



# smxUSB<sup>TM</sup> User's Guide

USB Device Stack

Version 2.60  
February 23, 2024

by Yingbo Hu

<b>1. Overview.....</b>	<b>1</b>
<b>2. Files.....</b>	<b>2</b>
2.1 Directory Structure.....	2
2.2 Files.....	2
<b>3. smxUSB<sup>TM</sup> Code and Demos .....</b>	<b>5</b>
3.1 smxUSB <sup>TM</sup> Configuration .....	5
3.2 Building the Code .....	14
3.3 Building and Running the Demos .....	14
3.4 Initialization .....	16
<b>4. Function Drivers.....</b>	<b>17</b>
4.1 Audio .....	17
4.2 Device Firmware Upgrade (DFU).....	23
4.3 HID Communication.....	26
4.4 Keyboard.....	28
4.5 Mass Storage.....	29
4.6 Media Transfer Protocol (MTP) .....	30
4.7 Mouse .....	34
4.8 Ethernet over USB .....	35
4.9 Serial (CDC-ACM).....	38
4.10 Video .....	42
<b>5. Writing a New Function Driver.....</b>	<b>46</b>
5.1 Function Driver Interface.....	46
5.2 Function Operation Interface .....	47
5.3 Device Information .....	48
5.4 Configuration Information .....	49
5.5 Send Request.....	51
5.6 Select Endpoint Number .....	51
<b>6. Writing a New Device Controller Driver.....</b>	<b>52</b>
6.1 Device Controller Operation Interface.....	52
6.2 Handle Device Controller Interrupt .....	55
6.3 Logical Endpoint Number and Physical Endpoint Number .....	56

<b>7. Composite Device</b> .....	<b>58</b>
7.1 Composite Device Framework.....	58
7.2 Adding an Existing Function to the Composite Device Framework .....	58
7.3 Composite Device Product and Interface IDs .....	59
7.4 Composite Device Limitations .....	60
<b>8. Hardware Porting Notes</b> .....	<b>61</b>
8.1 udport.h.....	61
8.2 udport.c .....	61
8.3 DMA Transfer.....	63
<b>9. Windows Drivers / Application</b> .....	<b>64</b>
9.1 Multiple Port Serial Device (or Single Port Limited Endpoints) .....	65
9.2 Device Firmware Upgrade (DFU) Device .....	66
9.3 HID Communication.....	69
<b>10. Application Notes</b> .....	<b>71</b>
10.1 Flow Control of the Serial Port .....	71
10.2 Mass Storage and File System Share the Same Media .....	71
10.3 Switching to Different Functions at Run Time.....	72
10.4 Mass Storage Function Driver Buffer Size .....	73
10.5 Improving USB to Serial Function Driver Performance .....	74
10.6 Linux Support .....	75
10.7 MAC OS X Support.....	75
10.8 USB Device Controller Soft Connect Feature .....	75
10.9 Opening a Serial Port on the Host.....	75
10.10 Multiple Same Type Devices on the Same USB Host .....	75
10.11 HID vs. Serial for Data Communication .....	76
10.12 HID Communication Multiple Reports.....	76
10.13 Video Camera Software .....	77
<b>Appendix A. Memory Usage and Performance Summary</b> .....	<b>78</b>
A.1 Size.....	78
A.2 Performance .....	81
<b>Appendix B. Block Device Driver Interface</b> .....	<b>83</b>
<b>Appendix C. Installing Devices under Windows 2000</b> .....	<b>84</b>
C.1 Audio.....	84
C.2 Mass Storage .....	86
C.3 Mouse/Keyboard.....	86
C.4 Ethernet over USB .....	89
C.5 Serial Port.....	96
<b>Appendix D. Installing Devices under Windows XP</b> .....	<b>101</b>
D.1 Audio .....	101
D.2 Device Firmware Upgrade (DFU).....	103
D.3 Mass Storage.....	114
D.4 Media Transfer Protocol (MTP) .....	114
D.5 Mouse/Keyboard.....	119
D.6 Ethernet over USB .....	119
D.7 Serial Port .....	121
D.8 Video.....	127

<b>Appendix E. Installing Devices under Windows Vista, 7, and 8</b> .....	<b>131</b>
E.1 Audio .....	131
E.2 Device Firmware Upgrade (DFU) .....	134
E.3 Mass Storage .....	140
E.4 Media Transfer Protocol (MTP).....	140
E.5 Mouse/Keyboard .....	143
E.6 Ethernet over USB.....	143
E.7 Serial Port.....	143
E.8 Video .....	154
<b>Appendix F. Specification Reference</b> .....	<b>156</b>
F.1 USB Specifications .....	156
F.2 Device Controller Specifications .....	156
F.3 Audio Devices Specifications .....	156
F.4 Communication Devices Specifications.....	156
F.5 Device Firmware Upgrade (DFU) Specifications .....	156
F.6 HID Specifications.....	156
F.7 Mass Storage Specifications .....	157
F.8 Media Transfer Protocol (MTP) Specifications.....	157
F.9 Remote NDIS Specifications .....	157
F.10 Video Device Specifications.....	157
<b>Appendix G. Testing</b> .....	<b>158</b>
<b>Appendix H. Host OS Certification</b> .....	<b>159</b>
H.1 Windows Logo Program / Windows Hardware Certification Program.....	159
<b>Appendix I. Glossary</b> .....	<b>161</b>

© Copyright 2005-2024

Micro Digital Associates, Inc.  
(714) 437-7333  
support@smxrtos.com  
www.smxrtos.com

All rights reserved.

smxUSBD is a Trademark of Micro Digital, Inc.  
smx is a Registered Trademark of Micro Digital, Inc.

# 1. Overview

smxUSBD™ is a full-featured USB device stack for the SMX® RTOS. It offers a clean, modular design that allows embedded system developers to easily add USB device capabilities to their projects. Normally this is done to permit connection to a PC in order to upload or download data, tables, code, etc.

For easy connectivity to a PC, smxUSBD offers these function drivers: **audio, keyboard, mouse, mass storage, MTP/PTP, Ethernet over USB, serial, and video**. Each is compatible with the corresponding Windows USB class driver. Thus, a device using the above smxUSBD function drivers does not require a custom Windows driver in order to connect to a PC. Some other smxUSBD function drivers, such as DFU, do need a Windows driver to make it work on Windows. For details, see chapter 9. Windows Drivers / Application. You need to decide which device connection is most appropriate for your device and to use the corresponding API for that device. See chapter 4. Function Drivers.

smxUSBD function drivers are also supported by Linux and MAC OS X. You don't need write your own driver for them.

The reader should be familiar with the USB 2.0 specification. All USB specifications can be found at [www.usb.org/](http://www.usb.org/).

smxUSBD has four layers:

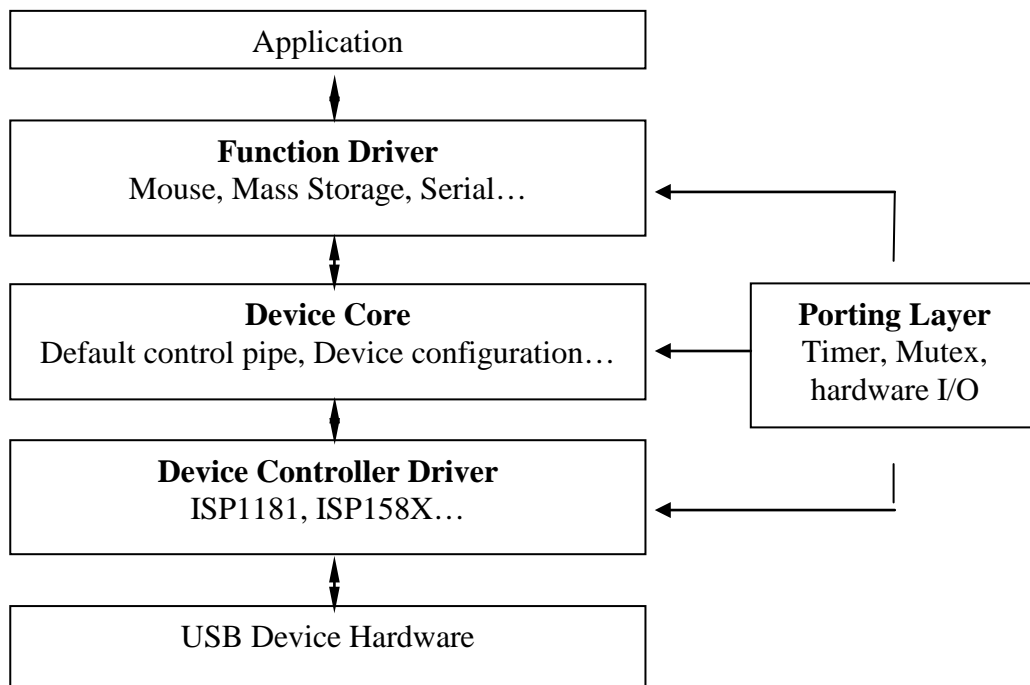


Figure 1 smxUSBD Structure

1. **Function Driver Layer:** Provides different USB functions to the high level application, such as serial emulator, mouse, and mass storage.
2. **Device Core Layer:** Provides the common USB Device framework. See chapter 9 in the USB 2.0 specification for details.
3. **Device Controller Driver (DCD) Layer:** Provides a unique interface for different USB device controllers such as ISP1181 (ISP1161, ISP1362) or ISP158x (ISP1761).
4. **Porting Layer:** Provides functions related to the hardware and compiler.

smxUSBDB supports processors that can only do 16-bit memory addressing, such as some Texas Instruments DSPs. See discussion of SB\_CPU\_MEM\_ADDR\_8BIT and SB\_PACKED\_STRUCT\_SUPPORT in the smxBase User's Guide. These settings have been tested for the core code, serial function driver, and ISP1161/81/1362 driver. Other function drivers and device controller drivers have not been tested, and may require modification.

## 2. Files

Like other SMX<sup>®</sup> RTOS products, all source code for smxUSBDB is stored in its own directory, named "XUSBDB", under the main SMX directory. Below is a summary of the directory structure.

### 2.1 Directory Structure

SMX

APP

DEMO usbddemo.c (for SMX)

NORTOS Build directory for standalone (non-SMX) releases. Has demo too.

XUSBDB Configuration and porting layer files

XX.YY Build directory for SMX releases

Core Device code layer support

DCD All device controller drivers

Function All function drivers such as serial, mouse, and mass storage

## 2.2 Files

### 2.2.1 Main Files

FILE	DESCRIPTION
smxusbdb.h	smxUSBDB API header file. Use in application files.
udcfg.h	smxUSBDB configuration file. Allows enabling/disabling main components of smxUSBDB.
udinit.c,h	Initialization of the USB device stack, including the selected function driver and selected device controller.
udintern.h	Main internal header file. Included by smxUSBDB files

	rather than including individual header files, in order to ensure files are included in the proper order.
--	---

## 2.2.2 Porting Layer

FILE	DESCRIPTION
udport.c,h	Porting files.

## 2.2.3 Device Core

FILE	DESCRIPTION
udutil.h	Common utility inline functions.
uddesc.h	USB descriptor definition.
uddcd.c,h	Device controller driver interface. Provides a unique interface between the device controller driver and device stack core layer.
uddevice.c,h	The core of the USB device stack
udfunc.c,h	Function driver interface. Provides a method to let the function driver register with the device core stack
udep0.c,h	Handles the default control pipe (endpoint 0) request.

## 2.2.4 Function Driver

FILE	DESCRIPTION
udaudio.c,h	USB Audio/MIDI function driver.
udcompos.c,h	USB Composite Device framework driver.
uddfu.c,h	USB Device Firmware Upgrade function driver.
udftempl.c,h	USB Function Driver template.
udhidcom.c,h	USB HID Communication function driver.
udkbd.c,h	USB Keyboard function driver.
udmouse.c,h	USB Mouse function driver.
udmstor.c,h	USB Mass Storage driver.
udmtp.c,h	USB MTP function driver.
udnet.c,h	USB Ethernet over USB driver.
udserial.c,h	USB Serial Emulator function driver.
udvideo.c,h	USB Video function driver.
*.inf	Installation files to install devices on Windows. See: Appendix C. Installing Devices under Windows 2000, Appendix D. Installing Devices under Windows XP, Appendix E. Installing Devices under Windows Vista, 7

## 2.2.5 Device Controller Driver

FILE	DESCRIPTION
ud1181.c,h	NXP ISP1181-compatible device controller driver. Also supports ISP1161 and ISP1362.
ud158x.c,h	NXP ISP1582/3-compatible device controller driver. Also supports ISP1761.
ud2272.c,h	PLX Net2272 USB device controller driver.
ud720150.c,h	NEC uPD720150 device controller driver.
upd720150.c,h	NEC uPD720150 low-level register access routines. May need to be built into the BSP code instead of the USB code.
udat91.c,h	Atmel AT91SAM7S/7X/7A3, AT91SAM9260/61/63, AT91SAM9XE512, AT91RM9200 USB device controller driver.
udat91hs.c,h	Atmel AT91SAM9G45/M10/RL64, SAM3U4, CAP9 USB high speed device controller driver.
udat91o.c,h	Atmel AT91SAM3X USB high speed device controller driver.
udblkfn.c,h	Analog Devices Blackfin USB high speed device controller driver.
udcf522x.c,h	Freescale CF5225x/1x/2x USB device controller driver.
udcf5329.c,h	Freescale CF532x/7x USB device controller driver.
udcf548x.c,h	Freescale CF548x/7x USB device controller driver.
udip3511.c,h	NPX LPC55SXX USB device controller driver.
udlh7a4.c,h	Sharp LH7A400/4 USB device controller driver.
udlm3s.c,h	Luminary Micro LM3Sxxxx USB device controller driver.
udlpc.c,h	NXP LPCxxxx USB device controller driver.
udmx1.c,h	Freescale i.MX1/MXL USB device controller driver.
udstr7.c,h	ST STR710, 711, 720, STR912 USB device controller driver.
udsyndwc.c,h	Synopsys DesignWare IP driver.
udctempl.c,h	Device Controller Driver template.

## **3. smxUSB Code and Demos**

This section documents details of configuration and building the code and demos.

### **3.1 smxUSB Configuration**

#### **3.1.1 udcfg.h**

smxUSB can be configured so that it includes support for specific USB devices, thus saving code space. The following sections describe the settings.

#### **Operating System Selection**

Operating System is selected in bcfg.h in smxBase. Please see the smxBase User's Guide for detailed information.

#### **Controller Selection**

##### **SUD\_AT91**

Set to "1" to include support for an Atmel AT91 USB full speed device controller such as AT91SAM7S/7X, AT91SAM9, and AT91RM9200 device controllers. Set to "0" to exclude support.

##### **SUD\_AT91HS**

Set to "1" to include support for an Atmel AT91 USB high speed device controller such as AT91SAM9RL64 device controllers. Set to "0" to exclude support.

##### **SUD\_AT91OTG**

Set to "1" to include support for an Atmel AT91 SAM3X USB high speed device controller. Set to "0" to exclude support.

##### **SUD\_BLACKFIN**

Set to "1" to include support for an Analog Devices Blackfin USB high speed device controller such as BF527 device controller. Set to "0" to exclude support.

##### **SUD\_CF522XX**

Set to "1" to include support for a Freescale CF5225x/CF5221x/CF5222x USB device controller. Set to "0" to exclude support.

##### **SUD\_CF5227X**

Set to "1" to include support for a Freescale CF5227x USB device controller. Set to "0" to exclude support. Uses the CF5329 driver.



**SUD\_CF5251**

Set to “1” to include support for a Freescale CF5251 USB device controller. Set to “0” to exclude support.

**SUD\_CF5272**

Set to “1” to include support for a Freescale CF5272 USB device controller. Set to “0” to exclude support. Not done; only mouse works.

**SUD\_CF5329**

Set to “1” to include support for a Freescale CF532x USB device controller. Set to “0” to exclude support.

**SUD\_CF5373**

Set to “1” to include support for a Freescale CF537x USB device controller. Set to “0” to exclude support. Uses the CF5329 driver.

**SUD\_CF54455**

Set to “1” to include support for a Freescale CF5445x USB device controller. Set to “0” to exclude support. Uses the CF5329 driver.

**SUD\_CF548X**

Set to “1” to include support for a Freescale CF548x/7x USB device controller. Set to “0” to exclude support.

**SUD\_IMX1**

Set to “1” to include support for a Freescale i.MX1/MXL USB device controller. Set to “0” to exclude support.

**SUD\_IMX31**

Set to “1” to include support for a Freescale i.MX31 USB device controller. Set to “0” to exclude support. Uses the CF5329 driver.

**SUD\_ISP1161**

Set to “1” to include support for an NXP ISP1161 device controller. Set to “0” to exclude support.

**SUD\_IP3511**

Set to “1” to include support for a LPC55SXX device controller. Set to “0” to exclude support.

**SUD\_ISP1181**

Set to “1” to include support for an NXP ISP1181 device controller. Set to “0” to exclude support.

### **SUD\_ISP1362**

Set to “1” to include support for an NXP ISP1362 device controller. Set to “0” to exclude support.

### **SUD\_ISP158X**

Set to “1” to include support for an NXP ISP158x device controller. Set to “0” to exclude support.

### **SUD\_ISP1761**

Set to “1” to include support for an NXP ISP1761 device controller. Set to “0” to exclude support.

### **SUD\_LH7A4**

Set to “1” to include support for a Sharp LH7A400/4 USB device controller. Set to “0” to exclude support.

### **SUD\_LM3S**

Set to “1” to include support for a Luminary Micro LM3Sxxxx USB device controller. Set to “0” to exclude support.

### **SUD\_LPC**

Set to “1” to include support for an NXP LPCxxxx USB device controller such as that on LPC23xx, 24xx, 3180, and 32x0 processors. Set to “0” to exclude support.

### **SUD\_MAX342X**

Set to “1” to include support for a Maxim MAX342x USB device controller. Set to “0” to exclude support.

### **SUD\_NET2272**

Set to “1” to include support for PLX Net2272 USB device controller. We only tested it on the Analog Devices Blackfin USB-LAN EZ-EXTENDER board and BF533 EZLite board.

### **SUD\_PD720150**

Set to “1” to include support for the NEC uPD720150 USB controller. The driver has only been tested on the LPC1788 Embedded Artists board and uPD720150 Application Board (part #ET-D720150-HP) using a custom interface board.

Because the device controller and host controller share some registers, separate low-level register access routines are needed in the case the application needs both host and device features. Register access and pipe allocation need to be protected by a mutex. These routines are within upd720150.c/h, which is part of the BSP code (so both host and device controller driver can use it). uport.c/h does not include any uPD720150 low-level access code.

## SUD\_STM

Set to “1” to include support for STMicroelectronics STM32F103/2 and STM32F30x/37x USB device controllers. Set to “0” to exclude support.

## SUD\_STR7

Set to “1” to include support for an STMicroelectronics STR7 USB device controller such as that on STR710, STR711, and STR720 processors. Set to “0” to exclude support.

## SUD\_STR9

Set to “1” to include support for an STMicroelectronics STR9 USB device controller such as that on STR912 processor. Set to “0” to exclude support. SUD\_STR7 will also be set to “1” if you include this support.

## SUD\_SYNOPSISYS

Set to “1” to include support for Synopsys DesignWare USB IP. Only Slave mode is supported. Tested only on STM32F107/5 and STM32F20x processors.

**Note: Please only set ONE device controller macro to “1”. smxUSBBD does not currently support the use of multiple types of device controller simultaneously.**

## SUD\_HIGH\_SPEED

Set to “1” if your device controller supports USB 2.0 high speed capability. Set to “0” if it is only a full speed controller.

## On the Go Support

### SUD\_OTG

Set to “1” if smxUSBBD is being used with smxUSBO to support OTG feature. You must also enable smxUSBO in your project. Set to “0” to disable this feature.

## Device Support

### SUD\_COMPOSITE

Set to “1” to include support for a Composite device. You also need to enable at least two of the following devices, which will be combined into the composite device. Set to “0” to exclude support. **Some combinations of composite devices require you to write a Windows driver, such as Multiple Port Serial + Mass Storage. See the section 7.3 Composite Device Product and Interface IDs for details.**

**Note: When SUD\_COMPOSITE is 0, set only ONE of the following device function macros to “1”. Multiple device functions can be used only if composite device support is enabled.**

**SUD\_AUDIO**

Set to “1” to include support for an Audio device. Set to “0” to exclude support.

**SUD\_DFU**

Set to “1” to include support for a Device Firmware Upgrade device. Set to “0” to exclude support.

**SUD\_HIDCOM**

Set to “1” to include support for a HID device for simple communication. Set to “0” to exclude support.

**SUD\_KBD**

Set to “1” to include support for a Keyboard device. Set to “0” to exclude support.

**SUD\_MOUSE**

Set to “1” to include support for a Mouse device. Set to “0” to exclude support.

**SUD\_MSTOR**

Set to “1” to include support for Mass Storage class devices. Set to “0” to exclude support.

**SUD\_MTP**

Set to “1” to include support for Media Transfer Protocol (MTP) class devices. Set to “0” to exclude support. This driver also supports Picture Transfer Protocol (PTP).

**SUD\_NET**

Set to “1” to include support for an Ethernet over USB device. Set to “0” to exclude support. Requires a TCP/IP stack such as smxNS.

**SUD\_ECM****SUD\_NCM****SUD\_RNDIS**

Options for Ethernet over USB device. In order to let Windows, MacOS, Linux all support this function driver, enable both RNDIS and either ECM or NCM. Do not enable both ECM and NCM. For MacOS or Linux only, RNDIS may be disabled. For Windows only, both ECM and NCM may be disabled.

**SUD\_SERIAL**

Set to “1” to include support for a Serial Emulator device. Set to “0” to exclude support.

**SUD\_VIDEO**

Set to “1” to include support for a Video device. Set to “0” to exclude support.

## **SUD\_FTEMPL**

Set to “1” to include support for a generic function driver template. Set to “0” to exclude support.

## **SUD\_HID**

Set to “1” to include HID class specific descriptor handler. It is set to “1” automatically when SUD\_KBD or SUD\_MOUSE is set to “1”.

## **SUD\_CDC**

Set to “1” to include CDC class specific descriptor handler. It is set to “1” automatically when SUD\_NET is set to “1”.

## **SUD\_NEED\_IAD**

Set to “1” if your device has multiple interfaces and needs IAD to associate them together. It is set to “1” automatically when any of SUD\_AUDIO, SUD\_VIDEO, SUD\_NET or SUD\_COMPOSITE are set to “1”.

## **Miscellaneous Settings**

### **SUD\_MIN\_RAM**

Set to “1” for targets with very little RAM, such as SoC’s that don’t support external memory.

### **SUD\_EP0\_BUFFER\_SIZE**

Set the EP0 buffer data size. This size is related to which functions you have enabled. For example, the Audio function needs more data buffers to transfer class-specific descriptors.

### **SUD\_MANUFACTURER\_NAME**

Manufacturer’s name string, such as “MDI” for Micro Digital Inc.

### **SUD\_SELFPOWERED**

Set to “1” if your device is self powered. Set to “0” if your device needs USB to provide the power.

### **SUD\_DEBUG\_LEVEL**

Specifies the debug level. The following values are supported:

- 0 disables all debug output and debug statements are null macros
- 1 only output fatal error information
- 2 output additional warning information
- 3 output additional status information
- 4 output additional device change information

5 output additional data transfer information

6 output interrupt information

### **SUD\_BULK\_IN\_EPx, SUD\_BULK\_OUT\_EPx**

Logical Bulk type IN/OUT Endpoint number used by the function driver. See the note about Endpoint Settings below.

### **SUD\_INT\_IN\_EPx, SUD\_INT\_OUT\_EPx**

Logical Interrupt type IN/OUT Endpoint number used by the function driver. See the note about Endpoint Settings below.

### **SUD\_ISOC\_IN\_EPx, SUD\_ISOC\_OUT\_EPx**

Logical Isochronous type IN/OUT Endpoint number used by the function driver. See the note about Endpoint Settings below.

**Endpoint Settings:** Each endpoint number must be unique, and there may be limitations in the use of each endpoint by your USB controller. For example, the Sharp LH7A404 only has 4 endpoints, and EP3 is the only one that supports Interrupt IN transfers. The others have specific uses too. Better USB controllers have more endpoints, and each endpoint can be set for any transfer type. You must set the endpoint numbers as appropriate for your hardware.

### **SUD\_xxx\_PRODUCT\_NAME**

Product's name string, such as "smxUSBD Mouse"

### **SUD\_xxx\_VENDORID**

Vendor ID of your device. You may need to apply for a vendor ID from USB-IF.

### **SUD\_xxx\_PRODUCTID**

Product ID of your device. You can choose this number yourself.

### **SUD\_xxx\_NUMBER\_STR**

Serial number of your device. You can set it as an empty string ("") to disable this feature.

### **SUD\_xxx\_MAXPOWER**

Maximum power (in milliamps) that your device will request from the USB controller.

### **SUD\_xxx\_MAX\_NUM**

Maximum number of the same type devices you want to support. Currently, only mass storage and serial devices support multiple device instances. For Mouse and Ethernet over USB devices, only set it to 1.

### **SUD\_xxx\_IN\_ENDPOINT**

Bulk IN endpoint number for the function driver. The default value is SUD\_BULK\_IN\_EP but you may need to calculate it manually if you are using composite device.

### **SUD\_XXX\_OUT\_ENDPOINT**

Bulk OUT endpoint number for the function driver. The default value is SUD\_BULK\_OUT\_EP but you may need to calculate it manually if you are using composite device.

### **SUD\_XXX\_INT\_ENDPOINT**

Interrupt IN endpoint number for the function driver. The default value is SUD\_INT\_IN\_EP but you may need to calculate it manually if you are using composite device.

### **SUD\_DFU\_TRANSFER\_SIZE**

DFU download/upload transfer size.

### **SUD\_DFU\_INCLUDE\_INTERN\_FLASH**

### **SUD\_DFU\_INCLUDE\_SPI\_FLASH**

### **SUD\_DFU\_INCLUDE\_NOR\_FLASH**

### **SUD\_DFU\_INCLUDE\_NAND\_FLASH**

Specifies which segment your device has for the DFU driver. Set to “1” to enable that segment

### **SUD\_DFU\_WILL\_DETACH**

### **SUD\_DFU\_MANIFESTATION**

### **SUD\_DFU\_UPLOAD**

### **SUD\_DFU\_DNLOAD**

DFU driver capability of your device. Set to “1” to enable it.

### **SUD\_HID\_IN\_PACKET\_SIZE**

### **SUD\_HID\_OUT\_PACKET\_SIZE**

HID communication device input and output packet size. At most 64 bytes.

### **SUD\_HID\_IN\_INTERVAL**

### **SUD\_HID\_OUT\_INTERVAL**

HID communication device input and output polling interval.

### **SUD\_HID\_REPORT\_NUM**

Number of HID communication device reports.

### **SUD\_SERIAL\_USE\_INT\_ENDPOINT**

The default setting is ”1”, which lets you use the Windows built-in serial driver usbser.sys to support this device.

Set to “0” if you do not want to use an INT endpoint in the Serial function driver, to conserve endpoints, allowing support of more serial channels or more device functions. However you will need to use the Micro Digital Multiple Port USB-Serial Adapter Windows driver, available at extra cost.

## **SUD\_SERIAL\_SUPPORT\_ACM**

Set to “1” if you want the Linux CDC-ACM driver to support the serial function driver. It adds an interface descriptor to meet the ACM requirement. It also need to be set to “1” if you need to support multi-port serial configuration, unless you are using the Micro Digital multi-port serial driver. See section 9.1 Multiple Port Serial Device (or Single Port Limited Endpoints).

## **SUD\_SERIAL\_MTU**

Internal serial port receive ring buffer size. Reduce it if your system has limited RAM.

## **SUD\_MSTOR\_ASYNC\_ACCESS**

Set to “1” to create a separate task to handle the actual media access. This feature increases the performance of smxUSBD mass storage but requires a multitasking environment and requires more RAM since it increases the internal mass storage buffer size by  $3 * \text{SUD\_MSTOR\_PACKET\_SIZE}$ .

## **SUD\_MSTOR\_PACKET\_SIZE**

Internal Mass Storage buffer size. Reduce it if your system has limited RAM. Ensure it is a multiple of one sector’s size and at least one sector. See 10.4 Mass Storage Function Driver Buffer Size for details.

## **SUD\_MSTOR\_TASK\_STACK**

Mass storage async task stack size. Size depends on the low level disk driver. RAM disk may only need 100 bytes, but the smxNAND flash driver may need 1000.

## **SUD\_MTP\_PACKET\_SIZE**

Internal Media Transfer Protocol (MTP) transfer buffer size.

## **SUD\_VIDEO\_USE\_11**

Use UVC spec 1.1 or not. Because Windows XP and Vista can only support v1.0, we recommend you to set this option to 0 for best compatibility.

## **SUD\_VIDEO\_INCLUDE\_IN**

Set to 1 to include IN (like a web camera) interface in the video function driver.

## **SUD\_VIDEO\_INCLUDE\_OUT**

Set to 1 to include OUT (like a display) interface in the video function driver.

## **SUD\_VIDEO\_IN\_STILL\_IMAGE**

Set to 1 to include still image capture support. Not tested yet.

## **SUD\_VIDEO\_USE\_YUV422**

Set to 1 to use YUV422 for uncompressed data. Set to 0 to use YUV420.



### **SUD\_VIDEO\_IN\_FORMAT\_XXX**

Video Streaming IN Format. Enable at least one. MPEG2TS is not tested yet.

### **SUD\_VIDEO\_OUT\_FORMAT\_XXX**

Video Streaming OUT Format. Enable at least one. MPEG2TS is not tested yet.

### **3.1.2 udport.h**

The following are smxUSBBD-specific hardware-related configuration settings. In addition to these, it may be necessary to re-implement the functions in udhdw.c for your hardware. See chapter 8. Hardware Porting Notes.

### **SUD\_ISP1181\_BASE, SUD\_ISP1181\_IRQ**

Set to the base address and IRQ number for your device controller.

### **SUD\_ISP158X\_BASE, SUD\_ISP158X\_IRQ**

Set to the base address and IRQ number for your device controller.

## **3.2 Building the Code**

After configuring udcfg.h (see previous section), add the XUSBBD source files and paths to the application project, if not already in it.

## **3.3 Building and Running the Demos**

For non-SMX releases, a simple demo is provided in \SMX\APP\NORTOS\usbddemo.c. For SMX releases, the demos are in \SMX\APP\DEMO\usbddemo.c.

The demo file is integrated with the smx Protosystem. It is enabled just like all other SMX module demos, as documented in the SMX Quick Start.

Each device demo is enabled when that device function is enabled in udcfg.h (e.g. SUD\_MOUSE). If you enable composite device support, multiple demos can run at the same time, for example, mouse and mass storage device. Micro Digital recommends that you run the mouse demo first since it is the simplest.

Note that when you plug in the USB cable, Windows will detect your target board as a USB device and it will often pop up the “Found New Hardware” wizard. See Appendix C. Installing Devices under Windows 2000; Appendix D. Installing Devices under Windows XP; Appendix E. Installing Devices under Windows Vista, 7 for help with this. Be patient; it takes Windows several seconds to detect and mount the device.

The following is a summary of what each device demo does:

**HID Communication** This demo will get any data sent from the host and send the same data back to the host, using `usbhid.exe` (provided with `smxUSB`) on Windows. Please check the Windows host utility `usbhid` source code for how to communicate with this HID device in Windows. We currently do not provide a host application for any other operating system, such as Linux or OSX.

**Keyboard** This demo generates key press and release events and sends them to the PC. It sends the text message “`abcdefghijklmnopqrstuvwxy1234567890`” to the host and repeats forever. Please be sure to open an empty document first (e.g. Notepad) before you plug in the cable. Otherwise those simulated key inputs may corrupt whatever document is in the currently active window.

**Mouse** This demo generates mouse movement events and sends them to the PC. In Windows, you will see the mouse pointer move. By default it traces a diamond shape. Other choices can be selected in `demo.c` or `usbddemo.c`. Every time it traces the diamond, it pauses a few seconds so you can use your real mouse to control Windows, e.g. to stop the debugger. The demo purposely does not generate button events to avoid clicking on something and causing a problem.

**Mass Storage** This demo makes the target board look like a flash disk to the Windows PC. You can copy files to it, just like a normal flash disk. It uses 4MB, 8MB, or 16MB RAM on the target (configurable by setting `RAMDISK_SIZE` at the top of `usbddemo.c`). To test this demo, you can copy files to this disk and back to your hard disk and do a file compare against the originals. Also, you can copy PDF files to it and open them in Acrobat. The status information printed to the terminal indicates the number of bytes read from and written to the disk. The demo automatically formats the disk.

Tips:

1. If you change `RAMDISK_SIZE` and Windows stops seeing the disk, unplug and re-plug the USB cable.
2. Unplug and re-plug the cable to force Windows to read files from the mass storage device rather than the cache.

**Serial Port Emulator** This demo waits to receive some bytes from the USB host and then immediately echoes them back. If you run a terminal emulator program such as HyperTerminal or Tera Term, you can see the echo of what you typed.

You can also run against our `TestComm` utility (in the `BIN` dir). With `TestComm`, you can select a file, with different options for automated full duplex sending and receiving. On processors with limited internal RAM and no external RAM (i.e. those for which `SUD_MIN_RAM` is 1), packets can be lost due to insufficient heap, and the number of received bytes, as reported by `TestComm`, will be less than the number of sent bytes. The demo uses a 64-byte buffer for the incoming data. It calls `sud_SerialWriteData()` to echo the data back. But if that function fails to allocate a buffer, due to limited heap space, the received data cannot be sent and is lost. For these processors, specify a `TestComm Packet Size <= 64` bytes. `TestComm` sends out packets continuously. Windows waits 1 or 2 ms between packets, so if the `TestComm Packet Size` is `<= 64` bytes, the demo can keep up. Larger packets are broken into 64 byte frames, and Windows sends these frames immediately one after another, so the demo may not be able to keep up.

**Ethernet over USB** This demo needs a TCP/IP stack, such as `smxNS`, to run properly. It waits to receive some bytes from the USB host and then immediately echoes them back. To the OS, this USB

device looks like a Network Adapter, and you can use any program which can send and receive socket data to test it. You can also run against our TestSocket utility (Windows only, in the BIN dir).

**Audio** This demo sends pre-recorded sound data to the USB host. You can use the Windows Sound Recorder program to record the sound on Windows. The pre-recorded data format is 44100 samples/second, 16 bits per sample, and 2 channels. You need to set Sound Recorder to use the same format, if you plan to playback the data to this audio device. The demo displays the number of bytes it received from the host.

**Video** This demo sends pre-recorded video data (only one frame, 160x120 YUV422 format) to the USB host. You can go to the Windows Control Panel->Scanners and Cameras and double click the USB Video Device to view the video data. For OSX and Linux use the web camera utility of those OS to view the video data. Remember to set the resolution to 160x120.

**MTP** This demo uses the smxFS API to access the disk, so SMXFS must also be enabled. Your device will look like a Digital Still Camera device to Windows, and you can use a Windows built-in utility to import the pictures or videos. **The MTP interface implementation in usbddemo.c is not fully tested for all types of image files. You can use it as starting point but you need to carefully test it and make any necessary changes/fixes to meet your system's requirements.**

**DFU** This demo simulates firmware download and upload operations. Downloaded firmware is just discarded. Uploaded firmware is 32KB of a fixed pattern. You may need our Windows driver and utility usdbufu.exe or DFUHost.exe to run the demo, in the BIN directory. You can also use other DFU compatible driver, such as libUSB and dfu-util.exe to do the same demo.

### 3.4 Initialization

smxUSBBD is automatically initialized by an SMX application, if SMXUSBBD is defined by the application project file. This is done by smxusbdd\_init() which is called by smx\_modules\_init(), called by ainit(). For non-SMX applications, call sud\_Initialize() from your initialization code.

Note that the hardware initialization requires delays. These are done with a polling loop (see sb\_DelayUsec() and sb\_DelayMsec()). These are implemented using the constant **SB\_CPU\_MHZ**. Check bsp.h (in BSP) to ensure this is configured properly for your target. Otherwise, the delays could be much longer (or shorter) than expected.

## 4. Function Drivers

This section gives an overview of each device function and documents the API of each, in detail. The overall structure of the application interface to a device depends on the type of device.

### 4.1 Audio

The Audio function driver makes your device look like a simple sound card to the USB host, such as Windows, Macintosh, or Linux. You can include a speaker and/or microphone in this audio device so you can playback and/or record sound. Advanced features of a sound card such as Mixer Unit and Process Unit are not supported. You can integrate a MIDI port into this Audio device so it can also accept MIDI data. There is no need to install any driver or .inf file in Windows to support this device but you may need to implement the sound device driver by yourself, according to your system hardware and software environment to control your audio codec chipset.

The application must register a notification function to get events sent by the host. Possible events include start/stop recording, start/stop playback, and received playback data. The application also needs to send audio data to the host every 1 ms, after the recording is started by the host.

The application interface is defined in **udaudio.h**.

```
int      sud_AudioIsConnected(int port);
int      sud_AudioSendAudioData(int port, u8 *pData, int iLen);
int      sud_AudioGetAudioData(int port, u8 *pData, int iLen);
int      sud_AudioGetCurSpkSettings(int port, SUD_AUDIO_SETTINGS *pSettings);
int      sud_AudioGetCurMicSettings(int port, SUD_AUDIO_SETTINGS *pSettings);
int      sud_AudioSendMIDIData(int port, u8 *pData, int iLen);
int      sud_AudioGetMIDIData(int port, u8 *pData, int iLen);
void     sud_AudioRegisterNotify(int port, PAUDIOFUNC handler);
int      sud_AudioPackMIDIEvent(int port, u8 *pData, SUD_AUDIO_MIDI_EVENT *pEvent);
int      sud_AudioUnpackMIDIEvent(int port, u8 *pData, SUD_AUDIO_MIDI_EVENT *pEvent);
```

int **sud\_AudioIsConnected**(int port);

**Summary** Indicates whether the USB device is connected and the host controller has configured this device.

**Details** Can be called any time after `sud_Initialize()` has been called.

**Parameters** port Audio device port index. For the current version, only pass 0.

**Returns** 1 Host has configured this device.  
0 Cable is not connected or the host does not support this device.

**See Also** `sud_AudioRegisterNotify()`

int **sud\_AudioSendAudioData**(int port, u8 \*pData, int iLen);

**Summary** Send some Audio streaming data (sound) to the host.

**Details** When the Host is recording the sound, call this function to send the sound data to the host. Normally this function should be called after you get a MIC\_START event, every 1 ms. Stop calling it after you get a MIC\_STOP event. The data in the buffer should be the data for one (1) millisecond. For example, if the recording format is 48000 samples/second, 16 bits and 2 channels/sample, then the data length for 1 millisecond is calculated as  $48000 * 16 / 8 * 2 / 1000 = 192$  bytes

**Parameters** port Audio device port index. For the current version, only pass 0.  
pData The data buffer pointer.  
iLen The length of the buffer.

**Returns** 0 Data have been sent to the host.  
-1 Error occurred when sending data.

**See Also** `sud_AudioRegisterNotify()`

int **sud\_AudioGetAudioData**(int port, u8 \*pData, int iLen);

**Summary** Get the received Audio streaming data (sound).

**Details** When Host is playing back the sound, call this function to get the received sound data from the host. Normally it should be called after you get an ISOCDATAREADY event.

**Parameters** port Audio device port index. For the current version, only pass 0.  
pData The data buffer pointer.  
iLen The length of the buffer.

**Returns** > 0 Received data length.  
-1 Error occurred when receiving data.

**See Also** sud\_AudioRegisterNotify()

int **sud\_AudioSendMIDIData**(int port, u8 \*pData, int iLen);

**Summary** Send some MIDI streaming data (notes) to the host.

**Details** When Host is recording the music, call this function to send the MIDI data to the host.

**Parameters** port Audio device port index. For the current version, only pass 0.  
pData The data buffer pointer.  
iLen The length of the buffer.

**Returns** 0 Data have been sent to the host.  
-1 Error occurred when sending data.

**See Also** sud\_AudioRegisterNotify()

int **sud\_AudioGetMIDIData**(int port, u8 \*pData, int iLen);

**Summary** Get the received MIDI streaming data (notes).

**Details** When Host is playing back the music, call this function to get the received MIDI data from the host. Normally you should call this function after you get a BULKDATAREADY event.

**Parameters** port Audio device port index. For the current version, only pass 0.  
pData The data buffer pointer.  
iLen The length of the buffer.

**Returns** > 0 Received data length.  
-1 Error occurred when receiving data.

**See Also** sud\_AudioRegisterNotify()

int sud\_AudioGetCurSpkSettings(int port, SUD\_AUDIO\_SETTINGS \*pSettings);

**Summary** Get current Speaker sound format settings.

**Details** Before Host plays back the sound, it will set up the sound data format. Call this function to get the format set by the host. Normally you should call this function after you get a SPK\_START event. You may need to set the playback hardware to match these settings. SUD\_AUDIO\_SETTINGS is defined as:

```
typedef struct
{
    uint iFormat;
    uint iChannels;
    uint iBits;
    u32 dwSampleRate;
    u16 wVolume[7];
    u16 wMuteEnabled[7];
}SUD_AUDIO_SETTINGS;
```

iFormat is the format of the data, for example, PCM.

iChannels is the number of data channels, for example, 1 or 2.

iBits is the number of bits per sample, for example 8 or 16.

dwSampleRate is the current sample rate, for example, 48000.

wVolume is the current volume settings for each channels.

wMuteEnabled is the mute feature for each channel.

**Parameters** port Audio device port index. For the current version, only pass 0.  
pSettings Audio settings structure pointer. See above.

**Returns** 0 Get the current settings.  
-1 Error occurred when getting the settings.

**See Also** sud\_AudioRegisterNotify()

int **sud\_AudioGetCurMicSettings**(int port, SUD\_AUDIO\_SETTINGS \*pSettings);

**Summary** Get current Microphone sound format settings.

**Details** Before Host records the sound, it will set up the sound data format. Call this function to get the format set by the host. Normally you should call this function after you get a MIC\_START event. You may need to set the recording hardware to match these settings

**Parameters** port Audio device port index. For the current version, only pass 0.  
pSettings Audio settings structure pointer. See above.

**Returns** 0 Get the current settings.  
-1 Error occurred when getting the settings.

**See Also** sud\_AudioRegisterNotify()

void **sud\_AudioPackMIDIEvent** (int port, u8 \*pData, SUD\_AUDIO\_MIDI\_EVENT \*pEvent);

**Summary** Generate a USB packet from a MIDI event.

**Details** You must call this function generate a USB packet from a MIDI event.  
A MIDI event is defined as:

typedef struct

```
{  
    uint iCN;  
    uint iCIN;  
    uint iMIDI_0;  
    uint iMIDI_1;  
    uint iMIDI_2;  
}SUD_AUDIO_MIDI_EVENT;
```

iCN is the cable number

iCIN is the Code Index Number.

iMIDI\_0, iMIDI\_1, iMIDI\_2 is the MIDI data. For details, please refer to USB Device Class Definition for MIDI Device.

**Parameters** port Audio device port index. For the current version, only pass 0.  
pData USB MIDI packet buffer pointer. Data length should be 4 bytes.  
pEvent Pointer to MIDI event.

**Returns** 0 Success.  
-1 Error occurred.

**See Also** sud\_AudioSendMIDData()



void **sud\_sud\_AudioUnpackMIDIEvent** (int port, u8 \*pData, SUD\_AUDIO\_MIDI\_EVENT \*pEvent);

**Summary** Parse a USB packet to generate a MIDI event.

**Details** You must call this function to parse the received USB packet to generate a MIDI event.

**Parameters**

port	Audio device port index. For the current version, only pass 0.
pData	USB MIDI packet buffer pointer received by sud_AudioGetMIDIData(). Data length should be 4 bytes.
pEvent	Pointer to MIDI event.

**Returns**

0	Success.
-1	Error occurred.

**See Also** sud\_AudioGetMIDIData()

void **sud\_AudioRegisterNotify**(int port, PAUDIOFUNC handler);

**Summary** Set the notification handler for the audio device.

**Details** You must call this function to register an event notification handler so you can process events for the USB audio device.

The notification is defined as:

SUD\_AUDIO\_NOTIFY\_ISOCDATAREADY  
SUD\_AUDIO\_NOTIFY\_BULKDATAREADY  
SUD\_AUDIO\_NOTIFY\_SPK\_START\_STOP  
SUD\_AUDIO\_NOTIFY\_MIC\_START\_STOP

**Parameters**

port	Audio device port index. For the current version, only pass 0.
handler	Function pointer for the notification handler.

**Returns** none

**See Also** none

## 4.2 Device Firmware Upgrade (DFU)

The DFU function driver provides part of the ability to update the firmware in your device. Specifically, it implements the communication protocol in the USB DFU specification. You must handle the format of the image file, manifest it in flash, and create a bootloader (if desired), as explained in this section.

A USB DFU enabled device normally works in two modes: Runtime mode and DFU mode. Runtime mode only report to the host that this device has DFU capability, and the device will work mainly for other functions such as serial and mass storage. In this case you also need composite device framework. Whenever the USB host needs to download or upload the firmware, the device needs to switch to the DFU mode first, and then all run time functions, such as serial, mass storage, etc., will be disabled.

It is also possible to use the DFU mode directly for firmware upgrade only in a boot loader

The DFU specification does not define the format of the firmware data, it can be binary or hex or any other format (even encrypted) that will be used on both host and device side. Encode/decode of the format of the firmware itself should be done by your host application and device. It is transparent to DFU. The suffix, which contains the CRC of the firmware data, version, and vendor information, needs to be added to the firmware file before the download process begins so the DFU driver can check and verify it before the real download procedure starts. Our windows driver provides the API to add and check the suffix.

USB DFU specification only defines the communication protocol to transfer the firmware over the USB bus. It does NOT define how you will write the firmware to the flash and manifest the firmware. For example you may write the binary firmware code directly to flash or you may use some kind of mapping layer to translate the physical and logical flash address, like smxNAND or smxNOR. To manifest the firmware, you may create two partitions for your flash, one is active partition and another is backup partition and you may need to switch between these two partitions after the upgrade is done. **All these details must be done by your application and are not handled by the DFU driver.** The DFU driver provides you the APIs to let your application to register the interface to access your flash (`sudo_DFURegisterInterface()`) and notify the host when the manifestation is done (`sudo_DFUWriteDone()`).

The Windows operating system does not have a built-in driver for DFU, so you need to use the driver provided by MDI or another driver such as the open source drivers libUSB and dfu-util.exe. See section 9.2 Device Firmware Upgrade (DFU) Device for more information about the requirements for the host side. On the device side, the DFU function may be part of your runtime function such as serial or mass storage, in which case you also need to composite device framework.

The device-side application interface is defined in **uddfu.h**.

```
int      sudo_DFUIsConnected(void);
void     sudo_DFURegisterInterface (SUD_DFU_IF *pIF);
void     sudo_DFUWriteDone (uint result, uint condition);
int      sudo_DFUIsRuntimeMode(void);
```

int **sud\_DFUIsConnected** (void)

**Summary** Check if the DFU device is connected and enumerated by the USB host.

**Details** Check if the DFU device is connected and enumerated by the USB host. Can be called any time after `sud_Initialize()` has been called.

**Parameters** none

**Returns** 1 Device connected and enumerated to the host.  
0 Device is not connected or enumerated to the host.

**See Also** none

void **sud\_DFURegisterInterface** (SUD\_DFU\_IF \*pIF)

**Summary** Register the DFU application interface function to the DFU

**Details** The application should implement its hardware related memory function and register it to the DFU function driver so the DFU does not need to know anything special about the target memory. Nonvolatile memory write operations are normally time-consuming, so we recommend doing the actual write operation in a separate task. See the demo code in our `usbddemo.c` for how we simulate this kind of asynchronous operation.

**Parameters** pIF The interface of the application memory  
SUD\_DFU\_IF is defined as

```
typedef struct
{
    int (*DFUInit)(uint segment);
    int (*DFUWrite)(uint segment, u32 dwOffset, u8 *pData, uint iSize);
    int (*DFURead)(uint segment, u32 dwOffset, u8 *pData, uint iSize);
    int (*DFUDone)(uint segment);
} SUD_DFU_IF;
```

**DFUInit()** is called whenever the DFU driver needs to init the firmware segment. You may return the status of that segment, which is defined in `uddfu.h` as `SUD_DFU_STATUS_XXX`. For example, if the device could not find the valid firmware on that segment, return `SUD_DFU_STATUS_errFIRMWARE`. Otherwise, return `SUD_DFU_STATUS_OK`.

**DFUWrite()** is called whenever the DFU driver receives a downloaded packet of the firmware. You can buffer the received data until it will fill one whole block. `segment` is the memory segment. `dwOffset` is the offset of this packet from the beginning of that

segment. pData is the data buffer. iSize is the data buffer size. Return value should be the data size written to the buffer or memory.

**DFURead()** is called whenever the DFU driver needs the firmware data to upload. segment is the memory segment. dwOffset is the offset of this packet from the beginning of that segment. pData is the data buffer. iSize is the data buffer size. Return value is the read data size. If the firmware size is a multiple of the packet size, the last read operation should return 0.

**DFUDone()** is called whenever the DFU driver has no more data to download. The application may need to do the manifestation after this function is called.

**Returns** none

**See Also** sud\_DFUWriteDone()

void **sud\_DFUWriteDone** (uint result, uint condition)

**Summary** Notifies the DFU driver that the memory write or manifestation is done.

**Details** Normally flash write operations take awhile, so we recommend the application do it in a separate task instead of in the DFUWrite() interface function. If DFUWrite() will only write the downloaded firmware to a buffer, the application still needs to call this function after the memory copy.

**Parameters**

result	The result of this block write or manifestation. It is defined as SUD_DFU_STATUS_XXX in uddfu.h
condition	Which write operation is done, either SUD_DFU_WRITE_BLOCK_DONE or SUD_DFU_MANIFESTATION_DONE.

**Returns** none

**See Also** sud\_DFURegisterInterface ()

void **sud\_DFUIsRuntimeMode** (void)

**Summary** Checks if the DFU device is in the runtime mode.

**Details** A DFU device can work under two modes. Runtime mode is actually a composite device, and the DFU function only reports the DFU capacity to the USB host. The host will send the DFU request to the device DFU runtime interface to let it switch to the DFU mode. The

host will reset and re-enumerate device after the device switches to the DFU mode. Under DFU mode, the device will report different descriptors to the host, so the host can do the actual firmware download/upload.

**Parameters** none.

**Returns** 1 Device is in the runtime mode.  
0 Device is in the DFU mode.  
1

**See Also** none

### 4.3 HID Communication

The HID communication function driver can be used to transfer data to and from a USB host. You don't need to install any driver on the Windows PC, but you need to write a special Windows application to send and receive data.

An HID communication device will use both Interrupt IN and OUT endpoints to do full duplex data transfer between USB device and host. However, the Interrupt endpoint is not designed to transfer a large amount of data, so the data transfer speed of the HID Communication device is slow.

The device-side application interface is defined in **udhidcom.h**.

```
int      sud_HIDIsConnected(void);  
int      sud_HIDSendInput(uint iReportID, void *pDataBuf, uint size);  
void     sud_HIDRegisterOutputNotify(PHIDFUNC handler);
```

```
int sud_HIDIsConnected(void);
```

**Summary** Indicates whether this USB device is connected and the host controller has configured it.

**Details** Can be called any time after sud\_Initialize() has been called.

**Parameters** none

**Returns** 1 Host has configured this device.  
0 Cable is not connected or the host does not support this device.

**See Also** sud\_HIDRegisterOutputNotify()

int **sud\_HIDSendInput** (uint iReportID, void \*pDataBuf, uint size);

**Summary** Send data to the USB host.

**Details** Can be called any time after `sud_Initialize()` has been called.

**Parameters** iReportID the report ID of the input; if there is only one report, use 1  
pDataBuf pointer to the data buffer  
Size size of the data

**Returns** 1 Data sent out.  
0 Cable is not connected, the host does not support this device, or send data failed.

**See Also** `sud_HIDRegisterOutputNotify()`

void **sud\_HIDRegisterOutputNotify** (PHIDFUNC handler);

**Summary** Register the output data ready callback function.

**Details** Can be called any time after `sud_Initialize()` has been called.  
The callback function is defined as  
typedef void (\* PHIDFUNC)(uint iReportID, void \*pDataBuf, uint size);  
iReportID is the output report ID.  
pDataBuf is the output (incoming for device) payload data pointer. The data does not include report id.  
size is the output payload data size.  
Within the callback function, the application needs to copy the data to the application buffer and wake up another task to process the data. You cannot block or wait for something within this callback function.  
See our HID communication sample code in `usbddemo.c`.

**Parameters** handler callback function pointer

**Returns** none

**See Also** `sud_HIDSendInput()`

## 4.4 Keyboard

The Keyboard function driver makes your device look like an HID keyboard to the USB host, such as Windows, Macintosh, or Linux. The interface is simple. To use the function driver, the application simply sends keyboard event such as key pressed or released to the host.

The application interface is defined in **udkbd.h**.

```
int      sud_KBDInput(u8 bModifier, u8 *pKey, uint count);
```

```
int sud_KBDInput (u8 bModifier, u8 *pKey, uint count)
```

**Summary** Sends a key event to the USB host.

**Details** Sends a key press/release event to the USB host, indicating key modifier and state of multiple keys pressed/released. Can be called any time after sud\_Initialize() has been called.

**Parameters** bModifier The Modifier bitmap value for special function keys. They are defined as:

```
SUD_KBD_LEFT_CTRL
SUD_KBD_LEFT_SHIFT
SUD_KBD_LEFT_ALT
SUD_KBD_LEFT_GUI
SUD_KBD_RIGHT_CTRL
SUD_KBD_RIGHT_SHIFT
SUD_KBD_RIGHT_ALT
SUD_KBD_RIGHT_GUI
```

pKey Up to 6 bytes of data for the key state. Each byte presents one key. The key code is defined in udkbd.h as SUD\_KEY\_CODE\_xx. To see how to use the key codes, look at the demo code in usbddemo.c.

count The data size of the pKey. Maximum value is 6.

**Returns** 1 Keyboard event has been sent to the host.  
0 Cable is not connected or the host does not support this device.

## 4.5 Mass Storage

The Mass Storage function driver makes your device look like a removable disk to the USB host, such as Windows, Macintosh, or Linux. You can copy files to and from it. The Mass Storage Specification only defines the protocol for transferring media disk data, at the sector level. It did not involve any details about how to read/write your actual media, such as flash or SD card. The application must register a disk driver so the mass storage driver can call the disk driver interface functions to write data to or read data from the actual media. The disk driver interface functions that must be supplied are shown in Appendix B. Block Device Driver Interface. It is a typical block-device driver interface, as is used in smxFS and other file systems. We provide some drivers such as RAM disk, NAND flash, NOR flash, SD/MMC card, CompactFlash, etc. which can be used in your application, but if you are using other media which we don't support yet, you may need to provide/implement it yourself.

If `SUD_MSTOR_ASYNC_ACCESS` is set to 1 in `udcfg.h`, the mass storage driver will create a separate task to handle the disk access operations, while the USB is still transferring the required data. Part of the data can be read/written to the disk simultaneously. This asynchronous operation will improve mass storage device driver performance significantly, especially if your disk access speed and USB data transfer speed are almost the same. Theoretically, the overall performance should be the slower of the USB data transfer speed and disk access speed.

The application interface is defined in **`udmstor.h`**.

```
void      sud_MSRegisterDisk(const SBD_IF *pDiskOper, int iLUN);
```

```
void sud_MSRegisterDisk(const SBD_IF *pDiskOper, int iLUN);
```

**Summary** Registers a disk driver to the mass storage function driver.

**Details** The mass storage driver calls the disk driver functions to read/write the data transferred by USB to the real storage media. Can be called any time after `sud_Initialize()` has been called.

**Parameters**

<code>iLUN</code>	The logical unit number of this disk. Should be 0 to <code>SUD_MSTOR_MAX_NUM - 1</code> .
<code>pDiskOper</code>	The generic block device driver interface. The block device driver interface is also used in our smxFS product. See Appendix B. Block Device Driver Interface for the details about how to implement this interface.

**Returns** None



## 4.6 Media Transfer Protocol (MTP)

The MTP function driver makes your device look like a media device to the USB host, such as Windows, Macintosh, or Linux. Unlike the mass storage function driver, which is based on block device level access, the MTP function driver is based on file level access. File system details are beyond the scope of MTP specification. You need to add/implement the file system interface in your application. `usbddemo.c` provides sample code to interface to `smxFS` that can be used in your application if you are using `smxFS`. There is no need to install any driver or `.inf` file in Windows to support this device but you may need to implement the file system interface by yourself, according to your system's hardware and software environment. You may also need to implement the interface to get the properties of the files your device will support, such as the JPEG image width, height, and thumbnail.

Capture-related functions must also be implemented by the application according to your hardware and software.

The application interface is defined in **`udmtp.h`**.

```
int      sud_MTPIsConnected(int port);
void     sud_MTPRegisterInterface(const SUD_MTP_IF *pObjOper);
int      sud_MTPSendEvent(u32 dwEventCode, uint iNumParameter, u32 *pdwParameter);
```

```
int sud_MTPConnected(int port);
```

**Summary**      Indicates whether the USB device is connected and the host controller has configured it.

**Details**        Can be called any time after `sud_Initialize()` has been called.

**Parameters**    `port`    MTP device port index. For the current version, only pass 0.

**Returns**        1        Host has configured this device.  
                  0        Cable is not connected or the host does not support this device.

**See Also**       none

```
void sud_MTPRegisterInterface(const SUD_MTP_IF *pObjOper);
```

**Summary**        Register the application level interface for the MTP device.

**Details**        Can be called any time after `sud_Initialize()` has been called.

**Parameters** pObjOper Application interface to support the operations of this MTP device. The application interface is defined as:

```
typedef struct
{
    int (*MTPDiskMounted)(uint index);
    int (*MTPStorageNum)(void);
    void (*MTPOpen)(uint index, char *pFileName, uint iMode);
    int (*MTPClose)(void *pFileHandle);
    int (*MTPRead)(u8 * pRAMAddr, u32 dwSize, void *pFileHandle);
    int (*MTPWrite)(u8 * pRAMAddr, u32 dwSize, void *pFileHandle);
    int (*MTPMkDir)(uint index, char *pPathName);
    int (*MTPRmdir)(uint index, char *pPathName);
    int (*MTPDelete)(uint index, char *pFileName);
    int (*MTPTotalSize)(uint index);
    int (*MTPFreeSize)(uint index);
    int (*MTPFindFirst)(uint index, char *pFindSpec, SUD_MTP_FILE_INFO
        *pFileInfo);
    int (*MTPFindNext)(uint iID, SUD_MTP_FILE_INFO *pFileInfo);
    int (*MTPFindClose)(SUD_MTP_FILE_INFO *pFileInfo);
    int (*MTPGetImgProp)(uint index, char *pFileName, uint *piWidth,
        uint *piHeight, uint *piBits, uint *piThumbSize, uint *piThumbFormat,
        uint *piThumbWidth, uint *piThumbHeight);
    void (*MTPOpenImgThumb)(uint index, char *pFileName);
    int (*MTPReadImgThumb)(u8 * pRAMAddr, u32 dwSize, void *pHandle);
    int (*MTPCloseImgThumb)(void *pHandle);
    int (*MTPGetFormat)(uint index, char *pFileName, u16 *pwFormat);
    int (*MTPInitCapture)(u32 index, u32 format);
    int (*MTPStartCapture)(void);
    int (*MTPStopCapture)(void);
} SUD_MTP_IF;
```

MTP function driver will call **MTPDiskMounted()** when the USB host tries to enumerate all the storage IDs. Parameter index is the index of the disk.

MTP function driver will call **MTPStorageNum()** to get the total disk on your device.

MTP function driver will call **MTPOpen()** to open an object (file) on your disk. Index is the storage (disk) ID. pFileName is the full path of that object, for example “\path1\image1.jpg”. If your file system is not using “\” as path separator, change it within this function before you pass the file name to your file system call. iMode is one of the following constants:

```
SUD_MTP_OBJ_MODE_READWRITE
SUD_MTP_OBJ_MODE_READONLY
```

The return value should be the file handle. It will be used by the following MTPRead()/MTPWrite() operation.

MTP function driver will call **MTPClose()** to close the opened object. Parameter pFileHandle is the file handle returned by MTPOpen().

MTP function driver will call **MTPRead()** to read the data of the object and send them to the USB host. Parameter pRAMAddr is the pointer to the data buffer. dwSize is the size of the data buffer. pFileHandle is the file handle returned by MTPOpen().

MTP function driver will call **MTPWrite()** to write the object data from the USB host. Parameter pRAMAddr is the pointer to the data buffer. dwSize is the size of the data buffer. pFileHandle is the file handle returned by MTPOpen(). Currently it is not used.

MTP function driver will call **MTPMkdir()** to create a new file folder on your disk. Index is the storage ID. pPathName is the full path. Currently it is not used.

MTP function driver will call **MTPRmdir()** to remove an existing file folder on your disk. Index is the storage ID. pPathName is the full path. Currently it is not used.

MTP function driver will call **MTPDelete()** to remove an existing object on your disk. Index is the storage ID. pFileName is the full path name of that object.

MTP function driver will call **MTPTotalSize()** to get the total size of the storage. Index is the storage ID. The unit of the return value is KB (1024 bytes).

MTP function driver will call **MTPFreeSize()** to get the free size of the storage. Index is the storage ID. The unit of the return value is KB (1024 bytes).

MTP function driver will call **MTPFindFirst()** to get/list the object information on your disk. Index is the storage ID. pFindSpec is the search pattern, such as “\DCIM\\*.\*” or “DCIM\Image1.jpg”. pFileInfo is a pointer to SUD\_MTP\_FILE\_INFO to hold the object’s information. SUD\_MTP\_FILE\_INFO is defined as:

```
typedef struct
{
    u8          name[260];
    uint        attr;
    u32         st_size_l;
    u32         st_size_h;
    SUD_MTP_DATETIME st_ctime;
    SUD_MTP_DATETIME st_mtime;
    SUD_MTP_DATETIME st_atime;
    void        *pPrivate;
}SUD_MTP_FILE_INFO;
```

Return value is the internal search ID. -1 means no more files found.

MTP function driver will call **MTPFindNext()** to search more object information on your disk. iID is the internal search ID returned by MTPFindFirst()/MTPFindNext(). pFileInfo is a pointer to SUD\_MTP\_FILE\_INFO to hold the object's information.

MTP function driver will call **MTPFindClose()** when the search is finished, You can release any resources allocated by MTPFindFirst() here.

MTP function driver will call **MTPGetImgProp()** to get the image's properties of the object on your disk. Index is the storage (disk) ID. pFileName is the full path of that object. piWidth is the width of that image. piHeight is the height of that image. piBit is the color depth of that image. piThumbSize is the size of the thumbnail image in bytes. piThumbFormat is the image format of the thumbnail. piThumbWidth is the width of the thumbnail. piThumbHeight is the height of the thumbnail.

MTP function driver will call **MTPOpenImgThumb()** to open an image's thumbnail on your disk. Index is the storage (disk) ID. pFileName is the full path of that object. Thumbnail maybe stored in the EXIF/JPEG file or in separate thumbs.db file. It depends on your implementation. The return value should be the file handle. It will be used by the following MTPReadImgThumb()/MTPCloseImgThumb() operation.

MTP function driver will call **MTPReadImgThumb()** to read the thumbnail of the object and send it to the USB host. Parameter pRAMAddr is the pointer to the data buffer, dwSize is the size of the data buffer, pHandle is the file handle returned by MTPOpenImgThumb()

MTP function driver will call **MTPCloseImgThumb()** to close the open thumbnail. Parameter pHandle is the file handle returned by MTPOpenImgThumb().

MTP function driver will call **MTPGetFormat()** to get the format of the object. Parameter index is the storage ID. pFileName is the full path name of that object. pwFormat is the returned the format, which is one of the SUD\_MTP\_OBJ\_FMT\_XXX defined in udmtph

MTP function driver will call **MTPInitCapture()** to prepare and set up the capture format. Index is the storage ID. format is the object format you want to capture, which is one of the SUD\_MTP\_OBJ\_FMT\_XXX defined in udmtph.

MTP function driver will call **MTPStartCapture()** to start the capture.

MTP function driver will call **MTPStopCapture()** to stop the capture.

**Returns** none

**See Also** none

```
int MTPSendEvent(u32 dwEventCode, uint iNumParameter, u32 *pdwParameter);
```

**Summary** Send MTP event to the host, such as the storage is removed.

**Details** Can be called any time after `sud_MTPIsConnected()` returns 1.

**Parameters**

<code>dwEventCode</code>	MTP event. For details check the MTP specification.
<code>iNumParameter</code>	The number of parameters of the event. For details, check the MTP specification for what parameters you should pass for each event.
<code>pdwParameter</code>	The array of parameters. Each parameter is 32-bit data, which is one of the <code>SUD_MTP_EVT_XXX</code> defined in <code>udmtp.h</code> .

**Returns**

0	Event is sent out
-1	Send failed

**See Also** none

## 4.7 Mouse

The Mouse function driver makes your device look like an HID mouse to the USB host, such as Windows, Macintosh, or Linux. The interface is simple. To use the function driver, the application simply sends a mouse event, such as new coordinates and/or button pressed or released, to the host. It moves the mouse cursor on your PC.

The application interface is defined in **udmouse.h**.

```
int sud_MouseInput(u16 button, s16 x, s16 y, s8 wheel);
```

```
int sud_MouseInput (u16 button, s16 x, s16 y, s8 wheel)
```

**Summary** Sends a mouse event to the USB host.

**Details** Sends a mouse event to the USB host, indicating button state, position, and wheel state. Can be called any time after `sud_Initialize()` has been called.

**Parameters**    button    The bitmap encoding the button state. When a bit is set to ‘1’, it indicates that the corresponding button is pressed. The following buttons are supported:

SUD\_MOUSE\_BTN\_LEFT  
SUD\_MOUSE\_BTN\_RIGHT  
SUD\_MOUSE\_BTN\_MIDDLE  
SUD\_MOUSE\_BTN\_SIDE  
SUD\_MOUSE\_BTN\_EXTRA

x            The movement along the x-axis, using a right-handed coordinate system. If the user is facing the USB mouse, then the reported values should increase as the mouse is moved from left to right. Movement is relative to the current mouse position. A negative value moves the pointer left.

y            The movement along the y-axis, using a right-handed coordinate system. If the user is facing the USB mouse, then the reported values should increase as the mouse is dragged nearer to the user. Movement is relative to the current mouse position. A negative value moves the pointer up.

wheel    The direction of mouse wheel rotation. SUD\_MOUSE\_WHEEL\_UP indicates that the wheel is rolling up, and SUD\_MOUSE\_WHEEL\_DOWN indicates that the wheel is rolling down.

**Returns**        1            Mouse event has been sent to the host.  
                  0            Cable is not connected or the host does not support this device.

## 4.8 Ethernet over USB

The Ethernet over USB function driver makes your device look like a Network Adapter to a Windows or MacOS/Linux USB host. This function can be configured to use either RNDIS or CDC-ECM/NCM or both. RNDIS is a Microsoft extension for the CDC Ethernet driver which MacOS does not support. In order to let Windows and MacOS/Linux to support this driver without any custom driver, you need to enable both RNDIS and CDC-ECM or NCM. The PC can communicate with this device via Ethernet data packets. Normally you will need a TCP/IP stack on your device and use the APIs provided by this function driver to emulate an Ethernet device and add it to your network stack. The Ethernet over USB function driver has been integrated with Micro Digital’s TCP/IP stack, smxNS. For other TCP/IP stacks, you need to add a virtual Ethernet driver (using the following APIs) to your stack. The USB host and your device can communicate with each other by TCP/IP with a USB cable instead of an Ethernet cable. One use of Ethernet over USB is to allow configuring a device from the web browser on a host PC communicating with a web server on your device. This is especially useful if your processor has only a USB device controller and no Ethernet controller on chip.

**Note: If both RNDIS and CDC-ECM or NCM are enabled you cannot configure it also as part of composite device.**

The application interface is defined in **udnet.h**.

```
int      sud_NetIsPortConnected(int port);
int      sud_NetIsAggregationSupported(int port);
int      sud_NetWriteData(int port, u8 *pBuf[], u16 len[], uint packets);
void     sud_NetRegisterPortNotify(int port, PNETFUNC handler);
void     sud_NetSetEthernetAddr(int port, u8 *pMACAddr);
```

int **sud\_NetIsPortConnected** (int port)

**Summary** Indicates whether the USB device is connected and the host controller has configured this device.

**Details** Can be called any time after `sud_Initialize()` has been called.

**Parameters** port Ethernet over USB device port index. For the current version, only pass 0.

**Returns** 1 Host has configured this device.  
0 Cable is not connected or the host does not support this device.

**See Also** `sud_NetRegisterPortNotify()`

int **sud\_NetIsAggregationSupported** (int port)

**Summary** Indicates whether the USB is configured to support packets aggregation. For the current version only CDC-NCM will support packets aggregation.

**Details** Can be called any time after `sud_Initialize()` has been called.

**Parameters** port Ethernet over USB device port index. For the current version, only pass 0.

**Returns** 1 Host has configured this device to the mode that support packets aggregation.  
0 This device currently does not support packets aggregation.

**See Also** `sud_NetRegisterPortNotify()`

int **sud\_NetWriteData**(int port, u8 \*pBuf[], u16 len[], uint packets)

**Summary** TCP/IP calls to send Ethernet data to the USB host.

**Details** Can be called any time after `sud_Initialize()` has been called to send Ethernet data. If CDC-NCM is enabled and configured by the USB host, this function allows the Ethernet over USB driver to aggregate multiple Ethernet frames into one USB transfer. It is very useful if the TCP/IP stack has multiple small packets it needs to send. The system also needs to be configured to have enough USB buffers to send that aggregated USB transfer. If the USB host did not configure to use NCM mode, the function will only send the first packet in the array. Other elements, if supplied, will be ignored. Use `sud_NetIsAggregationSupported()` to check if aggregation is supported.

**Parameters** port Ethernet over USB device port index. For the current version, only pass 0.  
pBuf Data buffer pointer array if aggregation is supported.  
len Data buffer length array if aggregation is supported.  
packets Number of packets of the packet array.

**Returns**  $\geq 0$  The data length (number of bytes) sent to the host.  
-1 Cable is not connected or out of memory or the host does not support this device.

**See Also** `sud_NetRegisterPortNotify()`, `sud_NetIsAggregationSupported()`

void **sud\_NetRegisterPortNotify**(int port, PNETFUNC handler)

**Summary** Registers a notification function for the Ethernet over USB device.

**Details** This notification function will be called after the Ethernet over USB function driver receives new data. Can be called any time after `sud_Initialize()` has been called.

**Parameters** port Ethernet over USB device index. For the current version, only pass 0.  
handler Function pointer to the notification function. The function is defined as:  
typedef int (\* PNETFUNC)(int port, u8 \*pBuf, uint len);  
pBuf is the new received data buffer pointer and len is the buffer length. You need to copy the data to another buffer if you need to use the data after this function returns.

**Returns** None

**See Also** `sud_NETWriteData()`



void **sud\_NetSetEthernetAddr**(int port, u8 \* pMACAddr)

**Summary** Set up the Ethernet MAC address.

**Details** Can be called any time after `sud_Initialize()` has been called. An Ethernet over USB device is not a real Ethernet adapter so the TCP/IP stack must tell this driver what its Ethernet address is.

**Parameters** port Ethernet over USB device index. For the current version, only pass 0.  
pMACAddr Ethernet address; a 6-byte array.

**Returns** None

**See Also** None

## 4.9 Serial (CDC-ACM)

The Serial function driver makes your device look like one or more COM ports to a Windows 7, Vista, XP, or 2000 USB host. For serial devices, the application can call the API functions immediately since they are self-initializing. (They call `sud_SerialIsPortConnected()` to check if communication is possible.) The Windows built-in USB serial driver makes it look like the USB port is another COM port. This allows Windows applications written for a true serial port, such as HyperTerminal, to run unchanged. To them, it looks like the target board is connected via an RS232 port.

For the multi-port option, you need to use Windows XP SP3 or a later version of Windows. You also need to set `SU_COMPOSITE` and `SUD_SERIAL_SUPPORT_ACM` to 1. Set `SUD_SERIAL_MAX_NUM` to the number of port you need. Also make sure your USB device controller has enough endpoints (each serial port needs 3 endpoints). If you don't have enough endpoints, we provide a custom Windows driver which allows only two endpoints for each port.

The application interface is defined in **udserial.h**.

```
int      sud_SerialIsPortConnected(int port);
int      sud_SerialWriteData(int port, u8 *pBuf, uint len);
int      sud_SerialDataLen(int port);
int      sud_SerialReadData(int port, u8 *pBuf, uint len);
int      sud_SerialSetLineState(int port, uint iState);
int      sud_SerialGetLineState(int port, uint *piState);
int      sud_SerialGetLineCoding(int port, u32 *pdwDTERate, u8 *pbParityType, u8 *pbDataBits,
                                u8 *pbStopBits);
void     sud_SerialRegisterPortNotify(int port, PSEIRALFUNC handler);
```

int **sud\_SerialIsPortConnected** (int port)

**Summary** Indicates the host controller has configured this device and this port has been opened by a serial port application on the host machine.

**Details** Can be called any time after `sud_Initialize()` has been called.

**Parameters** port Serial port index. From 0 to `SUD_SERIAL_MAX_NUM - 1`.

**Returns** 1 Host has opened this serial port.  
0 Cable is not connected, host does not support this device, or port is not opened by the host's application.

**See Also** `sud_SerialRegisterPortNotify()`

int **sud\_SerialWriteData**(int port, u8 \*pBuf, uint len)

**Summary** Sends data to the USB host.

**Details** Can be called any time after `sud_Initialize()` has been called.

**Parameters** port Serial port index. From 0 to `SUD_SERIAL_MAX_NUM - 1`.  
pBuf Data buffer pointer.  
len Data buffer length.

**Returns** > 0 The data length (number of bytes) sent to the host.  
-1 Cable is not connected or the host does not support this device.

**See Also** `sud_SerialReadData()`

int **sud\_SerialDataLen** (int port)

**Summary** Gets the data length (number of bytes) in the receive buffer.

**Details** The serial function driver maintains an internal data buffer to store received data. This function returns the number of valid data bytes in this buffer. Can be called any time after `sud_Initialize()` has been called.

**Parameters** port Serial port index. From 0 to `SUD_SERIAL_MAX_NUM - 1`.

**Returns**     >= 0   Data length (number of bytes) in the receive buffer.  
              -1     Cable is not connected or the host does not support this device.

**See Also**     sud\_SerialReadData()

int **sud\_SerialReadData**(int port, u8 \*pBuf, uint len)

**Summary**     Reads serial data received from the USB host.

**Details**     This function retrieves the data stored in the internal receive buffer. Can be called any time after sud\_Initialize() has been called.

**Parameters**   port    Serial port index. From 0 to SUD\_SERIAL\_MAX\_NUM - 1.  
                 pBuf   Data buffer pointer.  
                 len     Data buffer length.

**Returns**     > 0     The data length (bytes) read from the internal buffer.  
              -1     Cable is not connected or the host does not support this device.

**See Also**     sud\_SerialDataLen()

int **sud\_SerialSetLineState**(int port, uint iState)

**Summary**     Notifies the USB host that the serial line state has changed.

**Details**     This is just like the serial line state change interrupt that indicates that some serial signal has been set or line error detected. This function should be used to relay this information to the USB host. Can be called any time after sud\_Initialize() has been called.

**Parameters**   port    Serial port index. From 0 to SUD\_SERIAL\_MAX\_NUM - 1.  
                 iState  New line state. Logical OR of the following constants:  
                          **SUD\_CDC\_LINE\_IN\_DCD**  
                          **SUD\_CDC\_LINE\_IN\_DSR**  
                          **SUD\_CDC\_LINE\_IN\_BRK**  
                          **SUD\_CDC\_LINE\_IN\_RI**  
                          **SUD\_CDC\_LINE\_IN\_FRAMING**  
                          **SUD\_CDC\_LINE\_IN\_PARITY**  
                          **SUD\_CDC\_LINE\_IN\_OVERRUN**

**Returns**     1     New state has been sent to the host.  
              0     Cable is not connected or the host does not support this device.

**See Also**      `sud_SerialGetLineState()`

`int sud_SerialGetLineState(int port, uint *piState)`

**Summary**      Gets the current line state set by the USB host.

**Details**      This should be called by the callback function that is called by `smxUSB` when the USB host changes the line state. Can also be called any time to get the current state after `sud_Initialize()` has been called.

**Parameters**    `port`      Serial port index. From 0 to `SUD_SERIAL_MAX_NUM - 1`.  
`piState`    New line state. Logical OR of the following constants:  
                  **SUD\_CDC\_LINE\_OUT\_DTR**  
                  **SUD\_CDC\_LINE\_OUT\_RTS**

**Returns**      1          Got the new line state.  
                  0          Cable is not connected or the host does not support this device.

**See Also**      `sud_SerialSetLineState()`,`sud_SerialRegisterPortNotify()`

`int sud_SerialGetLineCoding(int port, u32 *pdwDTERate, u8 *pbParityType,  
                                  u8 *pbDataBits, u8 *pbStopBits)`

**Summary**      Gets the current line coding set by the USB host.

**Details**      Normally called after you receive notification that line state has changed. The default value is set to baud rate 115200, no parity, 8 data bits and 1 stop bit. Can be called any time after `sud_Initialize()` has been called.

**Parameters**    `port`          Serial port index. From 0 to `SUD_SERIAL_MAX_NUM - 1`.  
`pdwDTERate`    Pointer to return baud rate.  
`pbParityType`    Pointer to return the parity type setting, with one of the following values:  
                  **SUD\_CDC\_PARITY\_NONE**  
                  **SUD\_CDC\_PARITY\_ODD**  
                  **SUD\_CDC\_PARITY\_EVEN**  
                  **SUD\_CDC\_PARITY\_MARK**  
                  **SUD\_CDC\_PARITY\_SPACE**  
`pbDataBits`     Pointer to return the number of data bits setting.  
`pbStopBits`     Pointer to return the number of stop bits setting, with one of the following values:

**SUD\_CDC\_STOP\_BITS\_1**  
**SUD\_CDC\_STOP\_BITS\_1\_5**  
**SUD\_CDC\_STOP\_BITS\_2**

**Returns**     1     New line coding returned.  
              0     Cable is not connected or the host does not support this device.

**See Also**     sud\_SerialGetLineState(),sud\_SerialRegisterPortNotify()

void **sud\_SerialRegisterPortNotify**(int port, PSEIRALFUNC handler)

**Summary**     Registers a notification function for the serial port.

**Details**     The handler function will be called after the serial function driver receives new data or the line state or line coding changes. Can be called any time after sud\_Initialize() has been called.

**Parameters**  port            Serial port index. From 0 to SUD\_SERIAL\_MAX\_NUM - 1.  
              handler        Function pointer to the notification function.  
                              typedef void (\* PSEIRALFUNC)(int port, int notification);

**Returns**     None

**See Also**     sud\_SerialGetLineState(),sud\_SerialGetLineCoding()

## 4.10 Video

The Video function driver makes your device look like a web camera to the USB host, such as Windows, Macintosh, or Linux. There is no need to install any driver or .inf file in Windows to support this device but you may need to implement the camera sensor driver by yourself, according to your system's hardware and software environment. You may also need to customize the configuration in udvideo.c for your real hardware features, such as the set of features your camera sensor will support.

See section 10.13 Video Camera Software for important notes.

The application interface is defined in **udvideo.h**.

```
int           sud_VideoIsConnected(int port);
int           sud_VideoSendVideoData(int port, u8 *pData, int iLen);
int           sud_VideoGetVideoData(int port, u8 *pData, int iLen);
void          sud_VideoRegisterNotify(int port, SUD_PVIDEOFUNC handler);
```

int **sud\_VideoIsConnected**(int port);

**Summary** Indicates whether the USB device is connected and the host controller has configured this device.

**Details** Can be called any time after `sud_Initialize()` has been called.

**Parameters** port Video device port index. For the current version, only pass 0.

**Returns** 1 Host has configured this device.  
0 Cable is not connected or the host does not support this device.

**See Also** `sud_VideoRegisterNotify()`

int **sud\_VideoSendVideoData**(int port, u8 \*pData, int iLen);

**Summary** Send one frame of video streaming data to the host.

**Details** When the Host is capturing the video, call this function to send one frame of video data to the host. Normally this function should be called after you get an `IN_START` event. Stop calling it after you get an `IN_STOP` event. The data in the buffer should be the data of one frame. For example, the format of the video streaming is 160x120 YUV422, calculated as  $160*120*2 = 38400$  bytes.

**Parameters** port Video device port index. For the current version, only pass 0.  
pData The data buffer pointer.  
iLen The length of the buffer.

**Returns** 0 Data have been sent to the host.  
-1 Error occurred when sending data.

**See Also** `sud_VideoRegisterNotify()`

int **sud\_VideoGetVideoData**(int port, u8 \*pData, int iLen);

**Summary** Get the received video streaming data (display).

**Details** When the Host is sending data to the display, call this function to get the received sound data from the host. Normally it should be called after you get an `ISOCDATAREADY` event.

**Parameters**    port    Video device port index. For the current version, only pass 0.  
                  pData    The data buffer pointer.  
                  iLen    The length of the buffer.

**Returns**        > 0    Received data length.  
                  -1    Error occurred when receiving data.

**See Also**        sud\_VideoRegisterNotify()

```
void sud_VideoRegisterNotify(int port, SUD_PVIDEOFUNC handler);
```

**Summary**        Set the notification handler for the video device.

**Details**        You must call this function to register an event notification handler so you can process events for the USB video device.

The notification is defined as:

**SUD\_VIDEO\_NOTIFY\_ISOCDATAREADY**  
**SUD\_VIDEO\_NOTIFY\_IN\_START\_STOP**  
**SUD\_VIDEO\_NOTIFY\_OUT\_START\_STOP**  
**SUD\_VIDEO\_NOTIFY\_INIT\_DEF\_SETTINGS**  
**SUD\_VIDEO\_NOTIFY\_INIT\_MIN\_SETTINGS**  
**SUD\_VIDEO\_NOTIFY\_INIT\_MAX\_SETTINGS**  
**SUD\_VIDEO\_NOTIFY\_INIT\_INFO\_SETTINGS**  
**SUD\_VIDEO\_NOTIFY\_SET\_SETTINGS**  
**SUD\_VIDEO\_NOTIFY\_SET\_PROBE\_FORMAT**  
**SUD\_VIDEO\_NOTIFY\_SET\_STILL\_FORMAT**

Notification handler is defined as

```
typedef void (* SUD_PVIDEOFUNC)(int port, int notification, u32 parameter);
```

Notification handler is called by SUD\_VIDEO\_NOTIFY\_ISOCDATAREADY when the device gets an incoming video frame from the host. This only happens when SUD\_VIDEO\_INCLUDE\_OUT is set to 1.

Notification handler is called by SUD\_VIDEO\_NOTIFY\_IN\_START\_STOP when the host starts/stops the video IN interface. Parameter is 1 for start, 0 for stop.

Notification handler is called by SUD\_VIDEO\_NOTIFY\_OUT\_START\_STOP when the host starts/stops the video OUT interface. Parameter is 1 for start, 0 for stop. This only happens when SUD\_VIDEO\_INCLUDE\_OUT is set to 1.

Notification handler is called by SUD\_VIDEO\_NOTIFY\_INIT\_DEF\_SETTINGS when the device needs to init the default settings of the video device. Parameter is a pointer to

structure SUD\_VIDEO\_SETTINGS. The application can fill the fields of that structure for the customized default settings.

Notification handler is called by SUD\_VIDEO\_NOTIFY\_INIT\_MIN\_SETTINGS when the device needs to init the minimum settings of the video device. Parameter is a pointer to structure SUD\_VIDEO\_MIN\_MAX. The application can fill the fields of that structure for the customized minimum settings.

Notification handler is called by SUD\_VIDEO\_NOTIFY\_INIT\_MAX\_SETTINGS when the device needs to init the maximum settings of the video device. Parameter is a pointer to structure SUD\_VIDEO\_MIN\_MAX. The application can fill the fields of that structure for the customized maximum settings.

Notification handler is called by SUD\_VIDEO\_NOTIFY\_INIT\_INFO\_SETTINGS when the device needs to init the information settings of the video device. Parameter is a pointer to structure SUD\_VIDEO\_INFO. The application can fill the fields of that structure for the customized information settings. For the meaning of the one byte information, check the USB Video Class Specification.

Notification handler is called by SUD\_VIDEO\_NOTIFY\_SET\_SETTINGS when the host tries to change the video device settings. Parameter is a pointer to structure SUD\_VIDEO\_SETTINGS. The application may need to compare these settings with the saved ones to find out which have changed and also set the corresponding hardware properly.

Notification handler is called by SUD\_VIDEO\_NOTIFY\_SET\_PROBE\_FORMAT when the host tries to change the video device probe (video capture) settings. Parameter is a pointer to structure SUD\_VIDEO\_FORMAT. The application may need to set the corresponding hardware properly.

Notification handler is called by SUD\_VIDEO\_NOTIFY\_SET\_STILL\_FORMAT when the host tries to change the video device still (photo capture) settings. Parameter is a pointer to structure SUD\_VIDEO\_FORMAT. The application may need to set the corresponding hardware properly.

<b>Parameters</b>	port	Video device port index. For the current version, only pass 0.
	handler	Function pointer for the notification handler.

<b>Returns</b>	none
----------------	------

<b>See Also</b>	none
-----------------	------



## 5. Writing a New Function Driver

**Note: Writing a new function driver is not easy, and may require assistance from Micro Digital. Please discuss this with Micro Digital before you decide to do it.**

This section describes how to write a new smxUSBBD function driver.

Function drivers are a separate sub-module of smxUSBBD. You do not need to know the details about the other parts, such as the device controller driver (6.1 Device Controller Operation Interface) and the smxUSBBD core layer.

Adding a new function driver is very simple for smxUSBBD. The only thing you need to do is implement a function driver interface and then register it with the smxUSBBD core layer in `udinit.c`, by calling `sud_RegisterFunction()`. The function should be called before the device controller driver register function.

We provide a working function driver template so you can study the source code and the following description to understand how it works. You can also use it as a startup point to create your own function driver. See `XUSBBD\Function\udftempl.*`.

### 5.1 Function Driver Interface

The function driver interface provides all the necessary information to the smxUSBBD core layer. This information includes a set of operation function pointers and device and configuration descriptor information. The interface is defined in `udfunc.h` as the following structure:

```
typedef struct SUD_Function_Driver_T
{
    const char *pFunctionName;
    const SUD_FUNCTION_OPER_T *pOperation;

    /* device and configuration information */
    SUD_DEVICE_INFO_T *pDeviceInfo;
    SUD_CONFIGURATION_INFO_T *pConfigurationInfo;

    /* device and configuration descriptor */
    SUD_DEVICE_DESC_T *pDeviceDescriptor;
    SUD_CONFIGURATION_HANDLE_T *pConfigurationHandle;

    /* number of configuration */
    uint iConfigurations;

#ifdef SUD_HIGH_SPEED
    SUD_DEV_QUALIFIER_DESC_T *pDevQualifierDescriptor;
#endif
}SUD_FUNCTION_DRIVER_T;
```

Only the first three fields and `iConfigurations` field need to be provided by your code. smxUSBBD will automatically fill in the other fields according to the information you provided.

## 5.2 Function Operation Interface

The function operation interface is a set of function pointers which will be called by the smxUSBD core layer when it receives requests from the device controller. These function pointers are defined in udfunc.h as:

```
typedef struct
{
    void (*DoEvent) (SUD_DEVICE_HANDLE_T *pDevice, uint event);
    int (*RequestSentDone) (SUD_REQUESTINFO_T *pRequestInfo, int status);
    int (*ReceiveRequest) (SUD_REQUESTINFO_T *pRequestInfo);
    int (*ReceiveSetup) (SUD_REQUESTINFO_T *pRequestInfo);
}SUD_FUNCTION_OPER_T;
```

Your new function driver must initialize these function pointers and create the functions, as documented below.

### 5.2.1 DoEvent()

When the USB state has changed, this function is called by the smxUSBD core layer. Four important events you need to handle are as follows:

- 1. SUD\_DEVICE\_CREATE**  
This event is called when smxUSBD is starting up. You can allocate private data structures in this event.
- 2. SUD\_DEVICE\_CONFIGURED**  
This event is called when the USB host has configured the device. After this, your function can begin to work.
- 3. SUD\_DEVICE\_DE\_CONFIGURED**  
This event is called when the USB host has de-configured the device or the cable has been unplugged from the port. The function driver should stop doing work after getting this event.
- 4. SUD\_DEVICE\_DESTROY**  
This event is called when smxUSBD is shutting down. You should free any private data allocated in the CREATE event.

### 5.2.2 RequestSentDone()

This is the callback function that is called when the request to the host has been sent. You can notify your application using this callback function. If status is 0, it means success; otherwise, an error has occurred.

You can access pRequestInfo->iActualLength to get the actual sent data size. Normally it should be the same as pRequestInfo->iDataLength which was set when you made the send request.

### 5.2.3 ReceiveRequest()

This is the callback function that is called when the host has sent your function driver some data. You can pass the received data to your application.

**pRequestInfo->pBuffer** points to the data buffer.  
**pRequestInfo->iActualLength** is the received data size.

## 5.2.4 ReceiveSetup()

This is the callback function that is called when the host has received a class/vendor request and it does not know how to handle it.

**pRequestInfo->RequestStru** holds the request header.  
**pRequestInfo->pBuffer** points to the received data buffer.  
**pRequestInfo->iActualLength** is the data size.

If you need to send data back to the host, just copy the data to **pRequestInfo->pBuffer**, set **pRequestInfo->iActualLength** to 0, and set **pRequestInfo->iDataLength** to the data size you want to send back. The core sends back the response data automatically.

If you need to send back a zero length Status package, set **pRequestInfo->iSendNotify** to 1.

This function returns -1 if there is an error; otherwise it returns 0.

## 5.3 Device Information

Device information tells the smxUSB core layer what your device function looks like. The smxUSB core layer generates the device descriptor according to the information provided here and sends it to the host, matching the device driver to the device. Device information is defined in `uddevice.h` as:

```
typedef struct
{
    u8 bDeviceClass;
    u8 bDeviceSubClass;
    u8 bDeviceProtocol;
    u8 bPadding;

    u16 idVendor;
    u16 idProduct;

    char *iManufacturer;
    char *iProduct;
    char *iSerialNumber;
}SUD_DEVICE_INFO_T;
```

*bDeviceClass*, *bDeviceSubClass*, and *bDeviceProtocol* describe your device's class information. See the USB defined class codes document to check which values to use. In some cases they can be set to 0 and the host will then check the configuration descriptor and/or interface descriptor to match the driver.

*idVendor* and *idProduct* is your device's Vendor ID and Product ID. Vendor ID is allocated by USB-IF. You can choose the product ID yourself.

*iManufacturer*, *iProduct*, and *iSerialNumber* describe your device. Any strings are acceptable.

## 5.4 Configuration Information

Configuration information tells the host how many configurations, interfaces, and endpoints are used in your device function and their attributes. This structure is defined in `uddevice.h` as:

```
typedef struct
{
    char *iConfiguration;
    u8 bmAttributes;
    u8 bMaxPower;
    u16 wPadding;
    uint bInterfaceNum;
    SUD_INTERFACE_INFO_T *pInterfaceInfo;
#ifdef SUD_COMPOSITE
    uint bADNum;
    SUD_IAD_INFO_T *pIADInfo;
#endif
}SUD_CONFIGURATION_INFO_T;
```

*iConfiguration* is the description for this configuration. Any string is acceptable.

*bmAttributes* is the attribute for this configuration. Valid values include: `SUD_BMATATTRIBUTE_RESERVED` and `SUD_BMATATTRIBUTE_SELF_POWERED`.

You can OR these together if the device is self powered.

*bMaxPower* is the maximum power your device will consume.

*bInterfaceNum* is the size of your interface information array.

*pInterfaceInfo* is the interface information array of your interface. Each element of the array is a structure pointer defined in `uddevice.h` as:

```
typedef struct
{
    char *iInterface;
    u8 bInterfaceClass;
    u8 bInterfaceSubClass;
    u8 bInterfaceProtocol;
    u8 bPadding;
    uint iAlternateNum;
    SUD_ALTERNATE_INFO_T *pAlternateInfo;
}SUD_INTERFACE_INFO_T;
```

*iInterface* is the description for this endpoint. Any string is acceptable.

*bInterfaceClass*, *bInterfaceSubClass*, and *bInterfaceProtocol* comprise the interface class code of this interface. See the USB defined class codes document to check which value you should use.

*iAlternateNum* is the size of your alternate information array.

*pAlternateInfo* is the alternate setting array for your interface. Each element of the array is a structure pointer defined in `uddevice.h` as:

```

typedef struct
{
    char *iInterface;
    uint bAlternateSetting;
    uint bClassNum;
    SUD_CLASS_INFO_T *pClassInfo;
    uint bEndpointNum;
    SUD_ENDPOINT_INFO_T *pEndpointInfo;
}SUD_ALTERNATE_INFO_T;

```

*iInterface* is the description of this alternate setting.

*bAlternateSetting* is this alternate setting's index.

*bClassNum* is the size of class information array. If you have no class specified information, set it to 0.

*pClassInfo* is the class information array pointer. If you have no class specified information, set it to NULL.

*bEndpointNum* is the size of your endpoint information array.

*pEndpointInfo* is the endpoint information array. Each element of the array is a structure pointer defined in `uddevice.h` as

```

typedef struct
{
    u8 bEndpointAddress;
    u8 bmAttributes;
    u16 wMaxPacketSize;
    u8 bInterval;
    u8 bDirection;
}SUD_ENDPOINT_INFO_T;

```

*bEndpointAddress* is your endpoint's logical address. Numbering should start at 1.

*bmAttributes* is the attributes of your endpoint. Valid combinations include `SUD_ENDPOINT_XFER_CONTROL`, `SUD_ENDPOINT_XFER_ISOC`, `SUD_ENDPOINT_XFER_BULK`, and `SUD_ENDPOINT_XFER_INT`.

*wMaxPacketSize* is the max packet size of this endpoint. Check your device controller to determine the value. For a USB 1.1 device, the BULK endpoint's packet size is 64.

*bInterval* is the interrupt transfer interval, in milliseconds. For BULK and CONTROL endpoints, sent it to 0.

*bDirection* is this endpoint's direction. `SUD_DIR_IN` means IN endpoint; `SUD_DIR_OUT` means OUT endpoint.

*dwTransferSize* is the maximum data transfer size. It depends on your function driver's internal data buffer size.

## 5.5 Send Request

If your device has an IN endpoint, you may need to send data to the host. To do that, you should:

1. Call `sud_AllocateRequestInfo()` to allocate a request info structure.
2. Copy your data to `pRequestInfo->pBuffer` and set `pRequestInfo->iDataLength` to the buffer size value. Set `pRequestInfo->iActualLength` to 0.
3. Call `sud_SendRequest()` to send out the request. The callback function defined in `RequestSentDone()` will be called after `sud_SendRequest()` completes.

## 5.6 Select Endpoint Number

There are no software device driver restrictions for endpoint number definitions. You can select any number less than 16 for your endpoint number. However some USB device controllers use fixed endpoint numbers for certain types of endpoints. For example, with the LPC3180 USB device controller, endpoint number 1 can only be Interrupt, and endpoint number 2 can only be Bulk. In this case, you need to change the value of macros `SUD_XXX_IN_EP` and/or `SUD_XXX_OUT_EP`, defined in `ucfg.h`, to meet the requirements of the device controller.

## 6. Writing a New Device Controller Driver

**Note: Writing a new device controller driver is difficult and may require a lot of assistance from Micro Digital. It is not recommended that you write one yourself. Please discuss this with Micro Digital before you decide to do it.**

This section describes how to write a new smxUSB D device controller driver.

The USB device controller drivers are a sub-module of smxUSB D. You do not need to know the details about the other parts such as the function drivers (5.1 Function Driver Interface) and the smxUSB D core layer.

Adding a new USB device driver is very simple for smxUSB D. The only thing you need to do is implement a Device Controller operation interface and then register it with the smxUSB D core layer in udinit.c, by calling sud\_DCRegister(). This function should be called after any other smxUSB D initialization function calls.

We provide a DCD template in the DCD directory to use as a starting point. Check all comments marked “TODO” and implement those sections according to the instructions.

### 6.1 Device Controller Operation Interface

The device controller operation interface is defined in uddevice.h as:

```
typedef struct
{
    int (*DCInit) (void);
    int (*DCRelease) (void);
    int (*DCStartXmit) (SUD_ENDPOINT_HANDLE *pEndpoint);
    int (*DCStallEndpoint) (uint ep, uint stall);
    int (*DCSetAddress) (u8 address);
    int (*DCEndpointHalted)(uint ep);
    int (*DCSetupOneEndpoint)(SUD_DEVICE_HANDLE_T *pDevice, uint ep,
                              SUD_ENDPOINT_HANDLE *pEndpoint);
    int (*DCSetupEndpointDone)(SUD_DEVICE_HANDLE_T *pDevice);
    int (*DCDisableEndpoint)(uint ep);
    int (*DCConnect)(void);
    int (*DCDisconnect)(void);
    int (*DCEnable)(SUD_DEVICE_HANDLE_T *pDevice);
    int (*DCDisable)(void);
    int (*DCEnableInt)(void);
    int (*DCDisableInt)(void);
    int (*DCEP0PacketSize)(void);
    int (*DCMaxEndpoints)(void);
    int (*DCStartup)(SUD_DEVICE_HANDLE_T *pDevice);
    int (*DCMapEndpoint)(uint ep);
    int (*DCFrameNum)(void);
}SUD_DC_OPERATION_T;
```

Your new device controller must initialize these device controller pointers and create the functions, as documented below.

### **6.1.1 DCInit()**

The smxUSBD core layer calls this function first to do any necessary DC initialization. Returns 0 for success; -1 for error.

### **6.1.2 DCRelease()**

The smxUSBD core layer calls this function before it shuts down, to do any necessary DC cleanup. Returns 0 for success; -1 for error.

### **6.1.3 DCStartXmit()**

The smxUSBD core layer calls this function when the application wants to send some data to the host by an IN endpoint. The request data structure is in pEndpoint->pCurRequestInfo. You can get the data through this pointer. The data buffer is pRequestInfo->pBuffer. Data size is pRequestInfo->iDataLength. The size of data already sent is pRequestInfo->iActualLength. Return 0 for success; -1 for error.

### **6.1.4 DCStallEndpoint()**

The smxUSBD core layer calls this function when it needs to stall an endpoint. Normally you need to write a DC register to force stalling an endpoint. Return 0 for success; -1 for error.

### **6.1.5 DCSetAddress()**

The smxUSBD core layer calls this function after it gets the SET ADDRESS request. Setting the address to 0 means the device is de-configured. Normally you should write a DC register to inform DC the new address. Return 0 for success; -1 for error.

### **6.1.6 DCEndpointHalted()**

The smxUSBD core layer calls this function to determine if the endpoint is stalled or not. Normally you should read a DC register to get the information. Return 1 if it is halted (stalled); return 0 if not stalled.

### **6.1.7 DCSetupOneEndpoint()**

The smxUSBD core layer calls this function to set up each endpoint's properties such as the type, direction, and max packet size. You may also need to initialize the FIFO associated with this endpoint and enable the interrupt for this endpoint. Return 0 for success; -1 for error.

### **6.1.8 DCSetupEndpointDone()**

The smxUSBD core layer calls this function after all the endpoints have been set up. Normally the device is now in the configured state and can begin to work. You can do any job required by the device controller in the configured state. Return 0 for success; -1 for error.



### **6.1.9 DCDisableEndpoint()**

The smxUSB core layer calls this function to disable an endpoint. Return 0 for success; -1 for error.

### **6.1.10 DCConnect()**

The smxUSB core layer calls this function after it has connected to the USB and is ready to be enumerated by the host. Return 0 for success; -1 for error.

### **6.1.11 DCDisconnect()**

The smxUSB core layer calls this function after it has been disconnected from the USB. Return 0 for success; -1 for error.

### **6.1.12 DCEnable()**

The smxUSB core layer calls this function to enable the device controller module. Return 0 for success; -1 for error.

### **6.1.13 DCDisable()**

The smxUSB core layer calls this function to disable the device controller module. Return 0 for success; -1 for error.

### **6.1.14 DCEnableInt()**

The smxUSB core layer calls this function to enable the device controller module interrupt. Return 0 for success; -1 for error.

### **6.1.15 DCDisableInt()**

The smxUSB core layer calls this function to disable the device controller module interrupt. Return 0 for success; -1 for error.

### **6.1.16 DCEP0PacketSize()**

The smxUSB core layer calls this function to get the default control endpoint (EP0) max packet size.

### **6.1.17 DCMaxEndpoints()**

The smxUSB core layer calls this function to get the max number of logical endpoints (including ep0).

### **6.1.18 DCStartup()**

The smxUSB core layer calls this function to start the device controller. Normally you should call the following two functions and then set up the default controller endpoint.

```
sud_DoDeviceEvent (pDevice, SUD_DEVICE_INIT);  
sud_DoDeviceEvent (pDevice, SUD_DEVICE_CREATE);
```

You may also need to enable the corresponding interrupt for your device controller so it can get the correct interrupt, and you may need to allocate a global EP0 Request data to handle the EP0 request and data.

### **6.1.19 DCMapEndpoint()**

The smxUSB core layer calls this function to map a logical endpoint to the device controller's physical endpoint.

### **6.1.20 DCFrameNum()**

The smxUSB function driver, such as video, may call this function to get the current frame number.

## **6.2 Handle Device Controller Interrupt**

The interrupt handler is the most important part of the device controller driver. Different device controllers have different implementations, so you should study the device controller's data sheet or sample code to understand the details, and then make it work for the MDI interface. The following interrupts need to be handled correctly to make the device controller work. Some device controllers may not have all these interrupts.

### **6.2.1 BUS Reset Done**

The device controller will generate this interrupt when it detects the hub's reset signal has completed. You should setup and enable endpoint 0, the default control endpoint, to be ready to get the enumeration request from the host. Normally you should call

```
    sud_DoDeviceEvent (SUD_DEVICE_HUB_CONFIGURED);  
    sud_DoDeviceEvent (SUD_DEVICE_RESET);
```

in this interrupt service routine.

### **6.2.2 Suspend**

The device controller will generate this interrupt if the device has been detached from the USB or the host suspends the bus. Normally you should call

```
    sud_DoDeviceEvent (pDevice, SUD_DEVICE_BUS_INACTIVE);
```

to force the device stack to the de-configured state.

### 6.2.3 Resume

The device controller will generate this interrupt if the host resumes the bus. Normally you should call

```
sud_DoDeviceEvent (pDevice, SUD_DEVICE_BUS_ACTIVITY);
```

### 6.2.4 SETUP

The device controller will interrupt if it receives a SETUP packet. You need to:

1. Read the request header into EP0Request.RequestStru.
2. Wait and receive the data into EP0Request.pBuffer and set up the data length to EP0Request.iActualLength and iDataLength if the setup request has a data phase.
3. When all data is received, call function sud\_ProcessSetupPacket(&EP0Request) to pass the data to the smxUSB core layer. If this function returns 0, send the data in EP0Request.pBuffer and EP0Request.iDataLength to the host. If EP0Request.iSendNotify is 1, send a zero length status packet. If sud\_ProcessSetupPacket() returns error, stall endpoint 0.
4. Call function sud\_RequestSendDone (&EP0RequestInfo, SUD\_SEND\_FINISHED\_OK) when all the data has been sent out.

### 6.2.5 EPx

The device controller will interrupt if it receives a data packet of endpoint x. You need to:

1. For an IN endpoint, check if all data has been sent out. If not, continue sending data in pEndpoint->pCurRequest. Once all the data has been sent, call function sud\_TransmitCompleted() to pass the information to the smxUSB core layer. After that, check if pEndpoint->pCurRequest is empty, to send the next request in the same endpoint. If there is still some data left in the pRequestInfo->pBuffer then update pRequestInfo->iActualLength and wait for the next interrupt.
2. For an OUT endpoint, check if you received a short packet or zero length packet, which means the data transfer has completed. If yes, call function sud\_RecvCompleted() to pass the data to the smxUSB core layer. Otherwise, copy the data to pRequestInfo->pBuffer, update the buffer size of pRequestInfo->iActualLength, and wait for the next interrupt.

## 6.3 Logical Endpoint Number and Physical Endpoint Number

A logical endpoint number is used by the smxUSB core layer in the endpoint descriptor. But the device controller may map the logical endpoint number to another physical endpoint number if its endpoints have a fixed transfer mode or direction. For example, in the LPC3180 USB device controller, Logical Endpoint 1 is an interrupt endpoint and mapped to two physical endpoints 2 and 3 for OUT and IN transfer. The smxUSB core layer uses physical endpoint numbers in all Device Controller Driver functions, so DCMapEndpoint() is called by the core layer to convert the logical endpoint used within

the core layer to the physical endpoint when it needs to call a DCD interface function. You must implement this mapping function carefully if your device controller maps endpoints like this.

## 7. Composite Device

A device that has multiple interfaces for multiple functions in one configuration that are active at the same time while using a single device controller chip is a composite device. An alternate way to have multiple devices is to use multiple device controllers connected by an internal hub.

smxUSBBD supports creating a composite device by providing a general composite device framework and by reusing existing function driver code. So from the application's point of view, all existing function driver APIs can still be used, and the application does not know whether it is implementing a composite device or multiple devices.

### 7.1 Composite Device Framework

The smxUSBBD composite device framework combines all the existing functions together to generate a new composite device and registers it with the core layer. The framework defines a new device descriptor and configuration descriptor that is special for a composite device. It re-uses the interface information and function operation pointer to do the real, function related job, so the composite device framework does not need to know the details of its sub-function, such as the SCSI commands of the mass storage device or the line coding of the serial port.

### 7.2 Adding an Existing Function to the Composite Device Framework

If you have already written a function driver, such as Audio, and you want to add it to the composite device framework, define SUD\_COMPOSITE in the file udcfg.h and do the following:

#### 7.2.1. Add Two New Functions in the Function Driver

```
#if SUD_COMPOSITE
void *sud_AudioGetOps();
void *sud_AudioGetInterface();
#endif

#if SUD_COMPOSITE
void *sud_AudioGetOps()
{
    return &AudioOps;
}

void *sud_AudioGetInterface()
{
    return &AudioInterface[0];
}
#endif
```

These two functions return the function operation pointer and interface information so the composite device framework can use it to handle the request and data transfer.

## 7.2.2. Modify the Init and Release Functions

```
int sud_AudioInit(void)
{
    #if !SUD_COMPOSITE
        if (sud_RegisterFunction (&AudioFuncDriver))
        {
            return -1;
        }
    #endif

    return 0;
}

void sud_AudioRelease(void)
{
    #if !SUD_COMPOSITE
        sud_DeregisterFunction (&AudioFuncDriver);
    #endif
}
```

These modifications will not register the audio function driver with the device stack core layer; the composite device will register itself.

## 7.2.3. Add the New Device to the Composite Device (udcompos.c)

1. Add the Audio device interface to the composite device in `sud_CompositeInit()` and the `CompositeInterface` array.
2. Add the Audio device event handler to `CompositeEventHandler()`.
3. Add the Audio request sent done handler to `CompositeRequestSent()`.
4. Add the Audio receive request handler to `CompositeReceiveRequest()`.
5. Add the Audio class request handler to `CompositeRecvSetup()`.

## 7.2.4. Adjust Endpoint Number for Different Functions

Make sure all the endpoint numbers for different functions in your composite device are unique. Your USB device controller may have some restrictions for the endpoint transfer type and direction. You may need to study the data sheet of your device controller or contact Micro Digital for help.

## 7.3 Composite Device Product and Interface IDs

It is necessary to define a unique Product ID for your particular combination of functions. If you later add or remove a function, you need to assign a new Product ID. Otherwise, it will confuse the host (e.g. Windows), and you will need to uninstall the old driver on the host. `udcfg.h` defines IDs using a series of conditionals for different functions (e.g. serial, audio, video), but that cannot handle every case. You should clean out unneeded lines from that section and set the ID as desired.

For Windows .inf files, it is necessary to specify the interface ID, as in the MI\_00 in the following:

```
%SERIAL%=SerialInstall, USB\VID_04CC&PID_0010&MI_00
```

The number after MI\_ is the index into CompositeInterface[] in udcompos.c for the corresponding function. It is necessary to adjust the numbers in the .inf files we provide, to match which lines are present in that table.

## 7.4 Composite Device Limitations

1. If the function within the composite device has only one interface, then most host stacks such as Windows, Linux, and our smxUSBH can support it without any problem. But if the function needs two or more interfaces, such as Ethernet over USB, Serial with ACM support, or Serial multi-port, you need the IAD (Interface Association Descriptor) to tell the host which interfaces should be associated together to format a function. IAD is only supported for Windows XP SP3 and later versions such as Vista and Windows 7/8. (For older versions of Windows, to use serial in a composite device, set SUD\_SERIAL\_SUPPORT\_ACM to 0.)

We collected the following information from the Internet about this problem:

Microsoft WHDC has some information for composite device support:

([www.microsoft.com/whdc/archive/IAD.msp](http://www.microsoft.com/whdc/archive/IAD.msp))

### *Windows Operating System Support*

*Microsoft is currently working with IHVs to develop devices that support IAD. A beta version of USBCCGP.sys that supports IAD is currently available for testing on Windows XP (SP1) systems.*

*Microsoft will support IAD in the Windows Vista operating system. Windows XP (SP2) might support IAD.*

*To avoid compatibility concerns, you should work closely with your Windows liaison to ensure that IAD is implemented correctly in your device. Microsoft considers IAD to be most applicable to composite devices that contain the following device classes:*

- *USB Video Class Specification (Class Code - 0x0E)*
- *USB Audio Class Specification (Class Code - 0x01)*
- *USB Bluetooth Class Specification (Class Code - 0xE0)*

2. Ethernet over USB function driver (SUD\_NET) needs two configurations if both RNDIS and CDC-ECM/NCM are enabled. smxUSB composite device framework only work on one configuration so you cannot enable SUD\_COMPOSITE and both RNDIS and CDC-ECM/NCM.

## 8. Hardware Porting Notes

The hardware porting layer is in `udport.h` and `udport.c`. These files contain definitions, macros, and functions to port `smxUSBD` to particular target hardware.

### 8.1 `udport.h`

`smxBase` handles most of the hardware porting defines. Below are the ones that are `smxUSBD`-specific.

1. `SUD_BUS_INVERTED`: Set this to 1 if you have a big endian CPU and you inverted the connection of the device controller to the data bus. (If you are designing your own hardware, we recommend you do this for better performance, so there is no need to invert data i/o to device controller registers in `smxUSBD`.)
2. Driver `BASE` and `IRQ` settings: Set these to the proper addresses for your device controller.
3. Settings for specific controllers: See comments in this file.

### 8.2 `udport.c`

**Some of the functions in `udport.c` may need to be adapted for your target.**

#### `sud_HdwInit()`

This function is called first when initializing the `smxUSBD` device stack. It does the following:

Initializes the hardware platform's USB subsystem. For example, it enables the USB device controller, and sets up the clock.

Determines the Device Controller's I/O base, memory base, and `IRQ` number.

Initializes other hardware required by `smxUSBD`. For example, it opens a serial port and sets up the parameters for the `sud_DebugL()` function to output debug information.

#### `sud_HdwRelease()`

Disables the USB subsystem of the hardware.

#### `sud_GetSerialNum()`

Get the unique serial number of this USB device. Some USB hosts, such as Windows, use Vendor ID/Product ID/Serial Number to identify the USB device. If you may plug in multiple devices (such as USB serial) into one such USB host, a unique serial number is required for multiple devices to work properly, simultaneously. Sometimes you can use the MAC address of your device or OTP bytes of your flash memory as a serial number for the USB.



Returning a NULL pointer from this function will cause smxUSBBD to use the default serial number defined in udcfg.h.

#### **sud\_GetISP1181Base(), sud\_GetISP1181Interrupt()**

These functions return the NXP ISP1181-compatible device controller I/O base address and IRQ found in sud\_HdwInit(). **You should not need to change the default implementation.**

#### **sud\_GetISP1181IntSetting()**

This function returns your system's interrupt setting such as whether it is level/edge triggered and output polarity. **Change the value according to your hardware's implementation.**

#### **sud\_ISP1181Read32(), sud\_ISP1181Read16(), sud\_ISP1181Write32(), sud\_ISP1181Write16()**

These functions are used when the smxUSBBD NXP ISP1181 device controller driver accesses I/O registers. **You may need to tune the implementation of these functions to meet the timing requirement of ISP1181 according to your hardware implementation.**

#### **sud\_ISP1181ReadBuf(), sud\_ISP1181WriteBuf()**

These functions are used when the smxUSBBD NXP ISP1181 device controller driver accesses the FIFO buffer. **You may need to tune the implementation of these functions to meet the timing requirement of ISP1181 according to your hardware implementation.**

#### **sud\_GetISP158XBase(), sud\_GetISP158XInterrupt()**

These functions only return the NXP ISP158x-compatible device controller I/O base address and IRQ found in sud\_HdwInit(). **You should not need to change the default implementation.**

#### **sud\_GetISP158XIntSetting()**

This function will return your system's interrupt setting such as whether it is level/edge triggered and output polarity. **You should change the value according to your hardware's implementation.**

#### **sud\_ISP158XRead32(), sud\_ISP158XRead16(), sud\_ISP158XWrite32(), sud\_ISP158XWrite16(), sud\_ISP158XSet16(), sud\_ISP158XClear16()**

These functions are used when the smxUSBBD NXP ISP158x device controller driver accesses I/O registers. **You may need to tune the implementation of these functions to meet the timing requirement of ISP158x according to your hardware implementation.**

#### **ISP1761 has an erratum about the timing:**

If there is a RD\_N pulse, occurring in less than 60 ns after WR\_N to the Endpoint Index or Control Function register, data corruption occurs, irrespective of the CS\_N signal. The data corruption problem occurs intermittently.

So if you are using a fast processor, **you may need to add the following delay function to `sud_ISP158XWrite16()`** to avoid this problem.

```
sud_IODelay(5);
```

**`sud_ISP158XReadBuf()`, `sud_ISP158XWriteBuf()`**

These functions are used when the smxUSBD NXP ISP158x device controller driver accesses the FIFO buffer. **You may need to tune the implementation of these functions to meet the timing requirement of ISP158x according to your hardware implementation.**

### **8.3 DMA Transfer**

Most smxUSBD external Device Controller Drivers do not use DMA transfer. For ISP1181 and ISP1581 external USB device controller chips, DMA transfer of data from the microprocessor to those chips' internal FIFO is an option. DMA is highly dependent on the microprocessor so it is hard to write general, efficient, and portable DMA code. Contact Micro Digital for more information if DMA is necessary.

## 9. Windows Drivers / Application

Most smxUSB device functions work with standard Windows drivers. This section documents Windows drivers and applications we have developed for special functions.

These drivers are tested on Windows XP 32-bit, Windows 7 32-bit and 64-bit, and Windows 8 64-bit. For Windows 8, use the Windows 7 drivers.

For Windows 64-bit, you need to sign the driver package. Check the following URLs for detailed information about this.

[http://msdn.microsoft.com/en-us/library/ff552289\(v=vs.85\)](http://msdn.microsoft.com/en-us/library/ff552289(v=vs.85))

During development, you can test-sign the driver using the information provided at the following URL. Our testing is based on this approach.

<http://msdn.microsoft.com/en-us/library/ff546236.aspx>

Alternatively, for Windows 7 and Windows 8 64-bit, it is possible to temporarily disable driver signature enforcement when you boot up Windows, so you can try the smxUSB demo without having to go through the steps above. For Windows 7, press F8 during the Windows boot up procedure and select the option “Disable Driver Signature Enforcement”. For Windows 8, go to PC settings->General->Advanced startup->Start Now, wait a few seconds then Troubleshoot->Advanced options->Startup Settings->Restart. After the PC restarts, use 7 or F7 to disable driver signature enforcement. You need to do this every time you install or use the unsigned 64-bit driver. Each time you reboot your PC, if you don’t select this option, the unsigned driver will not work at all even if you already installed it by disabling the driver signature enforcement. Windows may report the following message:

*Windows cannot verify the digital signature for the drivers required for this device. A recent hardware or software change might have installed a file that is signed incorrectly or damaged, or that might be malicious software from an unknown source. (Code 52)*

After you have run the smxUSB demo, you should study the information at the URLs above to learn how to sign the driver to permanently avoid the problem.

Tip: During development, if you did not install the correct serial port driver, next time Windows may use another COM port number. For example, if COM8 installation failed, next time Windows will try to use COM9. To reclaim those unused COM ports, you need to change the ComDB key value in the Windows registry HKEY\_LOCAL\_MACHINE\SYSTEM\CurrentControlSet\Control\COM Name Arbiter. ComDB key is a bitmask of which COM ports are in use (8 ports per byte, starting from COM1). For example, to reclaim COM8 you need to clear the highest bit of the first byte of that key.

Tip: During development, if there was a problem installing a driver, and you need to completely uninstall it, delete the driver package using the pnputil.exe utility. To use this utility, go to Start->Accessories->Command Prompt, right click it and select “Run as administrator”. To list all the

OEM drivers installed, use pnputil.exe -e. check the output messages to find out which oem\*.inf is the driver you want to delete. For example,

```
Microsoft Windows [Version 6.1.7601]
Copyright (c) 2009 Microsoft Corporation. All rights reserved.
```

```
C:\Windows\system32>pnputil -e
Microsoft PnP Utility
```

```
Published name :      oem5.inf
Driver package provider :  Micro Digital Inc
Class :             Ports (COM & LPT)
Driver date and version :  02/12/2013 1.0.2
Signer name :
```

oem5.inf is the Micro Digital Inc multiple port USB-Serial adapter driver. To delete it use the following command:

```
C:\Windows\system32>pnputil -f -d oem5.inf
```

```
Microsoft PnP Utility
```

```
Driver package deleted successfully.
```

## 9.1 Multiple Port Serial Device (or Single Port Limited Endpoints)

A multiple port serial device can be supported by the built-in Windows driver usbser.sys, for Windows XP SP3 and later versions of Windows, if you have enough endpoints. Each port needs two BULK endpoints and one interrupt endpoint. For this case, use XUSBDFunction\Serial\mdiserialm.inf.

If you don't have enough endpoints, you can use the WDM driver we developed, named usbserm.sys, for multiple port serial devices, using only two endpoints per port. The device can be single function device or composite device. To install the Windows driver, you need to use XUSBDFunction\SerialM\mdiserialm.inf for a single function device or XUSBDFunction\SerialM\mdiserialmc.inf for a composite device. If your serial device has only one port but you don't want to use an INT endpoint, you still need to use this Windows driver and XUSBDFunction\SerialM\mdiserials.inf, and you need to set SUD\_SERIAL\_USE\_INT\_ENDPOINT to 0 in udcfg.h. This conserves endpoints, allowing you to support more serial channels or more device functions. However for Windows 7 and newer, it requires the driver to be signed or else Windows will not install it. Refer to Appendix H. Host OS Certification for more information. This Windows driver is provided only with the smxUSBDFunction multiple port serial driver.

This Windows Driver is based on Windows DDK serial sample code, and we modified it to make it work for USB device.

## 9.2 Device Firmware Upgrade (DFU) Device

Windows has no built-in DFU class driver. MDI provides a DFU Windows driver and utility in the BIN directory. `usbdfu.exe` is a command-line version of the utility, and `DFUHost.exe` is a GUI version. Both run on Windows. Alternatively, you can use another driver, such as open source driver `libUSB` and `dfu-util.exe`.

`usbdfu.h` and `usbdfu.c` are the Windows DFU class driver, running in the user space. You can build this driver with your own application if you don't want to use our utilities. `usbdfu.sys` is still needed because this kernel mode driver provides the basic USB functions to the DFU class driver in user space.

`usbdfu_cmd.c` has the main function of the command line utility. It is a good starting point if you want to integrate the USB DFU function into your application. To build the command line utility, you may need the Windows Driver Kit package. The WDK can be downloaded from Microsoft web site, for free.

`DFUHost` is a Visual C++ and GUI application. It basically does the same thing as the `usbdfu.exe` but with a graphical user interface.

DFU Class Driver API includes:

```
int    UsbDfuOpen(HWND hWnd);
int    UsbDfuClose(void);
int    UsbDfuGetSegNum(void);
int    UsbDfuGetSegName(int index, char *pSegName, ULONG size);
int    UsbDfuCheckSuffix(const char *pDFUFile, USHORT idVendor, USHORT idProduct);
int    UsbDfuAddSuffix(const char *pFirmwareFile, const char *pDFUFile, USHORT idVendor,
                      USHORT idProduct, USHORT wDeviceVersion);
int    UsbDfuDo(const char *pDFUFile, BOOLEAN bDownload, int iSegment);
```

int **UsbDfuOpen** (HWND hWnd)

**Summary** Open the DFU device.

**Details** Call this function to open the DFU device. Also pass the handle of the windows if you need to get the notification messages.

**Parameters** `hWnd` The handle of the windows which will receive the notifications from the DFU class driver.

**Returns** 0 Found DFU device and opened it.  
 -1 Could not find or open the DFU device

**See Also** **UsbDfuClose()**

int **UsbDfuClose** (void)

**Summary** Close the open DFU device.

**Details** After you are done uploading or downloading the firmware, call this function to release the device.

**Parameters** None

**Returns** 0 Device closed.  
-1 Device is not opened.

**See Also** **UsbDfuOpen()**

int **UsbDfuGetSegNum** (void)

**Summary** Get memory segment number.

**Details** The device may provide multiple memory segments, such as NOR flash for code and SPI flash for the configuration. This function lets you retrieve the number of the memory segment. You should call **UsbDfuOpen()** first.

**Parameters** None

**Returns** >0 Memory segment number.  
-1 Device is not opened.

**See Also** **UsbDfuGetSegName ()**

int **UsbDfuGetSegName** (int index, char \*pSegName, ULONG size)

**Summary** Get the memory segment name.

**Details** The device may provide a string to display the name of each memory segment. Call this function to get the name by the index. You should call **UsbDfuOpen()** first.

**Parameters** index Memory segment index, from 0 to **UsbDfuGetSegNum()** – 1  
pSegName String buffer for the name.  
size Size of the string buffer.

**Returns** >=0 String buffer length.  
-1 Device is not opened.

**See Also**      **UsbDfuGetSegNum()**

int **UsbDfuCheckSuffix**(const char \*pDFUFile, USHORT idVendor, USHORT idProduct)

**Summary**      Check if the DFU file contains a valid suffix.

**Details**      The suffix is used to check the consistency of the firmware, such as the vendor ID, product ID, and CRC. You can call this function without even opening the device, to check that the firmware is valid.

**Parameters**    pDFUFile      DFU file full path name.  
idVendor      Desired Vendor ID. Pass 0xFFFF if don't care.  
idProduct     Desired Product ID. Pass 0xFFFF if don't care.

**Returns**      0      Suffix is valid.  
-1      Could not open file or suffix is invalid.

**See Also**      **UsbDfuAddSuffix ()**

int **UsbDfuAddSuffix** (const char \*pFirmwareFile, const char \*pDFUFile, USHORT idVendor, USHORT idProduct, USHORT wDeviceVersion)

**Summary**      Generate a DFU file with suffix from the raw firmware file.

**Details**      The suffix is used to check the consistency of the firmware, such as the vendor ID, product ID, and CRC. You can call this function to generate a DFU file that contains the original firmware data and suffix. You don't need to open the device first.

**Parameters**    pFirmwareFile      Raw firmware file full path name.  
pDFUFile          Generated DFU file with suffix.  
idVendor          Vendor ID. Pass 0xFFFF if don't care.  
idProduct         Product ID. Pass 0xFFFF if don't care.  
wDeviceVersion    Version of this firmware.

**Returns**      0      DFU file is generated.  
-1      File generation failed.

**See Also**      **UsbDfuCheckSuffix ()**

int **UsbDfuDo** (const char \*pDFUFile, BOOLEAN bDownload, int iSegment)

**Summary** Download/Upload the firmware to the specific memory segment.

**Details** Call this function to download or upload the firmware to the device's memory segment. Downloading/uploading firmware takes some time so you may call this function in a separate thread. You need to call UsbDfuOpen() first.

**Parameters**

pDFUFile	DFU file to download or upload.
bDownload	TRUE for download, FALSE for upload.
iSegment	Memory segment index, from 0 to UsbDfuGetSegNum() - 1

**Returns**

0	Operation success.
-1	Operation failed.

**See Also** **UsbDfuOpen ()**

### 9.3 HID Communication

HID device does not need any special driver, not even an .inf file on Windows, but you need to write a special Windows application to handle the communication. Check usbhid.exe source code in BIN\WinDrv\usbhid\exe for the details. It needs the Windows DDK to build but you can modify the code to build it without DDK if you know the HID device's GUID or you know the device's full name.

To open the HID device:

```
GUID GUID_USB_HID;
HidD_GetHidGuid(&GUID_USB_HID);

if ( !GetUsbDeviceFileName((LPGUID) &GUID_USB_HID, deviceName, MAX_LENGTH) )
{
    printf("Failed to GetUsbDeviceFileName err - %d\n", GetLastError());
    return INVALID_HANDLE_VALUE;
}
HidDevice = CreateFile(deviceName,
    GENERIC_WRITE | GENERIC_READ,
    FILE_SHARE_WRITE | FILE_SHARE_READ,
    NULL,
    OPEN_EXISTING,
    0,
    NULL);
```

To write and read data:

You can use Win32 API WriteFile() and ReadFile() to send and receive data to and from the device. **You need to add one extra byte before your payload data.** That is for the report ID of the HID device. By the default the report ID starts from 1.



```
Buffer[0] = 1;
memcpy(Buffer + 1, pPayload, 63);
WriteFile (HidDevice, Buffer, 64, &bytesWritten, NULL);
...
memset(Buffer, 0, 64);
Buffer[0] = 1;
ReadFile (HidDevice, Buffer, 64, &bytesRead, NULL);
if(bytesRead == 64)
    memcpy(pPayload, Buffer + 1, 63);
```

## 10. Application Notes

### 10.1 Flow Control of the Serial Port

The serial port function uses an internal ring buffer to cache received data. However, because a USB transfer is faster than a normal RS232 port, if the USB host is sending data too fast and your firmware is a little slow to process the received data, you need to do some kind of flow control to prevent losing data. Basically, you need to handle the received data this way:

1. Allocate an application buffer which is bigger than the serial port internal ring buffer
2. Create a separate task to process the received data. After the code get SUD\_CDC\_NOTIFY\_DATA\_READY event in the Notification function, wake up that task and in that task, copy the received data from the internal ring buffer to the application buffer by calling `sud_SerialDataLen()` and `sud_SerialDataRead()`. **Do not do any lengthy processing job within the notification function because it will block the whole USB stack.**
3. If you find the application buffer is almost full, for example 80%, then call the following serial port API to disable RTS, so the USB host will stop sending more data.  
`sud_SerialSetLineState(0, SUD_CDC_LINE_OUT_DTR);`
4. After you process the received data and you get more space in your application buffer, for example, 80% empty, call the following API to enable RTS, so the USB host will resume the data sending.  
`sud_SerialSetLineState(0, SUD_CDC_LINE_OUT_DTR|SUD_CDC_LINE_OUT_RTS);`

### 10.2 Mass Storage and File System Share the Same Media

It is possible to have a file system in your target to share the same media with smxUSBD mass storage. This technique allows plugging your target into a USB host such as Windows, and then operating on your target's media like a USB flash disk. In this case, Windows has its own file system that has its own view of the disk. smxUSBD only uses a block device driver to access sectors. The problem is that with two file systems accessing the disk, errors will be introduced because neither is aware of changes the other is making. For example, one may have part of the FAT modified in cache but not yet written to the media. The other may make changes to the same sector of the FAT.

The solution is to permit access to only one at a time and each un-mounts before letting the other access the media. Before plugging in the USB cable, you should close all open files and un-mount your file system. When done with USB access, you can re-mount the file system. Before unplugging the USB cable, the user should do Safely Remove Hardware to shut down the disk just like he would before unplugging a USB flash disk, or you can force smxUSBD to shut down the mass storage device.

The following code shows an example of how to share a RAM disk between smxFS and smxUSBBD.

```
sfs_init();
sud_Initialize();

/* Now register the RAM disk so smxFS can use it. But smxUSBBD cannot access mass storage device at
this time. */
sfs_devreg(sfs_GetRAM0Interface(), 0);

/* Create a sample file */
fp = sfs_fopen("A:\\fscreate.txt", "wb");
if(fp)
    sfs_fclose(fp);

/* Do other smxFS operations here. */

/* Shut down the disk for smxFS. */
sf_devunreg(0);

/* Now register the device driver with smxUSBBD. */
sud_MSRegisterDisk(sfs_GetRAM0Interface(),0);

/* Tell the user to plug in the USB cable so USB host can access the disk. */

/* Do other smxUSBBD operations here by USB Host through USB link. */

/* Force shutdown of the USBBD mass storage device. Then the USB host will not access it. */
sud_MSRegisterDisk(NULL,0);

/* Register it with smxFS again. If the USB host changed anything on the disk you will see it now. */
sfs_devreg(sfs_GetRAM0Interface(), 0);
```

This same technique should be applied if, for some reason, another file system is able to access the media (while mounted in your target).

## 10.3 Switching to Different Functions at Run Time

Your device may need to report to the USB host that it is a different function at different times. For example, normally it may be a mass storage device, but sometimes it may need to report to the host that it is a USB Ethernet over USB device, in order to change some administration settings. This is controlled by the application within your device. You need to cause the USB host to re-enumerate the device. Follow these steps:

### If you are using smxUSBBD v2.53 or later:

1. Enable multiple functions, such as SUD\_MSTOR and SUD\_NET in udcfg.h.
2. Call sud\_Initialize(SUD\_MSTOR\_MASK) to initialize the device's function to Mass Storage first.
3. When you need to switch the function, you need to call sud\_Release() first, wait about 100 ms and then call sud\_Initialize(SUD\_NET\_MASK) again switch to the Ethernet over USB function.

4. If you don't want to shutdown the whole stack, as is done in step 3, you can call `sud_Reconfig(SUD_NET_MASK)` first and then call `sud_ReAttach()` to force the host to re-enumerate the device. When using this approach, you need to make sure the host and device will not transfer any data and/or request during that period.

**If you are using smxUSBD v2.52 or earlier:**

1. Modify `sud_Initialize()` and `sud_Release()` prototype to add a flag parameter to indicate which function you want to use. For example;
2. Enable both functions you want to use, in `ucfg.h`
3. Modify the code in `sud_Initialize()` and `sud_Release()` to register and release the function driver according to the new parameter.

```

if (iFunction == SUD_FUNC_MS)
{
    if(sud_MSInit())
        return 0;
}
else if (iFunction == SUD_FUNC_NET)
{
    if(sud_NetInit())
        return 0;
}

if (iFunction == SUD_FUNC_MS)
{
    sud_MSRelease();
}
else if(iFunction == SUD_FUNC_NET)
{
    sud_NetRelease();
}

```

4. Initialize the USBBD stack with the default function parameters.
5. When you are ready to switch the function, first call `sud_Release()` to shutdown the whole stack then call `sud_Initialize()` with a different function flag. After this is done, shut it down again and restore it to the default function.

## 10.4 Mass Storage Function Driver Buffer Size

The mass storage function driver should have a buffer large enough to hold at least one sector data of the block device driver, because the block device can only be accessed by sector. Normally one sector is 512 bytes but it can be a multiple of 512. For example, if you are using the `smxFFS NAND` flash driver, the sector size may be 2048 for some large flash chips. So ensure `SUD_MSTOR_PACKET_SIZE` in `ucfg.h` is at least one sector. The default setting is 2048 which should meet all requirements. If

SUD\_MIN\_RAM is set to “1”, it becomes 512 which may not be correct if you are using a large NAND flash chip.

You may need to increase this buffer size if you are using some other media like MMC/SD card. If you write only 2048 byte for each MMC/SD Write Block Data command, you will not get the best write performance for most MMC/SD card. Windows PC normally may write a chunk of 64KB data for large data write operation through USB. If you increase this buffer size to 64KB and then write them at once to the media card, you can get the best performance based on your hardware. There is no big difference for reading with small or large buffer size.

If SUD\_MSTOR\_ASYNC\_ACCESS is set to 1 in udcfg.h, the mass storage driver needs to allocate more memory to handle asynchronous media disk accesses. The additional memory requirement is 3\*SUD\_MSTOR\_PACKET\_SIZE. See 3.1.1 udcfg.h for more information about this option.

## 10.5 Improving USB to Serial Function Driver Performance

A real serial port (RS232 or UART) is byte oriented, but USB data transfer is packet oriented. So only sending one byte each time to the USB is not a good design from the memory usage and performance point of view.

- For memory, the USB stack needs to allocate some extra data to maintain each data transfer request. For the current smxUSB implementation, besides the payload, each data transfer request needs an additional 48 bytes. If you only transfer one byte each time, the memory overhead is 4800%. But if you transfer 128 bytes each time, the overhead is only 37.5%. The memory overhead is very important for small memory systems.
- For performance, each IN or OUT data transfer needs three packets, Token, Data, and Handshake. The overhead for the extra packets and header is called **Protocol Overhead**. For a full speed BULK transfer, the protocol overhead is 13 bytes (3 SYNC bytes, 3 PID bytes, 2 Endpoint + CRC bytes, 2 CRC bytes, and a 3-byte inter-packet delay). If you are transferring one byte each time, the performance overhead is 1300%. But if you transfer 128 bytes each time, the overhead is only 23.4%.

Based on the above calculations, you cannot use the same programming approach of real serial port to program for a USB to Serial function driver.

Recommendations for best efficiency:

- On the host side, your application should call the USB host stack APIs to transfer as much data as possible each time. Always pass a buffer whose size is a multiple of the maximum USB packet size (64 bytes for full speed and 512 bytes for high speed) to the USB host stack. Never transfer only one or a few bytes unless it is required by the protocol.
- On the device side, cache the received data as much as possible in a buffer first, and then send the cached data by calling `sud_SerialWriteData()`. Never send one or only a few bytes unless you do not receive any more data or it is required by your protocol.

## 10.6 Linux Support

All smxUSBBD function drivers are supported by Linux. We tested them using Fedora 6, kernel 2.6.18. However, there are some special issues for Serial Port function drivers.

**Serial Port:** If you want the serial port to be supported by the Linux CDC-ACM class driver, you need to set `SUD_SERIAL_SUPPORT_ACM` to 1 in `udcfg.h`. We also found that the internal write buffer within the Linux CDC-ACM class driver by default has only two 64-byte buffers, so if you call `write()` to send more than 128 bytes data, some data will be truncated. You need to check the return value of `write()`; if it is less than the data buffer size, call it again to send remaining data.

## 10.7 MAC OS X Support

All the function drivers of the smxUSBBD were tested using iBook G4 and OS X 10.5.6. But there is a special issue for the Serial Port function driver.

**Serial Port:** If you want the serial port to be supported by the MAC OS CDC-ACM class driver, you need to set `SUD_SERIAL_SUPPORT_ACM` to 1 in `udcfg.h`.

## 10.8 USB Device Controller Soft Connect Feature

Some USB device controllers, such as NXP's, have the Soft Connect feature, which means if the `SoftConnect` bit is enabled by the driver, the controller will apply an internal 1.5KB pull up resistor to the D+ pin so the USB host port can detect a USB device is plugged in. Our device controller driver will by default enable this feature (in the `DCCConnect()` function). If you applied an external pull up resistor or you want to set you device as low speed (apply pull up resistor on D-), you may need to manually disable this feature in the source code of device controller driver. `SoftConnect` is useful if you want to force the USB host to re-enumerate the device without the need to remove the cable.

## 10.9 Opening a Serial Port on the Host

When you use any API on your host PC to open a USB serial port, be sure to set the DTR signal to 1. Our serial function driver checks that signal to ensure the serial port on the host has been opened.

## 10.10 Multiple Same Type Devices on the Same USB Host

Some USB hosts, such as Windows, use Vendor ID/Product ID/Serial Number to identify the USB device. If you may plug in multiple devices (such as USB serial) into one such USB host, a unique serial number is required for multiple devices to work properly, simultaneously. Sometimes you can use the Ethernet MAC address of your device or OTP bytes of your flash memory as a serial number for the USB.

**Plugging two serial devices that have the same VID/PIO/SerialNum into a Windows XP PC will cause Windows to hang.**

## 10.11 HID vs. Serial for Data Communication

The user can choose whether to use the HID communication or serial function driver to do data transfer between the USB device and host PC.

### Advantages for HID device

1. There is no need to install a driver on the host, so the user need not have administrator privileges.
2. HID device is well-supported by most commercial operating systems.

### Disadvantages for HID device

1. It is only suitable for small amounts of data. You can only transfer 64 bytes of data in each packet, and the interrupt polling interval is a few milliseconds, so you can not get high throughput
2. Interrupt transfer may lose data.
3. You need to write a special host application to access the HID device.

### Advantages for serial device

1. You may get very high performance.
2. Bulk transfer is reliable.
3. You can re-use most existing legacy serial port application code or utilities.

### Disadvantages for serial device

1. You need to install a driver on Windows (or at least an .inf file). For 64-bit Windows, you may need to sign the driver.
2. There are always some limitations for the serial class driver in the operating system.

## 10.12 HID Communication Multiple Reports

The HID Communication function driver supports multiple reports. It is similar to multiple serial ports to transfer data for different functions within the same USB device.

By default the HID Communication function driver will use only one report for both Input and Output. The default report ID is 1.

To enable multiple reports, you need to:

1. Set `SUD_HID_REPORT_NUM` to the number of reports you want to have.
2. Modify the HID descriptor defined in `udhidcom.c` for the reports' ID, count, and size if the default setting does not meet your requirement. If `SUD_HID_REPORT_NUM` is larger than 4, you may need to add more entries to the descriptor.
3. Check the setting `SUD_EP0_BUFFER_SIZE` in `udcfg.h` to make sure it is big enough to hold the whole HID descriptor. The default size for the HID device is 256.

## 10.13 Video Camera Software

Camera software packages vary a lot in their operation. Some require setting the resolution to match the image output by our demo. Some require switching from a built-in webcam to the `smxUSBD` device or to start/stop it.

The Windows 8 camera app does not seem to allow setting the desired resolution and instead selects the highest resolution the driver says it supports, so if no data is actually supplied for that case (as may be true in our demo), nothing will display. In `uvideo.c`, change the `VideoInUncompressedFrameCfg[]` definition to have only lines for the resolutions you will actually use and set that number of lines in `SUD_VIDEO_FORMAT_UNCOMPRESSED_NUM`. Also, it requires images to be QVGA (320x240) or larger, which we found from experimentation. It displays nothing for smaller images.



## Appendix A. Memory Usage and Performance Summary

### A.1 Size

#### A.1.1 Code Size

Code size will vary widely depending upon CPU, compiler, and optimization level. The figures below are intended as examples.

<b>Component</b>	<b>ARM Thumb IAR v4.41</b>	<b>ARM Thumb-2 IAR v6.10</b>	<b>ARM IAR v4.41</b>	<b>Blackfin VisualDSP</b>	<b>CF CW v6.4</b>
Core	5 KB	4 KB	8 KB	12 KB	9 KB
Audio driver	3 KB	3 KB	6 KB	3.5 KB	6.5 KB
DFU driver	N/A	N/A	2 KB	N/A	N/A
HID Communication driver	0.5 KB	0.5 KB	1 KB	1 KB	1 KB
Keyboard driver	0.5 KB	0.5 KB	1 KB	1 KB	1 KB
Mass Storage driver	3.1 KB	3 KB	5 KB	5.5 KB	5 KB
Mouse driver	0.5 KB	0.5 KB	1 KB	1 KB	1 KB
MTP driver	N/A	N/A	8 KB	N/A	N/A
Ethernet over USB driver	2.5 KB	2 KB	3.5 KB	2.5 KB	4.7 KB
Serial driver	1.5 KB	1.5 KB	2.5 KB	2.5 KB	2.7 KB
Video driver	N/A	11 KB	16 KB	N/A	N/A
Composite driver	0.5 KB	0.5 KB	1 KB	1 KB	1 KB
<b>Device Controller Drivers</b>					
Analog Devices Blackfin	N/A	N/A	N/A	3.3 KB	N/A
Atmel AT91SAM9260/1/3, AT91SAM9SE, AT91RM9200	2 KB	N/A	3 KB	N/A	N/A
Atmel AT91SAM9M10/G45, AT91SAM9RL64, AT91CAP9, AT91SAM3U4	N/A	N/A	4 KB	N/A	N/A
Freescale CF5225x/1x/2x, Kxx	N/A	4 KB		N/A	5 KB
Freescale CF532x/7x, 525x, 5445x, iMX31, LPC3131/41/51	N/A	N/A	3.5KB	N/A	4 KB

Freescale CF548x	N/A	N/A	N/A	N/A	9 KB
Freescale i.MX1	N/A	N/A	5.4 KB (CW)	N/A	N/A
Maxim MAX342x	N/A	N/A	3.5 KB	N/A	N/A
NEC uPD720150	N/A	2 KB	N/A	N/A	N/A
NXP ISP1181, ISP1161, ISP1362	N/A	N/A	N/A	N/A	4 KB
NXP ISP158x, ISP1761/3	N/A	N/A	N/A	N/A	N/A
NXP LPCxxxx	2.6 KB	2 KB	4 KB	N/A	N/A
PLX Net2272	N/A	N/A	N/A	4.5 KB	N/A
Sharp LH7A400/4	2.5 KB	N/A	4 KB	N/A	N/A
STMicro STR7/9, STM32F101/2/3	2.5 KB	N/A	4 KB	N/A	N/A
Synopsys, STMicro STM32F105/7, STM32F20x	N/A	2.5 KB	N/A	N/A	N/A
TI AM1x/AM35x, LM3Sxxxx	N/A	2 KB	3 KB	N/A	N/A

## A.1.2 Data Size (RAM Requirement)

The following is a table of RAM usage.

<u>Component</u>	<u>Size</u>
Core	1.5 KB
Audio driver	2 KB
DFU driver	1 KB
HID Communication driver	0.5 KB
Keyboard driver	0.5 KB
Mass Storage driver	2 KB
Mouse driver	0.5 KB
MTP driver	6 KB + ObjectsNum*64
Ethernet over USB driver	2 KB
Serial driver (each port)	1 KB
Video driver (full speed)	4 KB
Video driver (high speed)	7 KB
Composite driver	0.5 KB
<b>Device Controller Drivers</b>	
Analog Devices Blackfin	0.5 KB
Atmel AT91SAM9260/1/3, AT91SAM9SE, AT91RM9200	0.5 KB
Atmel AT91SAM9M10/G45, AT91SAM9RL64, AT91CAP9, AT91SAM3U4	0.5 KB
Freescale CF5225x/1x/2x, Kxx	1 KB
Freescale CF532x/7x, 525x, 5445x, iMX31, LPC3131/41/51	1 KB
Freescale CF548x	1 KB
Freescale i.MX1	0.5 KB
Maxim MAX342x	0.5 KB
NEC uPD720150	0.5 KB
NXP ISP1181/1161/1362	0.5 KB
NXP ISP158x/1761/1763	1 KB
NXP LPCxxxx	0.5 KB

PLX Net2272	0.5 KB
Sharp LH7A400/4	0.5 KB
STMicro STR7/9, STM32F101/2/3	0.5 KB
Synopsys, STMicro STM32F105/7, STM32F20x	0.5 KB
TI AM1x/AM35x, LM3Sxxxx	0.5 KB

### A.1.3 Stack Size (RAM Requirement)

smxUSBD has one internal task in the device controller driver that uses about 1KB stack (an additional 1KB for MTP). Application tasks typically use 0.5 to 1.5KB depending on the function driver.

## A.2 Performance

For theoretical performance limits, refer to the tables in Chapter 5 of the Universal Serial Bus Specification, Revision 2.0. Keep in mind that they do not account for software overhead, and the class driver also introduces some overhead. Reaching even 60% of the limit in real world use is a very good result, especially for high speed.

### A.2.1 Mass Storage Performance

The following is a table of read/write performance (based on RAM disk).

<u>Device Controller</u>	<u>Reading</u>	<u>Writing</u>
Analog Devices Blackfin	5000 KB/s	5000 KB/s
NXP ISP1181	1071 KB/s	1071 KB/s
NXP ISP158x	5300 KB/s	3890 KB/s

### A.2.2 Remote NDIS Performance

<u>Device Controller</u>	<u>Packet Size</u>	<u>Sending/Receiving</u>
Freescale CF532x/7x	512	265 KB/s

### A.2.3 Serial Performance

The following is a table of sending/receiving bi-directional data performance.

<u>Device Controller</u>	<u>Packet Size</u>	<u>Sending/Receiving</u>
Analog Devices Blackfin	256	800 KB/s
Analog Devices Blackfin	1024	2500 KB/s

Freescale MCF54455	16K	8000 KB/s
NXP ISP1181	64	140 KB/s
NXP ISP1181	256	460 KB/s
NXP ISP1181	512	804 KB/s
NXP ISP1181	1024	887 KB/s
NXP ISP158x	512	1830 KB/s
NXP ISP158x	1024	2870 KB/s

## ***Appendix B. Block Device Driver Interface***

smxBase defines a generic block device driver interface that can be used for any block device, such as smxFS and smxUSBD Mass Storage driver. This interface is defined in XBASE\bbd.h, and it is not dependent on any other component of the SMX RTOS. Please see the smxBase User's Guide for the details of this Block Device Interface.

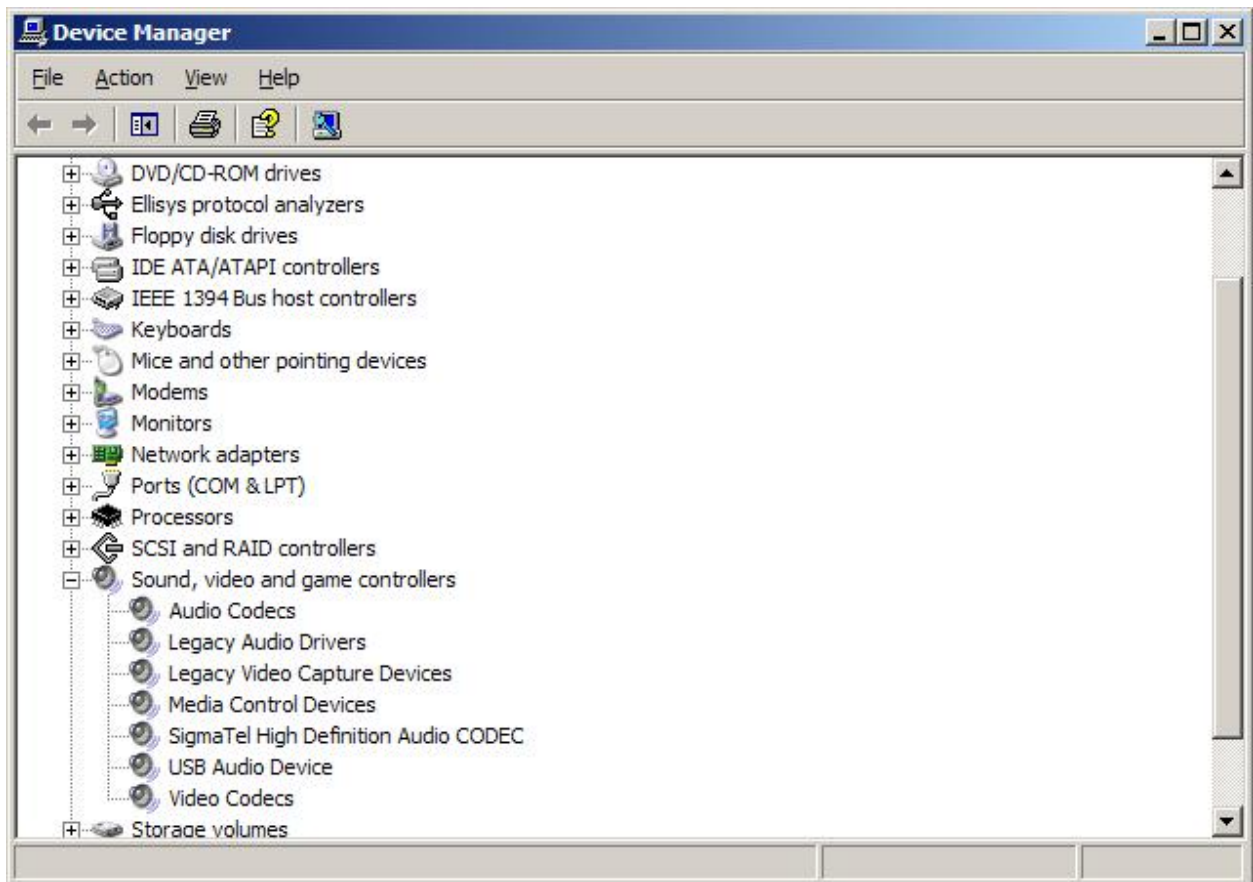
## Appendix C. Installing Devices under Windows 2000

smxUSBD devices are supported by Windows 2000 built-in drivers. The following screen shots show the steps for Windows 2000. If it doesn't work after installing the device driver, try ending the target debug session and starting a new one. If that doesn't work, try restarting Windows. Contact Micro Digital if you still have trouble.

First see section 3.3 Building and Running the Demos before running the demos.

### C.1 Audio

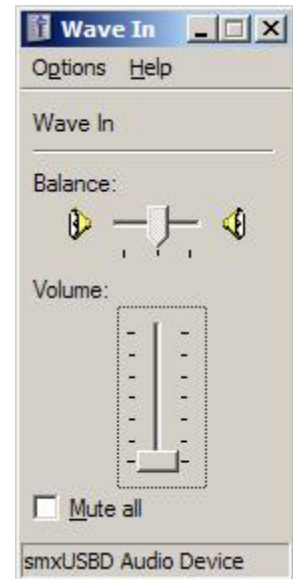
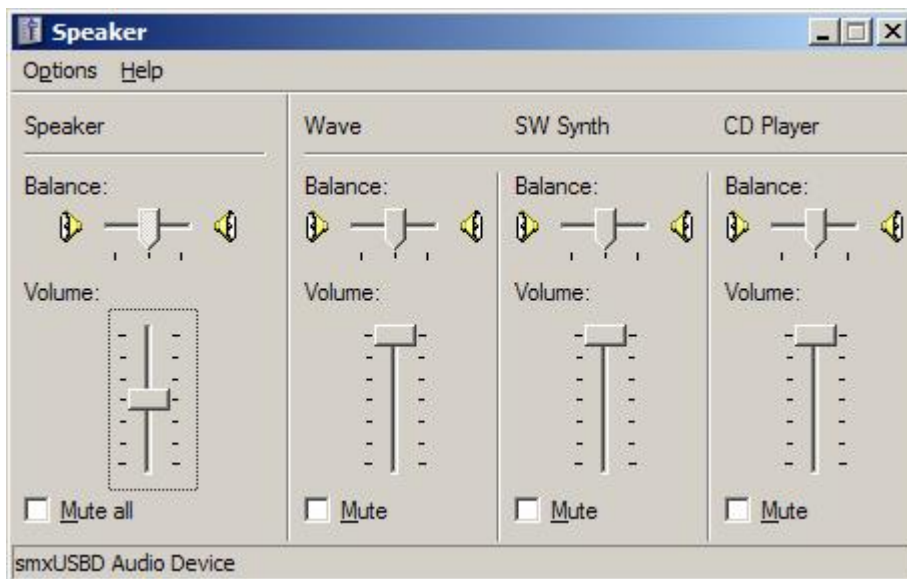
Windows 2000 will install the audio device automatically when you plug in the device (your target board). After installation, you can check the audio device in Device Manager.



You can also check the device in Control Panel | Sounds and Audio Devices | Audio



Speaker and Microphone volume control (Mixer):





## C.2 Mass Storage

Windows will install the mass storage device automatically when you plug in the device (your target board). Our demo program (which emulates a flash disk using RAM on your board) automatically partitions and formats the RAM disk.

## C.3 Mouse/Keyboard

Windows 2000 will install the mouse/keyboard driver automatically when you plug in the device (your target board).



De-select all choices:

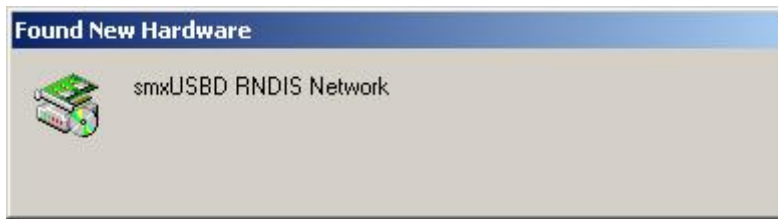




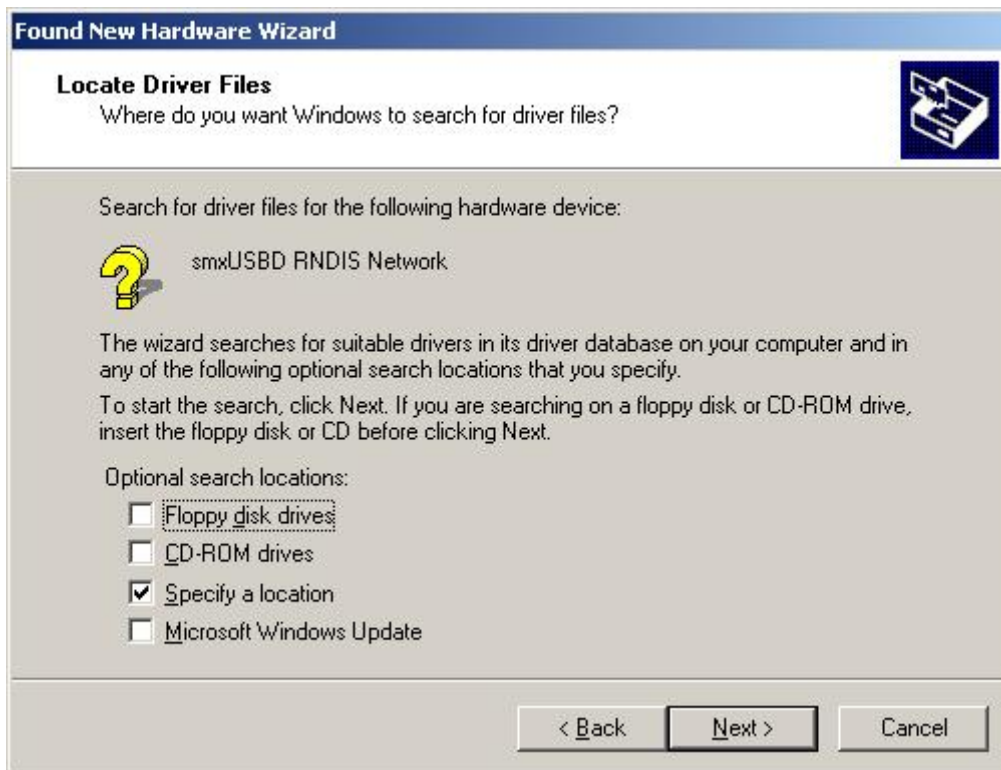
## C.4 Ethernet over USB

To install the Ethernet over USB driver, you may need an .inf file and two Microsoft Windows 2000 patch files. Use mdirndis.inf provided in the smxUSBD Function subdirectory. The patch files (rndismpy.sys and usb8023y.sys in the RNDISW2K subdirectory of the Function directory) are provided by Microsoft to support this feature on Windows 2000. We provide them for convenience, since they are normally part of a large package (Microsoft Remote NDIS USB Driver Kit, available at [www.microsoft.com/whdc/device/network/default.mspx](http://www.microsoft.com/whdc/device/network/default.mspx)).

Windows 2000 will pop up a dialog when you plug in your target, to inform you it has found new hardware.

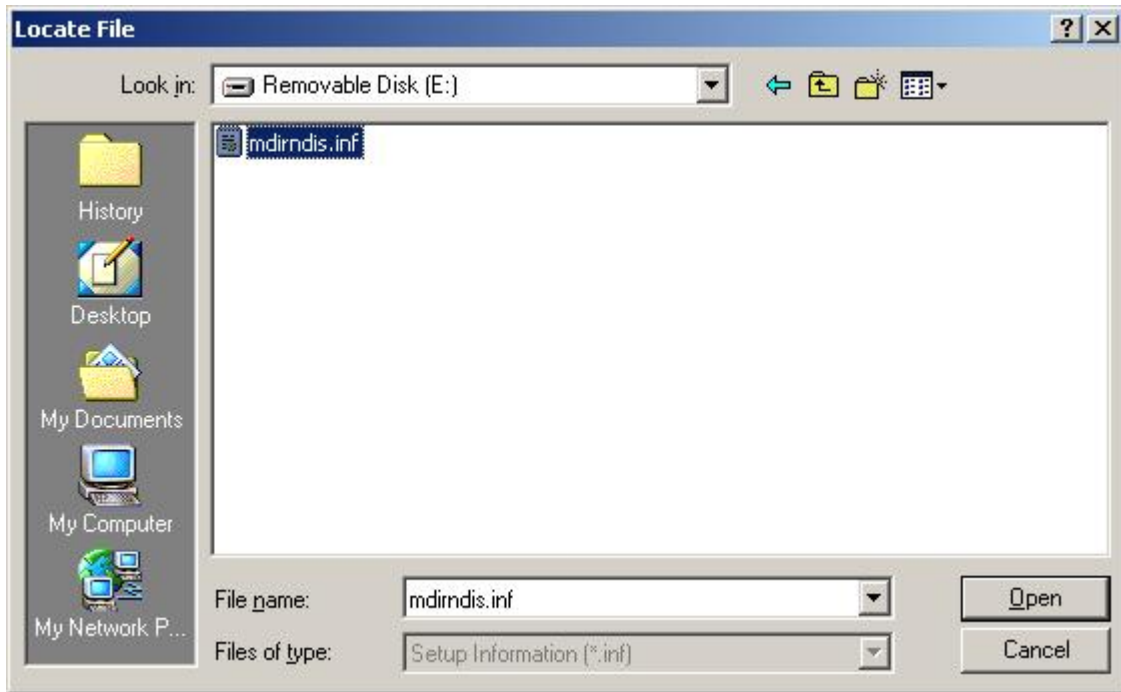


Let Windows search for a suitable driver:



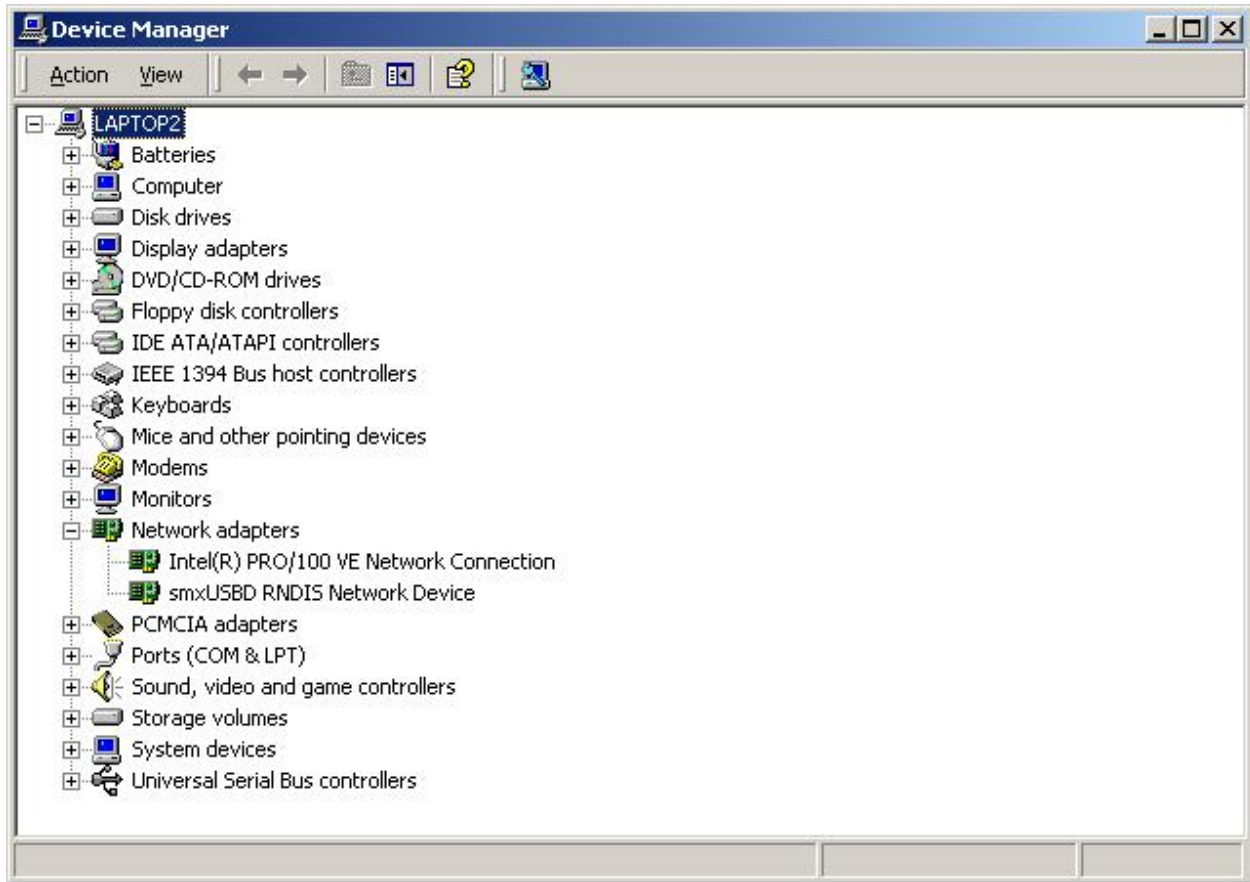


You may need to copy mdirndis.inf and the two driver files to a temporary directory:



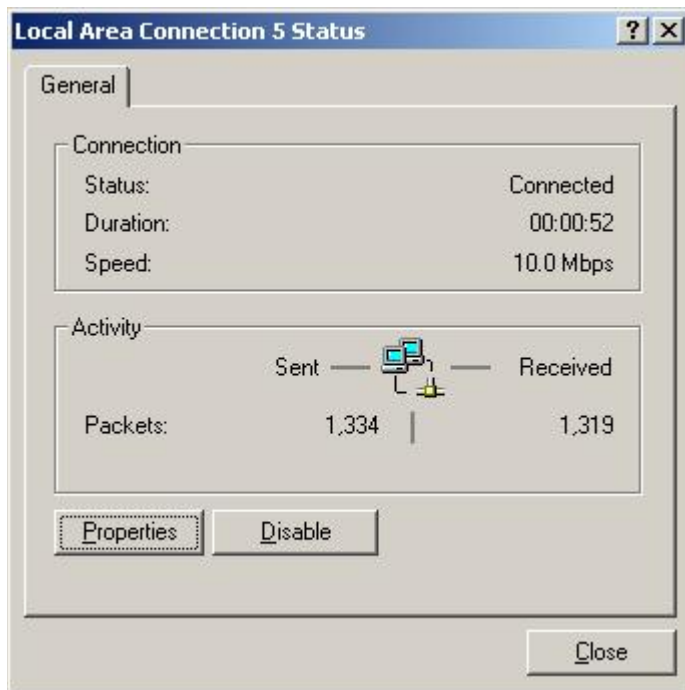


In the Device Manager, you will see a new Network Adapter has been added:



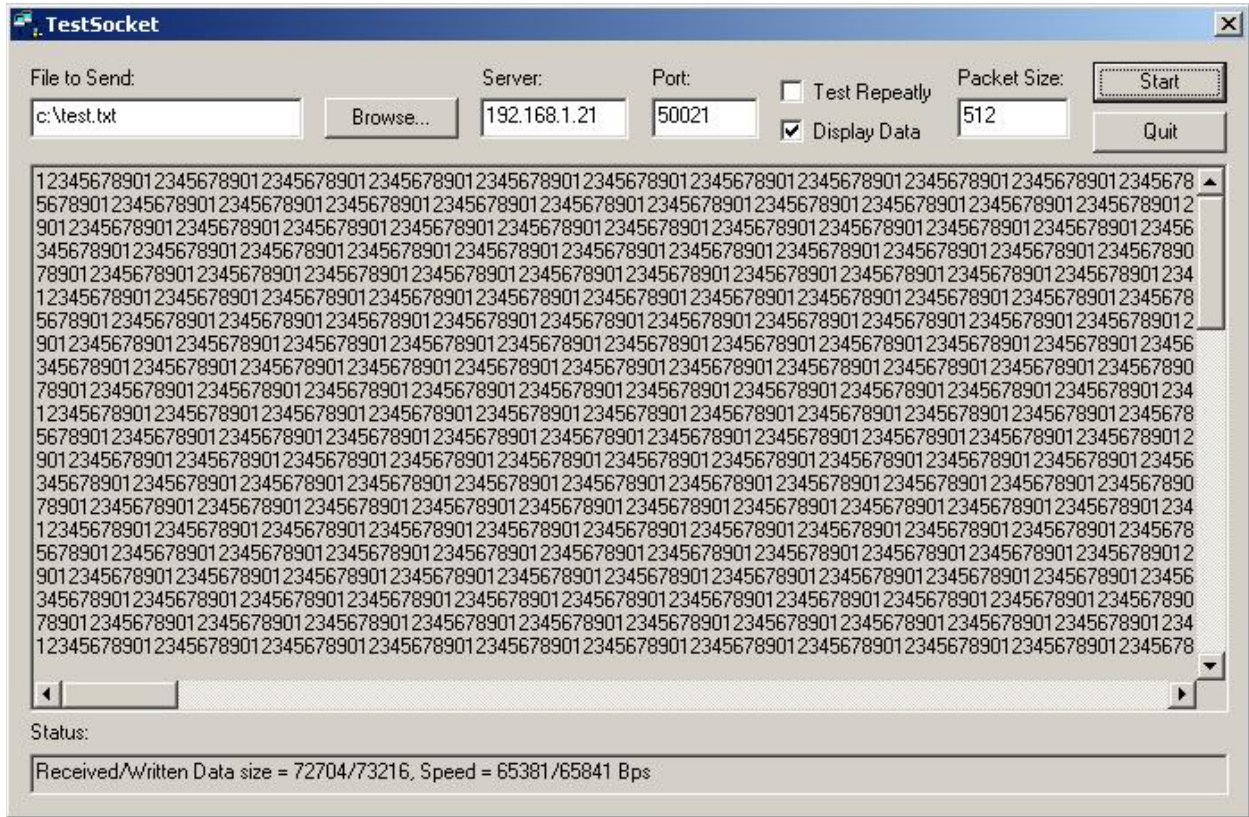


You can use Control Panel | Network Connections to check the status of this virtual adapter:



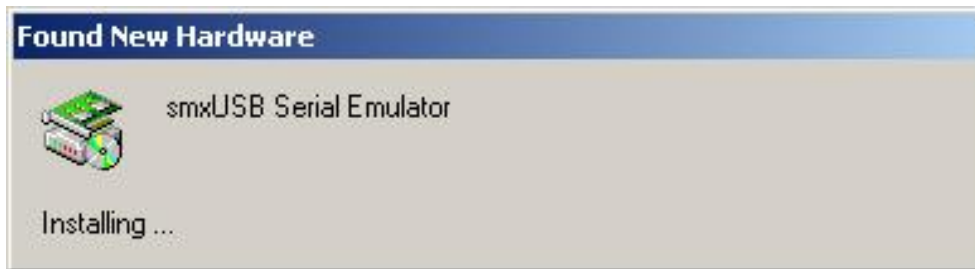
Click the Properties button and manually configure the IP address and mask. This step is not necessary if you are using the smxNS DHCP server, since the address is assigned automatically.

You can use our TestSocket.exe utility to test it:

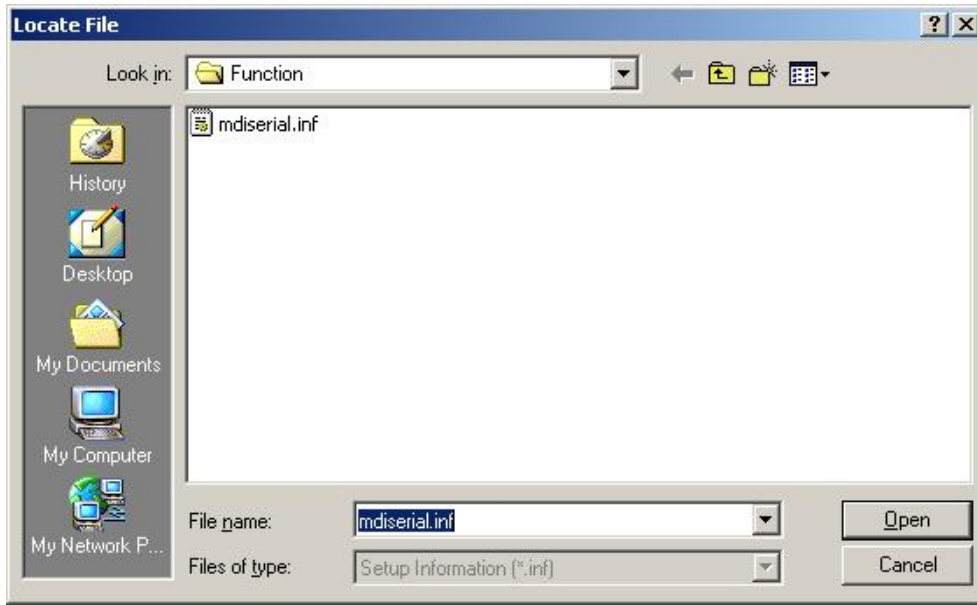


## C.5 Serial Port

To install the serial port driver, you may need an .inf file. Several are provided in the XUSBDFunction\Serial and SerialM directories for different cases (single port, multi-port, composite, and whether it uses the Windows built-in driver or the Micro Digital driver. The files in XUSBDFunction\Serial are for the Windows driver; the files in XUSBDFunction\SerialM are for the Micro Digital Driver. See section 9.1 Multiple Port Serial Device (or Single Port Limited Endpoints) for more information. Note that for Windows 2000, multi-port serial requires the Micro Digital driver.







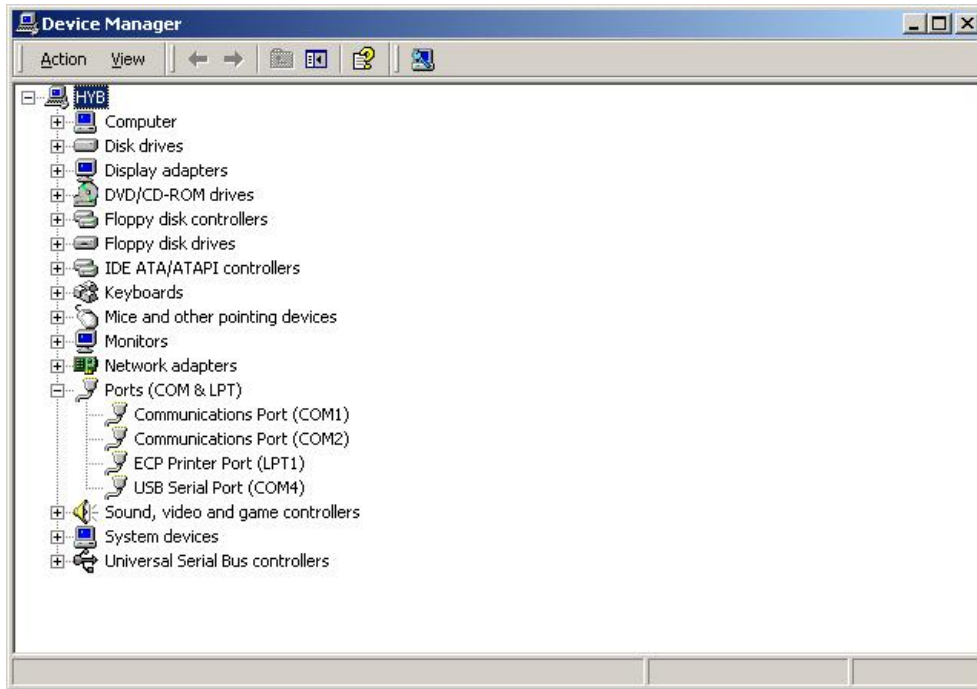


See Appendix H. Host OS Certification for more information about this warning.

Windows will search for the driver `usbser.sys`. It may prompt you to insert your Windows installation CD if this file is not in your Driver Cache or Service Pack .cab file.



You should see USB Serial Port (COMx) listed under Ports in Device Manager. If not, try restarting Windows.



You can use HyperTerminal to test if your serial port emulator works properly. Or you can use our TestComm utility to do the performance and stress testing. It is in the BIN directory. TestComm initially selects the highest COM port because that is the most likely one to be for the smxUSBD target just plugged in. If you started TestComm before plugging in the USB cable, click the refresh button to update the COM port list.

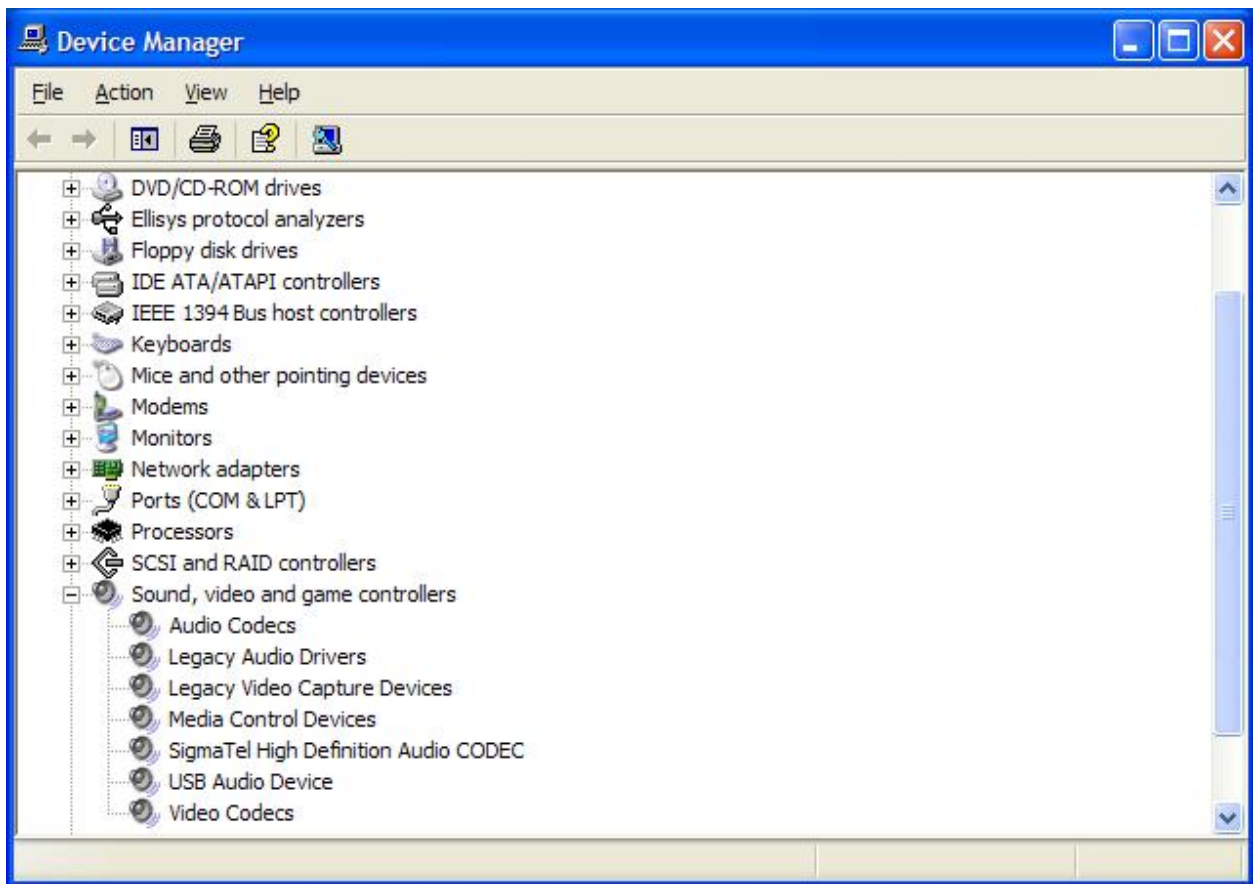
## Appendix D. Installing Devices under Windows XP

smxUSBD devices are supported by Windows XP built-in drivers. The following screen shots show Windows XP installation steps. Because Windows XP has better USB support than Windows 2000, the steps are simpler. If it doesn't work after installing the device driver, try ending the target debug session and starting a new one. If that doesn't work, try restarting Windows. Contact Micro Digital if you still have trouble.

First see section 3.3 Building and Running the Demos before running the demos

### D.1 Audio

Windows XP will install the audio device automatically when you plug in the device (your target board). After installation you can check the audio device in Device Manager.

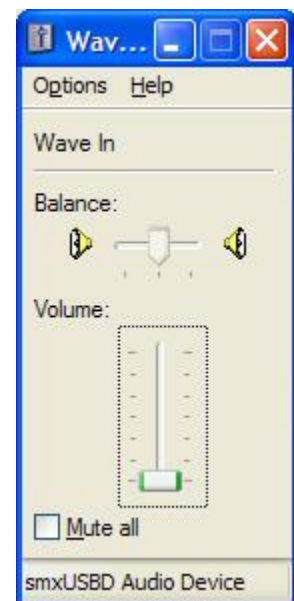
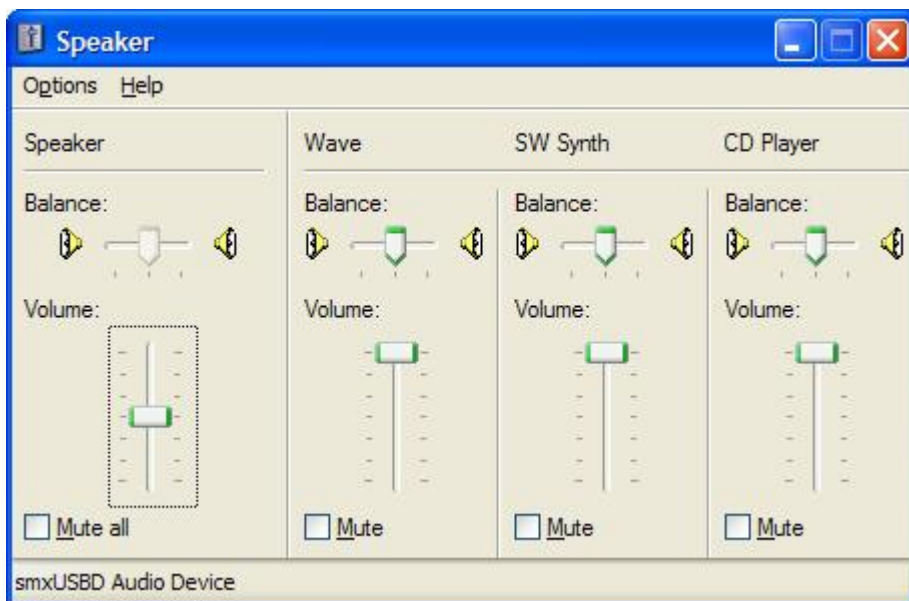




You can also check the device in Control Panel | Sounds and Audio Devices | Audio



Speaker and Microphone volume control (Mixer):



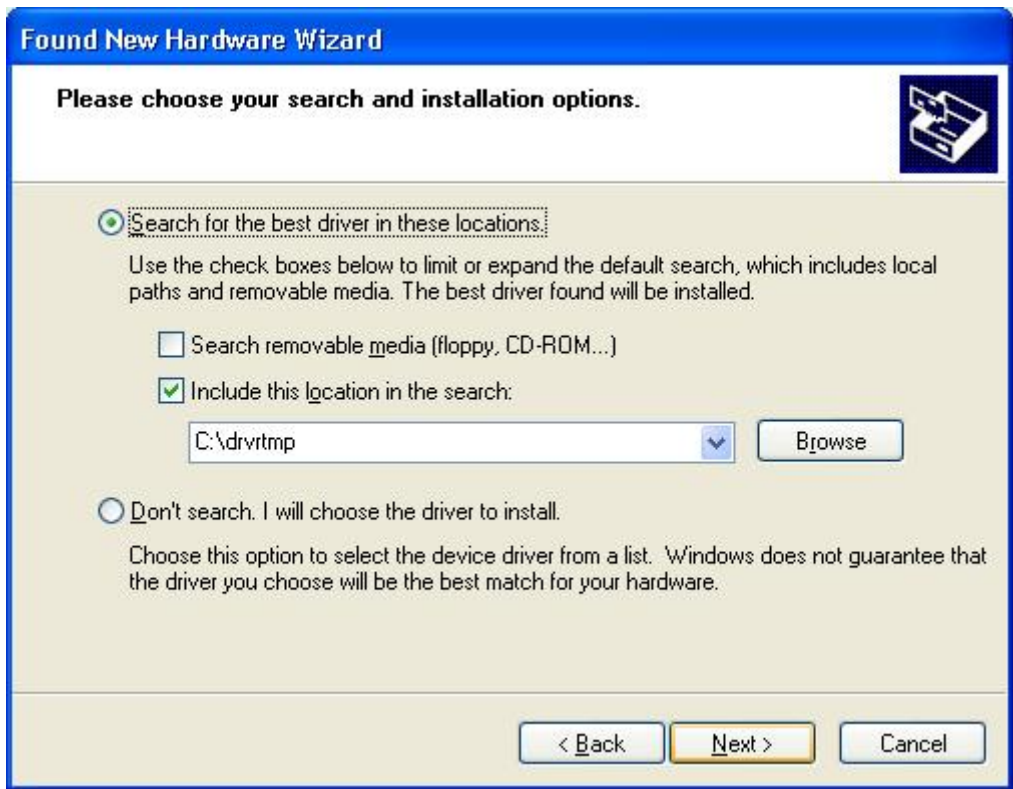
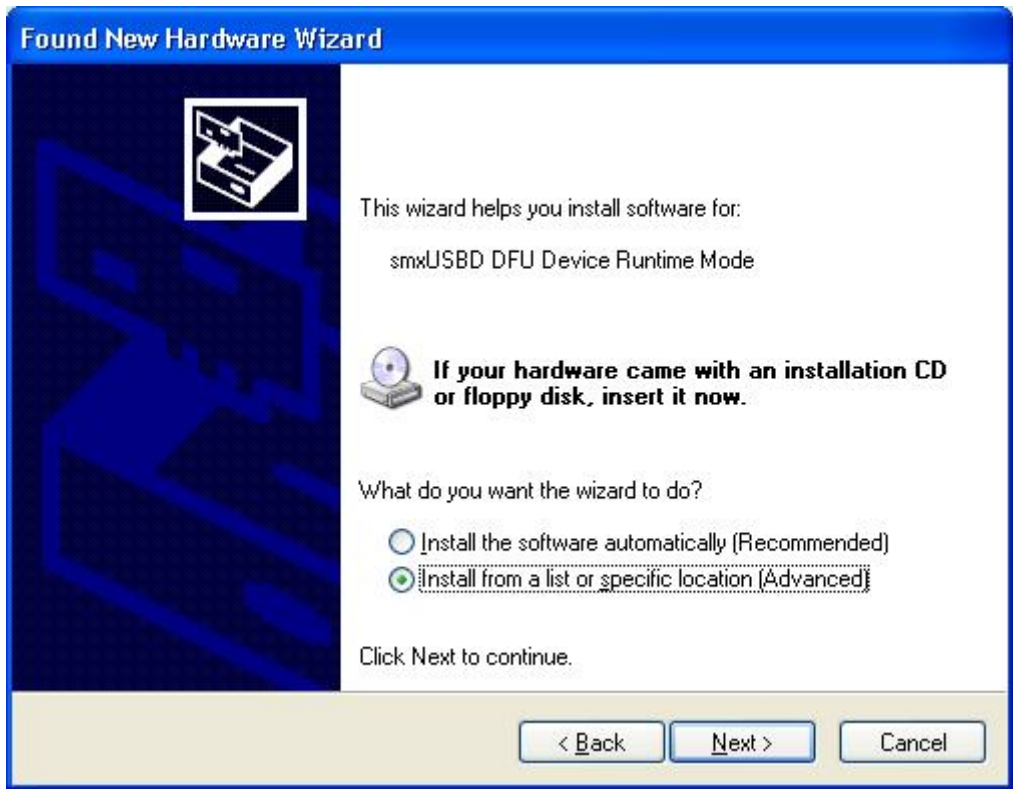
## D.2 Device Firmware Upgrade (DFU)

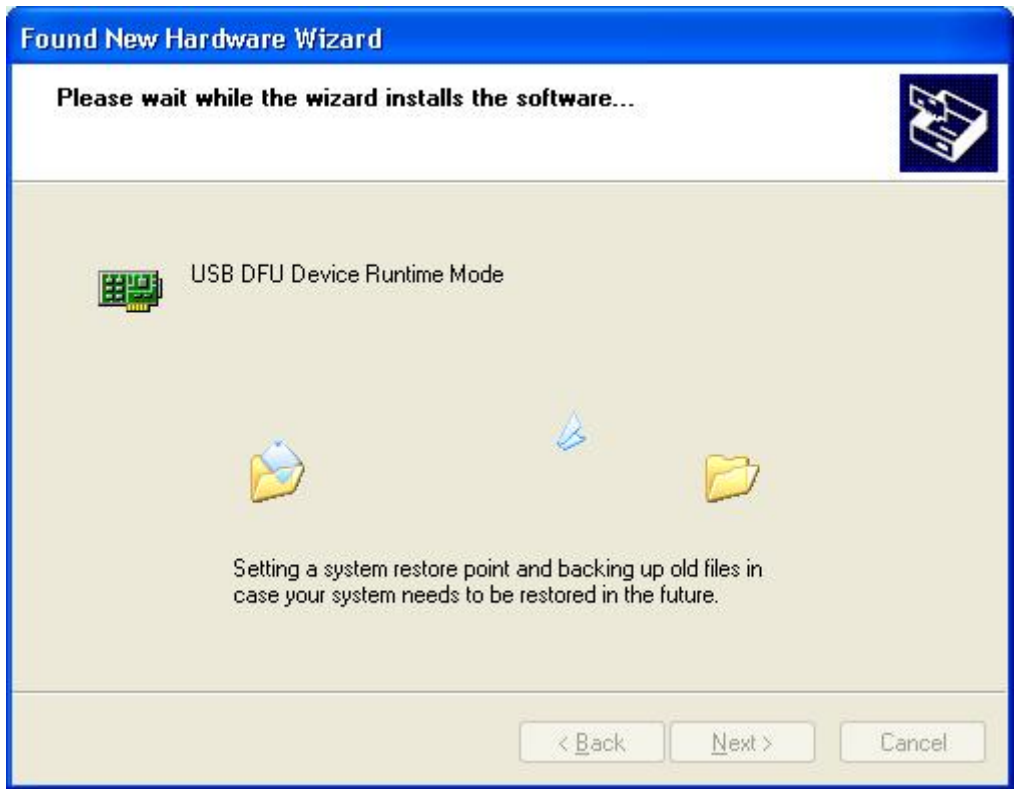
Windows XP has no built-in driver and utility for the DFU device. You need to install the driver provided by MDI or another. See section 9.2 Device Firmware Upgrade (DFU) Device.

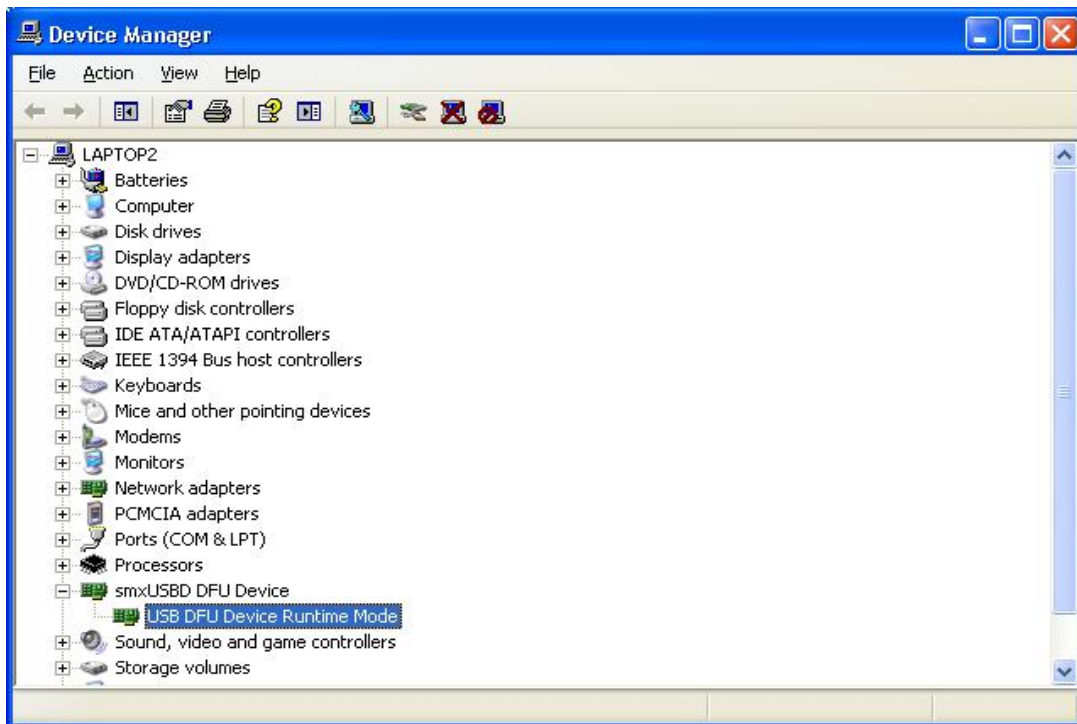
A DFU device has two modes. One is the runtime mode, which shows the normal function and DFU ability. The other is DFU mode, which is for firmware upgrade only. These two modes may use different VID/PID, so you may need to install the driver twice.

**Here are screenshots for runtime mode:**



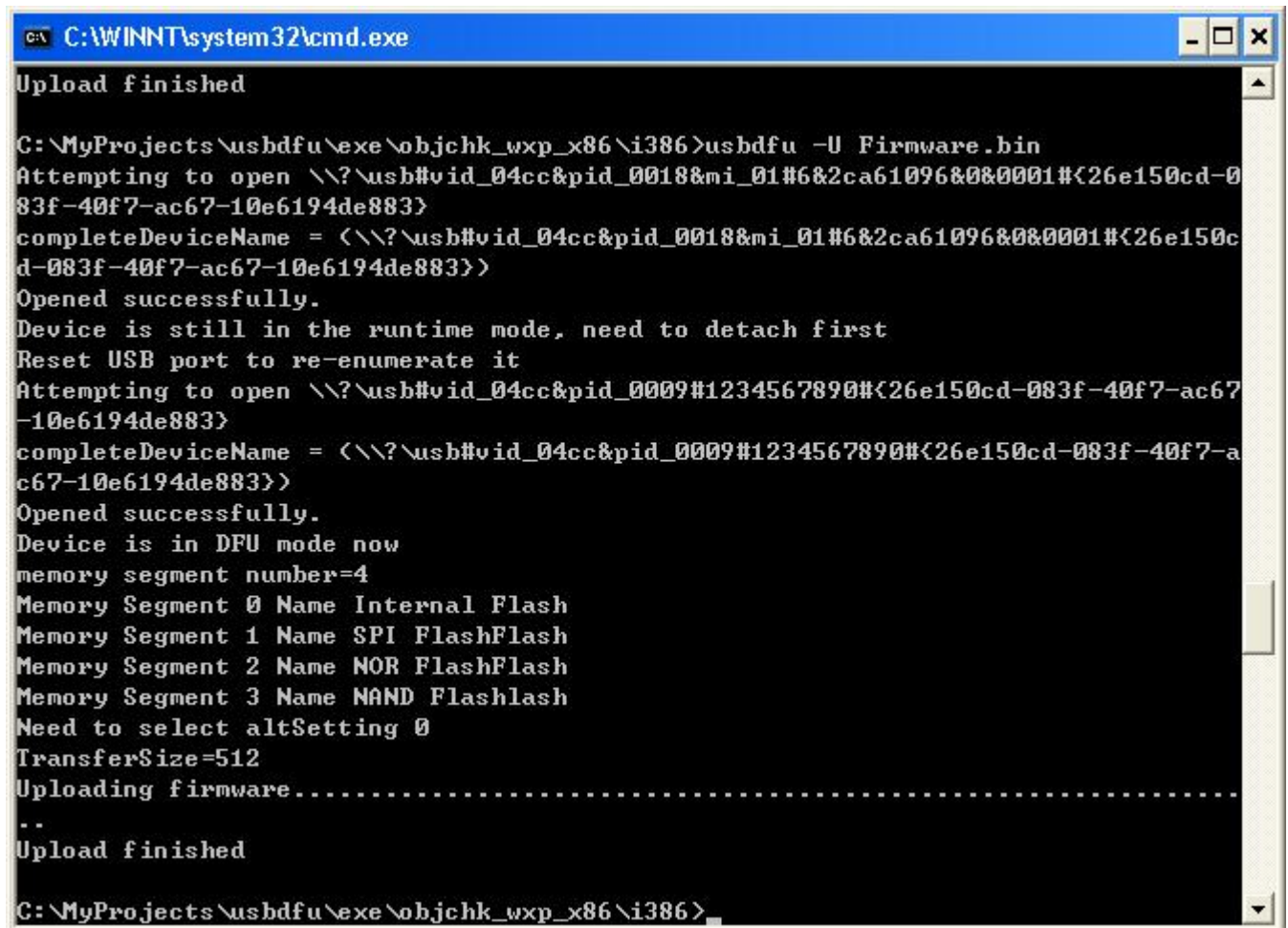








You can use our BIN\usbdfu.exe command line utility to upload/download firmware. This command line utility provides a simple way to verify your device's DFU ability. It is more like a testing tool. To re-build usbdfu.exe, you need the Windows Driver Kits environment.

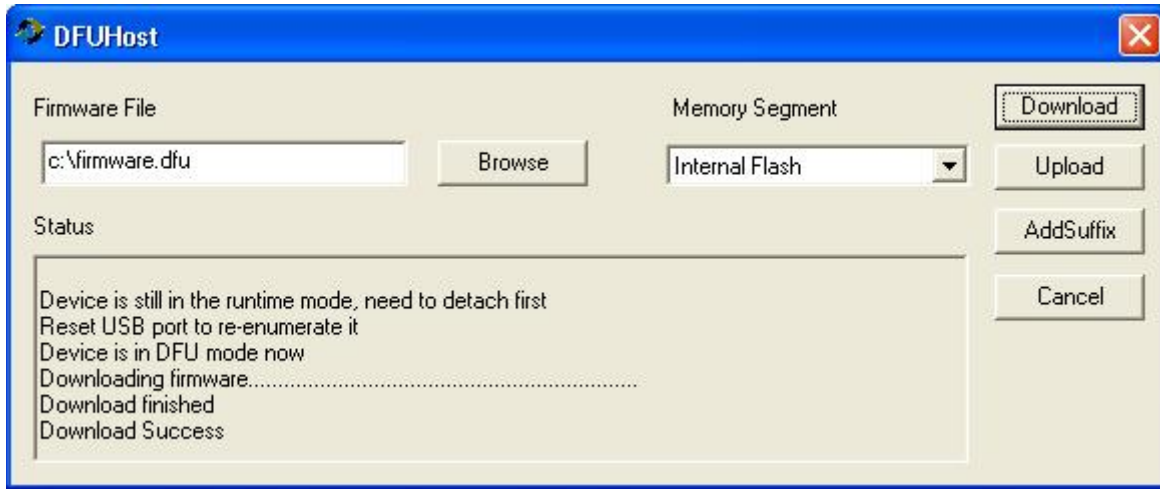


```
C:\WINNT\system32\cmd.exe
Upload finished

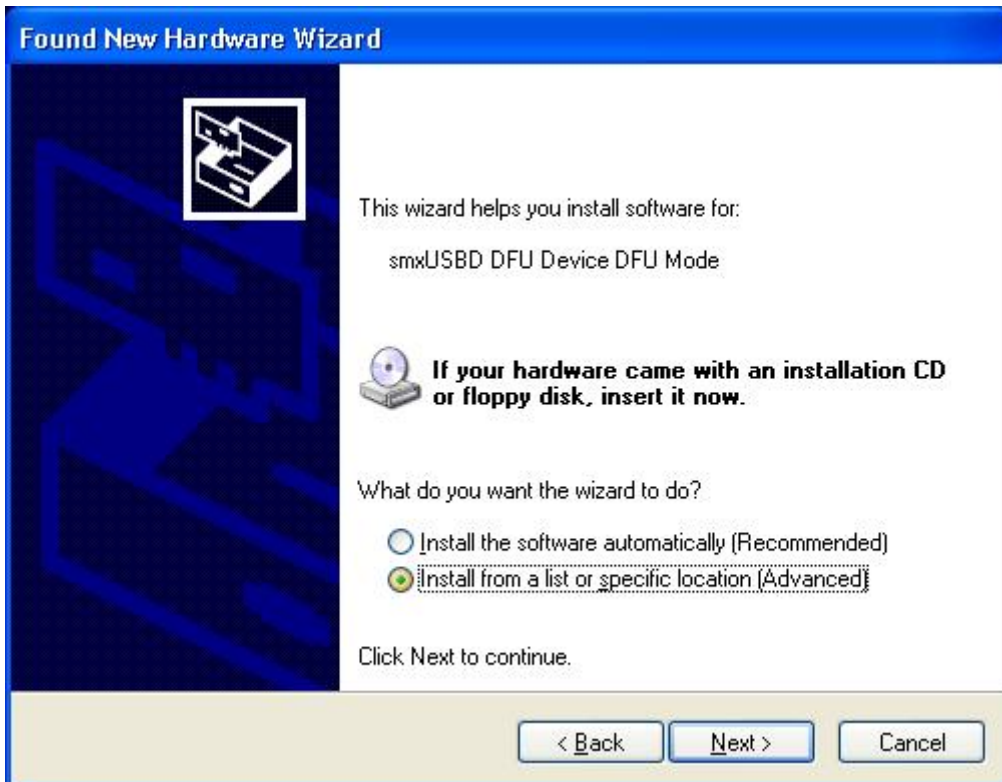
C:\MyProjects\usbdfu\exe\objchk_wxp_x86\i386>usbdfu -U Firmware.bin
Attempting to open \\?\usb#vid_04cc&pid_0018&mi_01#6&2ca61096&0&0001#\{26e150cd-083f-40f7-ac67-10e6194de883}
completeDeviceName = (<\\?\usb#vid_04cc&pid_0018&mi_01#6&2ca61096&0&0001#\{26e150cd-083f-40f7-ac67-10e6194de883}>)
Opened successfully.
Device is still in the runtime mode, need to detach first
Reset USB port to re-enumerate it
Attempting to open \\?\usb#vid_04cc&pid_0009#1234567890#\{26e150cd-083f-40f7-ac67-10e6194de883}
completeDeviceName = (<\\?\usb#vid_04cc&pid_0009#1234567890#\{26e150cd-083f-40f7-ac67-10e6194de883}>)
Opened successfully.
Device is in DFU mode now
memory segment number=4
Memory Segment 0 Name Internal Flash
Memory Segment 1 Name SPI FlashFlash
Memory Segment 2 Name NOR FlashFlash
Memory Segment 3 Name NAND Flashlash
Need to select altSetting 0
TransferSize=512
Uploading firmware.....
..
Upload finished

C:\MyProjects\usbdfu\exe\objchk_wxp_x86\i386>
```

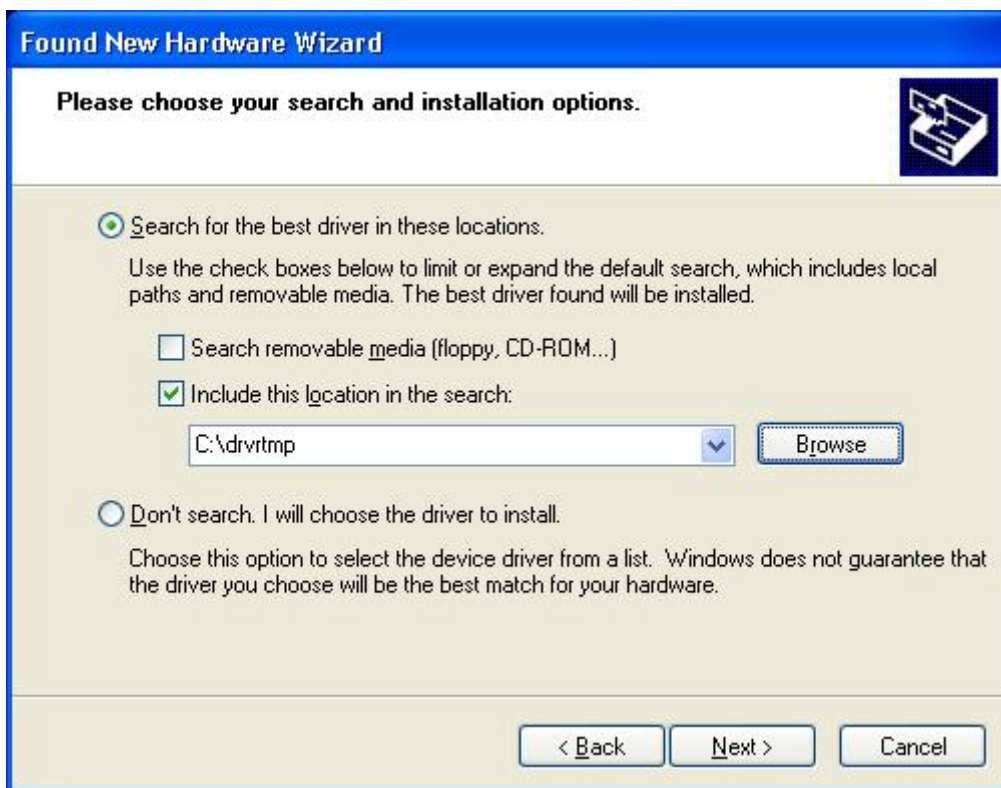
Or you can use our BIN\DFUHost.exe GUI application to upload/download firmware. Contact MDI if you need to integrate the DFU feature into your own GUI application. DFUHost.exe is a good example showing how to do that. To re-build it, you need Visual C++ only, no need for Windows Driver Kits.

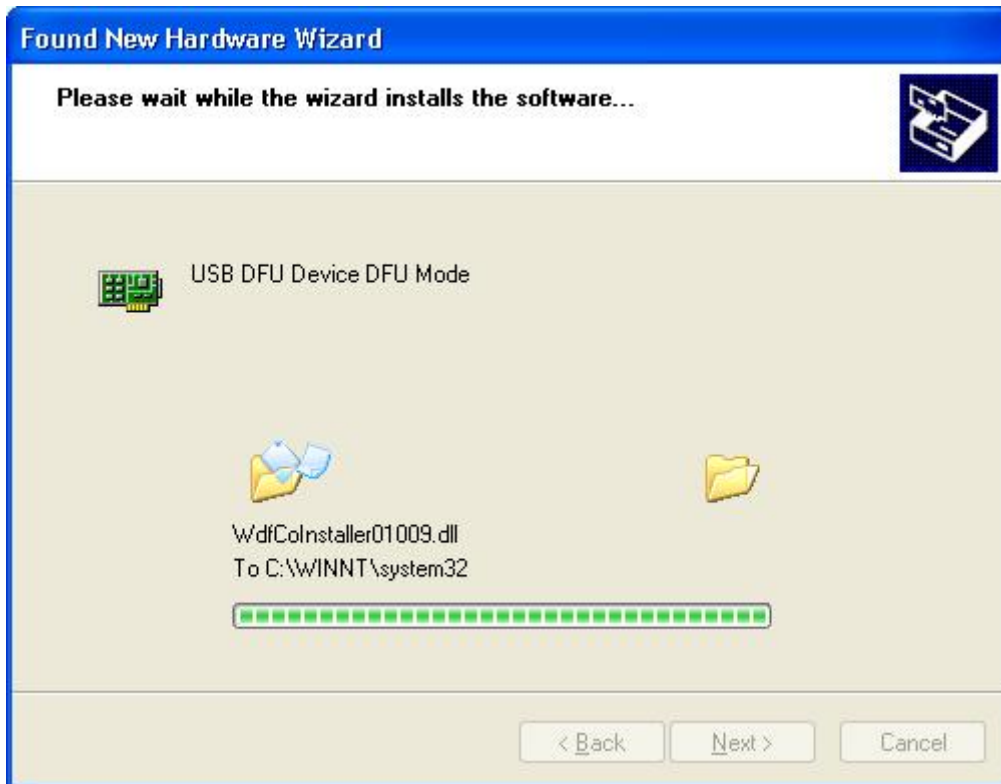


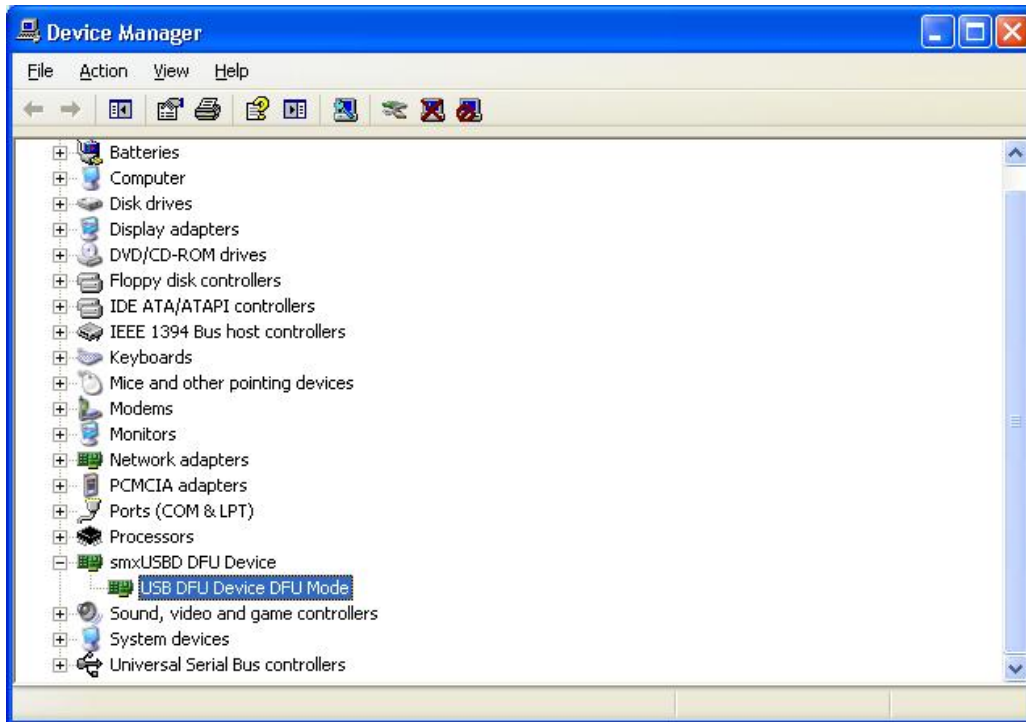
Here are screenshots for DFU mode:











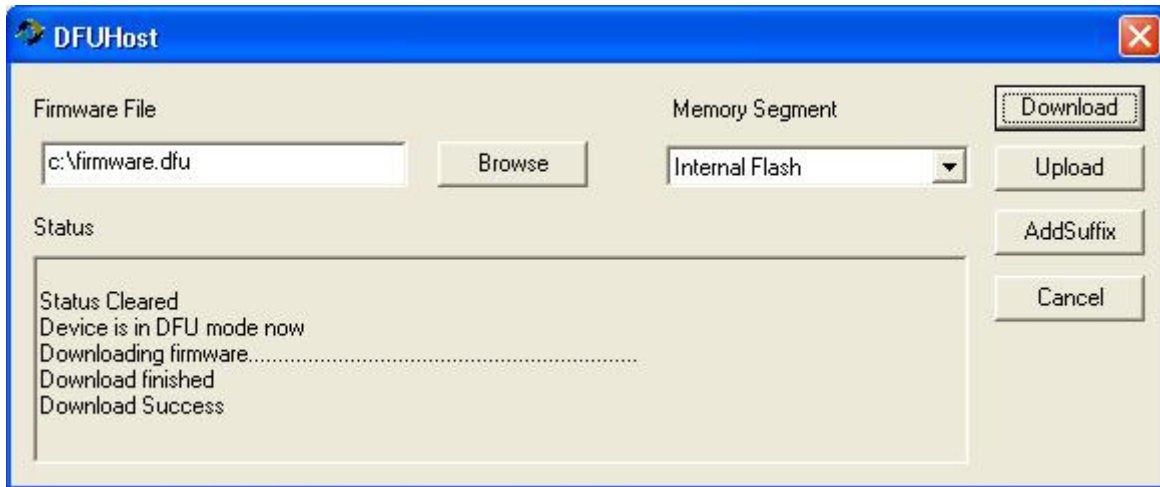
You can use our `BIN\usbdfu.exe` command line utility to upload/download firmware.

```
C:\WINNT\system32\cmd.exe

C:\MyProjects\usbdfu\exe\objchk_wxp_x86\i386>usbdfu -U Firmware.bin
Attempting to open \\?\usb#vid_04cc&pid_0009#1234567890#<26e150cd-083f-40f7-ac67-10e6194de883>
completeDeviceName = (<\\?\usb#vid_04cc&pid_0009#1234567890#<26e150cd-083f-40f7-ac67-10e6194de883>)
Opened successfully.
Status Cleared
Device is in DPU mode now
memory segment number=4
Memory Segment 0 Name Internal Flash
Memory Segment 1 Name SPI FlashFlash
Memory Segment 2 Name NOR FlashFlash
Memory Segment 3 Name NAND Flashlash
Need to select altSetting 0
TransferSize=512
Uploading firmware.....
..
Upload finished

C:\MyProjects\usbdfu\exe\objchk_wxp_x86\i386>
```

Or you can use our BIN\DFUHost.exe GUI application to upload/download firmware.

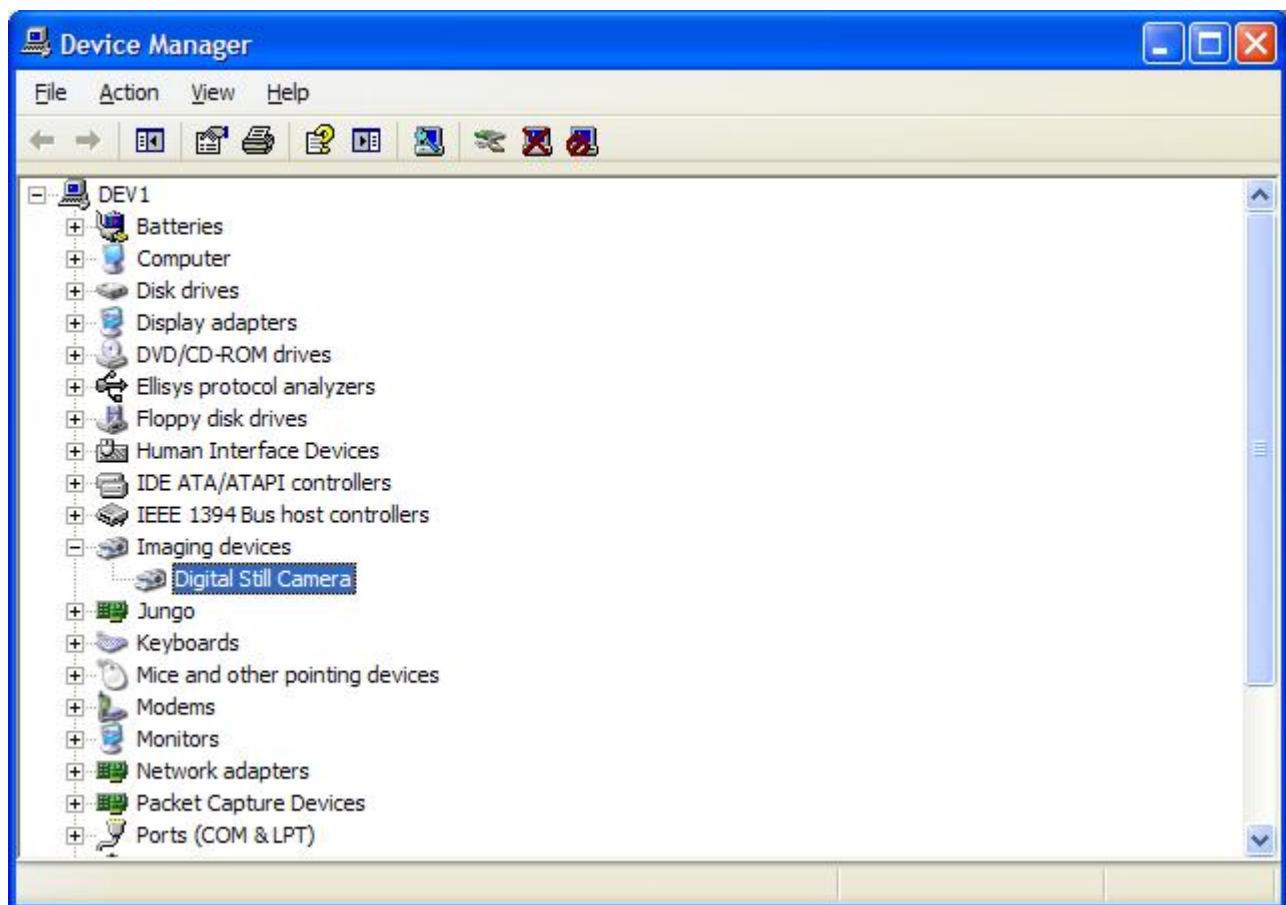


### D.3 Mass Storage

Windows XP will install the mass storage device automatically when you plug in the device (your target board). Our demo program (which emulates a flash disk using RAM on your board) automatically partitions and formats the RAM disk.

### D.4 Media Transfer Protocol (MTP)

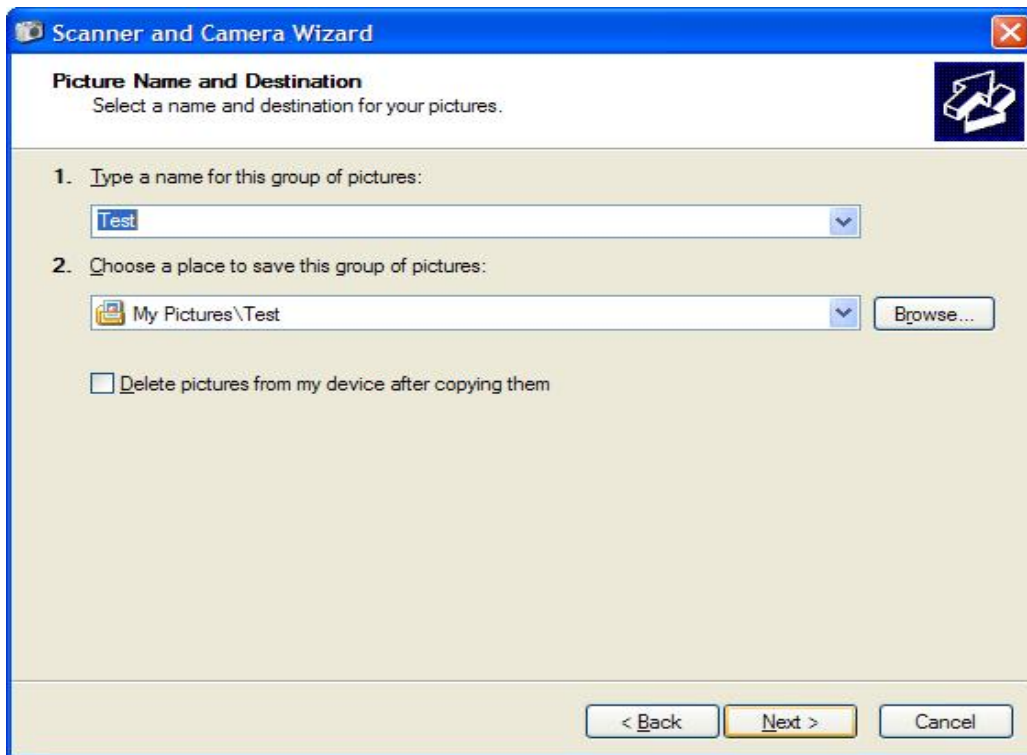
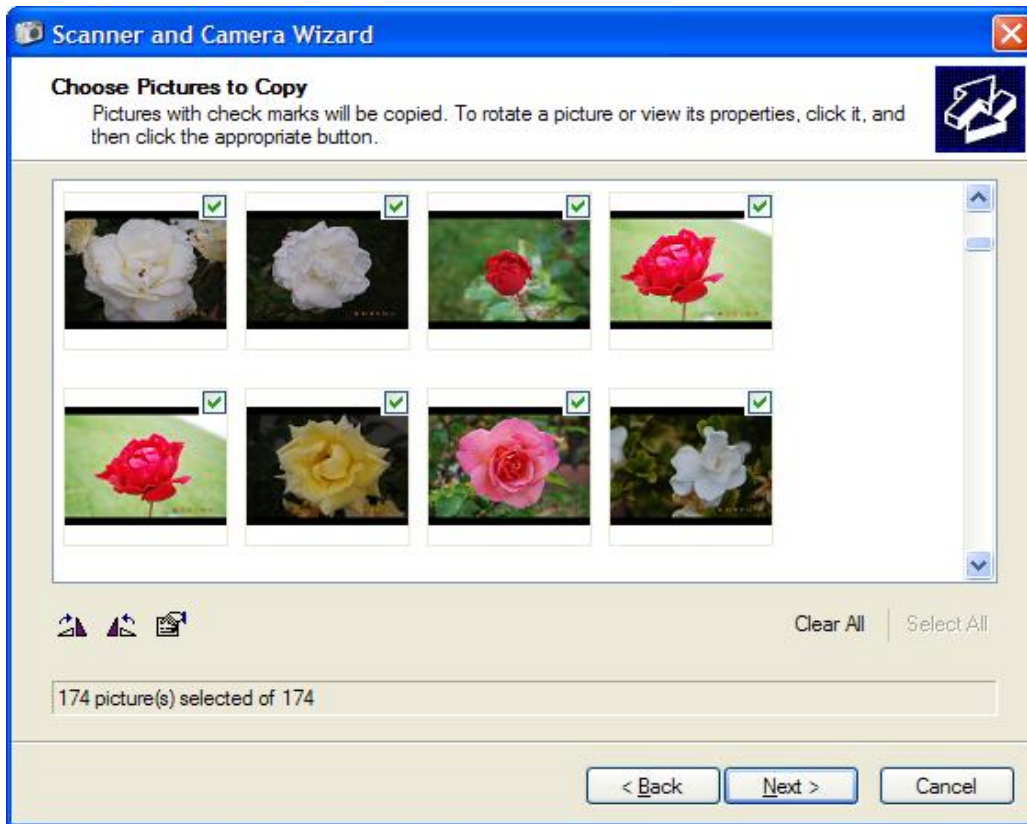
Windows XP will install the MTP device automatically when you plug in the device (your target board). After installation you can check the Digital Still Camera device in Device Manager.

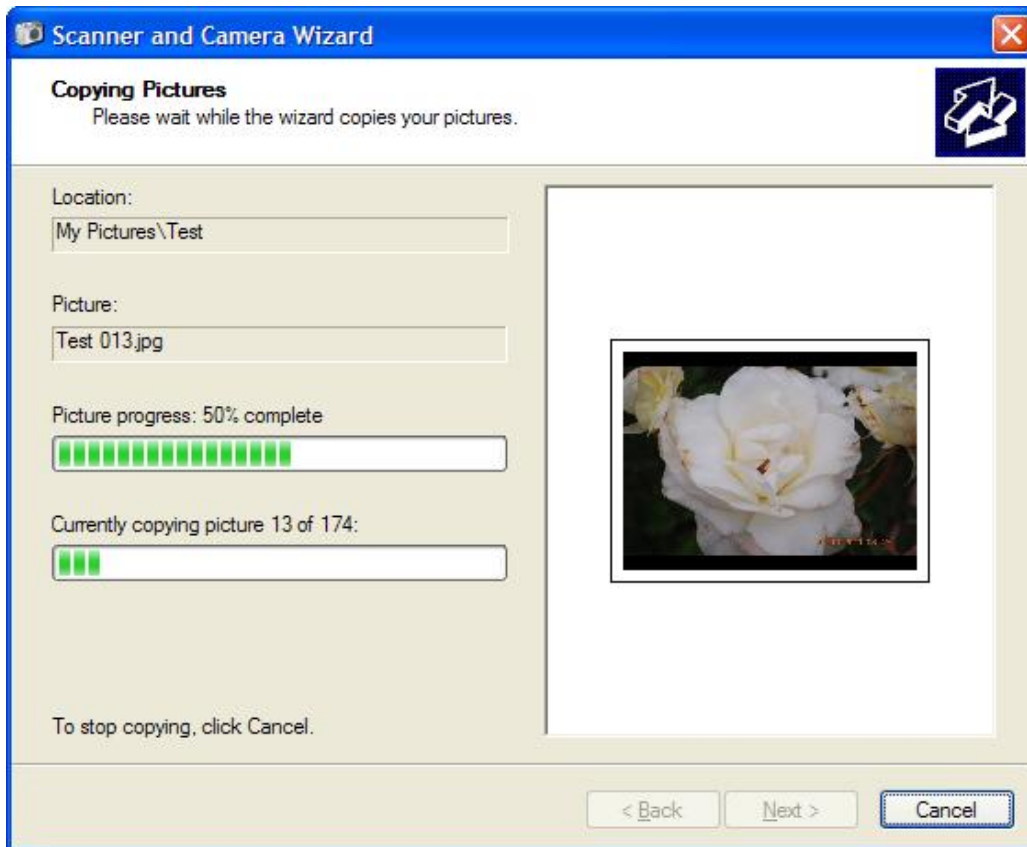


You can use Windows built-in Scanner and Camera Wizard to import the pictures from this device.

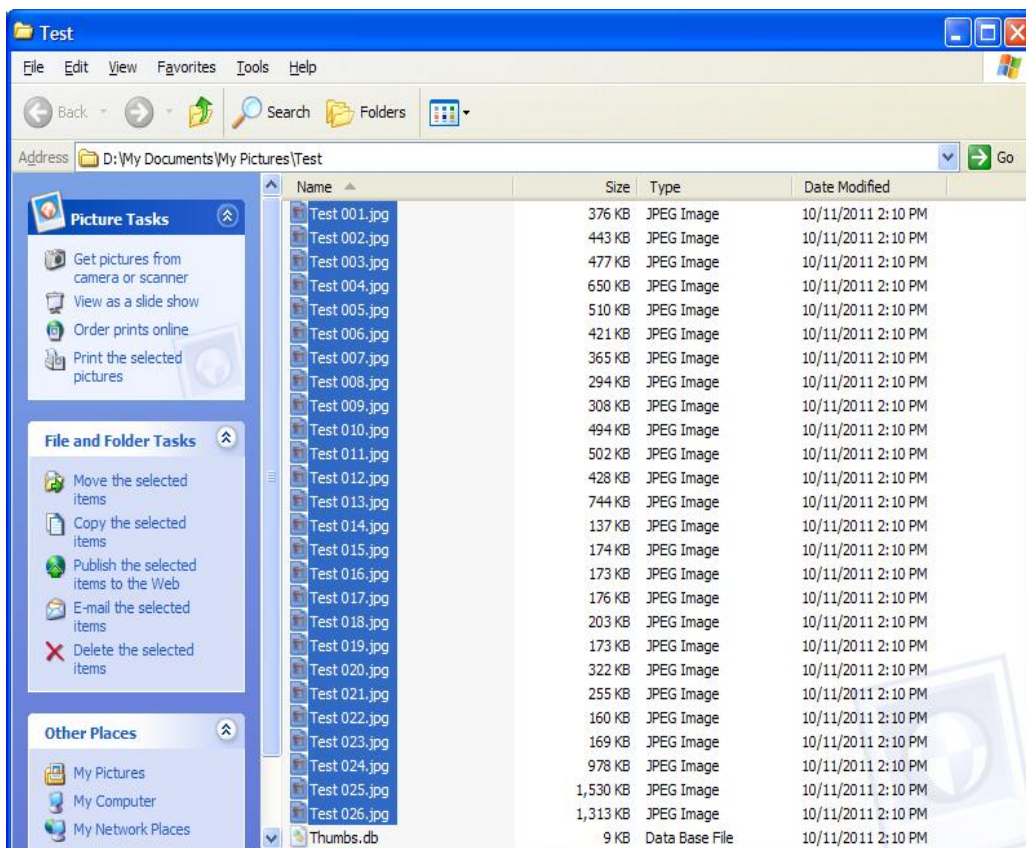
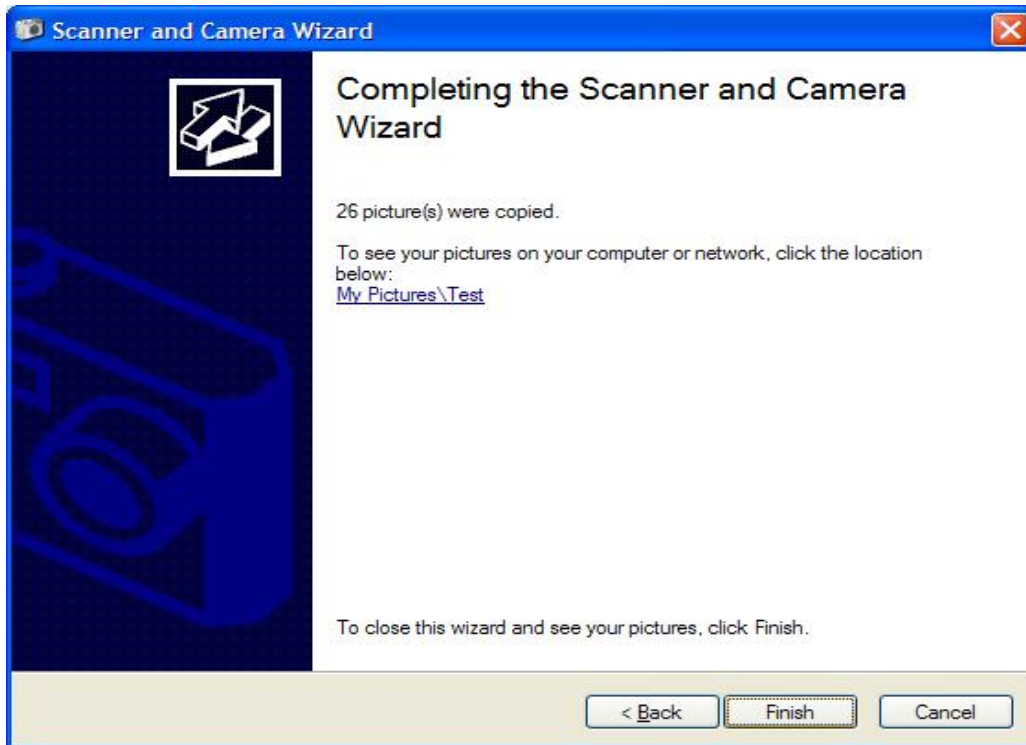




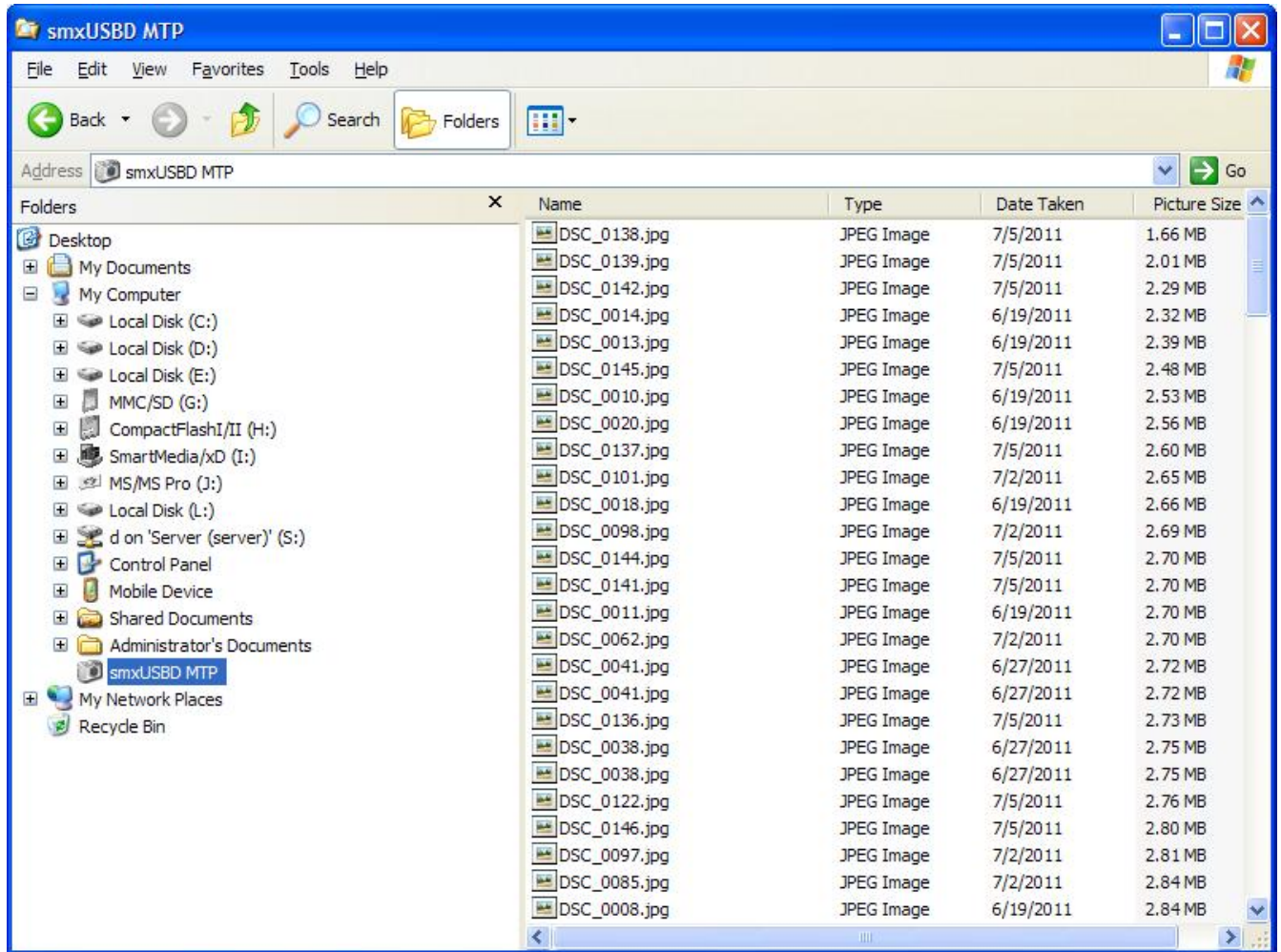








You can also use Windows Explorer to list the pictures on this device.



## D.5 Mouse/Keyboard

Windows XP will install the mouse/keyboard driver automatically when you plug in the device (your target board). No additional operations are necessary.

## D.6 Ethernet over USB

Windows XP and MacOS/Linux will install the driver automatically when you plug in the device (your target board). No additional operations are necessary.





## D.7 Serial Port

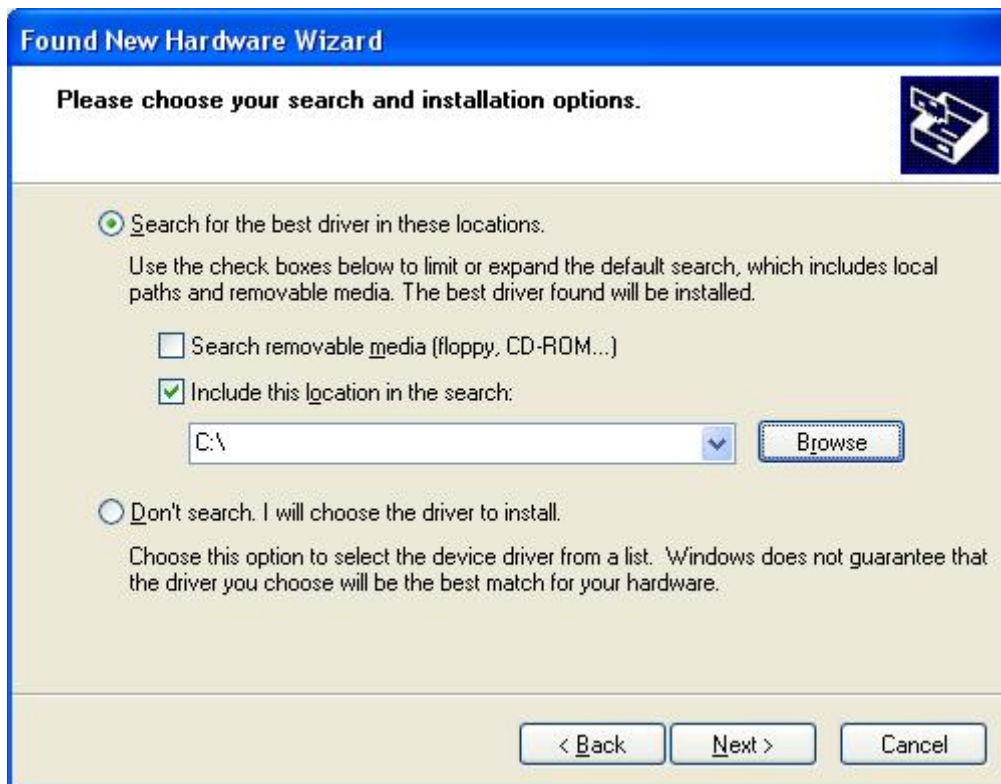
To install the serial port driver, you may need an .inf file. Several are provided in the XUSBDFunction\Serial and SerialM directories for different cases (single port, multi-port, composite, and whether it uses the Windows built-in driver or the Micro Digital driver. The files in XUSBDFunction\Serial are for the Windows driver; the files in XUSBDFunction\SerialM are for the Micro Digital Driver. See section 9.1 Multiple Port Serial Device (or Single Port Limited Endpoints) for more information. Note that for Windows XP versions before SP3, multi-port serial requires the Micro Digital driver.

Windows XP will pop up a dialog when you plug in your target, to inform you it has found new hardware.

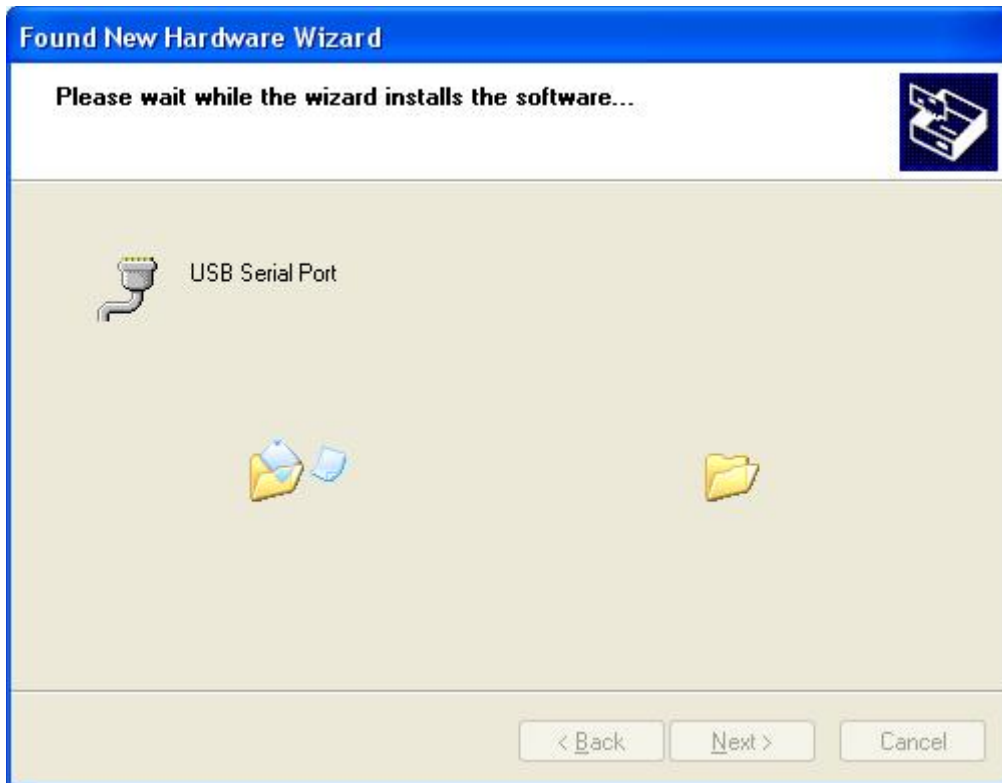




You may need to copy the .inf file to a temporary directory, such as C:\



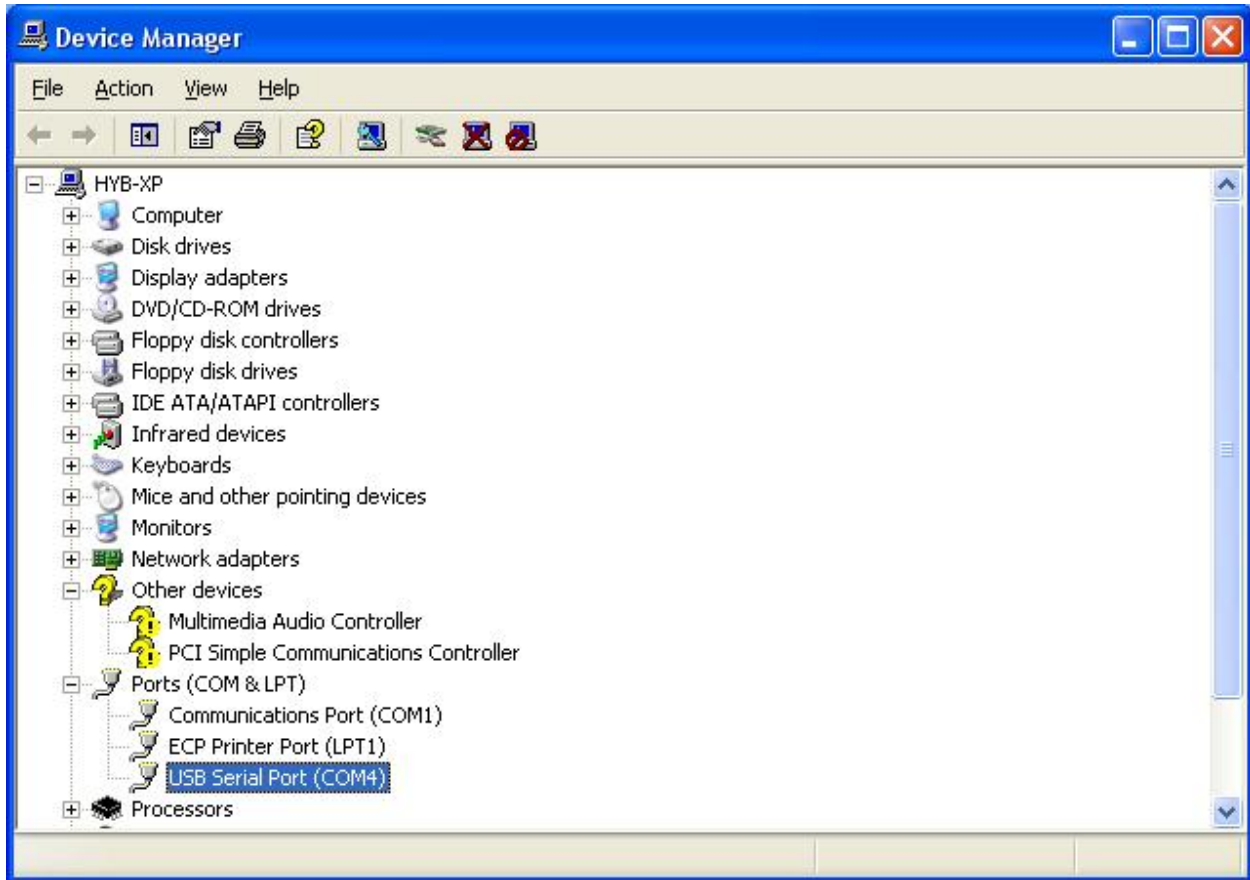
You can ignore the following warning dialog (see Appendix H. Host OS Certification for more information):



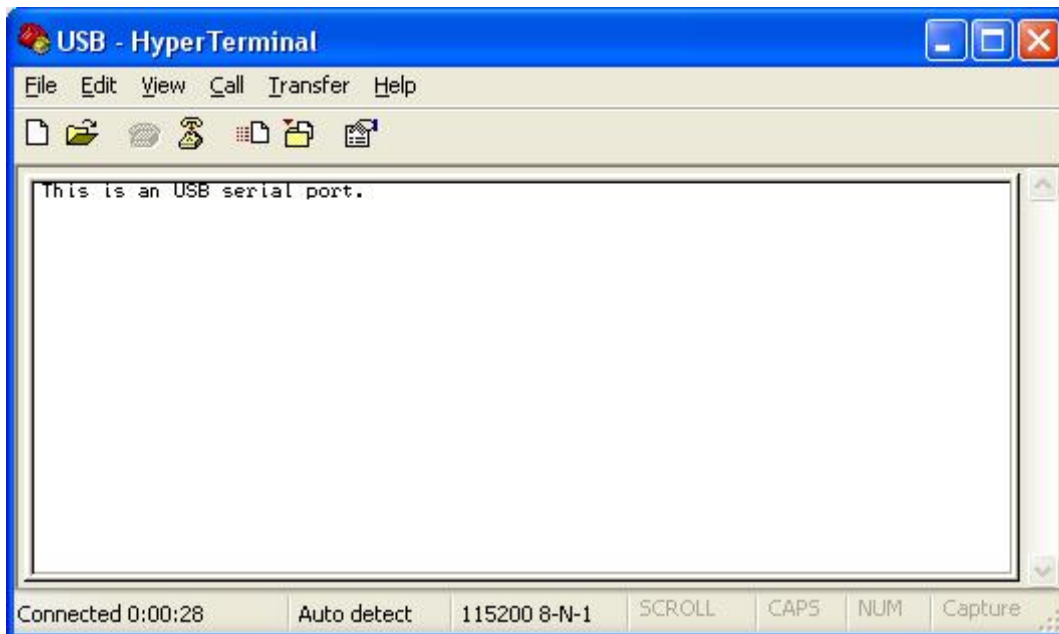
Windows will search for the driver `usbser.sys`. It may prompt you to insert your Windows installation CD if this file is not in your Driver Cache or Service Pack .cab file.



In the Device Manager, you will see a new COM port has been added:

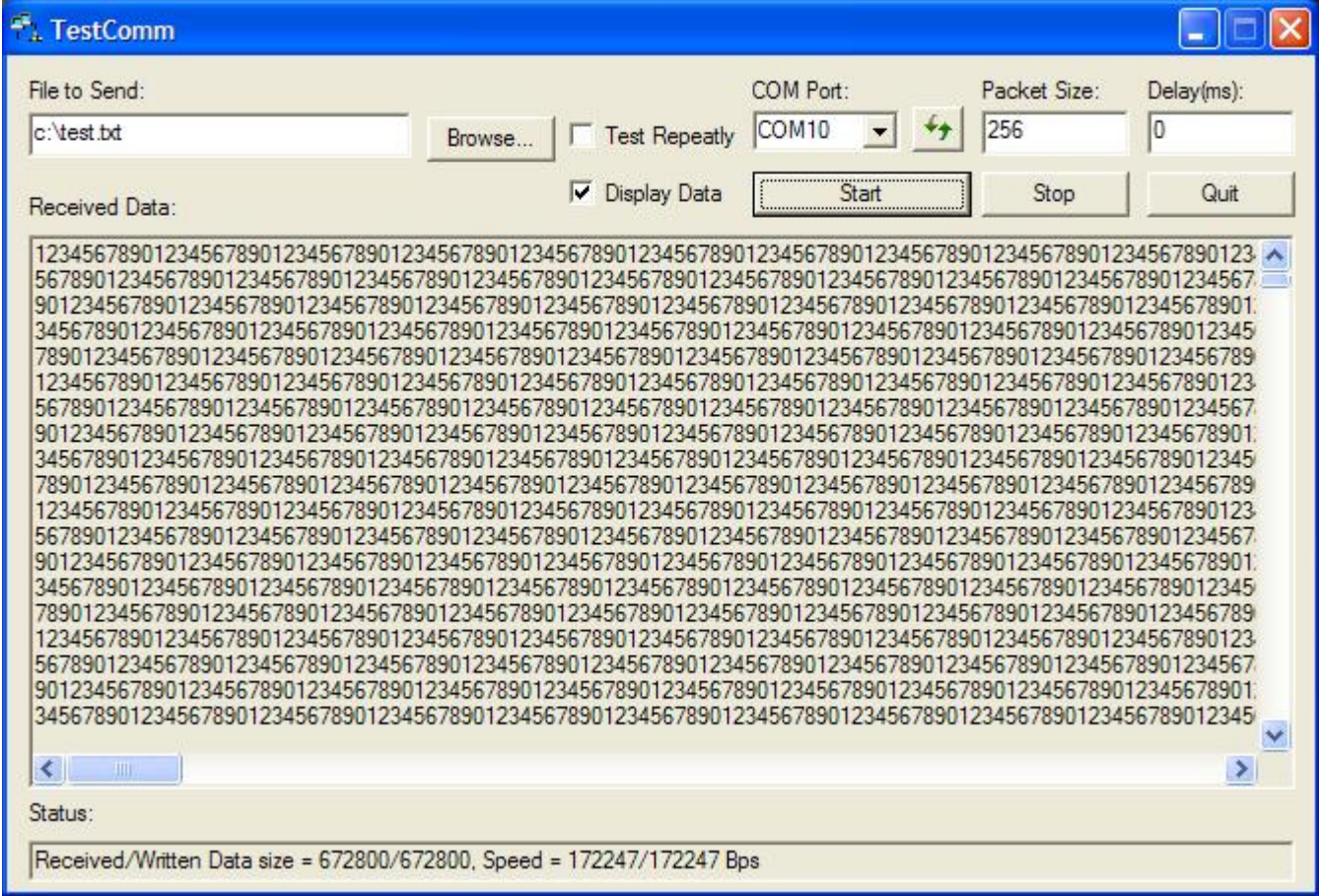


You can use HyperTerminal to test if your serial port emulator works properly.



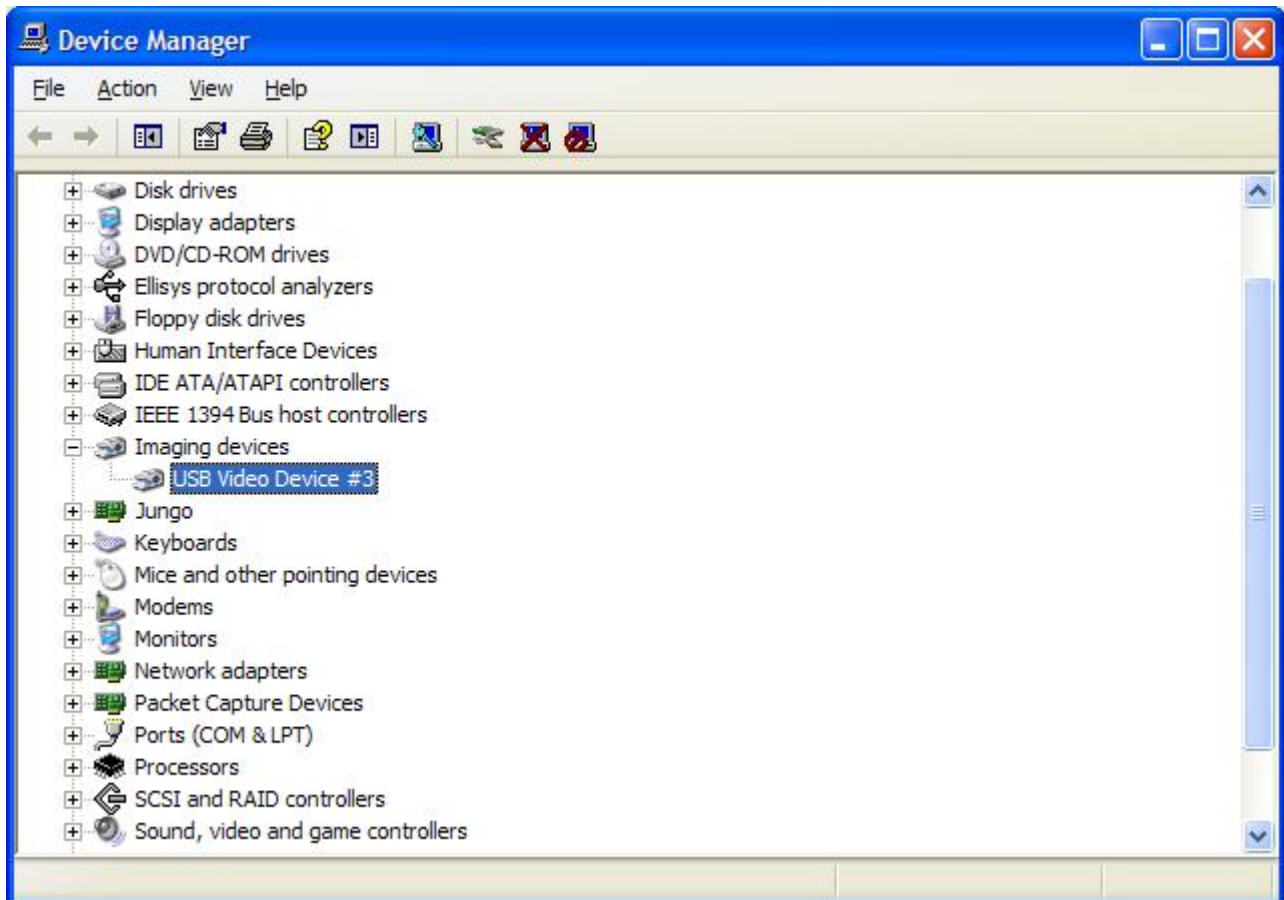


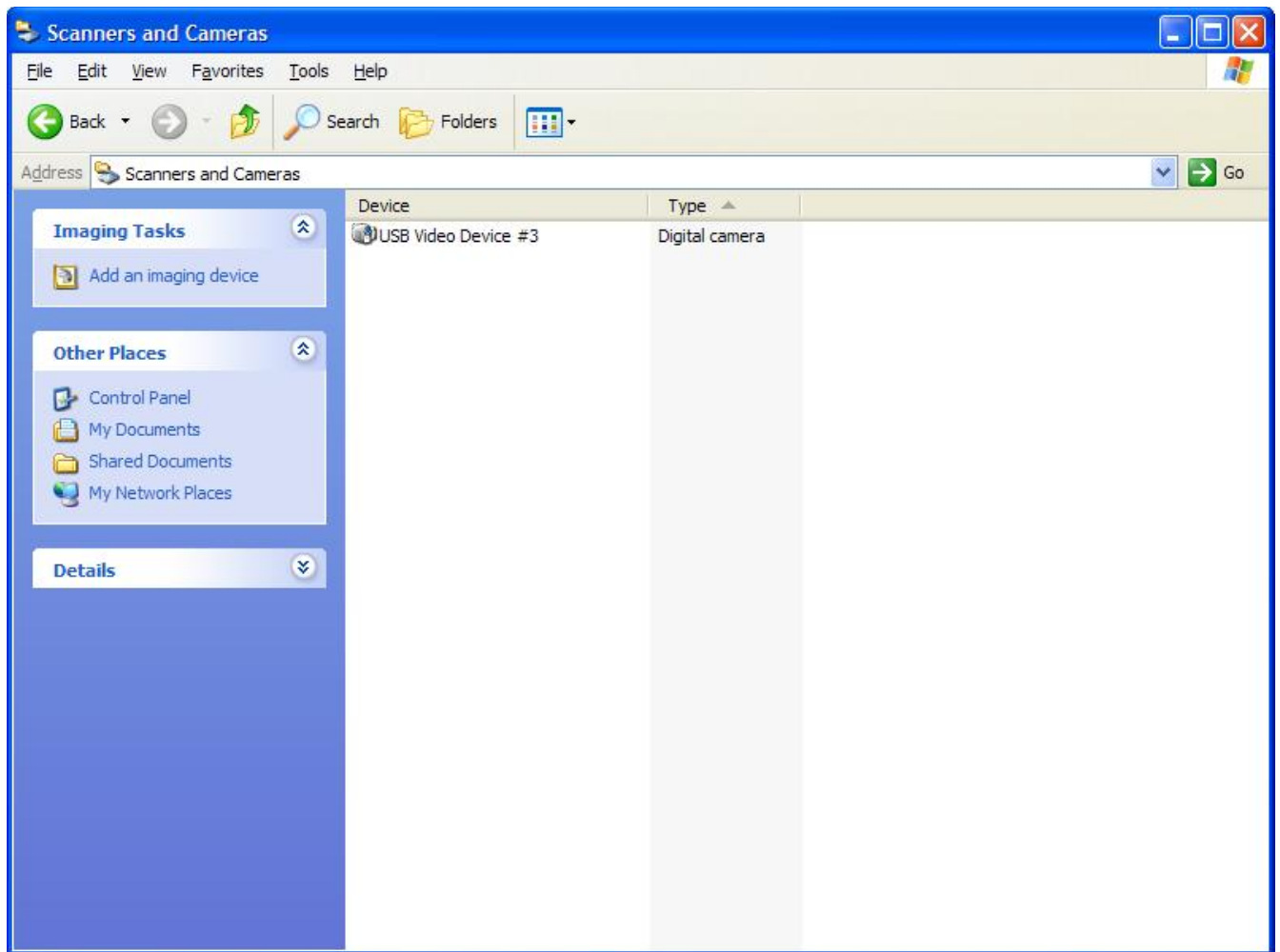
Or you can use our TestComm utility to do the performance and stress testing. It is in the BIN directory. TestComm initially selects the highest COM port because that is the most likely one to be for the smxUSB target just plugged in. If you started TestComm before plugging in the USB cable, click the refresh button to update the COM port list.



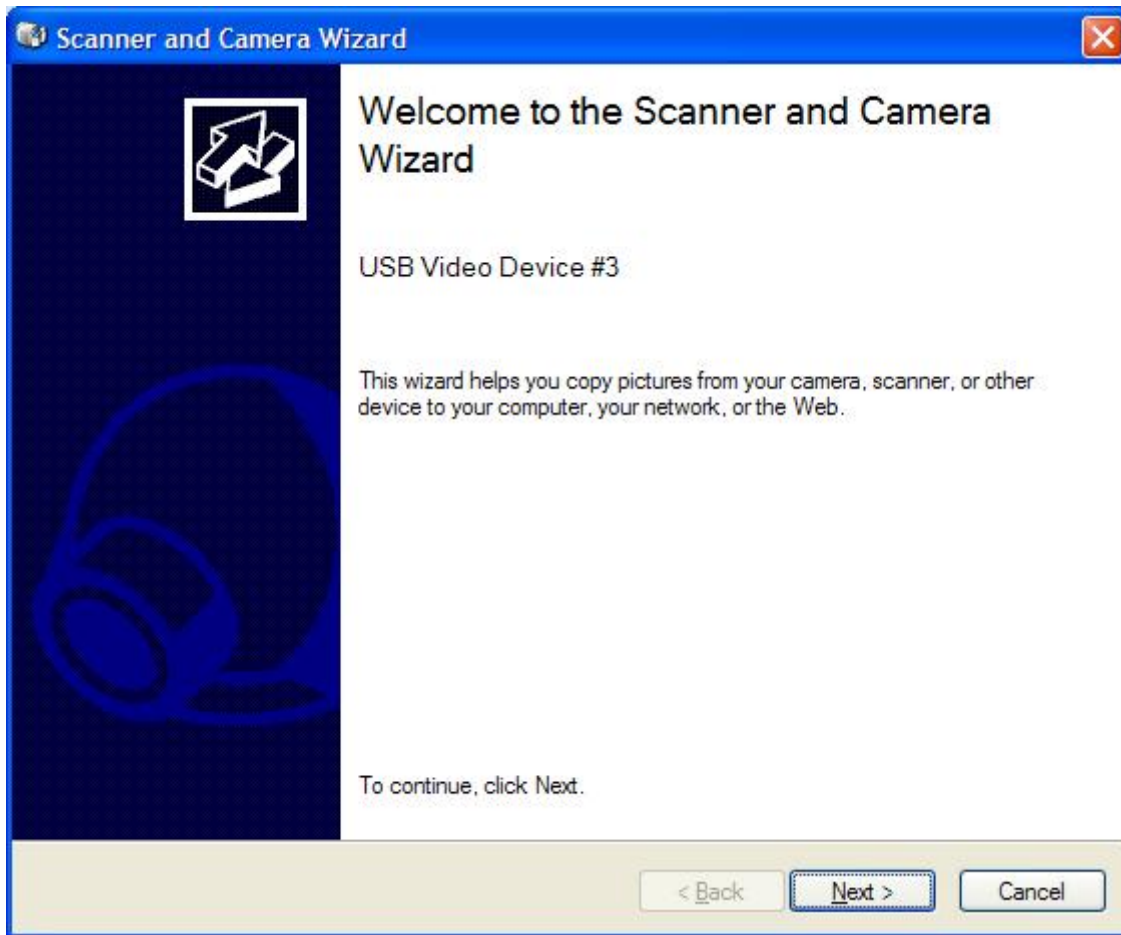
## D.8 Video

Windows XP will install the video device automatically when you plug in the device (your target board). After installation you can check the video device in Device Manager.

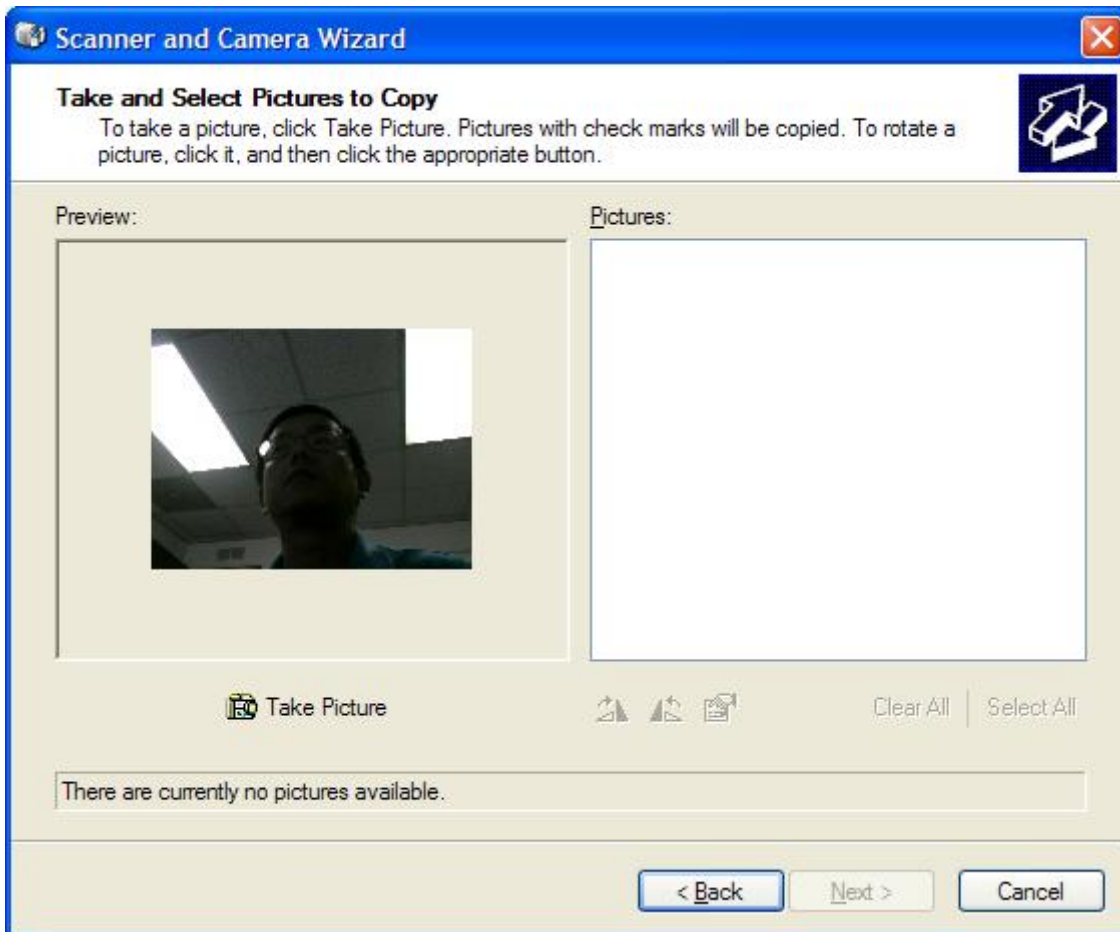




If you double click the USB Video Device, the Scanner and Camera Wizard will pop up.



Click Next and you will see the video (160x120).





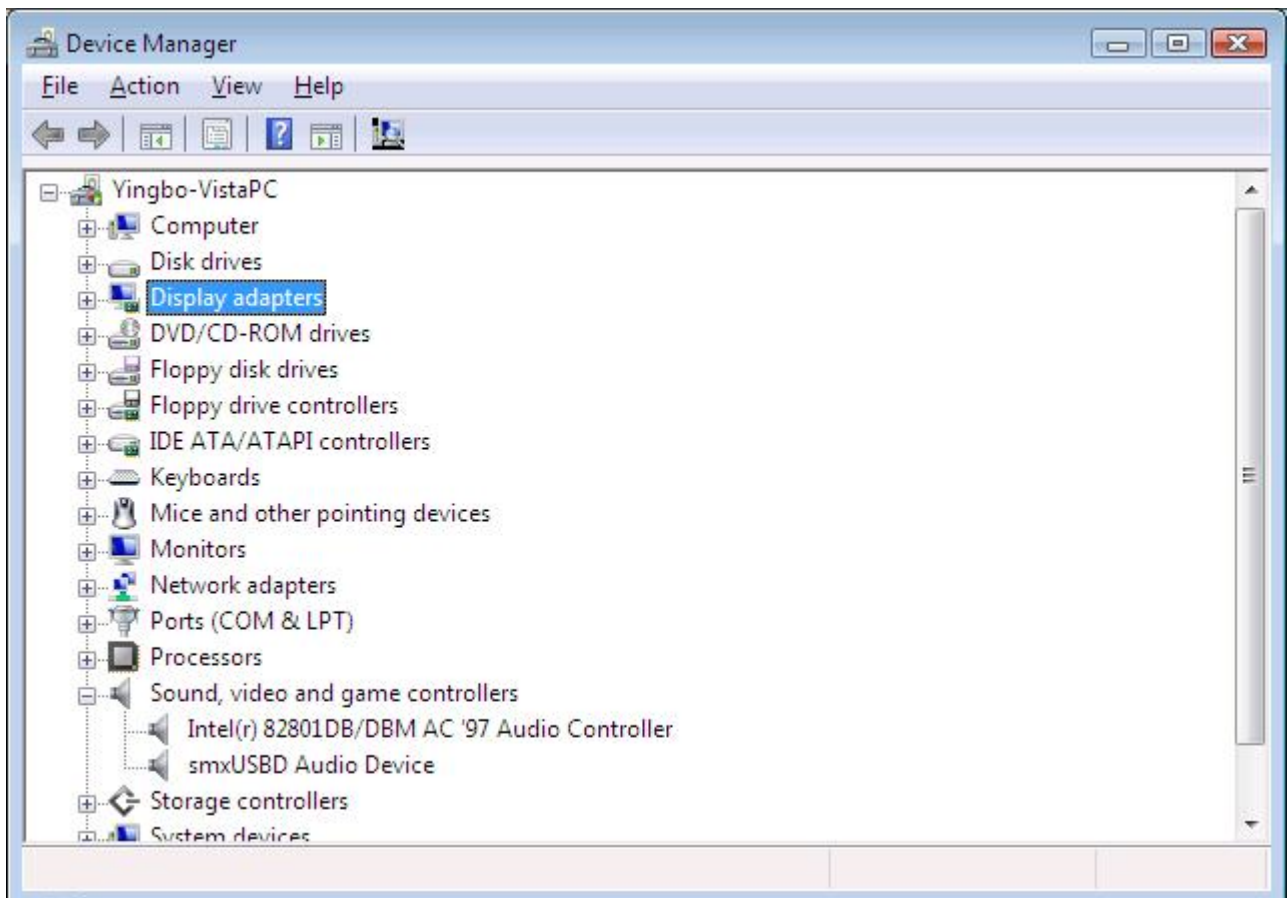
## Appendix E. Installing Devices under Windows Vista, 7, and 8

smxUSBD devices are supported by Windows Vista and Windows 7 built-in drivers. The following screen shots (from Vista) show the Windows installation steps, which are the same for both versions of Windows. (The shorthand Vista/7 is used below to mean both.) If it doesn't work after installing the device driver, try ending the target debug session and starting a new one. If that doesn't work, try restarting Windows. Contact Micro Digital if you still have trouble.

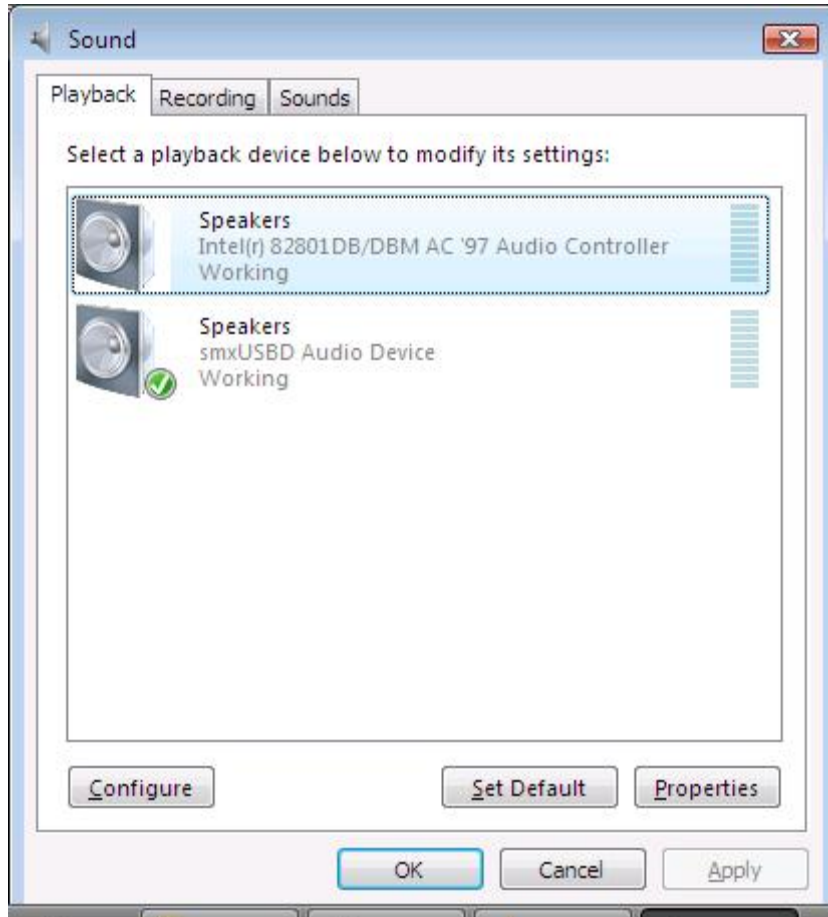
First see section 3.3 Building and Running the Demos before running the demos.

### E.1 Audio

Windows Vista/7 will install the audio device automatically when you plug in the device (your target board). After installation you can check the audio device in Device Manager.



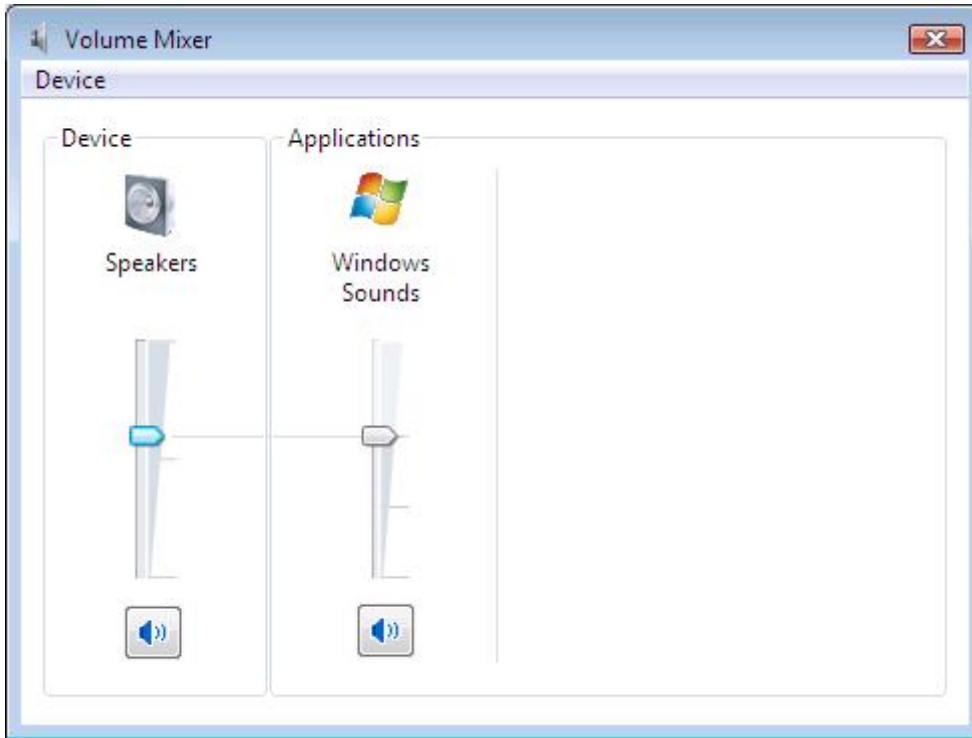
You can also check the device in Control Panel | Sounds | Playback and Recording







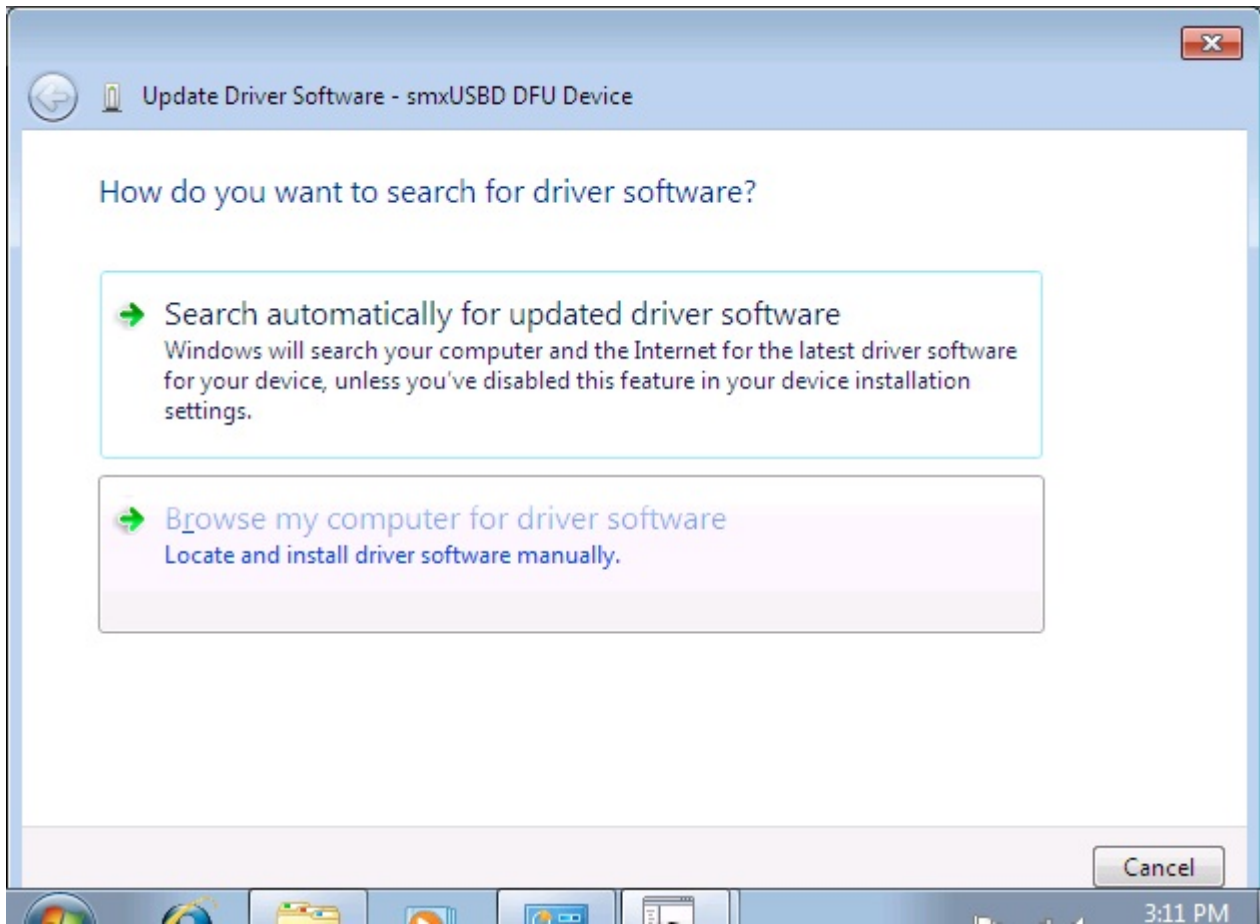
Speaker volume control (Mixer):

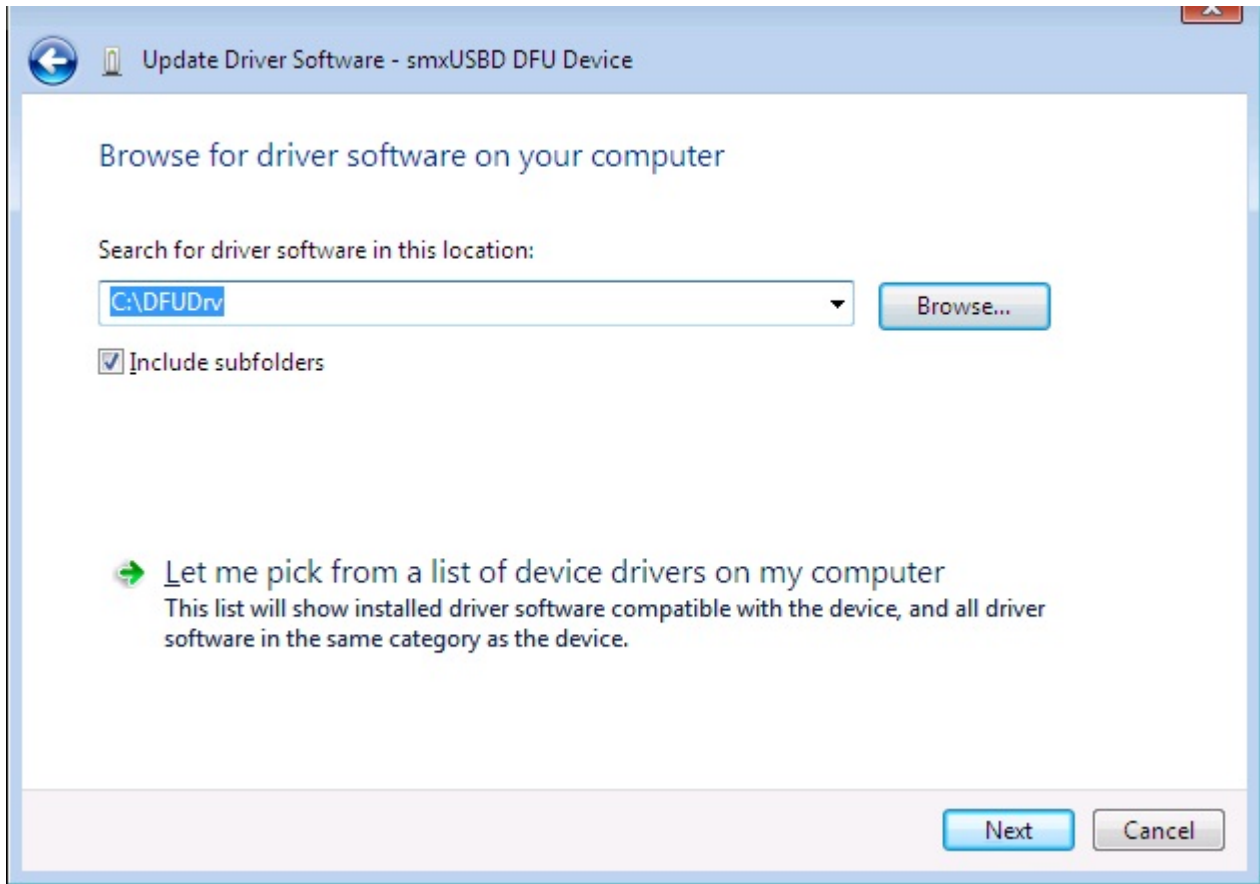


## E.2 Device Firmware Upgrade (DFU)

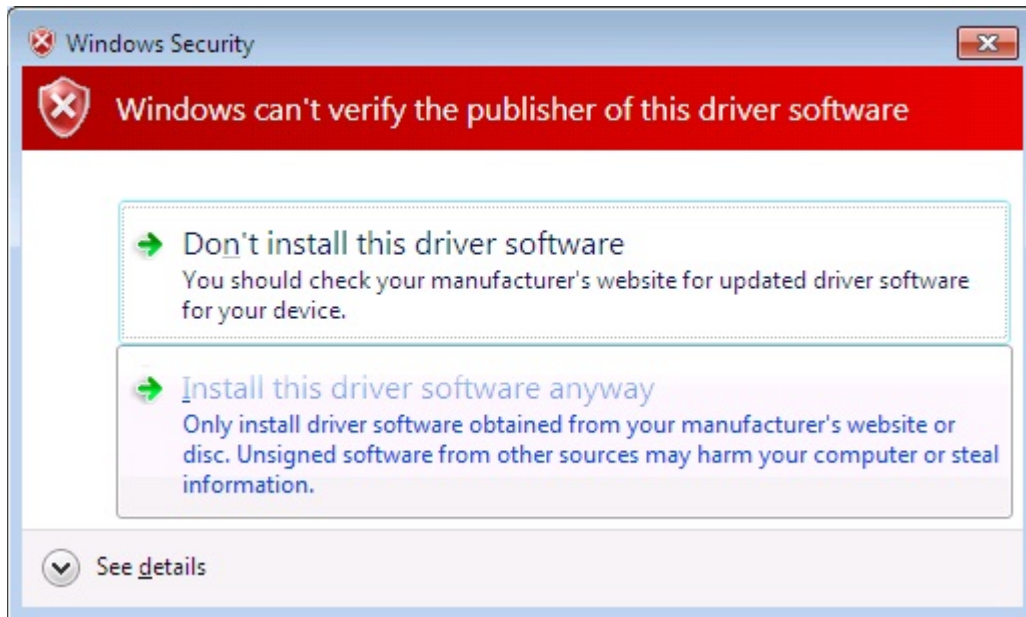
Windows Vista/7 has no built-in driver and utility for the DFU device. You need to install the driver provided by MDI or another. See section 9.2 Device Firmware Upgrade (DFU) Device.

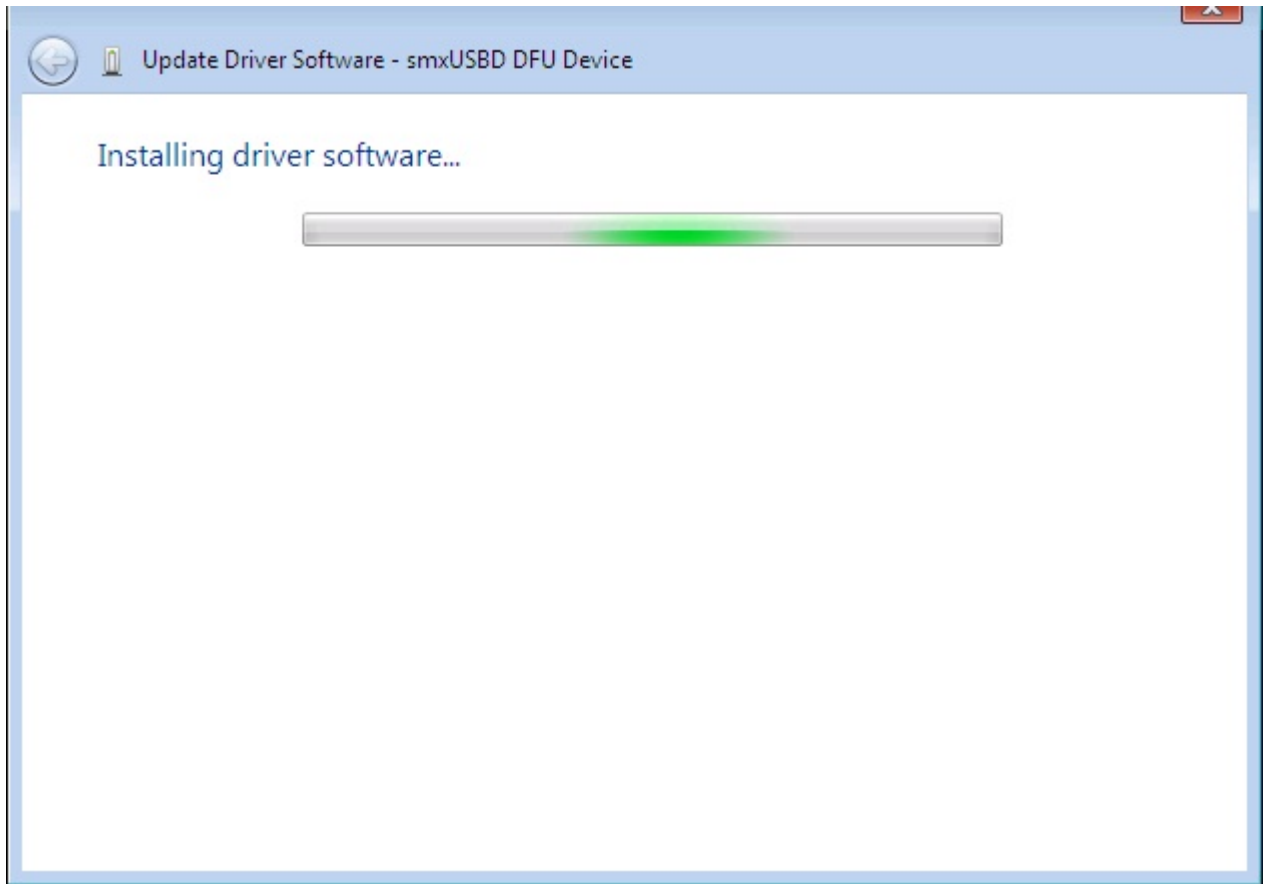
A DFU device has two modes. One is the runtime mode, which shows the normal function and DFU ability. The other is DFU mode, which is for firmware upgrade only.

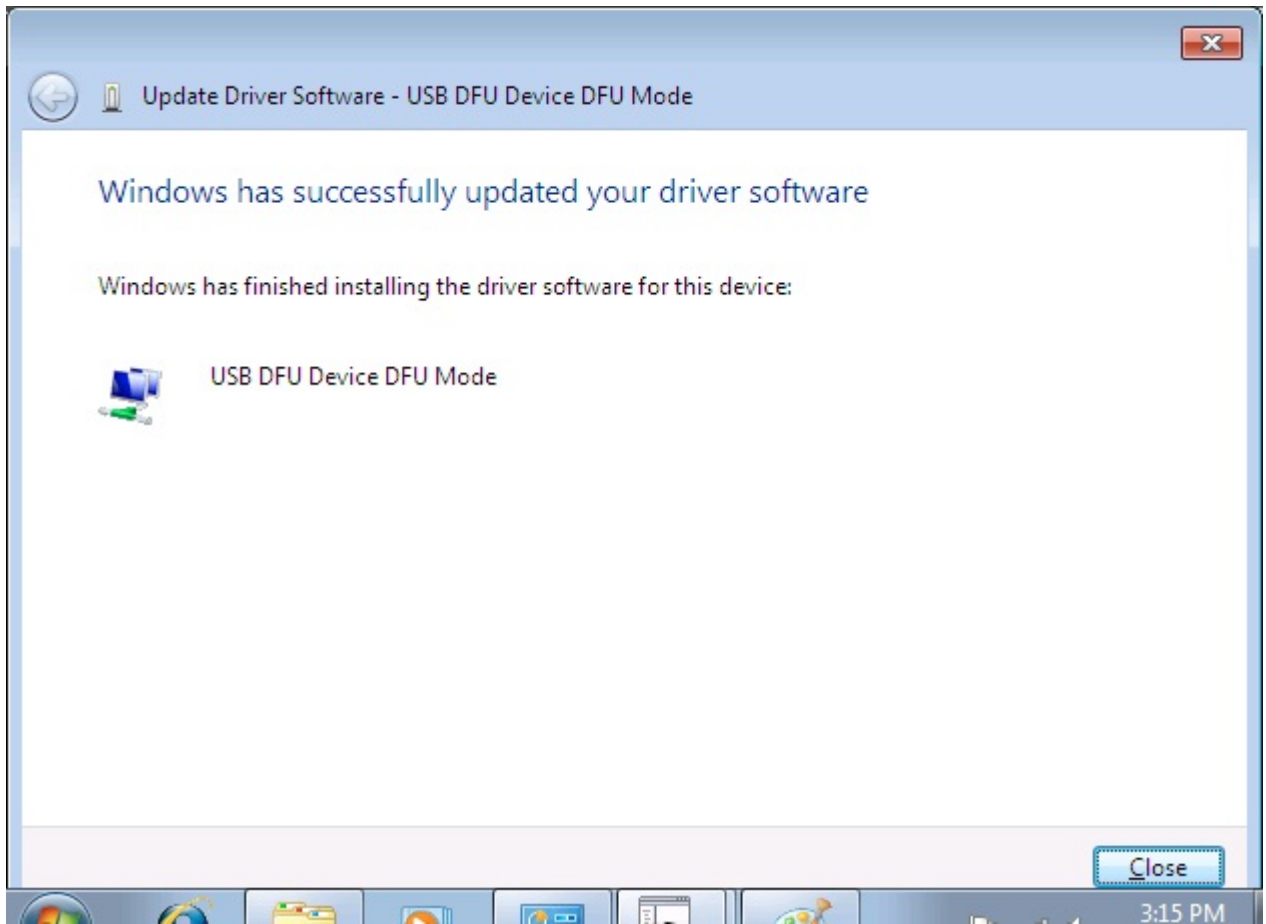


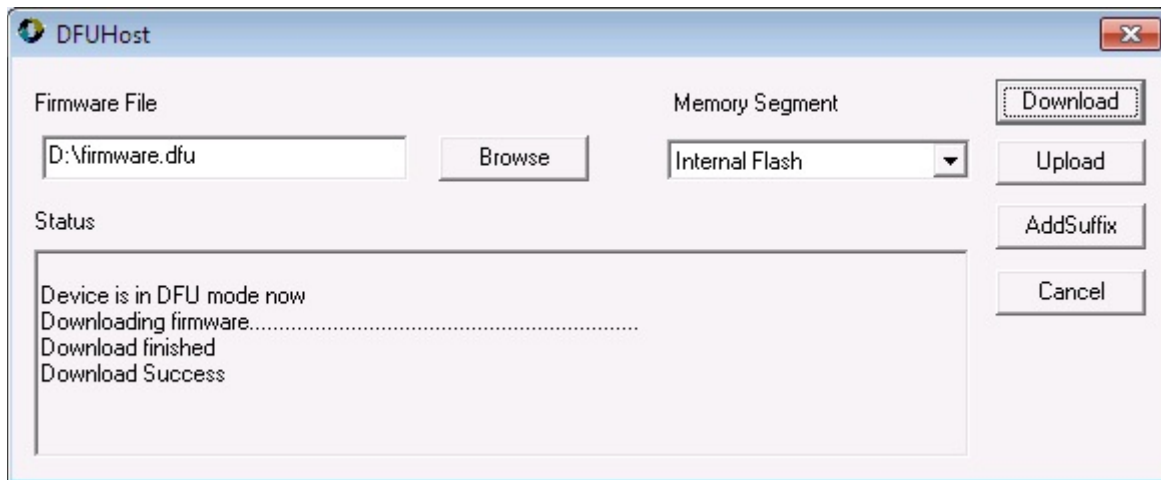
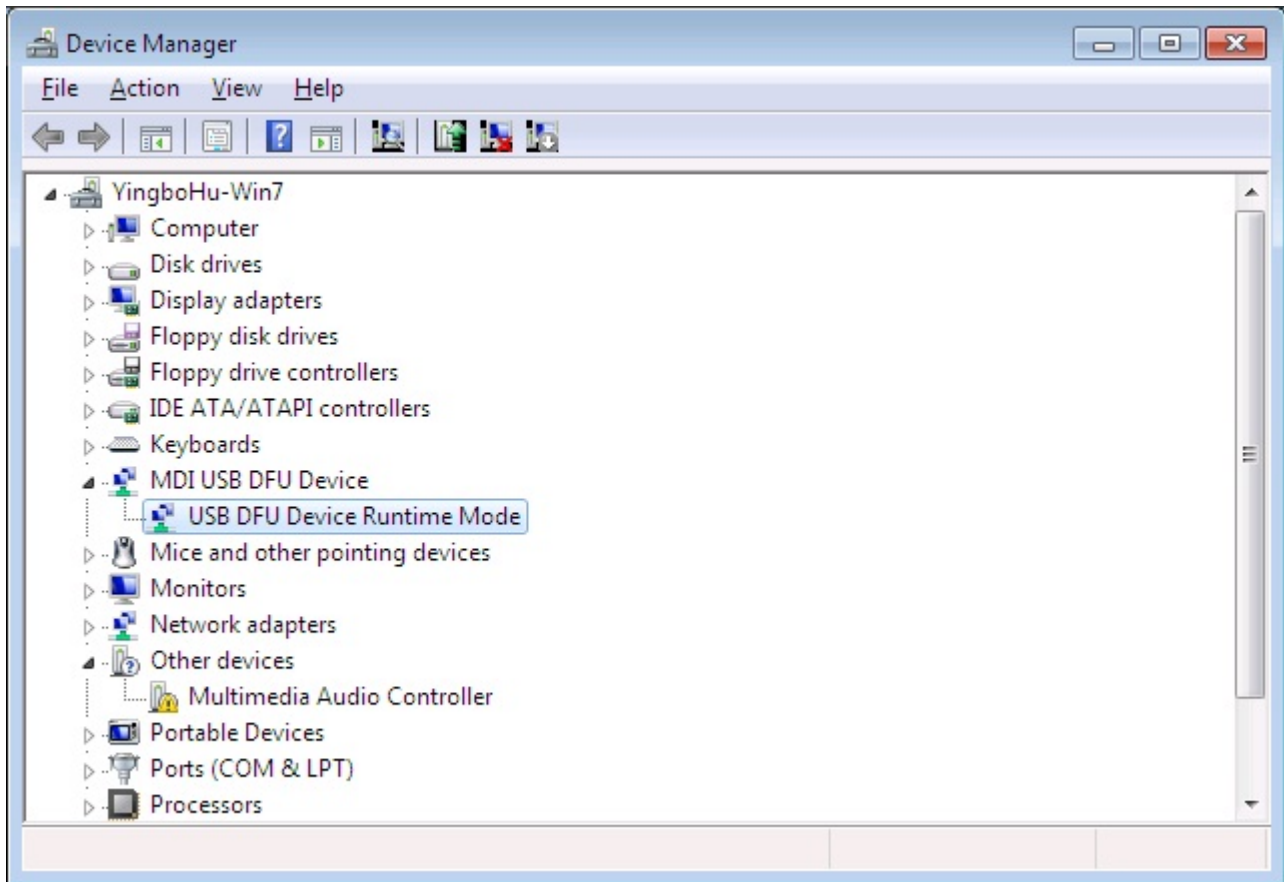


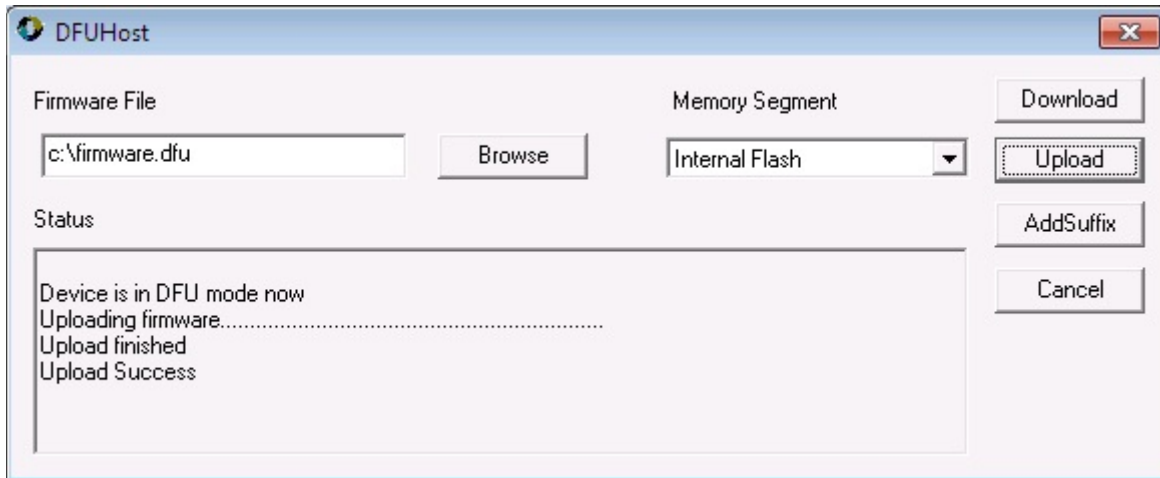
Ignore this warning dialog









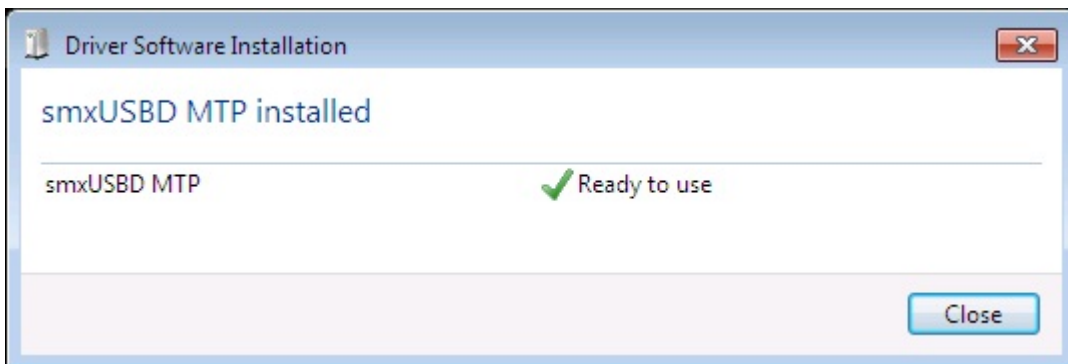


### E.3 Mass Storage

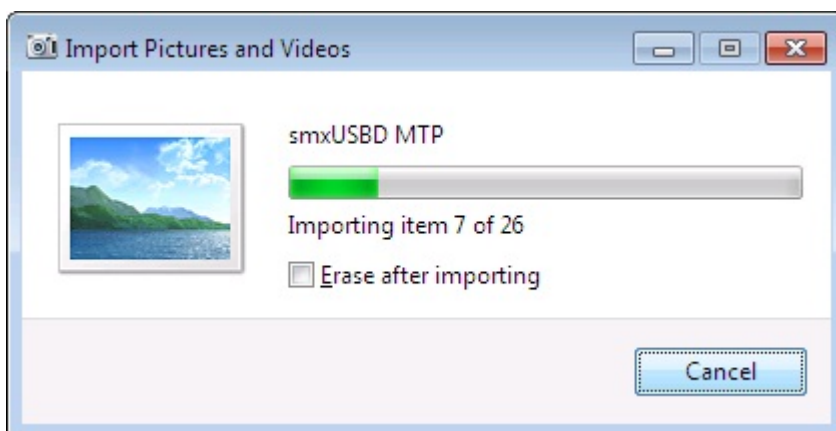
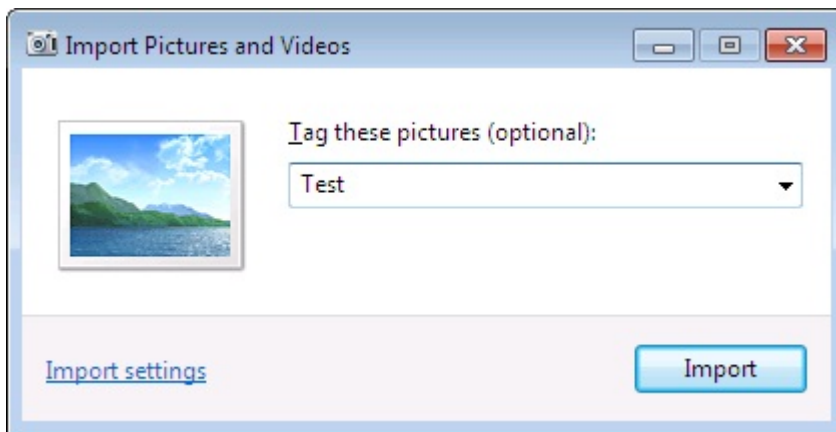
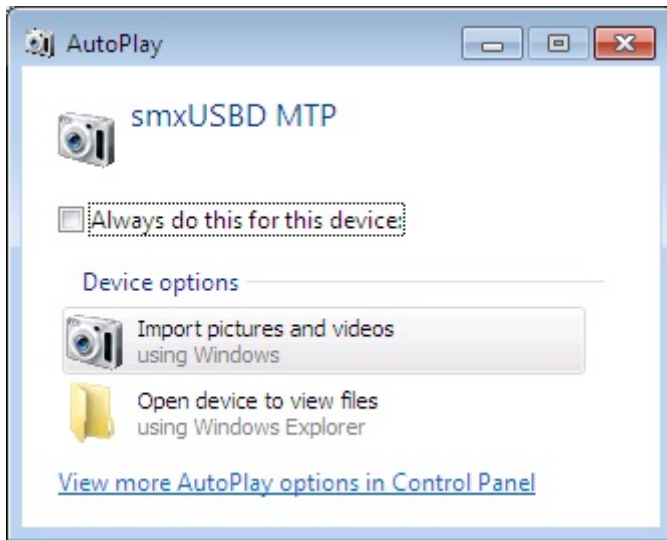
Windows Vista/7 will install the mass storage device automatically when you plug in the device (your target board). Our demo program (which emulates a flash disk using RAM on your board) automatically partitions and formats the RAM disk.

### E.4 Media Transfer Protocol (MTP)

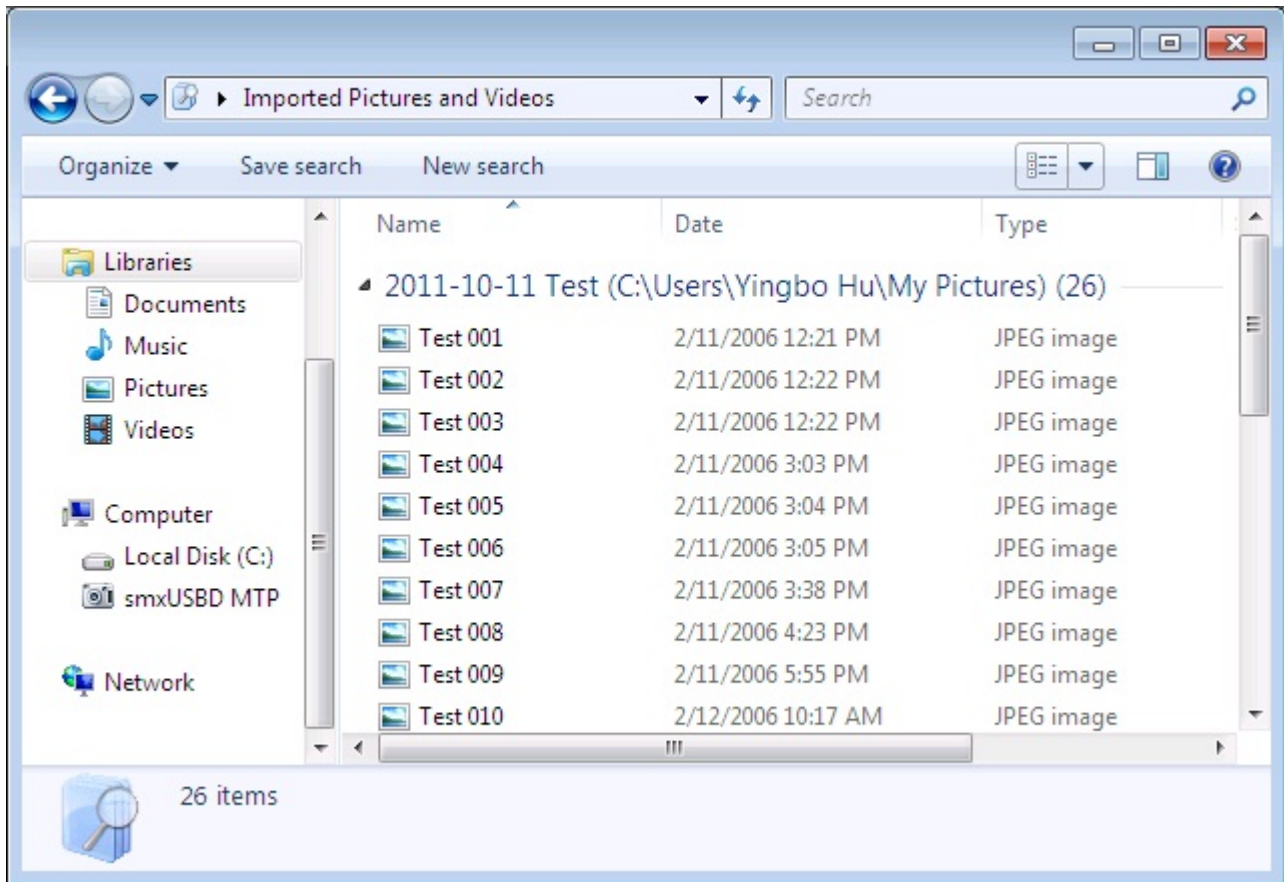
Windows Vista/7 will install the MTP device automatically when you plug in the device (your target board). After installation you can check the MTP device in Device Manager.



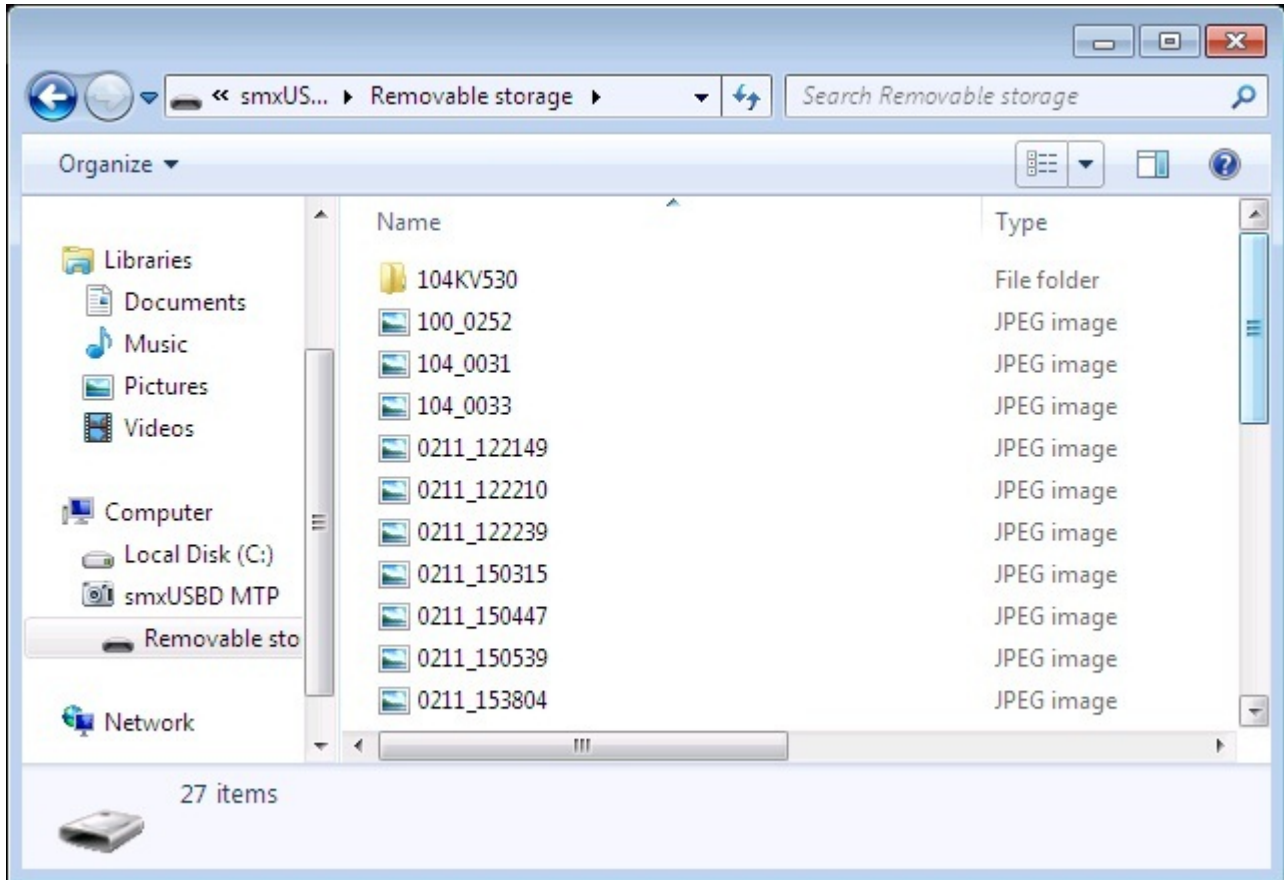
You can use the Windows built-in utility to import the pictures and videos:







You can also use Windows Explorer to list the files on this MTP device.



## E.5 Mouse/Keyboard

Windows Vista/7 will install the mouse/keyboard driver automatically when you plug in the device (your target board). No additional operations are necessary.

## E.6 Ethernet over USB

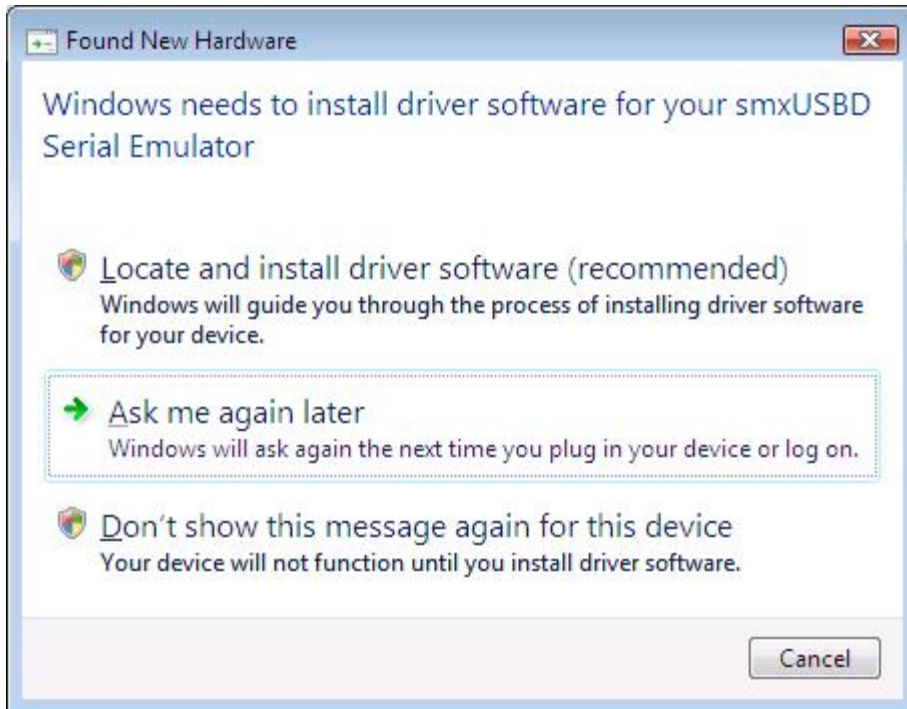
Windows Vista/7 and MacOS/Linux will install the driver automatically when you plug in the device (your target board). No additional operations are necessary.

## E.7 Serial Port

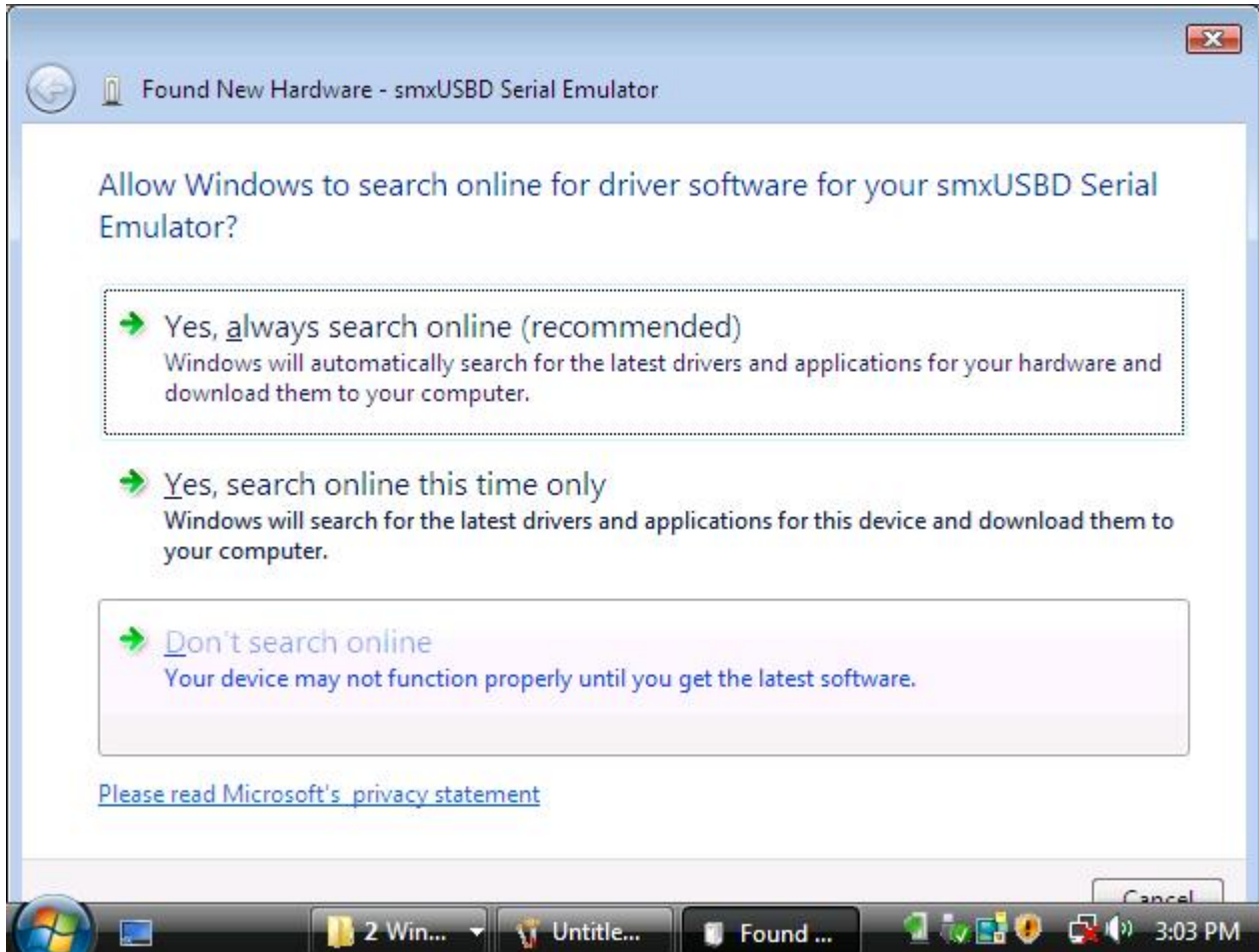
To install the serial port driver, you may need an .inf file. Several are provided in the XUSBDFunction\Serial and SerialM directories for different cases (single port, multi-port, composite, and whether it uses the Windows built-in driver or the Micro Digital driver. The files in XUSBDFunction\Serial are for the Windows driver; the files in XUSBDFunction\SerialM are for the

Micro Digital Driver. See section 9.1 Multiple Port Serial Device (or Single Port Limited Endpoints) for more information.

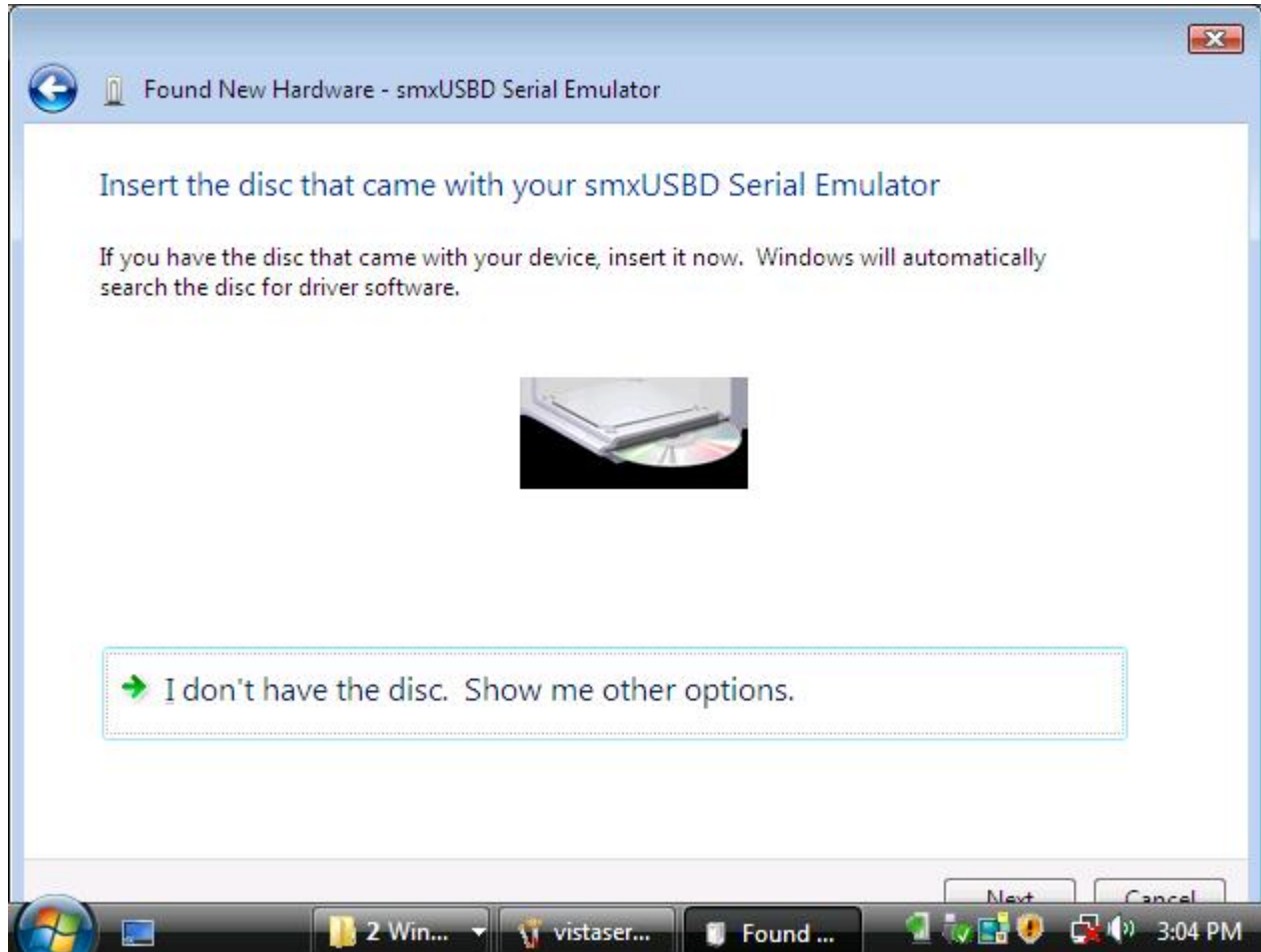
Windows Vista/7 will pop up a dialog when you plug in your target, to inform you it has found new hardware.



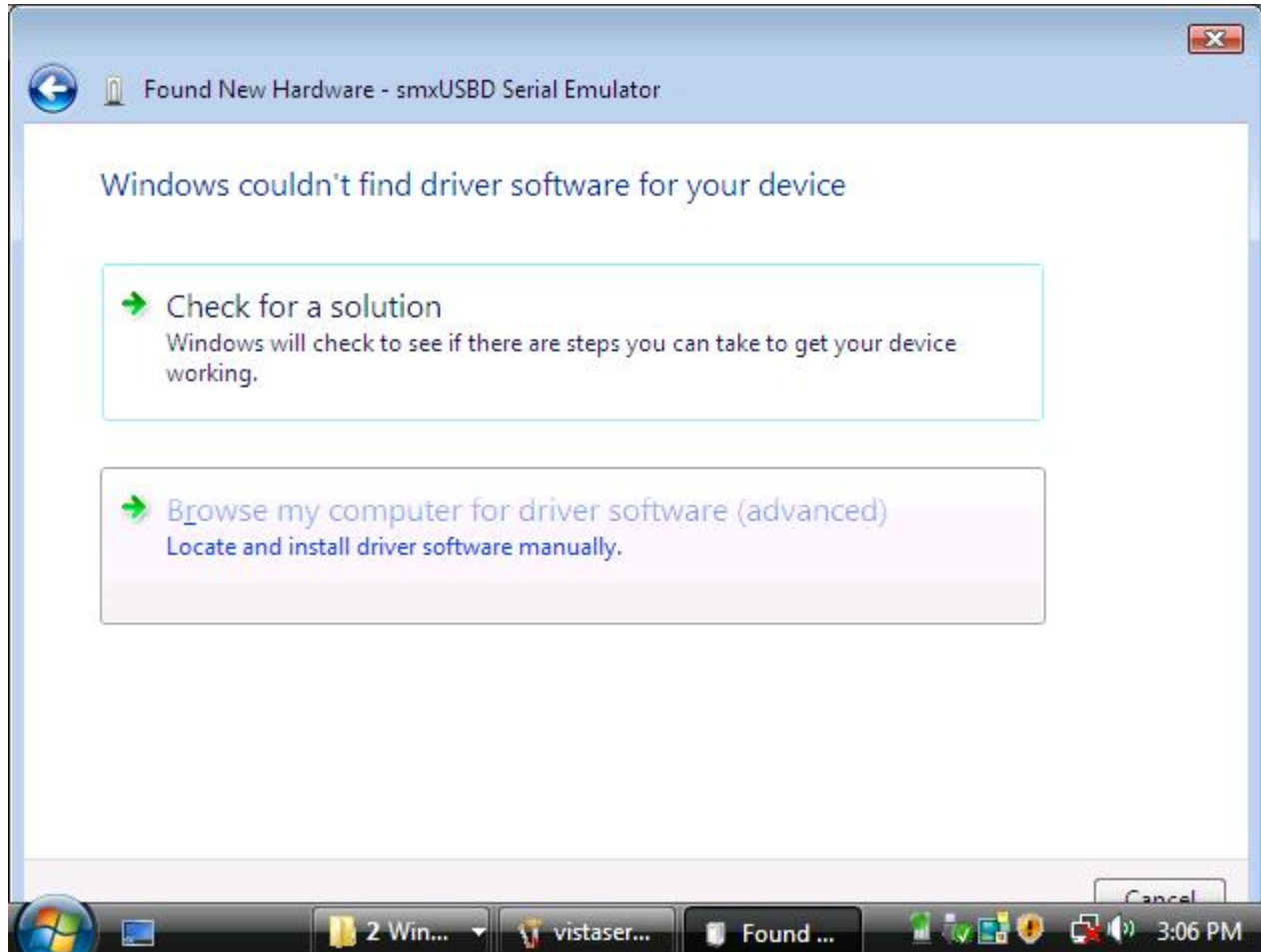
Select "Don't search online"



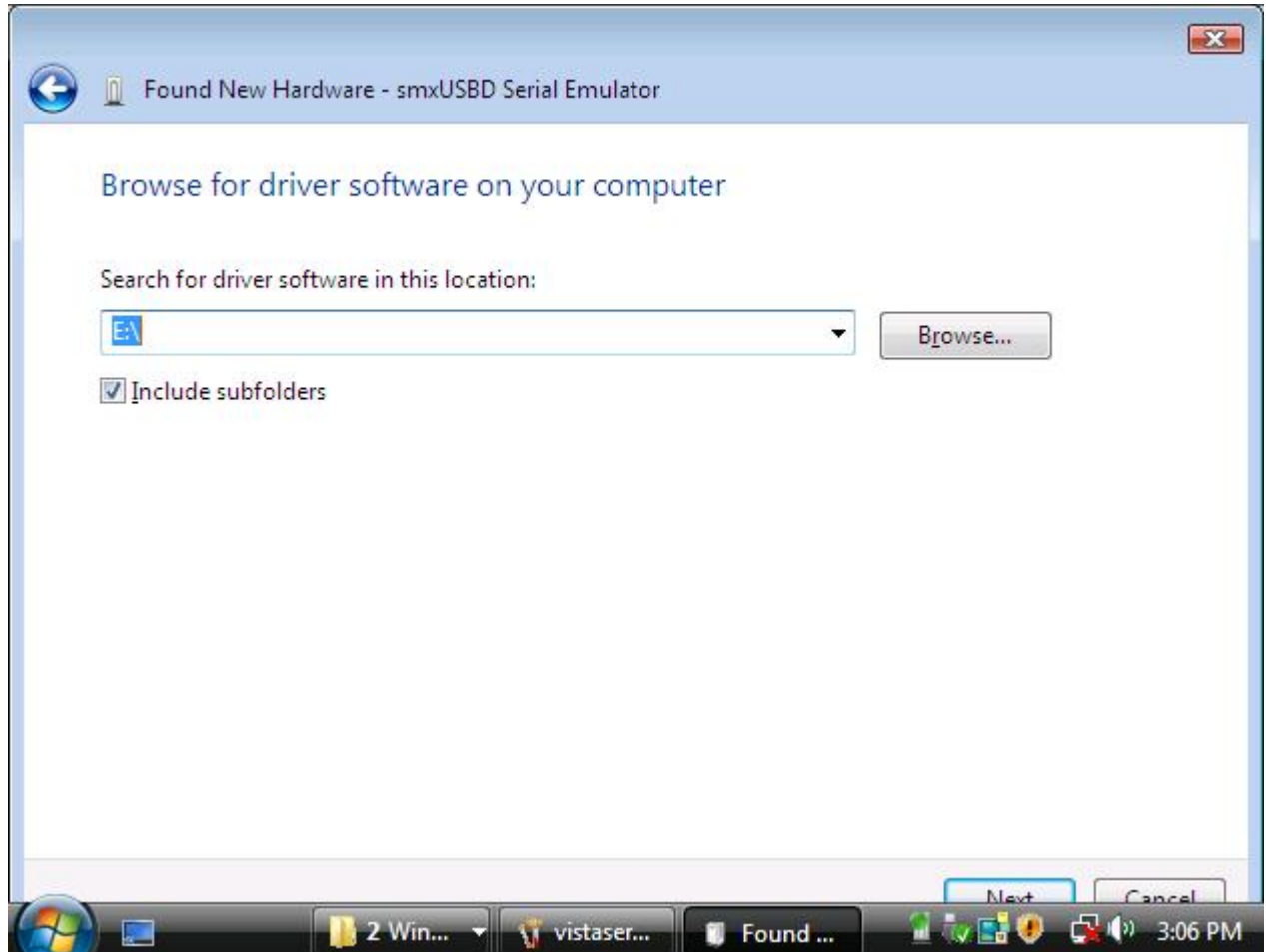
Select "I don't have a disc, show me other options"



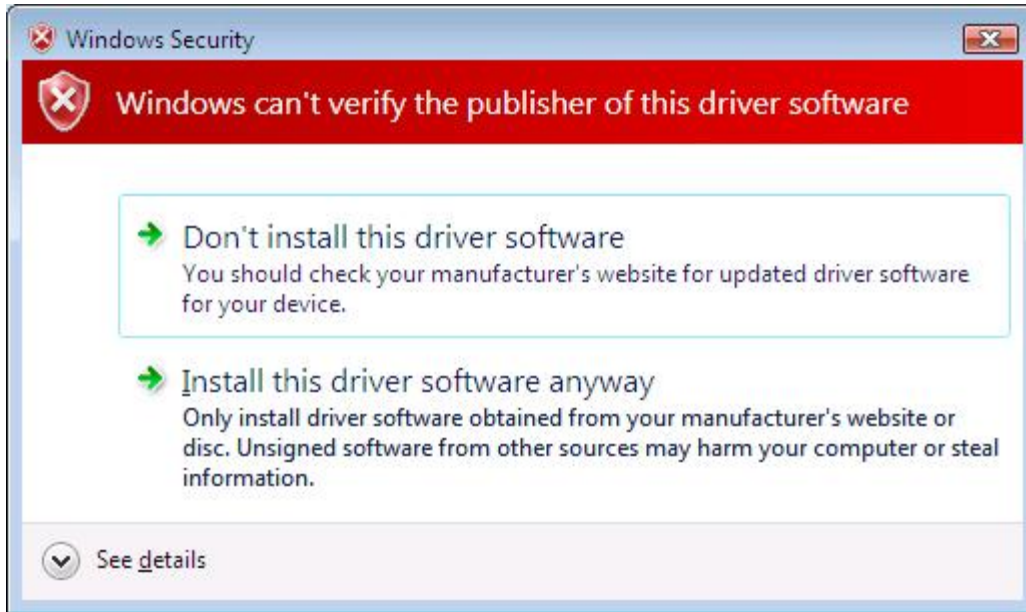
Select “Browse my computer for driver software”



You may need to copy the .inf file to a temporary directory:

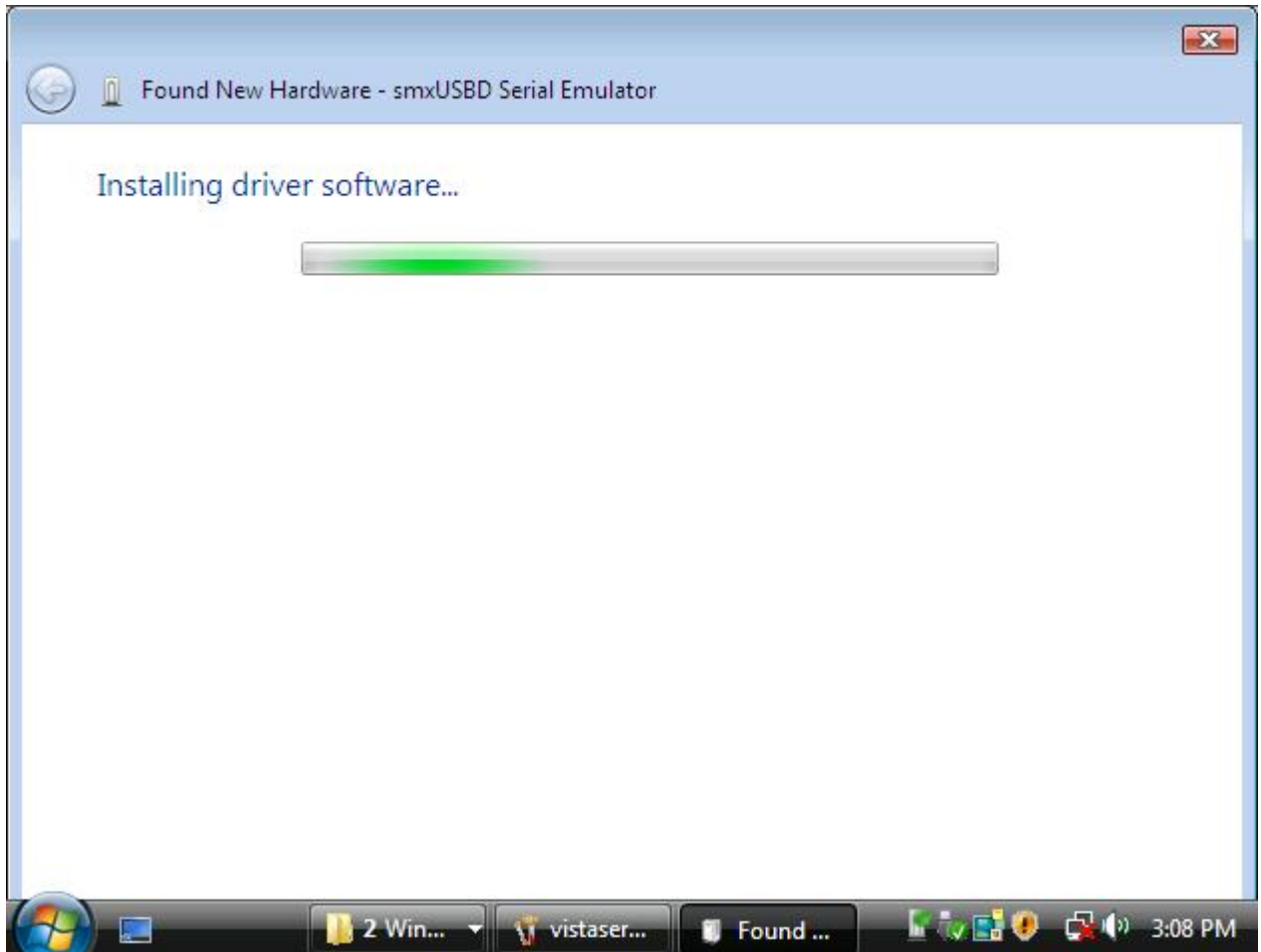


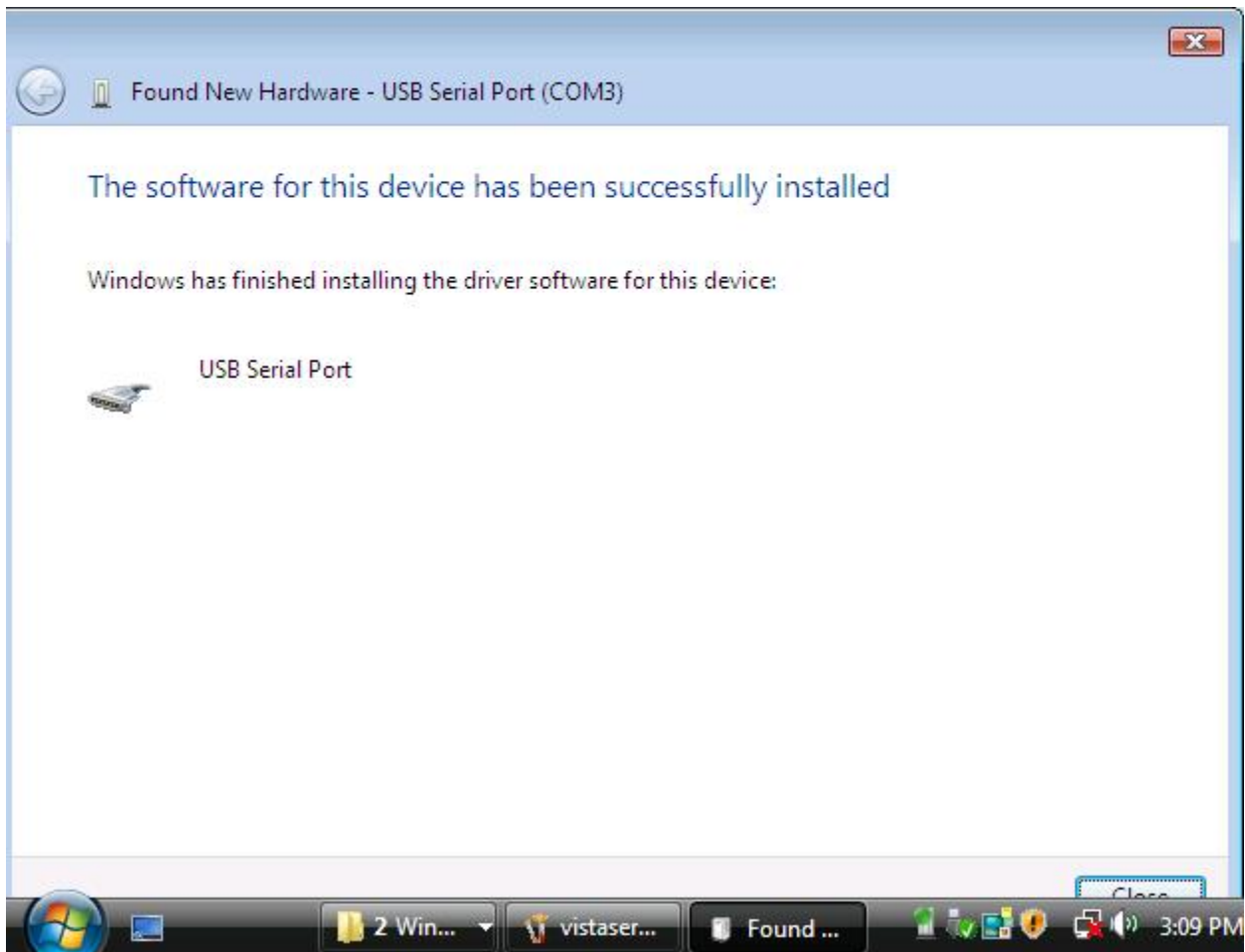
You need to select “Install this driver software anyway” when you get the following warning dialog (see Appendix H. Host OS Certification for more information):



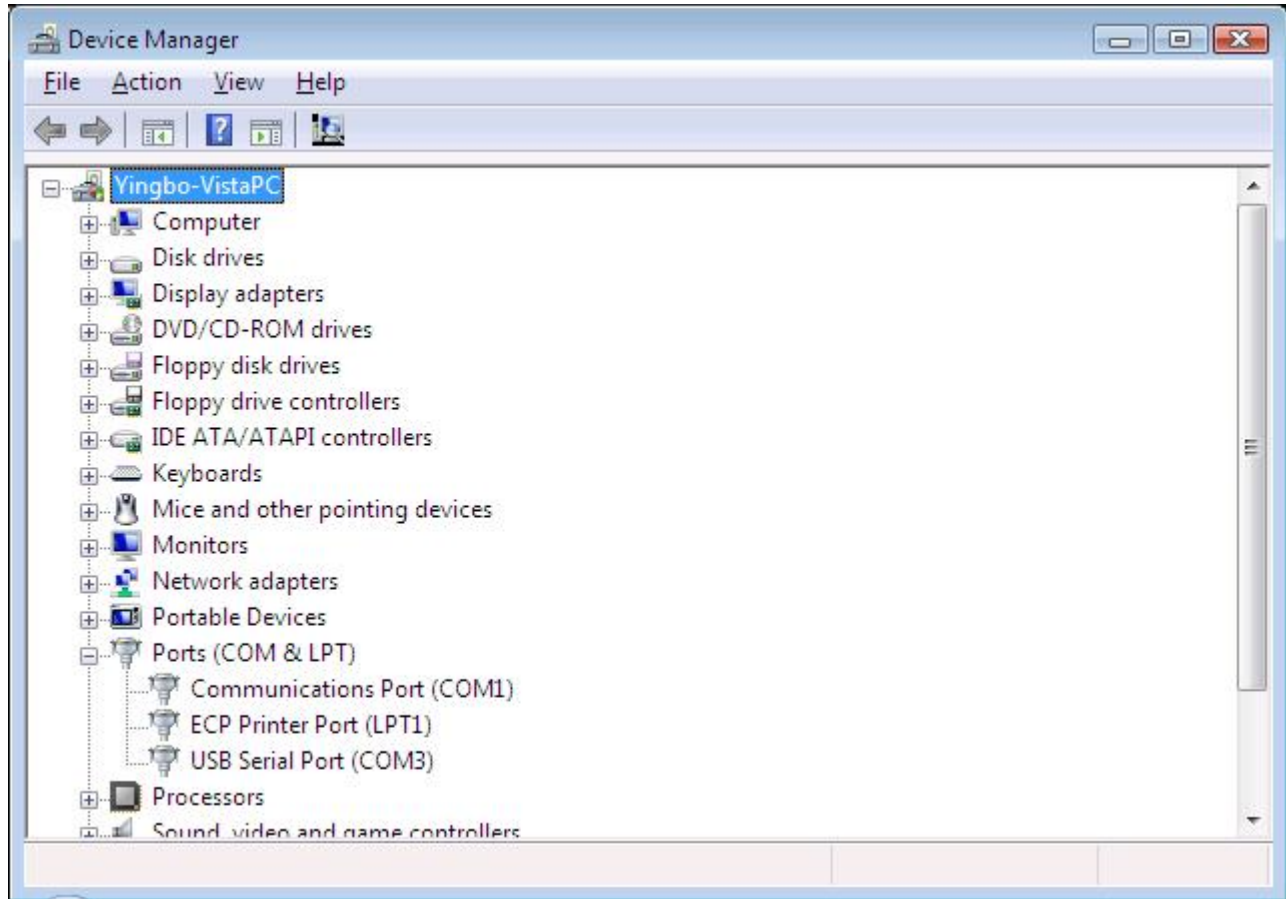
Windows will search for the driver `usbser.sys`. It may prompt you to insert your Windows installation CD if this file is not in your Driver Cache or Service Pack .cab file.



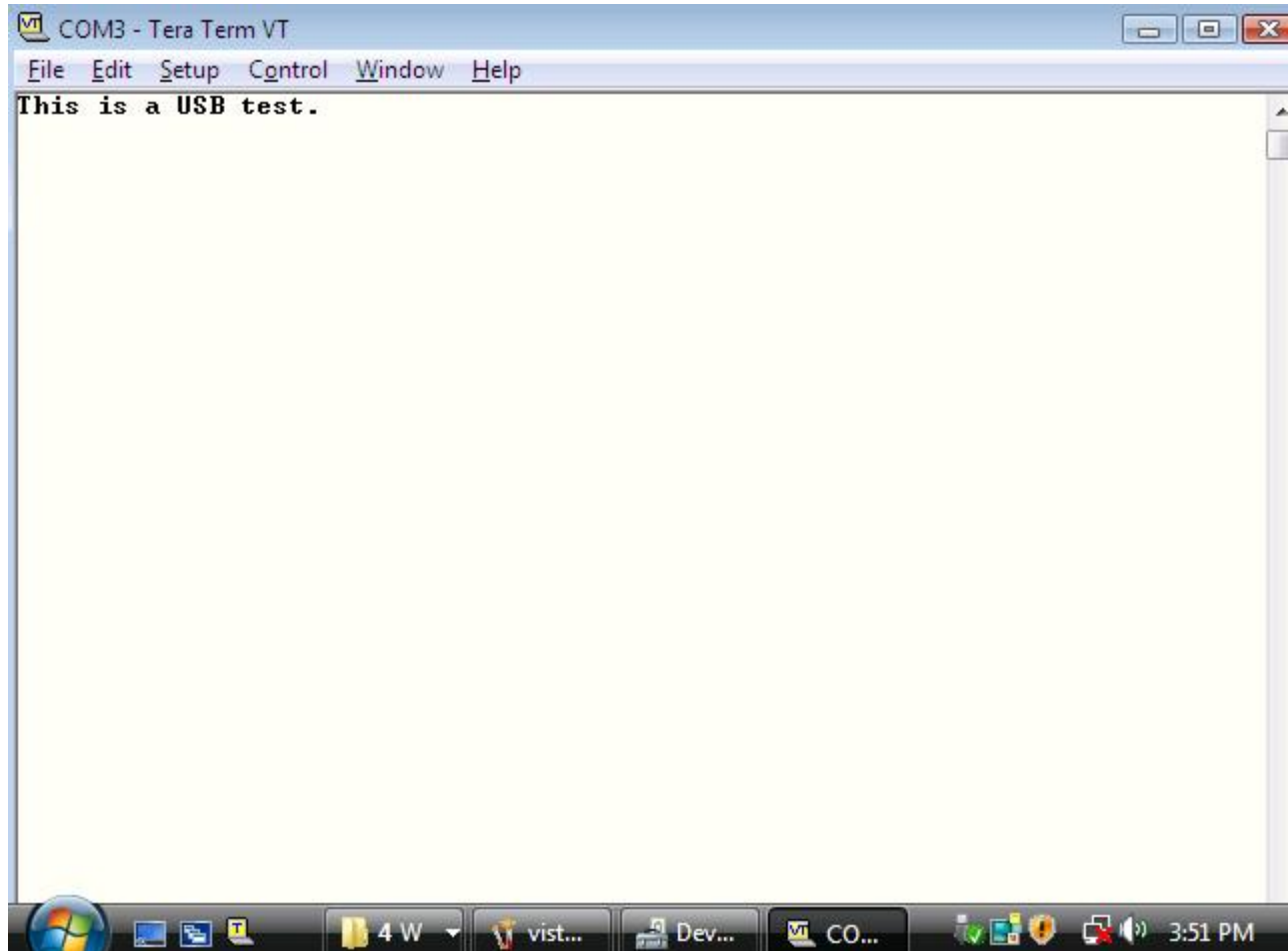




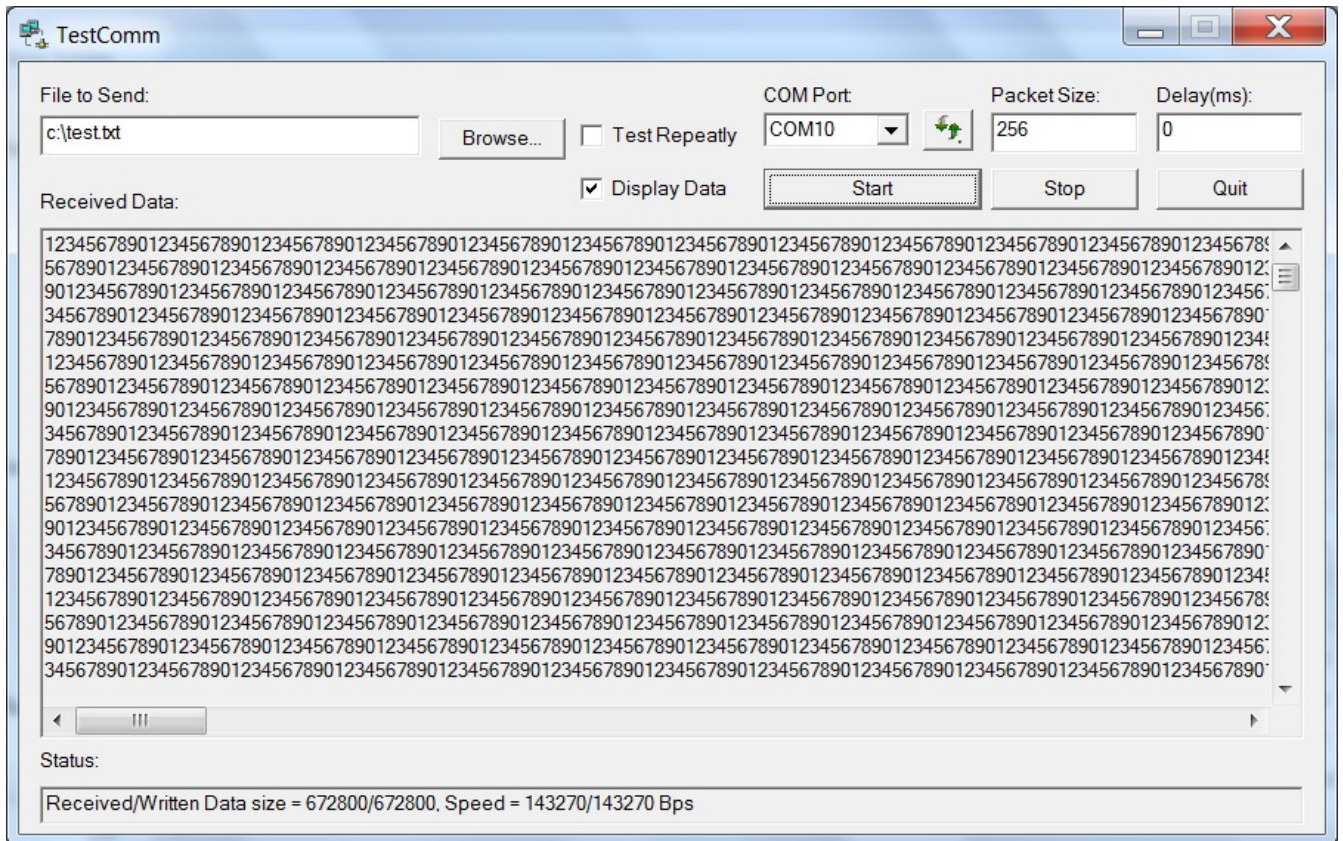
In the Device Manager, you will see a new Serial Port has been added:



You can use a terminal emulator program (e.g. HyperTerminal or Tera Term) to test if your serial port emulator works properly.

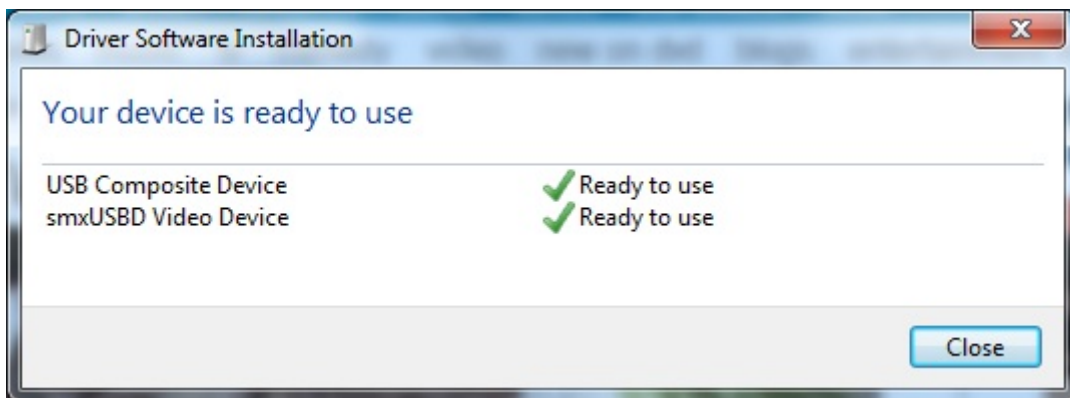


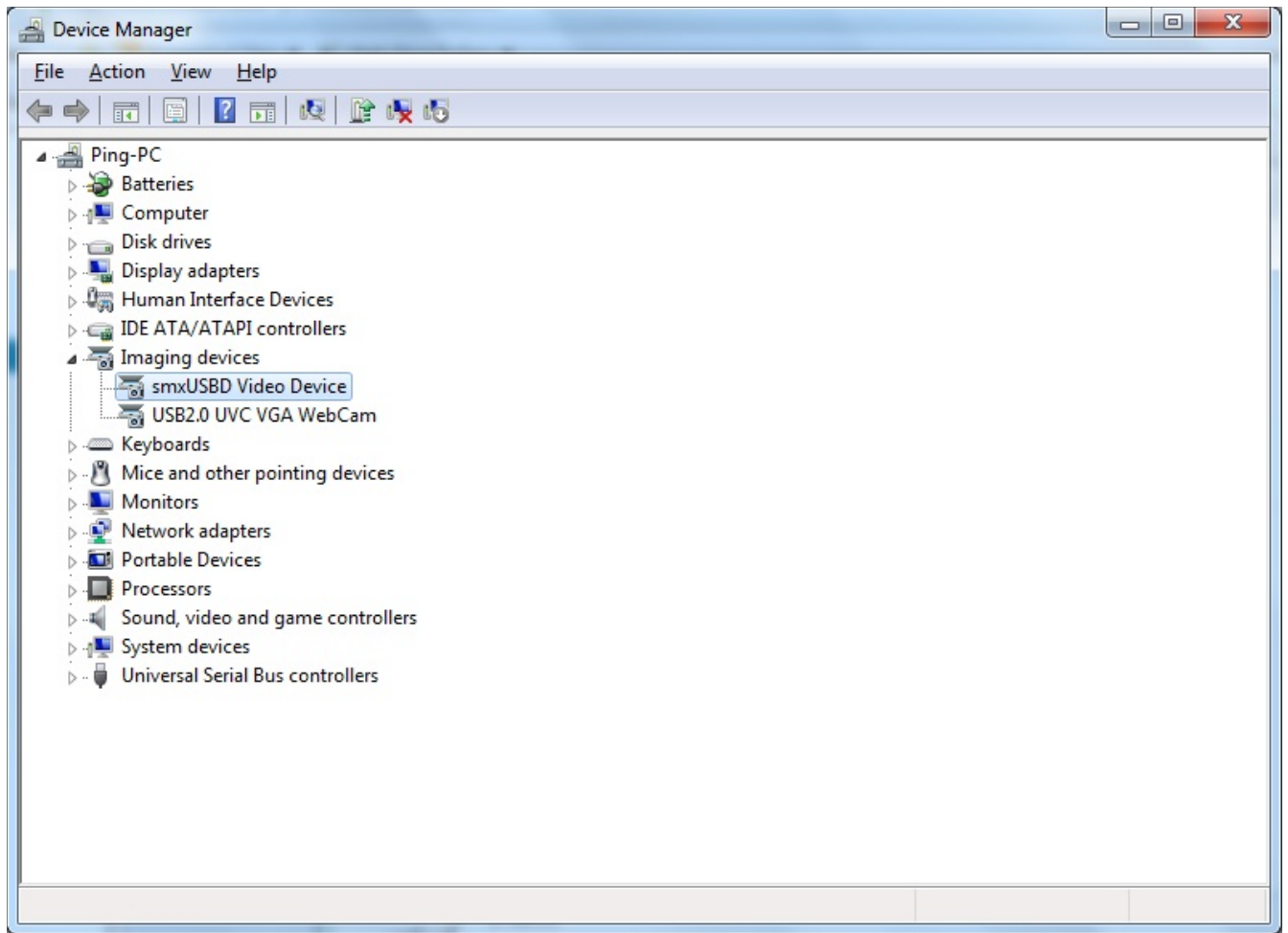
Or you can use our TestComm utility to do the performance and stress testing. It is in the BIN directory. TestComm initially selects the highest COM port because that is the most likely one to be for the smxUSB target just plugged in. If you started TestComm before plugging in the USB cable, click the refresh button to update the COM port list.



## E.8 Video

Windows Vista/7 will install the video device automatically when you plug in the device (your target board). After installation you can check the video device in Device Manager.





You can use the AMCap utility to capture the video.



## **Appendix F. Specification Reference**

smxUSBD is based on the following specifications. USB related documents are available at [www.usb.org](http://www.usb.org) or at [www.usb.org/developers/docs](http://www.usb.org/developers/docs).

### **F.1 USB Specifications**

*Universal Serial Bus Specification, Revision 1.1*

*Universal Serial Bus Specification, Revision 2.0*

### **F.2 Device Controller Specifications**

*ISP1161 Full-speed Universal Serial Bus single-chip host and device controller, Rev. 02*

*ISP1181 Full-speed Universal Serial Bus interface device, Rev. 04*

*ISP1362 Single-chip Universal Serial Bus On-The-Go controller, Rev. 04*

*ISP1582 Hi-Speed Universal Serial Bus peripheral controller, Rev. 03*

*ISP1761 Hi-Speed Universal Serial Bus On-The-Go controller, Rev. 02*

*AT91 ARM Thumb-based Microcontrollers AT91SAM7S256, 6175E-ATARM-04-Apr-06*

*AT91 ARM® Thumb®-based Microcontrollers AT91SAM7X256, 6120C-ATARM-16-Jan-06*

*ARM920T™-based Microcontroller AT91RM9200, 1768E-ATARM-30-Sep-05*

*LH7A404 Universal SoC User's Guide, Version 1.1*

*LPC3180 User Manual, Rev. 01, 1 June 2006*

*MC9328MX1 i.MX Integrated Portable System Processor Reference Manual, Rev. 5*

*MCF5329 Reference Manual, Rev. 1, 07/2006*

*MCF548x Reference Manual, Rev. 2.1, 10/2004*

*STR71x Microcontroller Reference Manual, Rev. 7*

*STR91xF ARM9®-based Microcontroller Family, Rev. 1*

### **F.3 Audio Devices Specifications**

*Universal Serial Bus Device Class Definition for Audio Devices, Revision 2.0*

### **F.4 Communication Devices Specifications**

*Universal Serial Bus Device Class Definition for Communication Devices, Revision 1.1*

### **F.5 Device Firmware Upgrade (DFU) Specifications**

*Universal Serial Bus Device Class Specification for Device Firmware Upgrade, Revision 1.1*

### **F.6 HID Specifications**

*Universal Serial Bus Device Class Definition for Human Interface Devices, Revision 1.11*

## **F.7 Mass Storage Specifications**

*Universal Serial Bus Mass Storage Class Specification Overview, Revision 1.2*

*Universal Serial Bus Mass Storage Class Bulk-Only Transport, Revision 1.0*

*SCSI Primary Commands - 2 (SPC-2), Revision 20*

*SCSI Block Commands - 2 (SBC-2), Revision 14*

## **F.8 Media Transfer Protocol (MTP) Specifications**

*Universal Serial Bus Device Still Image Capture Device Definition, Revision 1.0*

*Picture Transfer Protocol (PTP) for Digital Still Photography Devices, PIMA 15749:2000*

*USB Media Transfer Protocol Specification, Revision 1.1*

## **F.9 Remote NDIS Specifications**

*Microsoft Remote NDIS Specifications, Rev 1.1*

## **F.10 Video Device Specifications**

*Universal Serial Bus Device Class Definition for Video Devices, Revision 1.0a*

*Universal Serial Bus Device Class Definition for Video Devices, Revision 1.1*



## ***Appendix G. Testing***

USB testing is not only related to software (USB stack) but also to the device controller and other hardware, so there is no certificate for the USB stack itself. However, we did some testing based on some evaluation boards, as discussed below.

We tested smxUSBD with **USBCheck v5.1** on a Windows PC to verify that it passes the Chapter 9 compliance tests for full speed and high speed (if the controller supports high speed).

High Speed Test Mode is also supported.

On the following boards, smxUSBD also passed the **USBCV v1.3** Chapter 9 tests, HID tests, and MSC tests.

- Atmel AT91 (AT91SAM9260-EK)
- Atmel AT91HS (AT91SAM9M10G45-EK)
- Freescale MCF5329 and MCF5251 EVBs
- Freescale K70 (TWR-K70F120M)
- NXP ISP1181 compatible (ISP1362 PCI)
- NXP ISP1581 compatible (ISP1763 PCI)
- NXP LPCxxxx (LPC1788EA)
- STMicro STR7/9 (STR912KS)
- STMicro STM32F20x/40x full speed port (STM3240G-EVAL)

Freescale M5485EVB passed USBCV version 1.3 chapter 9 tests but failed the MSC tests because the MCF5485 device controller has some problem when sending 1-byte descriptors. MSC tests will retrieve 1 byte descriptors for string descriptor length.

Sharp LH7A40x failed USBCV test for suspend/resume because the device controller driver needs to know the USB cable is removed. Otherwise it cannot handle the suspend event properly.

## Appendix H. Host OS Certification

### H.1 Windows Logo Program / Windows Hardware Certification Program

Some device types such as mass storage and audio are standard device types with a built-in Windows driver, and no special steps are needed to use them. For other devices, such as serial and custom devices, it is necessary to go through an installation procedure the first time the device is plugged into the Windows PC. You point to the location of the driver .inf and .sys files, and then Windows installs the driver. However, it displays a warning dialog that says:



Many devices ship this way, but if you want to improve your customers' experience and avoid this warning, it is necessary to go through the steps of the Windows Logo Program (Windows XP) or Windows Hardware Certification Program (Windows 7 and 8). For full details, search for these names in your browser. Here is a summary of steps:

1. Get a VeriSign ID and Winqual account to identify your company.
2. Download Window Logo Kit and run the automatic test on your hardware.
3. After the test passes, submit the results to Microsoft and get the signature of the driver package.
4. Distribute the driver package with your product.

Notes:

1. It is necessary to do testing and pay a fee for each version of Windows you want to support.
2. It is necessary to do these steps even for the serial drivers provided in Windows.

3. It is necessary to do this for composite devices, even if they use standard drivers, e.g. mass storage + serial.

As part of the process, you will modify the .inf file to have your company and device name, so that will be what appears when your users plug in the device and Windows installs the driver. If we were to do this, it would be Micro Digital Inc and smxUSB Serial Driver (or similar) that would appear, which is not likely what you want for your product.

**Important:** It can be difficult to pass the testing, and the test program provided by Microsoft is difficult and time consuming to use. It provides inadequate diagnostics to determine what is wrong. We recommend that you start early on this process. Select a company to do the testing and try testing your device mid-way through development, not at the end. We will charge hourly for time spent assisting you with this process.

## ***Appendix I. Glossary***

API	Application Interface
BSP	Board Support Package
CDC	Communications Device Class
EOI	End Of Interrupt
HID	Human Interface Device
IAD	Interface Association Descriptor
IHV	Independent Hardware Vendor
.inf	Setup Information file, in plain text format
ISOC	Isochronous
ISR	Interrupt Service Routine
LSR	Link Service Routine
OS	Operating System
RTOS	Real Time Operating System
USB	Universal Serial Bus
USBD	USB Driver
USBDI	USB Driver Interface
USB-IF	USB Implementers Forum, Inc., a nonprofit corporation formed to facilitate the development of USB compliant products.