(AT 指令集)是否经过 Comm. Class Interface； bDataInterface 表示如有电话时，电话数据内容对应的接口号。

- Union 功能描述符

Union Functional Descriptor	
bFunctionLength	5
bDescriptorType	CS_INTERFACE (0x24)
bDescriptorSubtype	Union Functional Descriptor (0x06)
bControlInterface	0
bSubordinateInterface0	1

图 9 Union 功能描述符

Union 描述符就是用来告诉主机端，哪些接口是联合在一起的，对应着一个功能，这个功能需要主机装载对应的驱动来实现，因此，功能与驱动是一对一的关系。这里 bControlInterface 值为 0，则表示接口 0 为控制接口，bSubBoardinateInterface0 值为 1，表示接口 1 为控制接口 0 的下级接口，即数据接口。在 CDC 标准中，控制接口是必须的，而数据接口是可选的，因此，数据接口为控制接口的附属。

2.3.2 数据接口

Interface Descriptor	
bLength	9
bDescriptorType	INTERFACE(0x04)
bInterfaceNumber	1
bAlternateSetting	0
bNumEndpoints	2
bInterfaceClass	CDC Data (0x0a)
bInterfaceSubClass	Unknown (0x00)
bInterfaceProtocol	No class specific protocol required (0x00)
iInterface	None (0)

图 10 数据接口

数据接口比较简单，就是数据通信的，用到两个端点 IN/OUT 0x81/0x01,为块传输类型。

2.3.3 IAD(Interface Association Descriptor)

IAD	
bFunctionLength	8
bDescriptorType	INTERFACE_ASSOCIATION (0x0b)
bFirstInterface	0
bInterfaceCount	2
bFunctionClass	Communications and CDC Control (0x02)
bFunctionSubClass	Abstract Control Model (0x02)
bFunctionProtocol	No class specific protocol required (0x00)
iFunction	None (0)

图 11 IAD 描述符

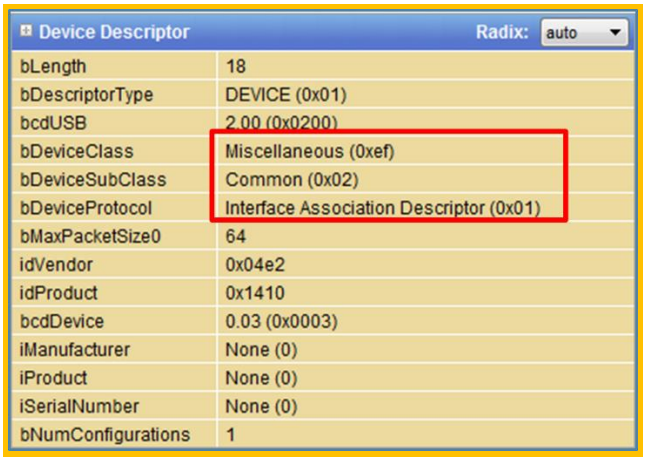
USB 刚出来的时候，一开始默认是一个接口对应一个功能，而一个功能对应着主机端的一个驱动，这在当时是 OK 的，但是后来，人们发现，需要多个接口对应一个功能的时候，比如这个 CDC，除了数据接口外还需要控制接口，这在当时是没有这方面的统一标准，于是就出了 Union 来表示多个接口对应一个功能的情况。再后来，USB 标准协会又增加了 IAD。

IAD 与 Union 类似，Union 是旧版本下实现多个接口对应一个功能的功能描述符，而 IAD 是 USB 协会后来针对多个接口对应一个功能的情况而扩展的，旧的主机可能只支持 Union 方式，但 IAD 并不会影响旧版本主机对设备的识别，因为旧版本主机会通过 Union 来识别哪些接口是联合在一起的，对于 IAD 则跳过忽略；而新版主机则可以通过 IAD 来识别，跳过忽略老的 Union，因此两者可以完美兼容，互不影响。因而主机端可以精确地装载对应的驱动。

IAD 只用在设备描述符中只用了 device class code,并且指明了使用 IAD 来识别设备，比如 bDeviceClass: Miscellaneous (0xef), bDeviceSubClass: Common (0x02), bDeviceProtocol: Interface Association Descriptor (0x01)就是一个例子; 0x02,0x00,0x00 是另外一个例子。

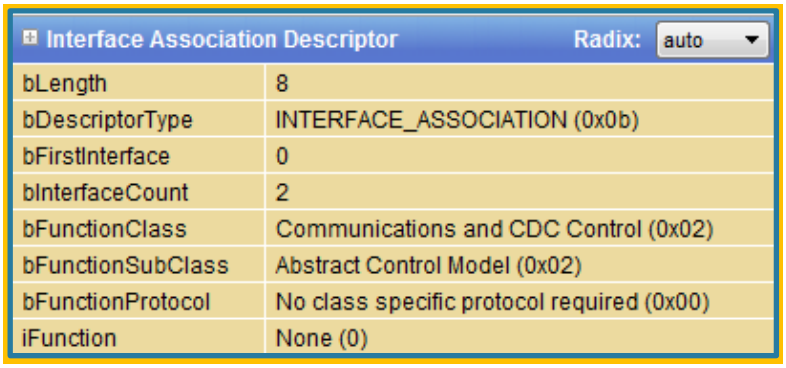
如上图，bFirstInterface 值为 0，表示第一个接口个接口 0，默认为控制接口；bInterfaceCount 值为 2，标志此功能总共存在 2 个接口，那么第二个接口就是接口 1，因为 USB2.0 IAD ECN 补充标准规定，这里提到的接口号必须是连续的，也就是说，接口 0 为第一个控制接口，那么接口 1 则为数据接口。

下面我们来个具体的 IAD 例子：



Device Descriptor	Radix: auto
bLength	18
bDescriptorType	DEVICE (0x01)
bcdUSB	2.00 (0x0200)
bDeviceClass	Miscellaneous (0xef)
bDeviceSubClass	Common (0x02)
bDeviceProtocol	Interface Association Descriptor (0x01)
bMaxPacketSize0	64
idVendor	0x04e2
idProduct	0x1410
bcdDevice	0.03 (0x0003)
iManufacturer	None (0)
iProduct	None (0)
iSerialNumber	None (0)
bNumConfigurations	1

图 12 IAD 存在时的设备描述符



Interface Association Descriptor	Radix: auto
bLength	8
bDescriptorType	INTERFACE_ASSOCIATION (0x0b)
bFirstInterface	0
bInterfaceCount	2
bFunctionClass	Communications and CDC Control (0x02)
bFunctionSubClass	Abstract Control Model (0x02)
bFunctionProtocol	No class specific protocol required (0x00)
iFunction	None (0)

图 13 IAD

如上图所示，一般 IAD 存在的情况下，在设备描述符中 DeviceClass 等三个参数不再都为 0x00,图 12 中为 0xef,0x02,0x01，这个表示是复合设备，此时，可以使用 IAD 来定义多个接口联合起来对应一个 USB 驱动。从 IAD 中可以看出，bFunctionClass 参数就定义了此 IAD 表示的设备为 CDC 类设备，ACM 模型。就这样，通过 IAD 描述符，实现了与 Union 功能描述符相同的功能。

2.3.4 ACM 模型

之前我们已经在控制接口中的功能描述符中已有对 ACM(Abstract Control Mode)模型的简介，也有提到过，在 PSTN 中，除了 ACM 模式，还有 TCM，DLM 模式。这三种模式，不同的模式下包含的控制指令集是不尽相同的，有部分控制指令可能同时存在两个或三个模式下，除了控制指令，还有异步通知消息，这个在三个不同模式下也是不相同的。

Request	Summary	Req'd/Opt	reference
SendEncapsulatedCommand	Issues a command in the format of the supported control protocol.	Required	[USBCDC1.2]
GetEncapsulatedResponse	Requests a response in the format of the supported control protocol.	Required	[USBCDC1.2]
SetCommFeature	Controls the settings for a particular communications feature.	Optional	6.3.1
GetCommFeature	Returns the current settings for the communications feature.	Optional	6.3.2
ClearCommFeature	Clears the settings for a particular communications feature.	Optional	6.3.3
SetLineCoding	Configures DTE rate, stop-bits, parity, and number-of-character bits.	Optional +	6.3.10
GetLineCoding	Requests current DTE rate, stop-bits, parity, and number-of-character bits.	Optional +	6.3.11
SetControlLineState	[V24] signal used to tell the DCE device the DTE device is now present.	Optional	6.3.12
SendBreak	Sends special carrier modulation used to specify [V24] style break.	Optional	6.3.13

图 14 ACM 模式下的控制指令集

Notification	Summary	Req'd/Opt	reference
NetworkConnection	Notification to host of network connection status.	Optional +	[USBCDC1.2]
ResponseAvailable	Notification to host to issue a GET_ENCAPSULATED_RESPONSE request.	Required	6.5.1
SerialState	Returns the current state of the carrier detect, DSR, break, and ring signal.	Optional +	6.5.4

图 15 ACM 模式下的异步通知消息

Request	Summary	Req'd/Opt	reference
SetAuxLineState	Request to connect or disconnect secondary jack from PSTN circuit or CODEC, depending on hook state.	Optional	6.3.4
SetHookState	Select relay setting for on-hook, off-hook, and caller ID.	Required	6.3.5
PulseSetup	Initiate pulse dialing preparation.	Optional	6.3.6
SendPulse	Request number of make/break cycles to generate.	Optional	6.3.7
SetPulseTime	Setup value for time of make and break periods when pulse dialing.	Optional	6.3.8
RingAuxJack	Request for a ring signal to be generated on secondary phone jack.	Optional	6.3.9

图 16 DLM 模式下的控制指令集

Notification	Summary	Req'd/Opt	reference
AuxJackHookState	Indicates hook state of secondary device plugged into the auxiliary phone jack.	Optional	6.5.2
RingDetect	Message to notify host that ring voltage was detected on PSTN interface.	Required	6.5.3

图 17 DLM 模式下的异步通知消息

Notification	Summary	Req'd/Opt	reference
AuxJackHookState	Indicates hook state of secondary device plugged into the auxiliary phone jack.	Optional	6.5.2
RingDetect	Message to notify host that ring voltage was detected on PSTN interface.	Required	6.5.3

Request	Summary	Req'd/Opt	reference
SetRingerParms	Configures the ringer for a telephone device.	Optional	6.3.14
GetRingerParms	Gets the current ringer configuration for a telephone device.	Required	6.3.15
SetOperationParms	Configures the operational mode of the telephone.	Optional	6.3.16
GetOperationParms	Gets the current operational mode of the telephone.	Optional	6.3.17
SetLineParms	Allows changing the current state of the line associated with the interface, providing basic call capabilities, such as dialing and answering calls.	Required	6.3.18
GetLineParms	Gets current status of the line.	Required	6.3.19
DialDigits	Dials digits on the network connection.	Required	6.3.20

图 18 TCM 模式下的控制指令集

Notification	Summary	Req'd/Opt	reference
CallStateChange	Reports a state change on a call.	Required	6.5.5
LineStateChange	Reports a state change on a line.	Optional	6.5.6

图 19 TCM 模式下的异步通知消息

由图 14~19 可知，当设备选择了某个模型后，其控制指令集和异步通知消息也就得符合此模式下的对应集合，否则则不符合标准。这里我们主要是使用到 ACM 模式，因此，此 ACM 模式下的有 Host 端发现 Device 端的控制指令和有 Device 端向 Host 端发送的异步通知消息都是固定的那么几条指令或消息，但并不是说，只要是 ACM 模式，那么就表示此模式下的所有控制指令和异步通知消息都必须支持。控制指令在设备端的控制接口描述符中的 ACM 功能描述符中的 bCapabilities 字段有按

位定义 ACM 模式下的控制指令的支持情况，而异步通知消息，则完全看 device 端的应用情况是否需要，并没有在任何描述符中指出那些消息是否支持。

在 ST 给出的 CDC 例程中，主要是使用到了 SetLineCoding 指令来设置和修改虚拟串口的波特率，使用 GetLineCoding 来获取当前波特率，使用 SetControlLineState 来打开或关闭串口，这种操作是在 Host 端 CDC 驱动来具体映射实现的，至于 Device 端收到这些个控制指令该怎么处理，就是另外一回事了，Device 端也可以完全不做任何处理，有 CubeMx 自动生成的 CDC 类代码就是这样，对接收到的任何控制指令到没有做任何处理，当然，如果需要的话，则按应用的需要来处理，这个完全取决于用户。

07	00	Get Line Coding	0bps 0N1
07	00	SETUP txn	A1 21 00 00 00 00 07 00
07	00	IN txn [1 POLL]	00 00 00 00 00 00 00
07	00	OUT txn [1 POLL]	
07	00	Get Line Coding	0bps 0N1
07	00	SETUP txn	A1 21 00 00 00 00 07 00
07	00	IN txn [1 POLL]	00 00 00 00 00 00 00
07	00	OUT txn [1 POLL]	
07	00	Set Line Coding	115200bps 0N1
07	00	SETUP txn	21 20 00 00 00 00 07 00
07	00	OUT txn [1 POLL]	00 C2 01 00 00 00 00
07	00	IN txn [1 POLL]	
07	00	Get Line Coding	115200bps 0N1
07	00	SETUP txn	A1 21 00 00 00 00 07 00
07	00	IN txn [1 POLL]	00 C2 01 00 00 00 00
07	00	OUT txn [1 POLL]	
07	00	Set Control Line State	DTR
07	00	SETUP txn	21 22 01 00 00 00 00 00
07	00	IN txn [1 POLL]	
07	00	Set Line Coding	115200bps 8N1
07	00	SETUP txn	21 20 00 00 00 00 07 00
07	00	OUT txn [1 POLL]	00 C2 01 00 00 00 08
07	00	IN txn [1 POLL]	
07	00	Get Line Coding	115200bps 8N1
07	00	SETUP txn	A1 21 00 00 00 00 07 00
07	00	IN txn [1 POLL]	00 C2 01 00 00 00 08
07	00	OUT txn [1 POLL]	

图 20 控制指令操作虚拟串口

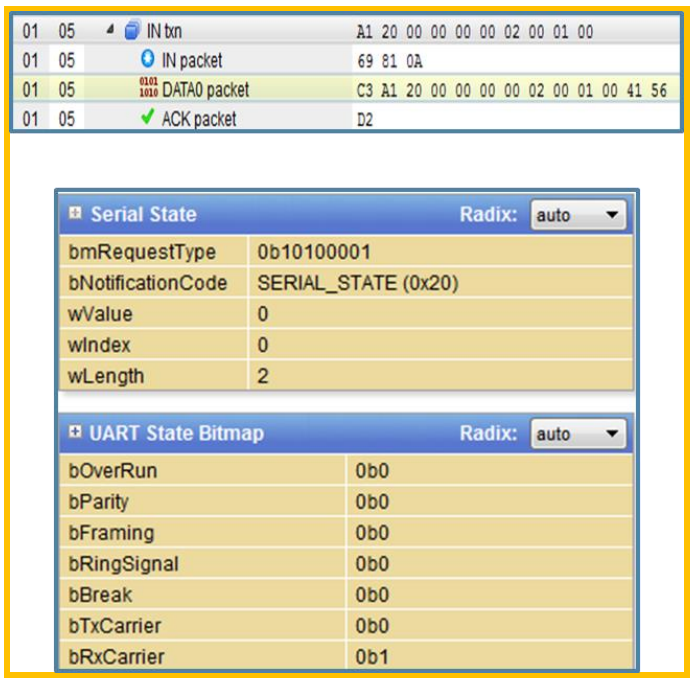


图 21 一个 ACM 模式下的异步通知消息例子

3 CDC 类软件框架介绍

3.1 CDC 软件框架简介

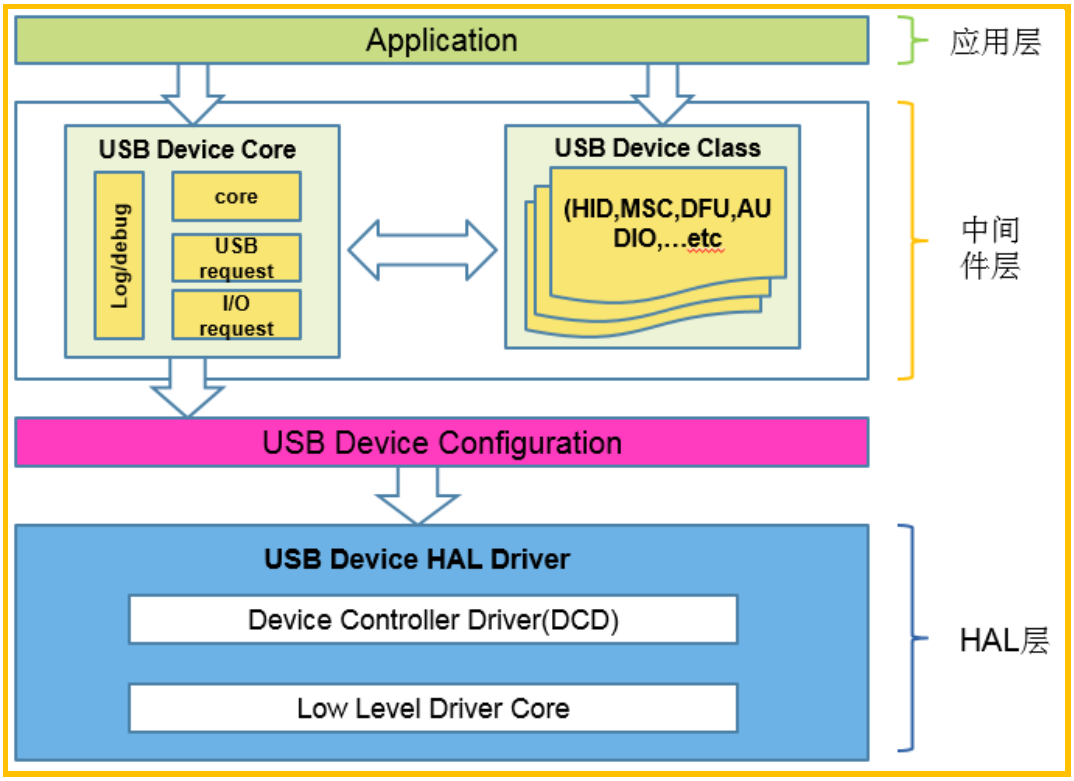


图 22 CDC 类软件框架

如上图所示，黄色 USB Device Core 部分为 USB 设备库文件,属于中间件,它为 USB 协议栈的核心源文件，一般不需要修改：

- USB Device Core 中,Log/debug 为打印/调试开关;
- core 为 USB 设备核心;
- USB request 中定义了枚举过程中各种标准请求的处理；
- I/O request 为底层针对 USB 通信接口的封装。

黄色 USB Device Class 部分为 USB 类文件，也属于中间件,USB 设备库，目前 ST DEMO 中支持的类有 HID, Customer HID, CDC, MSC, DFU, Audio, ST 提供了这些类的源码框架，其他的 Class 或者是复合设备需要自己根据实际需求情况进行扩展或定制。如果用户需求只是需要一个标准类，比如 CDC 通信，那么最好就使用现成的代码，不需要做任何修改就可以实现这个 CDC 类通信的功能。

蓝色 USB Device HAL Driver 为 HAL 库部分，是对 USB 外设接口的封装，属于底层驱动，不需要修改，它分为 PCD 和 LL Driver，PCD 处于 LL Driver 之上。

洋红色 USB Device Configuration 为 USB 配置封装，位于 USB 底层 HAL 层驱动与中间件 USB 协议栈之间，一方面向上层(USB 设备库)提供各种操作调用接口，另一方面，向底层 USB 驱动提供各种回调接口。正是由于它的存在，使得 USB 协议栈(USB 设备库)与底层硬件完全分离，从而使 USB 设备库具有更加兼容所有 STM32 的通用性。USB Device Configuration 为开放给用户的源文件，用户可以根据自己的某些特殊需要进行修改，也可以使用默认的源文件，假如没有任何特殊要求的话，我们使用默认即可。

Application 为应用层，USB Device Class 有可能将自己对应该的操作接口封装在一个操作数据结构中，由应用来具体实现这些操作，在系统初始化时，由应用将已经定义好的操作接口注册到对应的 USB 类中，比如 usbd_cdc_if， 就这样，使得应用层的应用代码与属于中间件层的 USB 协议栈分离。同时，USB 协议栈会将一些字符串描述符放到 APP 中，当 USB 初始化时将这些已经定义好的字符串通过指针初始化到 USB 协议栈中，以便后续需要时获取。

3.2 工程源码文件与软件框架的对应关系

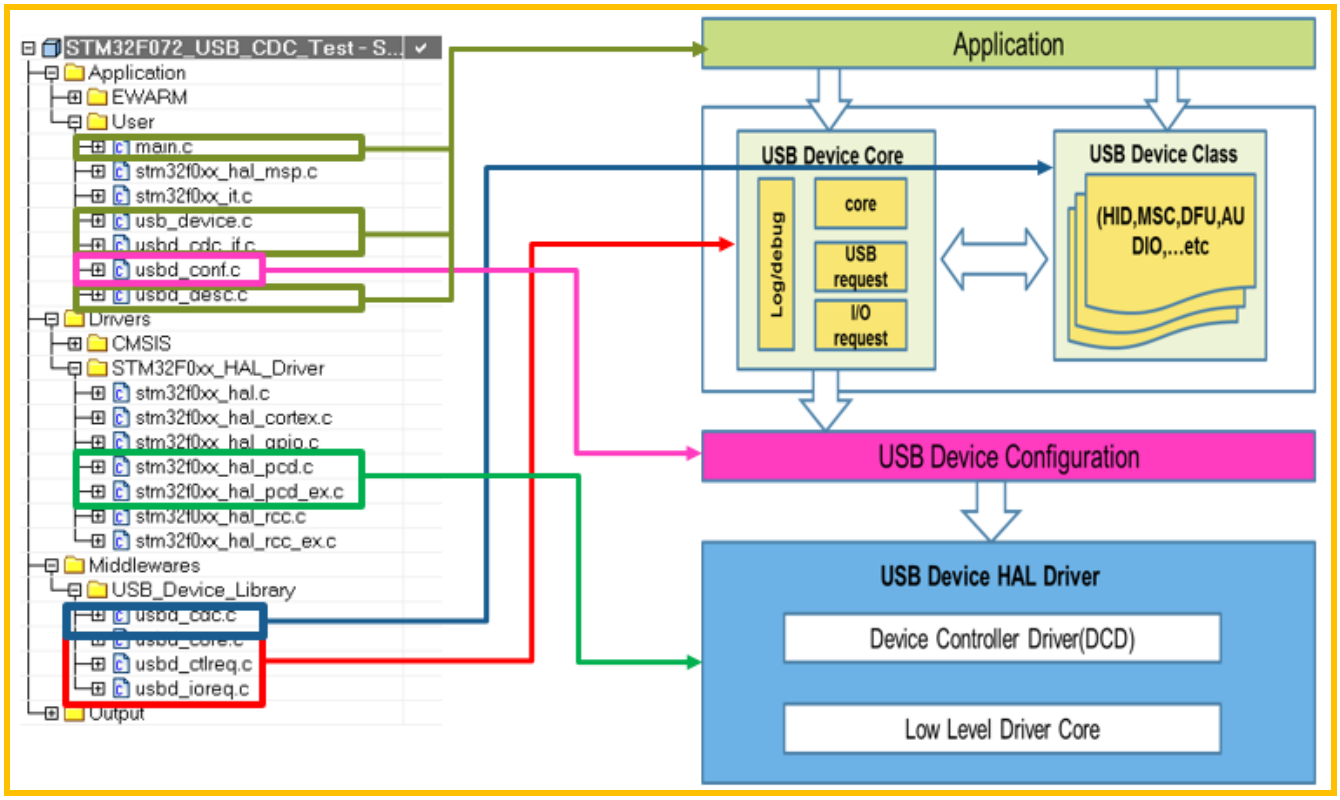


图 23 CDC 工程中源码与软件框架的对应关系

3.3 USBD 内核与 USBD_CDC 的关系

3.1 节中，我们已经提到过 ST 官方 Cube 库中提供的官方 USB 协议栈，主要是包含了 USBD 内核与 USB 各种类。USBD 内核一般是固定的，用户一般不需要修改，但 USBD 类，如果用户需要修改或者扩展，比如复合设备或者用户自定义设备，还有就是，ST 目前官方提供的 USB 设备类的 DEMO 程序并没有囊括所有 USB 类，因此，若用户需要实现这些官方提供 DEMO 之外的 USB 类时，则用户需要根据自己的需要来定制化自己的 USB 类，那么又该如何开始呢？

我们已经知道，ST 提供的 USB 协议栈中已经有 USBD 内核，且这个内核源文件一般是不需要修改的，那么这里我们需要自定义这么一个 USB 类，那么我们首先得知道，这个我们需要自定义的 USB 类是如何与 USBD 内核打交道的？

USB 协议栈将所有 USB 类都抽象成一个数据结构：USBD_ClassTypeDef，其定义如下所示：

```
typedef struct _Device_cb
{
  uint8_t  (*Init)             (struct _USBD_HandleTypeDef *pdev , uint8_t cfgidx);
  uint8_t  (*DeInit)           (struct _USBD_HandleTypeDef *pdev , uint8_t cfgidx);
 /* Control Endpoints*/
  uint8_t  (*Setup)            (struct _USBD_HandleTypeDef *pdev , USBD_SetupReqTypedef
*req);
  uint8_t  (*EP0_TxSent)       (struct _USBD_HandleTypeDef *pdev );
  uint8_t  (*EP0_RxReady)      (struct _USBD_HandleTypeDef *pdev );
```

```
  /* Class Specific Endpoints*/
  uint8_t  (*DataIn)           (struct _USBD_HandleTypeDef *pdev , uint8_t epnum);
  uint8_t  (*DataOut)          (struct _USBD_HandleTypeDef *pdev , uint8_t epnum);
  uint8_t  (*SOF)              (struct _USBD_HandleTypeDef *pdev);
  uint8_t  (*IsoINIncomplete)  (struct _USBD_HandleTypeDef *pdev , uint8_t epnum);
  uint8_t  (*IsoOUTIncomplete) (struct _USBD_HandleTypeDef *pdev , uint8_t epnum);

  uint8_t  *(*GetHSConfigDescriptor)(uint16_t *length);
  uint8_t  *(*GetFSConfigDescriptor)(uint16_t *length);
  uint8_t  *(*GetOtherSpeedConfigDescriptor)(uint16_t *length);
  uint8_t  *(*GetDeviceQualifierDescriptor)(uint16_t *length);
#if (USBD_SUPPORT_USER_STRING == 1)
  uint8_t  *(*GetUsrStrDescriptor)(struct _USBD_HandleTypeDef *pdev ,uint8_t index,
uint16_t *length);
#endif

} USBD_ClassTypeDef;
```

这个结构体是一个抽象类，定义了一些虚拟函数，比如初始化，反初始化，类请求指令处理函数，端点 0 发送完成，端点 0 接收处理，数据发送完成，数据接收处理，SOF 中断处理，同步传输发送未完成，同步传输接收未完成处理等等；用户在实现自己具体的 USB 类的时候需要将它实例化，USBD_ClassTypeDef 结构体是 USBD 内核提供给外部定义一个 USB 设备类的窗口，而 USB 类文件(如 usbd_cdc.c)实际就是实现这个结构体具体实例化的过程。最后将这个具体实例化的对象注册到 USBD 内核的同时, USBD 内核与 USBD 类也进行了关联。

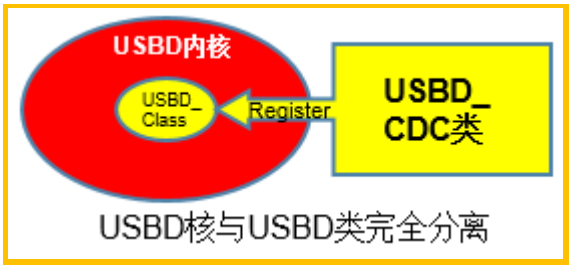


图 24 USBD 核与 CDC 类的关系

可以这么说，USBD 内核与 USBD 类之间的纽带就是 USBD_ClassType 这个结构体。

下面我们来看看 ST 提供的 CDC DEMO 中具体 CDC 类：

```
USBD_ClassTypeDef  USBD_CDC =
{
  USBD_CDC_Init,           //初始化
  USBD_CDC_DeInit,         //反初始化
  USBD_CDC_Setup,          //CDC 类请求指令处理，也就是指 CDC 类控制指令的处理
  NULL,                  /* EP0_TxSent, */  //端点 0 发送完成，不需要处理
  USBD_CDC_EP0_RxReady,          //端点 0 接收处理，实际上当做 CDC 类控制指令来处理
  USBD_CDC_DataIn,         //CDC 发送数据完成处理
  USBD_CDC_DataOut,        //CDC 类接收数据处理
  NULL,                    //SOF 中断不做处理
```

```
  NULL,                   //同步传输发送未完成中断不做处理
  NULL,                   //同步传输接收未完成中断也不做处理
  USBD_CDC_GetHSCfgDesc,  //获取高速 USB 配置描述符
  USBD_CDC_GetFSCfgDesc,          //获取全速设备配置描述符
  USBD_CDC_GetOtherSpeedCfgDesc,
  USBD_CDC_GetDeviceQualifierDescriptor,
};
```

这个就是具体一个 CDC 类实例化的对象，上层应用通过 USBD_RegisterClass 函数，将此对象注册到 usbd 内核：

```
USBD_RegisterClass(&hUsbDeviceFS, &USBD_CDC);
```

它主要在 usbd_cdc.c 源文件中实现它的各个成员函数，当然，usbd_cdc.c 源文件中，除了这些 CDC 类成员函数的具体实现之外，还包含其他一些对上层提供的接口，比如发送 USBD_CDC_TransmitPacket, USBD_CDC_RegisterInterface,上层应用通过调用 USBD_CDC_TransmitPacket 来发送数据，通过 USBD_CDC_RegisterInterface 来注册操作接口，这也是我们接下来将要讲述的内容。

3.4 USBD_CDC 与 USBD_CDC_If 的关系

讲完了 USBD 内核与 USBD_CDC 的关系，接下来我们来讲下 USB_CDC 与上层应用是如何对接的。为了将 USBD_CDC 与上层应用层完全分离出来，类似 USBD 内核与 USBD_CDC 类完全分离一般，USBD_CDC 类对上层同样提供一个抽象的数据操作接口 USBD_CDC_If 结构体:

```
typedef struct _USBD_CDC_Itf
{
  int8_t (* Init)          (void);
  int8_t (* DeInit)        (void);
  int8_t (* Control)       (uint8_t, uint8_t * , uint16_t);//处理收到的来自 Host 端的控制指令
  int8_t (* Receive)       (uint8_t *, uint32_t *); //处理收到的数据

}USBD_CDC_ItfTypeDef;
```

如上所示，如何处理来自 Host 端发送过来的控制指令和数据，完全是由应用层来决定，具体实现是应用层将此抽象的操作接口具体实例化，并注册到 USBD_CDC 类对象中：

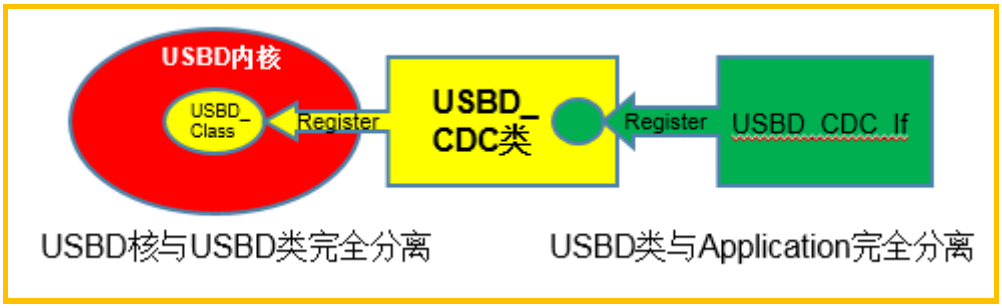


图 25 USBD_CDC 类与 USBD_CDC_If 的关系

如上图所示，通过引入 USBD_CDC_If 这么一个数据结构，就实现了 USBD_CDC 类与应用层的完全分离。USBD_CDC_If 的具体实例化对象如下:

```
USBD_CDC_ItfTypeDef USBD_Interface_fops_FS =
{
  CDC_Init_FS,
  CDC_DeInit_FS,
  CDC_Control_FS,
  CDC_Receive_FS
};
```

源文件 usbd_cdc_if.c 就是实现这些成员函数的过程，除此之外，还包含发送接口。最后应用层通过调用 USBD_CDC_RegisterInterface 函数将此操作接口注册到 USBD_CDC 类中：

```
USBD_CDC_RegisterInterface(&hUsbDeviceFS, &USBD_Interface_fops_FS);
```

3.5 应用接口

- **初始化：**

```
void MX_USB_DEVICE_Init(void)
{
  /* Init Device Library,Add Supported Class and Start the library*/
  USBD_Init(&hUsbDeviceFS, &FS_Desc, DEVICE_FS);

  USBD_RegisterClass(&hUsbDeviceFS, &USBD_CDC);     //注册 USBD_CDC 类到内核

  USBD_CDC_RegisterInterface(&hUsbDeviceFS, &USBD_Interface_fops_FS);//注册 USBD_CDC_If 到
CDC 类

  USBD_Start(&hUsbDeviceFS);

}
```

如上图所示：

初始化分 4 步:

1> 初始化 USBD 内核

2> 给 USBD 内核注册 USBD_CDC 类

3> 给 USBD_CDC 类注册 USBD_CDC_If 接口

4> 正式启动 USBD

- **发送数据:**

uint8_t CDC_Transmit_FS(uint8_t* Buf, uint16_t Len);

- **接收回调处理:**

static int8_t CDC_Receive_FS (uint8_t* Buf, uint32_t *Len);

- **接收控制指令处理：**

static int8_t CDC_Control_FS (uint8_t cmd, uint8_t* pbuf, uint16_t length);

4 实践动手部分

本实验的目的是让用户学会自己动手创建一个 STM32CubeMx 工程，一步一步实现通过使用 USB CDC 类与 PC 端进行串口通信。

4.1 实验环境及 STM32F072-Discovery 板简介

硬件准备:

- STM32F072 Discovery 板一块
- Mini USB 线两根
- PC 一台

软件准备:

- IAR V6.7.0 或者以上版本
- STM32CubeF0 V1.7.0
- STM32CubeMX V4.19
- SSCOM 串口工具
- VCP 虚拟串口驱动

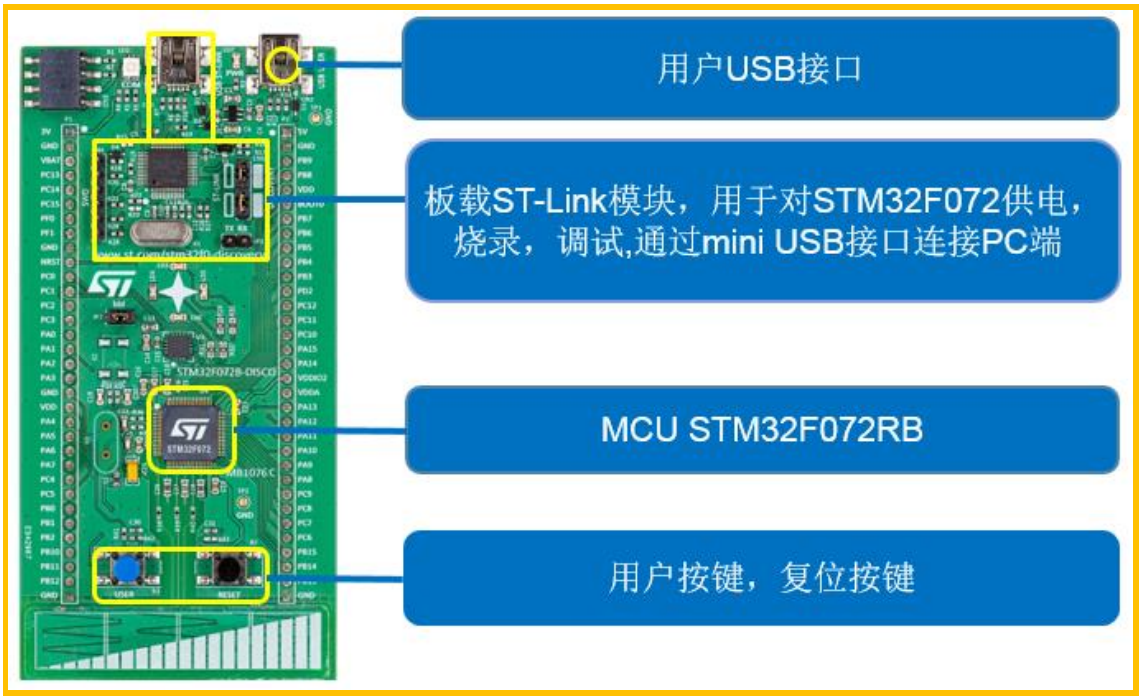


图 26 STM32F072-Discovery 开发板介绍

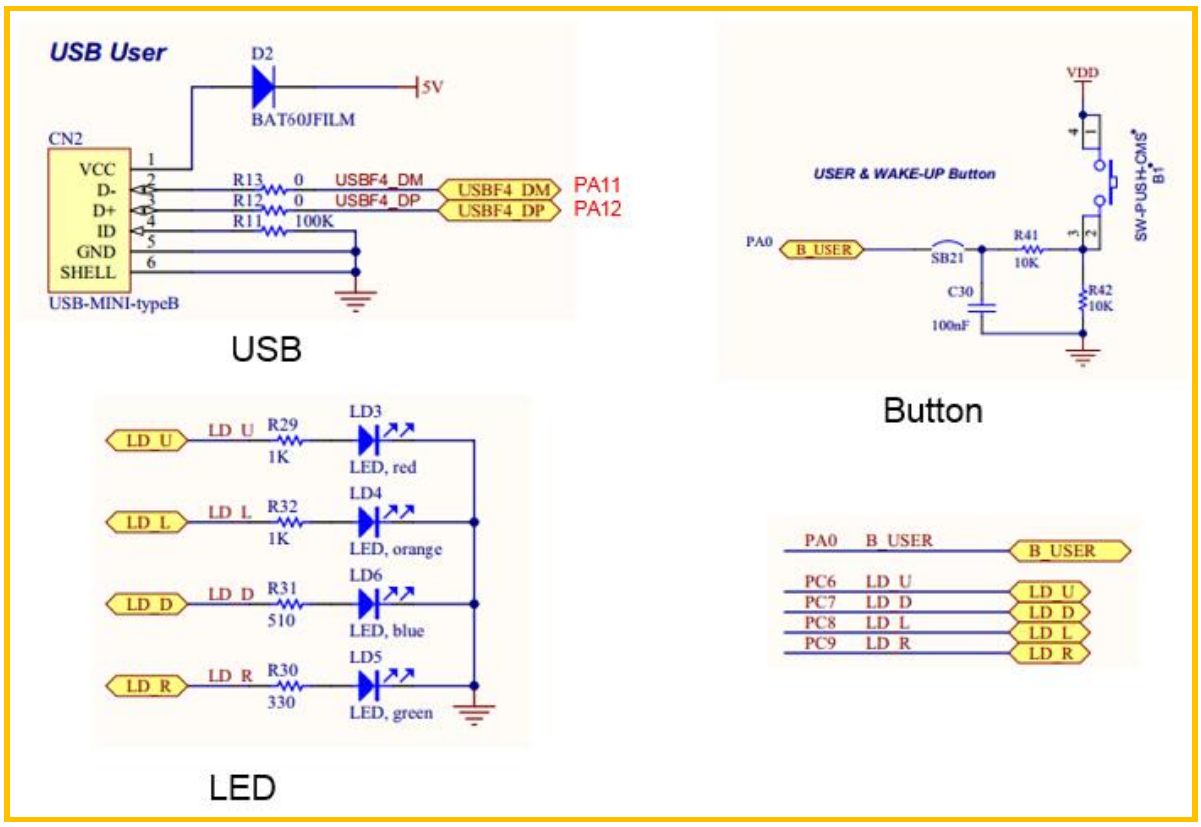


图 27 板载电路简介

4.2 使用 STM32CubeMx 制作 CDC 工程

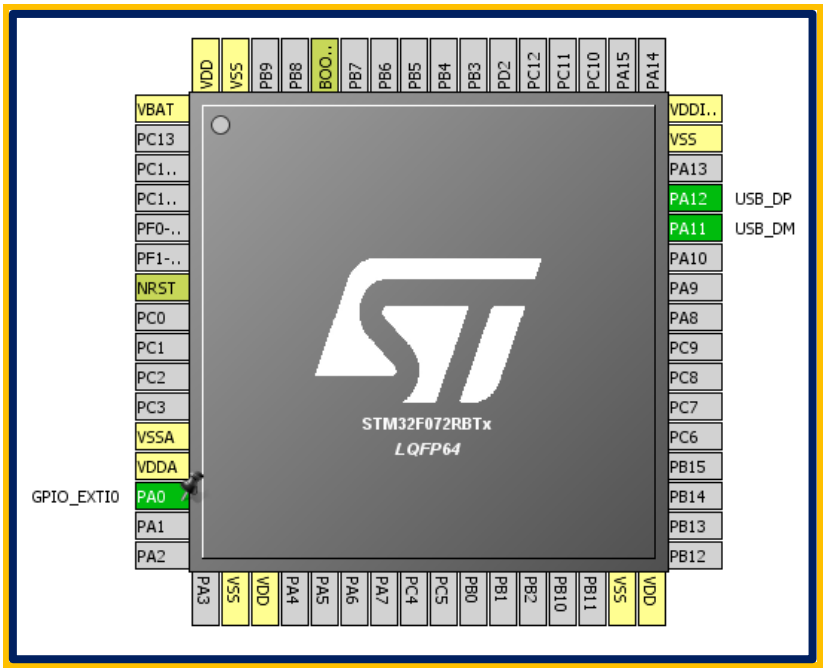


图 28 管脚配置

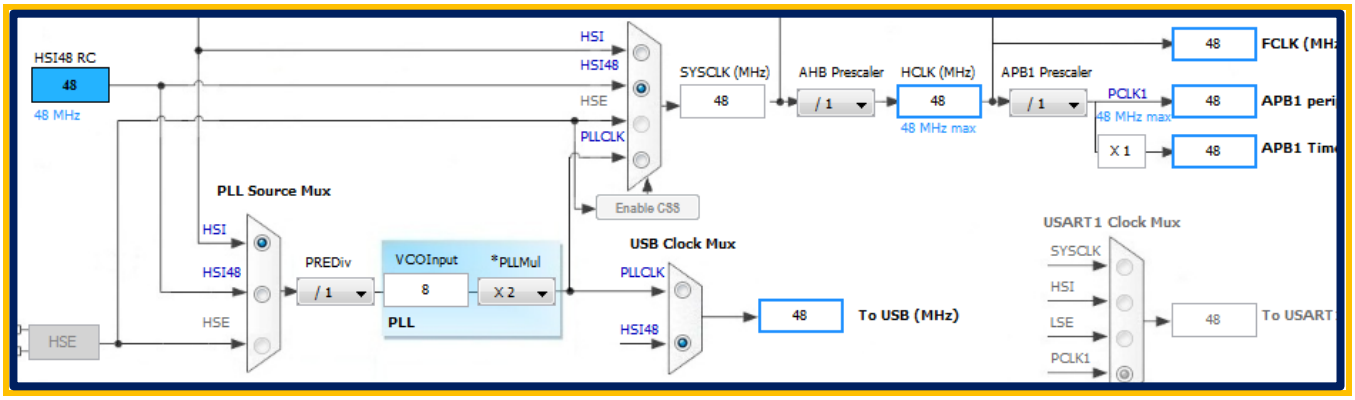


图 29 时钟树配置

使用内部 48M 的 HSI48 RC 作为时钟源。

图 30 选择 CDC 作为 USB 类

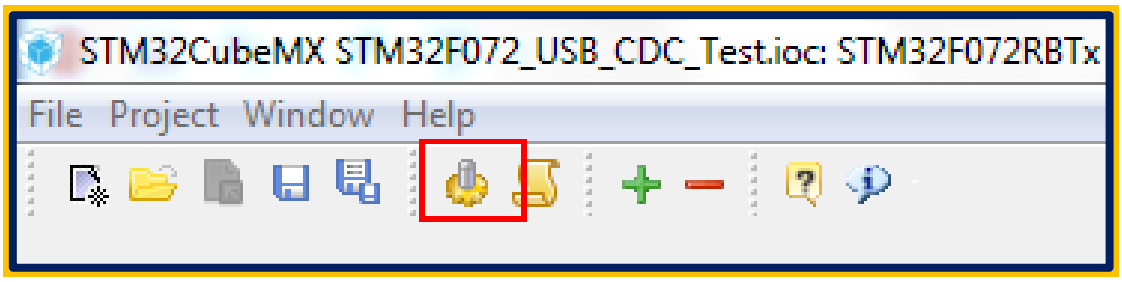


图 31 点击齿轮按键准备生产代码

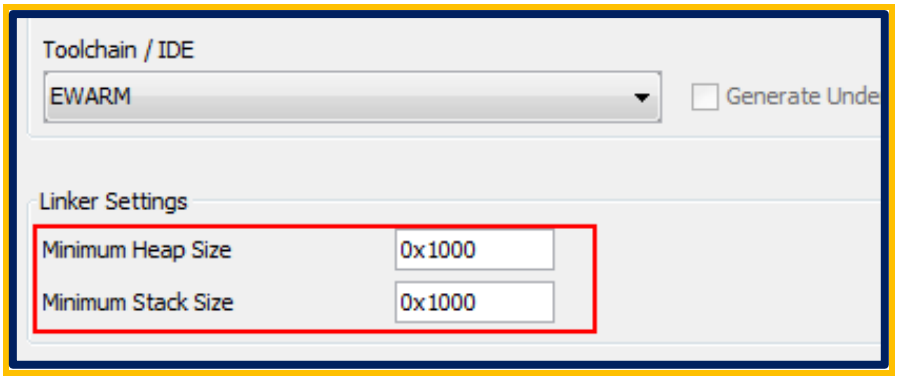


图 32 配置堆栈大小，和 IAR 工程

最终生成的代码工程与 USB CDC 类软件框架的对应关系:

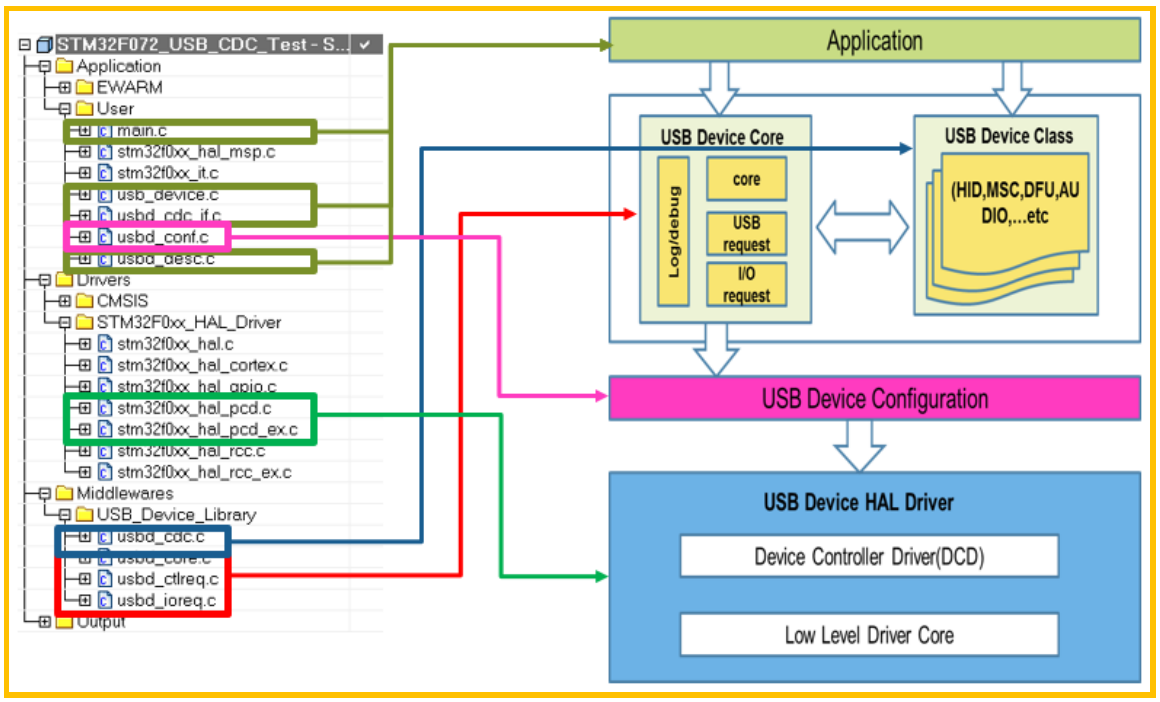


图 33 USB CDC 工程与软件框架之间的对应关系

4.3 添加测试代码

为了更好的验证通信，我们需要添加点测试代码:

```
static int8_t CDC_Receive_FS (uint8_t* Buf, uint32_t *Len)
{
  /* USER CODE BEGIN 6 */
      HandleReceiveData(Buf,*Len);

  USBD_CDC_SetRxBuffer(&hUsbDeviceFS, &Buf[0]);
  USBD_CDC_ReceivePacket(&hUsbDeviceFS);
  return (USBD_OK);
  /* USER CODE END 6 */
}
```

在接收回调中，我们将接收到的数据转给 HandleReceiveData 函数处理:

```
void HandleReceiveData(uint8_t *pData, uint32_t len)
{
      CDC_Transmit_FS(pData,len);
}
```

而在 HandleReceiveData 函数中我们将收到的数据原样返回给 Host 端，这样一来，Host 端的串口工具将发送什么就将收到什么。

```
uint8_t SendData[256];
static uint8_t StartFlag =0;
void HAL_GPIO_EXTI_Callback(uint16_t GPIO_Pin)
{
      //CDC_Transmit_FS(SendData,64);
```

```
        if(StartFlag ==0)
        {
                StartFlag =1;
        }
        else
        {
                StartFlag =0;
        }
}
```

另一方面，我们定义了一全局变量 StartFlag，它用来标志是否循环从 Device 端向 Host 端主动发送数据，其值由外部按键控制。然后在 Main 函数内的 while(1)循环内添加如下测试代码:

```
  /* USER CODE BEGIN WHILE */
  while (1)
  {
  /* USER CODE END WHILE */

  /* USER CODE BEGIN 3 */
            if(StartFlag ==1)
            {
                  if(hUsbDeviceFS.dev_state ==USBD_STATE_CONFIGURED)
                  {
                        CDC_Transmit_FS(SendData,60);
                  }
            }
  }
  /* USER CODE END 3 */
```

只要 StartFlag 标志为 1，在枚举结束后则不断向 Host 端发送数据。

4.4 验证结果

在编译完并将代码烧录进 MCU 后，我们首先验证 PC 端通过串口工具发送数据的情况:

图 34 发送什么返回什么

如上图所示，串口工具发送 63 个字节到 Device 端后，能够接收到从 Device 端返回到的一模一样的数据，这说明发送与接收都是正常的。

图 35 只收

在按下用户按键后，串口工具能够无限收到来自 Device 端的数据。

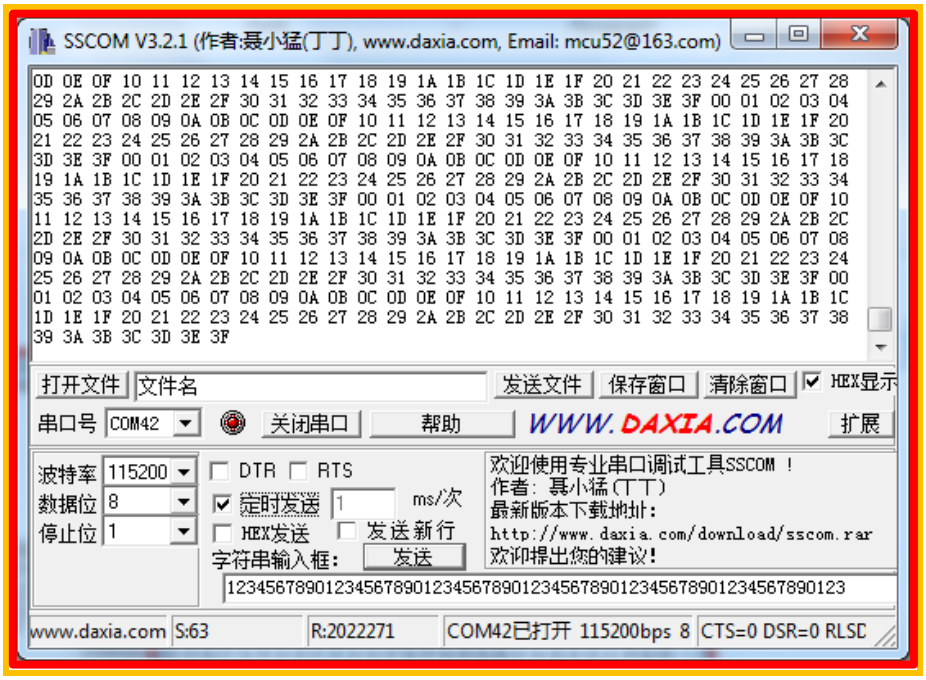


图 36 收发同时进行

收发同时进行也是正常的。至此，USB CDC 设备端的收发验证均未正常。

4.5 注意事项

- STM32CubeMx 默认生成的工程在发送 64 整数倍数据的时候 PC 端收不到，这个问题已经在之前的文章中有所描述，这里不再重复。

- CDC device 端若无限向 PC 端发送数据，若 PC 端没有及时读走数据，导致 PC 端接收缓存爆满，此时 PC 端回复 NACK，此时会导致发送返回 BUSY.