Semiconductors

AN10420

USB virtual COM port on LPC214x

Rev. 01 — 04 January 2006 Application note

Document information

Info	Content
Keywords	USB, virtual COM port, class driver, Microcontroller, MCU, USB device request, USB reset phase, USB enumeration phase.
Abstract	This document describes how to design a virtual COM port driver using LPC214x USB port.

PHILIPS

Revision history

Rev	Date	Description
01	20060104	Initial version.

Contact information

For additional information, please visit: http://www.semiconductors.philips.com

For sales office addresses, please send an email to: sales.addresses@www.semiconductors.philips.com

1. Introduction

This application note describes how to design a virtual COM driver using USB device on LPC214x. It provides complete sample software to configure LPC214x USB port as a virtual COM port and use it to communicate with the Microsoft Windows Hyper Terminal software.

The sample software is tested on the Keil's MCB214x board along with their ARM IDE uVision III.

The complete source code, both LPC214x USB device driver and USB host driver for the PC, can be found from "Technical Document (TechDoc(s))" section.

This application note is organized as below:

- General information on virtual COM port architecture.
- Basic USB concept and operation.
- Protocol and design consideration on virtual COM port.
- Virtual COM port protocol stack, driver, and key APIs.
- Virtual to physical COM port communication.
- File structure of the sample software.
- Sample software.

2. Virtual COM port

This section describes the basic concept of virtual COM port and its data flow model.

2.1 Virtual COM port architecture

Virtual COM port driver allows your PC to recognize and communicate with the remote target as a COM port regardless the under-layer hardware connection between the PC and target system. In this application note, the under-layer hardware communication is between the USB device on the LPC214x and the USB host on the PC. When the USB cable is connected, the target looks like a real serial port communicating with the PC Hyper Terminal Software on the Windows platform.

The data flow model is shown in Fig 1.

2.2 Key components in the virtual COM port

The key components in the virtual COM port implementation include:

- USB Host driver
- USB device driver on LPC214x
- Proprietary protocol stack for COM port configuration and communication
- A physical COM (UART) port driver. This driver is used to test the communication between a virtual COM port and a physical COM port.
- Application level test program to test the virtual COM port functionality on the USB device side.

- Windows' HyperTerminal software or any COM port terminal utility on the Microsoft Windows platform

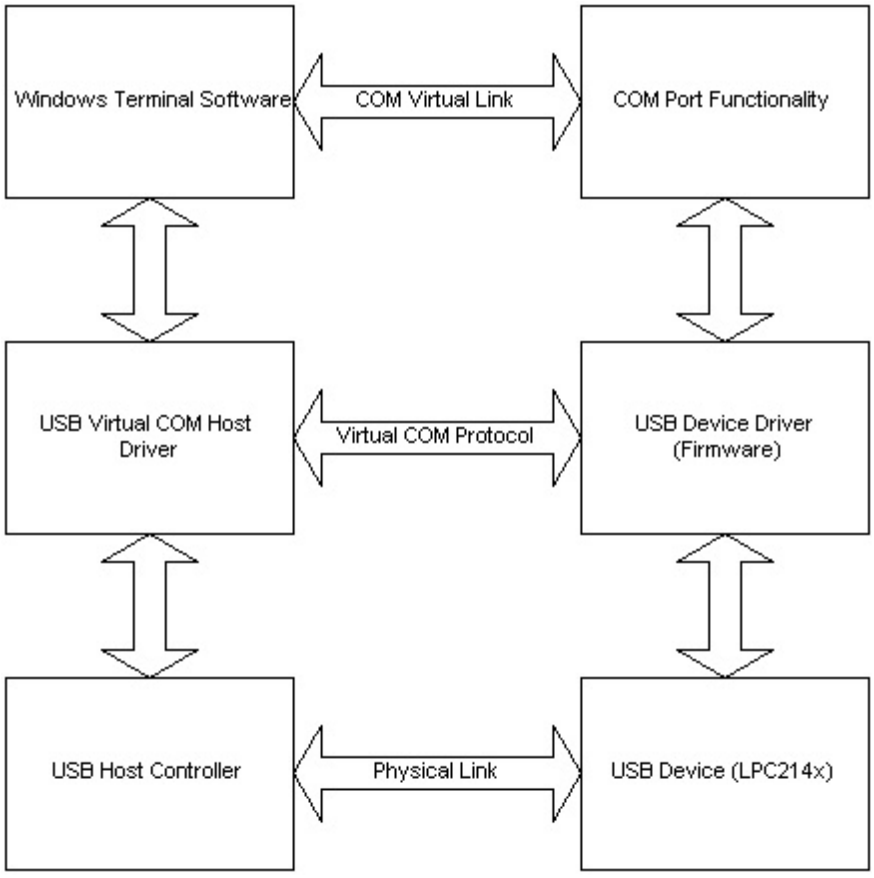

Fig 1. Virtual COM port data flow model

This application note will primarily focus on the implementation of the virtual COM device driver on the LPC214x USB device.

The source code of the virtual COM port host driver and the host driver installer will be provided as well for testing.

3. USB device driver concept and operation

Before describing virtual COM port USB protocol stack in details, some basic USB device driver concept and operation is introduced below.

3.1 LPC214x USB interface

The USB is a four-wire bus that supports data communication between the host and a large number of devices (max 128) simultaneously. LPC214x USB is a full-speed device controller compliant with USB 2.0 Specification. It supports up to 32 physical endpoints, and all four modes: Control, Bulk, Interrupt, and Isochronous. To maximize the USB throughput, on LPC2146/8, a DMA engine along with 8K internal DMA RAM supports DMA transfer support for all but control endpoints.

3.2 USB operation

The operation of the USB device driver can be simply defined in three phases: reset, enumeration, and finally, operation phase.

3.2.1 USB reset phase

The USB device will be in the reset phase after power-on reset. When the USB device is attached to the PC USB host, the host will issue a reset signal. When a USB reset signal is detected on the bus, on the device side, the DEV_STAT bit in the Device Interrupt Register is set and a USB interrupt will be generated. The USB device will process the RESET interrupt and set itself to the default configuration state. The initial address of the USB device is set to zero at reset phase.

After the reset signal is released and RESET interrupt has been processed, the device will enter the enumeration phase.

3.2.2 USB enumeration and standard requests

During the enumeration phase, the host performs a bus enumeration to identify the attached devices by sending a series of requests on the control pipe (endpoint 0 OUT) using **standard device request** to get the device information and configuration, and then, assign a unique address to it. Based on the information it gets, if necessary, send SET_FEATURE, SET_CONFIGURATION, and/or SET_INTERFACE requests to reconfigure the device. The device responds to the host requests on its default control pipe (endpoint 0 IN).

Basic standard requests from the host during the enumeration phase includes:

- GET_STATUS – the host sends this request to get the status for a specified recipient.
- GET_DESCRIPTOR - the host sends a get device descriptor request with the request type, based on the request type, the device replies with its attributes including Device Descriptor, Configuration Descriptor, vendor ID, product ID, etc.
- GET_CONFIGURATION - when the host sends this request to the device, the USB device responds with its configuration status. In the response data field, if the configuration value is zero, the device has not been configured. If a non-zero value is returned, the device has been configured. Please note that, this configuration value, is not the same as that in the configuration descriptor.
- GET_INTERFACE – once the host sends this request, the device replies with selected alternate setting for a specific interface defined in the interface descriptor. This request is not valid until the device has been configured.
- CLEAR_FEATURE – the host sends this request to clear or disable a specific feature. Recipient determines the selection of the feature. For more details about CLEAR_FEATURE and SET_FEATURE, look for Standard Feature Selector table in the USB 2.0 Specification.
- SET_FEATURE – the host send this request to set or enable a specific feature.
- SET_ADDRESS - a USB device uses the default address zero after reset until the host assigns a unique address using the SET_ADDRESS request. The device driver

gets the address from the host and set bit 7 to enable the embedded function of the USB engine.

- SET_DESCRIPTOR – the host sends this request to update an existing descriptor or add a new descriptor. This is an optional request.
- SET_CONFIGURATION - the host assigns the configuration value to the device. Based on the configuration information including interface and endpoint descriptors, the device driver will configure the device, then enable or disable the endpoints at the end.
- SET_INTERFACE – the host updates the interface descriptor information and associated endpoint descriptor information. If necessary, the device driver will configure, enable or disable the endpoints at the end.

For more details, the “Universal Serial Bus Specification 2.0”, Chapter 9, “USB Device Framework”, has all the information about the USB requests during the enumeration phase.

After the enumeration phase, the USB device is in the operation phase and ready for data communication with the USB host at any time.

All the USB standard requests related APIs are located in usbcore.c.

4. Virtual COM port protocol

This chapter describes the USB descriptor of the virtual COM port, the USB interface configuration, the end point configuration, and the protocol details of the virtual COM port device driver.

4.1 USB virtual COM descriptor

As mentioned in the previous chapter, USB host can configure devices at start-up or when the devices are plugged-in at run time. These devices are divided into various device classes. Each device class defines the common behavior and protocols for devices that serve similar functions. For example, USB mouse and keyboard devices all belong to the Human Interface Device (HID) class.

Regardless the device class type, each device class has one descriptor structure that is subdivided into following segments or sub descriptors. For the virtual COM port descriptor, it includes:

- Device Descriptor, describes the general information about the device.

```
/* USB Standard Device Descriptor */
const BYTE USB_DeviceDescriptor[] = {
  USB_DEVICE_DESC_SIZE,              /* bLength */
  USB_DEVICE_DESCRIPTOR_TYPE,        /* bDescriptorType */
  WBVAL(0x0100), /* 1.00 */          /* bcdUSB */
  0x00,                              /* bDeviceClass */
  0x00,                              /* bDeviceSubClass */
  0x00,                              /* bDeviceProtocol */
  USB_MAX_PACKET0,                   /* bMaxPacketSize0 */
  WBVAL(0xC251),                     /* idVendor */
  WBVAL(0x1305),                     /* idProduct */
  WBVAL(0x0110),                     /* bcdDevice */
```

```
  0x04,                                 /* iManufacturer */
  0x20,                                 /* iProduct */
  0x4A,                                 /* iSerialNumber */
  0x01                                  /* bNumConfigurations */
 };
```

- Configuration Descriptor, describes the configuration information about the device such as number of interfaces used for this application, the power source and its attributes, and the maximum power consumption for this device.
- Interface Descriptor, describes a specific interface within a configuration descriptor. The interface descriptor describes the number of endpoints used by this interface, the class and the subclass of the interface, the interface protocol, etc.
- Endpoint Descriptor, describes the information required by the host to determine the bandwidth requirements of each endpoint. It also describes the transfer type supported, the direction of the transfer, etc. The Endpoint Descriptor is always within the Interface Descriptor.

```
/* USB Configuration Descriptor */
/*   All Descriptors (Configuration, Interface, Endpoint ) */
const BYTE USB_ConfigDescriptor[] = {
/* Configuration 1 */
USB_CONFIGUARTION_DESC_SIZE,       /* bLength */
USB_CONFIGURATION_DESCRIPTOR_TYPE, /* bDescriptorType */
WBVAL(                             /* wTotalLength */
    USB_CONFIGUARTION_DESC_SIZE +
    USB_INTERFACE_DESC_SIZE     +
    NUM_ENDPOINTS * USB_ENDPOINT_DESC_SIZE +
    USB_INTERFACE_DESC_SIZE     +
    NUM_ENDPOINTS * USB_ENDPOINT_DESC_SIZE
),
0x02,                              /* bNumInterfaces */
0x01,                              /* bConfigurationValue */
0x00,                              /* iConfiguration */
USB_CONFIG_BUS_POWERED |           /* bmAttributes */
USB_CONFIG_REMOTE_WAKEUP,
USB_CONFIG_POWER_MA(100),          /* bMaxPower */
/* Interface 0, Alternate Setting 0, Class Code Unknown */
USB_INTERFACE_DESC_SIZE,           /* bLength */
USB_INTERFACE_DESCRIPTOR_TYPE,     /* bDescriptorType */
0x00,                              /* bInterfaceNumber */
0x00,                              /* bAlternateSetting */
NUM_ENDPOINTS,                     /* bNumEndpoints */
USB_DEVICE_CLASS_VENDOR_SPECIFIC,  /* bInterfaceClass */
0xFF,     /* bInterfaceSubClass, USB_SUBCLASS_CODE_UNKNOWN */
0xFF,     /* bInterfaceProtocol, USB_PROTOCOL_CODE_UNKNOWN */
0x00,     /* iInterface, STR_INDEX_INTERFACE = no_string */
/* Endpoint, EP1 Interrupt In */
USB_ENDPOINT_DESC_SIZE,            /* bLength */
USB_ENDPOINT_DESCRIPTOR_TYPE,      /* bDescriptorType */
USB_ENDPOINT_IN(1),                /* bEndpointAddress */
USB_ENDPOINT_TYPE_INTERRUPT,       /* bmAttributes */
WBVAL(0x0004),                     /* wMaxPacketSize */
0x20,          /* 32ms */          /* bInterval */
/* Endpoint, EP2 Bulk Out */
USB_ENDPOINT_DESC_SIZE,            /* bLength */
```

```
  USB_ENDPOINT_DESCRIPTOR_TYPE,       /* bDescriptorType */
  USB_ENDPOINT_OUT(2),                /* bEndpointAddress */
  USB_ENDPOINT_TYPE_BULK,             /* bmAttributes */
  WBVAL(0x0040),                      /* wMaxPacketSize */
  0x20,                              /* 32ms */
  /* Endpoint, EP2 Bulk In */
  USB_ENDPOINT_DESC_SIZE,             /* bLength */
  USB_ENDPOINT_DESCRIPTOR_TYPE,       /* bDescriptorType */
  USB_ENDPOINT_IN(2),                 /* bEndpointAddress */
  USB_ENDPOINT_TYPE_BULK,             /* bmAttributes */
  WBVAL(0x0040),                      /* wMaxPacketSize */
  0x20,                              /* 32ms */
  /* Interface 1, Alternate Setting 0, Class Code Unknown */
  USB_INTERFACE_DESC_SIZE,            /* bLength */
  USB_INTERFACE_DESCRIPTOR_TYPE,      /* bDescriptorType */
  0x01,                               /* bInterfaceNumber */
  0x00,                               /* bAlternateSetting */
  NUM_ENDPOINTS,                      /* bNumEndpoints */
  USB_DEVICE_CLASS_VENDOR_SPECIFIC,   /* bInterfaceClass */
  0xFF,   /* bInterfaceSubClass, USB_SUBCLASS_CODE_UNKNOWN */
  0xFF,   /* bInterfaceProtocol, USB_PROTOCOL_CODE_UNKNOWN */
  0x00,   /* iInterface, STR_INDEX_INTERFACE = no_string */
  /* Endpoint, EP4 Interrupt In */
  USB_ENDPOINT_DESC_SIZE,             /* bLength */
  USB_ENDPOINT_DESCRIPTOR_TYPE,       /* bDescriptorType */
  USB_ENDPOINT_IN(4),                 /* bEndpointAddress */
  USB_ENDPOINT_TYPE_INTERRUPT,        /* bmAttributes */
  WBVAL(0x0004),                      /* wMaxPacketSize */
  0x20,          /* 32ms */           /* bInterval */
  /* Endpoint, EP5 Bulk Out */
  USB_ENDPOINT_DESC_SIZE,             /* bLength */
  USB_ENDPOINT_DESCRIPTOR_TYPE,       /* bDescriptorType */
  USB_ENDPOINT_OUT(5),                /* bEndpointAddress */
  USB_ENDPOINT_TYPE_BULK,             /* bmAttributes */
  WBVAL(0x0040),                      /* wMaxPacketSize */
  0x20,                              /* 32ms */
  /* Endpoint, EP5 Bulk In */
  USB_ENDPOINT_DESC_SIZE,             /* bLength */
  USB_ENDPOINT_DESCRIPTOR_TYPE,       /* bDescriptorType */
  USB_ENDPOINT_IN(5),                 /* bEndpointAddress */
  USB_ENDPOINT_TYPE_BULK,             /* bmAttributes */
  WBVAL(0x0040),                      /* wMaxPacketSize */
  0x20,          /* 32ms */
  /* Terminator */
  0                                   /* bLength */
};
```

- String Descriptor, includes information such as vendor name, product information, etc.

All the descriptors have been defined in “usbdesc.c” file. For more details about USB descriptor, please refer to “Chapter 9.6, Standard USB Descriptor Definitions, USB Specification 2.0”.

The following chapter will describe how the interfaces and endpoints are being set to support virtual COM port communication.

4.2 Endpoint configuration for virtual COM port

In the virtual COM port device driver implementation, two interface descriptors have been created to accommodate two virtual COM ports. Under each interface descriptor, **vendor specific** class code (0xFF) has been chosen. Three endpoints have been used for each interface. Here is the endpoint configuration table:

Table 1: Interface and Endpoint Setting For virtual COM Port

Interface Number	Endpoint Number (physical EP#, type)	Description
0	EP1 IN (3, interrupt)	Report modem status of the device in UART0 or virtual COM port 0
0	EP2 OUT (4, bulk)	Data transfer from host to the device on UART0 or virtual COM port 0
0	EP2 IN (5, bulk)	Data transfer from device to the host on UART0 or virtual COM port 0
1	EP4 IN (9, interrupt)	Report modem status of the device in UART1 or virtual COM port 1
1	EP5 OUT (10, bulk)	Data transfer from host to the device on UART1 or virtual COM port 1
1	EP5 IN (11, bulk)	Data transfer from device to the host on UART1 or virtual COM port 1

4.3 Virtual COM port protocol

In order to establish the COM port communication, after the USB Reset phase and entering the enumeration phase, the USB host is responsible for sending the COM port information such as COM port baud rate, data bits, stop bits, hardware handshaking to the USB device. To do so, USB host uses the **vendor specific request** to set up the virtual COM port on the USB device. For more details about USB Device Request and the format of the setup data, please refer to “Chapter 9.3 USB Device Request: in “USB Specification 2.0”

Based on the **vendor specific request** from the host, the USB device will configure the on-chip COM port accordingly. Once the configuration is accomplished, the data from the USB host can be transmitted or received to/from the on-chip COM port seamlessly via the virtual COM port.

Below table is a proprietary protocol between the USB host and device for the virtual COM port configuration.

Using the USB device request type **vendor specific request**, the host is responsible to establish the request values sent to the device in the setup packet. The length of every USB setup packet is 8 bytes. The 8 bytes contain the type of the request, the request, the setup value, the index, and the length as shown in the following Table:

Table 2: Vendor specific interface table

0	1	2	3	4	5	6	7
RequestType	Request	Value		Index		Length	

The first byte of the setup packet represents “Request Type”. The Request Type identifies the characteristics of the request shown as follows:

Table 3: Request Type Configuration Request Table

Bit	7	6	5	4	3	2	1	0
	Direction	Type		Recipient				
	0=Host to Device 1=Device to Host	0=Standard 1=Class 2=Vendor 3=Reserved		0=Device, 1=Interface, 2=Endpoint, 3=Other 4..31=Reserved				
0x41	0	1	0	0	0	0	0	1
	Host to Device direction	Vendor type		Interface recipient				

In the second byte, “Request” field specifies the particular request for configuring the UART device followed with a value shown as follows:

Table 4: Virtual COM Configuration Table

Request (bit 4~0 only)		Data Value (2 bytes)		
		MSB	LSB	
0x00	-			
0x01	-			
0x02	-			
0x03	Set Baud Rate	0x00	0x0C	9600 Baud Rate
			0x06	19.2K
			0x03	38.4K
			0x02	57.6K
			0x01	115.2K
0x04	Set Stop Bit	0x00	0x00	1 stop bit
			0x01	2 stop bits
0x05	Set Data Bit	0x00	0x00	5 bits data length
			0x01	6 bits
			0x02	7 bits
			0x03	8 bits
0x06	Set Parity	0x00	0x00	Odd parity
			0x01	Even parity
0x07	Set Flow Control			N/A
0x08	Set DTR	0x00	0x00	DTR clear (LPC2148 UART1 only)
			0x01	DTR set (LPC2148 UART1 only)
0x09	Set RTS	0x00	0x00	RTS clear (LPC2148 UART1 only)
			0x01	RTS set (LPC2148 UART1 only)

0x0A	-			
0x0B	Burst Transmit EP0			
0x0C	Modem Status			

Bit 7, 6, and 5 of the “Request” is used to represent the channel number of UART as shown on the following table.

Table 5: Virtual COM channel selection table

Bit 7	Bit 6	Bit 5	Request byte	Channel number
0	0	0	0x0	1 (UART0 on LPC214x)
0	0	1	0x2	2 (UART1 on LPC214x)
0	1	0	0x4	3 (N/A)
0	1	1	0x6	4 (N/A)
1	0	0	0x8	5 (N/A)
1	0	1	0xA	6 (N/A)
1	1	0	0xC	7 (N/A)
1	1	1	0xE	8 (N/A)

Here is an example of a Vendor Interface device request:

Table 6: Vendor interface device request example

Byte	0	1	2	3	4	5	6	7
	RequestType	Request	Value		Index		Length	
Setup *	0x41	0x03	0x00	0x0C	00	00	00	00

As seen in the last column of above table, 0x41 indicates the direction of the setup request is from host to device (bit 7 is 0), the type is “vendor” (bit 6 and 5 is 10b), and the recipient is “interface” (bit 4 through 0 is 00001b). Finally, the SETUP request is to set the baud rate of the COM port 0 at 9600.

The index and length fields in the vendor specific interface request table are defined but not used. They are reserved for future expansion.

5. USB device driver and APIs

This chapter describes the initialization sequence and some lower level APIs for USB device configuration and communication. The central piece of the USB driver that deals with the SETUP packets from the host at endpoint 0 is mentioned below as well.

5.1 LPC214x USB initialization

After the power up, the USB initialization should include below steps:

- Turn on USB PCLK
- Configure 48Mhz PLL1 for USB clock
- Setup Vectored Interrupt Controller (VIC) for USB
- Set up minimum numbers of USB registers including index and packet size register for Control OUT (0) and Control IN (1) endpoints.
- Set USB Device Interrupt Enable register

- Use protocol engine commands SET_ADDRESS to reset device address to zero, and SET_DEVICE_STATUS to make a soft connection.

5.2 Key APIs for lower-level USB driver

The key APIs for the lower-level USB driver in the attached sample program include:

- void WrCmd (DWORD Cmd) – the API to write a command to the protocol command engine. The list of the commands can be found in Protocol Engine Command Code Table, Chapter 14, USB Device Controller, LPC214x User's Manual.
- void WrCmdDat (DWORD Cmd, DWORD Dat) – the API to write both command and data to the protocol command engine.
- DWORD RdCmdDat (DWORD Cmd) – the API to write the command to and then read the data from the protocol command engine. The return value of this API is the value read from the protocol command engine.
- void USB_Init (void) – this routine initializes all the supporting modules for the USB, for more details, see previous section, "LPC214x USB Initialization".
- void USB_Reset (void) – this routine resets the value of index register and maximum packet size registers for CONTROL endpoints 0 and 1, clear all the interrupts for all the endpoints.
- DWORD USB_ReadEP (BYTE EPNumber, BYTE *DataPtr) – this module is a key API used to read data from an endpoint. The first parameter is the endpoint number where the data will be read from, the second parameter is the pointer pointing to the data area just read from. The return value is the number of bytes read from this endpoint.
- DWORD USB_WriteEP (BYTE EPNumber, BYTE DataPtr, DWORD Length) – this module is a key API used to write data to an endpoint. The first parameter is the endpoint number where the data will be written, the second parameter is the pointer pointing to the data area to be written to, the third parameter is the length of the data block. The return value is the number of bytes written to this endpoint.
- void USB_EnableEP (BYTE EPNumber) – Enable a particular endpoint. The parameter is the endpoint number.
- viod USB_DisableEP (BYTE EPNumber) – Disable a particular endpoint. The parameter is the endpoint number.
- void USB_ISR(void) – The most important module in the driver, the USB interrupt handler. It handles interrupts such as DEVI_STAT, FRAME interrupt for isochronous mode, and EP_SLOW data transfer. Please note, only EP_SLOW interrupt transfer is used for the sample software, EP_FAST interrupt transfer is disabled. The interrupt handler is responsible for all the device interrupts whenever the state has changed in the USB Device Interrupt register. The flow chart of the interrupt handler is in the following figure.
- USB_EndPointx(DWORD Event) - This is the event callback routine for each endpoint, "x" is the logical endpoint number. The defined endpoint "Event"s in "vcomuser.c" include,
  - USB_EVT_SETUP
  - USB_EVT_OUT
  - USB_EVT_IN
  - USB_EVT_OUT_NAK

- USB_EVT_IN_NAK
- USB_EVT_OUT_STALL
- USB_EVT_IN_STALL

5.3 Setup packets at endpoint 0

The data transfer on each endpoint is determined when an EP_SLOW interrupt is generated. Once the EP_SLOW interrupt occurs, the interrupt handler checks the status of the endpoint interrupt status register. If the interrupts occur on the control endpoint, EP0 OUT indicating a USB device request from the host has arrived, the interrupt handler then will send SELECT_ENDPOINT command to the command protocol engine using WrCmd() followed by a RdCmdDat() to get the type of the USB packet, once the “STP” bit is set in Select Endpoint Register indicating a SETUP packet has arrived. When this happens, the USB_EndPoint0() in usbcore.c event callback will be invoked. This is the central piece of the USB driver to deal with the setup requests from the host in the enumeration phase.

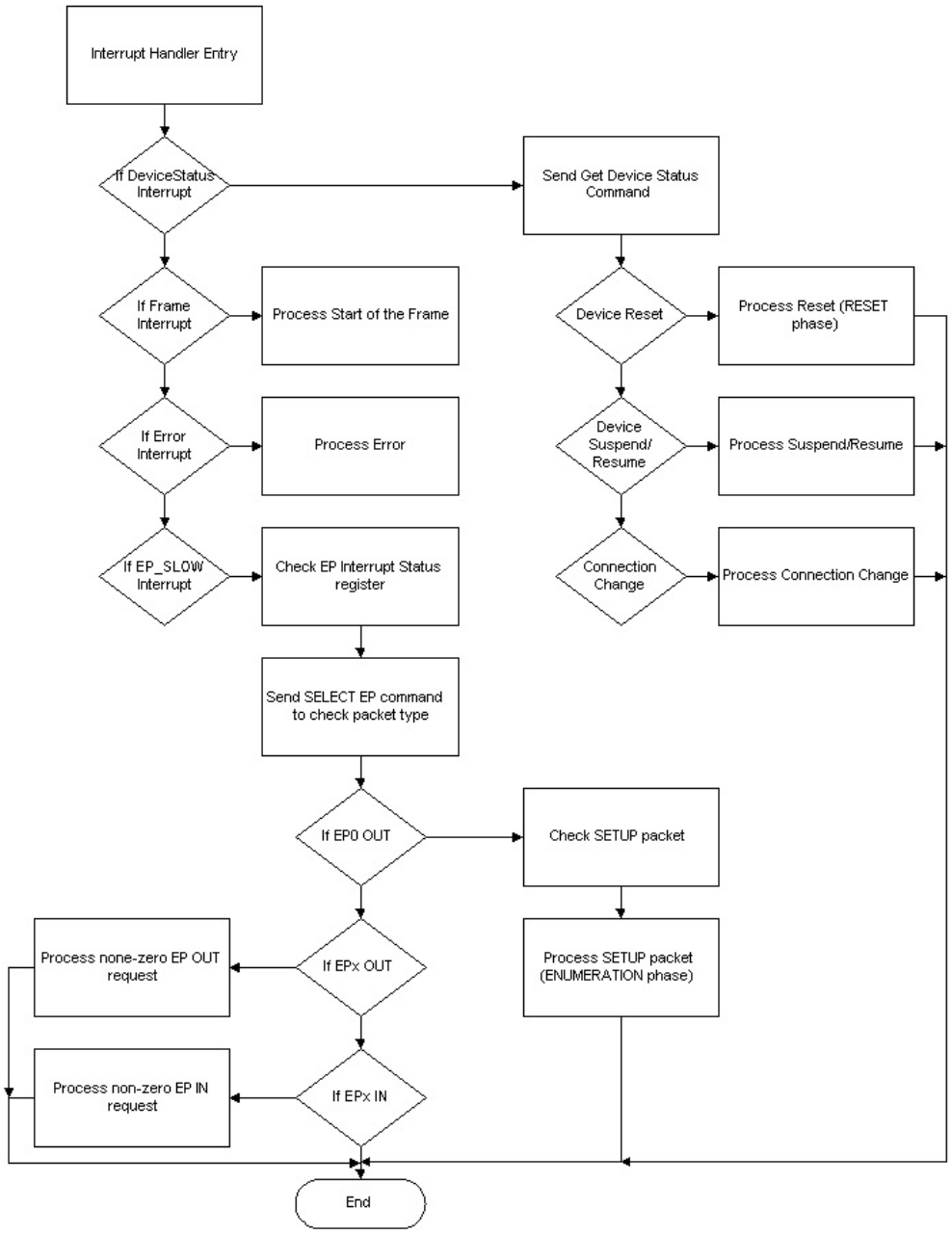

Fig 2. USB interrupt handler flowchart

6. Virtual COM port key APIs and file structure

This section will cover some functional level APIs, describe implementation details of the virtual COM port sample software, and a test sample to move data between a virtual COM port and a physical COM port.

6.1 Virtual COM port key APIs

After the standard requests from the host in USB enumeration stage, the host will finally send vendor specific request to configure the virtual COM port. The key module to process this vendor specific request is,

- BYTE VCOM_SetSIOSetup (BYTE Cmd, BYTE Data) - Based on the request table and channel table in Table 6.3 and Table 6.4, once the USB device receives the request from the host, it will act as a command processor, interpret the command byte, use “Cmd” byte bit 5,6,7 to determine the UART channel (see Table 6.4), and then, and “Cmd” byte bit 0 through 4 to configure the UART port baud rate, data bit, stop bits, hardware handshaking signal, etc. (See Table 6.3) “Cmd” byte contains both channel information and request in Table 6.3, while “Data” byte contains the LSB configuration data value in column 4 of Table 6.3. MSB byte is always zero, thus, will be ignored. The return value indicates the status of the UART configuration.

  Many UART API support modules are only used to support above setup module such as:

  - Void SetSIOBaudrate (BYTE channel, BYTE ConfigValue)
  - Void SetSIOStopBit (BYTE channel, BYTE ConfigValue)
  - Void SetSIODataBit (BYTE channel, BYTE ConfigValue)
  - Void SetSIOParity (BYTE channel, BYTE ConfigValue)

- Void init_serial (void) – This initialization routine is used to establish two physical COM ports, UART0 and UART1, on LPC214x. The physical COM ports are used to test the communication between the virtual COM ports and the physical COM ports.

- void DeviceData2Host (BYTE PortNum) - the module which transmit data from USB device to the host. Based on the UART port number, it gets data from the UART buffer or physical COM port, and then call USB_WriteEP() to send data to the host virtual terminal.

- void DeviceData2UART (BYTE PortNum) - the module which receives data from the USB host. It calls USB_ReadEP() to get data from the USB endpoint and dump it to the UART buffer or physical COM port based on the port number.

6.2 Virtual to physical COM port communication

The main routine of the software to test the virtual COM port driver is, use hyper-terminal to send data to the LPC214x via USB virtual COM port, whenever the LPC214x receives the data, it uses physical COM port to forward data to the hyper-terminal via UART 0 or UART1 on the LPC214x, vice versa. See below diagram:

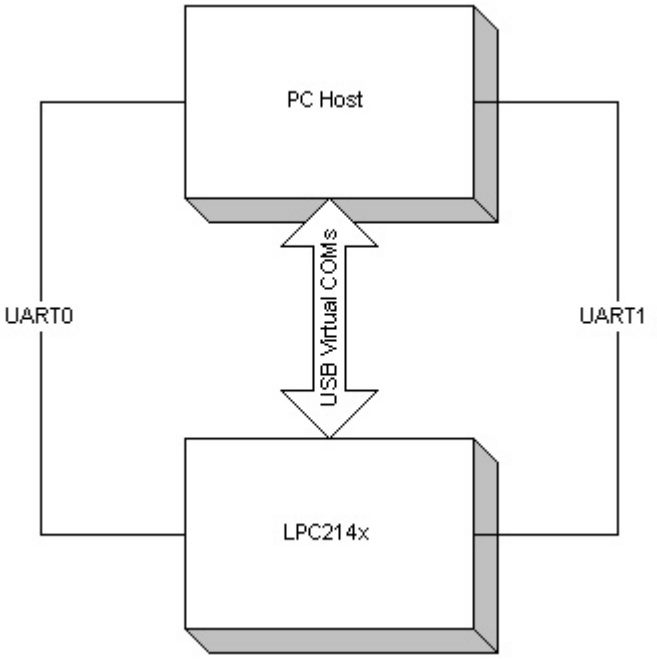

Fig 3. Virtual COM and physical COM

The USB host driver will create two USB virtual COM ports on LPC214x, COMx and COM(x+1). “x” is not a fixed number, as the host driver will check the status of the PC COM port configuration first, and then assign the number to each virtual COM port. Virtual COMx is used to communicate with physical UART0 while virtual COM(x+1) is used to communicate with physical UART1.

Two modules mentioned above, DevceData2UART() and DeviceData2Host(), are primarily used for virtual COM to physical COM port communication. However, please note that, if the UART cable is not available, the module could take data from the USB host terminal, save into the UART buffer, and simply light up some LEDs to see the result, not necessarily forward the data to another physical COM port.

7. File structure of the virtual COM port sample

To facilitate the understanding of the sample software, the file structure of the virtual COM port source is provided below:

Abstract.txt	- Keil uVision III abstract file. It describes how to use two different methods, with UART cable or without UART cable, to test the virtual COM port driver.
Demo.c	- main entry of the sample software
Demo.h	- definitions and include file for the main entry
Startup.s	- boot-up file
Type.h	- various type definition
Usb.h	- USB descriptor data structures and class definitions
Usbcfg.h	- USB configuration header file
Usbcore.c	- USB standard class request, USB command processor
Usbcore.h	- definition and include file for usbcore.c
Usbdesc.c	- USB descriptor configuration
Usbdesc.h	- definition and include file for usbdesc.c
Usbhw.c	- USB hardware initialization and lower-level driver APIs
Usbhw.h	- prototyping for usbhw.c
Usbreg.h	- hardware USB block register definitions for LPC214x
Vcomuser.c	- Virtual COM port configuration and physical COM port APIs
Vcomuser.h	- endpoint event definition and include files for vcomuser.c
Virtualcom.prj	- Keil uVision III project file

8. Disclaimers

Life support — These products are not designed for use in life support appliances, devices, or systems where malfunction of these products can reasonably be expected to result in personal injury. Philips Semiconductors customers using or selling these products for use in such applications do so at their own risk and agree to fully indemnify Philips Semiconductors for any damages resulting from such application.

Right to make changes — Philips Semiconductors reserves the right to make changes in the products - including circuits, standard cells, and/or software - described or contained herein in order to improve design and/or performance. When the product is in full production (status 'Production'), relevant changes will be communicated via a Customer Product/Process Change Notification (CPCN). Philips Semiconductors assumes no responsibility or liability for the use of any of these products, conveys no licence or title under any patent, copyright, or mask work right to these products, and makes no representations or warranties that these products are free from patent, copyright, or mask work right infringement, unless otherwise specified.

Application information — Applications that are described herein for any of these products are for illustrative purposes only. Philips Semiconductors make no representation or warranty that such applications will be suitable for the specified use without further testing or modification.

9. Trademarks

Notice — All referenced brands, product names, service names and trademarks are the property of the respective owners.

10. Contents

Date of release:04 January 2006
Document number: AN10420_1

Published in The Netherlands