AN12458
USB to Virtual COM on LPC54018 and LPC5500

Rev. 0 — June 2019 Application Note

Contents

1 Introduction

Serial Asynchronous Receiver/Transmitter (UART) is often used as a standard peripheral in MCUs and embedded applications. With the development of PC, serial ports have gradually disappeared from PC. For embedded engineers and developers, they encountered the problem that cannot be debugged directly through the UART. In this case, a USB interface can be used to implement the function of the Virtual COM Port (VCOM), so that communication can be performed through the VCOM on the PC and the embedded system. In some embedded applications, if the number of UARTs on the MCU does not meet the requirements of the system, the function of USB to VCOM can also be used. The function of USB to VCOM can be implemented by using the common AT commands in the sub-class of the abstract control model in the CDC class specified by the USB protocol.

This application note describes how to implement a USB to multiple VCOMs functions on the LPC54018 EVK and LPC55S69 EVK boards. A USB device can support one or more VCOMs and the number of VCOM depends on the number of physical endpoints supported by USB device. The LPC54018 and LPC55S69's Full-Speed (FS) USB device supports 10 physical endpoints, which can only support up to two VCOMs. High-Speed (HS) USB device supports 12 physical endpoints, and also supports up to two VCOMs. To support more VCOMs, based on SDK code, this application note implement the function of supporting four VCOMs by using FS USB device and HS USB device together. The development tool is MCUXpresso IDE.

2 USB descriptor configuration

The function of a USB to multiple VCOMs can be implemented by using the USB composite class. The composite class is a special USB class that can implement multiple different functions in a USB device. For example, a device can implement the **Mouse + Keyboard** function, or **VCOM + Keyboard** function. In fact, the USB composite class can implement almost any combination of USB functions. It is not just a combination of two functions but can be three or more. Therefore, you can use the composite class to implement the functions of two CDC subclasses or multiple CDC subclasses. However, due to the limitation of the number of physical endpoints, the HS USB device and FS USB device in LPC54018 and LPC55S69 can only support up to two CDC class. That is, one USB device can only support up to two VCOMs. If HS and FS USB devices are used together, the function of two USB devices to four VCOMs can be implemented.

The USB descriptor is equivalent to the business card of the USB device. It describes all the attributes and configurable information of the USB device, such as, the class, interface information, and endpoint information. If the descriptor of the device is obtained, the type, purpose and the parameters of the communication of the device, etc. are known, and the USB host can configure it so that both parties of the communication work with the same parameters.

A CDC class device consists of two subclass interfaces: a CDC class interface and a data class interface.

- The CDC class interface uses a standard interface descriptor that requires an interrupt input endpoint.
- The data class interface is the interface that the communication device must configure. It requires two endpoints: a Bulk IN endpoint and a Bulk OUT endpoint.

If you want to implement a CDC class device, two interfaces and three endpoints are required. If you want to implement two CDC classes with one USB device, four interfaces and six endpoints are required. The descriptor structure of the composite class contains two CDC subclasses used in this application note, as shown in Figure 1 on page 2.

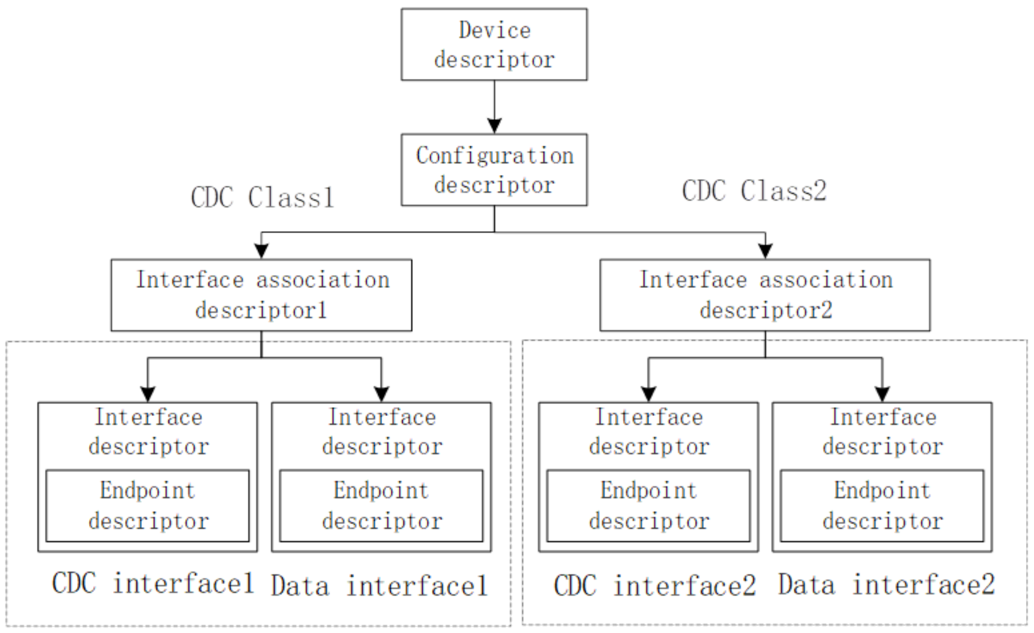


Figure 1. USB descriptor block diagram

In Figure 1 on page 2,

- CDC interface 1 and data interface 1 are associated together by interface association descriptor 1 to describe a VCOM function.
- CDC interface 2 and data interface 2 are associated together by interface association descriptor 2 to describe another VCOM function.

For details on the USB descriptor, see Chapter 9.6, Standard USB Descriptor Definitions, in USB Specification 2.0 and SDK code.

3 Physical interface usage

As described in USB descriptor configuration on page 1, each CDC class requires three physical endpoints, and two CDC classes require six physical endpoints. When both FS and HS USB device are used, the usage of physical endpoints is shown in Table 1. Physical endpoint usage of FS USB device on page 2 and Table 2. Physical endpoint usage of HS USB device on page 3.

Table 1. Physical endpoint usage of FS USB device

Logical endpoint	Physical endpoint	Direction	Endpoint type	Packet size (byte)	Use or not
EP0	0	OUT	Control	64	YES
EP0	1	IN	Control	64	YES
EP1	2	OUT	—	—	NO
EP1	3	IN	CDC1 interrupt	512	YES
EP2	4	OUT	—	—	NO
EP2	5	IN	CDC2 interrupt	512	YES
EP3	6	OUT	CDC1 bulk	512	YES
EP3	7	IN	CDC1 bulk	512	YES

Table continues on the next page...

Table 1. Physical endpoint usage of FS USB device (continued)

Logical endpoint	Physical endpoint	Direction	Endpoint type	Packet size (byte)	Use or not
EP4	8	OUT	CDC2 bulk	512	YES
EP4	9	IN	CDC2 bulk	512	YES

Two CDC classes use six non-control endpoints, plus two control endpoints, which require eight physical endpoints. As FS USB device supports only 10 physical endpoints, it can only support two VCOMs.

Unlike the FS USB device, the HS USB device supports 12 physical (6 logical) endpoints.

Table 2. Physical endpoint usage of HS USB device on page 3 shows the physical endpoint usage of the HS USB.

Table 2. Physical endpoint usage of HS USB device

Logical endpoint	Physical endpoint	Direction	Endpoint type	Packet size (byte)	Use or not
EP0	0	OUT	Control	64	YES
EP0	1	IN	Control	64	YES
EP1	2	OUT	—	—	NO
EP1	3	IN	CDC1 interrupt	512	YES
EP2	4	OUT	—	—	NO
EP2	5	IN	CDC2 interrupt	512	YES
EP3	6	OUT	CDC1 bulk	512	YES
EP3	7	IN	CDC1 bulk	512	YES
EP4	8	OUT	CDC2 bulk	512	YES
EP4	9	IN	CDC2 bulk	512	YES
EP5	10	OUT	—	—	NO
EP5	11	IN	—	—	NO

NOTE

The packet sizes described in Table 1. Physical endpoint usage of FS USB device on page 2 and Table 2. Physical endpoint usage of HS USB device on page 3 are not the maximum values.

Each CDC class requires two physical endpoints in the IN direction and one endpoint in the OUT direction. Since there is only one unused IN endpoint in the HS USB device, the HS USB device supports only two VCOMs.

4 USB SRAM usage

Each physical EndPoint (EP) needs a buffer to store the received data or data to be sent. This section describes the configuration of the USB endpoint buffer. The HS USB device can only use the **USB SRAM (0x4010 0000-0x4010 2000)** area as the EP buffer and the HS USB EPLIST must also be placed in the USB SRAM. In order to be compatible with HS and FS USB device driver in the SDK code, the FS USB device also uses the USB SRAM area as the EP buffer.

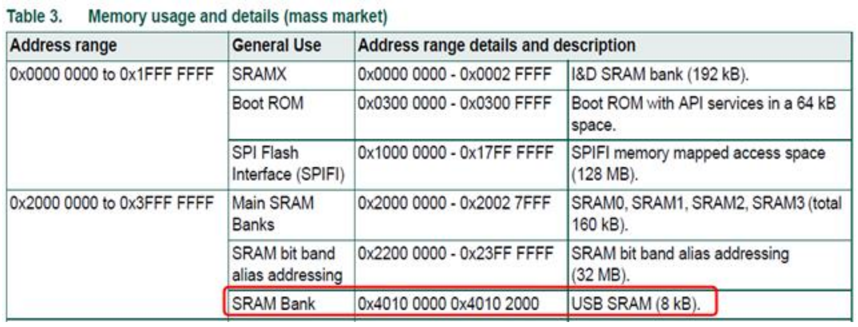


Table 3. Memory usage and details (mass market)

Address range	General Use	Address range details and description	
0x0000 0000 to 0x1FFF FFFF	SRAMX	0x0000 0000 - 0x0002 FFFF	I&D SRAM bank (192 kB).
	Boot ROM	0x0300 0000 - 0x0300 FFFF	Boot ROM with API services in a 64 kB space.
	SPI Flash Interface (SPIFI)	0x1000 0000 - 0x17FF FFFF	SPIFI memory mapped access space (128 MB).
0x2000 0000 to 0x3FFF FFFF	Main SRAM Banks	0x2000 0000 - 0x2002 7FFF	SRAM0, SRAM1, SRAM2, SRAM3 (total 160 kB).
	SRAM bit band alias addressing	0x2200 0000 - 0x23FF FFFF	SRAM bit band alias addressing (32 MB).
	SRAM Bank	0x4010 0000 0x4010 2000	USB SRAM (8 kB).

Figure 2. Distribution of USB SRAM

The configuration of the USB endpoint buffer is done by **EPLISTSTART** register and **DATABUFSTART** register. The **EPLISTSTART** register indicates the start address of the USB EP Command/Status List (EPLIST). In the SDK code, the **EPLISTSTART** register is configured in the `USB_DeviceLpc3511IpSetDefaultState()` function, pointing to the **s_EpCommandStatusList1** global array.

```
lpc3511IpState-> registerBase -> EPLISTSTART = ( uint32_t)lpc3511IpState-> epCommandStatusList;
```

The definition of the `s_EpCommandStatusList1` array is as follows:

```
USB_CONTROLLER_DATA USB_RAM_ADDRESS_ALIGNMENT(256) static uint32 _t
s_EpCommandStatusList1[((USB_DEVICE_IP3511_ENDPOINTS_NUM)) * 4];
```

It is a 256-byte aligned global data stored in a 256-byte space in the USB RAM. When using an endpoint for data transfer, the buffer used by each endpoint is as shown in Figure 3 on page 4.

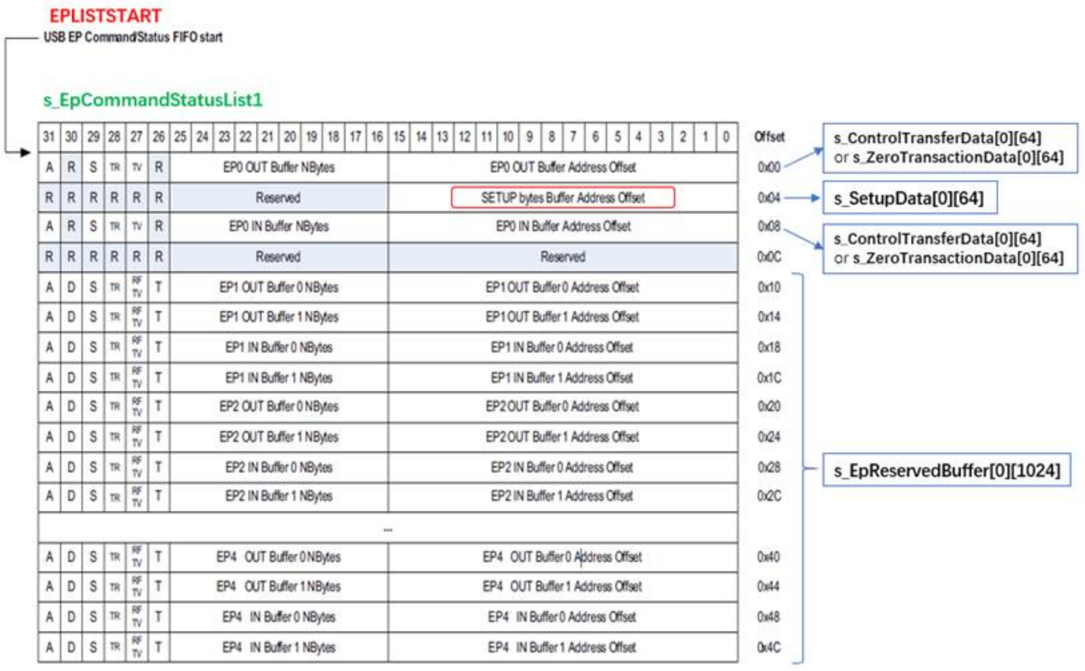

EPLISTSTART
USB EP Command/Status FIFO start
s_EpCommandStatusList1
31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
Offset
A R S TR TV R EP0 OUT Buffer NBytes EP0 OUT Buffer Address Offset 0x00
R R R R R R Reserved SETUP bytes Buffer Address Offset 0x04
A R S TR TV R EP0 IN Buffer NBytes EP0 IN Buffer Address Offset 0x08
R R R R R R Reserved Reserved 0x0C
A D S TR RF TV T EP1 OUT Buffer 0 NBytes EP1 OUT Buffer 0 Address Offset 0x10
A D S TR RF TV T EP1 OUT Buffer 1 NBytes EP1 OUT Buffer 1 Address Offset 0x14
A D S TR RF TV T EP1 IN Buffer 0 NBytes EP1 IN Buffer 0 Address Offset 0x18
A D S TR RF TV T EP1 IN Buffer 1 NBytes EP1 IN Buffer 1 Address Offset 0x1C
A D S TR RF TV T EP2 OUT Buffer 0 NBytes EP2 OUT Buffer 0 Address Offset 0x20
A D S TR RF TV T EP2 OUT Buffer 1 NBytes EP2 OUT Buffer 1 Address Offset 0x24
A D S TR RF TV T EP2 IN Buffer 0 NBytes EP2 IN Buffer 0 Address Offset 0x28
A D S TR RF TV T EP2 IN Buffer 1 NBytes EP2 IN Buffer 1 Address Offset 0x2C
...
A D S TR RF TV T EP4 OUT Buffer 0 NBytes EP4 OUT Buffer 0 Address Offset 0x40
A D S TR RF TV T EP4 OUT Buffer 1 NBytes EP4 OUT Buffer 1 Address Offset 0x44
A D S TR RF TV T EP4 IN Buffer 0 NBytes EP4 IN Buffer 0 Address Offset 0x48
A D S TR RF TV T EP4 IN Buffer 1 NBytes EP4 IN Buffer 1 Address Offset 0x4C
s_ControlTransferData[0][64] or s_ZeroTransactionData[0][64]
s_SetupData[0][64]
s_ControlTransferData[0][64] or s_ZeroTransactionData[0][64]
s_EpReservedBuffer[0][1024]

Figure 3. Content in the USB EPLIST

NOTE

The content in the Figure 3 on page 4 is based on FS USB device and all OUT/IN mentioned in this application note is relative to the USB host.

The EP OUT/IN buffer address offset field in the EPLIST stores **bits 6-21** of the corresponding buffer array. `s_SetupData`, `s_ControlTransferData`, `s_ZeroTransactionData`, and `s_EpReservedBuffer` are global arrays.

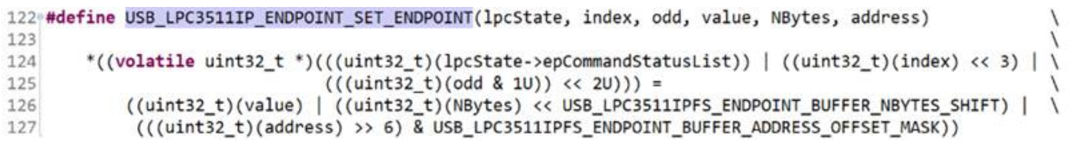
- The `s_SetupData` array is used to receive setup packet data.
- The `s_ControlTransferData` array is used as the buffer for control endpoints: EP0 OUT and EP0 IN. For example, when processing standard request, before using EP0 IN buffer to send data or EP0 OUT buffer to receive data, the bits 6-21 of the address of `s_ControlTransferData[0]` needs to be written to the EP IN/OUT Buffer Address Offset field.
- The `s_ZeroTransactionData` array is used to send a 0-length packet to the host.
- The `s_EpReservedBuffer` array is the input and output buffer of the non-control endpoint, each physical endpoint uses 512 bytes in the array.

Before using the OUT EP buffer to receive data or using the IN EP buffer to send data to the USB host, the corresponding EP OUT/IN buffer address offset field in the EPLIST needs to be updated, writing the bits 6-21 of the address of the array to be used to corresponding address offset field. Then set the **ACTIVE bit in the EPLIST** of the corresponding endpoint to trigger the hardware to receive or send data.

The method of updating the EPLIST in the SDK code is as follows:

```
USB_LPC3511IP_ENDPOINT_SET_ENDPOINT(
        lpc3511IpState, endpointIndex, odd,
        (epState->stateUnion.stateBitField.epControlDefault  <<
USB_LPC3511IP_ENDPOINT_CONFIGURE_BITS_SHIFT) |
            USB_LPC3511IP_ENDPOINT_ACTIVE_MASK, length, (uint32_t)buffer);
```

Figure 4 on page 5 shows the contents of the `USB_LPC3511IP_ENDPOINT_SET_ENDPOINT` macro definition.



```
#define USB_LPC3511IP_ENDPOINT_SET_ENDPOINT(lpcState, index, odd, value, NBytes, address)          \
                                                                                                  \
    *((volatile uint32_t *)(((uint32_t)(lpcState->epCommandStatusList)) | ((uint32_t)(index) << 3) | \
                            (((uint32_t)(odd & 1U)) << 2U))) =                                    \
        ((uint32_t)(value) | ((uint32_t)(NBytes) << USB_LPC3511IPFS_ENDPOINT_BUFFER_NBYTES_SHIFT) |  \
         (((uint32_t)(address) >> 6) & USB_LPC3511IPFS_ENDPOINT_BUFFER_ADDRESS_OFFSET_MASK))
```

Figure 4. Contents of `USB_LPC3511IP_ENDPOINT_SET_ENDPOINT` macro definition

When the USB host and the USB device transmit data through the endpoint, only the address offset is insufficient, and the base address of these buffer arrays is also required. The base address is set in the **DATABUFSTART** register. The lower 22 bits of the DATABUFSTART register are 0, which points to a 4 M-aligned memory space. Since these EP buffer arrays are stored in the USB RAM (`0x40100000`-`0x401020000`) and the USB RAM is contained in a 4 M space starting at `0x40000000`, the value of the **DATABUFSTART** register should be `0x40000000`. The configuration method of **DATABUFSTART** register is as follows:

```
(( USB_Type *)(lpc3511IpState-> registerBase ))-> DATABUFSTART = (uint32_t )lpc3511IpState->
setupData ;
```

The `s_SetupData`, `s_ControlTransferData`, `s_ZeroTransactionData`, and `s_EpReservedBuffer` arrays are all stored in a 4 M-aligned memory pointed to by **DATABUFSTART** with a base address of `0x40000000`. Using the **USB_GLOBAL** attribute to define the `s_SetupData`, `s_ControlTransferData`, `s_ZeroTransactionData`, `s_EpReservedBuffer` arrays, placing these arrays in the **m_usb_global section** and placing the **m_usb_global section** in the USB RAM space. When using HS or FS USB device alone, the USB RAM usage is as shown in Figure 5 on page 6.

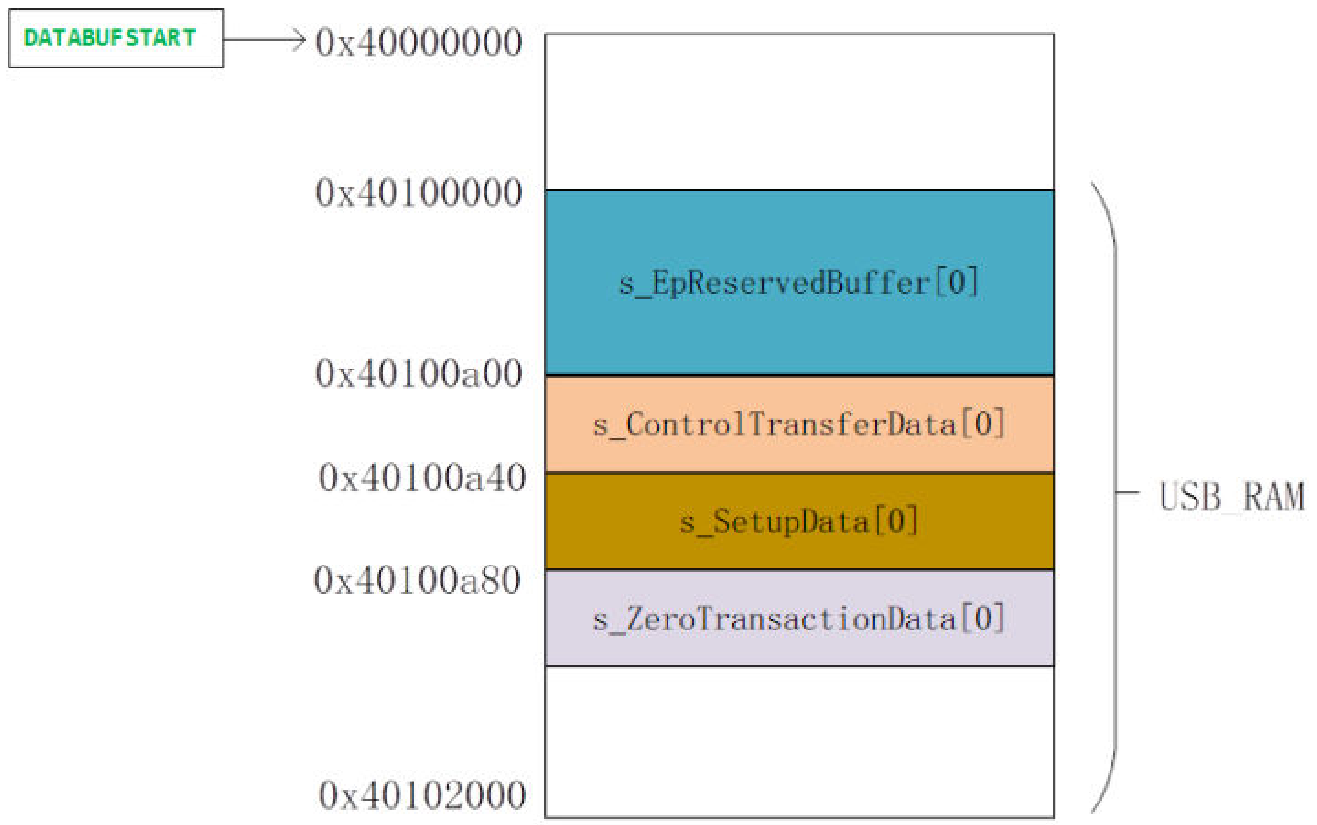


Figure 5. USB RAM usage when using a USB device

When using both HS and FS USB device, the USB RAM usage is as shown in Figure 6 on page 7.

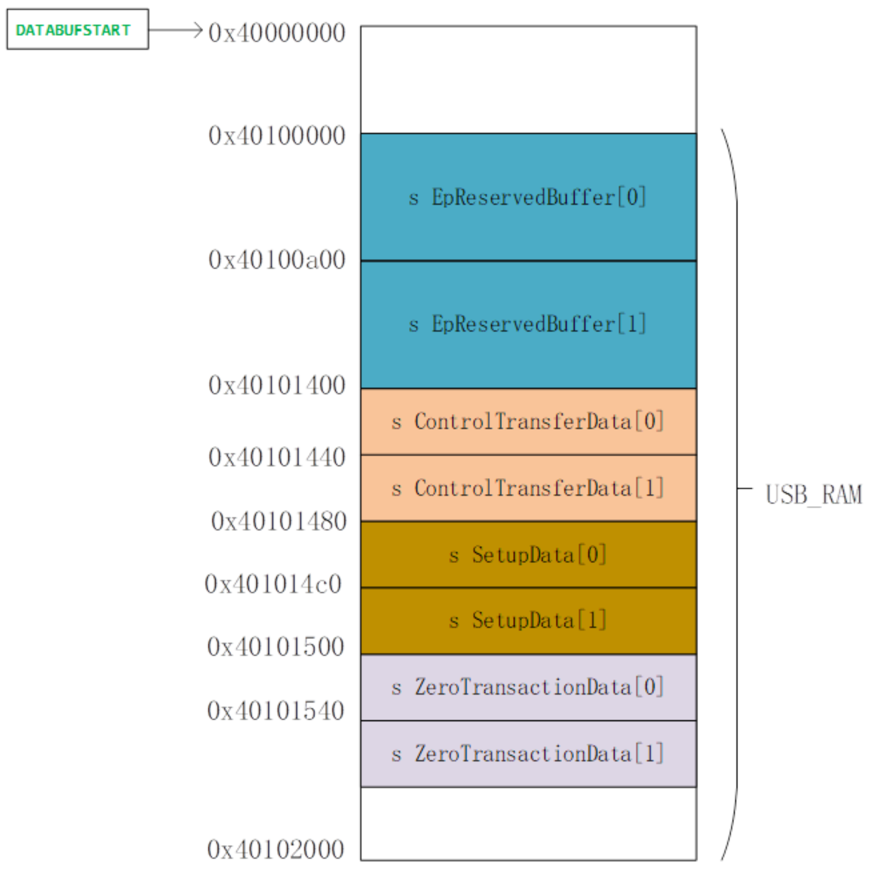


Figure 6. USB RAM usage when using both FS and HS USB

NOTE

In order to allow FS and HS USB devices to use USB RAM together, do not exceed the 8 K space of the USB RAM. In this application note, the size of the macro definition, **USB_DEVICE_IP3511_ENDPOINT_RESERVED_BUFFER_SIZE**, is modified from (5 × 1024) to (5 × 512).

5 Software work flow chart

The code used in this application note is based on the `usb_device_composite_cdc_vcom_cdc_vcom_lite` example in the SDK. The basic workflow of the Device stack depends on callback functions and function calls. The callback functions notify all state changes and data requests of the device stack to application.

There are two types of callback functions in the device stack:

- **Device callback function** - It notifies the state of device stack to the application.
- **Endpoint callback function** - It notifies the data transfer result of the corresponding endpoint to application. The control endpoint callback function handles all USB standard requests and class requests.

The callback function in this Lite version code is implemented by the application. Figure 7 on page 8 describes the processing of the callback function.

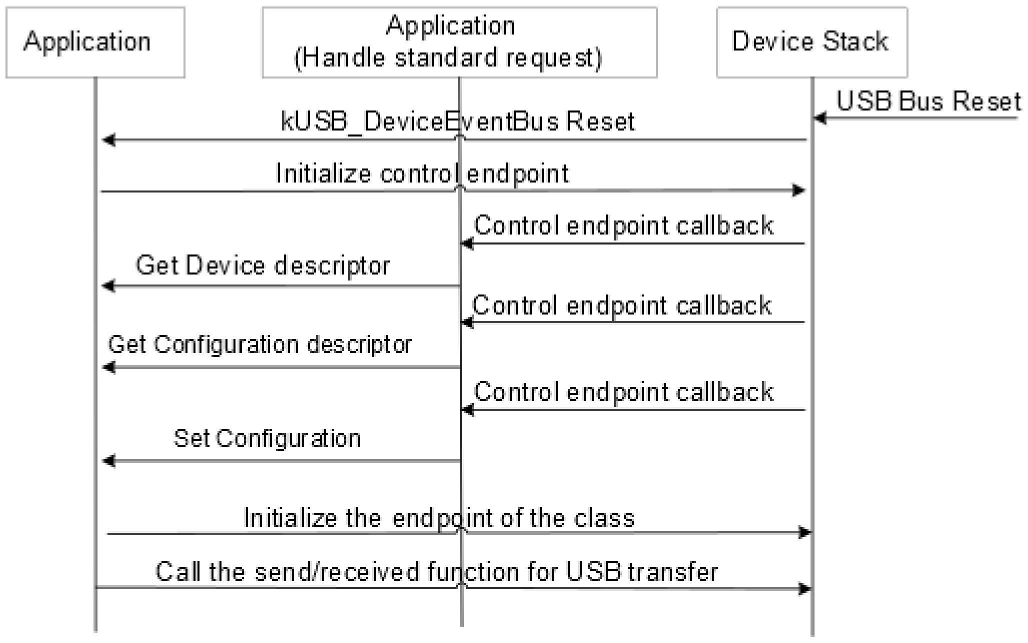

Application
Application (Handle standard request)
Device Stack
USB Bus Reset
kUSB_DeviceEventBus Reset
Initialize control endpoint
Control endpoint callback
Get Device descriptor
Control endpoint callback
Get Configuration descriptor
Control endpoint callback
Set Configuration
Initialize the endpoint of the class
Call the send/received function for USB transfer

Figure 7. Callback function processing

When the USB host recognizes that a USB device is plugged into the USB interface, it will start an enumeration process. The essence of USB enumeration is the process in which the USB host obtains the parameter information of the USB device and configures the configurable parameters. The USB enumeration process is mainly done in the USB interrupt service function. The workflow of the FS and HS USB interrupt service functions in this application note is the same. Here, FS USB is taken as an example, and the processing flow of the USB0 interrupt service function is shown in Figure 8 on page 9.

5.1 USB interrupt service function flow chart

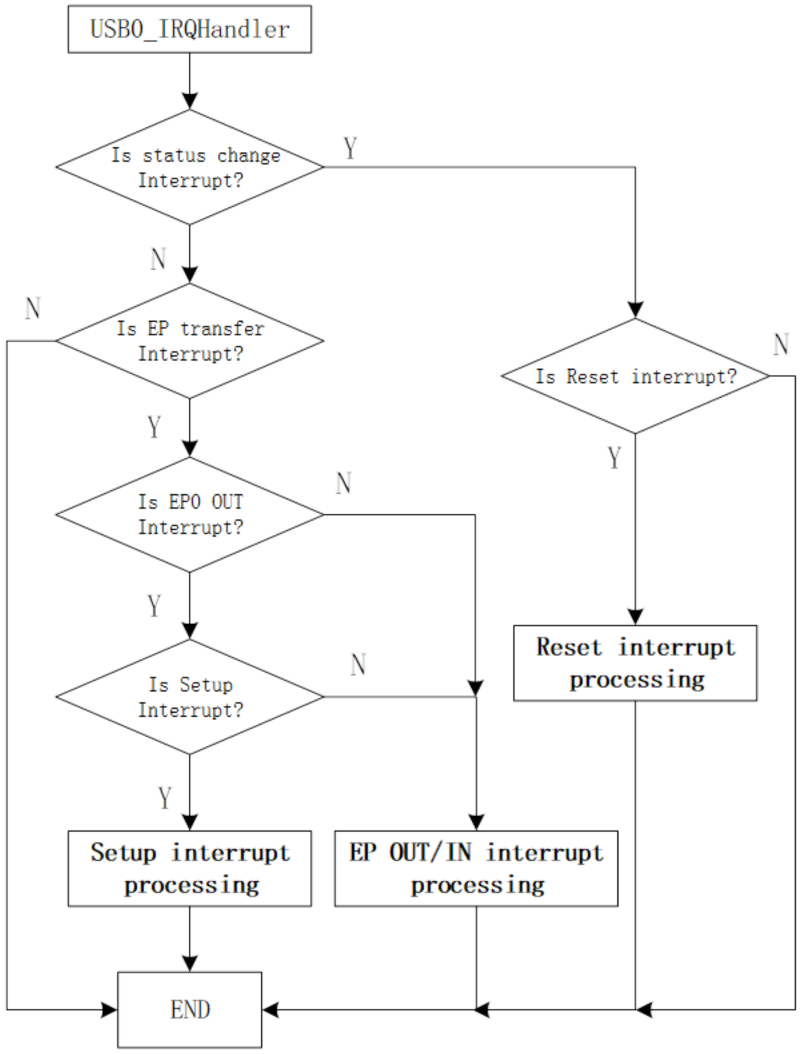

USB0_IRQHandler
Is status change Interrupt?
Y
N
Is EP transfer Interrupt?
N
Y
Is EP0 OUT Interrupt?
N
Y
Is Setup Interrupt?
N
Y
Setup interrupt processing
EP OUT/IN interrupt processing
Is Reset interrupt?
N
Y
Reset interrupt processing
END

Figure 8. Flowchart of FS USB interrupt service function

After the USB0 interrupt occurs and the USB0_IRQHandler interrupt service function is called, the program first determines whether the interrupt is caused by USB status change (Reset/Suspend/Resume) or the endpoint transfer interrupt (EP IN/OUT interrupt). If it is an EP0 OUT interrupt, there are two cases: **Setup interrupt** and normal EP0 OUT interrupt.

5.1.1 Reset interrupt process

If the interrupt is caused by the status change and it is a reset interrupt, the program will execute the flow shown in Figure 9 on page 10.

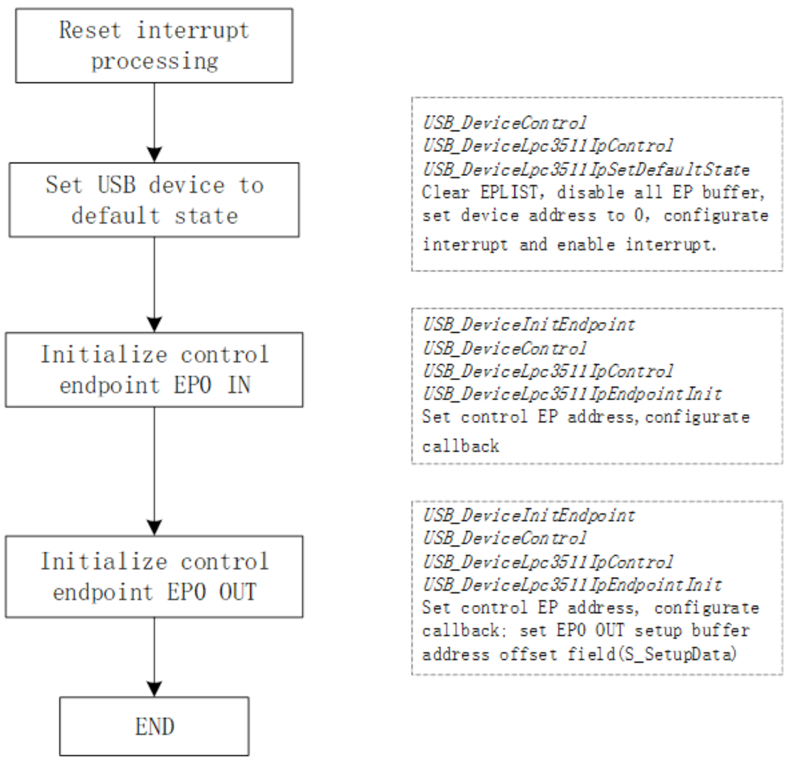

Reset interrupt processing
Set USB device to default state
Initialize control endpoint EP0 IN
Initialize control endpoint EP0 OUT
END
USB_DeviceControl
USB_DeviceLpc3511IpControl
USB_DeviceLpc3511IpSetDefaultState
Clear EPLIST, disable all EP buffer, set device address to 0, configurate interrupt and enable interrupt.
USB_DeviceInitEndpoint
USB_DeviceControl
USB_DeviceLpc3511IpControl
USB_DeviceLpc3511IpEndpointInit
Set control EP address, configurate callback
USB_DeviceInitEndpoint
USB_DeviceControl
USB_DeviceLpc3511IpControl
USB_DeviceLpc3511IpEndpointInit
Set control EP address, configurate callback; set EP0 OUT setup buffer address offset field(S_SetupData)

Figure 9. USB device reset process

In the reset interrupt, be sure to set the USB device to the default state and initialize the control endpoints, EP0 OUT and EP0 IN.

5.1.2 Setup interrupt processing

If it is an EP transfer interrupt and it is a setup interrupt, the program will execute the process shown in Figure 10 on page 11.

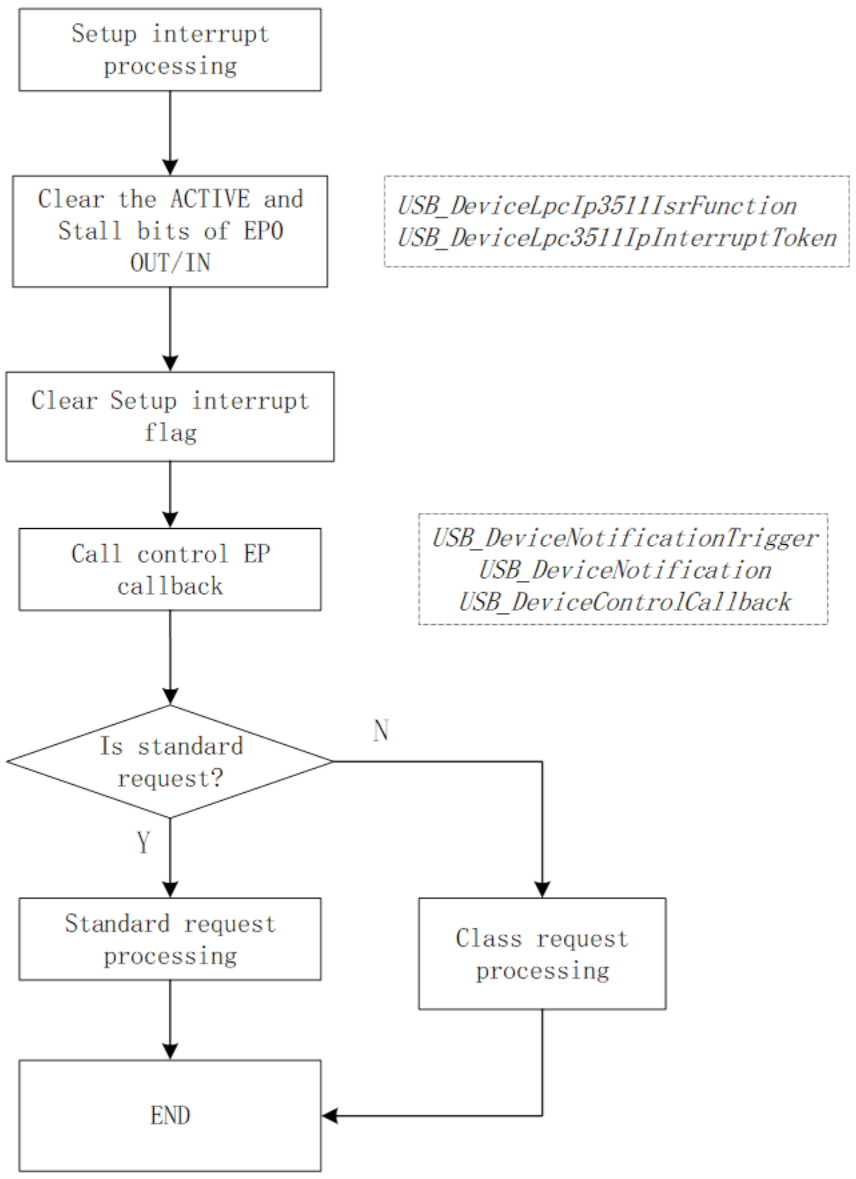

Setup interrupt processing
Clear the ACTIVE and Stall bits of EP0 OUT/IN
USB_DeviceLpcIp3511IsrFunction
USB_DeviceLpc3511IpInterruptToken
Clear Setup interrupt flag
Call control EP callback
USB_DeviceNotificationTrigger
USB_DeviceNotification
USB_DeviceControlCallback
Is standard request?
N
Y
Standard request processing
Class request processing
END

Figure 10. Setup interrupt processing flow chart

When the USB host sends a standard request or a class request to a USB device, it first sends a setup transaction. After receiving the setup transaction and successfully responding, the USB device generates a setup interrupt. The USB device determines what request is in the setup interrupt. Actions differ with the type of request.

5.1.2.1 Standard request processing

If the USB device determines that it is a standard request from the USB host according to the contents of the setup package, it will execute the process shown in Figure 11 on page 12.

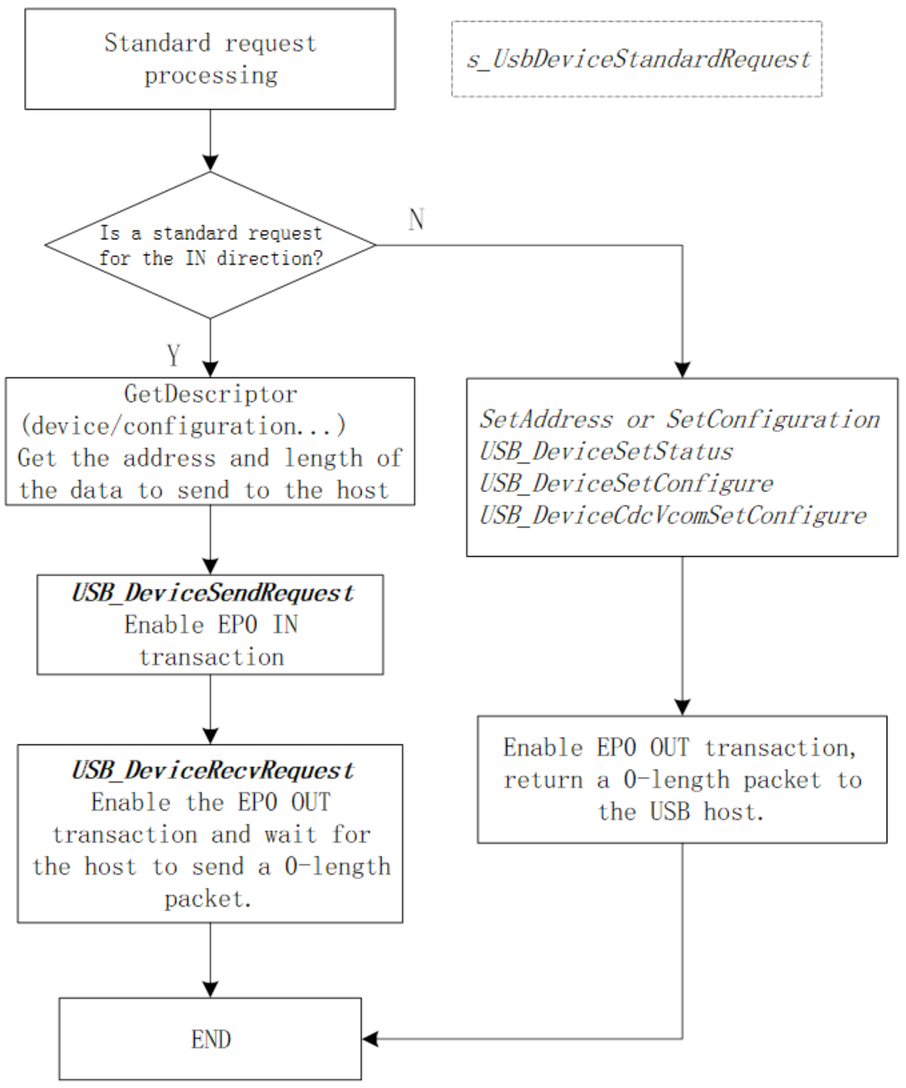

Standard request processing
s_UsbDeviceStandardRequest
Is a standard request for the IN direction?
N
Y
GetDescriptor (device/configuration...) Get the address and length of the data to send to the host
SetAddress or SetConfiguration
USB_DeviceSetStatus
USB_DeviceSetConfigure
USB_DeviceCdcVcomSetConfigure
USB_DeviceSendRequest Enable EP0 IN transaction
USB_DeviceRecvRequest Enable the EP0 OUT transaction and wait for the host to send a 0-length packet.
Enable EP0 OUT transaction, return a 0-length packet to the USB host.
END

Figure 11. Standard request processing

The program will judge whether this standard request is in the OUT direction or the IN direction based on the data in the setup package.

- If it is a request in the OUT direction, the USB device will ready to receive the next OUT transaction.
- If it is a request in the IN direction, the USB device will return an IN transaction to the USB host.

Table 3. Common Standard requests on page 12 described common standard requests.

Table 3. Common Standard requests

Standard request	Direction
`SetAddress`	OUT
`SetConfiguration`	OUT
`GetDescriptor` (Device/Configuration/String)	IN

5.1.2.2 Class request processing

CDC class devices also support some class requests in addition to the standard requests in Standard request processing on page 11. After the USB host obtains various descriptor information of the CDC class device, it sends some class requests to the USB device. Table 4. Common class requests on page 13 describes common class requests.

Table 4. Common class requests

Class request	Direction
`SetLineCoding`	OUT
`SetControlLineState`	OUT
`GetLineCoding`	IN

The `GetLineCoding` request is a request for the host to obtain the serial port attribute, including the baud rate, the stop bit, the check type, and the number of data bits. Table 5. GetLineCoding request structure on page 13 describes the structure of the `GetLineCoding` request.

Table 5. `GetLineCoding` request structure

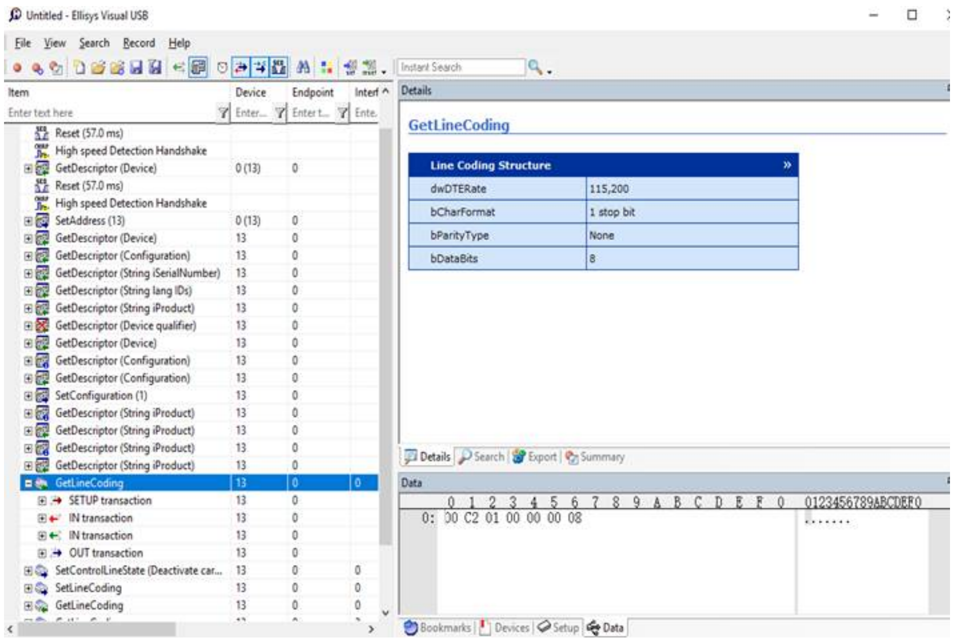
bmRequestType	bRequestCode	wValue	wIndex	wLength	Data
10100001B	**`GET_LINE_CODING`**	Zero	Interface	Size of structure	Line coding structure

Table 6. Line coding structure on page 13 describes the contents of line coding structure.

Table 6. Line coding structure

Offset	Field	Size/byte	Description
0	`dwDTERate`	4	Data terminal rate, in bits per second
4	`bCharFormat`	1	Stop bits 0: 1 Stop bit 1: 1.5 Stop bit 2: 2 Stop
5	`bParityType`	1	Parity 0: None 3: Mark 1: Odd 4: Space 2: Even
6	`bDataBits`	1	Data bits (5, 6, 7, 8, or 16)

Figure 12. on page 14 shows the contents of the line coding structure in this example.


Untitled - Ellisys Visual USB
File View Search Record Help
Instant Search
Details
Details Search Export Summary
Bookmarks Devices Setup Data

Item	Device	Endpoint	Interf
Reset (57.0 ms)			
High speed Detection Handshake			
GetDescriptor (Device)	0 (13)	0	
Reset (57.0 ms)			
High speed Detection Handshake			
SetAddress (13)	0 (13)	0	
GetDescriptor (Device)	13	0	
GetDescriptor (Configuration)	13	0	
GetDescriptor (String iSerialNumber)	13	0	
GetDescriptor (String lang IDs)	13	0	
GetDescriptor (String iProduct)	13	0	
GetDescriptor (Device qualifier)	13	0	
GetDescriptor (Device)	13	0	
GetDescriptor (Configuration)	13	0	
GetDescriptor (Configuration)	13	0	
SetConfiguration (1)	13	0	
GetDescriptor (String iProduct)	13	0	
GetDescriptor (String iProduct)	13	0	
GetDescriptor (String iProduct)	13	0	
GetDescriptor (String iProduct)	13	0	
GetLineCoding	13	0	0
SETUP transaction	13	0	
IN transaction	13	0	
IN transaction	13	0	
OUT transaction	13	0	
SetControlLineState (Deactivate car...	13	0	0
SetLineCoding	13	0	0
GetLineCoding	13	0	0

GetLineCoding

Line Coding Structure	
dwDTERate	115,200
bCharFormat	1 stop bit
bParityType	None
bDataBits	8

Data

```
     0  1  2  3  4  5  6  7  8  9  A  B  C  D  E  F  0   0123456789ABCDEF0
 0:  00 C2 01 00 00 00 08                                 .......
```

Figure 12. Data of `GetLineCoding`

As shown in Figure 12 on page 14, the configuration of VCOM: baud rate is 115200, 1 stop bit, no parity bit, 8 data bits.

If the USB device determines that it is a class request from the USB host according to the data in the setup package, execute the process shown in Figure 13 on page 15.

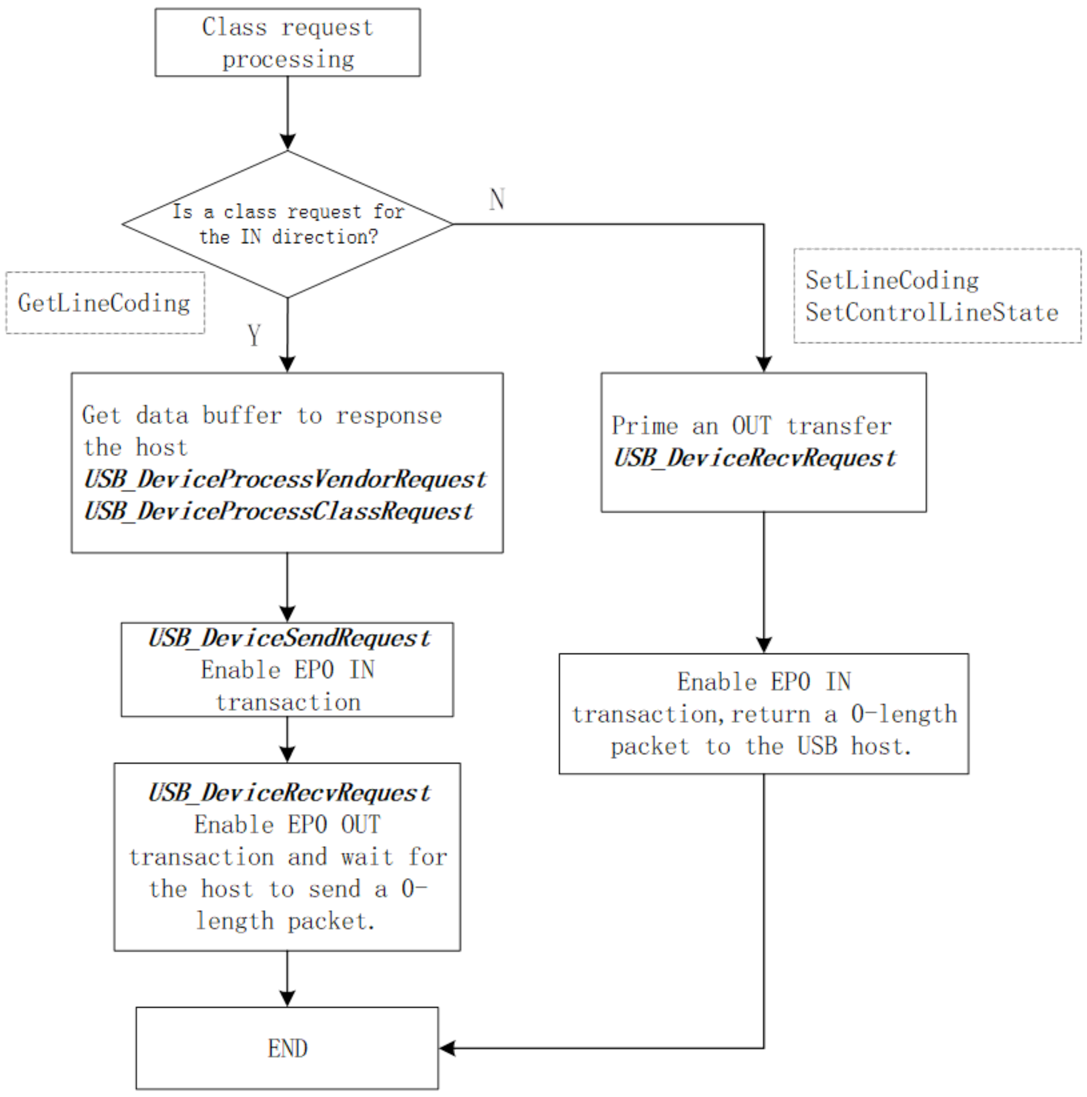

Class request processing
Is a class request for the IN direction?
N
Y
GetLineCoding
SetLineCoding
SetControlLineState
Get data buffer to response the host
USB_DeviceProcessVendorRequest
USB_DeviceProcessClassRequest
Prime an OUT transfer
USB_DeviceRecvRequest
USB_DeviceSendRequest
Enable EP0 IN transaction
Enable EP0 IN transaction, return a 0-length packet to the USB host.
USB_DeviceRecvRequest
Enable EP0 OUT transaction and wait for the host to send a 0-length packet.
END

Figure 13. Class request processing

The program will judge whether this class request is in the OUT direction or the IN direction based on the data in the setup package.

- If it is a request in the OUT direction, the USB device will ready to receive the next OUT transaction.
- If it is a request in the IN direction, the USB device will return an IN transaction to the USB host.

5.1.3 Endpoint interrupt processing

If it is an EP transfer interrupt, execute the process shown in Figure 14 on page 16 to receive and transmit the endpoint data.

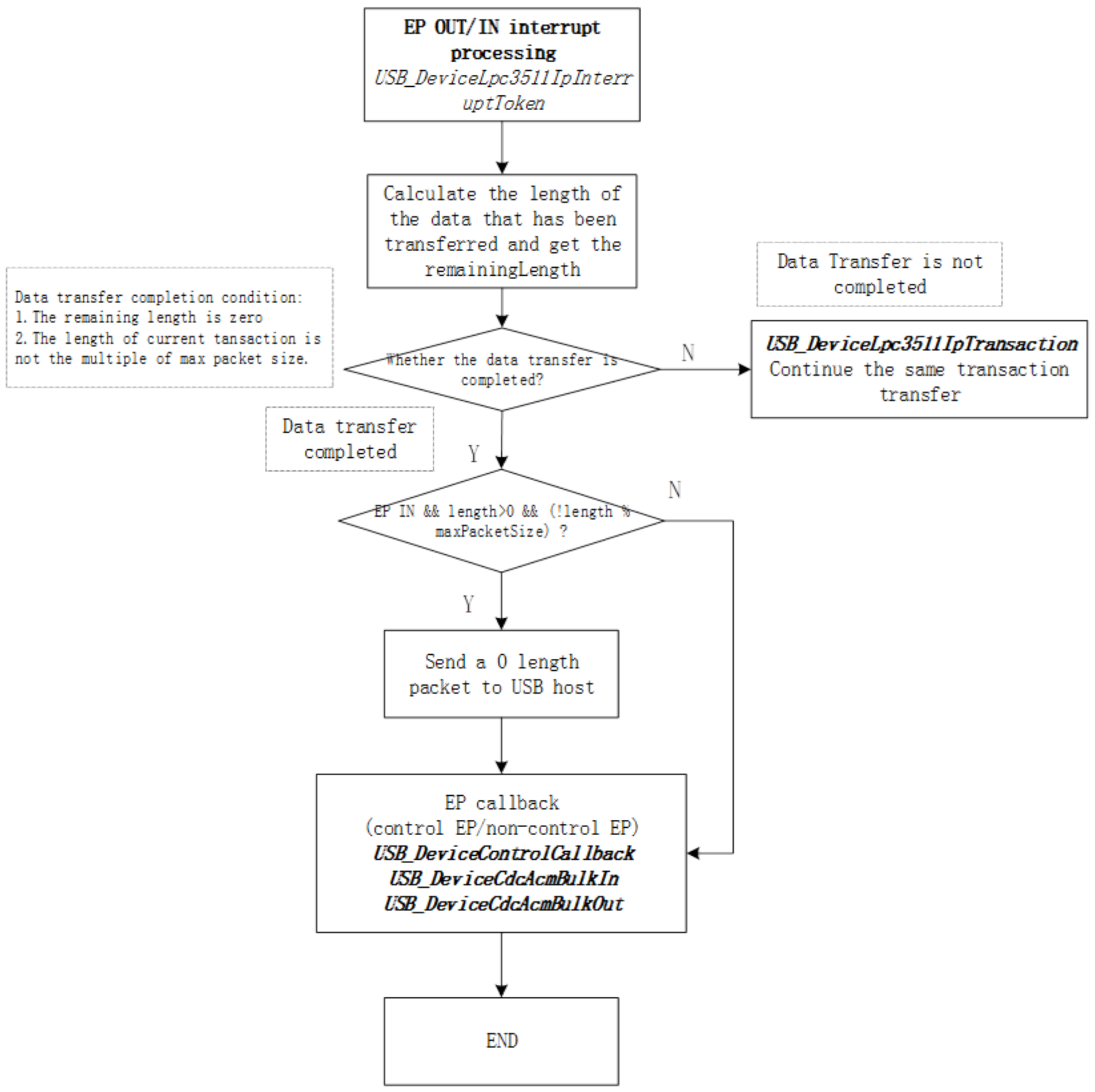

EP OUT/IN interrupt processing
USB_DeviceLpc3511IpInterruptToken
Calculate the length of the data that has been transferred and get the remainingLength
Data transfer completion condition:
1. The remaining length is zero
2. The length of current tansaction is not the multiple of max packet size.
Whether the data transfer is completed?
Data Transfer is not completed
N
USB_DeviceLpc3511IpTransaction
Continue the same transaction transfer
Data transfer completed
Y
EP IN && length>0 && (!length % maxPacketSize) ?
N
Y
Send a 0 length packet to USB host
EP callback
(control EP/non-control EP)
USB_DeviceControlCallback
USB_DeviceCdcAcmBulkIn
USB_DeviceCdcAcmBulkOut
END

Figure 14. Endpoint interrupt processing

When processing the endpoint transmission interrupt, the length of the data that has been transmitted and the remaining data length to be transmitted are first calculated to determine whether the data transmission is completed.

Transfer completed flag:

- Length > 0 and is not an integer multiple of the maximum packet length.
- RemainLength = = 0.

If the data transmission has not been completed, continue to transfer the same transaction. If the data transmission is completed and is an EP interrupt in the IN direction, and the last packet length is equal to the maximum packet length, then a 0-length data packet needs to be sent to inform the host that the transmission is complete. When the transmission is completed, the corresponding endpoint callback function is called. In this example, the endpoint callback functions provided by the SDK are as shown in Table 7. Endpoint callback on page 17.

Table 7. Endpoint callback

Endpoints	Endpoint callback
EP0 OUT	`USB_DeviceControlCallback()`
EP0 IN	`USB_DeviceControlCallback()`
EP1 IN	`USB_DeviceCdcAcmBulkIn()`
EP2 IN	`USB_DeviceCdcAcmBulkIn2()`
EP3 OUT	`USB_DeviceCdcAcmBulkOut()`
EP3 IN	`USB_DeviceCdcAcmBulkIn()`
EP4 OUT	`USB_DeviceCdcAcmBulkOut2()`
EP4 IN	`USB_DeviceCdcAcmBulkIn2()`

In this example, the function implemented by the callback function of four non-control endpoints is the same: update the length of data received and copy the data from EP buffer (`s_EpReserveBuffer`) to the global receive array (`s_currRecvBuf`). Then in the while loop, the data in the `s_currRecvBuf` array is returned to the host.

6 FS and HS USB configuration and verification

In this application note, the configuration of the FS USB and HS USB descriptors is identical, that is, they share the same descriptor array. The functions of the endpoint callback functions of FS and HS USB are the same: return the data received to the USB host. You can use one USB device or two USB devices by setting different macro definitions.

1. Use only FS USB device.

```
/*! @brief LPC USB IP3511 FS instance count */
#define USB_DEVICE_CONFIG_LPCIP3511FS (1U)
/*! @brief LPC USB IP3511 HS instance count */
#define USB_DEVICE_CONFIG_LPCIP3511HS (0U)
```

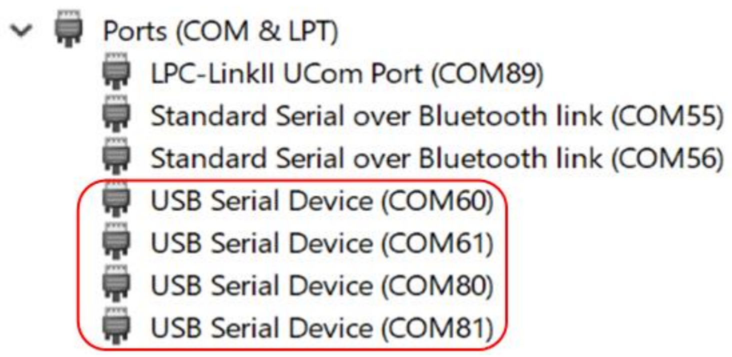
2. Use only HS USB device.

```
/*! @brief LPC USB IP3511 FS instance count */
#define USB_DEVICE_CONFIG_LPCIP3511FS (0U)
/*! @brief LPC USB IP3511 HS instance count */
#define USB_DEVICE_CONFIG_LPCIP3511HS (1U)
```

3. Use both FS USB device and HS USB device.

```
/*! @brief LPC USB IP3511 FS instance count */
#define USB_DEVICE_CONFIG_LPCIP3511FS (1U)
/*! @brief LPC USB IP3511 HS instance count */
#define USB_DEVICE_CONFIG_LPCIP3511HS (1U)
```

If both FS and HS USB are used, the USB host will recognize four VCOMs after the enumeration process is completed, as shown in Figure 15 on page 18.



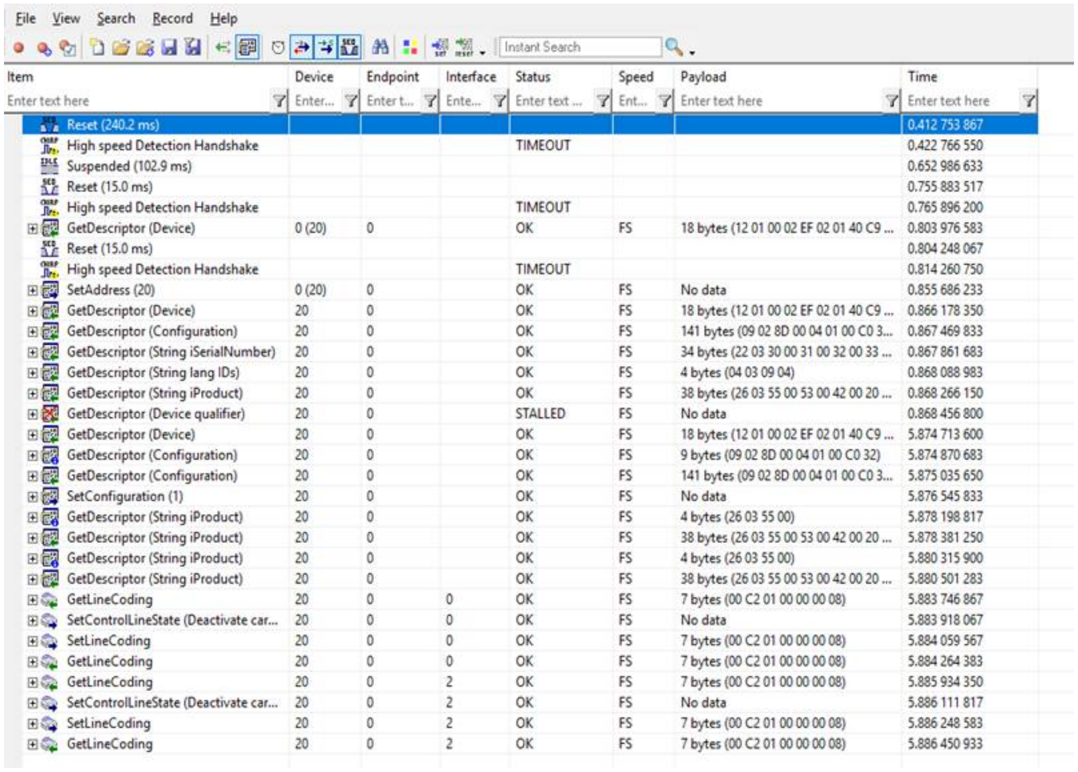
Ports (COM & LPT)
LPC-LinkII UCom Port (COM89)
Standard Serial over Bluetooth link (COM55)
Standard Serial over Bluetooth link (COM56)
USB Serial Device (COM60)
USB Serial Device (COM61)
USB Serial Device (COM80)
USB Serial Device (COM81)

Figure 15. USB host recognition result

NOTE

For the PC using the Windows, a CDC driver is required to be installed. For the driver installation, please refer to the `readme.pdf` file in the software.

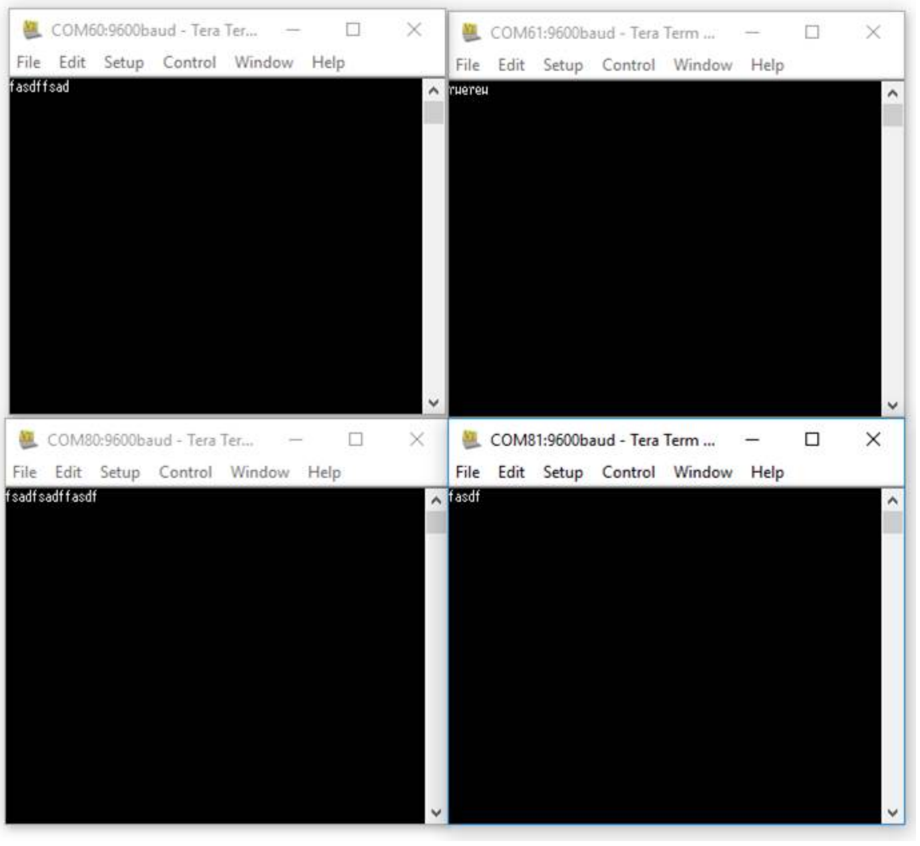
Figure 16 on page 18 shows the enumeration process of the USB host captured by the USB analyzer.



Item	Device	Endpoint	Interface	Status	Speed	Payload	Time
Enter text here	Enter...	Enter t...	Ente...	Enter text ...	Ent...	Enter text here	Enter text here
Reset (240.2 ms)							0.412 753 867
High speed Detection Handshake				TIMEOUT			0.422 766 550
Suspended (102.9 ms)							0.652 986 633
Reset (15.0 ms)							0.755 883 517
High speed Detection Handshake				TIMEOUT			0.765 896 200
GetDescriptor (Device)	0 (20)	0		OK	FS	18 bytes (12 01 00 02 EF 02 01 40 C9 ...	0.803 976 583
Reset (15.0 ms)							0.804 248 067
High speed Detection Handshake				TIMEOUT			0.814 260 750
SetAddress (20)	0 (20)	0		OK	FS	No data	0.855 686 233
GetDescriptor (Device)	20	0		OK	FS	18 bytes (12 01 00 02 EF 02 01 40 C9 ...	0.866 178 350
GetDescriptor (Configuration)	20	0		OK	FS	141 bytes (09 02 8D 00 04 01 00 C0 3...	0.867 469 833
GetDescriptor (String iSerialNumber)	20	0		OK	FS	34 bytes (22 03 30 00 31 00 32 00 33 ...	0.867 861 683
GetDescriptor (String lang IDs)	20	0		OK	FS	4 bytes (04 03 09 04)	0.868 088 983
GetDescriptor (String iProduct)	20	0		OK	FS	38 bytes (26 03 55 00 53 00 42 00 20 ...	0.868 266 150
GetDescriptor (Device qualifier)	20	0		STALLED	FS	No data	0.868 456 800
GetDescriptor (Device)	20	0		OK	FS	18 bytes (12 01 00 02 EF 02 01 40 C9 ...	5.874 713 600
GetDescriptor (Configuration)	20	0		OK	FS	9 bytes (09 02 8D 00 04 01 00 C0 32)	5.874 870 683
GetDescriptor (Configuration)	20	0		OK	FS	141 bytes (09 02 8D 00 04 01 00 C0 3...	5.875 035 650
SetConfiguration (1)	20	0		OK	FS	No data	5.876 545 833
GetDescriptor (String iProduct)	20	0		OK	FS	4 bytes (26 03 55 00)	5.878 198 817
GetDescriptor (String iProduct)	20	0		OK	FS	38 bytes (26 03 55 00 53 00 42 00 20 ...	5.878 381 250
GetDescriptor (String iProduct)	20	0		OK	FS	4 bytes (26 03 55 00)	5.880 315 900
GetDescriptor (String iProduct)	20	0		OK	FS	38 bytes (26 03 55 00 53 00 42 00 20 ...	5.880 501 283
GetLineCoding	20	0	0	OK	FS	7 bytes (00 C2 01 00 00 00 08)	5.883 746 867
SetControlLineState (Deactivate car...	20	0	0	OK	FS	No data	5.883 918 067
SetLineCoding	20	0	0	OK	FS	7 bytes (00 C2 01 00 00 00 08)	5.884 059 567
GetLineCoding	20	0	0	OK	FS	7 bytes (00 C2 01 00 00 00 08)	5.884 264 383
GetLineCoding	20	0	2	OK	FS	7 bytes (00 C2 01 00 00 00 08)	5.885 934 350
SetControlLineState (Deactivate car...	20	0	2	OK	FS	No data	5.886 111 817
SetLineCoding	20	0	2	OK	FS	7 bytes (00 C2 01 00 00 00 08)	5.886 248 583
GetLineCoding	20	0	2	OK	FS	7 bytes (00 C2 01 00 00 00 08)	5.886 450 933

Figure 16. USB enumeration process

The four virtual serial ports are implemented to return the data received to the USB host. Figure 17on page 19 shows the test results.


COM60:9600baud - Tera Ter...
File Edit Setup Control Window Help
COM61:9600baud - Tera Term ...
File Edit Setup Control Window Help
COM80:9600baud - Tera Ter...
File Edit Setup Control Window Help
COM81:9600baud - Tera Term ...
File Edit Setup Control Window Help

```
fasdffsad
```

```
ruereu
```

```
fsadfsadffasdf
```

```
fasdf
```

Figure 17. Test results of four VCOMs

7 Conclusion

This application note implements:

- the function of a USB device to two VCOM by using a USB composite class containing two CDC subclasses.
- the function of two USB devices to four VCOMs by using FS and HS USB device together.
- verifications on the LPC54018 EVK and LPC55S69 EVK board are feasible.

8 References

1. AN10420 USB virtual COM port on LPC214x.
2. USB 2.0 Specification
3. USB in MCU-Signal and Protocol.
4. Access USB Technology and Application based on Microcontrollers.

Date of release: June 2019
Document identifier: AN12458

arm